1999

Training Methods for Shunting Inhibitory Artificial Neural Networks

Son Lam Phung
Edith Cowan University

Training Methods For Shunting Inhibitory Artificial Neural Networks

A Thesis Submitted in Partial Fulfilment of the Requirements for the Award of Bachelor of Engineering (Computer Systems) with First Class Honours

Son Lam Phung

Faculty of Communications, Heath and Science
Edith Cowan University
Western Australia

Date of submission: 9 November 1999

USE OF THESIS

The Use of Thesis statement is not included in this version of the thesis.

Abstract

This project investigates a new class of high-order neural networks called *shunting inhibitory artificial neural networks* (SIANN's) and their training methods. SIANN's are biologically inspired neural networks whose dynamics are governed by a set of coupled nonlinear differential equations. The interactions among neurons are mediated via a nonlinear mechanism called shunting inhibition, which allows the neurons to operate as adaptive nonlinear filters.

The project's main objective is to devise training methods, based on error backpropagation type of algorithms, which would allow SIANN's to be trained to perform feature extraction for classification and nonlinear regression tasks. The training algorithms developed will simplify the task of designing complex, powerful neural networks for applications in pattern recognition, image processing, signal processing, machine vision and control.

The five training methods adapted in this project for SIANN's are error-backpropagation based on gradient descent (GD), gradient descent with variable learning rate (GDV), gradient descent with momentum (GDM), gradient descent with direct solution step (GDD) and APOLEX algorithm. SIANN's and these training methods are implemented in MATLAB. Testing on several benchmarks including the parity problems, classification of 2-D patterns, and function approximation shows that SIANN's trained using these methods yield comparable or better performance with multilayer perceptrons (MLP's).

DECLARATION

I certify that this thesis does not, to the best of my knowledge and belief:

(i) incorporate without acknowledgement any material previously submitted for a degree or diploma in any institution of higher education;

(ii) contain any material previously published or written by another person except where due reference is made in the text; or

(iii) contain any defamatory material.

Signature [redacted]

Date 8 Mar 2000

Acknowledgements

I would like to express my deepest gratitude to my supervisor Associate Professor Abdesselam Bouzerdoum for his tireless support and guidance through out the course of this project. I thank him for providing me with abundant materials on artificial neural networks in general, and several articles on shunting inhibitory neural networks in particular. He answered several of my questions on various aspects of the project and spent a considerable amount of time and effort reviewing my reports.

I am also thankful to Dr. Ganesh Kothapalli for his interest in the project and several helpful comments on the early drafts of this thesis.

Finally, I would like to take this opportunity to express my gratitude to AusAID for providing me with a scholarship, which allowed me to pursuit an undergraduate degree at Edith Cowan University.

Table of Contents

List of Figures

List of Tables

Abbreviations

ANN	Artificial Neural Networks
APOLEX	Algorithm for Pattern Extraction
GD	Standard error-backpropagation based on Gradient Descent
GDV	Gradient Descent with Variable learning rate training method
GDM	Gradient Descent with Momentum training method
GDD	Gradient Descent with Direct solution step training method
CFS	Contrast Sensitivity Function
MLP	Multilayer Perceptron
SIANN	Shunting Inhibitory Artificial Neural Network
SICNN	Shunting Inhibitory Cellular Neural Network

Notations

E	Network error function
N	Number of network layers
P	Network input
Y	Network output
T	Network target
s_k	Number of neurons in layer k
$f_k(.)$	Activation function of layer k
${}^k z_i$	Output of neuron i in layer k
${}^k p_{ij}$	Input to neuron i in layer k from neuron j in the previous layer
${}^k w_{ij}$	Connection weight to neuron i in layer k from neuron j in layer k-1
${}^k a_i$	Bias for neuron i in layer k
${}^k a_i$	Decay rate for neuron i in layer k
${}^k e_i$	Error sensitivity for neuron i in layer k

Chapter 1 General Introduction

1.1 Overview

With high speed and powerful processors readily available today, few are in doubt of the computational capability of modern computers. Yet one must be wondering why computers still have immense difficulty in carrying out tasks we humans seem to do effortlessly such as understanding speech and hand-written text, recognising objects, etc. Automating these tasks is challenging because decision rules are hidden in exceedingly complex, noisy and in some cases incomplete data. The search is on in the field of artificial neural networks for systems (modelled after biological processes) that possess the capabilities to reason, generalise and learn from experiences.

This project investigates a new class of high-order neural networks called *shunting inhibitory artificial neural networks* (SIANN's) and their training methods. SIANN's are biologically inspired networks whose dynamics are governed by a set of coupled nonlinear differential equations. The interactions among neurons are mediated via a nonlinear mechanism called shunting inhibition, which allows the neurons to operate as adaptive nonlinear filters. The project's main objective is to devise training methods, based on error backpropagation type of algorithms, which would allow shunting inhibitory neural networks to be trained to perform feature extraction for classification and nonlinear regression tasks. The training algorithms developed will simplify the task of designing complex, powerful neural networks for applications in pattern recognition, image processing, signal processing, machine vision and control.

1.2 Background and Motivation

Over the past decade, artificial neural networks have emerged as an effective information processing technique and a powerful computational tool. They have become the preferred method for many applications in which data relationships, decision processes and predictions have to be learned or modelled from examples. Nowadays, neural networks

find applications in telecommunications, defence, multimedia, banking, medicine, forensics, robotics, and remote sensing. They are used for credit card fraud detection, automatic check reading, signature verification, automatic target recognition, optical character recognition, financial forecasting, weather prediction, etc.

The most widely used neural network architecture is the feed-forward multilayer perceptron (MLP). MLP networks have been applied successfully to solve some difficult and diverse problems ranging from time series prediction to pattern classification and object recognition. However, before an MLP network can be successfully applied to a particular problem, many questions must be answered:

- What is the most suitable architecture? How many layers and how many neurons in each layer are required for a given task? How should the network be connected, fully or partially? Which activation function should be used?
- How can a network be programmed? Can it learn in real-time while functioning, or must it be trained then tested? How many examples are needed for good performance? How many times should the examples be presented?
- What are the capabilities of the networks? How many examples can it learn? How fast can it learn? How well can it generalise from known to unknown patterns? What classes of input-to-output functions can it represent?

Part of the weaknesses of the MLP is due to the simplicity of its neuron. Even though the neuron of a MLP is equipped with a nonlinear activation function, the decision surface it represents is a line, which is inadequate in most practical situations. To overcome this problem, practitioners use many neurons, arranged into layers, to approximate complex nonlinear surfaces. This gives rise to two problems. One is that the number of required neurons and the number of layers are not known beforehand. The other problem is that as more neurons are added, increase the number of parameters to be determined and the amount of time and data required to train the network. Recurrent networks have been considered as an alternative to feed-forward networks, but they are often avoided because of their inordinate learning time and the complexity of the learning algorithms.

Furthermore, their learning phase is subject to instabilities and/or bifurcation (Doya, 1992). Attempts have been made to introduce high-order neurons in the form of sigma-pi neurons (or polynomial neurons) into the feedforward architecture, but these attempts have not been very successful. There are two main reasons for this: firstly only polynomial non-linearity can be adequately represented; secondly, the number of connections grows exponentially with the order of the network. In addition, the user has to decide on the type of non-linearity (quadratic, cubic, or higher) to be used, which amounts to guessing what the solution is.

It is well known that shunting lateral inhibition plays a very important role in vision. It enhances edges and contrast, causes adaptation of the organisation of the spatial receptive field and of the contrast sensitivity function (CSF), enhances visual selectivity for small objects and edges, and mediates directional selectivity (Beare & Bouzerdoum, 1999; Bouzerdoum and Pinter, 1989). In 1993, a class of cellular neural networks based on shunting lateral inhibition was proposed (Bouzerdoum & Pinter, 1993). These networks have shown great promises as information processors in vision, and image processing tasks; they function as nonlinear adaptive filters, can serve as front-end pre-processors for pattern recognition and visual decision and control tasks. They have been used to model some early visual processing function such as edge detection, image enhancement, evolution of receptive field profiles, and motion detection (Bouzerdoum, 1993) but they have not yet been used for classification and function approximation tasks. This is due to two main reasons:

- The first reason is lack of proper training algorithms. So far the design of these networks has been relying on the expert knowledge of the user, who would choose the connection weights of the networks based on the task at hand; this limits the scope of applications of these networks. Moreover, it is not possible to solve complex pattern recognition tasks by handwiring the network connections.

- The second reason is that these networks are dynamic systems, and hence to find the output for an input pattern, we must solve a system of nonlinear differential

equations. Training such networks is impractical since the input pattern have to be presented tens of thousands of times and each time the output must be found and the connection weights adjusted; therefore training becomes tedious, cumbersome, and very time consuming.

Shunting Inhibitory Artificial Neural Networks have a shunting neuron as the basic building block and inherit the layered architecture of feed-forward multilayer perceptron. Due to their inherent non-linearity, shunting networks can represent complex nonlinear decision surfaces more efficiently than multilayer perceptron. Each neuron in a MLP can only represent a linear surface, whereas a neuron in a shunting inhibitory network can represent linear as well as nonlinear surfaces. Therefore, developing good training algorithms for these networks should result in much more powerful classifiers and function approximators. Furthermore, shunting inhibitory neural networks can serve as feature extractors. By combining a shunting inhibitory layer with a perceptron layer, it is possible to design systems that perform feature extraction and classification simultaneously.

1.3 Project Objectives

The principal aim of the project is to derive training methods for a new class of artificial neural network called *Shunting Inhibitory Artificial Neural Network* (SIANN). The project is divided into the following main tasks:

Task 1 The first task is to study the representation power of shunting inhibitory artificial neural networks. In particular we'll study the influence of the activation function (linear, sigmoidal, etc.) on the types of surfaces that can be represented.

Task 2 Derive backpropagation-type algorithms to train feed-forward SIANN's and optimise them for shunting networks. There are many backpropagation algorithms for MLP including gradient descent, gradient descent with momentum, gradient descent with variable learning rate, etc.

We'll investigate the suitability and efficiency of all these algorithms to train SIANN's.

Task 3 Implement SIANN's and the training algorithms using a software package such as MATLAB.

Task 4 Test the developed algorithms on some benchmark classification and regression problems, and compare their performance to that of MLP networks. For this we'll use benchmark problems available in the public domain as well as synthetic data that will be created to test specifically certain hypotheses.

1.4 Thesis Outline

The thesis consists of seven chapters. In this introductory chapter, we have described the motivation and background for this project. The project's aim and major tasks were also outlined. Chapter 2 is a general introduction to artificial neural networks and their applications. It reviews one of the most common neural network architectures known as the Multilayer Perceptron. It was the publications of backpropagation training algorithm for MLP early in the eighties that renewed interest in the field of artificial neural networks and have made possible an increasing number of ANN applications. Chapter 3 proposes a new class of artificial neural networks called *Shunting Inhibitory Artificial Neural Networks*. The chapter investigates the representation power of shunting inhibitory neurons and the potentials of SIANN's in solving classification problems. In chapter 4, which is the main focus of this project, several supervised training methods for SIANN's are derived. Chapter 5 documents MATLAB implementation of SIANN's and the training methods. Chapter 6 puts SIANN's and the training methods developed in this project to test against several benchmarks including the XOR and parity problems, pattern classification and function approximation. The last chapter summarises what has been achieved in this project and gives concluding remarks. Further research directions are also proposed in this chapter.

Chapter 2 Artificial Neural Networks

Before we proceed to investigate Shunting Inhibitory Artificial Neural Networks, it is essential to grasp an understanding of what artificial neural networks are. This chapter serves as an introduction to artificial neural networks and their applications. It also describes one of the most popular neural network architectures known as Multilayer Perceptron (MLP) and its error-backpropagation training algorithm. Understanding this algorithm is the foundation for chapter 4, in which we'll derive training algorithms for the new class of neural networks.

2.1 Biological Inspiration

It is known that the processes in human neural network are electrochemical in nature. Nervous signals propagate at rates in the order of millisecond, which are much slower than conventional digital serial computers. Despite this wide gap in unit speed, state-of-the-art computer systems such as vision systems still fall far short of the performance exhibited by biological systems in their processing ability. As an example, we are able to recognise hand-written characters in a robust manner, despite distortions, omissions, and major variations. The same capabilities can be observed in the context of speech recognition. Humans also have the ability to retrieve information, when only part of the pattern is presented. For example, one can recognise a friend in a crowd at a distance even when most of the image is occluded.

The development of artificial neural networks (ANN's) began some 50 years ago. It was motivated by the need to understand the principles on which the human brain works, and the desire to produce artificial systems capable of carrying out sophisticated or perhaps "intelligent" tasks that the human brain routinely performs. Artificial Neural Networks are simplified models of the central nervous system. They are networks of a large number of highly interconnected processing elements that possess the ability to respond to input stimuli and to learn to adapt to the environment. It is believed by many researchers in the

field that neural network models offer the most promising unified approach to building truly intelligent computer systems. ANN's share many characteristics with their biological counterpart including learning and generalisation, robust performance and parallel processing (Patterson, 1996).

Biological Neurons

Following, we describe biological neurons and artificial neurons in an attempt to draw the similarities between them. Discoveries by 1906 Nobel Prize winners Camillo Golgi and Ramon y Cajal proved that the human neural network is composed of a vast number of interconnected nerve cells or *neurons*. It is revealed under electron microscope that independent of their size and shape all neurons are constructed from the same basic parts: soma, dendrites, and axon (Figure 2.1):

Soma is the cell body that processes the incoming signals and when there is sufficient input, it fires.

Dendrites are rather short extensions of the cell body and are involved in reception of stimuli

Axon, by contrast is usually a single elongated extension. It is especially important in the transmission of nerve impulses from the soma to other cells.

Synapse is the junction between an axon and a dendrite

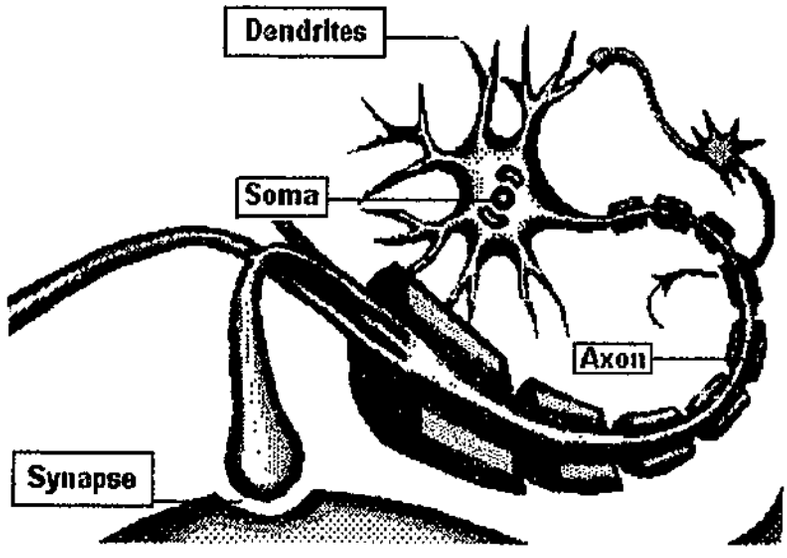


Figure 2.1: A biological neuron (*Source:* Microsoft Encarta, 1996)

It is estimated that the human central nervous system consists of near 100 billion neurons, each of which on average communicates with some thousand other neurons (Fausett, 1994). Nervous signals are transmitted either electrically or chemically. Electrical transmission prevails in the interior of a neuron, whereas chemical mechanisms operate between different neurons, i.e. at the synapses.

Synapses are either *excitatory* or *inhibitory*. An excitatory synapse tends to activate the receiving neuron, whereas an inhibitory synapse prevents impulses from arising in the receiving neuron. The later synapse is of particular interest to us because as will be seen later in chapter 3, (artificial) shunting inhibitory neurons consist of many inhibitory synapses.

Table 2.1: Gray matter statistics

Number of neurons	10^{11}
Number of synapses	10^{14}
Number of input neurons (afferent)	10%
Number of output neurons (efferent)	90%
Storage capacity	10^{13}-10^{15} bits
Utilisation factor	10%
Firing frequency	50-100 spikes/s
Signal propagation speed	5-25 m/s
Average brain weight	1.5 kg

Adapted from [Patterson, 1996]

Artificial Neurons

The basic processing element in an ANN is called an *artificial neuron*, or a *processing node*. Each node collects weighted inputs from many other nodes. It generates a single output, which is sent to other nodes as shown in figure 2.2. This is analogous to the manner in which the biological neuron receives signals from other neurons through its dendrites and sends its output to other neurons via its axons. The connection weights between nodes are adjustable and by adjusting the connection weights, different outputs can be produced. The ways in which nodes in an ANN are connected (e.g. partially or fully) or organised (e.g. into layers or as an array of cells) define the *network topology*. Many networks topologies exist including single-layer, multilayer, feedforward or recurrent.

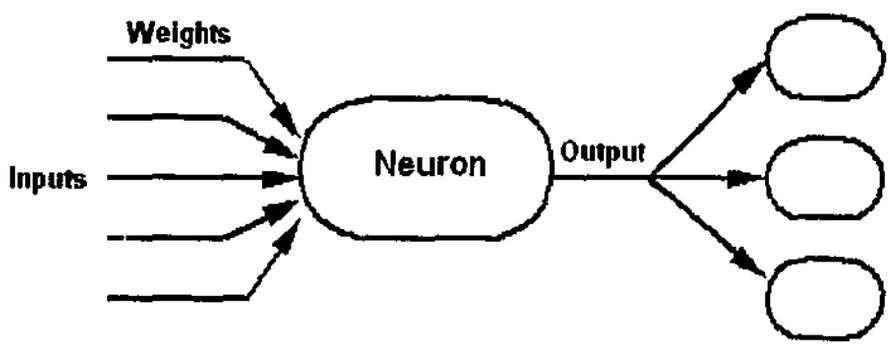


Figure 2.2: An artificial neuron

Fausett (1994) summaries several key similarities between an artificial neuron and a biological neuron:

- The processing node receives many signals
- Signals may be modified by a weight at the receiving synapse
- The processing node sums the weighted inputs
- The neuron transmits a single output
- The output may go to many other neurons through axon branches
- Information is stored both in synaptic weights and at neurons
- Information processing is local
- A synapse's strength maybe modified by experience
- Synapses may be excitatory or inhibitory

2.2 Applications of Artificial Neural Networks

Patterson (1996) outlines possible areas where ANN's can and have been applied:

General Mapping One of the most salient features of ANN's is their ability to learn arbitrary functions from a set of training examples. This is particularly useful in estimating functions that have no explicit mathematical model because neural networks often can find hidden relationship in the data. Some ANN's are used in data compression to map input vectors from high dimensional space to output vectors in lower one.

Forecasting ANN's have been shown to be successful predictive tools. After being trained a set of examples of past pattern/future outcome pairs, an ANN may be able to generalise and extrapolate to predict associative outcomes for new patterns.

Pattern recognition ANN's are good at learning to recognise complex patterns such as visual images of objects, hand-written characters and speech. The ability to associate between input and output patterns is also applied in creating content addressable memories or diagnosis problems in medicine, engineering, and manufacturing.

Optimisation Part of the ANN training process is to minimise the cost function by adaptively changing network parameters, which is essentially an optimisation problem. The range of optimisation problems that ANN's could be applied to solve is limited only by the ingenuity of the practitioner in defining the cost function and building an accurate ANN model of the system.

Next, we briefly describe some ANN applications in a medicine, finance and manufacturing. A more comprehensive accounts of ANN applications in diverse range of industries can be found in, for example, [Hubick, 1992] and [Fausett, 1994].

Medicine

Seismed Instruments marketed Seismic Quantitative Analysis - an ANN based software package that analyses the vibrational waves generated by a beating heart during exercise. The software is able to detect coronary heart disease by looking for specific patterns in a seismocardiograph (SCG). The software is embedded in an SCG-2000 seismocardiograph system to be sold in USA and twelve other countries.

In the UK, Errington and Graham (1993) combine a multilayer perceptron and a competitive neural network into a two-phase classifier for classification of chromosomes - a task that requires specially trained staff and is labour intensive. ANN's are also being used for cardiotocograms analysis in order to monitor the heart rate of foetal during labour, and to decide whether Caesarean sections are necessary. The ANN was trained with over 50,000 five-minute sections of cardiotocogram traces and its output is combined with an expert system, which contains 600 rules.

Researchers at Telectronics Pacing Systems (a company that controls 14 percent of the global heart-pacemaker market) and SEDAL at the University of Sydney are developing software and hardware for ANN implementation in heart-pacemakers. A pacemaker must identify a class of heartbeat, decide on whether the beat is normal, if not give an appropriate electrical treatment. Other medical applications of ANN's include breast cancer cell analysis, detection of abnormal practices in medical servicing.

Finance and Banking

The Mellon Bank in USA uses an ANN developed by Nestor Inc to detect credit-card fraud. The network detects changes in behaviour such as frequency of credit-card use, types of purchases, and size of payments. The training set is derived from credit-card transactions in six months. The Chase Manhattan Bank has begun to review loan applications with ANN software developed by Inductive Inference Inc under a multimillion-dollar contract. For each potential borrower, The ANN analyses six year of financial information. The result credit scoring is based on the financial strengths and weaknesses of a company. The software also performs three-year forecasts that indicate

the likelihood of a company being assigned a risk classification of 'good', 'criticised' or 'charged off' (in Chase terminology). Fujitsu Ltd claims that its Japan-Government bond-yield-rate forecasting ANN can achieve an accuracy of 75 percent (compared with 60 percent accuracy expected of human forecasting).

The research centre of Siemens has developed an ANN system using unsupervised learning, which detects in economic data the variables that are important in stockmarket analysis. Refenes and Zaidi (1993) use ANN for managing exchange rate trading strategies. The key idea is to predict, on the basis of past performance, which of the strategies is likely to perform best in the current context, and thus to minimise losses. The network is trained on daily currency exchange data from 1984 to 1986 and is tested "out-of-sample" from 1986 to February 1992. The network's annual returns outperform the best classical techniques by an average of 6 percentage points on a $1m position (including transaction costs).

Manufacturing

Kaiser Aluminium (Germany), one of the world's largest manufacturers of aluminium products uses an ANN in its surface-inspection system. A camera system observes the aluminium sheets as they are wound up at the rate of 2.5 meters per second. The digital output from the camera is fed into an ANN that looks for defects such as broken surfaces like scratches and non-broken surfaces like stains.

ANN's has been incorporated into an inspection system to detect internal flaws in electronics components at ATR Optical, Japan. The ANN is used to identify patterns of light reflected from a laser directed at samples of objects likes ICs. Companies such as Siemens GmBH, NEC Corp, BHP Australia are very active in applying ANN's to improve their manufacturing processes.

2.3 History of Artificial Neural Networks

Historical perspectives on the development of artificial neural networks are given in [Fausett, 1994], [Hagan, et al., 1996], and [Patterson, 1996]. The following summary shows how knowledge in the field has progressed.

In 1943, Warren McCulloch and Walter Pitts designed networks consisting of binary neurons with fixed threshold that are capable of representing logic functions. In 1949, based on the observation that that if two neurons were active simultaneously, then the strength of connection between them should be increased, Donald Hebb designed the first learning law for artificial neural networks. Late in the 50's, Frank Rosenblatt developed a class of artificial neural networks called *perceptrons* and the perceptron learning rule. His proof that the perceptron learning rule gives correct weights after finite iterations, if such weights exist, was a sensational research accomplishment at the time. In 1960, Bernard Widrow and Marcian Hoff developed the delta learning rule, which is a precursor of the backpropagation rule for multilayer nets. Their work – ADALINE (short for **ADA**aptive **L**inear **NE**uron) – was applied in one of the first commercial applications of neural networks to suppress noise over a telephone line.

In 1969, Marvin Minsky and Seymour Papert published *Perceptrons: Introduction to Computational Geometry*, in which they showed that perceptrons can classify only linearly separable sets. As a result, interest in ANN's diminished and for almost a decade neural network research had been largely suspended. However, some important works continued during the 1970s. In 1972, Teuvo Kohonen and James Anderson independently developed new neural networks that could act as memories. Stephen Grossberg was also very active during this period in the investigation of self-organising networks. Grossberg is perhaps best known for the highly successful adaptive resonance theory (ART) networks that he co-invented with Gail Carpenter.

The limitation of perceptrons was not addressed until early 1980s when a learning algorithm to adjust the weights in multilayer perceptrons was independently discovered and refined by several researchers David Parker, Yann LeCun, David Rumelhart. The

algorithm is known as error backpropagation since the error signals used for updating weights are propagated from the output layer backwards layer-by-layer. The algorithm in another form was discovered by Paul Werbos in his 1974 doctoral thesis: *Beyond Regression: New Tools For Prediction and Analysis in the Behavioural Sciences.*

The discovery of error backpropagation learning reinvigorated the field of neural networks. In the last ten years, thousands of papers on the topic have been written and an increasing number of ANN's applications have been reported. In 1982, John Hopfield - Nobel Prize winner in physics- developed Hopfield nets based on fixed weights and adaptive activations, which can serve as associative memory nets. Late 80's, Kunihiko Fukushima and his colleagues developed a specialised neural net called *neocognitron* for character recognition. Several researchers are involved in the development of non-deterministic neural nets in which weights or activations are changed on the basis of probability density function. Hardware implementation of neural networks is also an active area in artificial neural network research. Research in ANN's has gained considerate momentum in the early 90's.

2.4 Multilayer Perceptrons

Of any of the early neural nets, perceptrons perhaps had the most far-reaching impact. The perceptron learning rule is proved to converge to the correct weights, on the assumption that such weights exist (e.g. see Fausett, 1994). Correct weights are those that allow the net to produce the correct output value for each of the training input patterns. However, in 1969 Papert and Minsky pointed out a serious limitation of perceptrons. It was shown that perceptrons are capable of classifying only limited input patterns (i.e. linearly separable patterns). For example, there exist no correct weights for the XOR problem.

This limitation can be overcome by adding one or more hidden layers to perceptrons (thus the name multilayer perceptron). MLP with non-linear activation functions are capable of classifying more complex sets of input vectors. It is shown that a MLP with

three layers of neurons (input layer, hidden layer and output layer) can represent any continuous function to any degree of accuracy (i.e. any mapping between input vectors and target vectors). The most widely known training rule for a multilayer perceptron is error back-propagation algorithm based on gradient descent.

2.4.1 Network architecture

A multilayer perceptron (figure 2.3) consists of many layers, each of which has a number of neurons. There are two fix-sized layers called *input layer* and *output layer*, and between these two layers is one or more hidden layers. The input layer receives data from the environment and distributes them to the next inner layer. Each element of the input layer does no processing. The output layer passes the network's result back to the environment. Hidden layers are so called because they have no direct connection with the outside environment. All processing in the neural network is done in the hidden and output layers as neurons in each layer receive input from the previous layer, calculate the output and then send it to the next layer. Connections between nodes are weighted and it is through adjusting the weights that different network outputs can be produced. The number of net layers in literature is often considered as the number of adjustable weight layers (e.g. 3 for the net in figure 2.3).

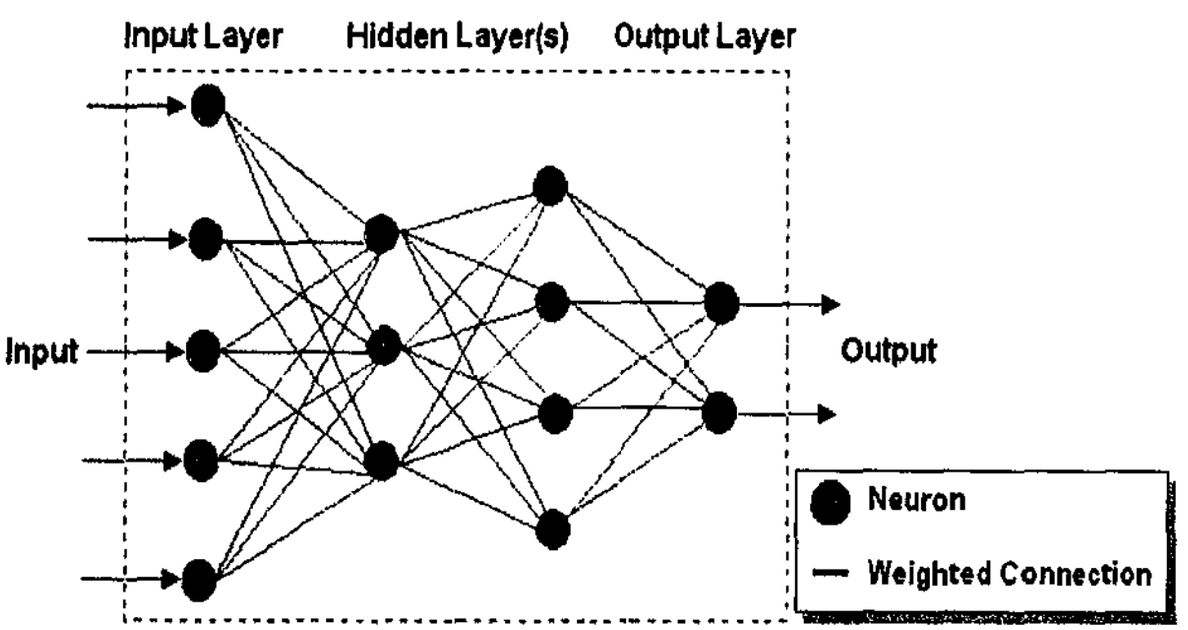


Figure 2.3: Multilayer Perceptron

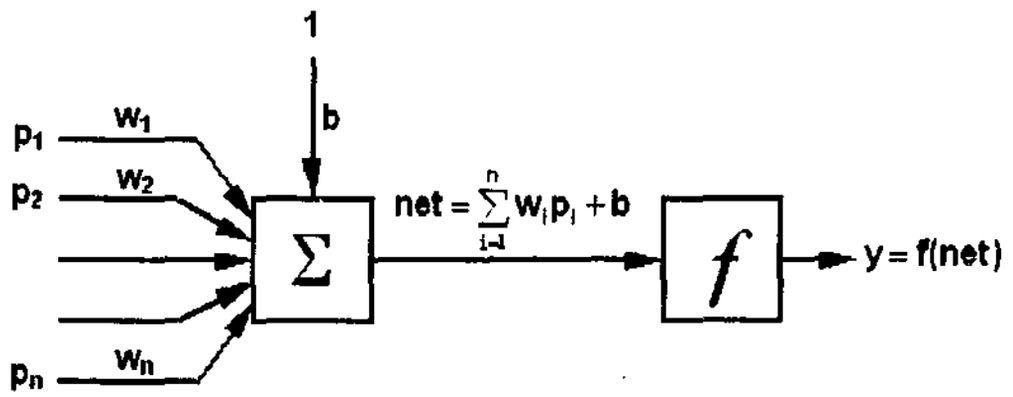


Figure 2.4: A neuron in MLP

Figure 2.4 above shows a closer look at a neuron. Let $p_1, p_2, \ldots, p_n$ be the inputs to a neuron and $w_1, w_2, \ldots, w_n$ be the weights of the input links. The neuron calculates the weighted sum of inputs and applies the activation function f to obtain the output:

$$f(net) = f(b + \sum_{i=1}^{n} w_i p_i)$$

The adjustable value b is called bias, its use is to shift the decision surface. The bias can be considered as weight to a constant input of 1.

2.4.2 Supervised and Unsupervised Learning

There are two types of learning for ANN's, both based on psychological observations: supervised learning and unsupervised learning. *Supervised learning* is like learning under watchful eyes of a teacher who points out where errors are made and where response is correct. This learning scheme applies to multilayer perceptrons because the user need to supply with each input vector **P**, a target vector **T**, which the net is expected to produce. Training typically is an iterative process as depicted in figure 2.5. The actual output of the net **Z** is repeatedly compared to its target **T** and adjustments to weights and biases are made until the difference between the actual output and the target is within an acceptable tolerance defined by the user. Note that for supervised learning the terms *training algorithm* and *learning algorithm* are used interchangeably in literature; both refer to the rule, by which network parameters are adjusted.

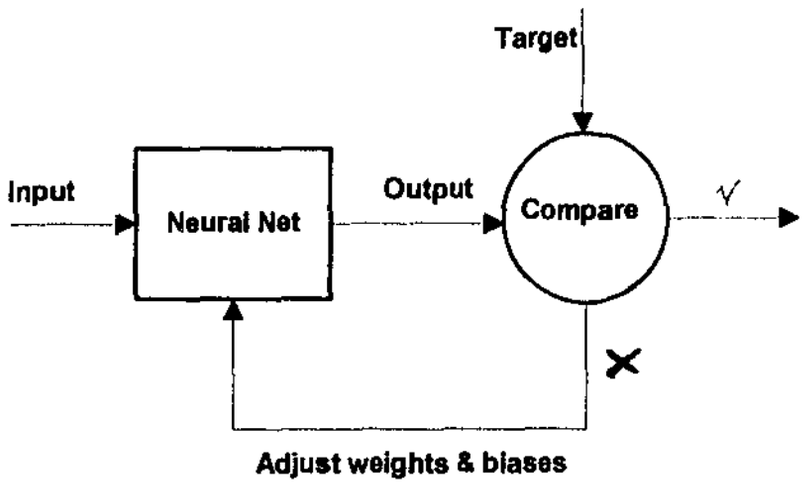


Figure 2.5: Supervised learning model

The other type of learning is *unsupervised learning*. As its name suggests, in unsupervised learning, there is no such thing as a "correct" response. Instead, the network learns to separate training patterns into groups or classes by looking for features that discriminate them. This form of learning is also useful because the network may be able to identify characteristics in the data that are unknown to the trainers.

2.4.3 Error Backpropagation Algorithm

Let us define the cost function as the function that describes the difference between the actual output and the expected output of the net. Many possible cost functions exist, of which the following sum squared error (*sse*) is one of the most commonly used:

$$E = .5 \sum_i \|\mathbf{Z}(i)-\mathbf{T}(i)\|^2 \tag{2.1}$$

where $\mathbf{Z}(i)$ and $\mathbf{T}(i)$ are the actual output vector and the target vector for input vector numbered i. The summation is over the entire training set. Note the factor inside the summation sign is

$$\|\mathbf{Z}(i)-\mathbf{T}(i)\|^2 = \sum_j (z_j(i) - t_j(i))^2$$

where subscript j denotes the jth element of a vector.

Basically, training starts with a random set of weights and biases. At each training iteration, we change the weights and biases by small steps in a direction chosen so as to

decrease the cost function with expectation that after a finite number of updates, the cost function will fall below an acceptable limit. Because the targets are fixed, the cost function E can be expressed as a multivariate function whose variables are the net weights and biases (here **V** is a column vector derived by putting together all weights and biases):

$$E = F(\mathbf{V}) = F(v_1, v_2, \ldots, v_m)$$

In error backpropagation algorithm, the update direction is chosen be the negative gradient of the cost function, which is the direction of steepest descent (for proof, see e.g. O'Neil, 1995, pp. 576-579). The gradient of F is

$$\nabla \mathbf{F} = (\frac{\partial F}{\partial v_1}, \frac{\partial F}{\partial v_2}, \ldots, \frac{\partial F}{\partial v_m})$$

A small learning rate α is used to calculate the amount of update for each parameter:

$$v_i(\text{new}) = v_i(\text{old}) - \alpha \frac{\partial F}{\partial v_i} \tag{2.2}$$

Our focus next is how to compute the gradient of F. To facilitate discussion, we need to introduce a few new notations:

${}^k w_{ij}$	Weight going *from* neuron j in layer k-1 *to* neuron i in layer k
${}^k b_i$	Bias for neuron i in layer k
${}^k z_i$	Output of neuron i in layer k
${}^k c_i$	Weighted sum input to neuron i in layer k
s_k	Number of neurons in layer k

As a rule, the superscript to the left denotes the layer number (i.e. k), if there are two subscripts, the first subscript specifies "destination" neuron, the second "source" neuron.

The output of layers can be described by the following recursive formula:

$${}^k z_i = f(\sum_{j=1}^{s_{k-1}} {}^k w_{ij} \cdot {}^{k-1} z_j + {}^k b_i) \tag{2.3}$$

That is neuron i in the k layer sums the weighted inputs to it from all neurons in the previous layer (i.e. layer k-1), and applies the activation function f to the weighted sum to compute its output. This formula applies also for the first layer (layer 1) if we consider

the input vector is just the output of layer 0: ${}^{0}z_i = p_i$. The net output is the output of the last layer (layer N).

Error sensitivities

Let ${}^{k}c_i$ be the weighted sum to neuron i in layer k. That is

$$ {}^{k}c_i = \sum_{j=1}^{s_{k-1}} {}^{k}w_{ij} \cdot {}^{k-1}z_j + {}^{k}b_i \tag{2.4} $$

Now define the *error sensitivity* for each neuron i in layer k as

$$ {}^{k}e_i = \frac{\partial E}{\partial {}^{k}c_i} \tag{2.5} $$

The error sensitivity measures the rate of change in the cost function as the weighted input to a neuron changes. Given all error sensitivities, it is pretty straightforward to compute the gradient of E by applying the chain rule of differentiation. Because the cost function can be considered as a function of ${}^{k}c_i$ of layer k, the partial derivative of E with respect to (w. r. t.) a weight ${}^{k}w_{ij}$ is just:

$$ \frac{\partial E}{\partial {}^{k}w_{ij}} = \frac{\partial E}{\partial {}^{k}c_i} \cdot \frac{\partial {}^{k}c_i}{\partial {}^{k}w_{ij}} = {}^{k}e_i \cdot \frac{\partial {}^{k}c_i}{\partial {}^{k}w_{ij}} $$

Using the definition (2.4) of ${}^{k}c_i$ gives

$$ \frac{\partial E}{\partial {}^{k}w_{ij}} = {}^{k}e_i \cdot {}^{k-1}z_j \tag{2.6a} $$

and similarly,

$$ \frac{\partial E}{\partial {}^{k}b_i} = {}^{k}e_i \tag{2.6b} $$

Computing error sensitivities by backpropagation

The problem now is to compute error sensitivities. It turns out that similar to the way we use the recursive formula (2.3) to compute the net output, all error sensitivities can be computed using another recursive formula. The only difference is that the former computes the net output in forward direction from layer 1 to last layer, whereas the later

computes error sensitivities backwards from the last layer to layer 1. Interestingly, the connection weights play a similar role in both recursive formulae, as will be shown next.

For the output layer,

$$ {}^{N}e_i = \frac{\partial E}{\partial {}^{N}c_i} = \frac{\partial E}{\partial z_i} \cdot \frac{\partial z_i}{\partial {}^{N}c_i} = (z_i - t_i).f({}^{N}c_i) \tag{2.7} $$

For an inner layer k, k = N-1, N-2, ... 1, consider E as a function of ${}^{k+1}c_j$ in layer k+1, and apply the chain rule:

$$ \begin{aligned} {}^{k}e_i = \frac{\partial E}{\partial {}^{k}c_i} &= \frac{\partial E({}^{k+1}c_1, {}^{k+1}c_2, \ldots, {}^{k+1}c_{s_{k+1}})}{\partial {}^{k}c_i} \\ &= \sum_{j=1}^{s_{k+1}} \frac{\partial E}{\partial {}^{k+1}c_j} \cdot \frac{\partial {}^{k+1}c_j}{\partial {}^{k}c_i} \\ &= \sum_{j=1}^{s_{k+1}} {}^{k+1}e_j \cdot \frac{\partial {}^{k+1}c_j}{\partial {}^{k}c_i} \\ &= \sum_{j=1}^{s_{k+1}} {}^{k+1}e_j \cdot \frac{\partial {}^{k+1}c_j}{\partial {}^{k}z_i} \cdot \frac{\partial {}^{k}z_i}{\partial {}^{k}c_i} \end{aligned} $$

Applying definition (2.3) for ${}^{k+1}c_j$, we obtain:

$$ {}^{k}e_i = \sum_{j=1}^{s_{k+1}} {}^{k+1}e_j . \, {}^{k+1}w_{ji} . f'({}^{k}c_i) $$

The common factor $f({}^{k}c_i)$ can be taken out of the summation sign:

$$ {}^{k}e_i = f({}^{k}c_i) . \sum_{j=1}^{s_{k+1}} {}^{k+1}e_j . \, {}^{k+1}w_{ji} \tag{2.8} $$

This recursive relation for computing error sensitivities is illustrated in figure 2.6. It is the way in which error sensitivities propagate from the output layer backward layer-by-layer that gives this algorithm the name *error backpropagation*. Figure 2.7 summarises the major steps in the training algorithm.

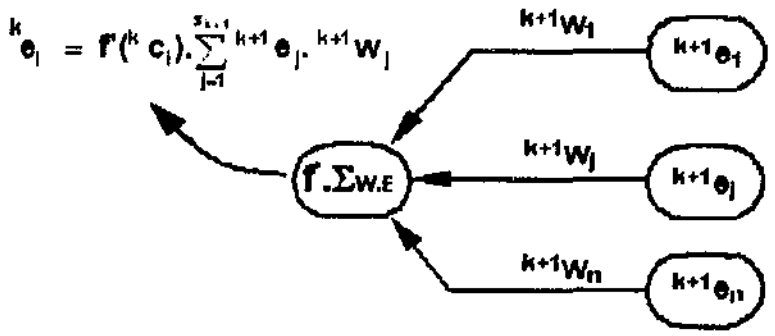


Figure 2.6: Backpropagation of error sensitivities

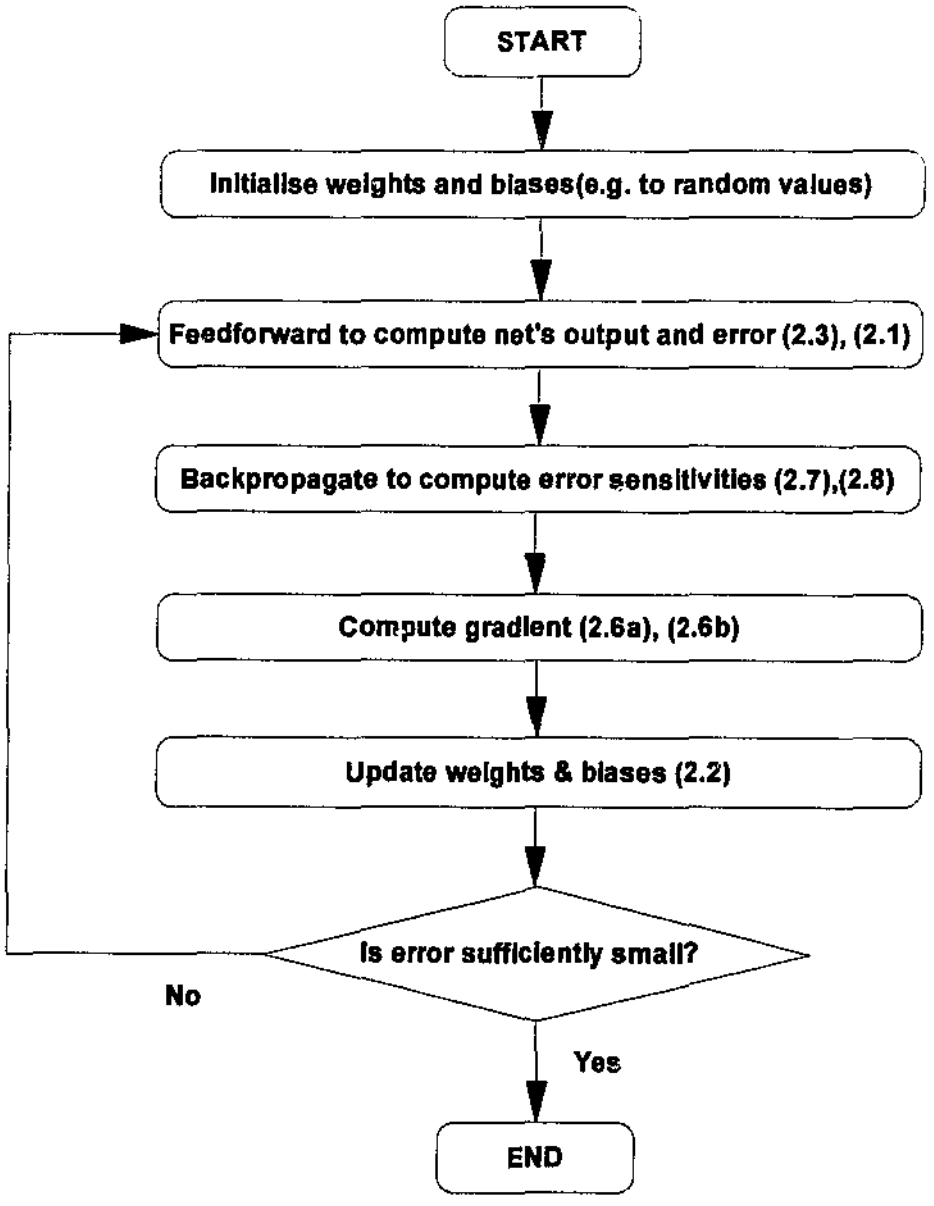


Figure 2.7: Error Backpropagation Algorithm for MLP

Using error backpropagation, the learning rate should be chosen sufficiently small for stable training. However, learning then may become unacceptably slow. Another serious problem is that unlike learning in single-layer perceptron, there is no guarantee that MLP training will converge after a finite number of iterations. One reason is that the network may get stuck in a local minimum. These limitations have prompted the study of many improved training algorithms. In Chapter 4, these algorithms will be discussed and adapted to train Shunting Inhibitory Artificial Neural Networks.

Chapter 3
Shunting Inhibitory Artificial Neural Networks

The aim of this chapter is to study in general the architecture and capabilities of shunting inhibitory artificial neural networks (SIANN's). After introducing the most important building block of SIANN's - the shunting inhibitory neuron – we'll describe the architecture of the new class of neural networks and how the output of SIANN is computed. Through an example of a two-neuron network, we investigate the representation power of shunting inhibitory neurons. The chapter concludes with a brief review of previous works on shunting inhibitory neural networks.

3.1 Shunting Inhibitory Neuron

This section describes the basic building block of SIANN: the shunting inhibitory neuron. Shunting inhibitory neurons, shown in figure 3.1 exert mutual inhibitory interactions of the shunting type.

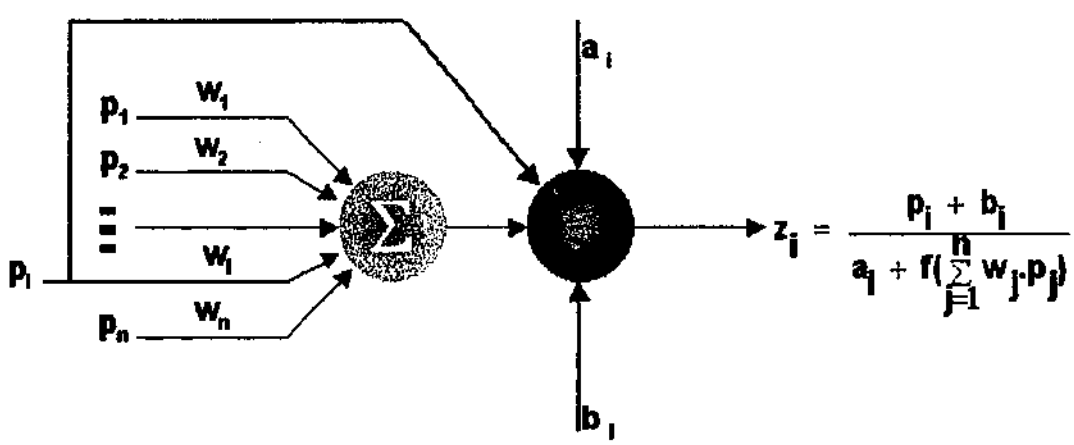


Figure 3.1: Shunting Inhibitory Neuron (steady state activation)

The activity of each feed-forward neuron is described by a nonlinear differential equation of the form:

$$\frac{dz_i}{dt} = p_i - a_i z_i - f(\sum_j w_{ij} p_j) z_i + b_i \tag{3.1}$$

where:

- z_i is activity (output) of neuron i,
- p_j is the external input,
- w_{ij} is connection weight from neuron j to neuron i,
- b_i is a bias constant,
- a_i is a constant representing the passive decay rate of neuron activity.
- f is the activation function of neuron i.

In the absence of inputs p_j and bias b_i, the neuron activity simply decreases asymptotically to zero (due to $dz_i/dt = -a_i z_i$). For a constant input pattern, the steady state solution of equation (3.1) is found by letting $dz_i/dt = 0$,

$$\frac{dz_i}{dt} = p_i - a_i z_i - f(\sum_j w_{ij} p_j) z_i + b_i = 0$$

or

$$z_i.[a_i + f(\sum_j w_{ij} p_j)] z_i = p_i + b_i$$

$$z_i = \frac{p_i + b_i}{a_i + f(\sum_j w_{ij} p_j)} \tag{3.2}$$

Equation (3.2) is used throughout this project to compute the steady-state output of shunting inhibitory neurons.

3.2 SIANN Network Architecture

A feed-forward SIANN shares a similar architecture with Multilayer Perceptron (Figure 3.2). The network consists of many layers, each of which has a number of neurons. There are two layers called *input layer* and *output layer*, and between these two layers is one or more hidden layers. No processing is done in the input layer because it only acts as a receptor that receives inputs from the environment and broadcasts them to the next layer. The output layer passes the network's result to the environment. Hidden layers are so called because they have no direct connection with the outside environment.

All processing in the network is done in the hidden and output layers as neurons in each layer receive inputs from the previous layer, calculate their outputs and then forward them to the next layer. Connections between nodes are weighted and the weights are adjustable giving the network the capability to generate different outputs. In this project, the term *number of layers* means the number of adjustable weight layers (e.g. 3 for the network in figure 3.2). Input layer is layer 0, layer 1 is the first hidden layer and so on to the output layer.

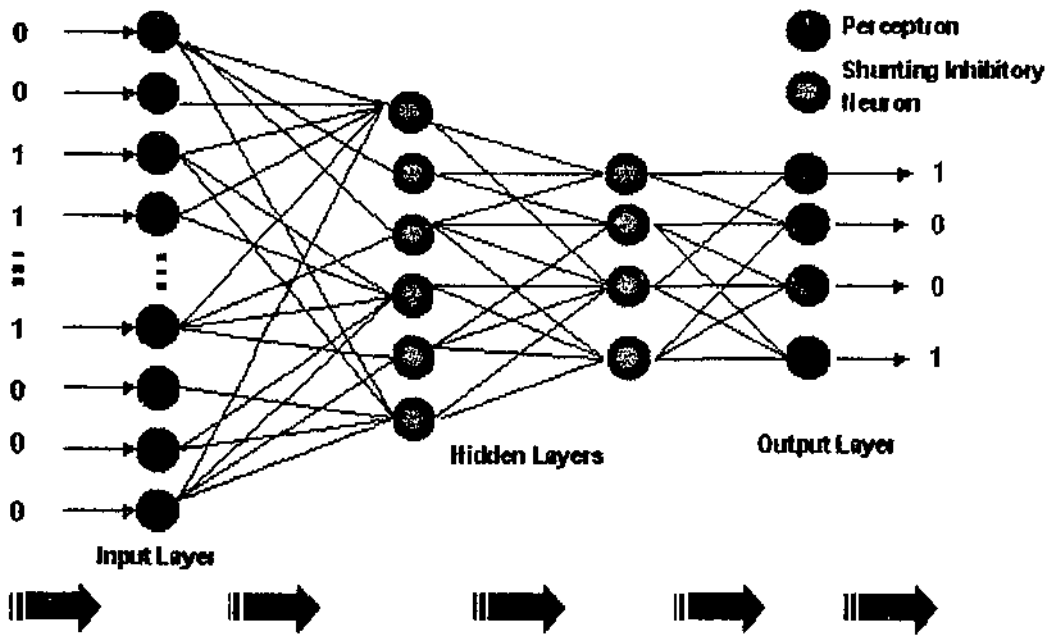


Figure 3.2: SIANN Network Architecture

The output layer is a perceptron layer, i.e. it consists of only perceptrons. A detailed description of perceptrons was given in the previous chapter (section 2.4). Basically, a perceptron calculates the weighted sum of its input and then applies an activation function to compute a single output. Its activity can be summarised by the following equation:

$$z_i = f(\sum_{j=1}^{n} w_{ij} \cdot p_{ij}) \tag{3.3}$$

All hidden layers of the neural network are shunting inhibitory layers and consist of only shunting inhibitory neurons. This is the key difference between SIANN and MLP. The role of shunting layers is to perform linear as well as complex nonlinear transformation on input data so that the results can be combined easily by perceptrons in the output layers to solve classification and function approximation problems. This role will be illustrated in the next section (3.3), but before doing that, let's describe quantitatively how the output of SIANN is computed.

3.2.1 Output of SIANN

Following is a short list of notations for the parameters in SIANN.

N	The number of weight layers
s_k	Number of neurons in layer k
${}^{k}w_{ij}$	Weight going *from* neuron j in (k-1) layer *to* neuron i in layer k
${}^{k}b_i$	Bias for neuron i in layer k
${}^{k}z_i$	Output of neuron i in layer k
${}^{k}c_i$	Weighted sum input to neuron i in layer k
$f_k(.)$	Activation function of layer k
p_i	Element i of the input pattern to the network

As a rule, the superscript to the left denotes the layer number (i.e. k), if there are two subscripts, the first subscript specifies *"destination"* neuron, the second *"source"* neuron. If we consider the input to the network as the *"output"* of layer 0 (i.e. $p_i = {}^{0}z_i$, then computing the final output can be expressed in a concise way as follows:

Calculating the output of SIANN

Compute output of shunting inhibitory layers

For k = 1 to N-1

Compute the output of layer k

For i = 1 to s_k,

Compute the output of neuron i

$$^{k}z_i = \frac{^{k-1}z_i + {}^{k}b_i}{^{k}a_i + f_k(\sum_{j=1}^{s_{k-1}} {}^{k}w_{ij} \cdot {}^{k-1}z_j)}$$

End For

Output of one layer becomes input to the next layer.

End For

Compute the output of the perceptron layer

For i = 1 to s_N

Compute the output of neuron i

$$y_i = {}^{N}z_i = f_N(\sum_{j=1}^{s_{N-1}} {}^{N}w_{ij} \cdot {}^{N-1}z_j + {}^{N}b_i)$$

End For

The network's final output is $^{N}\mathbf{z}$.

3.2.2 Activation functions

The above procedure can be used to compute the output of a SIANN for almost any activation function: continuous, non-continuous, differentiable or non-differentiable. However, for the network to be trainable using gradient descent method, there is a restriction that its activation functions must be differentiable. Table 3.1 lists several activation functions that are commonly used in SIANN's. Their plots are shown in figure 3.3.

Table 3.1: Common activation functions

Name	Description
exp	$f(x) = e^{x}$
expmn	$f(x) = e^{-x}$
hardlim	$f(x) = 1$ if $x \geq 0$, 0 otherwise
logsig	$f(x) = 1/(1 + e^{-x})$
purelin	$f(x) = x$
sin	$f(x) = \sin(x)$
tansig	$f(x) = (1- e^{-x}) / (1+e^{-x})$

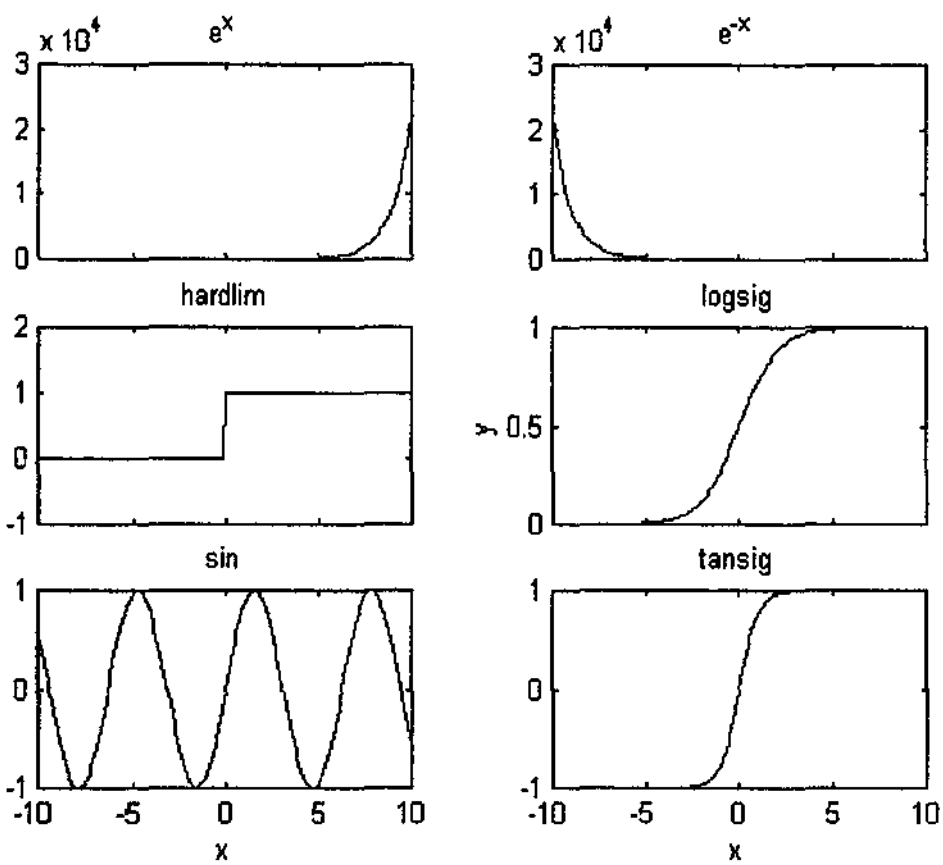


Figure 3.3: Plots of common activation functions

3.3 Decision boundaries of SIANN's

Perceptrons in MLP can only represent piecewise linear decision surfaces whereas shunting inhibitory neurons can represent linear as well as nonlinear surfaces. To illustrate this, we consider a simple SIANN with two neurons in the hidden layer. Their activations are given by:

$$x_1 = \frac{p_1 + b_1}{a_1 + f(w_{11}.p_1 + w_{12}.p_2)}$$

$$x_2 = \frac{p_2 + b_2}{a_2 + f(w_{21}.p_1 + w_{22}.p_2)}$$

The output of this network is:

$$y = g(x_1w_1 + x_2w_2 + b)$$

If we choose g to be the *hard-limiting* function:

$$g(n) = \text{hardlim}(n) = \begin{cases} 1 \text{ if } n \geq 0 \\ 0 \text{ if } n < 0 \end{cases}$$

then the input space (R^2 in this case) can be divided into two regions. One region corresponds to output value g = 1 and the other g = 0. The two regions are separated by a decision boundary defined by the following equation:

$$x_1w_1 + x_2w_2 + b = 0$$

or

$$\frac{w_1(p_1 + b_1)}{a_1 + f(w_{11}.p_1 + w_{12}.p_2)} + \frac{w_2(p_2 + b_2)}{a_2 + f(w_{21}.p_1 + w_{22}.p_2)} + b = 0$$

If f is a linear function f(n) = n, the decision boundary is quadratic because the above equation then can be written as:

$$\begin{aligned} &w_1(p_1 + b_1)(a_2 + w_{21}.p_1 + w_{22}.p_2) + w_2(p_2 + b_2)(a_1 + w_{11}.p_1 + w_{12}.p_2) \\ &+ b(a_1 + w_{11}.p_1 + w_{12}.p_2)(a_2 + w_{21}.p_1 + w_{22}.p_2) = 0 \end{aligned}$$

Much more complex boundaries can be represented if we choose a nonlinear function.

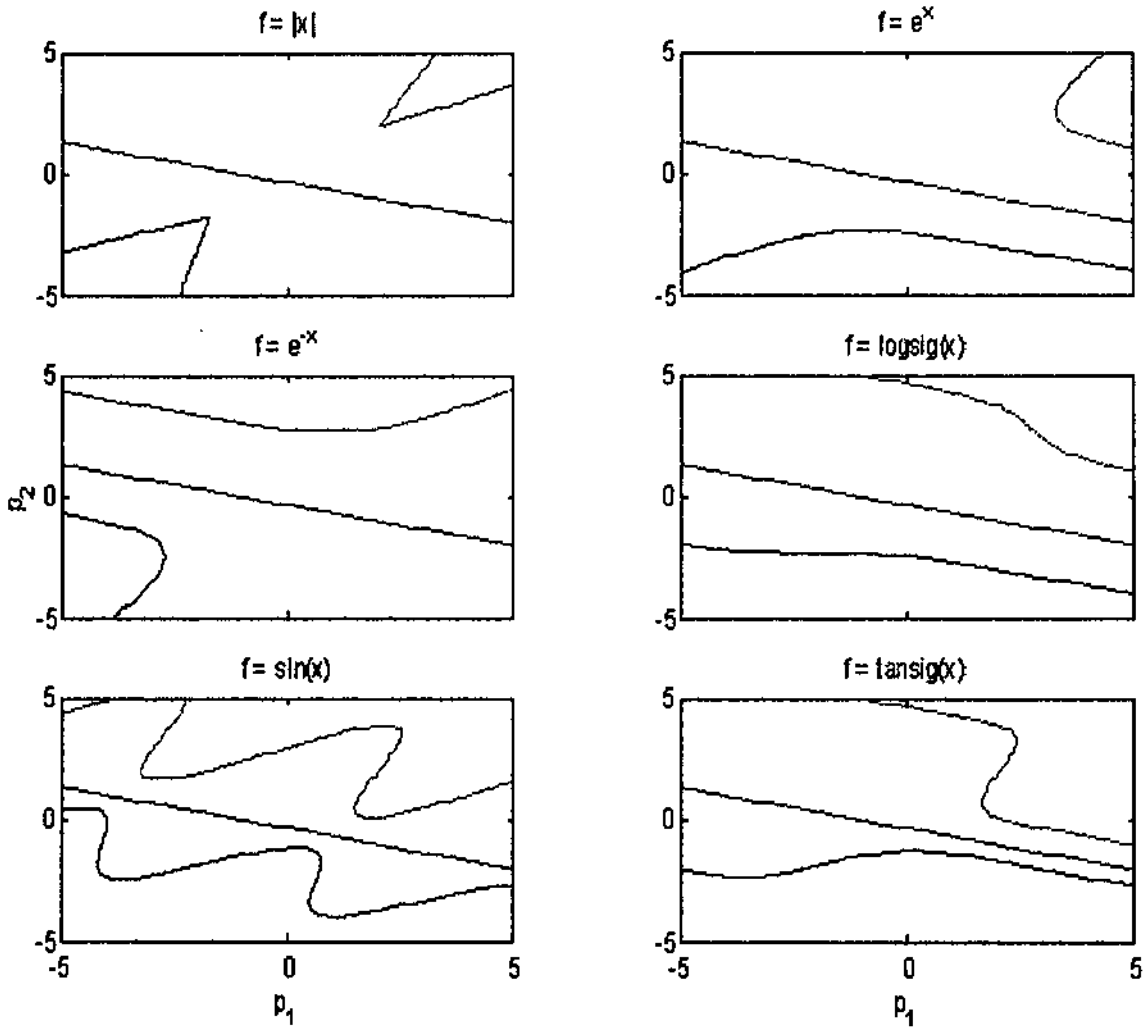


Figure 3.4: Decision surfaces of SIANN's

Examples of decision surfaces for various activation functions are shown in figure 3.4. The network used has parameters as follows:

$w_{11} = -1, w_{12} = 1, w_{21} = -1, w_{22} = 1$

$w_1 = .5, w_2 = 1.5$

$b_1 = 1, b_2 = 0$

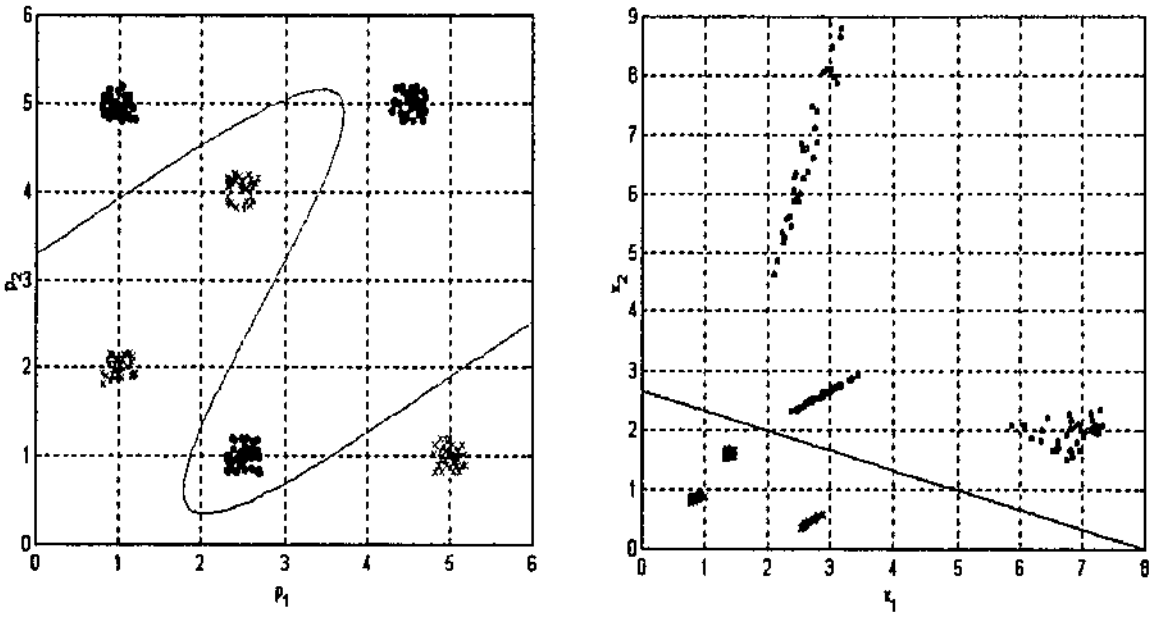
$a_1 = 1.5, a_2 = 1.5$

The red curves correspond to b = 2, the blue curves b = -3, and the green curves b = 0.

Classification example

The ability to represent complex decision boundaries with a very small number of neurons makes SIANN's attractive to classification applications. As an example, we design a SIANN to classify two classes of points (120 red and 120 blue) shown in Figure 3.5a. Clearly, these two classes are not linearly separable in a sense that there exists no single straight line that can separate them. In this example, the role of the two shunting inhibitory neurons is to perform a nonlinear transformation on the inputs:

$$(x_1, x_2) = \Phi(p_1, p_2)$$



(a) Linearly non-separable classes (b) Linearly separable classes

Figure 3.5: Classification example with SIANN

The network (created using the MATLAB functions developed in this project) is able to transform the given points (p_1,p_2) into a new set of points (x_1,x_2) which is linearly separable (Figure 3.5b). The parameters for this network are:

$w_{11} = -w_{12} = -1$, $w_{21} = -w_{22} = -1$

$w_1 = .5$, $w_2 = 1.5$

$b_1 = 1$, $b_2 = 0$, $b = -4$

$a_1 = a_2 = 1.5$

The decision surface of this network is also imposed on figure 3.5a (the magenta curves). The activation functions are $f(x) = \sin(x)$ and $g(x) = \text{hardlim}(x)$.

3.4 Previous works on Shunting Inhibition Model

The concept of lateral inhibition arose in the experimental research of Hartline and colleagues on the compound eye of the "horseshoe crab". Hartline showed that there is reciprocal antagonism between elements of the compound retina, which produces an enhancement of edges and pattern contrast (Nabet & Pinter, 1991).

Bouzerdoum (1991) investigated a class of biologically inspired cellular neural networks called shunting inhibitory cellular neural networks (SICNN's). A cellular neural network is as an array of mainly identical dynamical processing elements called cells, in which interactions are local within a finite radius and all state variables are continuous valued signals (Chua, 1993). The dynamics of a shunting inhibitory cell in SICNN's is described by the following differential equation, which has a similar form to equation 3.1 for feed-forward shunting inhibitory neuron:

$$\frac{dx_{ij}}{dt} = L_{ij} - a_{ij}x_{ij} - \sum_{C(k,l)\in N_r(i,j)} w_{ij}^{kl} f(x_{kl}).x_{ij} \tag{3.4}$$

where x_{ij} is the state of cell (i,j), L_{ij} is the input to cell (i,j), a_{ij} is the decay factor of cell (i,j), f is the activation function, and w_{ij}^{kl} is interaction weight between cell (k,l) and cell (i,j) and N_r is the neighbourhood function.

Bouzerdoum & Pinter (1993) showed that SICNN's are bounded input bounded output stable dynamical systems if f is continuous and non-negative. Furthermore, by deriving a global Liapunov for symmetric SICNN's and using LaSalle invariance principle, they proved that a SICNN converges to a set of equilibrium points; and this set consists of a unique equilibrium point if all inputs have the same polarity.

Pontecorvo and Bouzerdoum (1995) proposed a new approach to edge detection based on a simple model of primate visual system, in particular the cellular neural network model of shunting inhibition. By exploiting some of the inherent nonlinearities of the visual system, SICNN's provide better performance compared with conventional edge detectors.

The shunting inhibitory neural model has also been applied by Beare and Bouzerdoum (1999) in a directionally sensitive motion detection system, which is capable of detecting local motion without any significant pre-processing.

Although SICNN's have been successfully used to model visual processing functions such as edge and motion detection, they have not yet been used for classification and function approximation tasks. One of the main reasons is lack of proper training algorithms. So far connection weights of the networks are chosen empirically by users to suit the task at hand, which limits the scope of applications of these networks. Another reason is that these networks are dynamic systems, and hence to find the output for an input pattern, we must solve a system of nonlinear differential equations. Training such networks is impractical since input patterns have to be presented tens of thousands of times and each time output must be found and the connection weights adjusted; therefore training becomes cumbersome and very time consuming.

As seen early in this chapter, the new neural networks (SIANN's) are also based on the shunting inhibition model. However, they are different from SICNN's in that the output of a shunting inhibitory neuron corresponds to the steady-state solution to the differential equation. Another difference is that SIANN's have a feed-forward architecture. That is neurons in a SIANN receive inputs from the previous layer, compute their outputs and forward them to the next layer.

Having understood the architecture of SIANN's, we are now turning our focus to the derivation of supervised training methods for these networks.

Chapter 4 Training algorithms for SIANN's

In this chapter, we derive training algorithms based on error-backpropagation for SIANN's. We also adapt a very fast training algorithm, which combines error-backpropagation and the direct solution method, to train SIANN's. Finally, a stochastic training method called APOLEX (algorithm for pattern extraction) is discussed.

Before we proceed, let us agree on the notations to be used in this chapter to derive training algorithms for SIANN's. The list of notations is shown in table 4.1.

Table 4.1: Mathematical notations

Symbol	Description
N	Number of layers (excluding the input layer)
s_k	Number of neurons in layer k
$f_k(.)$	Activation function for layer k
${}^{k}w_{ij}$	Weight going *from* neuron j in layer k-1 *to* neuron i in layer k
${}^{k}a_i$	Decay constant for neuron i in layer k
${}^{k}b_i$	Bias for neuron i in layer k
${}^{k}z_{ij}$	Output of neuron i in layer k
${}^{k}p_{ij}$	Input signal *from* neuron j in layer k-1 *to* neuron i in layer k: ${}^{k}p_{ij} = {}^{k-1}z_j$
y_i	Output of neuron i in output layer. That is $y_i = {}^{N}z_i$
t_i	Target (i.e. expected output) of neuron i in the output layer
${}^{k}e_i$	Error sensitivity for neuron i in layer k

Error Function

The *error function* is the function that describes the difference between the net's actual output and its target. The most common error functions are of the type squared-error and we'll use the following version in this chapter:

$$E = \frac{1}{2}.\|\mathbf{Y}-\mathbf{T}\|^2 = \frac{1}{2}\sum_{i=1}^{s_N}(y_i - t_i)^2 \tag{4.1}$$

The constant factor 1/2 is used to simplify mathematical manipulation. Ideally E should be zero, but in practical cases, E is positive and our aim is to minimise E by adjusting the net parameters: weights ${}^{k}w_{ij}$, biases ${}^{k}b_i$, and decay rates ${}^{k}a_i$.

4.1 Error Backpropagation Training

4.1.1 Gradient Descent Method

Because the targets t_i are fixed, E varies with y_i, and is a multi-variable function of the network parameters ${}^{k}w_{ij}$, ${}^{k}ai$, and ${}^{k}b_i$:

$$E = g({}^{k}w_{ij}, {}^{k}a_i, {}^{k}b_i)$$

The gradient descent method to minimise E can be outlined as follows:

1. Start with a random set of net parameters $\{{}^{k}w_{ij}, {}^{k}a_i, {}^{k}b_i\}$
2. Compute the net output and thereby the error E
3. Compute the gradient of the error function E
4. Increment each parameter in the negative direction of the associated partial derivative of E. For example, let v be a parameter (${}^{k}w_{ij}$, ${}^{k}a_i$ or ${}^{k}b_i$). The update rule for v is

$$v_{new} = v_{old} - \alpha_v \cdot \frac{\partial E}{\partial v} \tag{4.2}$$

   where α_v is the learning rate for parameter v, which normally has a small value.
5. Repeat the steps 2 to 4 until E becomes sufficiently small.

4.1.2 Derivation of Error Backpropagation Training for SIANN's

Error Backpropagation refers to a systematic way of calculating the gradient of error function E by propagating error sensitivities from the output layer to inner layers.

Error Sensitivities

The *error sensitivity* associated with neuron ***i*** in layer ***k*** is defined as:

$${}^{k}e_i = \frac{\partial E}{\partial {}^{k}z_i} \tag{4.3}$$

It measures the rate of change in the network error with respect to that in the output of an individual neuron.

Then applying the chain rule, the partial derivative of E with respect to weight ${}^{k}w_{ij}$ is

$$\frac{\partial E}{\partial^{k} w_{ij}} = \frac{\partial E}{\partial^{k} z_i} \cdot \frac{\partial^{k} z_i}{\partial^{k} w_{ij}} = {}^{k}e_i \cdot \frac{\partial^{k} z_i}{\partial^{k} w_{ij}} \tag{4.4}$$

As shown next, the beauty of this method is that all error sensitivities ${}^{k}e_i$ can be calculated in a recursive manner.

Calculating Error Sensitivities

- ***Case k = N*** (for the output layer)

$${}^{N}e_i = \frac{\partial E}{\partial^{N} z_i} = \frac{\partial E}{\partial y_i} = y_i - t_i \tag{4.5}$$

- ***Case k < N*** (for inner layers)

  The error E can be considered as a function of outputs ${}^{k+1}z_j$ ($j = 1,2,\ldots,s_{k+1}$) of the above layer (i.e. layer k+1). By chain rule, the partial derivative of E with respect to ${}^{k}z_i$ is:

$${}^{k}e_i = \frac{\partial E}{\partial^{k} z_i} = \sum_{j=1}^{s_{k+1}} \frac{\partial E}{\partial\, {}^{k+1}z_j} \cdot \frac{{}^{k+1}z_j}{\partial\, {}^{k}z_i}$$

  By definition (4.3), the first term inside the summation sign is the error sensitivity of neuron j in layer k+1:

$${}^{k}e_i = \sum_{j=1}^{s_{k+1}} {}^{k+1}e_j \cdot \frac{\partial\, {}^{k+1}z_j}{\partial\, {}^{k}z_i} \tag{4.6}$$

  Now, we'll calculate the second term, $\partial\, {}^{k+1}z_j / \partial\, {}^{k}z_i$

  For **k = N-1**, the output of the perceptron layer is

$${}^{N}z_j = f_N\left(\sum_{l=1}^{s_{N-1}} {}^{N}w_{jl} \cdot {}^{N-1}z_l + {}^{N}b_j\right)$$

  Thus

$$\frac{\partial^{N} z_j}{\partial^{N-1} z_i} = f'_N\left(\sum_{l=1}^{s_{N-1}} {}^{N}w_{jl} \cdot {}^{N-1}z_l + {}^{N}b_j\right) \cdot {}^{N}w_{ji}$$

  and

$${}^{N-1}e_l = \sum_{j=1}^{s_N} {}^{N}e_j \cdot f'_N\left(\sum_{l=1}^{s_{N-1}} {}^{N}w_{jl} \cdot {}^{N-1}z_l + {}^{N}b_j\right) \cdot {}^{N}w_{ji} \tag{4.7}$$

For **k < N-1**, the output of k+1 layer is

$$ {}^{k+1}z_j = \frac{{}^{k}z_j + {}^{k+1}b_j}{{}^{k+1}a_j + f_{k+1}(\sum_{l=1}^{S_k} {}^{k+1}w_{jl}.{}^{k}z_l)} $$

If $i \neq j$, the term ${}^{k}z_i$ only appears in the denominator, and

$$ \frac{\partial\, {}^{k+1}z_j}{\partial\, {}^{k}z_i} = -\frac{({}^{k}z_j + {}^{k+1}b_j).f'_{k+1}(\sum_{l=1}^{S_k} {}^{k+1}w_{jl}\,.\,{}^{k}z_l).\,{}^{k}w_{ji}}{\left[{}^{k+1}a_j + f_{k+1}(\sum_{l=1}^{S_k} {}^{k+1}w_{jl}.{}^{k}z_l)\right]^2} \tag{4.8a} $$

When $i = j$, then the term ${}^{k}z_i$ appears in both denominator and numerator. The partial derivative has an additional term:

$$ \frac{\partial\, {}^{k+1}z_j}{\partial\, {}^{k}z_i} = -\frac{({}^{k}z_j + {}^{k+1}b_j).f'_{k+1}(\sum_{l=1}^{S_k} {}^{k+1}w_{jl}\,.\,{}^{k}z_l).\,{}^{k}w_{ji}}{\left[{}^{k+1}a_j + f_{k+1}(\sum_{l=1}^{S_k} {}^{k+1}w_{jl}.{}^{k}z_l)\right]^2} + \frac{1}{{}^{k+1}a_j + f_{k+1}(\sum_{l=1}^{S_k} {}^{k+1}w_{jl}.{}^{k}z_l)} \tag{4.8b} $$

Now we use 4.8a and 4.8b to compute error sensitivities for layer k,

$$ {}^{k}e_i = \sum_{j=1}^{s_{k+1}} {}^{k+1}e_j.\frac{\partial\, {}^{k+1}z_j}{\partial\, {}^{k}z_i} $$

If $s_{k+1} < i$, there is no j in 1,2,..., s_{k+1} such that $j = i$. Therefore,

$$ {}^{k}e_i = \sum_{j=1}^{s_{k+1}} {}^{k+1}e_j.\frac{-({}^{k}z_j + {}^{k+1}b_j).f'_{k+1}(\sum_{l=1}^{S_k} {}^{k+1}w_{jl}\,.\,{}^{k}z_l)\,.\,{}^{N}w_{ji}}{\left[{}^{k+1}a_j + f_{k+1}(\sum_{l=1}^{S_k} {}^{k+1}w_{jl}.{}^{k}z_l)\right]^2} \tag{4.9a} $$

If $s_{k+1} \geq i$, there exists j in 1,2,..., s_{k+1} such that $j = i$. Therefore,

$$ {}^{k}e_i = \frac{{}^{k+1}e_i}{{}^{k+1}a_j + f_{k+1}(\sum_{l=1}^{S_k} {}^{k}w_{jl}.{}^{k}z_l)} + \sum_{j=1}^{s_{k+1}} {}^{k+1}e_j.\frac{-({}^{k}z_j + {}^{k+1}b_j).f'_{k+1}(\sum_{l=1}^{S_k} {}^{k+1}w_{jl}.{}^{k}z_l).{}^{N}w_{ji}}{\left[{}^{k+1}a_j + f_{k+1}(\sum_{l=1}^{S_k} {}^{k}w_{jl}.{}^{k}z_l)\right]^2} \tag{4.9b} $$

Calculating the Gradient of E

Once error sensitivities have been calculated, the gradient of E is obtained by simply applying the chain rule of differentiation.

- **Partial derivative with respect to weights**

$$\frac{\partial E}{\partial^k w_{ij}} = \frac{\partial E}{\partial^k z_i}.\frac{\partial^k z_i}{\partial^k w_{ij}} = {}^k e_i.\frac{\partial^k z_i}{\partial^k w_{ij}}$$

For k = N, because the output layer is a perceptron layer

$${}^N z_i = f_N(\sum_{l=1}^{s_{N-1}} {}^N w_{il}.{}^{N-1} z_l + {}^N b_i)$$

$$\frac{\partial E}{\partial^N w_{ij}} = {}^N e_i.\frac{\partial^N z_i}{\partial^N w_{ij}}$$

$$\Rightarrow \frac{\partial E}{\partial^N w_{ij}} = {}^N e_i.f'_N(\sum_{l=1}^{s_{N-1}} {}^N w_{il}.{}^{N-1} z_l + {}^N b_i).\ {}^{N-1} z_j \tag{4.10}$$

For k < N, layer k is a shunting inhibitory layer,

$${}^k z_i = \frac{{}^{k-1} z_i + {}^k b_i}{{}^k a_i + f_k(\sum_{j=1}^{S_k} {}^k w_{ij}.{}^{k-1} z_j)}$$

$$\frac{\partial E}{\partial^k w_{ij}} = {}^k e_i.\frac{\partial^k z_i}{\partial^k w_{ij}}$$

$$\Rightarrow \frac{\partial E}{\partial^k w_{ij}} = {}^k e_i.\frac{-({}^{k-1} z_i + {}^k b_i).f'_k(\sum_{j=1}^{S_{k-1}} {}^k w_{ij}.{}^{k-1} z_j).\ {}^{k-1} z_j}{\left[{}^k a_i + f_k(\sum_{j=1}^{S_{k-1}} {}^k w_{ij}.{}^{k-1} z_j)\right]^2} \tag{4.11}$$

- **Partial derivative with respect to biases**

$$\frac{\partial E}{\partial^k b_i} = \frac{\partial E}{\partial^k z_i}.\frac{\partial^k z_i}{\partial^k b_i} = {}^k e_i.\frac{\partial^k z_i}{\partial^k b_i}$$

For k = N, because the output layer is a perceptron layer

$${}^N z_i = f_N(\sum_{j=1}^{s_{N-1}} {}^N w_{ij}.{}^{N-1} z_j + {}^N b_i)$$

$$\frac{\partial E}{\partial {}^{N}b_i} = {}^{N}e_i . \frac{\partial {}^{N}z_i}{\partial {}^{N}b_i}$$

$$\Rightarrow \quad \frac{\partial E}{\partial {}^{N}b_i} = {}^{N}e_i . f'_N(\sum_{j=1}^{S_{N-1}} {}^{N}w_{ij} . {}^{N-1}z_j + {}^{N}b_i) \tag{4.12}$$

For k < N, layer k is a shunting inhibitory layer,

$${}^{k}z_i = \frac{{}^{k-1}z_i + {}^{k}b_i}{{}^{k}a_i + f_k(\sum_{j=1}^{S_{k-1}} {}^{k}w_{ij} . {}^{k-1}z_j)}$$

$$\frac{\partial E}{\partial {}^{k}b_i} = {}^{k}e_i . \frac{\partial {}^{k}z_i}{\partial {}^{k}b_i}$$

$$\Rightarrow \quad \frac{\partial E}{\partial {}^{k}b_i} = {}^{k}e_i . \frac{1}{{}^{k}a_i + f_k(\sum_{j=1}^{S_{k-1}} {}^{k}w_{ij} . {}^{k-1}z_j)} \tag{4.13}$$

- **Partial derivative with respect to decay rates**

Decay rates ${}^{k}a_i$ only exist for shunting inhibition layers ($k \leq N-1$)

$$\frac{\partial E}{\partial {}^{k}a_i} = \frac{\partial E}{\partial {}^{k}z_i} . \frac{\partial {}^{k}z_i}{\partial {}^{k}a_i}$$

$$= {}^{k}e_i . \frac{\partial {}^{k}z_i}{\partial {}^{k}a_i}$$

$$\Rightarrow \quad \frac{\partial E}{\partial {}^{k}a_i} = {}^{k}e_i . \frac{-({}^{k-1}z_i + {}^{k}b_i)}{\left[{}^{k}a_i + f_k(\sum_{j=1}^{S_{k-1}} {}^{k}w_{ij} . {}^{k-1}z_j)\right]^2} \tag{4.14}$$

The steps in error-backpropagation algorithm for SIANN's are summarised in the next section.

4.1.3 Summary of Error Backpropagation training for SIANN's

Initialise network

Step 1: Initialise network

Set parameters such as weights, biases, decay rates to small random values.

Compute network output

Step 2: For k from 1 to N-1, compute the output of layer k

$$ {}^{k}z_i = \frac{{}^{k-1}z_i + {}^{k}b_i}{{}^{k}a_i + f_k(\sum_{j=1}^{S_{k-1}} {}^{k}w_{ij}.{}^{k-1}z_j)} \quad , i=\overline{1,s_k} $$

The output of one layer becomes the input to the next layer.

Step 3: Compute the output of the last layer

$$ y_i = {}^{N}z_i = f_N(\sum_{j=1}^{S_{N-1}} {}^{N}w_{ij}.{}^{N-1}z_j + {}^{N}b_i) \ , i=\overline{1,s_N} $$

Compute error

Step 4: Compute the sum squared error

$$ E = \frac{1}{2}\| Y - T \|^2 = \frac{1}{2}\sum_{i=1}^{S_N}(y_i - t_i)^2 $$

Compute error sensitivities

Step 5: Compute error sensitivities for the output layer

$$ {}^{N}e_i = y_i - t_i \ , \quad i=\overline{1,s_N} $$

Step 6: Compute error sensitivities at layer N-1

$$ {}^{N-1}e_i = \sum_{j=1}^{s_N} {}^{N}e_j . f'_N(\sum_{l=1}^{s_{N-1}} {}^{N}w_{jl} . {}^{N-1}z_l + {}^{N}b_j).{}^{N}w_{ji} \quad , i=\overline{1,s_{N-1}} $$

Step 7: For k from N-2 down to 1, compute error sensitivities for layer k

$$ {}^{k}e_i = \sum_{j=1}^{s_{k+1}} {}^{k+1}e_j . \frac{-f'_{k+1}(\sum_{l=1}^{s_k} {}^{k+1}w_{jl} . {}^{k}z_l).\ {}^{N}w_{ji}}{\left[{}^{k+1}a_j + f_{k+1}(\sum_{l=1}^{S_k} {}^{k+1}w_{jl}.{}^{k}z_l) \right]^2} \quad , i=\overline{1,s_k} $$

If $i \leq s_{k+1}$, add an additional term to the error sensitivity

$$ {}^{k}e_i = {}^{k}e_i + \frac{{}^{k+1}e_i}{{}^{k+1}a_i + f_{k+1}(\sum_{l=1}^{S_k} {}^{k+1}w_{il}.{}^{k}z_l)} $$

Compute gradient of E

Step 8: Compute partial derivative of E with respect to weights ${}^{k}w_{ij}$

For the output layer

$$\frac{\partial E}{\partial {}^{N}w_{ij}} = {}^{N}e_i . f'_N\left(\sum_{l=1}^{s_{N-1}} {}^{N}w_{il} . {}^{N-1}z_l + {}^{N}b_i\right) . {}^{N-1}z_j \quad , \; i = \overline{1, s_N} \text{ and } j = \overline{1, s_{N-1}}$$

$$\frac{\partial E}{\partial {}^{k}w_{ij}} = {}^{k}e_i . \frac{-({}^{k}z_i + {}^{k}b_i) . f'_k\left(\sum_{j=1}^{S_{k-1}} {}^{k}w_{ij} . {}^{k-1}z_j\right) . {}^{k-1}z_j}{\left[{}^{k}a_i + f_k\left(\sum_{j=1}^{S_{k-1}} {}^{k}w_{ij} . {}^{k-1}z_j\right)\right]^2} \quad , \; i = \overline{1, s_k} \text{ and } j = \overline{1, s_{k-1}}$$

For layer k, k < N

Step 9: Compute partial derivative of E with respect to biases ${}^{k}b_i$

For the output layer

$$\frac{\partial E}{\partial {}^{N}b_i} = {}^{N}e_i . f'_N\left(\sum_{l=1}^{s_{N-1}} {}^{N}w_{il} . {}^{N-1}z_l + {}^{N}b_i\right) \quad , \; i = \overline{1, s_N}$$

For layer k = 1,2,..., N-1

$$\frac{\partial E}{\partial {}^{k}b_i} = {}^{k}e_i . \frac{1}{{}^{k}a_i + f_k\left(\sum_{j=1}^{S_{k-1}} {}^{k}w_{ij} . {}^{k-1}z_j\right)} \quad , \; i = \overline{1, s_k}$$

Step 10: Compute the partial derivative of E with respect to decay rates ${}^{k}a_i$

$$\frac{\partial E}{\partial {}^{k}a_i} = {}^{k}e_i . \frac{-({}^{k-1}z_i + {}^{k}b_i)}{\left[{}^{k}a_i + f_k\left(\sum_{j=1}^{S_{k-1}} {}^{k}w_{ij} . {}^{k-1}z_j\right)\right]^2} \quad , \; i = \overline{1, s_k} \text{ and } k = \overline{1, N-1}$$

Update network parameters

Step 11: Update all network parameters weights, biases and decay rates as follows

$$v_{new} = v_{old} - \alpha . \frac{\partial E}{\partial v}$$

where v is a network parameter (${}^{k}w_{ij}$, ${}^{k}b_i$ or ${}^{k}a_i$), α is the learning rate and $\partial E/\partial v$ is the partial derivative computed in steps 8, 9, 10.

Repeat steps 2-11 until E is smaller than the error tolerance, which is a small value set before training.

4.2 Variations of Error Backpropagation Training

In this section, we investigate three heuristic variations of the standard backpropagation algorithm that are believed to increase convergence speed and training stability.

4.2.1 Batch training

The algorithm derived in section 4.1 updates networks parameters for each pair of input/target. If the training sets contains many such pairs, learning can be slow due to "unlearning" effect whereby minimising error for an individual pair may lead to an increase in errors for other pairs. There exists a better approach known as batch training in which updates only occur after each epoch (an epoch is a run through the entire training set). With this approach, each training pair contributes an additive term (calculated in the same way as in section 4.1.5) to partial derivative $\partial E/\partial v$. The cumulative gradient ∇E is used for updating network parameters. Batch training is suitable for off-line training. For online training where training pairs are presented to the network one after another, batch training should not be used.

4.2.2 Gradient Descent With Momentum (GDM)

Let v(n) be the value of network parameter v after n updates. The GDM update rule takes into account the amount of previous update:

$$v(n+1) = v(n) - \alpha.\frac{\partial E}{\partial v} + \beta.[\,v(n) - v(n-1)\,]$$

where β is a small constant called *momentum coefficient*. The effects of momentum term $\beta[v(n) - v(n-1)]$ is to smooth out oscillation in parameter updates. To see this, let's write the update rule in alternative form:

$$\Delta v(n) = -\alpha.\frac{\partial E}{\partial v} + \beta.\Delta v(n-1) \quad \text{where} \quad \Delta v(n) = v(n+1) - v(n)$$

This is equation of a low-pass filter whose input is $-\alpha.\partial E./\partial v$ and output is $\Delta v(n)$.

4.2.3 Gradient Descent With Variable Learning Rate (GDV)

Generally, the error surface is flat in some region and steep in others. The speed of convergence can be increased significantly by adjusting the learning rate during training (e.g. by increasing the learning rate in the "flat" region). A version of variable learning rate suggested by Hagan , et al. (1996) is adapted in this project to train SIANN:

- If the squared error E increases by more than some set percentage ζ after a weight update, then the weight update is discarded, the learning rate is multiplied by some factor $0<\rho<1$.
- If the squared error E increases by less than ζ, then the weight update is accepted and the learning rate is unchanged.
- If the squared error E decreases after a weight update, then the weight update is accepted and the learning rate is increased by a factor of $1/\rho$.

ζ is usually smaller than five percent and ρ is between 0.9 and 0.95.

4.3 Gradient Descent with Direct solution step (GDD)

This section investigates a very fast training method for SIANN's that combines iterative and direct solution methods. In this approach, the SIANN is trained by initially selecting small random weights and then presenting all training data repeatedly. The weights are adjusted after each iteration. After some iteration, or when the training process is trapped at a local minimum; training of the SIANN is paused, and the weights of the output layer and inner layers are calculated using the direct solution method. Verma (1997) first proposed this training method for Multilayer Perceptron. However for it to be applicable to SIANN, significant modifications are required.

Assume that the training set consists of n vectors P(l), l = 1, 2, ...,n and the output of neuron i in layer k for input vector P(l) is ${}^{k}z_i(l)$.

The output for neuron i in the last layer is:

$$ {}^{N}z_i(l) = f_N\left(\sum_{j=1}^{s} {}^{N-1}z_j(l).{}^{N}w_{ij} + {}^{N}b_i\right) $$

We want this output to be as close as possible to the target value: ${}^{k}z_i(l) \approx T_i(l)$. If the activation function $f_N(.)$ has an inverse, this can be written as

$$\sum_{j=1}^{s} {}^{N-1}z_j(l).{}^{N}w_{ij} + {}^{N}b_i = f_N^{-1}(T_i(l))$$

If we let $\mathbf{X} = [{}^{N}w_{i1}\ {}^{N}w_{i2} \ldots {}^{N}w_{is}\ {}^{N}b_i]$ and $\mathbf{A}(l) = [{}^{N-1}z_1(l)\ {}^{N-1}z_2(l) \ldots {}^{N-1}z_{is}(l)\ 1]'$, then in matrix form

$$\mathbf{X} * \mathbf{A}(l) = f^{-1}{}_N(Ti(l)),\ l = 1, 2, \ldots n$$

or

$$\mathbf{X} * \mathbf{A} = f^{-1}{}_N(\mathbf{T}_i)$$

This is a system of linear equations with unknown **X**. In most cases, the system is over-determined (i.e. $n >> s+1$) and there exists no solution for **X**. However, we can determine an $\mathbf{X}^*$ that minimises the squared error between the left-hand side and the right-hand side (least-squared method). This $\mathbf{X}^*$ provides us with weights and biases that make the output of neuron i nearest to its target.

This method can also be applied to shunting inhibitory layers by observing that

$${}^{k}z_i(l) = \frac{{}^{k-1}z_i(l) + {}^{k}b_i}{{}^{k}a_i + f_k(\sum_{j=1}^{S_{k-1}} {}^{k}w_{ij}.{}^{k-1}z_j(l))}$$

After a few manipulations, we obtain an equation for the weighted input sum:

$$\sum_{j=1}^{S_{k-1}} {}^{k}w_{ij}.{}^{k-1}z_j(l) = f_k^{-1}(\frac{{}^{k-1}z_i(l) + {}^{k}b_i}{{}^{k}z_i(l)} - {}^{k}a_i)$$

The right-hand side and ${}^{k-1}z_j(l)$ are known. This also gives us a system of linear equations with weights ${}^{k}w_{ij}$ as the unknown to which a least squared solution can be found.

This direct solution approach must be combined with the iterative error backpropagation training because two successive applications of direct solution always give the same error. This is because there is almost no change in the coefficients of the systems of linear equations. One question remains is when to apply direct solution step during training process. There are two options, and the first one is after every certain number of epochs.

The second is when there is '*little*' change in the error but it is still high, which is a sign that training is stuck in a local minimum. If this is the case, we may also restart training with a new random set of weights, biases and decay rates.

4.4 APOLEX algorithm

The last training algorithm for SIANN's to be investigated in this project is called APOLEX (an acronym for algorithm for pattern extraction). An in depth discussion of this algorithm can be found in [Pandy & Macy, 1996]. The APOLEX procedure was first studied in connection with the problem of ascertaining the shapes of visual receptive fields. Harth and Pandya (1988) apply the APOLEX algorithm to optimise a scalar function of N parameters where N is very larger number.

Unlike backpropagation, which computes new net parameters (weight, biases, decay rates) by recursively computing the gradient through back propagation of error sensitivities, APOLEX works by broadcasting the same global error to all layers. Net parameters are updated stochastically based on a correlation between the change in a parameter and the change in the global error.

Pandy & Macy (1996) outline several advantages of APOLEX:

- The APOLEX does not assume or require any special choice of activation function.
- It is computationally simple (no calculation of derivatives) and can be realised in VLSI since interconnections between processing elements which update the weights are not required.
- APOLEX is observed to generalise better than many of the alternative algorithms (including backpropagation) especially over noisy data sets.
- It is applicable to many neural network architectures.

Next, we'll describe quantitatively the APOLEX algorithm for SIANN's. Let v be a parameter of SIANN (weight, bias, or decay rate). To calculate new v, we need to store the previous change in v, and the total error:

Parameter Change $\Delta v = v(n-1) - v(n-2)$

Error Change $\Delta E = E(n-1) - E(n-2)$

The same as for other training methods, error E is computed by formula (4.1). The correlation between the change in v and E is defined as:

$$c_v(n) = \Delta v \,.\, \Delta E$$

The direction of change is determined by a probability function (T_v is a positive coefficient called *temperature*) :

$$P_v(n) = \frac{1}{1 + e^{\frac{c_v(n)}{T_v}}}$$

The amount of change is a small positive constant δ. Together the change in v is calculated as follows:

$$\Delta_v(n) = \begin{cases} -\delta & \text{with probability of } 1 - P_v(n) \\ \delta & \text{with probability of } P_v(n) \end{cases}$$

and $v(n) = v(n-1) + \Delta_v(n)$.

Table 4.2: Scenarios for APOLEX

	$\Delta v > 0$	$\Delta v < 0$
$\Delta E > 0$	$c_v(n) > 0$ $P_v(n) < 1 - P_v(n)$ Decrease v	$c_v(n) < 0$ $P_v(n) > 1 - P_v(n)$ Increase v
$\Delta E < 0$	$c_v(n) < 0$ $P_v(n) > 1 - P_v(n)$ Increase v	$c_v(n) > 0$ $P_v(n) < 1 - P_v(n)$ Decrease v
Else if $c_v(n) = 0$ then no clear trend		

Possible update scenarios are shown in table 4.2 which is adapted from [Pandy & Macy, 1996]. As an example, if the previous *increase* in v (i.e. $\Delta v > 0$) results in an *increase* in the error E (i.e. $\Delta E > 0$), then their product – the correlation $c_v(n)$ – is positive. It can be seen easily, in this case, that $P_v(n) < 0.5 < 1 - P_v(n)$. As a result, there is more chance that v is decreased in this update ($\Delta_v(n) = -\delta < 0$) so as to reduce E.

The algorithm takes a biased random walk in the direction of decreasing error E and the temperature T_v determines the effective randomness of the walk. T_v should be chosen large initially (to compensate for large covariance) and decreased as training progresses to the convergent value of the correlation:

$$T_v = \lim_{n \to \infty} |c_v(n)|$$

4. 5 Chapter Summary

In this chapter, we derived the backpropagation-type algorithm for shunting inhibitory neural networks. The algorithm developed is significantly more complex compared to that of MLP, which is understandable since shunting neurons have more complex activation than perceptrons. We discussed three heuristic variations of backpropagation training, which can be implemented for SIANN: batch training, gradient descent with momentum (GDM), and gradient descent with variable learning rate (GDV). Also derived in this chapter is a very fast training method based on direct solution. This method can potentially speed up training by finding the least-squared solution to systems of linear equation whose unknown are weights and biases. The last training method developed for this project is APOLEX algorithm, which updates net parameters stochastically based on a correlation between change in a parameter and change in the global error. APOLEX algorithm is computationally simple because it requires neither calculation of derivatives nor backpropagation of error sensitivities. Furthermore, there is little restriction on the type of activation functions.

Chapter 5 MATLAB Implementation

This chapter focuses on the MALAB implementation of SIANN's and the training methods developed in the previous chapter. The chapter first presents an overview of functions developed in this project. Second, it describes how shunting inhibitory artificial neural networks are mapped into a structure data type in MATLAB. The rest of the chapter is on how SIANN's are created, initialised, simulated and trained. The listings of MATLAB code on this chapter can be found in appendix A.

5.1 System Overview

At the highest level, the SIANN toolbox to be implemented consists of four main functions (Figure 5.1).

new_siann creates a new shunting inhibitory artificial neural network.

init_siann initialises parameters in a SIANN network including weights, biases, decay rates, and training parameters.

sim_siann simulates a SIANN, i.e. computes its final output for a given input

train_siann trains a SIANN, i.e. adjusts the network's parameters so that it produces final output within a defined tolerance of the desired output.

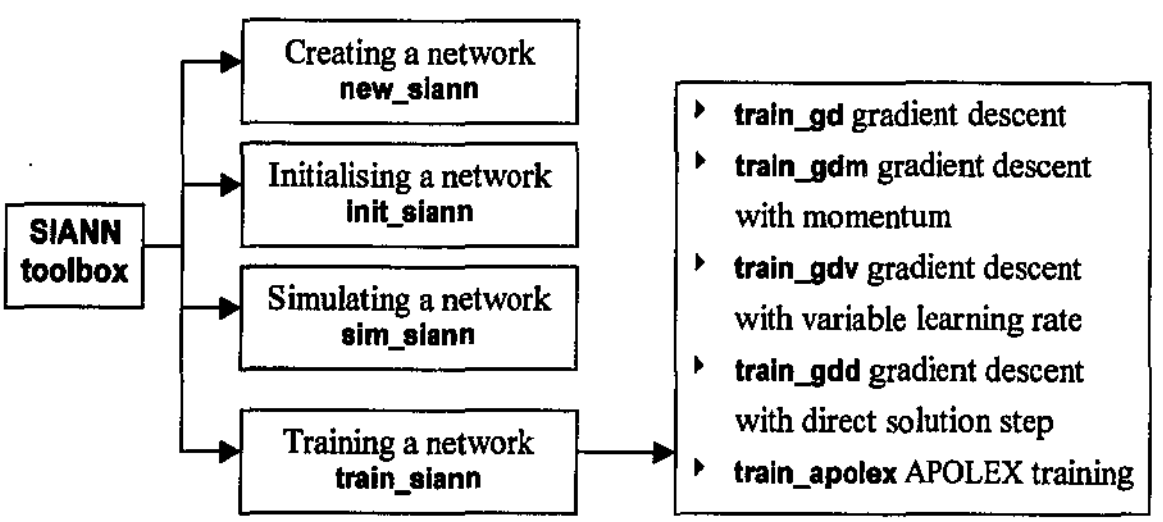


Figure 5.1: Overview of SIANN functions

Depending on which training algorithm is to be used, train_siann calls one of the training functions: train_gd, train_gdm, train_gdv, train_gdd, and train_apolex.

5.1.1 Why MATLAB?

The purpose of this section to justify the choice of MATLAB as a tool to implement SIANN's and their training methods. In simplest term, MATLAB - developed by The MathWorks Inc. - is a computer environment for performing calculations. The name MATLAB, which stands for MATrix LABoratory, suggests that MATLAB was designed initially for the manipulation of matrices. It has since added more functionality and it still remains a better tool for scientific computation.

MATLAB is a procedural programming language that combines an efficient programming structure with a bevy of predefined mathematical commands. One of the major advantages of MATLAB over other languages such as C and C++ is its ability to perform complex mathematical operations such as matrix algebra in a few lines of code. As will be seen later in this chapter, using MATLAB's matrix operators makes the code that implements the backpropagation algorithm 4.1.5 shorter and easier to understand. The rich set of MATLAB functions, which ranges from mathematical functions, plotting functions to GUI tools, also greatly simplifies coding and testing of SIANN's and training methods. As a result, more time is available for testing different approaches and improving the design.

MATLAB with its Neural Network Toolbox implements a vast number of neural network architectures. The most notable is Multilayer Perceptrons and several supervised training algorithms including gradient descent, conjugate gradient and Levenberg-Marquardt. The toolbox contains many common activation functions for neural networks such as tansig, purelin, and logsig. The Neural Network Toolbox is an excellent example on how to implement a neural network and its training methods. In fact, several code-optimisation techniques were learned by looking at scripts included in the toolbox. We have attempted to give SIANN functions similar interface as those in the MATLAB's Neural Network Toolbox.

5.1.2 SIANN functions: Usage Example

The short code example in listing 5.1 shows how the main functions are used. Any thing in a line after the percentage sign (%) is ignored by MATLAB. How these functions are specified and implemented will be discussed in great detail in subsequent sections.

Listing 5.1: Using main functions in the toolbox

```
P = [0 0 1 1;
     1 0 1 0]  % network input
T = [1 0 0 1]  % desired network output i.e. approximate XOR function
% Create a new SIANN
net = new_siann([0 1; 0 1],[2 1],{'exp' 'purelin'},'train_gdd', 'sse');
net = init_siann(net);            %Initialise SIANN parameters
Y = sim_siann(net,P)              %Compute network output
[net, tr] = train_siann(net,P,T);%Train SIANN to produce desired output
Y = sim_siann(net,P)              %Compute network output after training
```

5.2 System Design

MATLAB offers a *network* class that can be customised to create different neural networks. However, this class is not used in this project because it does not provide (after trial) satisfactory speed performance. The reason is that to accommodate as many as possible architectures (dynamic, static, recurrent, feed-forward, etc.) the class contains many settings, which are not needed in SIANN's.

5.2.1 SIANN Structure

In this project, SIANN's are implemented using a new data type introduced in MATLAB 5 called *structures*. Structure data type offers several advantages:

- It allows all information about a SIANN including architectural information such as the number of layers, the number of neurons per layer, operational information such as the values of connection weights, biases, and training information such as training

method, error tolerance, maximum number of iterations to be stored in a single variable.

- Structure data type is actually a built-in class in MATLAB, and therefore offers certain level of object-oriented programming. It is flexible in that fields can be added or removed from the network dynamically. This is very important because to implement a new training algorithm that uses new training parameters, we do not need to recreate the network from scratch. Instead new fields can be added into the set of training parameters during initialisation phase (i.e. in *init_siann*) and will be available for controlling training process (i.e. in *train_siann*).

- Compared with the approach of creating a separate SIANN class with its own constructors and member functions, using structures achieves faster execution speed. In fact, inside several computation-intensive functions (e.g. *train*) included in MATLAB's Neural Network Toolbox, network class is first converted into a structure (using *struct* command), and after processing is returned back to class (with *class* command).

The fields of a SIANN structure are listed in table 5.1. To access a field of a structure variable, use the variable name followed by a dot (.) and the field name. For example, let *net* be a SIANN, then its number of layers is *net.numLayers* and can be set as follows:

```
net.numberLayers =  4;  % set the number of layers to 4
```

Several fields in SIANN structure deserve an in-depth discussion. However, we need first to describe another new data type introduced in MATLAB 5: *cell arrays*. Unlike a normal array where all of its elements are scalars. A cell array, as the name suggests, is an array of cells, each of which can store a value of almost any MATLAB data type: scalar, string, array, structure, and even cell array. As will be seen shortly, cell arrays whose each cell is

an array or a structure provide very efficient and convenient storage for SIANN. To access the ith element of a cell array, we use the format: `cell_array_name{i}`.

Table 5.1: Fields in SIANN structure

	Field	Data type	Description
Architectural	numInputs	scalar	**Number of elements in each input vector**
	numOutputs	scalar	Number of elements in each output vector
	numLayers	scalar	Number of layers excluding the input layer
	layers	cell array	Information about each layer
	.size	scalar	Number of neurons in the layer
	.transferFcn	string	Transfer function for the layer
Operational	W	cell array	Weights
	B	cell array	Biases
	A	cell array	Decay rates
Training	trainParam	structure	Training parameters
	.trainFcn	string	Training function
	.goal	scalar	Error tolerance
	.epochs	scalar	Maximum number of iterations
	.errorFcn	string	Error function
	.lr_w	scalar	Learning rate for weights
	.lr_b	scalar	Learning rate for biases
	.lr_a	scalar	Learning rate for decay rates
	.show	scalar	Frequency of showing training progress
Others	inputRange	array	Range of input values

Information stored in a SIANN structure can be divided into four main categories: architectural information, operational information, training information and others.

Architectural information is information regarding the architecture of the neural network. That includes the size of an input vector (i.e. number of neurons in the input layer), the size of output vector (i.e. number of neurons in the output layer), and the number of layers. Information about layers (excluding the input layer) is stored in the field *Layers*, which is an array of structures. The number of structures is dictated by the number of layers - *numLayers*, and the ith structure stores information about layer i:

- layers{i}.*size* is the number of neurons in layer i
- layers{i}.*transferFcn* is the transfer function of neurons in layer i. It is in the form of a string such as 'purelin', 'tansig', etc. We should mention that *transfer function* is MATLAB's terminology for what we have been using so far: *activation function*. These two terms can be used interchangeably.

Operational information includes the values of weights, biases and decay rates that allow output of a SIANN to be computed.

- **W** is a cell array. Its kth cell is an array that stores weights going from layer k – 1 to layer k. We consider here that layer 0 is the input layer. Weight ${}^{k}w_{ij}$ going from neuron j in layer k-1 to neuron i in layer k is W{k}(i,j).
- **B** is a cell array. Its kth cell is a vector that stores biases for neurons in layer k Bias ${}^{k}b_i$ for neuron i in layer k is B{k}(i).
- **A** is a cell array. Its kth cell is a vector that stores decays rate for neurons in layer k Decay rate ${}^{k}a_i$ for neuron i in layer k is A{k}(i).

Training information controls the training process of a SIANN and is stored in the field *trainParam*. trainParam is a structure and by default it consists of fields that are used in most training algorithms:

- *trainFcn* is a string that specifies the training algorithm. For example if trainFcn is set to 'train_gd' then the error backpropagation with gradient descent will be used to train the network
- *goal* is error tolerance. Training stops when the difference between the network's actual output and its target falls below this value. How this difference is computed is dictated by the field errorFcn.
- *errorFcn* is a string that specifies how error is computed, and is either '*sse*' or '*mse*'. 'sse' returns sum squared error between actual output matrix Y and target matrix T. This error divided the number of elements in T is the result returned by 'mse' (short for mean squared error).
- *epochs* maximum number of epochs. An epoch, in the context of batch training, corresponds to an network update after the whole training set is presented. Training also stops if this number is reached.
- *show* specifies the rate at which the training progress is reported to user. For example, if show is 25, then after every 25 epochs, training status including error, epoch number, is displayed at MATLAB console.
- *lr_w, lr_b, lr_a* are learning rates for weights, biases and decay rates respectively.

As mentioned above, fields can be added to trainParam structure to accommodate parameters required by new training algorithms. For example, the gradient descent with momentum method (see 4.2.2 and 5.6.2) uses a parameter called β (beta) which is typically set to 0.1. This parameter is added to the set of training parameters by init_siann function if it finds the network's training method is 'train_gdm':

```
net.trainParam.beta = 0.1;
```

Other information

- *inputRange* is a *numInputs* x 2 array that specifies the range of input values to the network. It may be used by some training algorithms to set the initial weights, biases, and decay rates so as to speed up convergence.

Table 5.2 shows the mapping between mathematical notations (used in derivation of training algorithms in chapter 4) and MATLAB variables.

Table 5.2: Mathematical notations to MATLAB variables

Description	Symbol	MATLAB
Number of layers (excluding the input layer)	N	net.numLayers
Number of neurons in layer k	s_k	net.layers{k}.size
Activation function for layer k	$f_k(.)$	net.layers{k}.transferFcn
Weight *from* neuron j in layer k-1 *to* neuron i in layer k	${}^{k}w_{ij}$	net.W{k}(i,j)
Decay constant for neuron i in layer k	${}^{k}a_i$	net.A{k}(i)
Bias for neuron i in layer k	${}^{k}b_i$	net.B{k}(i)
Input to the net	P	P or Z{1}
Output of neuron i in layer k	${}^{k}z_{ij}$	Z{k+1}(i,:)
Input signal *from* neuron j in layer k-1 *to* neuron i in layer k	${}^{k}p_{ij}$	Z{k}(j,:)
Output of neuron i in output layer	y_i	Y(i,:) or Z{N+1}(i,:)
Target of neuron i in the output layer	t_i	T(i,:)

5.2.2 Other Design Considerations

This section discusses two issues that are important in achieving good performance for MATLAB implementation.

Memory versus Speed

Investigating the formulae in section 4.1.5, we can see many similar calculations are repeated in various steps of the training process. For example, outputs ${}^{k}z$ of each layer computed in feedforward phase are used in backpropagation phase to compute sensitivities. Training can be sped up if memory is allocated to store the result of these calculations for later use.

Improving Speed through Vectorisation

The execution speed of MATLAB's scripts can be greatly improved if operations are expressed in vector form. The following example of adding two vectors illustrates this point.

```
a = rand(1000,50);
b = rand(1000,50);
c = zeros(1000,50);
```

Test 1

```
for i = 1:50000
   c(i) = a(i) + b(i); %Individual element adding
end
```

Test 2

```
c = a + b; %Vector form
```

In one Pentium machine, test 1 took 12.25 seconds whereas test 2 took 0.06 seconds.

Having described the SIANN structure, we are now ready to develop SIANN functions. The focus of the next four sections is on how SIANN functions are implemented in MATLAB. The discussion will be facilitated by code examples where their inclusion is appropriate. Refer to appendix A at the end of this report for a complete code listing.

5.3 Creating SIANN Network

File

new_siann.m listing A.1

Syntax

net = new_siann(pr,[s1 s2...sNl],{tf1 tf2...tfNl}, btf, ef)

Inputs

pr	Rx2 matrix of min and max values for R input elements
si	number of neurons in layer i
tf	transfer function for layer i
btf	backpropagation training function
ef	error function

Outputs

net	a new SIANN network

Discussion

The following example creates a SIANN that receives 3 inputs, each ranges from 0 to 6. The network has three layers whose number of neurons is 5, 4 and 1. In that order, the transfer functions are 'exp', 'tansig' and 'purelin'. The net is trained with training function 'train_gd' and its error function is 'sse':

```
net = new_siann([0 6; 0 6; 0 6], [5,4,1],{'exp' 'tansig' 'purelin'},'train_gd','sse');
```

5.4 Initialising SIANN Network

File

inint_siann.m listing A.2

Syntax

net = init_siann(net)

Inputs

net	a SIANN net

Outputs

net	an initialised SIANN net

Discussion

This function performs two main tasks. First it initialises net parameters (weights, biases, decay rates) to small random values. Second, depending on the training algorithm of the net, it creates the required training parameters in trainParam structure, and sets them to default values. The following code shows initialisation for the GDD (gradient descent with direct solution step):

```
switch net.trainParam.trainFcn
   ...
   case 'train_gdd'  %gradient descent with direct solution
   net.trainParam.direct = 2;
   %i.e Direct solution takes place after every 2 epochs
   %Find and store inverse of transfer function
   for i = 1:net.numLayers
      transferInv = 'purelin'; %By default
      switch net.layers{i}.transferFcn
         case 'exp'
            transferInv = 'log';
         case 'purelin'
            transferInv = 'purelin';
         case 'tansig'
            transferInv = 'atanh';
         case 'logsig'
            transferInv = 'invlogsig';
      end
      net.layers{i}.transferInv = transferInv;
   end
   ...
end
```

5.5 Simulating SIANN Network

File

sim_siann.m listing 5.2 or A.3

Syntax

[Y, Z, D, S] = sim_siann(net, P)

Inputs

net Network

P Network input

Outputs

Y - Network output

Z - Output of individual layer

D - Denominator factor a + f(sum)

S - Input sum to each neuron

The last three outputs Z, D, S are used in training (see section 5.6.1).

Example

P = [1 3 6 8;

2 4 7 5]; % input

net = new_siann([0 3; -1 4], [4 3], {'tansig' 'purelin'});

Y = sim_siann(net, P) %output

Listing 5.2: sim_siann Computing the output of SIANN

```
function [Y, Z, D, S] = sim_siann(net,P)
%SIM_SIANN Simulate a Shunting Inhibitory Artificial Neural Network.
%Check arguments
if nargin ~= 2
  error ('Not enough input arguments.');
end

% Simulate network
N = net.numLayers;    % The number of layers
Ni = size(P,2);       % The number of input vectors in the training set
Z = cell(N + 1,1);    % Storage for N layers' output
D = cell(N-1,1);      % Storage for denominator factor a + f(sum)
S = cell(N,1);        % Storage for sum W*P
Z{1} = P;             % Input set considered as output of layer 0

% Z{k} stores the output of layer k-1
% Compute the output for all layers except the last layer
for k = 1:N-1
   T = net.layers{k}.size - size(Z{k},1);
   if T > 0
      ZT = [Z{k}; zeros(T,Ni)];
   else
      ZT = Z{k}(1:net.layers{k}.size,:);
   end
   S{k} = net.W{k} * Z{k};                        %Weighted sum
   D{k} = repmat(net.A{k},1,Ni) + ...
         feval(net.layers{k}.transferFcn,S{k}); %Denominator
   Z{k+1} = (ZT + repmat(net.B{k},1,Ni)) ./ D{k};
end
```

```
% Computer the output of the last
% layer -       a perceptron layer
S{N} = net.W{N} * Z{N} + repmat(net.B{N}, 1, Ni);
Z{N+1} = feval(net.layers{N}.transferFcn, S{N});
Y = Z{N+1};                %Final output
```

Discussion

This function computes the output of SIANN for a given input. Input is a matrix organised column-wise, i.e. each column is an individual input pattern. In the above example, there are four input patterns, each consists of two elements. Similarly the output is organised column-wise. This function also returns the output Z of all layers, the weighted sum **W*P** and the denominator factor a + f(**W.P**) for all neurons, which are needed during training to compute gradient.

The MATLAB function *feval* is used at several times in this project to evaluate a function given its name as a string and its numeric argument. There are two methods of evaluating an expression, for example $e^{1.5}$, in MATLAB by using either:

```
exp(1.5)     %exp is a built-in MATLAB function for e^x
```

or

```
feval('exp', 1.5)
```

The use of *feval* in the second method gives us the flexibility of incorporating almost any activation function for SIANN's. All we need to do is writing a script for the new activation function and passing its name to feval.

5.6 Training Network

File

train_siann.m listing 5.3 or A.4

Syntax

[net, tr] = train_siann(net, P, T)

Inputs

net	Network
P	Network input
T	Network target

Outputs

net - Trained network

tr - Training record

Listing 5.3: train_siann Supervised training of SIANN

```
function [net,tr]=train_siann(net,P,T)
%TRAIN_SIANN Train a Shunting Inhibitory Artificial Neural Network
% Check arguments
if nargin ~= 3
  error('Not enough input arguments.');
end

switch net.trainParam.trainFcn
   case 'train_gd'
      %standard gradient descent
      [net,tr] = train_gd(net,P,T);
   case 'train_gdm'
      %gradient descent with momentum
      [net,tr] = train_gdm(net,P,T);
   case 'train_gdv'
      %gradient descent with variable learning rate
      [net,tr] = train_gdv(net,P,T);
   case 'train_gdd'
      %gradient descent with direct solution
      [net,tr] = train_gdd(net,P,T);
   case 'train_apolex'
      %APOLEX training
      [net,tr] = train_apolex(net,P,T);
end
```

Discussion

An example of using this function is shown in listing 5.1. *train_siann*, after checking its arguments, calls the appropriate function to train the network. Training stops when the maximum number of epochs is reached or the actual error falls below the specified tolerance (i.e. net.trainParam.goal).

The training record *tr* returned by train_siann is a row vector that stores the actual errors at every epoch during training. It is used to monitor the training progress and can be plotted using MATLAB command: plot(tr).

All training based on gradient descent approach such as train_gd, train_gdv, train_gdm, and train_gdd uses the function *compgrad.* This function, listed in listing A.5 (appendix), computes the gradient of the error function and is one of the key issues in implementing the training methods. It covers the steps from 4 to 10 described in section 4.1.3 (chapter 4).

The *compgrad* function calculates the derivative of a function by using the *feval* function described in section 5.5. As a convention, the derivative of an activation function is found by adding a letter 'd' to the front of the activation function's name. That is if 'tansig' is an activation function, 'dtansig' is its derivative:

```
f = net.layers{k}.transferFcn;      % Activation function of layer k
deriv = feval{['d' f], S{k},Z{k}}; % Compute derivative f'(s{k})
```

5.6.1 Gradient Descent Functions

With *compgrad*, train_gd can be written very neatly as shown in listing 5.4. Slight modifications are needed for other functions such as train_gdv (variable learning rate), train_gdm (with momentum). These functions are listed in the appendix: listing A.8, A.9 respectively.

Listing 5.4: train_gd Gradient Descent Training

```
function [net,tr]=train_gd(net,P,T)
%TRAIN_GD Train SIANN using error backpropagation with gradient descent
%Syntax
%  [net,tr] = train_gd(NET,P,T)
epoch_count = 0;
tr = [];
e = Inf;
while (epoch_count <= net.trainParam.epochs)& (e > net.trainParam.goal)
   epoch_count = epoch_count + 1;
   [dedw, deda, dedb, e ] = compgrad(net, P, T); % Compute gradient
   net = updatenet(net,dedw, deda, dedb);        % Update net
   tr = [tr e];                                  % Record error
   if ~rem(epoch_count, net.trainParam.show)     % Display progress
      fprintf('Epoch %4g: Error = %2.4g\n',epoch_count,e);
   end
end
```

5.6.2 Gradient Descent with Direct Solution Step

The key issue in implementing GDD is to find the least-squared solution of a system of linear equations (see chapter 4, section 3). Again, the decision of implementing this project in MATLAB proves to be very effective. Using MATLAB, the least-squared solution can be found using a single command. First we write the system of linear equations in vector form as follows (**X** is an unknown vector):

$$\mathbf{X} \cdot \mathbf{A} = \mathbf{B}$$

The least-squared solution is a vector $\mathbf{X}^*$ such that $\|\mathbf{X}^*.\mathbf{A} - \mathbf{B}\|$ is minimum. $\mathbf{X}^*$ can be found in MATLAB by simply using / operator:

```
Xstar = B / A;
```

train_gdd is shown in listing A.10 (appendix).

5.6.3 APOLEX training

See chapter 4 section 4 for derivation of this training method, and listing A.11 for the code of *train_apolex* function. Calculating change Δv in net parameters, change ΔE in the global error, correlation factor $c_v(n)$, and probability function $P_v(n)$ is fairly straightforward. The two functions *serialise* and *deserialise* (listing A.12) are written to put all net parameters (weights, biases, decay rates) into a single row vector and vice versa. As shown below, putting all net parameters in a single row vector *s* has the advantage that calculation of Δv, $c_v(n)$, or $P_v(n)$ can be done in one vector manipulation for all parameters:

```
% serialise all net parameters into a single row vector
s = serialise(net.W, net.B, net.A);        %current parameters
sp =  serialise(netp.W, netp.B, netp.A);  %previous parameters

%Compute change in net parameters
delta_s = s - sp;

%Compute covariance between change in net parameters
%and change in the total error
cov = delta_s * (e - ep);

%Compute probability that determines update direction
prob = 1 ./ (1 + exp(2*cov/net.trainParam.T));
```

Another issue to consider is how to generate a constant with a certain probability, i.e. to implement the formula:

$$\Delta_v(n) = \begin{cases} -\delta & \text{with probability of } 1 - P_v(n) \\ \delta & \text{with probability of } P_v(n) \end{cases}$$

This is done using MATLAB function *rand* which generates a pseudo-random numbers with a uniform distribution in the range 0.0 to 1.0. Thus, the probability of the *rand* function to return a random value not greater than a fixed *a* (0<a<1) is equal to *a* itself. This fact is used in the following code (continued from the above code):

```
R = rand(size(s));   %generate random values
value = R < prob;
%value = 1 with probability of prob
%value = 0 with probability of 1 - prob
direction = 2*value - 1;
%direction =  1 with probability of prob
%direction = -1 with probability of 1 - prob

%Compute update
s = s + direction * net.trainParam.delta;
```

Due to its probabilistic nature, APOLEX training may be subject to sudden increases in the global error. This is prevented by a verification step within each epoch, which compares the new error with the previous error. If the new error exceeds the previous error by a fixed factor stored in *net.trainParam.maxIncrease*, the parameter updates are simply discarded. This factor is set by default to 1.2 and can be chosen by the users.

```
%Verification step
Y = sim_siann(nett, P);
et = feval(net.trainParam.errorFcn, Y-T);
%Discard the update if error increases
%by more than a fixed factor
if et < net.trainParam.maxIncrease * e
   net = nett;
end
```

Finally, it should also be mentioned that MATLAB supports object persistence by allowing variables to be saved to and loaded from disk files. The simplest way to do this is by using *save* and *loads* commands. For example,

- To save variables net, P and T to file named my_session.mat,

```
save my_session.mat net P T
```

- To load all variables in the file my_session.mat into workspace,

```
load my_session.mat
```

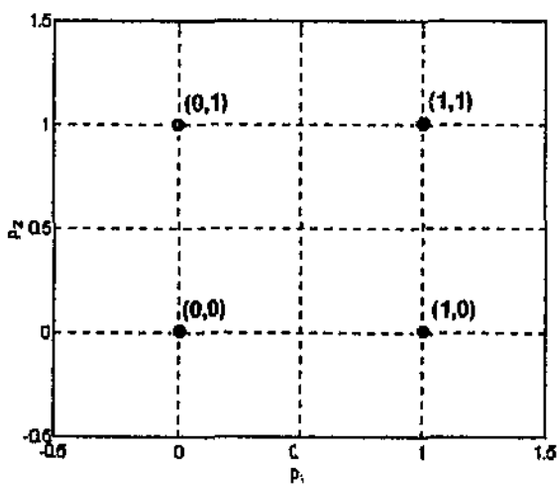
A more sophisticated way is by using MATLAB File I/O functions such as fopen, fclose, fread, fwrite. These functions can handle also non-MAT files and together, they allow greater control over file access.

Chapter 6 Simulation Results

In this chapter, we test the training algorithms derived and implemented in the previous chapters on some benchmarks such as the exclusive-or problem, parity problems, pattern classification and function approximation. We also compare the performance of SIANN's and MLP's by running error-backpropagation training algorithms on these two architectures.

6.1 The XOR Problem

The XOR problem is a classical benchmark for neural network training methods. It was used by Minsky and Papert (1988) to demonstrate the limitation of single-layer perceptrons. The question is to classify two classes of points arranged in an "*exclusive-or*" configuration as shown in figure 6.1.



(a) Two classes of points

Input P		Target T
p_1	p_2	t
0	0	0
0	1	1
1	0	1
1	1	0

(b) Training data

Figure 6.1: XOR Problem

The program in listing B.1 (appendix) trains a SIANN so that it produces the output as shown in figure 6.1b. Training is stopped when the sum squared error falls below 0.1. The program compares the performance of the five training algorithms whose training results are shown in figure 6.2.

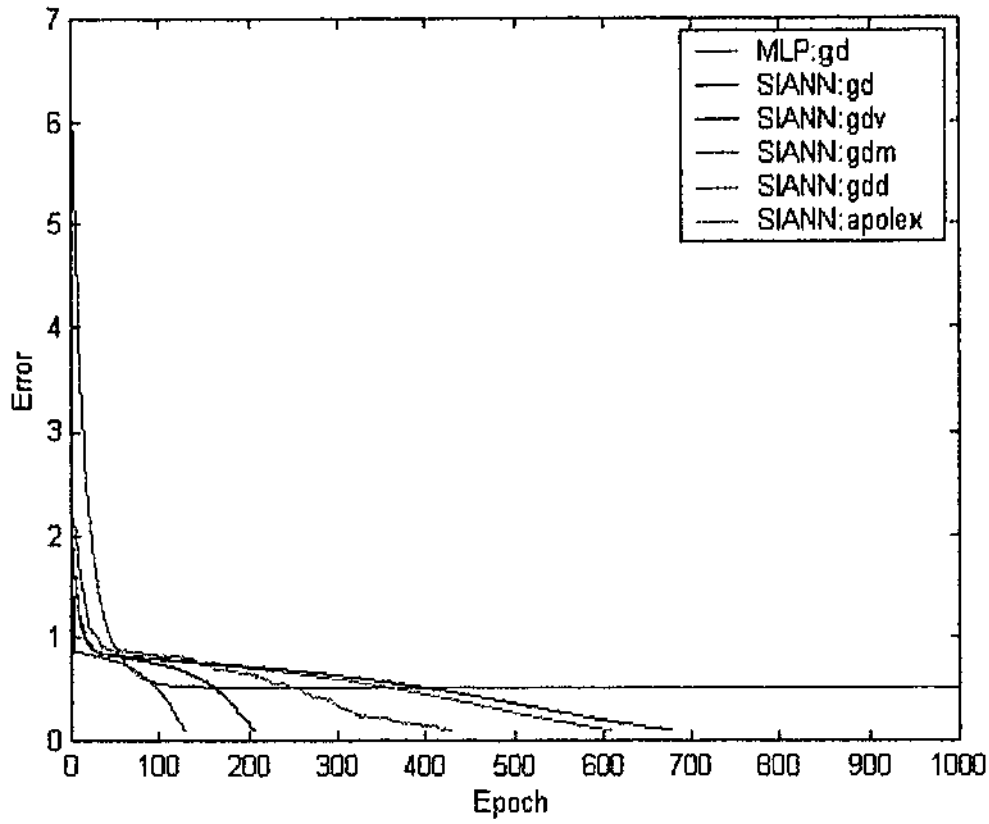


Figure 6.2: Comparisons of training methods: XOR problem
Goal = 0.1, nets with 2 hidden neurons

Of all training methods, the gradient descent with direct solution step (GDD) converges the fastest taking only 5 epochs. The next best result goes to the gradient descent with variable learning rate (GDV) with 220 epochs, followed by the APOLEX algorithm. The gradient descent with momentum (GDM) converges slightly faster than the standard error-backpropagation (GD).

As shown in figure 6.3, the SIANN's trained with the five training methods correctly separated the two classes of points, whereas the MLP did not after 1000 epochs.

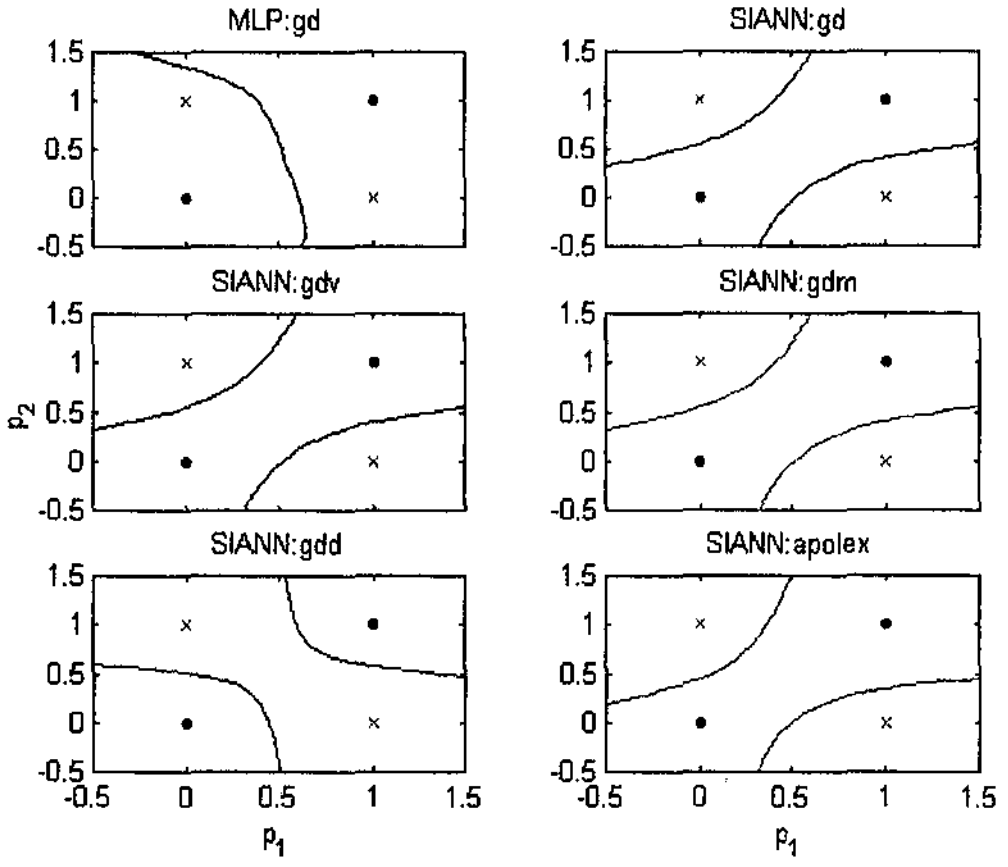


Figure 6.3: Decision boundaries for XOR problem
(Five SIANN's and a MLP)

6.2 Parity Problems

This section compares the performance of SIANN's and MLP's in solving parity problems, which are generalised versions of the exclusive-or problem. We attempt to train neural networks of both architectures to find the even parity bit for input patterns consisting of 3, 4, 5 and 6 bits. Table 3.1 shows the inputs and the expected outputs for the 3-bit even parity problem.

Functions shipped with the MATLAB's Neural Network Toolbox are used to create, initialise, and train MLP's, whereas SIANN's are handled by the functions written in this project. MLP's are trained using Levenberg-Marquardt backpropagation, which is believed currently one of the fastest training algorithms for multilayer perceptrons (Demuth & Beale, 1998). SIANN's are trained using the gradient descent with direct solution step (GDD), which is the fastest in five algorithms developed in this project.

Table 6.1: 3-bit even parity problem

Input P			Target T
p_1	p_2	p_3	t
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

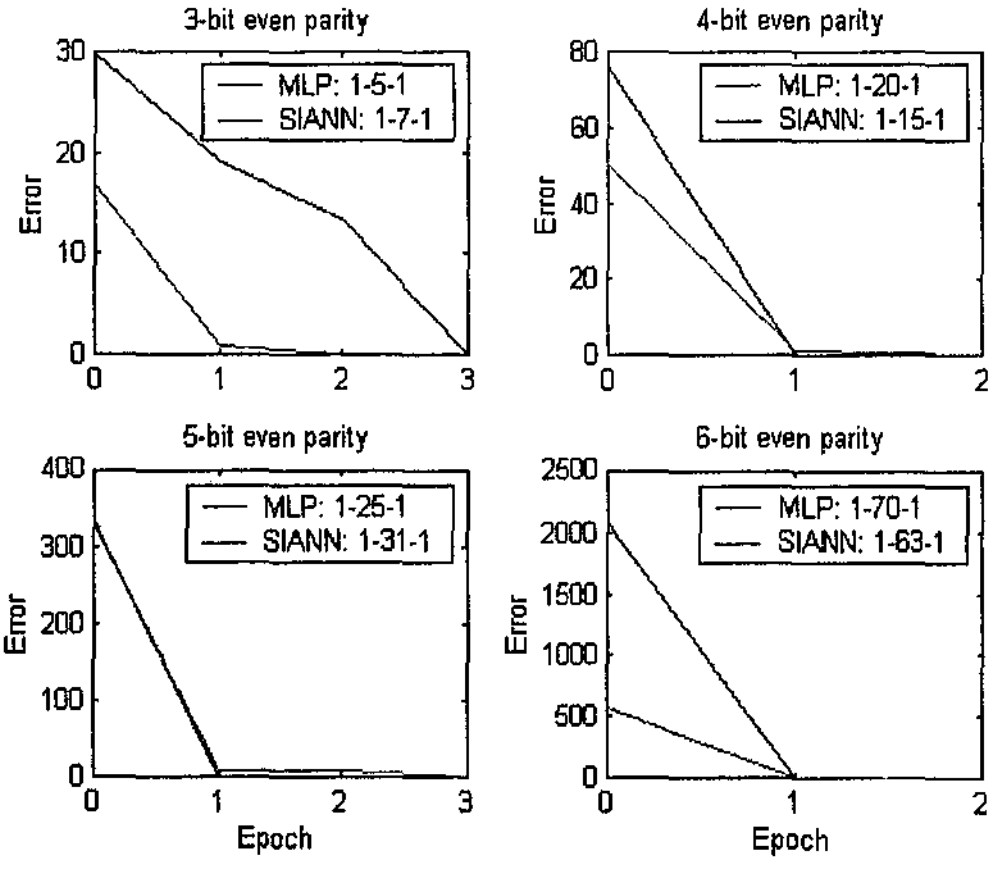


Figure 6.4: MLP versus SIANN: parity problems

Results in Figure 6.4 show that the gradient descent with direct solution step (GDD) for SIANN yields comparable performance with the fast Levenberg-Marquardt algorithm for MLP. A drawback of GDD would be that it requires more neurons in some cases. The code for this testing is shown in listing B.2 (appendix B, file chapter6_2.m).

6.3 Pattern classification

In this section, we develop a SIANN to classify two classes of points (red and blue) as shown in Figure 6.5. The training set is derived by selecting a small number of random points from the two classes. The SIANN has configuration 1-6-1 and the activation functions for the hidden layer and output layer are 'exp' and 'purelin' respectively. The code for this example can be found in listing B.3 in appendix B.

The SIANN is trained with the GDD. Figures 6.6 (a), (b), and (c) show how the decision boundary of the network changed during training. As can be seen, the SIANN classified correctly all 240 points after 24 trials.

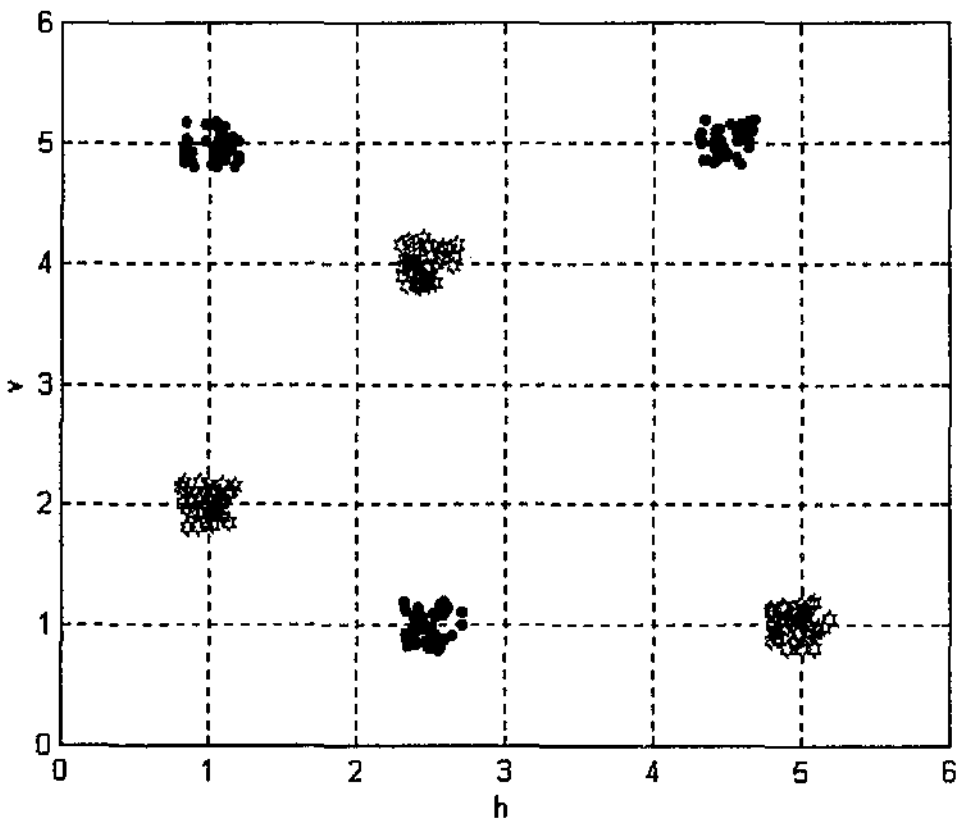
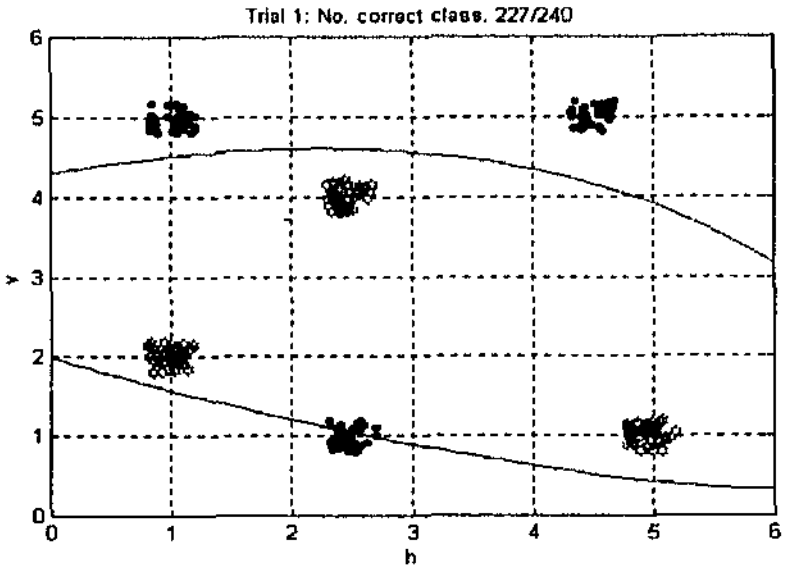
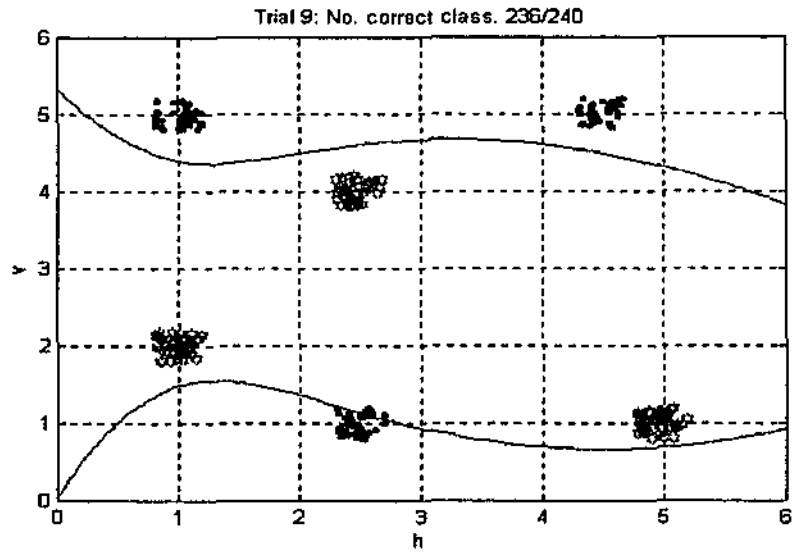


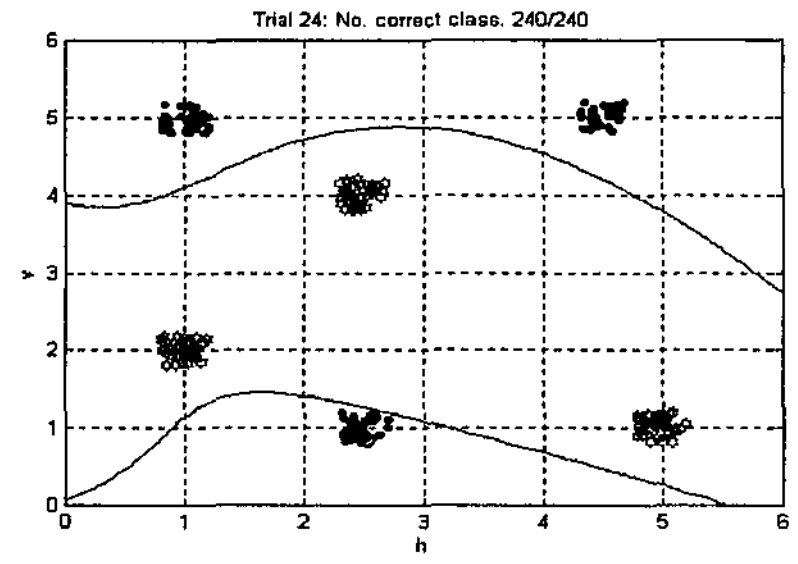
Figure 6.5: Classifying two classes of points (120 red and 120 blue)



(a)



(b)



(c)

Figure 6.6: Decision boundary during training

6.4 Function Approximation

The task of learning a function is a stringent one. In this section, we investigate the interpolative power of SIANN's. Basically, a network is presented with some sampled points of a curve and it is asked to learn the training points and then generate an estimate of the original function. The following function to be approximated is slightly more complex than the benchmark used by Verma (1997) to test his training algorithms for MLP's.

$$f(x) = 0.1 + x.[1.2 + 2.8*\sin(4\pi x^2)]\ ,\ \ 0 < x < 1$$

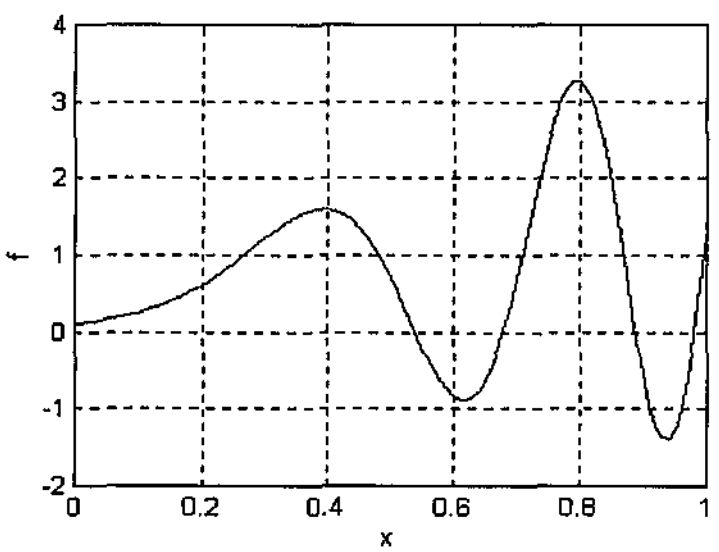


Figure 6.7: Plot of a function to be approximated

The plot of this function is shown in figure 6.7. In the first experiment, training data are taken at regular intervals of 0.1. That is we use 11 training points:

$$\{(0, f(0), (0.1, f(0.1)), (0.2, f(0.2)), \ldots, (1, f(1))\}.$$

In the second experiment, we double the number of training points (taken at regular intervals of 0.05). Finally, we compare the performance of SIANN and MLP if training points were selected randomly in the interval [0, 1].

To test the trained networks, we compute their actual outputs for data points at intervals of 0.01, and then find the mean-squared error between the actual outputs y and the expected values f:

$$E = \frac{1}{101}\sum_{i=0}^{100}(y(0.01i) - f(0.01i))^2$$

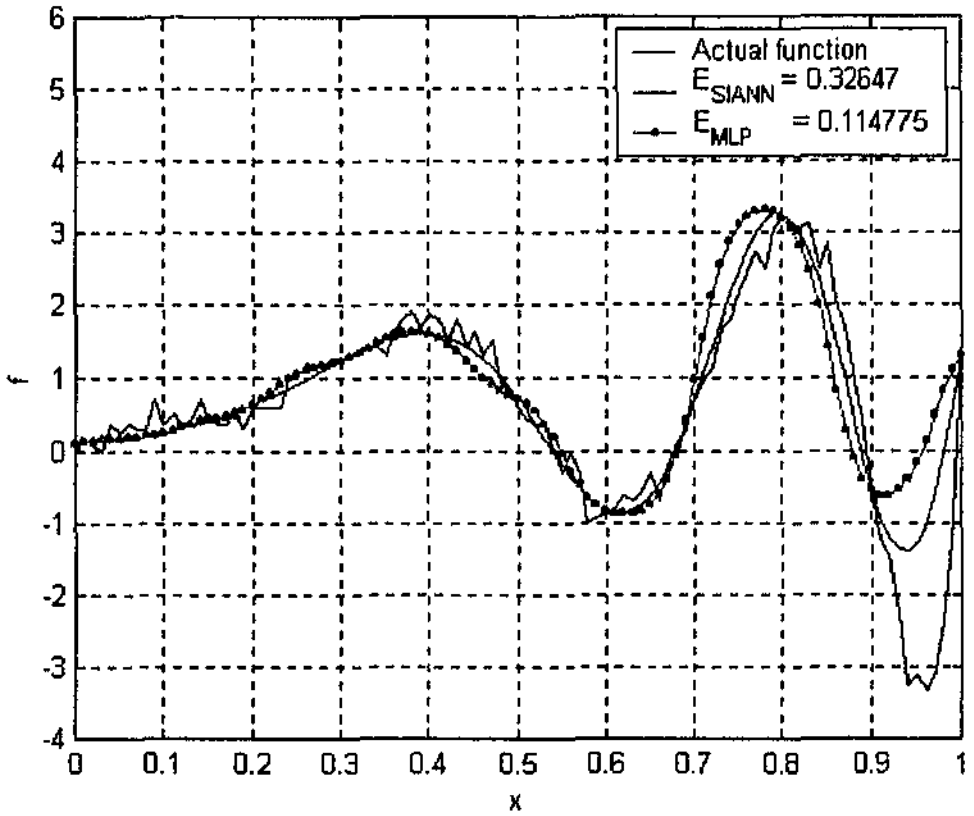


Figure 6.8a: SIANN (1-10-1, GDD) and MLP (1-10-1, trainlm)
11 regularly spaced training points

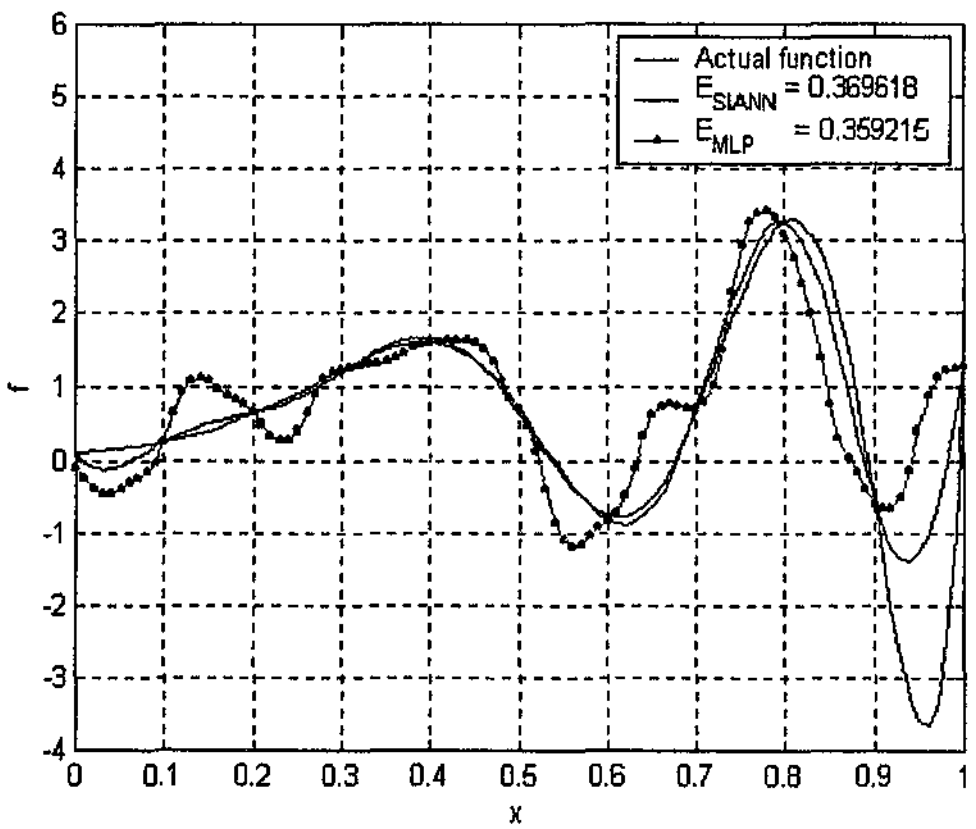


Figure 6.8b: SIANN (1-20-1, GDD) and MLP (1-20-1, trainlm)
21 regularly spaced training points

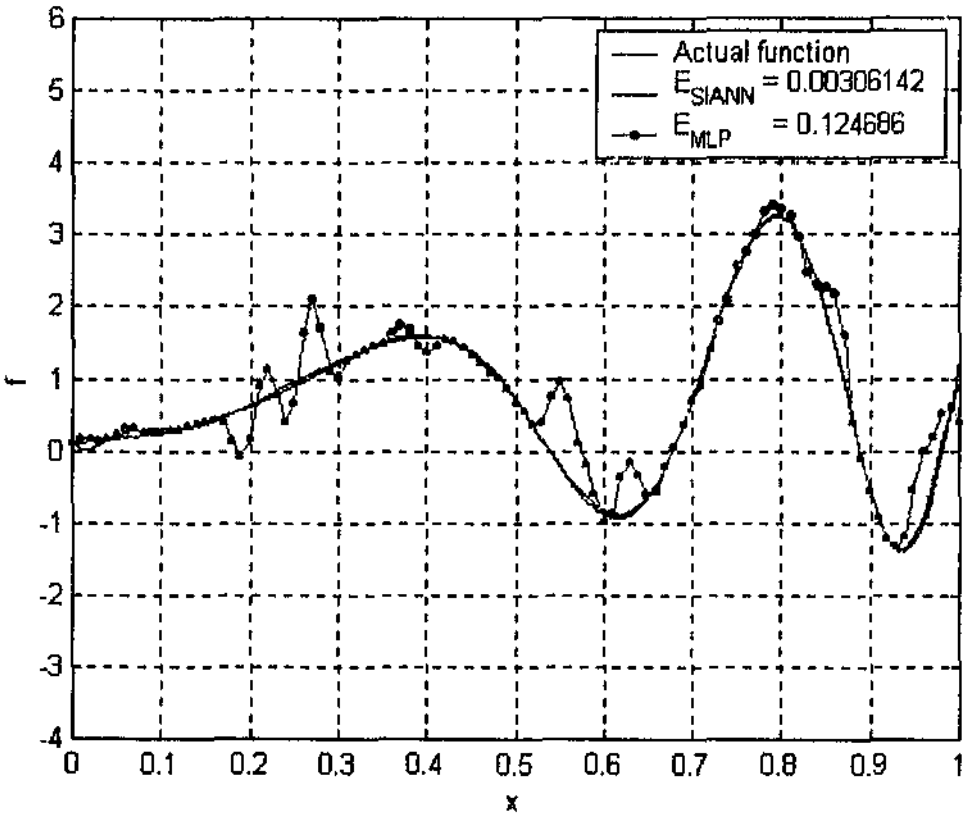


Figure 6.8c: SIANN (1-40-1, GDD) and MLP (1-40-1, trainlm)
40 randomly chosen training points

The results are shown in Figure 6.8 (a), (b) and (c). As can be seen, the SIANN performed poorly in the first experiment where only 10 training points are used. However, in the second experiment where there are twice as many training points as in the first one, or the third experiment where training points are chosen randomly, the SIANN's yielded comparable or even better performance than with MLP's (trained with the fast Levenberg-Marquardt algorithm).

The code for this section is found in appendix B, listing B.4 (file chapter6_4.m).

Chapter 7 Conclusions

This project is part of co-operative efforts of a group of students under the supervision of Associate Professor Salim Bouzerdoum to investigate classes of artificial neural networks that are based on the shunting inhibition model. One student is working on an application of shunting inhibitory cellular neural networks (SICNN's) to edge detection and enhancement. Another student is designing a VLSI chip that realises SICNN's in hardware. The principal aim of this project is to develop efficient training algorithms for a new class of artificial neural networks called Shunting Inhibitory Artificial Neural Networks (SIANN's). The project is motivated by the fact that, despite SICNN's proven potentials in image processing applications such as edge detection and motion detection, the coefficients of a SICNN are still determined by practitioners thorough basically a manual process, which limits SICNN applications.

SIANN's are also based on shunting inhibition model, but unlike the cellular SICNN they have a feed-forward architecture. A SIANN combines shunting inhibitory neurons in the hidden layers and perceptrons in the output layer into a system that is capable of representing very complex mapping between input and output with a smaller number of neurons compared with MLP's. Another key difference between SIANN and SICNN is that the former uses static (steady state) model of shunting inhibitory neurons. These modifications allow us to compute the output of a shunting inhibitory neuron in one step (equation 3.2), and to derive training algorithms for SIANN's based on error-backpropagation. The rest of this chapter summarises what has been achieved in this project and proposes further directions.

7.1 Project Contributions

One of the most significant contributions of this project is the derivation of a training method for SIANN's that is based on error-backpropagation. Although the error-backpropagation training for Multilayer Perceptron is well understood, adapting it to train SIANN's is a complex task. This is because unlike a perceptron, which has linear activation characteristics: $\mathbf{y} = f(\mathbf{w}.\mathbf{p})$, a shunting inhibitory neuron in SIANN, as shown in equation (3.2), has a highly non-linear activation. Adding adaptive parameters such as bias and decay rate to a shunting neuron, while making it more powerful and flexible than a perceptron, certainly increases the complexity of the training algorithm. The error-backpropagation algorithm for SIANN's is summarised in section 4.1.3 (pp. 40-41).

We also investigated three heuristic variations of the error-backpropagation, which are shown to increase convergence speed and training stability in some situation: batch training (GD), gradient descent with variable learning rate (GDV) and gradient descent with momentum (GDM). Batch training updates network parameters only after a group or the entire set of training patterns has been presented by accumulating the amount of changes contributed by each training pair. The method reduces the "*unlearning*" effect whereby minimising the error for a new pattern increases errors for previously learned patterns. GDV speeds up training by adaptively changing the learning rate (i.e. increasing it where the error surface is "flat", and reducing it otherwise). GDM makes training more stable by taking into account the amount of the previous update.

Another significant contribution is a very fast training method for SIANN's that combines error-backpropagation with a direct solution step (GDD). The basic idea is to look for weights that minimise the difference between the actual output and the target by finding the least-squared solution to (often over-determined) systems of linear equations. This method was shown (see Chapter 6) to achieve a comparable performance with the Levenberg-Marquardt algorithm - currently one of the fastest training algorithms for multilayer perceptron. A drawback would be that GDD requires a large number of neurons. The direct solution step was first suggested by Verma (1997).

Also investigated in this project is an interesting training method called APOLEX (algorithm for pattern extraction), which updates network parameters according to a stochastic rule. The major advantages of APOLEX include: it is computationally simple; it is applicable to many network architectures including SIANN; it has few restrictions on the choice of activation functions or network error function.

The second focus of this project is to implement SIANN's and their training algorithms in MATLAB. The MALAB functions developed in this project allow users to create a shunting network, compute its outputs for given inputs, and train the network to produce desired outputs for given input patterns. SIANN's are implemented in a very efficient way using the new data types introduced in MATLAB 5: *structures* and *cell arrays*. The implementation is very flexible in that new training algorithms can be implemented without extensive changes to the existing SIANN architecture. All we need to do is to add new training information fields to the *trainParam* structure, which can be done easily in MATLAB, and to write a MABLAB script for the new training algorithm utilising the training information in *trainParam*.

Together, these SIANN functions can be used to develop pattern classification applications based on shunting inhibitory artificial neural networks. The SIANN functions were applied in chapter 6 to solve XOR and other parity problems, classify 2-D patterns and approximate a function.

7.2 Further Directions

One possible follow-up of this project is to develop neural network applications based on SIANN's and the training methods derived in this project. Two such applications are being considered: detection of fatal heart beats in medicine and automatic detection of modulation types in communication. In this project, SIANNs and their training algorithms are implemented in MATLAB, which is an excellent tool for fast prototyping and development of mathematical models. However, in order to incorporate SIANNs in real applications, the algorithms should be implemented in C/C++, preferably in the form of a class library. This is another step that can be taken to promote further the work in this project.

Bibliography

[1] Beare, R. & Bouzerdoum A. (1999). *Biologically inspired local motion detector architecture*. Journal of Optical Society of America. Vol. 16(9). pp. 2059-2068.

[2] Bouzerdoum, A. (1999). *A New Class of High-Order Neural Networks with Nonlinear Decision Boundaries.* Edith Cowan University, Perth, Western Australia

[3] Bouzerdoum, A. (1991). *Nonlinear Lateral Inhibitory Neural Networks: Analysis and Application to Motion Detection.* Doctoral dissertation. University of Washington.

[4] Bouzerdoum, A., & Pinter, B. (1993). *Shunting Inhibitory Cellular Neural Networks: Derivation and Stability Analysis.* IEEE Transactions on Circuits and Systems. Vol. 40(3). pp. 215-221.

[5] Bouzerdoum, A. & Pinter, R. B. (1989). *Image motion processing in biological and computer vision systems*. Visual Communication and Image Processing. Vol. 1199. pp. 1229-1240.

[6] Chua, L. O. (1993). *The CNN Paradigm.* IEEE Transactions on Circuits and Systems. Vol. 40(3). pp. 215-221.

[7] Demuth, H., & Beale, M. (1998). *Neural Network Toolbox*. The MathWorks, Inc.

[8] Doya, K. (1992). *Bifurcation in the learning of recurrent neural network.* Proceedings of the IEEE International Symposium on Circuits and Systems. Vol. 6, pp. 2777-2780.

[9] Fausett, L. (1994) *Fundamentals of Neural Network-Architecture, Algorithms, and Applications*. New Jersey: Englewood Cliffs, Prentice-Hall, Inc.

[10] Hagan, M.T, & Demuth, H.B, & Beale M. (1996). Neural Network Design. Boston: PWS Publishing Company.

[11] Hanselman, D., & Littlefield, B. (1998). *Mastering MATLAB®5: A Comprehensive Tutorial and Reference*. New Jersey: Prentice-Hall, Inc.

[12] Harth, E., & Pandya, A. S. (1988). *Dynamics of the APOLEX Process: Applications to the Optimisation Problem*. In L. M. Ricciardi (Ed.) Biomathematics and Related Computational Problems pp. 495-471. Amsterdam: Reidel Publishing.

[13] Hubick, K. T. (1992). *Artificial Neural Networks in Australia*. Commonwealth of Australia: Department of Industry, Technology & Commerce.

[14] Kung, S.Y. (1993). *Digital Neural Networks*. New Jersey, Englewood Cliff: Prentice-Hall, Inc.

[15] Minsky, M. L, & Papert, S. A. (1988). *Perceptrons: An Introduction to Computational Geometry*. Expanded Edition (3rd). Massachusetts: The MIT Press.

[16] Nabet, B., & Pinter, R. B. (1991). *Sensory Neural Networks: Lateral Inhibition*. CRC Press, Inc.

[17] Pandy, A. S., & Macy, R.B. (1996). *Pattern Recognition with Neural Networks in C++*. CRC Press.

[18] Patterson, D. W. (1996). *Artificial Neural Networks Theory and Applications*. Singapore: Prentice Hall.

[19] Pinter, R. B. (1985). *Adaptation of spatial modulation transfer function via nonlinear lateral inhibition*. Biological Cybernetics. Vol. 51, pp. 285-291.

[20] Pontecorvo, C. & Bouzerdoum, A. (1995). *Edge Detection Using A Shunting Inhibitory Cellular Neural Network.* Digital Image Computing: Techniques and Applications. pp. 637-642. Brisbane, Queensland.

[21] Pontecorvo, C. & Bouzerdoum, A. (1997). Edge Detector in Multiplicative Noise using the Shunting Inhibitory Cellular Neural Network. Proceedings of the International Conference EANN '97. pp. 281-285. Stockholm, Sweden.

[22] Redfern, D., Campbell, C. (1998). *The MATLAB®5 Handbook.* New York: Springer-Verlag Inc.

[23] Refenes, A. N., & Zaidi, A. (1993). *Managing Exchange Rate Prediction Strategies with Neural Networks.* In M. Taylor & P. Lisboa (Eds.), Techniques and Applications of Neural Networks (pp. 109-116). London: Ellis Horwood Limited.

[24] Reinhardt J., & Muller, B. (1990). *Physics of Neural Networks.* Berlin: Springer-Verlag.

[25] Taylor, M., & Lisboa, P. (1993). *Techniques and Applications of Neural Networks.* London: Ellis Horwood Limited.

[26] Verma, B. (1997). *Fast Training of Multilayer Perceptrons.* IEEE Transactions on Neural Networks. Vol. 8 (6), pp. 1314-1319.

Appendix A SIANN Functions (Chapter 5)

Listing A.1: new_siann.m **Creating a new SIANN**

```
function net = new_siann(pr, s, tf, btf, ef)
%NEW_SIANN Create a feed-forward shunting inhibitory ANN.
%Syntax
%  net = new_siann(pr,[s1 s2...sNl],{tf1 tf2...tfNl}, btf, ef)
%
%Input
%  pr   Rx2 matrix of min and max values for R input elements
%  si   Number of neurons in layer i
%  tfi  Transfer function for layer i
%  btf  Backpropagation training function
%  ef   Error function
%
%Output
%  net  the new SIANN network
%
%Example
%  net = new_siann([0 1; 0 1], [5 2], {'exp' ...
%                  'purelin'},'train_gd','sse');
%
%  Creates a net that receives 2 inputs in range from 0 to 1
%  It has 2 layers, the first with 5 neuron, transfer function exp
%  The second has 2 neuron, transfer function purelin
%  The net uses train_gd as training function
%  It uses sse to compute error
%
%See also SIM_SIANN, INIT_SIANN, TRAIN_SIANN

if nargin < 2
  error('Not enough input arguments')
end
if size(pr,2) ~= 2
  error('1st argument must be a two column matrix.')
end
if any(pr(:,1) > pr(:,2))
  error('1st argument must have min values in 1st column.')
end
if isa(s,'cell') & (prod(size(s)) == length(s))
  s = [s{:}];
end

Nl = length(s);   % Number of layers excluding the input layer
```

```
if nargin < 3
   tf = {'exp'};
   tf = [tf(ones(1,Nl))];
end

if nargin < 4
   btf = 'train_gd';
end

if nargin < 5
   ef = 'sse';
end

% Architecture
net.numInputs = size(pr,1);
net.numOutputs = s(Nl);
net.numLayers = Nl;

for i=1:Nl
  net.layers{i}.size = s(i);        % Size of each layer
  net.layers{i}.transferFcn = tf{i};% Transfer function of each layer
end

%Weights, biases, decay rates
net.W = cell(Nl,1);                    % Weights
net.B = cell(Nl,1);                    % Biases
net.A = cell(Nl-1,1);                  % Decay rates

net.W{1} = zeros(net.layers{1}.size, net.numInputs);
net.B{1} = zeros(net.layers{1}.size, 1);
for i = 2:net.numLayers
   net.W{i} = zeros(net.layers{i}.size,net.layers{i-1}.size);
   net.B{i} = zeros(net.layers{i}.size, 1);
   net.A{i-1} = zeros(net.layers{i-1}.size, 1);
end

net.inputRange = pr;                   % Range of input values

% Training
net.trainParam.goal = 0.01;
net.trainParam.epochs = 50;
net.trainParam.show = 25;
net.trainParam.trainFcn = btf;
net.trainParam.errorFcn = ef;

%Learning rates
net.trainParam.lr_w = 0.01;
```

```
net.trainParam.lr_a = 0.01;
net.trainParam.lr_b = 0.01;

%Initialise net
net = init_siann(net);
```

Listing A.2: init_siann.m **Initialising a SIANN**

```
function net=init_siann(net)
%INIT_SIANN Initialize a Shunting Inhibitory Artificial Neural Network
%
%Syntax
%  net = init_siann(net)
%
%Description
%  returns NET with weights,biases, decay
%  rates initialised to random values.
%
%See also SIM_SIANN, TRAIN_SIANN

%Randomise net weights, biases, and decay rates
net.W{1} = rand(size(net.W{1}));
net.B{1} = rand(size(net.B{1}));
for i = 2:net.numLayers
   net.W{i} = rand(size(net.W{i}));
   net.B{i} = rand(size(net.B{i}));
   net.A{i-1} = rand(size(net.A{i-1}));
end

%Initialise required parameters according
%to the training function of the network
switch net.trainParam.trainFcn
   case 'train_gdm'  %gradient descent with momentum
      net.trainParam.beta = 0.1;

   case 'train_gdv'  %variable learning rate
      net.trainParam.zeta = 0.003;
      net.trainParam.rho = 0.99;

   case 'train_gdd'  %gradient descent with direct solution
      net.trainParam.direct = 2;
      %i.e Direct solution takes place after every 2 epochs
      %Find and store inverse of transfer function
      for i = 1:net.numLayers
         transferInv = 'purelin'; %By default
         switch net.layers{i}.transferFcn
            case 'exp'
               transferInv = 'log';
            case 'purelin'
               transferInv = 'purelin';
            case 'tansig'
               transferInv = 'atanh';
            case 'logsig'
```

```
            transferInv = 'invlogsig';
         case 'sin'
            transferInv = 'asin';
      end
      net.layers{i}.transferInv = transferInv;
   end
case 'train_apolex'
   net.trainParam.delta = 0.01;
   net.trainParam.T = 10;
end
```

Listing A.3: sim_siann.m **Simulating a SIANN**

```
function [Y, Z, D, S] = sim_siann(net,P)
%SIM_SIANN Simulate a Shunting Inhibitory Artificial Neural Network.
%
%Syntax:
%  [Y, Z, D, S] = sim_siann(net, P)
%
%Input:
%  NET - Network
%  P   - Network input
%Output:
%  Y   - Network output
%  Z   - Output of individual layer (optional, used in training)
%  D   - Denominator factor a + f(sum) (optional, used in training)
%  S   - Input sum to each neuron  (optional, used in training)
%
%Example
%  P = [1 3 6 8; 2 4 7 5];
%  net = new_siann([0 3; -1 4], [4 2], {'tansig' 'purelin'});
%  Y = sim_siann(net, P)
%
%See also INIT_SIANN, TRAIN_SIANN, NEW_SIANN

% Check arguments
if nargin ~= 2
  error ('Not enough input arguments.');
end

% Simulate network
N = net.numLayers;    % The number of layers
Ni = size(P,2);       % The number of input vectors in the training set
Z = cell(N + 1,1);    % Storage for N layers' output
D = cell(N-1,1);      % Storage for denominator factor a + f(sum)
S = cell(N,1);        % Storage for sum W*P
Z{1} = P;             % Input set considered as output of layer 0

% Z{k} stores the output of layer k-1
% Compute the output for all layers except the last layer
for k = 1:N-1
   T = net.layers{k}.size - size(Z{k},1);
   if T > 0
      ZT = [Z{k}; zeros(T,Ni)];
   else
      ZT = Z{k}(1:net.layers{k}.size,:);
   end
   S{k} = net.W{k} * Z{k};                        %Weighted sum
```

```
   D{k} = repmat(net.A{k},1,Ni) + ...
         feval(net.layers{k}.transferFcn,S{k}); %Denominator
   Z{k+1} = (ZT + repmat(net.B{k},1,Ni)) ./ D{k};
end

% Computer the output of the last
% layer -       a perceptron layer
S{N} = net.W{N} * Z{N} + repmat(net.B{N}, 1, Ni);
Z{N+1} = feval(net.layers{N}.transferFcn, S{N});
Y = Z{N+1};           %Final output
```

Listing A.4: train_siann.m **Training a SIANN**

```
function [net,tr]=train_siann(net,P,T)
%TRAIN_SIANN Train a Shunting Inhibitory Artificial Neural Network
%Syntax
%  [net,tr] = train_siann(net,P,T)
%Input
%  NET - Network
%  P   - Network inputs
%  T   - Network targets
%Output
%  NET - New network.
%  TR  - Training record
%Example
%  P = [0 0 1 1;
%       1 0 1 0] % input set
%  T = [1 0 0 1]
%  net = new_siann([0 1; 0 1], [2 1], {'purelin'
%                          'purelin'},'train_gdd');
%  net = init_siann(net);
%  [net, tr] = train_siann(net,P,T);
%  Y = sim_siann(net,P);
%Notes
%  Training stops when error is smaller than net.trainParam.goal
%  or the number of epochs exceeds net.trainParam.epochs
%See also  NEW_SIANN, INIT_SIANN, SIM_SIANN, TRAIN_GD
% Check arguments
if nargin ~= 3
  error('Not enough input arguments.');
end
switch net.trainParam.trainFcn
   case 'train_gd'
      %standard gradient descent
      [net,tr] = train_gd(net,P,T);
   case 'train_gdm'
      %gradient descent with momentum
      [net,tr] = train_gdm(net,P,T);
   case 'train_gdv'
      %gradient descent with variable learning rate
      [net,tr] = train_gdv(net,P,T);
   case 'train_gdd'
      %gradient descent with direct solution
      [net,tr] = train_gdd(net,P,T);
   case 'train_apolex'
      %APOLEX training
      [net,tr] = train_apolex(net,P,T);
end
```

Listing A.5: compgrad.m **Computing Gradient of Error Function**

```
function [dedw, deda, dedb, e] = compgrad(net,P,T)
%COMPGRAD Compute gradient of error function
%
Ni = size(P,2);       % Number of training sets
N = net.numLayers;    % Number of layers

es = cell(N,1);       % Error sensitivity for all layers
dedw = cell(N,1);     % Partial derivative of e with respect to weights
deda = cell(N,1); % Partial derivative of e with respect to decay rates
dedb = cell(N,1);     % Partial derivative of e with respect to biases
for k=1:N
   % kth cell is for kth layer
   es{k} = zeros(net.layers{k}.size,Ni);
   dedw{k} = zeros(size(net.W{k}));
   dedb{k} = zeros(size(net.B{k}));
end

for k=1:N-1
   deda{k} = zeros(size(net.A{k}));
end

[Y, Z, D, S] = sim_siann(net, P); % Compute output
es{N} = Y - T;                    % Output layer
e = feval(net.trainParam.errorFcn,es{N}); % Compute error

tf = net.layers{N}.transferFcn;
Te = es{N} .* feval(['d' tf], S{N},Z{N+1});
es{N-1} = net.W{N}' * Te;  %N -1 layer

for k = N-2:-1:1
   tf = net.layers{k+1}.transferFcn;
   for i = 1:net.layers{k}.size
      Te = es{k+1}.* feval(['d' tf],S{k+1},...
            D{k+1} - repmat(net.A{k+1},1,Ni)) ...
            .* (-Z{k+2}./ D{k+1});
      es{k}(i,:) = net.W{k+1}(:,i)' * Te;
      if i <= net.layers{k+1}.size
         es{k}(i,:) = es{k}(i,:) + es{k+1}(i,:)/D{k+1}(i,:);
      end
   end
end

% Compute gradient of e
% weghts w
% first the output layer
```

```
tf = net.layers{N}.transferFcn;
for i = 1:net.layers{N}.size
  for j = 1:net.layers{N-1}.size
     dedw{N}(i,j) = sum(es{N}(i,:) .* ...
                     feval(['d' tf], S{N}(i,:), Z{N+1}(i,:)) ...
                     .* Z{N}(j,:));
  end
end
% now for inner layers
for k = N-1:-1:1
   tf = net.layers{k}.transferFcn;
   for i = 1:size(net.W{k},1)
      for j = 1:size(net.W{k},2)
         dedw{k}(i,j) = sum ( ...
                              es{k}(i,:) ...
                              .* (-Z{k+1}(i,:) ./D{k}(i,:)) ...
                              .* Z{k}(j,:) ...
                              .* feval(['d' tf],S{k}(i,:), ...
                              D{k}(i,:) - net.A{k}(i)));
      end
   end
end

% Biases b
for k = 1:N-1
   dedb{k} = sum(es{k} ./ D{k},2);
end

tf = net.layers{N}.transferFcn;
dedb{N} = sum(es{N}.* feval(['d' tf], S{N}, Z{N+1}),2);

% Decay rates a
for k = 1:N-1
   deda{k} = sum((es{k} .* (-Z{k+1} ./ D{k})),2);
end
```

Listing A.6: updatenet.m **Updating Net Parameters**

```
function net = updatenet(net,dedw, deda, dedb)
%UPDATENET Given gradient of error, update net parameters
for k = 1:net.numLayers
   net.W{k} = net.W{k} - dedw{k} * net.trainParam.lr_w;  % Weights W
   net.B{k} = net.B{k} - dedb{k} * net.trainParam.lr_b;  % Biases B
end
for k = 1:net.numLayers-1 % Decay rates A
   net.A{k} = net.A{k}- deda{k} * net.trainParam.lr_a;
end
```

Listing A.7: train_gd.m **Gradient Descent Algorithm for SIANN**

```
function [net,tr]=train_gd(net,P,T)
%TRAIN_GD Train a SIANN using error backpropagation
%         with gradient descent
%Syntax
%  [net,tr] = train_gd(NET,P,T)

epoch_count = 0;
tr = [];
e = Inf;

while (epoch_count <= net.trainParam.epochs)&(e > net.trainParam.goal)
   epoch_count = epoch_count + 1;
   [dedw, deda, dedb, e ] = compgrad(net, P, T); % Compute gradient
   net = updatenet(net,dedw, deda, dedb);        % Update net
   tr = [tr e];                                  % Record error
   if ~rem(epoch_count, net.trainParam.show)     % Display progress
      fprintf('Epoch %4g: Error = %2.4g\n',epoch_count,e);
   end
end
```

Listing A.8: train_gdv.m Gradient Descent With Variable Learning Rate

```
function [net,tr]=train_gdv(net,P,T)
%TRAIN_GDV Gradient Descent with Variable Learning Rate
%Syntax
%   [net,tr]=train_gdv(net,P,T)
epoch_count = 0;
tr = [];
e = Inf;
e_pre = [];
net_pre = net; % previous net
zeta = net.trainParam.zeta;
rho = net.trainParam.rho;

while (epoch_count <= net.trainParam.epochs)& (e > net.trainParam.goal)
   epoch_count = epoch_count + 1;
   [dedw, deda, dedb, e ] = compgrad(net, P, T); % Compute gradient
   net = updatenet(net, dedw, deda,dedb);        % Update net
   if isempty(e_pre)
      e_pre = e;
   end
   delta_e = e - e_pre;              % The change in error
   if delta_e > zeta * e_pre         % If error increases too much
      %Discard weight change, revert to the previous net
      net = net_pre;
      %Reduce the learning rates
      net.trainParam.lr_a = net.trainParam.lr_a * rho;
      net.trainParam.lr_b = net.trainParam.lr_b * rho;
      net.trainParam.lr_w = net.trainParam.lr_w * rho;
   else
      %Accept weight change
      net_pre =net;  % save the current net
      e_pre = e;     % save previous error
      tr = [tr e];   % record error
      if ~rem(epoch_count, net.trainParam.show)
         fprintf('Epoch %4g: Error = %2.4g\n',epoch_count,e);
      end

      if delta_e < 0
         %Increase learning rates if e decreases
         net.trainParam.lr_a = net.trainParam.lr_a / rho;
         net.trainParam.lr_b = net.trainParam.lr_b / rho;
         net.trainParam.lr_w = net.trainParam.lr_w / rho;
      end
   end
end
```

Listing A.9: train_gdm.m Gradient Descent With Momentum

```
function [net,tr]=train_gdm(net,P,T)
%TRAIN_GDM Gradient Descent with Momentum
%Syntax
%   [net,tr]=train_gdm(net,P,T)
epoch_count = 0;
tr = [];
e = Inf;
delta_w = net.W;  % Change in weights
delta_b = net.B;  % Change in  biases
delta_a = net.A;  % Change in  decay rates

%Initialise previous change to 0
for k=1:net.numLayers
   % kth cell is for kth layer
   delta_w{k} = zeros(size(net.W{k}));
   delta_b{k} = zeros(size(net.B{k}));
end
for k=1:net.numLayers-1
   delta_a{k} = zeros(size(net.A{k}));
end

while (epoch_count <= net.trainParam.epochs)& (e > net.trainParam.goal)
   epoch_count = epoch_count + 1;
   [dedw, deda, dedb, e ] = compgrad(net, P, T); % Compute gradient
   tr = [tr e];                                  % Record error
   if ~rem(epoch_count, net.trainParam.show)     % Display progress
      fprintf('Epoch %4g: Error = %2.4g\n',epoch_count,e);
   end

   % Update net parameters
   for k = 1:net.numLayers
      delta_w{k} = - dedw{k} * net.trainParam.lr_w ...
                   + net.trainParam.beta * delta_w{k};
      net.W{k} = net.W{k} + delta_w{k};  % weights W

      delta_b{k} = - dedb{k} * net.trainParam.lr_b ...
                   + net.trainParam.beta * delta_b{k};
      net.B{k} = net.B{k} + delta_b{k};  % biases B
   end
   for k = 1:net.numLayers-1
      delta_a{k} = - deda{k} * net.trainParam.lr_a ...
                   + net.trainParam.beta * delta_a{k};
      net.A{k} = net.A{k} + delta_a{k};  % decay rates A
   end
end
```

Listing A.10: train_gdd.m **Gradient Descent With Direct Solution Step**

```
function [net,tr]=train_gdd(net,P,T)
%TRAIN_GDD Gradient Descent with Direct Solution Step
warning off          % Suppress warnings
N = net.numLayers;   % Number of layers
Ni = size(P,2);

epoch_count = 0;
tr = [];
e = Inf;
while (epoch_count <= net.trainParam.epochs)& (e > net.trainParam.goal)
   epoch_count = epoch_count + 1;
   [dedw, deda, dedb, e ] = compgrad(net, P, T); % Compute gradient
   tr = [tr e];                                  % Record error
   if ~rem(epoch_count, net.trainParam.show)
      fprintf('Epoch %4g: sum squared error = %2.4g\n',epoch_count,e);
   end
   net = updatenet(net,dedw, deda, dedb);        % Update net

   if ~rem(epoch_count, net.trainParam.direct)
      %Perform direct solution step
      [Y, Z, D, S] = sim_siann(net,P);
      %Solve directly for weights and biases in output layer
      %X * A = B -> X = B/A;
      A = [Z{N};ones(1,Ni)];
      B = feval(net.layers{N}.transferInv,T);
      X = B/A;                        %Find least squared solution
      net.B{N} = X(:,size(X,2));      %Biases in the last column of X
      net.W{N} = X(:,1:size(X,2)-1);

      %Solve directly for weights in shunting layers
      for k = N-1:-1:1
         transferInv = net.layers{k}.transferInv;
         for i = 1:net.layers{k}.size
            if i <= size(Z{k},1)
               G = (Z{k}(i,:)+net.B{k}(i))./Z{k+1}(i,:) - net.A{k}(i);
            else
               G = net.B{k}(i) ./ Z{k+1}(i,:) - net.A{k}(i);
            end
            B = feval(transferInv,G);
            X = B/Z{k};        %Find least squared solution
            net.W{k}(i,:) = X;
         end
      end
   end
end
```

Listing A.11: train_apolex.m **APOLEX algorithm**

```
function [net,tr]=train_apolex(net,P,T)
%TRAIN_GD Train a SIANN using error backpropagation
%         with gradient descent
%Syntax
%  [net,tr] = train_gd(NET,P,T)

epoch_count = 0;
anneal_count = 0;
N = net.numLayers;
tr = [];
e = Inf;    %current error
ep = 0;     %previous error
netp = net;
nett = net; %for verification purpose

while (epoch_count <= net.trainParam.epochs)& (e > net.trainParam.goal)
   epoch_count = epoch_count + 1;
   [Y,Z] = sim_siann(net, P);       % Compute output
   e = feval(net.trainParam.errorFcn,Y-T);
   tr = [tr e];                                  % Record error
   if ~rem(epoch_count, net.trainParam.show)     % Display progress
      fprintf('Epoch %4g: Error = %2.4g\n',epoch_count,e);
   end

   % serialise all net parameters into a single row vector
   s = serialise(net.W, net.B, net.A);       %current parameters
   sp =  serialise(netp.W, netp.B, netp.A);  %previous parameters

   %Compute change in net parameters
   delta_s = s - sp;

   %Compute covariance between change in net parameters
   %and change in the total error
   cov = delta_s * (e - ep);

   netp = net; %Store the current copy
   ep = e;

   %Compute update direction
   prob = 1 ./ (1 + exp(2*cov/net.trainParam.T));
   R = rand(size(s));
   direction = prob > R;          %direction is 0 or 1
   direction = 2*direction - 1;  %direction is -1 or 1;
   %Compute update
   s = s + direction * net.trainParam.delta;
```

```
    %Update net parameters
    [w, b, a] = deserialise(net,s);
    nett.W = w;
    nett.B = b;
    nett.A = a;

    %Verification step
    Y = sim_siann(nett,P);
    et = feval(net.trainParam.errorFcn,Y-T);
    %Discard the update if error increases
    %by more than a fixed factor
    if et < net.trainParam.maxIncrease*e
       net = nett;
    end

    %Determine temperature for next run
    net.trainParam.T = max(abs(cov));
end
```

Listing A.12: Other supporting functions

```
function s = serialise(w, b, a)
%SERIALISE Serialise w, b, a into a single column vector s
%Internal function subject to change
N = size(w,1);
s = [];

%Concatenate w, b, a in that order
for i = 1:N
   s = [s ; reshape(w{i},prod(size(w{i})),1)];
end

for i = 1:N
   s = [s ; reshape(b{i},prod(size(b{i})),1)];
end

for i = 1:N-1
   s = [s ; reshape(a{i},prod(size(a{i})),1)];
end
```

```
function [w, b, a] = deserialise(net,s)
%DESERIALISE Deserialise a single column vector into w, b, a
%Internal function subject to change
%See also SERIALISE

N = net.numLayers;
w = net.W;
b = net.B;
a = net.A;

k = 1;
for i = 1:N
   w{i} = reshape(s(k : k + prod(size(w{i}))-1), size(w{i}));
   k = k + prod(size(w{i}));
end

for i = 1:N
   b{i} = reshape(s(k : k + prod(size(b{i}))-1), size(b{i}));
   k = k + prod(size(b{i}));
end

for i = 1:N-1
   a{i} = reshape(s(k : k + prod(size(a{i}))-1), size(a{i}));
   k = k + prod(size(a{i}));
end
```

```
function Y = dexp(X,D)
%Derivative of y = exp(x)
Y = exp(X);
```

Appendix B Programs to test SIANN Functions (Chapter 6)

Listing B.1: chapter6_1.m **XOR problem**

```
%Train SIANN to recognise XOR pattern
%Compare performance of five training algorithms for SIANN
%Compare SIANN and MLP
clear
P = [0 0 1 1;
     1 0 1 0] % input set
T = [1 0 0 1] % target

%Create a SIANN with two shunting neurons
%Change train_gd to any of the following training functions
%train_gdd, train_gdv, train_gdm, train_apolex, train_gd
net_begin = new_siann([0 1; 0 1], [2 1], {'exp' 'purelin'},'train_gd');

%Initialise network weights, biases, decay rates
%All SIANN's have the same weights, biases, decay rates initially
net_begin = init_siann(net_begin);
net_begin.trainParam.epochs = 1000;
net_begin.trainParam.goal = 0.1;

%SIANN to be trained with GDM
net_gdm = net_begin;
net_gdm.trainParam.trainFcn = 'train_gdm';
net_gdm = init_siann(net_gdm);
net_gdm.W = net_begin.W;
net_gdm.B = net_begin.B;
net_gdm.A = net_begin.A;

%SIANN to be trained with GDV
net_gdv = net_begin;
net_gdv.trainParam.trainFcn = 'train_gdv';
net_gdv = init_siann(net_gdv);
net_gdv.W = net_begin.W;
net_gdv.B = net_begin.B;
net_gdv.A = net_begin.A;

%SIANN to be trained with GDD
net_gdd = net_begin;
net_gdd.trainParam.trainFcn = 'train_gdd';
net_gdd = init_siann(net_gdd);
net_gdd.W = net_begin.W;
net_gdd.B = net_begin.B;
net_gdd.A = net_begin.A;
```

```
%SIANN to be trained with APOLEX
net_apolex = net_begin;
net_apolex.trainParam.trainFcn = 'train_apolex';
net_apolex = init_siann(net_apolex);
net_apolex.W = net_begin.W;
net_apolex.B = net_begin.B;
net_apolex.A = net_begin.A;

%MLP to be trained using error-backpropagation
%NOTE: For MLP we use functions included
%in the MATLAB's Neural Network Toolbox
net_mlp = newff([0 1; 0 1], [2 1], {'tansig' 'purelin'}, ...
                 'traingd','learngd','sse');
net_mlp.trainParam.epochs = 1000;
net_mlp.trainParam.goal = 0.1;
net_mlp_begin = net_mlp;                %Save the initial network

%TRAIN SIANN's
%Training with gradient descent
fprintf('\nTrain SIANN using gradient descent...\n');
net_gd = net_begin;
t1 = clock;
[net_gd, tr_gd] = train_siann(net_gd,P,T);
t2 = clock;
t_gd = etime(t2,t1);          %Compute training time
Y_gd = sim_siann(net_gd,P);   %Compute output

%Train SIANN usinggradient descent with momentum
fprintf('\nTrain SIANN using gradient descent & momentum...\n');
t1 = clock;
[net_gdm, tr_gdm] = train_siann(net_gdm,P,T);   %train SIANN
t2 = clock;
t_gdm = etime(t2,t1);         %Compute training time
Y_gdm = sim_siann(net_gdm,P); %Compute output

%Train SIANN using gradient descent & variable learning rate
fprintf('Train SIANN using gradient descent&variable learning rate\n');
t1 = clock;
[net_gdv, tr_gdv] = train_siann(net_gdv,P,T); %train SIANN
t2 = clock;
t_gdv = etime(t2,t1);         %Compute training time
Y_gdv = sim_siann(net_gdv,P); %Compute output

%Train SIANN using gradient descent & direct solution step
fprintf('\nTrain using gradient descent & direct solution step\n');
t1 = clock;
```

```
[net_gdd, tr_gdd] = train_siann(net_gdd,P,T);%train SIANN
t2 = clock;
t_gdd = etime(t2,t1);          %Compute training time
Y_gdd = sim_siann(net_gdd,P); %Compute output

%Train SIANN using the APOLEX algorithm
fprintf('\nTrain using APOLEX algorithm...\n');
t1 = clock;
%train SIANN
[net_apolex, tr_apolex] = train_siann(net_apolex,P,T);
t2 = clock;
t_apolex = etime(t2,t1);             %Compute training time
Y_apolex = sim_siann(net_apolex,P); %Compute output

%Train MLP using error-backpropagation
fprintf('\nTrain MLP using error-backpropagation...\n');
t1 = clock;
figure(3)
[net_mlp, tr_mlp] = train(net_mlp, P,T);%train MLP
t2 = clock;
t_mlp = etime(t2,t1);                %Compute training time
Y_mlp = sim(net_mlp,P);              %Compute output

%CHECK CLASSIFICATION RESULT
%Compute the decision boundaries for the five SIANN's and the MLP
[p1, p2] = meshgrid(-.5:0.1:1.5, -.5:0.1:1.5); %Input space
i1 = reshape(p1,1,prod(size(p1)));
i2 = reshape(p2,1,prod(size(p2)));

%Output of SIANN trained with GD
y_gd = sim_siann(net_gd,[i1;i2]);
z_gd =reshape(y_gd,size(p1));

%Output of SIANN trained with GDV
y_gdv = sim_siann(net_gdv,[i1;i2]);
z_gdv =reshape(y_gdv,size(p1));

%Output of SIANN trained with GDM
y_gdm = sim_siann(net_gdm,[i1;i2]);
z_gdm =reshape(y_gdm,size(p1));

%Output of SIANN trained with GDD
y_gdd = sim_siann(net_gdd,[i1;i2]);
z_gdd =reshape(y_gdd,size(p1));

%Output of SIANN trained with APOLEX
y_apolex = sim_siann(net_apolex,[i1;i2]);
```

```
z_apolex =reshape(y_apolex,size(p1));

%Output of MLP trained with GD
y_mlp = sim(net_mlp,[i1;i2]);
z_mlp =reshape(y_mlp,size(p1));

%Draw decision boundaries
figure(1)
subplot(3,2,1)     %MLP
plot(0,0,'.b',1,1,'.b',0,1,'xr',1,0,'xr'), hold on
axis([-.5 1.5 -.5 1.5])
set(gca,'XTickLabel',[]);
title('MLP:gd')
contour(p1,p2,z_mlp, [.5 .5],'k'),hold off

subplot(3,2,2)     %SIANN with GD
plot(0,0,'.b',1,1,'.b',0,1,'xr',1,0,'xr'),hold on
contour(p1,p2,z_gd, [.5 .5],'b');
axis([-.5 1.5 -.5 1.5])
set(gca,'XTickLabel',[]);
title('SIANN:gd');

subplot(3,2,3)     %SIANN with GDV
plot(0,0,'.b',1,1,'.b',0,1,'xr',1,0,'xr'),hold on
contour(p1,p2,z_gdv, [.5 .5],'k');
axis([-.5 1.5 -.5 1.5])
set(gca,'XTickLabel',[]);
ylabel('p_2'),title('SIANN:gdv');

subplot(3,2,4)     %SIANN with GDM
plot(0,0,'.b',1,1,'.b',0,1,'xr',1,0,'xr'),hold on
contour(p1,p2,z_gdm, [.5 .5],'m');
axis([-.5 1.5 -.5 1.5])
set(gca,'XTickLabel',[]);
title('SIANN:gdm');

subplot(3,2,5)     %SIANN with GDD
plot(0,0,'.b',1,1,'.b',0,1,'xr',1,0,'xr'),hold on
contour(p1,p2,z_gdd, [.5 .5],'g');
axis([-.5 1.5 -.5 1.5])
xlabel('p_1'), title('SIANN:gdd');

subplot(3,2,6)     %SIANN with APOLEX
plot(0,0,'.b',1,1,'.b',0,1,'xr',1,0,'xr'), hold on
contour(p1,p2,z_apolex, [.5 .5],'r');
axis([-.5 1.5 -.5 1.5])
xlabel('p_1'), title('SIANN:apolex')
```

```
%Compare training methods for SIANN's and MLP
%Plot the error versus epoch
figure(2)
plot(0:length(tr_mlp.perf)-1,tr_mlp.perf,'c', ...
     0:length(tr_gd)-1,tr_gd,'b', ...
     0:length(tr_gdv)-1,tr_gdv,'k', ...
     0:length(tr_gdm)-1,tr_gdm,'m', ...
     0:length(tr_gdd)-1,tr_gdd,'g', ...
     0:length(tr_apolex)-1,tr_apolex,'r');
legend('MLP:gd', 'SIANN:gd', 'SIANN:gdv', 'SIANN:gdm','SIANN:gdd',
'SIANN:apolex');
xlabel('Epoch'), ylabel('Error')
title('Error for XOR problem - Goal=0.1')

%Save results of the training session
%save chapter6_1.mat
```

Listing B.2: chapter6_2.m **Parity Problems**

```
%Compare SIANN and MLP in even-parity finding problems
clear all

%Training data
P3 = binwords(3);        % input set 3-bit
T3 = rem(sum(P3,1),2);   % even parity 3-bit

P4 = binwords(4);        % input set 4-bit
T4 = rem(sum(P4,1),2);   % even parity 4-bit

P5 = binwords(5);        % input set 5-bit
T5 = rem(sum(P5,1),2);   % even parity 5-bit

P6 = binwords(6);        % input set 6-bit
T6 = rem(sum(P6,1),2);   % even parity 6-bit


%Train SIANN to find 3-bit parity
fprintf('\nTrain SIANN to solve 3-bit parity...\n');
net_siann3 = new_siann([zeros(3,1) ones(3,1)], [7 1],...
                       {'exp' 'purelin'},'train_gdd');
net_siann3 = init_siann(net_siann3);
net_siann3.trainParam.show = 1;
net_siann3.trainParam.epochs = 1000;
net_siann3.trainParam.goal = .001;
net_siann3_begin = net_siann3;
[netnet_siann3, tr_siann3] = train_siann(net_siann3,P3,T3);

%Train SIANN to find 4-bit parity
fprintf('\nTrain SIANN to solve 4-bit parity...\n');
net_siann4 = new_siann([zeros(4,1) ones(4,1)], [15 1],...
                       {'exp' 'purelin'},'train_gdd');
net_siann4 = init_siann(net_siann4);
net_siann4.trainParam.show = 1;
net_siann4.trainParam.epochs = 1000;
net_siann4.trainParam.goal = .1;
net_siann4.trainParam.direct = 1;
net_siann4_begin = net_siann4;
[net_siann4, tr_siann4] = train_siann(net_siann4,P4,T4);

%Train SIANN to find 5-bit parity
fprintf('\nTrain SIANN to solve 5-bit parity...\n');
net_siann5 = new_siann([zeros(5,1) ones(5,1)], [31 1], ...
                       {'exp' 'purelin'},'train_gdd');
net_siann5 = init_siann(net_siann5);
```

```
net_siann5.trainParam.show = 1;
net_siann5.trainParam.epochs = 1000;
net_siann5.trainParam.goal = .1;
net_siann5.trainParam.direct = 1;
net_siann5_begin = net_siann5;
[net_siann5, tr_siann5] = train_siann(net_siann5,P5,T5);

%Train SIANN to find 6-bit parity
fprintf('\nTrain SIANN to solve 6-bit parity...\n');
net_siann6 = new_siann([zeros(6,1) ones(6,1)], [63 1],...
                       {'exp' 'purelin'},'train_gdd');
net_siann6 = init_siann(net_siann6);
net_siann6.trainParam.show = 1;
net_siann6.trainParam.epochs = 1000;
net_siann6.trainParam.goal = .1;
net_siann6.trainParam.direct = 1;
net_siann6_begin = net_siann6;
[net_siann6, tr_siann6] = train_siann(net_siann6,P6,T6);

%Train MLP to find 3-bit parity
fprintf('\nTrain MLP to solve 3-bit parity...\n');
net_mlp3 = newff([zeros(3,1) ones(3,1)], [5 1], ...
                 {'tansig' 'purelin'}, 'trainlm', '', 'sse');
net_mlp3 = init(net_mlp3);
net_mlp3.trainParam.show = 1;
net_mlp3.trainParam.epochs = 1000;
net_mlp3.trainParam.goal = .1;
net_mlp3_begin = net_mlp3;
[net_mlp3, tr_mlp3] = train(net_mlp3,P3,T3);

%Train MLP to find 4-bit parity
fprintf('\nTrain MLP to solve 4-bit parity...\n');
net_mlp4 = newff([zeros(4,1) ones(4,1)], [20 1], ...
                 {'tansig' 'purelin'}, 'trainlm','', 'sse');
net_mlp4 = init(net_mlp4);
net_mlp4.trainParam.show = 1;
net_mlp4.trainParam.epochs = 1000;
net_mlp4.trainParam.goal = .1;
net_mlp4_begin = net_mlp4;
[net_mlp4, tr_mlp4] = train(net_mlp4,P4,T4);

%Train MLP to find 5-bit parity
fprintf('\nTrain MLP to solve 5-bit parity...\n');
net_mlp5 = newff([zeros(5,1) ones(5,1)], [25 1], ...
           {'tansig' 'purelin'}, 'trainlm','', 'sse');
net_mlp5 = init(net_mlp5);
net_mlp5.trainParam.show = 1;
```

```
net_mlp5.trainParam.epochs = 1000;
net_mlp5.trainParam.goal = .1;
net_mlp5_begin = net_mlp5;
[net_mlp5, tr_mlp5] = train(net_mlp5,P5,T5);

%Train MLP to find 6-bit parity
fprintf('\nTrain MLP to solve 6-bit parity...\n');
net_mlp6 = newff([zeros(6,1) ones(6,1)], [70 1], ...
                 {'tansig' 'purelin'}, 'trainlm','', 'sse');
net_mlp6 = init(net_mlp6);
net_mlp6.trainParam.show = 1;
net_mlp6.trainParam.epochs = 1000;
net_mlp6.trainParam.goal = .1;
net_mlp6_begin = net_mlp6;
[net_mlp6, tr_mlp6] = train(net_mlp6,P6,T6);

%Plot the error versus epoch
subplot(2,2,1)
plot(0:length(tr_mlp3.perf)-1,tr_mlp3.perf,'r', ...
      0:length(tr_siann3)-1,tr_siann3,'b')
set(gca,'XTick',[0:1:max(length(tr_mlp3.perf),length(tr_siann3))]);
title('3-bit even parity')
ylabel('Error'),legend('MLP: 1-5-1', 'SIANN: 1-7-1')

subplot(2,2,2)
plot(0:length(tr_mlp4.perf)-1,tr_mlp4.perf,'r', ...
   0:length(tr_siann4)-1,tr_siann4,'b')
set(gca,'XTick',[0:1:max(length(tr_mlp4.perf),length(tr_siann4))]);
title('4-bit even parity')
ylabel('Error'),legend('MLP: 1-20-1', 'SIANN: 1-15-1');

subplot(2,2,3)
plot(0:length(tr_mlp5.perf)-1,tr_mlp5.perf,'r', ...
     0:length(tr_siann5)-1,tr_siann5,'b')
set(gca,'XTick',[0:1:max(length(tr_mlp5.perf),length(tr_siann5))]);
title('5-bit even parity')
xlabel('Epoch'), ylabel('Error')
legend('MLP: 1-25-1', 'SIANN: 1-31-1');

subplot(2,2,4)
plot(0:length(tr_mlp6.perf)-1,tr_mlp6.perf,'r', ...
     0:length(tr_siann6)-1,tr_siann6,'b')
set(gca,'XTick',[0:1:max(length(tr_mlp6.perf),length(tr_siann6))]);
title('6-bit even parity')
xlabel('Epoch'), ylabel('Error')
legend('MLP: 1-70-1', 'SIANN: 1-63-1');
%save chapter6_2          %Save the result of this training session
```

```
%Train a SIANN to classify two classes
%of points on 2-dimensional space

%Class 0 consists of (h1,v1), (h3,v3), (h5,v5)
%h stands for horizontal
%v stands for vertical
clear
h1 = 1 + 0.4* rand(1,40) - 0.2;
v1 = 5 + 0.4 * rand(1,40) - 0.2;

h3 = 2.5 + 0.4* rand(1,40) - 0.2;
v3 = 1   + 0.4 * rand(1,40) - 0.2;

h5 = 4.5 + 0.4* rand(1,40) - 0.2;
v5 = 5   + 0.4 * rand(1,40) - 0.2;

%Class 1 consists of (h2,v2), (h4,v4), (h6,v6)
h2 = 1 + 0.4* rand(1,40) - 0.2;
v2 = 2 + 0.4 * rand(1,40) - 0.2;

h6 = 5 + 0.4* rand(1,40) - 0.2;
v6 = 1 + 0.4 * rand(1,40) - 0.2;

h4 = 2.5 + 0.4* rand(1,40) - 0.2;
v4 = 4   + 0.4 * rand(1,40) - 0.2;

C1_h = [h1 h3 h5]; %Class 0
C1_v = [v1 v3 v5];

C2_h = [h2 h4 h6]; %Class 1
C2_v = [v2 v4 v6];

Ch = [C1_h C2_h];
Cv = [C1_v C2_v];
C  = [zeros(size(C1_h)) ones(size(C2_h))]; %Correct class id
NumPatterns = length(C);

%Visualise points in class 0 and 1
plot(C1_h,C1_v,'.b',C2_h,C2_v,'hr');
title('Training a SIANN to classify these points');
axis([0 6 0 6]), xlabel('h'), ylabel('v')
waitforbuttonpress

net_cols = cell(100,1); %Collection of nets
res_cols = [];          %Collection of results

%First, create a SIANN 1-6-1
net = new_siann([-10 10; -10 10], [6 1], {'exp'
'purelin'},'train_gdd');
net = init_siann(net);
net.trainParam.epochs = 10;
net.trainParam.show = 100;
net.trainParam.goal = 0.1;
```

```
net.trainParam.direct = 1;

%Second, prepare training data
%bv selecting 3 random points from each classes
i = 1 + floor(39 * rand(6,1));
%Inputs
P = [h1(i(1)) h2(i(2)) h3(i(3)) h4(i(4)) h5(i(5)) h6(i(6)); ...
     v1(i(1)) v2(i(2)) v3(i(3)) v4(i(4)) v5(i(5)) v6(i(6))];
%Targets
T = [0        1        0        1        0        1        ];

%Third, train the net to recognise training set
%During training, display the decision boundary
%so that we can see how SIANN 'learns' to classify
%All points in the 2-D space
[h, v] = meshgrid(0:0.1:6, 0:0.1:6);
h1 = reshape(h,1,prod(size(h)));
v1 = reshape(v,1,prod(size(v)));

NumCor = 0;
NumTrial = 0;
while NumCor < NumPatterns
   NumTrial = NumTrial + 1; % Number of trials
   %Randomise net weights, biases, and decay rates
   net.W{1} = rand(size(net.W{1}));
   net.B{1} = rand(size(net.B{1}));
   for i = 2:net.numLayers
      net.W{i} = rand(size(net.W{i}));
      net.B{i} = rand(size(net.B{i}));
      net.A{i-1} = rand(size(net.A{i-1}));
   end

   %Train the net
   net = train_siann(net,P,T);

   %Compute the output of SIANN for all points in 2-D space
   y = sim_siann(net,[h1;v1]);
   y = reshape(y,size(h));

   %Visualise points in class 0 and 1
   plot(C1_h,C1_v,'.b',C2_h,C2_v,'hr');
   title('Training a SIANN to classify these points');
   axis([0 6 0 6]), hold on;

   %Compute the output of SIANN for the two classes of points
   Output = sim_siann(net,[Ch;Cv]);
   Output = hardlim(Output - 0.5); %Threshold output
   NumCor = sum(Output == C);      %Number of correct classification

   s = sprintf('Trial %g: No. correct class. %g/%g', NumTrial, NumCor,
NumPatterns);

   %Draw the decision boundary of the trained net
   %The output threshold is .5, that is if the output  <  .5
   %then the point is in class 0, otherwise it is in class 1
   contour(h,v,y, [0.5 0.5],'b');
   grid, xlabel('h'), ylabel('v'), title(s), hold off
```

```
   net_cols(NumTrial) = net; %Save the for later analysis
   res_cols = [res_cols NumCor];
   pause(1)                  %Pause for 1 second
   %waitforbuttonpress       %Or wait for user to press a key
end

%Save the results of this session
%save chapter6_3
```

Listing B.4: chapter6_4.m **Function Approximation Example**

```
%Train SIANN for function approximation
%Basically, a network is presented with some sampled points
%of a curve. It is asked to learn the training points
%and then generate an estimate of the original function
clear all

%Function to be approximated
x = 0:0.01:1;
f = 0.1 + 1.2*x + 2.8 * x .* sin(4*pi* x .* x);

%First training set is taken at 11 regular points x = 0, 0.1, ..., 1
P1 = x(1:10:101);
T1 = f(1:10:101);
%Second training set is taken at 21 regular points x = 0, 0.05, ..., 1
P2 = x(1:10:101);
T2 = f(1:10:101);
%Third training set is taken randomly (40 points)
Ind = floor(1 + (length(x) -1) * rand(1,40)); %Random indices to x
P3 = x(Ind);
T3 = f(Ind);


%Train SIANN using the first training set
fprintf('\nTraining SIANN with 11 regularly spaced data points...\n');
net_siann1 = new_siann([0 1], [10 1], {'exp' 'purelin'},'train_gdd');
net_siann1 = init_siann(net_siann1);
net_siann1.trainParam.show = 1;
net_siann1.trainParam.epochs = 20;
net_siann1.trainParam.goal = .1;
net_siann1.trainParam.direct = 1;
net_siann1_begin = net_siann1;
[net_siann1, tr_siann1] = train_siann(net_siann1,P1,T1);

%Train MLP using the first training set
fprintf('\nTrain MLP with 11 regularly spaced data points...\n');
net_mlp1 = newff([0 1], [10 1], {'tansig' 'purelin'},'trainlm', '',
'sse');
net_mlp1 = init(net_mlp1);
net_mlp1.trainParam.show = 1;
net_mlp1.trainParam.epochs = 20;
net_mlp1.trainParam.goal = .1;
net_mlp1_begin = net_mlp1;
figure(1)
[net_mlp1, tr_mlp1] = train(net_mlp1,P1,T1);
```

```
%Check result for the first training set
f_siann1 = sim_siann(net_siann1,x);
f_mlp1   = sim(net_mlp1,x);

error_siann1 = mse(f_siann1 - f);
error_mlp1   = mse(f_mlp1   - f);
s_siann = ['E_S_I_A_N_N = ' sprintf('%g',error_siann1)];
s_mlp =   ['E_M_L_P     = ' sprintf('%g',error_mlp1)  ];

plot(x,f,'b',x,f_siann1,'k',x,f_mlp1,'.-r');
title('11 regular training points');
legend('Actual function', s_siann, s_mlp);
axis([0 1 -4 6]), xlabel('x'), ylabel('f'),grid


%Train SIANN using the second training set
fprintf('\nTraining SIANN with 21 regularly spaced data points...\n');
net_siann2 = new_siann([0 1], [20 1], {'exp' 'purelin'},'train_gdd');
net_siann2 = init_siann(net_siann2);
net_siann2.trainParam.show = 1;
net_siann2.trainParam.epochs = 20;
net_siann2.trainParam.goal = .1;
net_siann2.trainParam.direct = 1;
net_siann2_begin = net_siann2;
[net_siann2, tr_siann2] = train_siann(net_siann2,P2,T2);

%Train MLP using the second training set
fprintf('\nTrain MLP with 21 regularly spaced data points...\n');
net_mlp2 = newff([0 1], [20 1], {'tansig' 'purelin'},'trainlm', '',
'sse');
net_mlp2 = init(net_mlp2);
net_mlp2.trainParam.show = 1;
net_mlp2.trainParam.epochs = 20;
net_mlp2.trainParam.goal = .1;
net_mlp2_begin = net_mlp2;
figure(2)
[net_mlp2, tr_mlp2] = train(net_mlp2,P2,T2);

%Check result for the second training set
f_siann2 = sim_siann(net_siann2,x);
f_mlp2   = sim(net_mlp2,x);

error_siann2 = mse(f_siann2 - f);
error_mlp2   = mse(f_mlp2   - f);
s_siann = ['E_S_I_A_N_N = ' sprintf('%g',error_siann2)];
s_mlp =   ['E_M_L_P     = ' sprintf('%g',error_mlp2)  ];
```

```
plot(x,f,'b',x,f_siann2,'k',x,f_mlp2,'.-r');
title('21 regular training points');
legend('Actual function', s_siann, s_mlp);
axis([0 1 -4 6]), xlabel('x'), ylabel('f'),grid

%Train SIANN using the third training set
fprintf('\nTraining SIANN with randomly chosen data points...\n');
net_siann3 = new_siann([0 1], [40 1], {'exp' 'purelin'},'train_gdd');
net_siann3 = init_siann(net_siann3);
net_siann3.trainParam.show = 1;
net_siann3.trainParam.epochs = 20;
net_siann3.trainParam.goal = .1;
net_siann3.trainParam.direct = 1;
net_siann3_begin = net_siann3;
[net_siann3, tr_siann3] = train_siann(net_siann3,P3,T3);

%Train MLP using the third training set
fprintf('\nTrain MLP with randomly chosen data points...\n');
net_mlp3 = newff([0 1], [40 1], {'tansig' 'purelin'},…
                 'trainlm', '', 'sse');
net_mlp3 = init(net_mlp3);
net_mlp3.trainParam.show = 1;
net_mlp3.trainParam.epochs = 20;
net_mlp3.trainParam.goal = .1;
net_mlp3_begin = net_mlp3;
figure(3)
[net_mlp3, tr_mlp3] = train(net_mlp3,P3,T3);

%Check result for the third training set
f_siann3 = sim_siann(net_siann3,x);
f_mlp3   = sim(net_mlp3,x);

error_siann3 = mse(f_siann3 - f);
error_mlp3   = mse(f_mlp3   - f);
s_siann = ['E_S_I_A_N_N = ' sprintf('%g',error_siann3)];
s_mlp =   ['E_M_L_P     = ' sprintf('%g',error_mlp3)  ];

plot(x,f,'b',x,f_siann3,'k',x,f_mlp3,'.-r');
title('40 randomly chosen training points');
legend('Actual function', s_siann, s_mlp);
axis([0 1 -4 6]), xlabel('x'), ylabel('f'),grid

%save chapter6_4        %Save data in this session
```