Chapter 2

MULTILAYER NEURAL NETWORK LEARNING

This chapter discusses learning in multilayer neural networks (MNNs). MNNs are by far the most common applications of artificial neural networks (ANNs). The chapter covers fundamental issues such as the different types of MNNs and available learning algorithms. Performance aspects of the different learning algorithms are discussed, as well as difficulties encountered in the learning process.

2.1 Introduction

An artificial neural network (ANN) is a model of the biological neural system of human beings, modeling one of the most important features of the brain - *the ability to learn*. This feature shows parallel to the intellectual development of human beings. As human beings, we learn how to write, read, understand speech, recognize and distinguish pattern - all by *learning from examples.* In the same way, ANNs are *trained*, rather than programmed. ANNs develop solutions to problems unlike conventional data processing techniques which require complex programming.

An artificial neural network consists of processing units, organized in layers of units (also referred to as artificial neurons). Training of an ANN is done using a training algorithm, which is an adaptive way by which a network of processing units organizes themselves to implement the desired behavior: When a network is presented with information to learn (consisting of input attributes and corresponding desired output values), the connection links in between, referred to as the weights, are adjusted to produce a response consistent to the desired output. This learning algorithm is a closed loop of presentation of patterns or examples and of corrections to the network according to a learning rule. An optimization algorithm such as gradient descent, conjugate gradient or second order derivatives techniques, is used to adjust the weights of the network [Becker *et al* 1988]. There are different classes of training algorithms and different topologies of artificial neural networks.

The rest of this chapter is organized as follows: The parallelism between biological and artificial neural networks is discussed in section 2.2 to show how ANNs were inspired from the biological counterpart. A taxonomy of different neural network training algorithms is given in section 2.3. Section 2.4 discusses the training of multilayer neural networks using gradient descent. The learning equations are derived in this section. Section 2.5 discusses problems of learning by gradient descent.

2.2 Biological Neural Networks

The basic building block of biological neural systems is the neuron. A neuron is a cell which communicates information to and from the various parts of the human body. Figure 2.1

shows a simplified representation of a neuron. A neuron consists of a cell body referred to as *soma*, several spline-like extensions of the cell body referred to as *dendrites* and a single nerve fiber referred to as an *axon*. An axon branches out from the soma and connects to many other neurons.

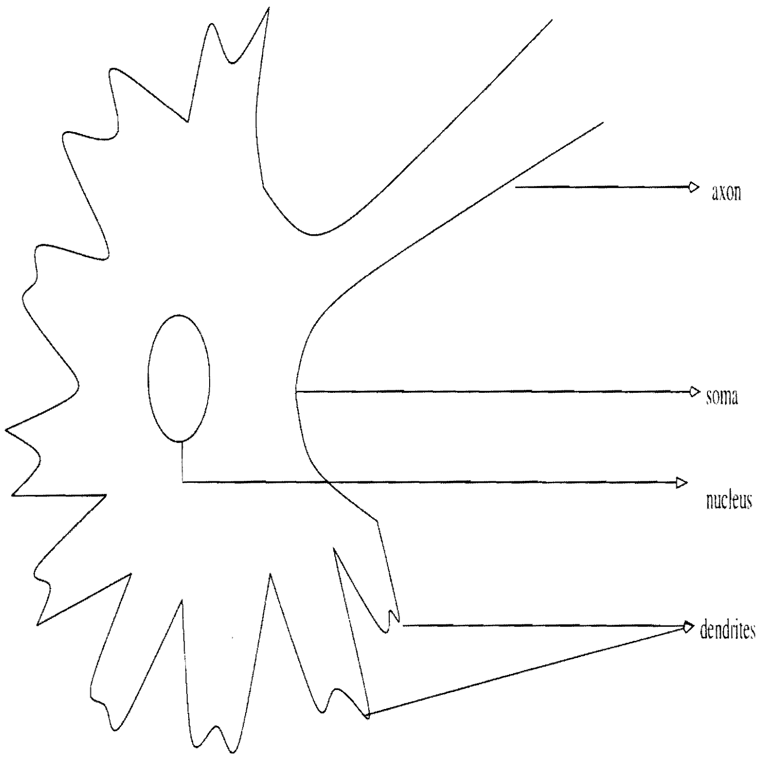


Figure 2.1: A simplified representation of a biological neuron

Dendrites extend from the cell body to other neurons where the dendrites receive signals at a connection point referred to as a *synapse*. These signals serve as inputs which are conducted to the soma (cell body). In the nucleus, these received inputs are summed up. If the cumulative excitation in the nucleus exceeds a threshold, the neuron fires, sending signals down the axon to other neurons. While the biological neural system is extremely complex, an ANN is an attempt at modeling the information processing capabilities of the biological neural system.

An artificial neuron was designed to mimic simple characteristics of the biological neuron. An artificial neuron receives input signals from the environment, or from other artificial neurons. These inputs signals are weighted with a value which models the synaptic strength of the corresponding connection. The weighted sum of the input signals is used to determine the activation level of the neuron. The activation of an artificial neuron is modeled using an activation (or transfer) function. The different activation functions are discussed in section 2.3.2.

Figure 2.2 illustrates a general representation of an artificial neuron. In the rest of this

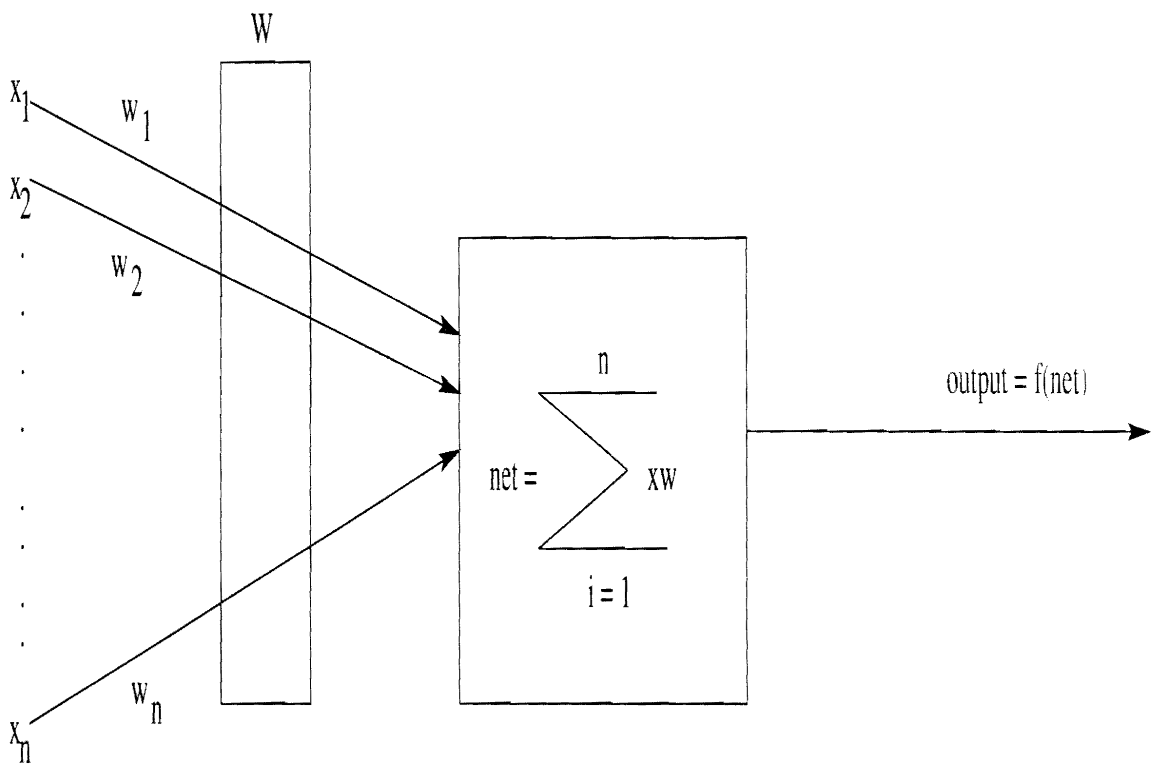


Figure 2.2: An artificial neuron

thesis, the term neural network (NN) is used instead of artificial neural network (ANN).

Several key features of the processing elements of a neural network are suggested by the properties of the biological neuron, namely that,

- a processing unit (neuron) receives many signals from other neurons or the environment;
- these signals may be modified by a weight;
- the processing units sum the weighted inputs which is transformed to an output signal using a squashing function to simulate firing;
- the neuron transmits this single output to other neurons, or to the environment; and
- the output from a particular neuron may be transmitted to many other neurons.

One important characteristic an ANN shares with biological neural systems (BNS) is fault tolerance. A BNS is fault tolerant in two ways: Firstly, human beings are able to recognize many input signals that are somewhat different from any signals they have seen before. Secondly, a BNS can tolerate damage to itself. Human beings are born with as many as 100 billion neurons. Most of these neurons are located in the brain and are not replaced when neurons die [Fausett 1994]. Despite the loss of these neurons, human beings still continue to learn. Even in cases of traumatic neural loss, other neurons can sometimes be trained to take over the function of the damaged cells [Fausett 1994]. In a similar manner, an ANN can be designed to be insensitive to small damage to the network and the network can be retrained in cases of significant damage.

The number of layers, and the way in which neurons are interconnected, resulted in the design of various ANN topologies. Section 2.3.1 surveys different ANN topologies and also discusses the different classes of training available.

2.3 A Taxonomy Of Training

One of the interesting features of neural networks is their ability to learn, which implies that the NN has to be trained. *How is this done?*

The objective of training a NN is to produce desired (or at least consistent) output when a set of inputs is applied to the network. A neural network is trained by applying an input vector to neurons while adjusting the weights according to a predetermined procedure in order to bring the NN's learned concept closer to the desired output. During training, weights gradually converge to values such that each set of input patterns produces a close approximation to the desired output patterns. There are two main training paradigms:

1. **Supervised training,** which is perhaps the most frequently used training method. For training purposes, a *training pattern* is required which consists of a vector of input values and a vector of associated target/desired output values. Patterns can be provided by external teachers or by the system which contains the network, in which case the network is self supervised. The network is usually trained by presenting an input vector to the NN, the actual output of the NN is calculated and compared to the corresponding desired (target) output. Training patterns are grouped into a training set. Each pattern in the training set is presented to the network, and the prediction error used to adjust weights. Patterns in the training set are repeatedly presented to the network until an acceptable error is achieved over the entire training set.

   Supervised learning is analogous to a lesson in school where the teacher applies the correct answer for each problem. Different approaches to supervised learning have been developed.

   - *Error correction learning* which adjusts the connection weights between processing units, in proportion to the difference between the desired and computed values of each neuron in the output layer [Simpson 1990].
   - *Reinforcement learning* which is similar to error-correction learning in that weights are reinforced for properly performed actions and punished for poorly performed actions [Simpson 1990].

   The difference between error correction and reinforcement learning, is that, error correction learning requires an error value for each output unit while reinforcement learning requires only a state to describe the output layer's performance.

2. **Unsupervised training**, also referred to as self-organization learning, requires no target or desired outputs. Hence, no comparison to predetermined responses are needed. Training sets consist solely of input patterns. The task of the NN is to learn to group together patterns that are similar and also to find common threads in a mass of data. The NN is supposed to discover statistically salient features of input patterns and develop its own representation of these patterns. Unsupervised learning is used for tasks such as clustering [Fausett 1994].

For the purpose of this thesis, only supervised training is considered.

2.3.1 Topology of Neural Networks

In addition to the classes of neural network training algorithms, another distinguishing characteristic of the different neural networks is topology. Topology refers to the architecture of neurons, including the interconnection scheme within the network.

Neurons are arranged in one or more than one layer. Neurons within the same layer usually have the same activation function, and are fully connected to the neurons in the next layer. A NN can consist of just a single layer of fully interconnected units, or can have an input and an output layer with zero or more hidden units, referred to as a multilayer neural network (MLNN). Figure 2.3 illustrates a MLNN with a hidden layer. The figure has three units in the input layer with a single output unit. The input layer consists of units that receive input signals from the environment and distributes the signals to the other layers in the network. The output layer returns signals to the environment. Hidden layers are those layers in between the input and output layers. The hidden units provide nonlinearities for the network.

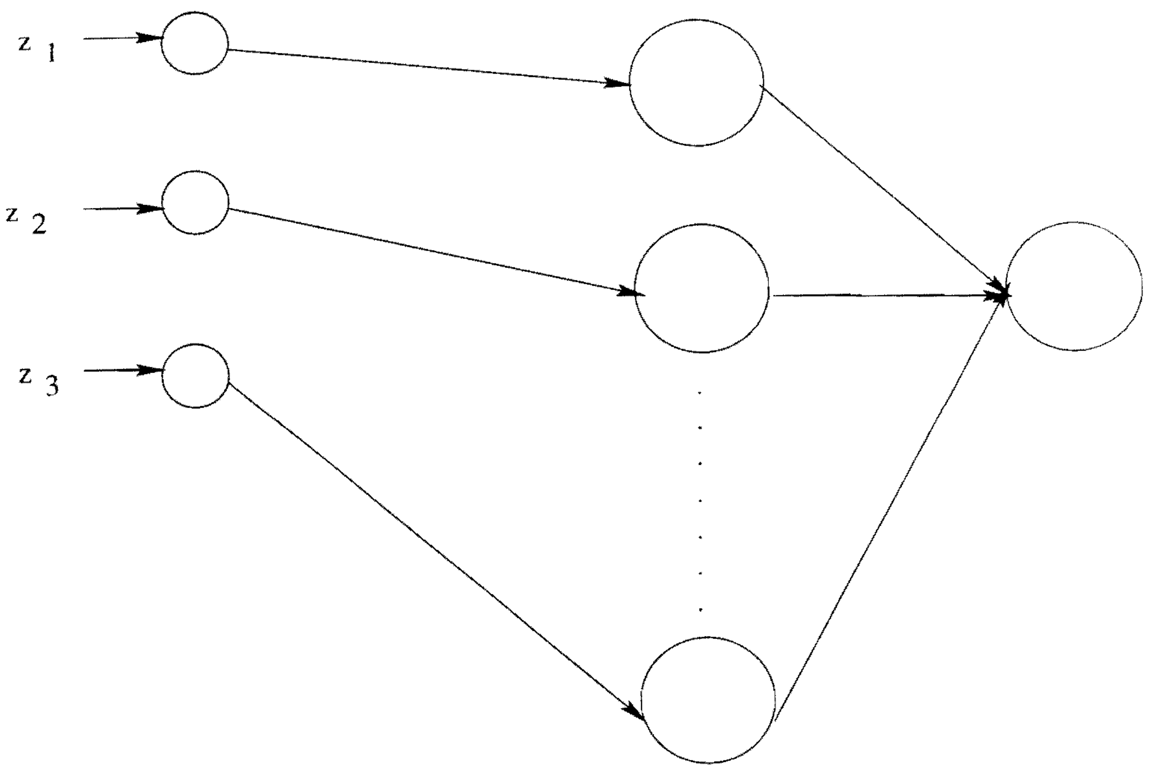


Figure 2.3: A multilayer neural network with a hidden layer

Each neuron produces an activation value (output signal) which usually is a function of the weighted sum of the input signals. The activation value represents the activation level for the neuron. Section 2.3.2 discusses activation functions that can be used in a NN.

2.3.2 Activation Functions

The basic operation of an artificial neuron (unit) involves summing the neuron's weighted input signal and to produce an output signal through application of an activation function to the net input signal. Activation functions map a neuron's domain, which is the input, to a prespecified range - the output. Figure 2.2 illustrated the basic building block of a NN. In figure 2.2 *net* is the weighted input signal. The output signal o is calculated as

$$o = f(net) \tag{2.1}$$

Various mathematical functions have been used as activation functions. There are functions that squashes the net input signal into a finite range. These functions can be discrete functions, such as the ramp and step functions, or continuous functions, for example the arctangent, sigmoid, sine or gaussian (radial basis). Linear functions can also be used as activation functions, in which case the input signal is not mapped into a finite range. Figure 2.4 illustrates the different activation functions that can be used.

One of the major reasons why earlier work on NNs came to a halt, was that the learning rule could not be substantially improved for multilayer NNs using the discrete and linear activation functions [Maren *et al* 1990]. Linear and discrete functions could only solve problems that are linearly separable, and being linearly separable limits the NN to problems (classification) in which the sets of points (corresponding to input values) can be separated geometrically. Hence, the network used then (perceptron) could not solve the XOR problem.

A new learning rule (backpropagation) was developed to handle linearly inseparable functions. However, backpropagation requires continuous, monotonic increasing activation functions, since these functions need to be differentiated when the gradient of the error surface is calculated during the weight update process.

The sigmoid function, given in equation (2.2), is widely used as activation function and is a continuous function bounded in the range (0,1). The sigmoid function is expressed mathematically as:

$$f(net) = \frac{1}{1 + e^{-net}} \tag{2.2}$$

The sigmoid function is desirable because of its simple derivative. The sigmoid function has the advantage of providing a form of automatic gain control. That is, for small signals (*net* near zero), the slope is steep producing high gain in the magnitude of the network's output and as the magnitude of *net* increases, the gain in the magnitude of the network's output decreases. In this way, large input signals can be accommodated by the network without saturation, while small signals are allowed to pass through without excessive attenuation.

i 1576252X
b15213675

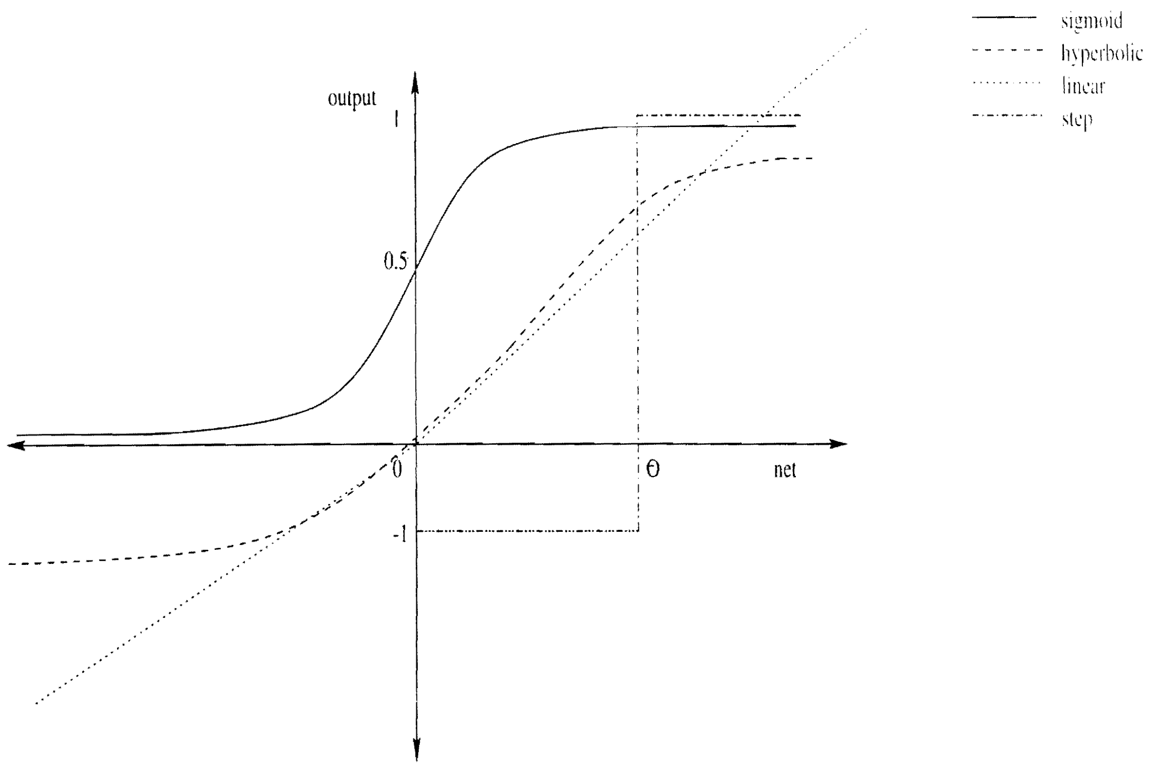


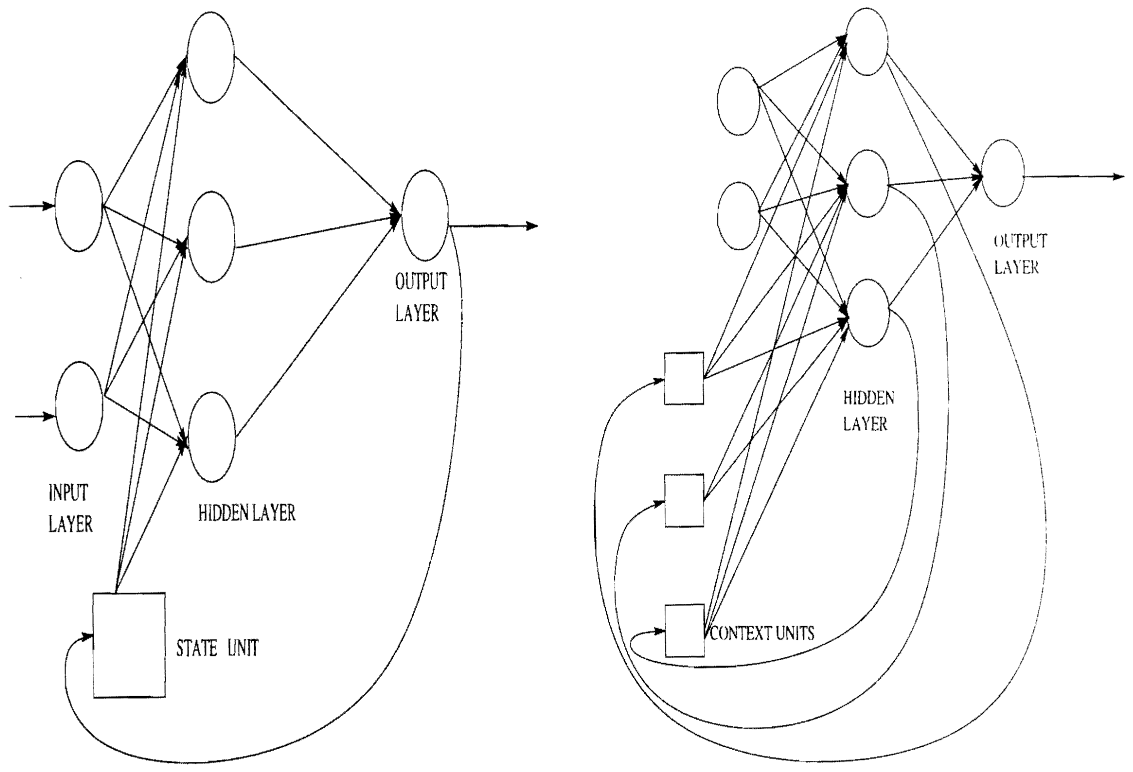
Figure 2.4: Activation functions

2.3.3 Neural Network Types

Based on the different network topologies and training approaches, different types of NNs have been developed. A summary of the different NN types are presented below:

1. **Recurrent neural network (RNN)**: A RNN, also referred to as a feedback neural network, employs feedback connections in order to learn temporal characteristics of data presented for learning. The feedback connections thus allow the network to produce complex time varying outputs in response to simple static input [Carling 1992]. RNNs exhibit properties very similar to short term memory in human beings. There are different types of RNNs, e.g. Jordan and Elman RNNs.

   In Jordan RNNs, the state of the output layer is fed back to state units in the input layer (see figure 2.5(a)), while the state of the hidden layer is copied into context



(a) Jordan RNN (b) Elmann RNN

Figure 2.5: Recurrent Neural Networks (RNNs)

units in the input layer for Elman RNNs (see figure 2.5(b)). Hybrid networks can also be built by combining Jordan and Elman networks. Also, any number of previous time steps can be incorporated by simply having additional state units (for Jordan RNN) and context units (for Elman RNN) for each time step [Carling 1992].

2. **Functional link neural network (FLNN)**: In a FLNN, the input layer is expanded to a layer of functional units, which consists of higher order combinations of the input units [Zurada 1992b, Hussain *et al* 1997]. Each functional unit is fully connected to the next layer. The addition of higher order combinations of inputs artificially increases the dimension of the input space. Figure 2.6 shows an illustration of a functional link neural network.

3. **Product unit neural network (PUNN)**: PUNNs allow learning of higher-order

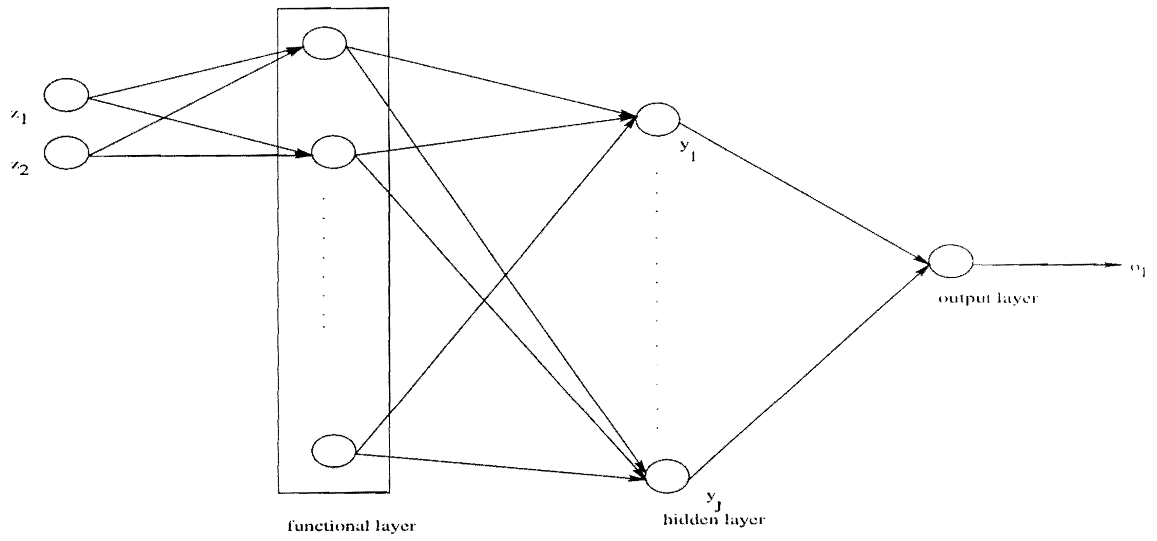


Figure 2.6: A functional link neural network (FLNN)

input terms, by using product units instead of summation units to compute the net signal to a neuron [Durbin *et al* 1989].

A weighted product

$$\prod_{i=1}^{I} z_i^{v_{ji}}$$

is therefore used instead of the usual weighted sum

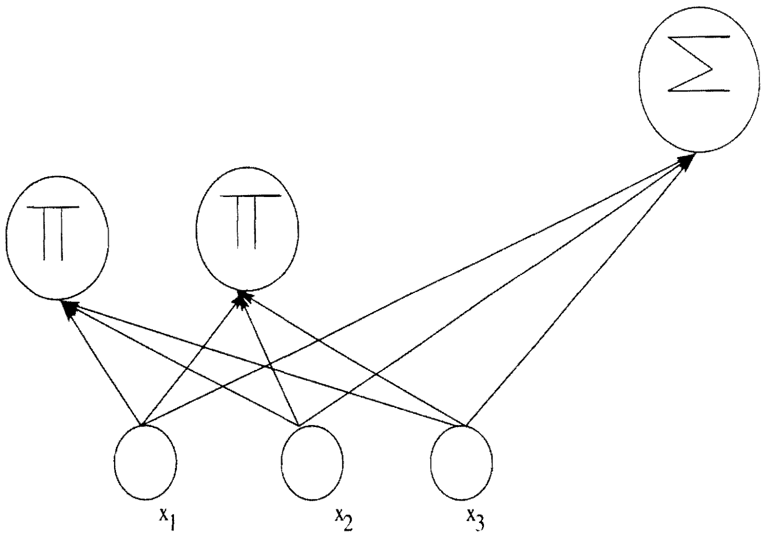
$$\sum_{i=1}^{I} z_i v_{ji}$$

where z_i is the input signal to neuron j, v_{ji} is the weight between neuron i in the previous layer and unit j. Durbin and Rumelhart proposed two PUNN architectures (refer to figure 2.7):

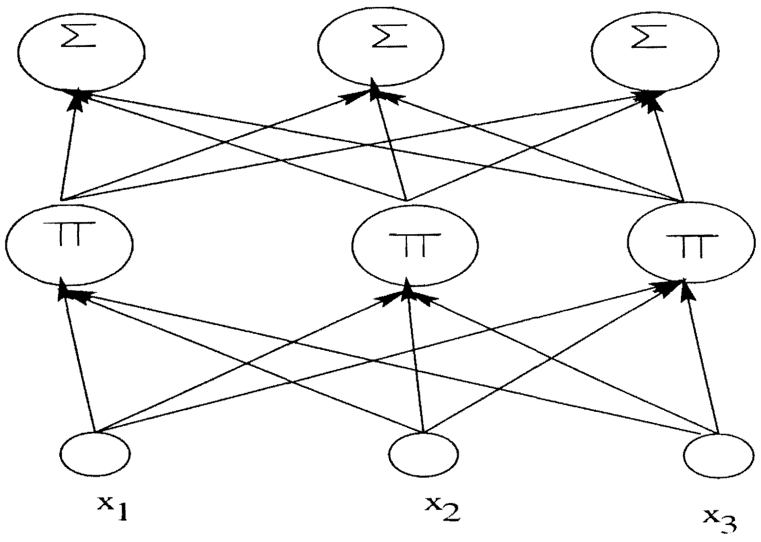
(a) In the first architecture, a set of product units is added to the current summation units in the hidden layer (refer to figure 2.7(a)).

(b) In the second arrangement, layers of product units alternate with layers of summation units (refer to figure 2.7(b))

The main reason for using PUNNs, is to learn to represent generalized polynomial



(a) The first arrangement



(b) The second arrangement

Figure 2.7: Product Unit Neural Networks

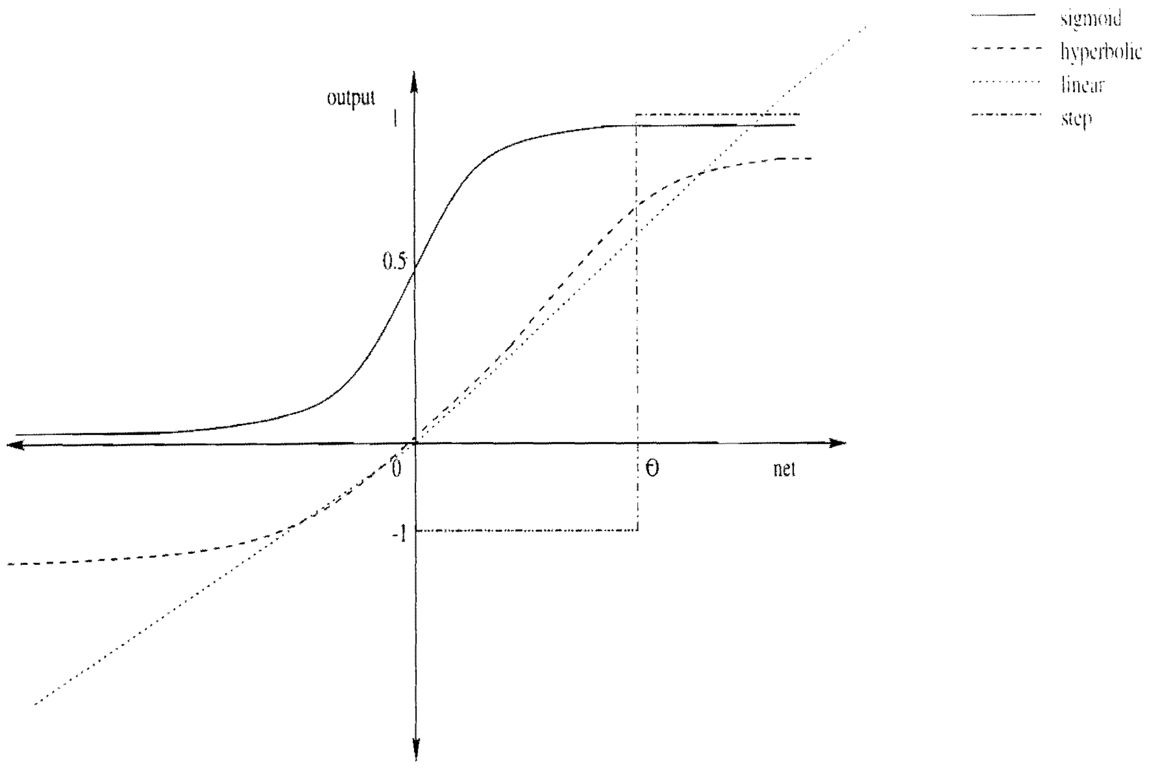


Figure 2.4: Activation functions

2.3.3 Neural Network Types

Based on the different network topologies and training approaches, different types of NNs have been developed. A summary of the different NN types are presented below:

1. **Recurrent neural network (RNN)**: A RNN, also referred to as a feedback neural network, employs feedback connections in order to learn temporal characteristics of data presented for learning. The feedback connections thus allow the network to produce complex time varying outputs in response to simple static input [Carling 1992]. RNNs exhibit properties very similar to short term memory in human beings. There are different types of RNNs, e.g. Jordan and Elman RNNs.

   In Jordan RNNs, the state of the output layer is fed back to state units in the input layer (see figure 2.5(a)), while the state of the hidden layer is copied into context terms in the input and hence a better representation of data in cases where higher order combinations of inputs are significant [Leerink *et al* 1995]. Adjusting the weights is, however, computational expensive since derivatives of these product units are complex due to an exponential term and the occurrence of complex numbers.

4. **Feedforward neural network (FFNN)**: In a FFNN data flows strictly from the input layer to the output layer. A FFNN has no memory and the output is solely determined by the current input and weights values. A feedforward neural network consists of one or more layers of usually non-linear processing units (can use linear activation functions as well). The output of each layer serves as input to the next layer. This thesis concentrates on FFNNs, and studies network learning using FFNNs as well as problems associated with learning in FFNNs.

Apart from the neural network types mentioned above, there are other NN types: for example the single layer Hopfield NN (HNN) [Hopfield 1982, Fausett 1994], and clustering NNs, for example the self organizing map (SOM), which use unsupervised learning [Simpson 1990].

Section 2.3.4 discusses optimization algorithms that can be used to adjust the weights of feedforward neural networks.

2.3.4 Optimization Algorithms

Training a neural network involves finding optimal values for the weights of the network through numerical optimization of a nonlinear objective function. The objective function is usually the sum squared error, computed from the actual network output and the desired output of the NN to be trained. Different optimization algorithms can be applied to NN learning. The algorithm chosen is usually based on the characteristics of the problem to be solved.

1. **Gradient descent optimization** is by far the most common technique used for weight optimization. In training the network, a gradient descent is performed on the error function, which is a function of the weights of the neural network. Weight-s are adjusted to move towards the negative gradient of the objective function [Masters 1993, Becker *et al* 1988]. Gradient optimization is discussed in more details in the next section.

2. **Newton optimization** uses a better approximation of the error function than the gradient descent technique. The newton technique uses second derivatives and gradient information of the error function to determine the next step direction. This helps in reducing the number of steps taken to reach a minimum, thus achieving faster convergence. However, Newton optimization has the disadvantage of being computationally expensive because the inverse of the Hessian matrix needs to be calculated at each training step. Newton's optimization should preferably be used with neural networks with a few number of weights due to the cost of computing the inverse of the Hessian matrix[Darken *et al* 1992, Becker *et al* 1988].

3. **Pseudo newton optimization** is an adaptation of Newton's method. Pseudo newton optimization computes an approximation to the inverse Hessian matrix, and is therefore more computationally efficient than Newton's optimization. Pseudo newton optimization should be preferably used for neural networks with a moderate number of weights due to the cost of approximating the Hessian matrix [Darken *et al* 1992].

4. **Conjugate gradient optimization** is used for large optimization problems, since it does not require the computation and storage of the Hessian matrix. Conjugate gradient uses only gradient information. The objective of conjugate gradient is to minimize both the weight vector and a direction vector. Conjugate gradient is related to gradient descent optimization using momentum, because the weight search in conjugate gradient optimization combines the new gradient direction and the previous gradient direction. Each step involves computing a conjugate direction followed by

a line search, to get an approximate minimum in the conjugate direction. Conjugate gradient optimization increases speed of training and the convergence of the network [Becker *et al* 1988, Møller 1993].

5. **Simulated Annealing** can be used where the objective function (the error function in neural network training) is not differentiable. Optimization is performed by randomly perturbing the independent variables (inputs in this case) and keeping track of the best (lowest error) function value for each randomized set of variables. Simulated annealing can be combined together with other optimization algorithms such as conjugate gradient, where simulated annealing is used to find a good initial weight vector, after which conjugate gradient is used to find the local minimum [Masters 1993, Desai *et al* 1996].

2.3.5 Why Neural Networks?

Neural network applications emphasize areas where NNs appear to offer a more appropriate approach than traditional computing has. NNs can be used when data, on which conclusions are to be based, is noisy.

When the influential or informative patterns are subtle or hidden, a neural network has the ability to discover patterns which are not clear, or unknown, to the human researcher or standard statistical methods. For example, to determine the credit worthiness of a loan applicant, the information needed is hidden within data on the spending and the payment history of loan applicants. NNs have shown to provide decisions superior to those made by human beings [Masters 1993]. Neural networks have also been applied to data that exhibits significant unpredictable nonlinearity [Masters 1993]. NNs adapt to predict future values not based on strictly defined models, and offer possibilities for solving problems that require pattern recognition, pattern mapping, dealing with noisy data, pattern classification and function approximation.

Specific areas where NNs have been applied include, amongst others:

- Neural networks have excelled in **pattern recognition**. NNs deal with the complexities inherent in many applications such as recognizing patterns in speech, radar and seismic readings. A real world application is the NETTALK, a neural network designed by Sejnowski and Rosenberg to produce phonetic strings which in turn specify pronunciation for written texts [Dayhoff 1990].

- NNs are used for **pattern classification**. Input patterns of a network are mapped into one or more classes. That is, each pattern belongs to one of the classes [Fausett 1994]. For example, NNs are used for medical diagnosis to identify diseases of the heart from electrocardiograms. NNs can also be used in plant classification to determine crop types from satellite photographs [Masters 1993].

- NNs have also been used in **adaptive control** applications such as in robots and automatic vehicles. Neural networks are used to control robots in the industry [Dayhoff 1990].

- Neural networks are used in **financial analysis** problems such as credit assessment and financial forecasting. NNs have also find application in optimization, scheduling and routing problems. A practical application is in optimizing resources for airlines [Dayhoff 1990].

- NNs are used in **function approximation** problems. A NN can learn a given function or time series problem when presented with training patterns representing that function or time series. This application has found its usefulness in forecasting, such as weather and in the stock exchange market.

- Neural networks are used for **database mining**. A major problem which surfaced in information retrieval is that explicit information can easily be retrieved while implicit information can not. Implicit information is distributed across the patterns stored in the database and is difficult to extract by human experts. NNs are one of the most promising technologies available to extract such implicit information - a process referred to as data mining. [Towell *et al* 1993, Fu 1994].

2.4 Gradient Descent Optimization

Multilayer neural networks (MLNNs) perform excellently in most applications, especially in classification problems because of the inclusion of one or more hidden layer. Training a MLNN is not as straight forward, nor as easy, as training a single layer network. This section discusses training of MLNNs using gradient descent. Complete derivationΦs of the learning equations are given and problems with gradient descent optimization are discussed.

2.4.1 Introduction

NNs that are trained using GD are referred to as backpropagation neural networks (BPNNs). In order to train the network successfully, the output of the network is made to approach the desired output by continually reducing the error between the network's output and the desired output. This is achieved by adjusting the weights between layers; by calculating the approximation error and backpropagating this error from the final layer to the first layer. The weights are then adjusted in such a way to reduce the approximation error. The approximation error is minimized using the gradient descent optimization technique [Rogas 1996].

The gradient descent technique searches for the minimum of the error function in the weight space. The combination of weights which minimizes the error function is considered to be the solution to the learning problem. When an input pattern is presented to the network. the network produces an output $o_k^{(p)}$ for output unit o_k which is different from the target value $t_k^{(p)}$.

The objective of training is then to minimize the error arising from these two values over the entire training set. The error function is defined as the sum squared error function (SSE):

$$E = \frac{1}{2}\sum_{p=1}^{P}\sum_{k=1}^{K}\left(t_k^{(p)} - o_k^{(p)}\right)^2 \tag{2.3}$$

where P is the total number of patterns in the training set, K is the total number of output units, $t_k^{(p)}$ is the target value for kth output unit for pattern p, and $o_k^{(p)}$ is the output value for the k-th output unit for pattern p.

The gradient for the error function is computed and is used to adjust the weights. Weight adjustment can be done in two ways:

- **Batch training** which adjusts and updates the weights after presenting a number of training patterns. Weight changes are accumulated and applied once only. Batch training is also referred to as offline training.
- **Online Training** where the weights are adjusted after each pattern presentation. Online training has the advantage of not needing a separate memory to store the derivatives of patterns as is needed by the offline training.

Training using GD involves two passes:

1. **The forward pass:** During the forward phase, each input unit $z_i^{(p)}$ receives an input signal and distribute this signal to the hidden units y_j for all $j = 1, \ldots, J$. Each hidden unit then computes its activation and sends the activation signal to each output unit at the output layer or to hidden units in the next hidden layer if

there are more than one hidden layer. As there are no connections within a layer, all the units in that layer can have their output computed in parallel, while the layers are dealt with in sequential order. The output layer provides the response of the network for a given input pattern.

2. **The backward pass:** Each output unit compares its computed activation $o_k^{(p)}$ with its target value $t_k^{(p)}$ to determine the associated error for that pattern. The error is backpropagated to all units in the previous layer and is used to update the weights between the output and hidden layers. The accumulated error at each hidden unit is then calculated, and backpropagated to adjust the weights between the input and hidden layers. The error value associated with each processing unit reflects the error of that unit. A larger error value indicates that a larger correction will be made to the corresponding weights.

2.4.2 Gradient descent training algorithm

Certain aspects have to be addressed before commencing training of multilayer networks. One important aspect is the activation function used in the hidden and output layers. GD requires the activation function to be continuous, differentiable and monotonically increasing. For the purposes of this thesis the logistic (sigmoid) function is assumed. Another issue is the data set: the output value of logistic function is always in the range (0,1), thus requiring scaling of the desired output (target) before training to fit into this range. Though it is not required to scale inputs, it is advisable to scale the inputs to [-1,1] if logistic function is used. The input values will then lie within the active range of the sigmoid function. The number of hidden layers also has to be considered. Although gradient descent can be applied to any number of layers, it has been shown that a single layer of hidden units is sufficient to approximate any function with many discontinuities to arbitrary precision provided that the activation function is non-linear [Krose *et al* 1993]. This thesis assumes a single hidden layer.

The training algorithm is summarized below:

1. Weight initialization: Set all weights to small random values. Let v_{ji} be the weight between the j-th hidden unit and i-th input unit, and w_{kj} the weight between the k-th output unit and j-th hidden unit.

2. Calculate the activation of the units in the network, layer-by-layer, starting from the input layer.

   - The activation level of each input is the value of the training pattern applied to the input.
   - The activation of each hidden and output unit is calculated as:

$$y_j^{(p)} = f_{y_j}^{(p)}(\sum_{i=1}^{I} v_{ji}z_i - v_{j0}) \tag{2.4}$$

$$o_k^{(p)} = f_{o_k}^{(p)}(\sum_{j=1}^{J} w_{kj}y_j - w_{k0}) \tag{2.5}$$

   where $y_j^{(p)}$ is the activation of the j-th hidden unit, and $o_k^{(p)}$ is the activation of the k-th output unit for pattern p. K is the total number of output units, I is the total number of input units and J is the total number of hidden units. v_{j0} is the weight connected to the bias unit in the input layer, while w_{k0} is the weight connected to the bias unit in the hidden layer. The term **bias** is discussed in the section 2.4.3.

3. Weight adjustment

   - Start at the output units and recursively propagate error signals to the input layer.

- Calculate the weight adjustments:

  The output $o_k^{(p)}$ is compared with the corresponding target value $t_k^{(p)}$ over the entire training set using the function

$$E^{(p)} = \frac{1}{2}\sum_{k=1}^{K}\left(t_k^{(p)} - o_k^{(p)}\right)^2 \tag{2.6}$$

to express the error in the network's approximation of the target function. Minimization of $E^{(p)}$ by GD requires the partial derivative of $E^{(p)}$ with respect to each weight in the network to be computed. The change in weight is proportional to the corresponding derivative:

$$\Delta v_{ji}(t+1) = -\eta\frac{\partial E^{(p)}}{\partial v_{ji}} + \alpha\Delta v_{ji}(t) \tag{2.7}$$

$$\Delta w_{kj}(t+1) = -\eta\frac{\partial E^{(p)}}{\partial w_{kj}} + \alpha\Delta w_{kj}(t) \tag{2.8}$$

where: η is the learning rate which is in the step length in the negative gradient direction. The value of η is usually between 0 and 1. The last term is a momentum term which is a function of the previous weight change. The concept of **momentum** is discussed in the section 2.4.3.

For notational convenience, the (p) superscript is dropped in the remainder of this section. The reader should keep in mind that the equations below are for a single pattern.

The partial derivative of E with respect to w_{kj} is computed as

$$\frac{\partial E}{\partial w_{kj}} = \frac{\partial E}{\partial o_k}\frac{\partial o_k}{\partial w_{kj}} \tag{2.9}$$

The term $\frac{\partial E}{\partial o_k}$ in equation (2.9) is calculated as

$$\begin{aligned}\frac{\partial E}{\partial o_k} &= \frac{\partial}{\partial o_k}\left[\frac{1}{2}\sum_{k=1}^{K}(t_k - o_k)^2\right] \\ &= -(t_k - o_k)\end{aligned} \tag{2.10}$$

and

$$\frac{\partial o_k}{\partial w_{kj}} = o_k(1 - o_k)y_j \tag{2.11}$$

From equations (2.10) and (2.11)

$$\frac{\partial E}{\partial w_{kj}} = -(t_k - o_k)o_k(1 - o_k)y_j \tag{2.12}$$

Therefore,

$$\Delta w_{kj} = \eta(t_k - o_k)o_k(1 - o_k)y_j \tag{2.13}$$

The contribution of hidden units to the output error is not readily known. However, the error measure can be written as a function of the error contribution over all output units.

$$\begin{aligned}\frac{\partial E}{\partial v_{ji}} &= \sum_{k=1}^{K} \frac{\partial E}{\partial o_k}\frac{\partial o_k}{\partial y_j}\frac{\partial y_j}{\partial v_{ji}} \\ &= \frac{\partial y_j}{\partial v_{ji}} \sum_{k=1}^{K} \frac{\partial E}{\partial o_k}\frac{\partial o_k}{\partial y_j} \\ &= \frac{\partial y_j}{\partial v_{ji}} \sum_{k=1}^{K} -(t_k - o_k)o_k(1 - o_k)y_j w_{kj}\end{aligned} \tag{2.14}$$

The partial derivative $\frac{\partial y_j}{\partial v_{ji}}$ is computed as

$$\frac{\partial y_j}{\partial v_{ji}} = y_j(1 - y_j)z_i \tag{2.15}$$

Therefore,

$$\Delta v_{ji} = \eta \sum_{k=1}^{K}(t_k - o_k)o_k(1 - o_k)y_j w_{kj} y_j(1 - y_j)z_i \tag{2.16}$$

4. Update the weights:

$$w_{kj}(t+1) = w_{kj}(t) + \Delta w_{kj}(t+1) \tag{2.17}$$

$$v_{ji}(t+1) = v_{ji}(t) + \Delta v_{ji}(t+1) \tag{2.18}$$

   where t represents the current time step, Δv_{ji} and Δw_{kj} are the weight adjustments from equations (2.13) and (2.16).

5. Test for convergence, for example if an acceptable MSE has been reached, or the maximum number of epochs has been exceeded. Go to step (2) and repeat until convergence in terms of selected stopping criteria.

   An iteration, which is referred to as an *epoch*, is one pass through the training set which includes presenting training patterns, calculating the activation values, and modifying the weights.

2.4.3 Additional Features To The Training Algorithm

Some features have been incorporated into the GD training algorithm to improve neural network learning.

- **Addition of neuron bias**: The addition of a bias to the neural networks is to offset the origin of the activation function. This allows more rapid convergence of the training process [Masters 1993, Wasserman 1989, Fausett 1994]. By adding a bias unit with a constant activation value of -1. The weight between the bias unit and a unit in the next layer serves as bias to that unit. These bias weights are trained in the same way as the other weights. Therefore, for hidden units

$$y_j = f(\sum_{i=0}^{I} v_{ji} z_i \tag{2.19}$$

  and for output units

$$o_k = f(\sum_{j=0}^{J} w_{kj} y_j \tag{2.20}$$

with $z_0 = -1$ and $y_0 = -1$. v_{j0} is the weight to the bias unit z_0 in the input layer and w_{k0} is the weight to the bias unit y_0 in the hidden layer.

- Another prominent feature that can be added to improve the performance of the network is to add a **momentum term**. The addition of a momentum term helps to avoid oscillations in weight adjustments [Beale *et al* 1990]. Momentum is proportional to the magnitude of previous weight changes. Weight changes are then in the direction that is a combination of the current gradient and the previous gradient. Momentum allows the network to make reasonably large weight adjustments, as long as corrections are in the same direction for several patterns, while using a smaller learning rate. Momentum also reduces the chances of getting stuck in a local minimum [Wasserman 1989, Dayhoff 1990] - a problem of learning with the gradient descent technique which is discussed in the next section. In effect, momentum tries to find the global minimum of the error surface by repeatedly jumping in the downhill direction. Momentum is typically implemented by multiplying a numeric parameter between zero and one with the previous weight change (refer to equations (2.7) and (2.8)).

2.5 Learning Difficulties With Gradient Descent Optimization

Despite gradient descent's usefulness in training multilayer neural networks, there are difficulties associated with learning using gradient descent. Problems with GD include network paralysis, local minima and slow convergence.

One of the problems that occurs when GD is used is *network paralysis*. Network paralysis occurs when the weights are adjusted to very large values during training. Large

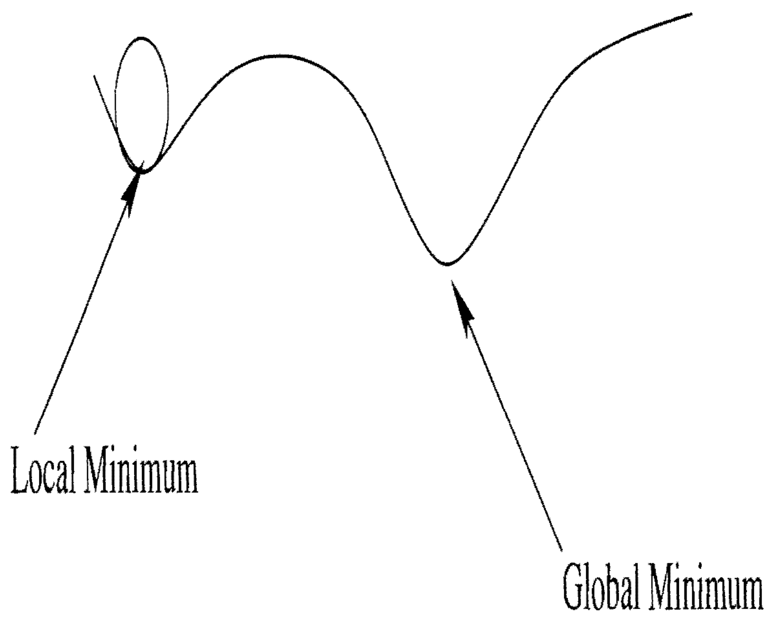


Figure 2.8: An illustration of local and global minimum

weights can force most of the units to operate at extreme values, in a region where the derivative of the activation function is very small. Since the error backpropagated is proportional to the derivative of the activation function (refer to equations (2.9) and (2.14), the training process can come to a stand still [Wasserman 1989].

A prominent problem with training using GD is the occurrence of *local minima* [Rumelhart *et al* 1986]. The network finds a combination of weights that that represents a local and not a global minimum. The gradient descent technique follows the slope of the error surface downward, constantly adjusting the weights towards the minimum. The error surface could be highly complex: full of hills, valleys, folds and gullies in high dimensional space. The network may therefore, get trapped in a local minimum (shallow valley), while there is a much deeper minimum nearby or elsewhere. Figure 2.8 illustrates the concept of local minimum and global minimum.

There is also the problem of *slow convergence*: A multilayer neural network requires many repeated presentations of the input patterns, for which the weights need to be adjusted before the network is able to settle down into an optimal solution. The method of gradient descent could be very slow to converge for a complex problem due to the complexity of the error surface [Wasserman 1989].

Overfitting and *underfitting* are not unique problems of GD but general problem-s of any learning or regression algorithm. Overfitting occurs when a network has too many training units (an oversized architecture), causing the network to produce good results with the training data, but performing badly with data not seen during training. Rather than learning the basic structure of the data, the network learns the irrelevant details, for example noise in the training patterns [Sarle 1995, Schittenkopf *et al* 1997]. A low training error therefore does not always imply a good performance of the network. A network can also be *underfitted*, which occurs when the number of training units in a network is too few, i.e. an undersized architecture. Thus the network fails to approximate the true form of the relationship between inputs and targets.

2.5.1 Solutions to these learning difficulties

Many research efforts have been invested in the study of how to improve the learning of multilayer neural networks. Approaches to improve performance range from finding the optimal learning rate to finding the optimal network architecture. Some of the most promising approaches are discussed below:

1. **Adaptive learning rate and momentum factor:** Rather than using a fixed learning rate in training, the learning rate and momentum can be adjusted dynamically during training [Weir 1990, Fausett 1994]. Decreased training time and improved convergence have been achieved using adaptive learning rate and momentum. A careful selection of the learning rate is often necessary to ensure smooth convergence. A large learning rate can cause network paralysis and a small learning rate causes slow convergence. An advantage of a large learning

rate is to accelerate learning when a plateau is reached in the weight space. A small learning rate, on the other hand, is helpful in slowing down learning when a valley is approach in the search space [Yu *et al* 1997].

A momentum factor is used to smooth error oscillation. Plaut *et al* have shown that low momentum is good to maintain movement along a particular direction in the error surface, but should be increased when the learning procedure has settled in a stable direction of movement [Plaut *et al* 1986]. The learning rate and momentum should therefore be varied according to the region where the weight adjustment is. An optimal learning rate for a learning problem can also be found [Weir 1990]. However, the optimal learning rate is problem dependent.

2. **Random weight initialization:** The choice of initial weight values influences whether the network converges quickly or not [Fausett 1994]. The weight update between two units depends on both the derivative of the objective (error) function with respect to weights, as well as the activation value of units. Initial weights must not be too large, to ensure that the initial input signal of the a hidden unit or output unit does not fall in the region where the derivative of the sigmoid function is very small. If the derivative is small, the net input of the hidden or the output unit will be close to zero and will cause extremely slow learning due to small weight updates. Weights are initialized *randomly* to break symmetry [Rumelhart *et al* 1986]. Symmetry occurs when all weights are initialized to the same value. Consequently, the hidden units are assigned identical error values. All weights in the network are then adjusted in an identical manner, and thus prevent the error function from being reduced. Weights are usually initialized *randomly* to small values [Rumelhart *et al* 1986].

3. **Optimal network architecture selection:** The achievement of good performance in a trained network is through careful selection of the network size. An oversized network can lead to overfitting of the data but on the other hand, a

small sized (simple) network can lead to underfitting [Le Cun 1990].

Optimal architecture selection is adaptive in the sense that adjusting neural network size is incorporated into the network training. Research into optimal architecture selection is split into three areas: growing the network during training by adding more parameters to the network [Hirose *et al* 1991, Jutten *et al* 1995], pruning the network by removing redundant parameters during training [Sietsma *et al* 1988, Engelbrecht *et al* 1996, Le Cun 1990] or regularization through penalty terms added to the objective function [Weigend *et al* 1991, Kamimura *et al* 1994, Karayiannis *et al* 1993].

– **Network pruning** involves training an oversized network and removing redundant and irrelevant network parameters, including units and / or weights. Starting with an oversized network rather than a small or undersized network, the network is guaranteed to learn the desired input and output mapping [Le Cun 1990]. Once a network has learn a solution to a problem, the network can then be pruned to the minimum size [Sietsma *et al* 1988]. Pruning aims at solving the problem of the overfitting as well as reducing the computational cost of training and applying the network [Le Cun 1990]. Selecting the parameters to remove is the main focus of pruning methods and is based on different criteria proposed by different researchers. Le Cun *et al* introduced the concept of network pruning through their work on optimal brain damage (OBD) [Le Cun 1990]. Le Cun *et al* empirically showed that by removing unimportant weights from a network, several improvements could be achieved. These improvements include better generalization, fewer training examples and improved speed of learning. OBD reduces the size of a network by selectively deleting weights. The goal of OBD is to find a set of parameters, that when deleted would cause the least increase in the error function. To find such set of parameters, Le Cun *et al* defined the *saliency* of parameter as the change in error caused by deleting that parameter. The parameter with least saliency is deleted. The second derivative information is used to calculate this saliency and therefore is computationally complex due to calculation of Hessian matrix. Hassibi *et al* extended OBD to remove the required retraining after pruning [Hassibi *et al* 1994]. Their approach, referred to as OBS, automatically computes the adjustments needed to the remaining weights due to the pruning of weights. Engelbrecht *et al* developed a pruning algorithm where the sensitivity of the output of the network to small parameter perturbations is used to identify irrelevant parameters [Engelbrecht *et al* 1996]. This algorithm prunes both input and hidden layers of feedforward neural networks. Units that have the least statistical influence on all units in the succeeding layers are pruned. An adaptation to this pruning algorithm was also proposed by Engelbrecht *et al* [Engelbrecht *et al* 1999b]. A new pruning heuristic based on variance analysis of sensitivity information is used to find irrelevant parameters.

- **Network growing** involves growing the network during training. Hidden units are added to the network when needed. Network growing reduces computational cost and complexity of the trained network [Jutten *et al* 1995]. A reduction in computational cost is achieved because the optimal architecture needed to train a network is problem dependent. A small network architechure have fewer weights than a large network and thus needs a few weight adjustments. Once the optimal solution for a problem is obtained, the resulting network has an optimal architecture [Jutten *et al* 1995]. Hirose *et al* also used network growing to solve the problem of local minima [Hirose *et al* 1991]. In their research, Hirose *et al* added more hidden units to a network being trained as soon as the network starts overfitting. The error function was used to detect local minima.
- **Regularization,** where all weights are penalized. Regularization

is achieved by adding a penalty term to the objective function [Weigend *et al* 1991]. In doing so, network complexity is penalized. The effect is that redundant weights are driven to zero, while active weights retain their importance [Kamimura *et al* 1994, Karnin 1990]. Weight decay is one form of regularization [Fu 1994].

4. **Training with jitter:** Jitter is artificial noise deliberately added to inputs during training. Training with jitter is a form of regularization, such as weight decay. An advantage of jitter is that the NN can be brought out of a local minimum [Beale *et al* 1990]. Injecting artificial noise into inputs during training is very effective in improving generalization performance when small training sets are used. Noise injected into inputs is assumed to have zero mean and a small variance in order not to change the distribution of the given training data.

5. **Adaptive learning function:** Activation functions can be adapted and trained just like the weights of a NN. This adaptation improves learning in terms of faster convergence and more accurate results [Zurada 1992a, Engelbrecht *et al* 1995, Fletcher *et al* 1994]. Zurada [Zurada 1992a] and Fletcher *et al* [Fletcher *et al* 1994] proposed a learning rule where the steepness or slope of the activation function used for learning is trained alongside with the weights. The learning rule produced better solutions and a faster convergence to problems when compared to conventional error backpropagation. Another research on adaptive learning functions is the gamma learning proposed by Engelbrecht *et al* [Engelbrecht *et al* 1995]. Gamma learning extends the lamda rule of Zurada, by dynamically adjusting the output range of the sigmoid activation function, thereby performing automatic scaling.

6. **Active learning** involves making optimal use of the training data. Much research has been done in developing active learning models [Engelbrecht *et al* 1998, Engelbrecht *et al* 1999a, Engelbrecht *et al* 1999c,

Zhang 1994, Röbel 1994a, Plutowski *et al* 1993, Cohn *et al* 1996]. Active learning refers to the selection of a subset of the available training data dynamically during training, where the subset contains the most informative data. Active learning has been found to save computational cost and reduce training time [Cohn *et al* 1996, Plutowski *et al* 1993, Röbel 1994a, Engelbrecht *et al* 1999a]. This thesis presents a survey and comparison of active learning algorithms for function approximation and time series problems. The next chapter elaborates on active learning.

2.6 CONCLUSION

This chapter discussed the training of the neural networks. A background introduction into multilayer neural networks was given. The chapter focused on training feedforward MLNNs using gradient descent optimization.

The learning equations were derived and the problems of training a NN using gradient descent as well as the solutions to these problems were discussed.

The next chapter discusses one of the methods to improve learning with gradient descent technique, i.e. *active learning*.

Problem	*Equation*	P_C-P_G-P_V	*Architecture*
F1	(3.3)	600 -200-200	2-5-1
TS1	(3.4)	600-200-200	1-5-1
TS2	(3.5)	600-200-200	2-5-1
TS3	(3.6)	140-60-60	10-10-1
TS4		600-200-200	2-5-1
TS5		600-200-200	2-7-1

Table 3.1: Summary of the functions and time series used

that

$$\begin{aligned} D_C \cap D_V &= \emptyset \\ D_C \cap D_G &= \emptyset \\ D_G \cap D_V &= \emptyset \end{aligned}$$

Let P_C be the number of training patterns in D_C, P_V the number of training patterns in D_V and P_G the number of patterns in test set D_G. Table 3.1 shows the size of these sets for each problem. D_C is the candidate training set from which training patterns are selected. D_V contains data used to determine the generalization factor during training. D_G contains data used to determine the generalization performance of the network.

The performance of the active learning algorithms was tested on clean and noisy data, as well as data containing outliers. Section 3.5.1 explains the experimental procedure, including a discussion of the performance criteria used to compare the learning algorithms. The results are compared in section 3.5.2.

The characteristics of the functions and time series used for experimentation are discussed next. The following functions and time series were used:

1. Function $F1$ is defined as (see figure 3.1(a))

$$F1 : F(z_1, z_2) = \frac{1}{2}(z_1^2 + z_2^2) \tag{3.4}$$

where $z_1, z_2 \sim U(-1,1)$. All target values were scaled to the range [0,1].

2. Time series TS1 is a sine function defined as (see figure 3.1(b)),

$$TS1 : F(z) = \sin(2\pi z)e^{(-z)} + \zeta \tag{3.5}$$

where $z \sim U(-1,1)$ and $\zeta \sim N(0, 0.1)$. Target values were scaled to the range [0,1].

3. Time series TS2 is the henon-map function defined as (refer to figure 3.1(c)),

$$\begin{aligned} TS2 : o_t &= z_t \\ z_t &= 1 + 0.3z_{t-2} + 1.4z_{t-1}^2 \end{aligned} \tag{3.6}$$

where $z_1, z_2 \sim U(-1,1)$. The target values were scaled to the range [0,1].

4. Time series TS3 is a difficult time series, having 10 input parameters of which 7 are irrelevant (see figure 3.2(c)).

$$\begin{aligned} TS3 : o_t &= z_t \\ z_t &= 0.3z_{t-6} - 0.6z_{t-4} + 0.5z_{t-1} + 0.3z_{t-6}^2 - 0.2z_{t-4}^2 + \zeta_t \end{aligned} \tag{3.7}$$

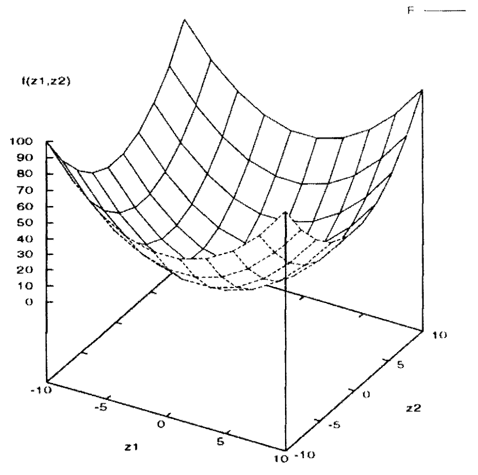
for $t = 1, \cdots, 10$, where $z_4, z_6, z_9 \sim U(-1,1)$ and $\zeta_t \sim N(0, 0.05)$. All target values were scaled to the range [0,1].

5. Time series TS4 is a convolution of two discrete functions with outliers. Figure 3.2(a) shows an illustration of this function.

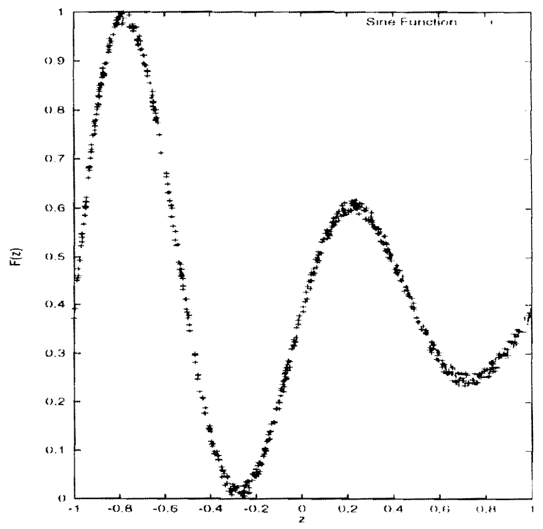
6. Time series TS5 is the sine function TS1 with 5% of the candidate training set consisting of outliers (see figure 3.2(b)).

3.5.1 Experimental Procedure

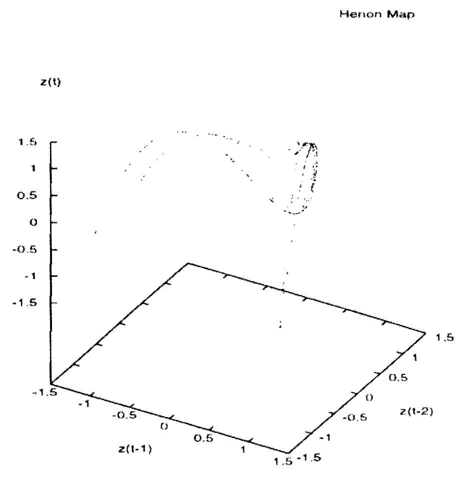
In order to obtain statistically valid assertions in comparing experimental results of the four learning algorithms, thirty simulations were performed for each problem.



(a) F1

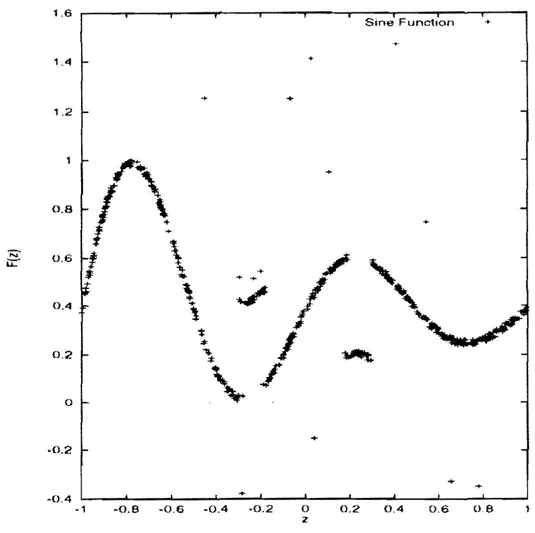


(b) TS1

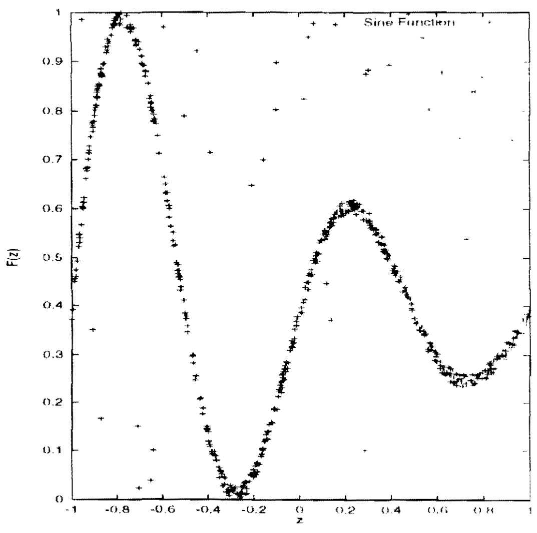


(c) TS2

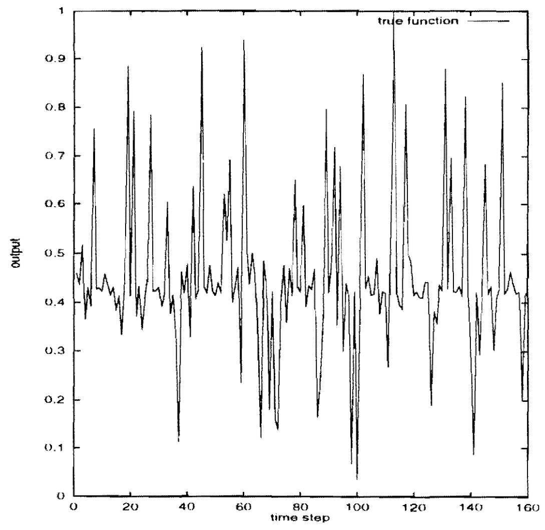
Figure 3.1: Function and Time series problems to be approximated



(a) TS4



(b) TS5



(c) TS3

Figure 3.2: Time series problems to be approximated

Online training was used for the active learning algorithms. The initial subset size for incremental learning algorithms consisted of one pattern and a subselection size of one pattern was used. Each simulations was executed for 2000 epochs. A learning rate 0.1 and momentum 0.9 were used for all the approximation problems . Results reported are averages over the 30 simulations together with 95% confidence intervals as obtained from the t-distribution.

The selective learning algorithm was not applied to F1, since F1 is not a time series problem. The τ value used in the subset selection criterion for AL was adjusted for each problem using a trial and error approach. For TS3, a high τ was used ($\tau = 1000$), a value of 100 was used for TS1, TS4 and TS5 while a value of 180 was used for TS2 and F1.

Performance measures

To evaluate the performance of each learning algorithm, the following performance criteria were used:

1. The mean squared error (MSE) was used as a measure of accuracy. The MSE measures how well a function is approximated by the network, and is defined as

$$MSE = \frac{\sum_{p=1}^{P} \sum_{k=1}^{K} \left(t_k^{(p)} - o_k^{(p)}\right)^2}{2\ K\ P}$$

.

   A MSE value close to zero shows a small error between the target and the output function. The MSE over the three sets D_V, D_G and D_C were computed. The MSE over D_G, denoted by E_G provides an unbiased estimate of the generalization error since the patterns in D_G were not used for training.

2. Röbel's generalization factor ρ was used to measure overfitting effects. The generalization factor was computed as $\rho = \frac{E_V}{E_C}$, where E_C is the MSE over candidate training set D_C and E_V is the MSE over the validation set D_V. A network overfits when the value of ρ increases substantially above 1.

3. The computational complexity of learning algorithms was also used as performance criterion. For the purpose of this thesis, computational cost is measured as the number of calculations needed to train the network. Calculations include subtraction, multiplication, addition and division.

   At any epoch ξ, the cost C_{fc} of training a NN on a training set, is expressed as

$$C_{fc} = (C_V + C_W) * P_T$$

   where C_V is the cost of updating weights between input and hidden units and C_W is the cost of updating weights between hidden and output units. P_T is the number of patterns in the training subset D_T. For conventional backpropagation with fixed set learning, $P_T = P_C$. Thus the cost of training C_{fsl} is computed as $C_{fsl} = (C_V + C_W) * P_C$.

   The costs of updating the weights are calculated as

$$\begin{aligned} C_V &= C_v * (N_V) \\ C_W &= C_w * (N_W) \end{aligned}$$

   where C_v is the cost of updating a single weight between the input and hidden layers, C_w is the cost of updating a single weight between the hidden and output layers. C_V is the total cost of updating the weight connections between the input and the hidden layers, and C_W is the total cost of updating the weight connections between the hidden and output layer. N_V is the total number of connections between the input and hidden layers and N_W is the total number of connections between the hidden and output layers.

   The total number of connections N_V and N_W are expressed as

$$N_V = (I + 1) * (J + 1)$$
$$N_W = (J + 1) * (K)$$

   and

$$C_v = 13$$

$$C_w = 11$$

Therefore,

$$\begin{aligned} C_V &= 13 * (I+1) * (J+1) \\ C_W &= 11 * (J+1) * K \end{aligned} \tag{3.8}$$

The cost of training a network using any active learning algorithm includes C_{fc}, the cost of selecting patterns for training and the cost of computing the subset termination criterion. Therefore, at any epoch ξ, the cost of training a network using SLA, SAILA, DPS and AL are:

$$\begin{aligned} C_{SL} &= C_{fc} + C_{sla} * P_C \\ C_{DP} &= C_{fc} + C_{dps} * (P_C - P_T) + (C_{S_{dps}} * P_T) \\ C_{AL} &= C_{fc} + C_{al} * (P_C - P_T) + (C_{S_{al}} * P_T) \\ C_{SA} &= C_{fc} + C_{sai} * (P_C - P_T) + (C_{S_{sai}} * P_T) \end{aligned}$$

For all the incremental learning algorithms, the subset selection criteria are tested on the remaining patterns in the candidate set D_C which is equal to $P_C - P_T$. Also, for incremental learning algorithms, an additional cost of selecting pattern is incurred when a pattern is selected.

C_{SL}, C_{SA}, C_{AL} and C_{DP} are the cost of training a network using SLA, SAILA, AL and DPS respectively. $C_{sla} = 15$ is the cost of computing the subset selection criteria for SLA, $C_{dps} = 11$ is the cost of computing the subset selection criteria for DPS, $C_{al} = 4$ is the cost of computing the subset selection criteria for AL and $C_{sai} = 18$ is the cost of computing the subset selection criteria for SAILA. $C_{S_{dps}} = 2$, $C_{S_{al}} = 2$ and $C_{S_{sai}} = 7$ are the cost of selecting patterns into D_T for DPS, AL and SAILA respectively.

Therefore,

$$C_{SL} = C_{fc} + 15P_C$$

$$\begin{aligned}
C_{DP} &= C_{fc} + 11(P_C - P_T) + 2P_T \\
C_{AL} &= C_{fc} + 4(P_C - P_T) + 2P_T \\
C_{SA} &= C_{fc} * P_T + 18(P_C - P_T) + 7P_T \qquad (3.9)
\end{aligned}$$

From equation (3.9), the cost of training is directly proportional to the number of patterns selected for training. The more patterns are selected for training, the higher the computational cost. Initially, P_T for SLA is greater than the other algorithms because DPS, AL and SAILA are incremental learning algorithm and a small initial trainig set and subset size is used in the simulations. Thus, C_{SL} is expected to be greater than C_{AL}, C_{DPS} and C_{SA} initially. SAILA is computationally more expensive in selection criteria than the other algorithms because SAILA has more subset selection criteria to implement than the other algorithms.

Section 3.5.2 illustrates the costs for the different algorithms.

3.5.2 Results

This section presents the results of the simulations carried out on the active learning algorithms.

Training error

In order to compare the performance of the four active learning algorithms, the MSE over the candidate set D_C was computed for the simulations and the average calculated. Tables (3.2) and (3.3) show the result over 2000 epochs for clean data and data with noise and outliers.

For TS1, DPS had a very low error with the lowest variance which means that all the errors of the simulations for DPS were all closer to the average error of 0.0003. Although, SLA had a low error as well. However SLA had a large variance when compared to DPS. AL had the largest error with a very large variance.

DPS achieved the smallest error for TS2, having a small variance. For TS3, all the algorithms had very low errors but SAILA had a high variance. DPS had the smallest error for F1 with a very small variance.

For TS4 and TS5, SLA achieved the smallest error with the lowest variance. AL had the largest error for TS4 and TS5. This is because AL selected and trained on just a single pattern for TS4 and an average of 4 patterns for TS5. Thus AL. had high errors for TS4 and TS5.

The training errors for all the problems with noise and outliers were larger ($\times 10^2$) than for problems with clean data. DPS had the lowest average error for clean data while SLA had the lowest error for noisy data.

Generalization error

To compare the generalization ability of the four active learning algorithms, the MSE over the generalization set, E_G, was computed and the average over the 30 simulations was plotted as a function of number of epochs. Figures 3.3 and 3.4 illustrates the trend of the generalization errors for the entire training period.

DPS achieved a very low average error faster than the other algorithms for F1 (refer to figure 3.3(a)). However, both SAILA and AL achieved a comparable result to DPS at epoch 1000. From table 3.3, DPS had the lowest error with the a very small variance ($5.07E - 05$) which means all errors of the simulations are closer to the average.

For TS1, SAILA initially had the highest generalization error but decreased to a low level of error (see figure 3.3(b)). SLA initially had the lowest average error, which can be explained by the fact that SLA used more patterns initially than the other algorithms (refer to figure 3.7(b)). Although SLA and DPS had small errors, DPS had the smallest variance and thus DPS achieved the smallest error. AL had the largest error after 2000 epochs with a large confidence interval.

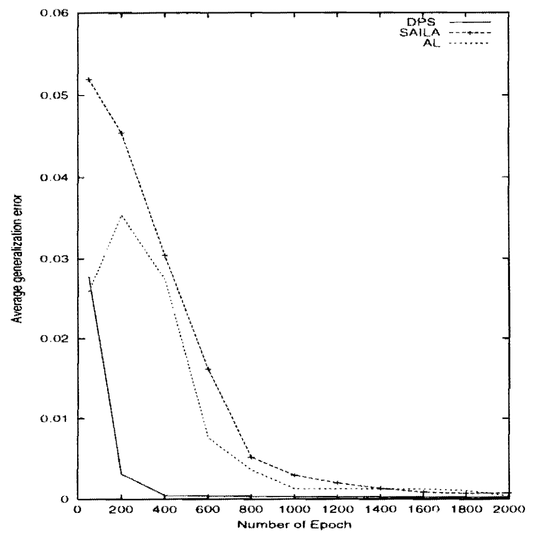
For TS2, DPS, AL and SLA achieved a very low average error before epoch 500.

Function	**Röbel**	**Zhang**	**Selective Learning**	**Sensitivity Analysis**
TS1				
Training Error	0.00036 ± 0.00017	0.02172 ± 0.04186	0.00045 ± 0.00035	0.00346 ± 0.00712
Generalization	0.00039 ± 0.0002	0.02241 ± 0.04191	0.00047 ± 0.00040	0.0035 ± 0.00737
Used Patterns	485.43 ± 234.79	4.73 ± 0.92	270.93 ± 3.48	571.67 ± 88.91
TS2				
Training Error	0.00014 ± 0.00011	0.00023 ± 0.00021	0.00029 ± 0.00038	0.00126 ± 0.00163
Generalization	0.00012 ± $0.25E-05$	0.00022 ± 0.00019	0.00029 ± 0.00037	0.00129 ± 0.00169
Used Patterns	411.77 ± 215.87	174.63 ± 61.48	272.57 ± 7.61	522.57 ± 173.37
TS3				
Training Error	0.00039 ± 0.00086	0.00044 ± 0.00091	0.00050 ± 0.00085	0.00068 ± 0.00146
Generalization	0.00275 ± 0.00155	0.00253 ± 0.00133	0.00302 ± 0.00138	0.00225 ± 0.00174
Used Patterns	180 ± 0	180 ± 0	78.17 ± 1.53	180 ± 0

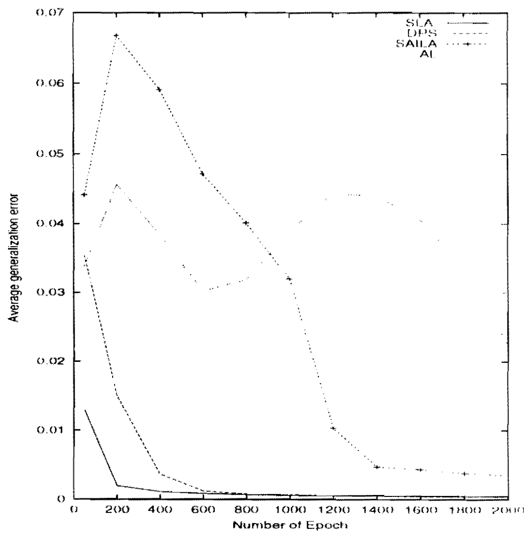
Table 3.2: Comparison results over 2000 epochs for times series problems

Function	Röbel	Zhang	Selective Learning	Sensitivity Analysis
F1				
Training Error	0.000226 ± $5.07E-05$	0.000412 ± 0.000366		0.000791 ± 0.001852
Generalization	0.000221 ± $5.2E-05$	0.000392 ± 0.000347		0.000764 ± 0.001624
Used Patterns	320.2 ± 167.6698	82.8 ± 32.37935		445.1333 ± 121.476
TS4				
Training Error	0.01141 ± 0.00573	0.19935 ± 0.03093	0.00516 ± 0.00393	0.02828 ± 0.09522
Generalization	0.01077 ± 0.00534	0.19051 ± 0.02169	0.00478 ± 0.00349	0.02739 ± 0.09170
Used Patterns	493.03 ± 193.37	1 ± 0	245.23 ± 7.89	597.03 ± 27.41
TS5				
Training Error	0.00683 ± 0.00468	0.10278 ± 0.08731	0.00155 ± 0.00158	0.00562 ± 0.00768
Generalization	0.00714 ± 0.00489	0.09904 ± 0.08932	0.00158 ± 0.00142	0.00595 ± 0.00842
Used Patterns	103.5 ± 20.14	4.67 ± 1.35	269.13 ± 9.89	584 ± 93.4

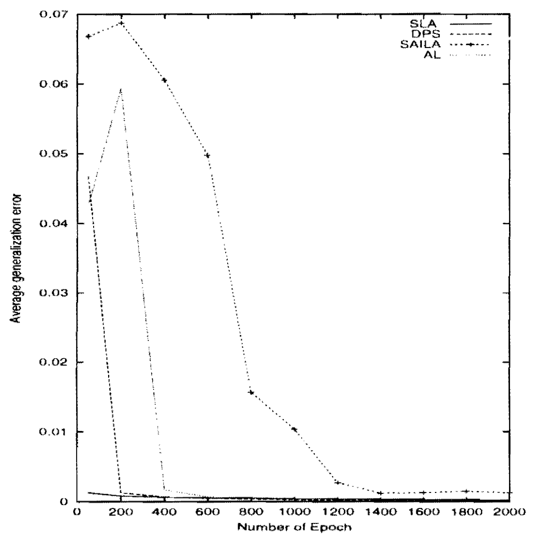
Table 3.3: Comparison results over 2000 epochs for problems F1 and times series with noise and outliers



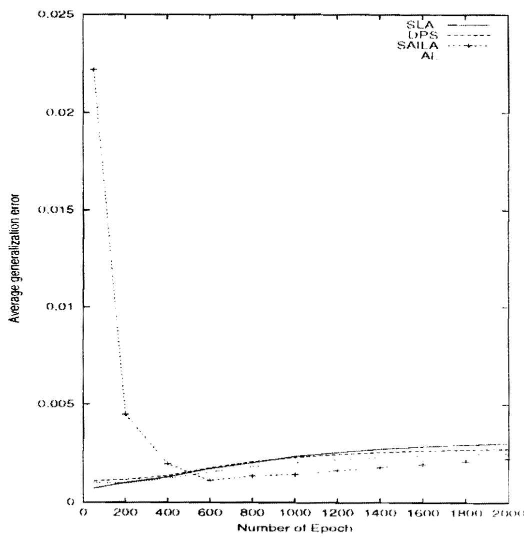
(a) Function F1



(b) TS1

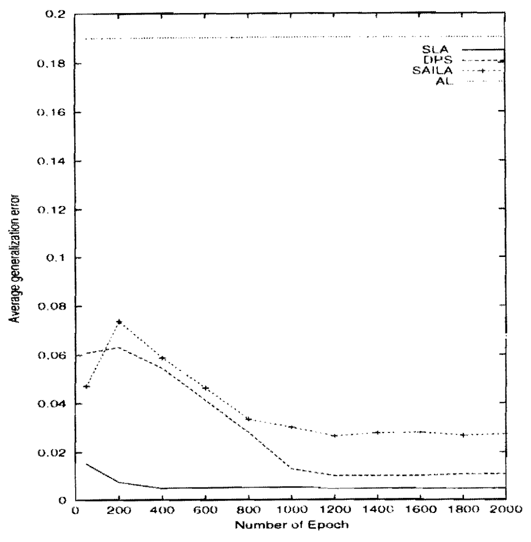


(c) TS2

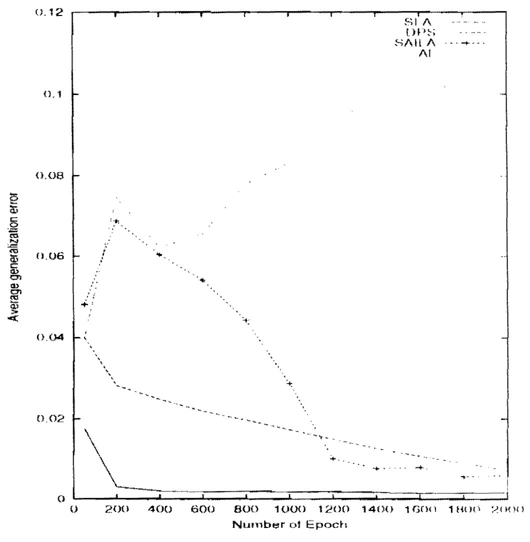


(d) TS3

Figure 3.3: Average generalization error vs epoch



(a) TS4



(b) TS5

Figure 3.4: Average generalization error vs epoch

SAILA was slower to achieve a comparable low error but SAILA had a low error by the end of training. From the table 3.2, DPS had the smallest error with a very small variance after 2000 epochs, implying that all errors of the simulations are closer to the average.

For TS3, the generalization error for all the algorithms increased as the number of epochs increased except SAILA (see figure 3.3(c)).

From the table 3.3, SLA had the lowest generalization errors for TS4 and TS5. While AL had the largest error for TS4 and TS5. AL did not learn the functions (refer to figure 3.4(a)). AL selected a few patterns for training, thus had little information about the time series to be approximated and therefore AL had a bad generalization.

DPS had the lowest generalization errors for functions with clean data while SLA had the lowest generalization errors for functions with noise and outliers. Although DPS had better generalization with clean functions than SLA, DPS used more patterns than SLA to achieve the low generalization error in all the problems. AL had very large generalization errors for TS1, TS4 and TS5. This bad generalization can be attributed to the extremely small training set sizes used by AL which is an indication of an inferior subset selection criterion. The subset selection criterion depends on the number of connections in the network. Redundant or irrelevant weights in the network will make the value of the performance level κ very large which can cause the network to train on the current training subset D_T too long without selecting additional patterns. Thus the network selects a few patterns, hence having insufficient information to train the network. On the other hand, too few weights in the network can make κ small. Thus, the network selects patterns more often than are needed for training.

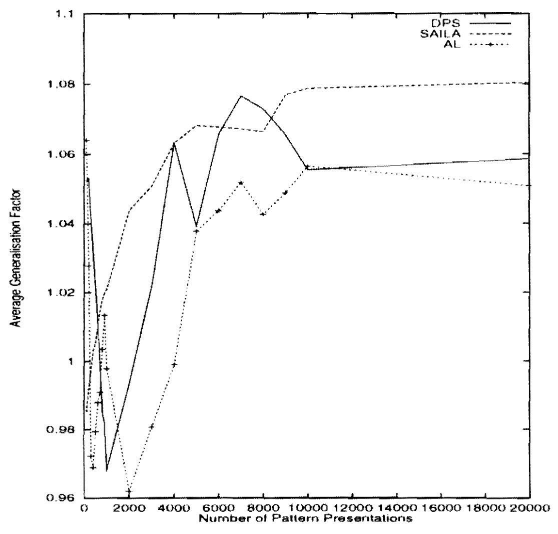
Overfitting effects

The average generalization factor ρ for all the problems were computed over the 30 simulations. Figures 3.5 and 3.6 show the charts for the average generalization factors. The average generalization factors were plotted as function of pattern presentations. A pattern presentation represents one weight update.

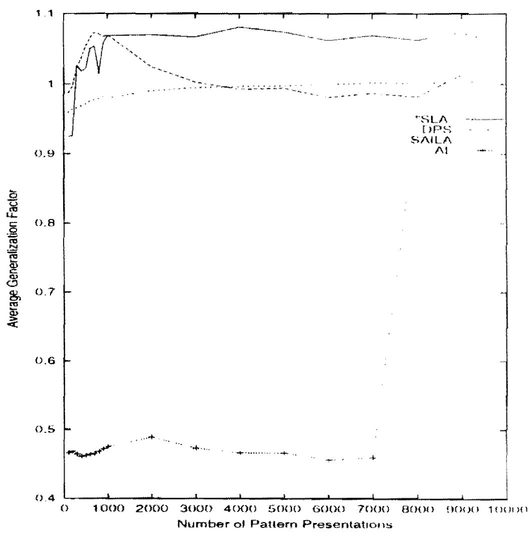
TS3 was the only function for which all the algorithms except SAILA, overfitted. SAILA had an average generalization factor of less than one, while the other algorithms had high generalization factors. For the entire training period for TS4, AL had a generalization factor constantly larger than 1, indicating that AL overfitted TS4. For the other functions, the average generalization factor values fluctuated. The flunctuation is due to the overfitting of a training subset until new patterns are selected for training. When new patterns are selected, the overfitting of the training subset is reduced. The average generalization factor for all the algorithms (except TS3) were slightly over one, and indicating a mild case of overfitting.

Computational costs

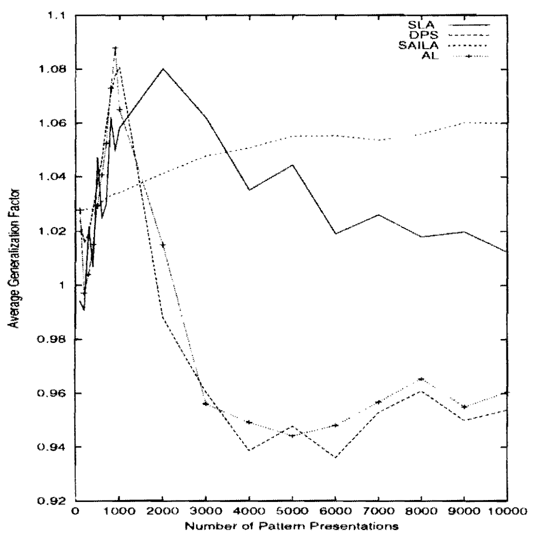
The computational costs for AL, DPS, SLA and SAILA were computed using e-quation (3.9) for specified epochs. The costs are plotted as a function of epochs as illustrated in figures (3.9) and (3.10).



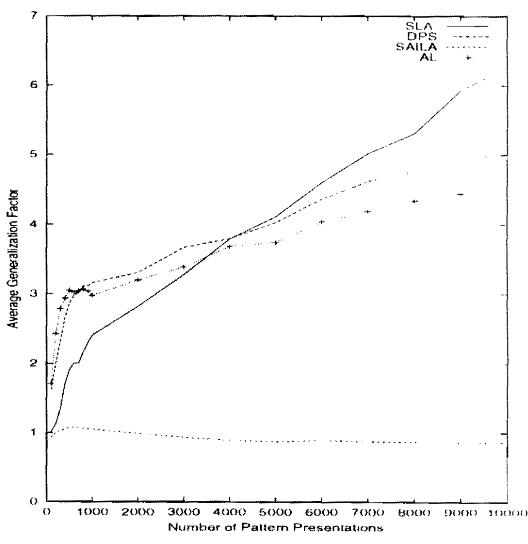
(a) Function F1



(b) TS1

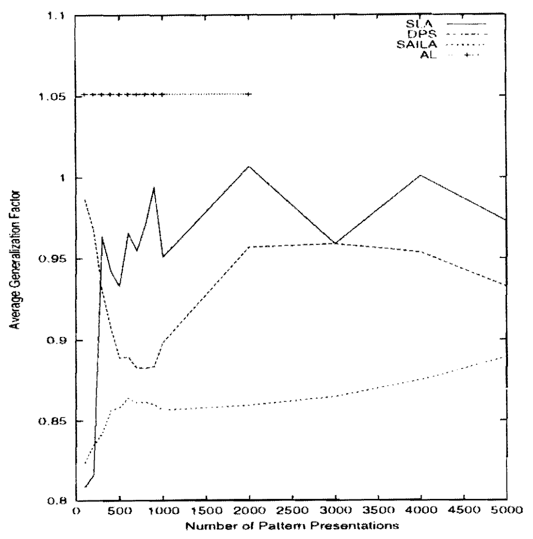


(c) TS2

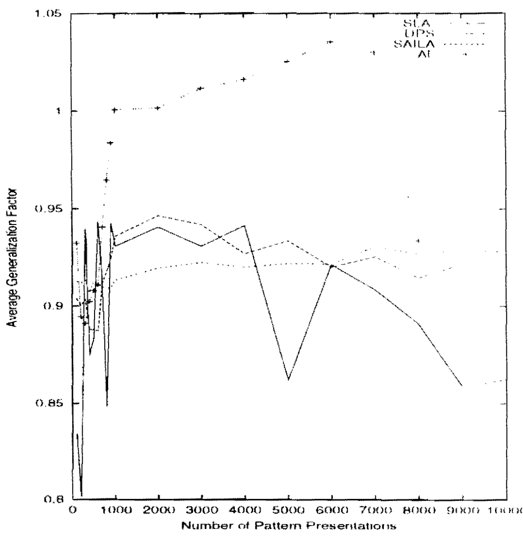


(d) TS3

Figure 3.5: Average generalization factor vs pattern presentations



(a) TS4



(b) TS5

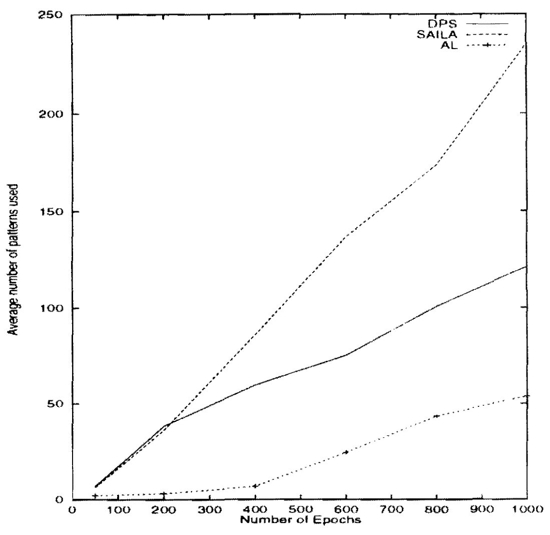
Figure 3.6: Average generalization factor vs pattern presentations

SAILA has the most expensive and AL has the least expensive subset selection criteria. However, AL performed badly (refer to tables 3.2 and 3.3). For all the approximation problems, DPS, AL and SAILA had increasing costs because they are incremental learning algorithms. More patterns were used as training progressed (refer to figures (3.7) and (3.8)).

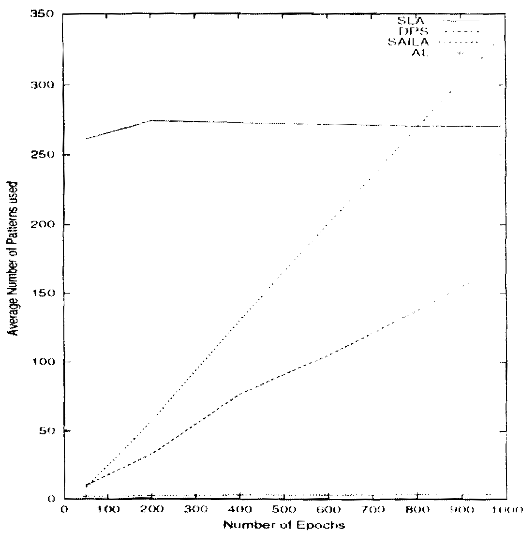
For F1 and TS2, AL had the smallest cost (see figure 3.9(a) and (b)). These small costs can be attributed to the cheap cost of the subset selection criterion as well as the fact that AL used the smallest number of patterns for training.

Despite the fact that AL has the cheapest subset selection criterion and a simple selection criterion, AL had the highest cost for TS3. This is because AL selected all the patterns in D_C within a short training interval (by epoch 400). SLA initially had the highest cost (first 350 epochs). However, at epoch 1000, the cost was half of the other algorithms.

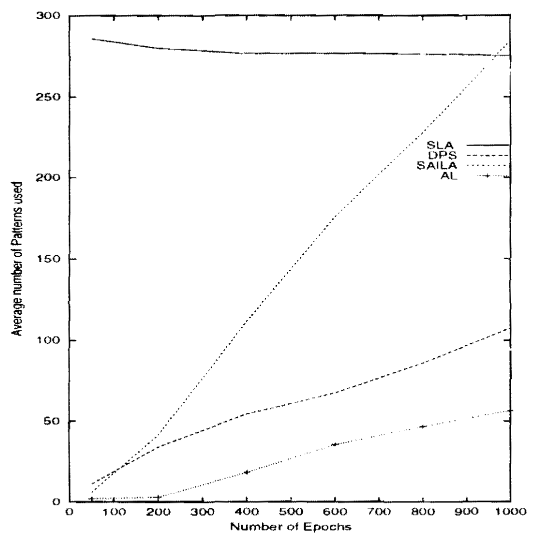
For all the functions approximated, SLA initially had a higher training cost than the other algorithms - almost four times the training cost of other algorithms, because



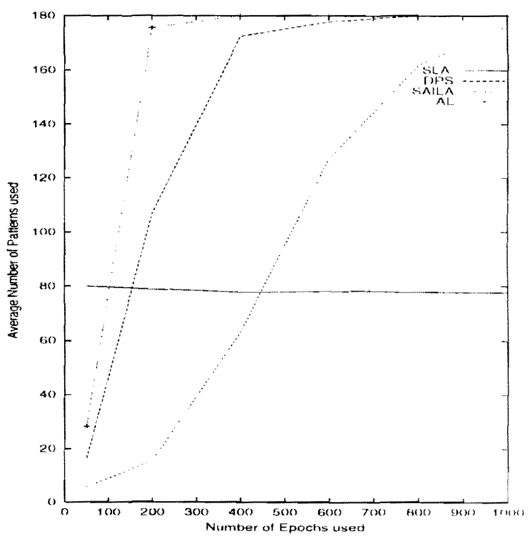
(a) Function F1



(b) TS1

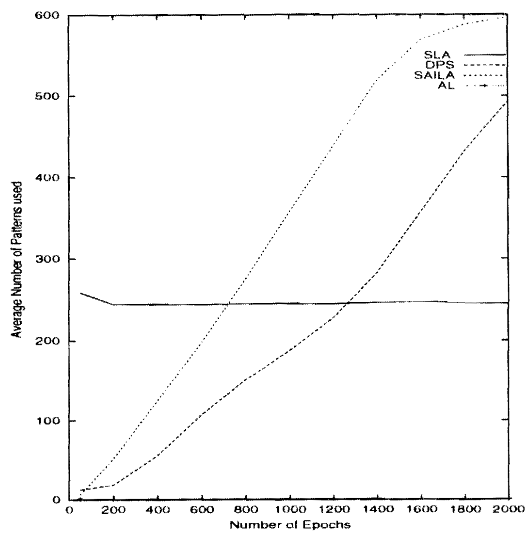


(c) TS2

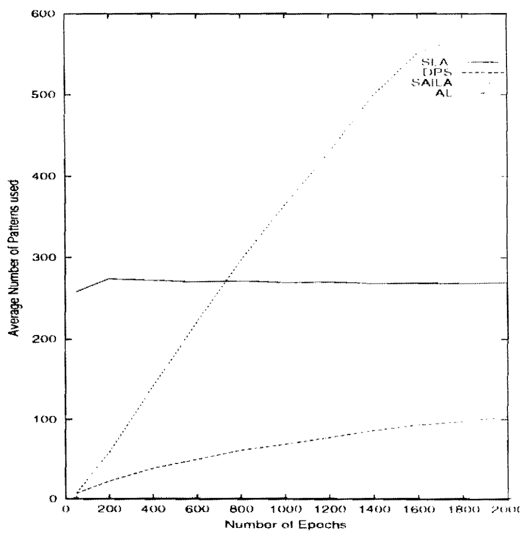


(d) TS3

Figure 3.7: Average number of patterns used per epoch



(a) TS4



(b) TS5

Figure 3.8: Average number of patterns used per epoch

SLA is a selective approach (see figure (3.9) and (3.10)).

From tables (3.2) and (3.2), SLA used less number of patterns after 2000 epochs than the other algorithms, thus SLA was computationally less expensive.

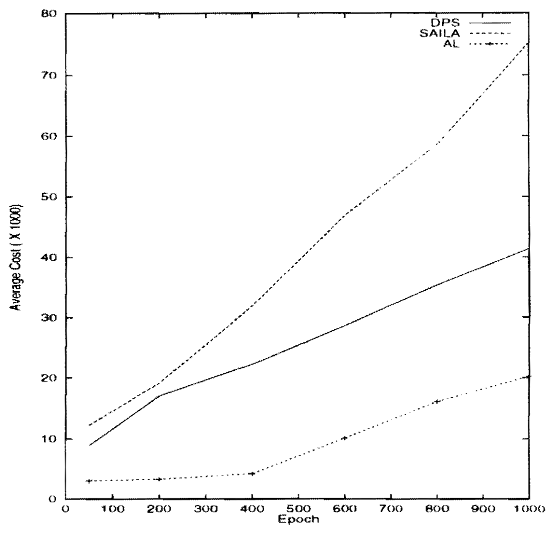
Convergence

The convergence performance of the four active learning algorithms are compared in figures 3.11 and 3.12. These figures plot the percentage of simulations that reached specific generalization errors.

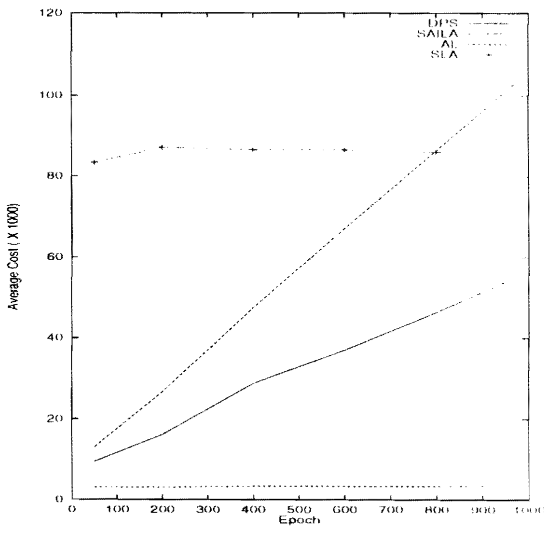
For F1, DPS had the best convergence, all the simulations converged to a very low error of 0.0004. AL also had a good convergence, more than half of the simulations converged to 0.0004 (refer to figure 3.11(a)).

None of AL's simulations converged to the specified error level for TS2. SLA and DPS achieved good convergence for TS2, as more than half of their simulations converged to a low error (refer to figure 3.11(b)).

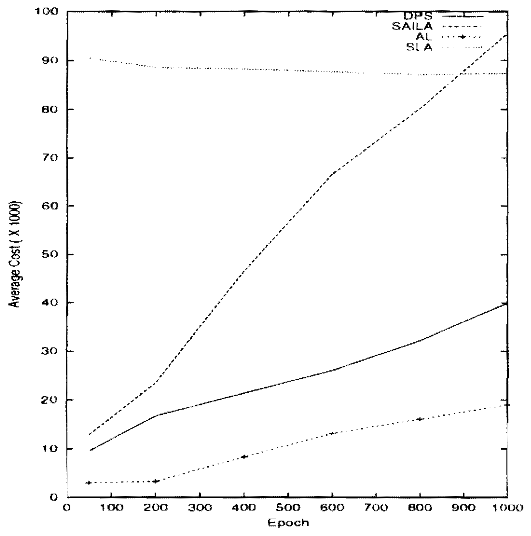
For TS2, DPS had the best generalization, most of all the simulations converged to a



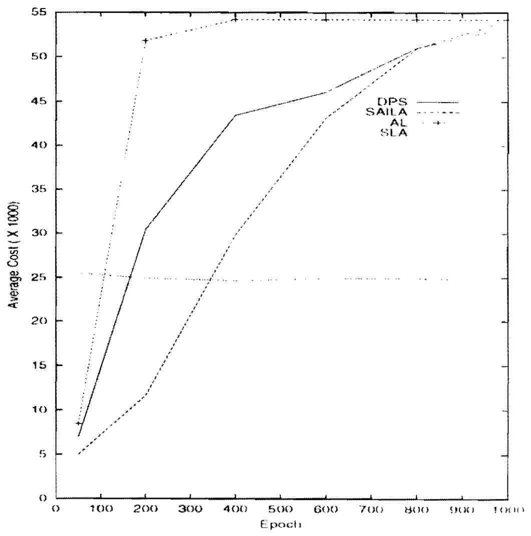
(a) Function F1



(b) TS1

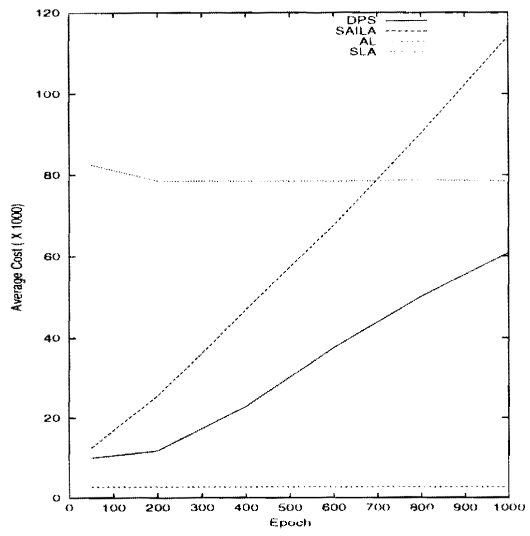


(c) TS2

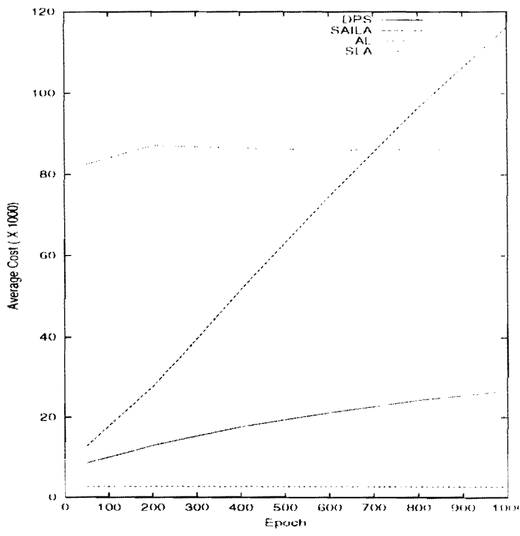


(d) TS3

Figure 3.9: Average computational cost per epoch



(a) TS4



(b) TS5

Figure 3.10: Average computational cost per epoch

very low error (0.0002). SLA and AL also had good convergence (see figure 3.11(c)). While the other algorithms had few converged simulations at 0.002, almost half of SAILA's simulations converged to this error (refer 3.11(d)).

AL had bad generalization for TS4 and TS5. None of AL's simulations converged to the specified error levels for TS4 while only a few converged for TS5.

SLA had the best generalization for TS4, with all the simulations converging to a low error of 0.004 (refer to figure 3.12(a)). SAILA had a good convergence with 40% of the simulations converging to this error of 0.004. Only a few of DPS's simulation and none of AL's simulation converged at this point.

SLA also had the best generalization for TS5. Almost all the simulations (74%) converged to a error level of 0.005 while only a few of the other algorithms simulations converged to this error level (see figure 3.12(b)).

SLA had the best convergence for data with outliers and noise. DPS had the best convergence for clean data, although SLA had good convergence for clean data. SAILA had a good convergence for TS3. For all the sine functions (TS1, TS4 and TS5), AL had bad convergence, none of its simulations converged to the specified error level-s. The errors specified for data with outliers and noise were larger than the errors specified for clean data. This is because the performance of all the algorithms were degraded in the presence of noise and outliers.

3.6 Conclusion

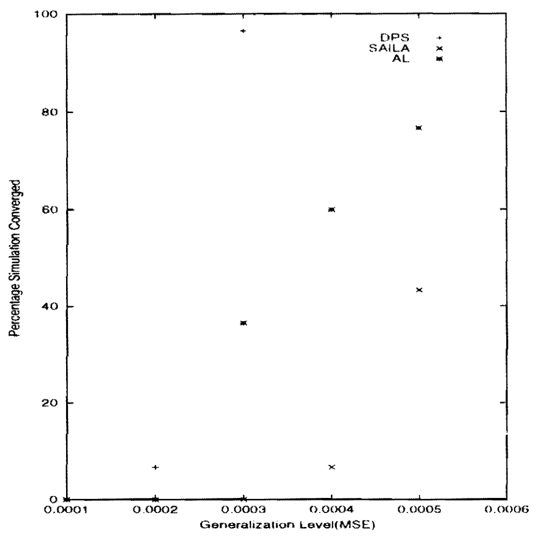
The objectives of the chapter were to present a new learning algorithm (SLA) and also to compare four active learning algorithms with respect to their accuracy, convergence and the complexity on both clean and noisy data as well as overfitting effects for the problems were also examined.

The results presented showed that AL was unstable, producing good results for the henon-map and F1 only. The bad training behavior can be attributed to the extreme-ly small training set sizes used by AL, which is an indication of an inferior subset selection trigger.

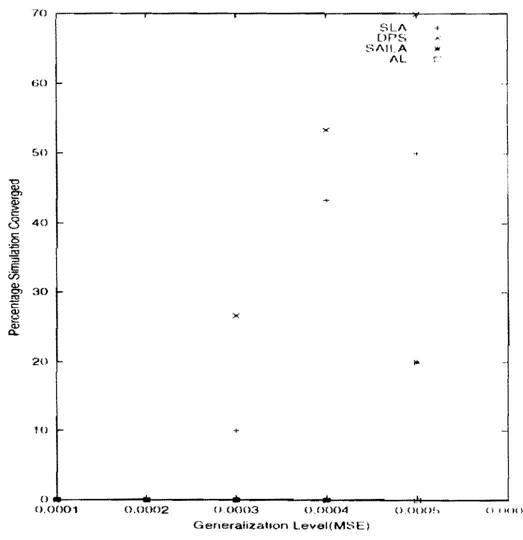
DPS and SLA performed very similar on the clean data, while SLA outperformed all the other algorithms on the noisy and outliers training data. The sensitivity analysis approach (SAILA) performed well under the occurrence of outliers and noisy time series, and very well for the complex function TS3. SAILA performed better than AL in TS1, TS4 and TS5, but worse than SLA and DPS. SAILA is computationally more expensive, requiring larger training subsets than the other algorithms.

As is expected, the performance of the error selection approaches degraded under the occurrence of outliers and noise. The degradation is due to the early selection of outliers, since outliers result in the largest prediction errors.

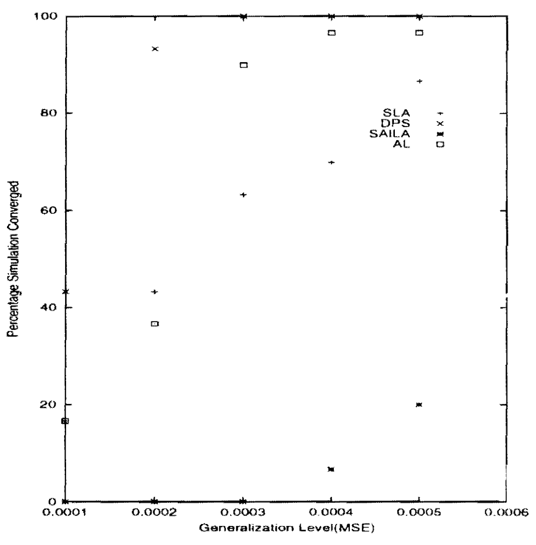
The comparison above showed that SLA had the best generalization performance, and lowest complexity. The selective learning approach (SLA) produced better accuracy



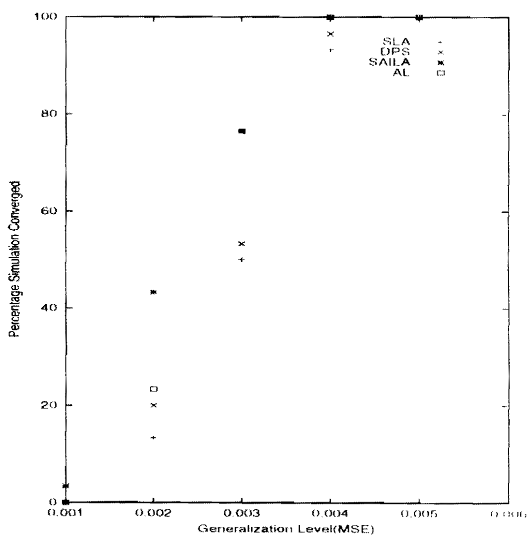
(a) F1



(b) TS1

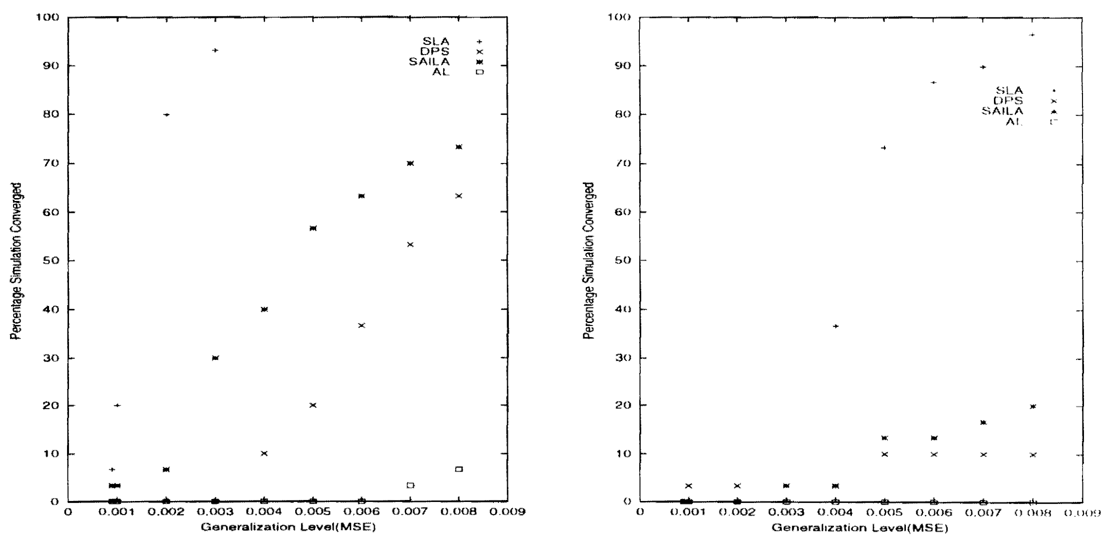


(c) TS2



(d) TS3

Figure 3.11: Percentage simulations that converged



(a) TS4 (b) TS5

Figure 3.12: Percentage simulations that converged

than the other approaches, and showed to be more robust in the occurrence of outliers and noise.