IMPROVED SEQUENTIAL AND BATCH LEARNING IN NEURAL NETWORKS USING THE TANGENT PLANE ALGORITHM

PAUL MAY

A thesis submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

University of Bolton

June 2012

ACKNOWLEDGEMENTS

I would like to thank Dr. Charles Lee for his kind patience and numerous helpful comments over the past years. Our frequent dialogues have helped me to develop as a research degree student. These lively discussions will be greatly missed. I would also like to thank Dr. Erping Zhou for her advice and guidance in the later years of the research. Finally I would like to thank my wife and son for the many years of sacrifice and for their patience and understanding without which this great journey would not have been possible

ABSTRACT

The principal aim of this research is to investigate and develop improved sequential and batch learning algorithms based upon the tangent plane algorithm for artificial neural networks. A secondary aim is to apply the newly developed algorithms to multi-category cancer classification problems in the bio-informatics area, which involves the study of dna or protein sequences, macro-molecular structures, and gene expressions.

The major contributions of this thesis are summarised as follows. In the first part of this thesis, sequential and batch learning algorithms based on the tangent plane algorithm are investigated

- The tangent plane algorithm (TPA) is investigated and compared with the back-propagation algorithm for three neural network benchmark tasks. The principal strength of the tangent plane algorithm is that it does not require manually tuning a learning rate parameter, but instead automatically adjusts the learning rate to give the correct step size. The algorithm has been further modified to accept almost zero starting conditions with the expectation that only the minimum number of weight necessary will be activated during the training phase. The results show that the tangent plane algorithm gives improved generalization relative to the back-propagation algorithm, and that generalization is independent of network size. The limitations of the tangent plane algorithm are also identified

- A new sequential algorithm is developed referred to as the tangent plane algorithm for real time recurrent learning (TPA-RTRL), targeted at improving the stability of the tangent plane algorithm when handling inexact data. The new algorithm is evaluated and compared with the original gradient descent real time recurrent learning (GD-RTRL) algorithm for two sequence recognition tasks. It is shown that using the new TPA-RTRL algorithm to train a fully recurrent neural network with feedback connections and context units is more stable than using the GD-RTRL algorithm, especially when the training data has been corrupted with a small amount of erroneous data.

- A new sequential algorithm referred to as the improved tangent plane algorithm (iTPA) is developed to further improve the generalization performance of second tangent plane algorithm. This new algorithm is evaluated and compared with the original algorithm and the back-propagation algorithm for three neural network benchmark tasks. The results show that moving along tangent planes in a direction that encourages weight elimination improves generalization performance. The results also show that including a tendency to move laterally in random directions along tangent planes helps the algorithm to avoid local minima of the error landscape.

- A new batch algorithm referred to as the Gauss-Newton tangent plane algorithm (GN-TPA) is developed for training small economical networks. This new algorithm uses a modified Gauss-Newton vector to guide the search toward the minimum training error. Another improvement is using a novel neural network structure recently described in the literature as the Extreme Learning Machine (ELM). The new algorithm is evaluated and compared with three newly developed network building techniques for three neural network benchmark tasks. The results show that the new GN-TPA algorithm is very fast and efficient.

In the later part of this thesis, the newly developed sequential and batch tangent plane algorithms are applied to real world classification tasks that have proven difficult for more conventional neural network techniques to solve. Multi-category cancer classification using gene expression profiles is a difficult task to solve due to the high dimensionality of the data. Traditionally this is done by combining binary classifiers in one-versus-one (OVO) or one-versus-all (OVA) schemes, which inevitably involves increasing the system complexity. Direct classification using artificial neural networks has been attempted but classification accuracy is known to drop sharply with increasing number of classes. The results show that the classification accuracy of the newly developed learning algorithms are comparable with the best learning algorithms found in the literature on two benchmark gene expression datasets.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

1.1 Motivation

The back propagation algorithm [1] is an iterative procedure for training the weights of a multi-layered feed-forward neural network to minimise a given error (or cost) function, typically the sum of square error. Geometrically, the error function defines a surface over weight space. At each iteration of the procedure, the weights are adjusted in the direction in which the error surface decreases most rapidly. The direction is given by the negative gradient of the error surface at the current point in weight space. The magnitude of the modification is given by the magnitude of the gradient and a positive constant called the learning coefficient. If the learning co-efficient is too large, then the movement in weight space will become oscillatory and the algorithm will fail to converge or convergence will be very slowly. On the other hand, if the learning co-efficient is too small then the algorithm will converge slowly resulting in long training times. Lee [2] proposed an alternative approach to training multilayered feed-forward neural networks that does not require setting the value of the learning coefficient. This tangent plane algorithm treats each teaching value as a constraint on the network weights that defines a surface in weight space. The weights are adjusted by moving from the current position to the tangent plane to this surface. Convergence is rapid because the step-size is determined solely by the Euclidean distance from the current state of the weights to the foot of the normal to the tangent plane. Unfortunately, the tangent plane algorithm will often fail to converge when a small amount of noise is added to the teaching values. Therefore, methods to improve the stability of the tangent plane algorithm are studied in this thesis. These methods include using fully recurrent neural networks that are capable of predicting a time step ahead the correct response to an item of data received previously by the network [3,4,5]

A principal concern in supervised training of neural networks is how to obtain good generalization. A network is said to generalize well when the input-output mapping computed by the network is the same for test data not used in creating or training the network [6]. It is known that with back-propagation learning, generalization is better in smaller networks. This is because the shortage of units forces the network to develop general rules to discriminate between input patterns. Unfortunately, it can be difficult to determine the optimum size of the network in advance without knowing the exact rules to be extract from the training data. These difficulties led to the development of a number of techniques to determine the optimum size for good generalization. To limit the size of a network, you can either use additive, subtractive, or regularization techniques. Additive techniques start with a small network, and insert new units and connections until the network has the right size [7,8,9]. Subtractive techniques (often called pruning) start out with a fully trained network and remove superfluous connections [10,11]. Regularization techniques use a network with a large number of connections but limit the size of each weight [12,13]. Lee [14] suggested growing the weights from small initial values close to zero with the expectation that only the minimum number of weights would be activated. This second tangent plane algorithm is essentially the same as the first tangent plane algorithm, but includes an additional term that pushes the weights in the direction away from the origin. Unfortunately, the second tangent plane algorithm tends to create large network structures that have a wide distribution of weight values. This means that any advantage gained by starting the training with initial conditions close to zero is soon lost. Therefore methods to improve the generalization of the second tangent plane are studied in this thesis

The convergence speed of the tangent plane algorithm is no better than the standard back-propagation algorithm in small parsimonious networks where generalization is found to be best. In small networks with only a few synaptic weights, it seems that updating the weights by approaching tangent planes would be a very slow, as one big weight update might actually corrupt the whole of the network. In the batch mode of learning the weights are updated after the presentation of the entire dataset. Collecting all the gradient information together before the weights are updated helps to avoid the mutual interference of weight changes that could occur with large learning rates in the sequential (or online) learning. This makes the batch learning particularly suitable for training small neural networks. An alternative approach to batch learning might be to take smaller steps in weight space, the smaller steps averaging out the variations in the data so that the weights follow a more clearly defined trajectory in weight space. Unfortunately reducing the step size leads to slow adaptation of the weights so training speeds can be very slow. Furthermore iterative methods that take smaller steps are prone to being trapped in local minima of the error landscape. Therefore a new batch implementation of the tangent plane algorithm for training small parsimonious networks is investigated in this thesis

Multi-category classification problems in the bio-informatics area are known to be particularly difficult problems to solve. There are a number of reasons for this. Firstly, biological data based on micro-array gene expression profiling has a very high input dimension, which is typically thousands of genes. Many of these genes are irrelevant to cancer classification. These irrelevant genes not only increase the complexity of the neural network, but also add noise to the training data which compromises generalization. Secondly, the size of the data available is usually very small, typically less than 100 samples. It is well known that a low ratio of the sample size to the input dimension produces a sparse input data space, which also leads to poor generalization. The traditional method for solving multi-category classification problems is to combine several binary classifiers, but this produces a heavy computational overhead. This means that fast and efficient algorithms are needed. The new sequential and batch tangent plane algorithms are particularly suitable for this purpose as they are very fast and avoid problems like overfitting and local minima. Therefore the new sequential and batch implementations of the tangent plane algorithm are applied to multi-category classification based on gene expression data

1.2 Objectives

The primary objectives of this research are to improve the convergence speed, stability and generalization of the tangent plane algorithm. More specifically, the objectives of this research can be summarised as follows

- Develop a new algorithm for fully recurrent neural networks targeted at improving the stability of the tangent plane algorithm. One example of where stability is an issue with the tangent plane algorithm is when the training data is contains a small amount of rogue data or errors. Therefore one objective of this thesis is to develop a new algorithm capable of accepting a small percentage of erroneous data in the training set and recovering quickly after it has been perturbed in this way
- Develop a new algorithm for neural networks based on the second tangent plane algorithm giving improved generalization relative to the original algorithm. The second tangent plane algorithm tends to create large network structures that have a wide distribution of weight values. Less important weights have a tendency of taking on completely arbitrary values that might actually degrade generalization. Therefore another objective of this thesis is

to develop a new algorithm giving improved generalization and evaluate its performance on non-trivial problems.

- Develop a new batch tangent plane algorithm for small parsimonious networks. The tangent plane algorithm is a fast method of training a neural network that does not require any parameters to be set manually to tune its performance. This is the principal strength of the tangent plane algorithm. However the tangent plane algorithm is no better than the back-propagation algorithm when applied to small economical networks where generalization is known to be best. Therefore another objective of this thesis is to develop a new algorithm for small economical networks capable of fast convergence and good generalization
- Apply the new sequential and batch algorithms developed in this thesis to multi-category classification tasks that have proven difficult for more conventional neural network techniques to solve. Cancer classification using gene expression data is considered a difficult task because of the high input dimension, and multi-category classification is far more difficult than binary classification. The classification accuracy of ANN is known to drop sharply as the number of classes increases. Therefore fast and efficient algorithms are needed that are capable of high classification accuracy

1.3 Major contributions of the thesis

The major contributions of this thesis can be summarised as follows. In the first part of this thesis, three new algorithms are developed to overcome the difficulties with the tangent plane algorithm

- A new sequential algorithm referred to as the tangent plane algorithm for real time recurrent learning (TPA-RTRL) is developed for fully recurrent neural networks (FRNNs). This algorithm is evaluated for classification and time series prediction tasks. It is shown that the new TPA-RTRL algorithm is a very fast and stable method of training FRNN that recovers quickly when presented with items of erroneous data. This is because the FRNN recycles information over many time steps and thereby learns to predict the correct response a time step ahead, provided that an ordering of the input data exists.

- A new sequential algorithm referred to as the improved tangent plane algorithm (iTPA) is developed to further improve the generalization performance of the second tangent plane algorithm. This new algorithm is evaluated for pattern classification and function approximation tasks. The results show that implementing a weight elimination procedure into the geometry of the algorithm actually improves generalization performance by producing a separation of the active and inactive weights in the network. The results also show that including a tendency to move laterally in random directions along tangent planes helps the algorithm to avoid local minima of the error landscape.
- A new batch tangent plane algorithm referred to as the Gauss-Newton tangent plane algorithm (GN-TPA) is developed for training small parsimonious networks. This new algorithm uses the Gauss-Newton vector to guide the search of the error surface toward the minimum training error. In order to improve the convergence and efficiency of the new algorithm, a novel procedure recently described in the literature as the Extreme Learning Machine (ELM) is employed. The new algorithm is evaluated for pattern classification and function approximation tasks. The results show that GN-TPA reaches the minimum training error and avoids problems like local minima of the error landscape.

In the later part of this thesis, the newly developed sequential and batch tangent plane algorithms are applied to real world classification tasks that have proven difficult for more conventional neural network techniques to solve. Multi-category cancer classification using gene expression profiles is a difficult task to solve due to the high dimensionality of the data. Traditionally this is done by combining binary classifiers in one-versus-one (OVO) or one-versus-all (OVA) schemes, which inevitably involves increasing the system complexity. Direct classification using artificial neural networks has been attempted but classification accuracy is known to drop sharply with increasing number of classes

- The new sequential algorithm is combined with a one-versus-all scheme for multi-classification using gene expression data. A modular network is used with each segment trained to discriminate one class from all others. The results show that the new scheme can produce classification accuracies comparable with other newly developed sequential learning algorithms, SANN and FGAP-RBF.
- The new batch algorithm is applied to multi-category micro-array gene expression data. One-of-c encoding is used with each output unit trained to discriminate one class from all others. Results show that the GN-TPA algorithm gives high classification accuracy comparable with SVM-OVO, which is the best SVM classifier.

1.4 Organisation of the thesis

The first two chapters introduce the background of the thesis. The artificial neural network (ANN) as a structure for representing complex non-linear mappings is introduced together with different first and second order adaptive processes that enable them to learn about their environment. The concept of generalization is discussed with strategies for improving generalization. Finally a brief overview of the bioinformatics area with an emphasis on gene expression data is discussed.

Chapter three investigates the convergence and generalization behaviour of the tangent plane algorithm. Comparative tests are performed using the standard back-propagation algorithm. The benchmark datasets used were N-bit parity, breast cancer and hearta. Two problems with the tangent plane algorithm were identified, namely slow convergence in small networks and instability when handling inexact data. Finally the differentiation and evolution of weight in networks trained by the tangent plane algorithm was investigated

In chapter four, a new algorithm referred to as TPA-RTRL is developed for fully recurrent neural networks targeted at improving the stability of the tangent plane algorithm. This new algorithm is similar to the GD-RTRL algorithm, differing from it in the treatment of the output unit and in the use of a global learning rate. Suggestions were made to improve the computational complexity of the new algorithm. Comparative tests are performed on the new TPA-RTRL algorithm and the gradient descent based GD-RTRL algorithm. The benchmark datasets used were pipelined Xor, the simple sequence problem, and the Henon map.

In chapter five, a new algorithm referred to as iTPA is developed to overcome the difficulty with the second tangent plane algorithm, namely the tendency of the algorithm to produce large network structures. Comparative tests were carried out on the new iTPA algorithm and the backpropagation algorithm. The benchmark datasets used were two spirals, Henon map and housing price.

In chapter six, a new algorithm referred to as GN-TPA is developed for small parsimonious networks. Two difficulties with this new algorithm are identified, namely instability due to the step size overshooting the error minimum and the computational complexity of the algorithm.

In chapter seven, the new GN-TPA algorithm is applied to the Extreme Learning Machine in order to overcome the difficulties with the algorithm. An additive technique for growing a neural network is used to improve the computational efficiency of the new algorithm. Comparative tests are performed using two additive procedures, cascade and the orthogonal sequential training technique. The benchmark datasets used were additive function, the Henon map and housing price.

In chapter eight, two multi-category classification problems using gene expression profiling are described, GCM and Lymphoma, together with a gene selection method for reducing the number of genes required for accurate cancer classification. Comparative tests were carried out using three sequential learning algorithms, iTPA, SANN, and FGAP-RBF, and three batch learning algorithms, GN-TPA, ELM, and SVM-OVO.

Finally the conclusions and future work are summarised in chapter nine.

Chapter 2

LITERATURE REVIEW

The back-propagation algorithm is a popular method for training feed-forward multilayered neural networks. It is easy to implement and computationally simple. The principal disadvantage of this learning method is its relatively slow rate of convergence in practical situations. It also requires manual tuning by appropriate choice of learning and momentum rate parameters, a process which is carried out by trial and error. Since one of the advantages of a neural network is the ease with which they may be applied to novel problems, it is essential to consider automated and robust learning methods with good performance on many classes of problems. In this chapter, we review some first order and second order optimization techniques known to accelerate convergence. Some of these methods require the adaptation of the parameters of the learning algorithm, whilst others use second order information about the error surface

Artificial neural networks have been widely used in the fields of pattern classification and function approximation due to their adaptability and generalization capabilities, and unique power for non-linear mappings. Generalization is the property of a neural network to produce the correct response to data previously unseen by the network but similar in some sense to data on which the network has been trained. Good generalization performance is influenced by a number of factors such as the size of the training set, the size of the network, and whether the function to be learned is sufficiently smooth. This chapter reviews some techniques used to improve generalization in neural networks. These techniques include network building and pruning, weight regularization, principal component analysis and early stopping.

In the cancer classification area, micro-array gene expression profiling has attracted more attention than conventional techniques such as microscopic histology and tumour morphology due to recent advances in micro-array technologies. Gene expression profiling allows for the monitoring of thousands of gene expression levels in any cell, cell line or tissue. Hence it provides more information and more reliable classification accuracy. There have been many classification methods used for cancer classification. However, there are some characteristics of gene expression data that make them difficult tasks. In this chapter we review some of the methods used for cancer classification with an emphasis on multi-category classification using gene expression data

2.1 Artificial neural networks

Artificial neural networks (ANNs) are complex structures for representing non-linear input-output mappings [6,15]. Their development has been inspired by the biological structure of the human brain. The human brain is a complex non-linear parallel processing machine. It has the capacity to organise its structural components, known as neurons, so as to perform computations much faster than modern computers today. In much the same way, ANNs are composed of units called neurons that perform non-linear transformations on the input data. These neurons are connected together by synaptic weights to form different layers; one input layer, one output layer, and one or more hidden layers. Like the human brain, ANNs can acquire knowledge through learning, and that knowledge can be stored in the network. ANNs can be classified into two different classes according to the way information flows through the network e.g. feed-forward neural networks, and recurrent neural networks.

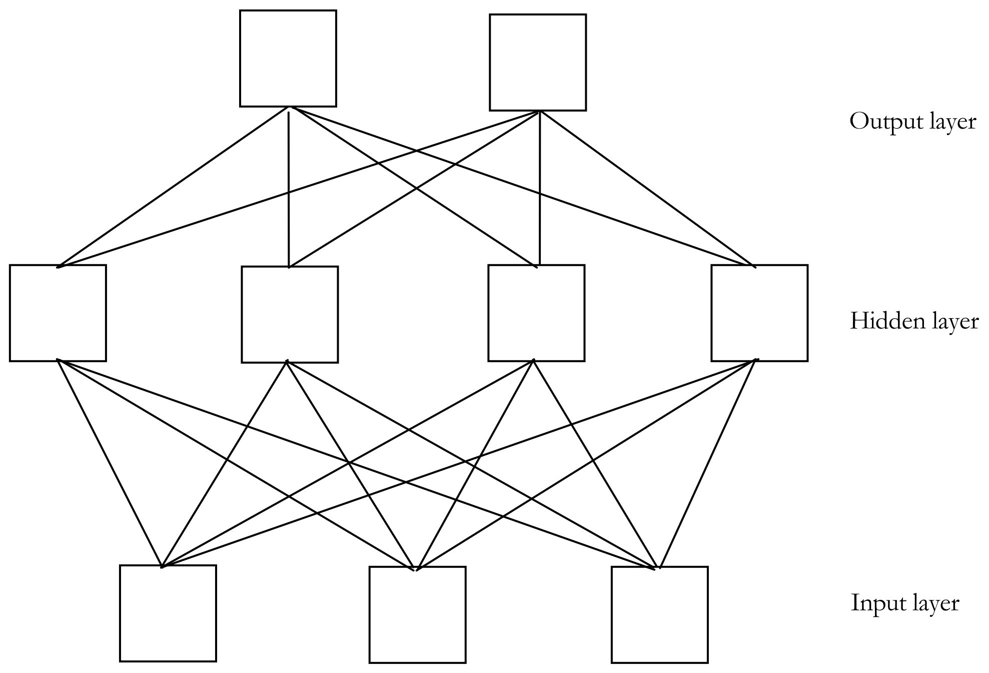


Fig 2.1. An example of a feed-forward neural network with one hidden layer

In feed-forward neural networks, the information flows in one direction from one layer to the next. The neurons in the input layer supply information, or activations, to the inputs of the neurons in the first hidden layer. The output signals of the neurons in the first hidden layer supply information, or activations, to the inputs of the neurons in the second hidden layer, and so on. Typically the neurons in each layer receive inputs from the output of the neurons in the preceding layer. Fig 2.1 illustrates the architecture of a multilayered feed-forward neural network (or multilayered perceptron). This network is referred to as a 3-4-2 network because it has 3 input neurons, 4 hidden neurons, and 2 output neurons. It is fully connected in the sense that every neuron in each layer is connected to every neuron in the next layer.

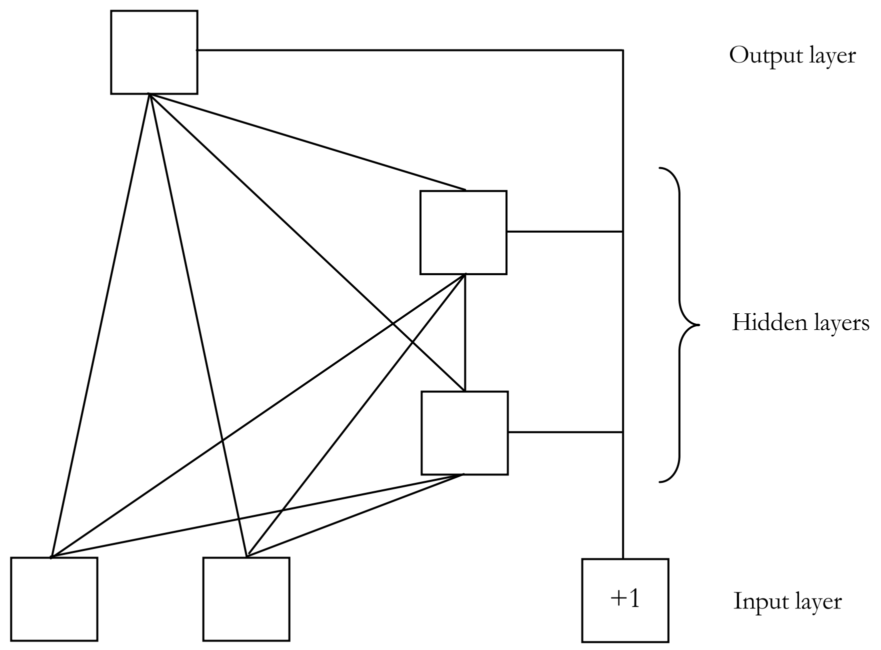


Fig 2.2. Cascade network architecture with three input units, two hidden units and one output unit

Another type of multilayered feed-forward structure has each neuron forming its own layer as illustrated in Fig 2.2. The cascaded neurons each receive an input from the neurons in the previous layers together with an input from the original network inputs, and pass their output to the neuron in the next layer. This cascade architecture was first proposed by Fahlman and Lebiere [7], and has been used successfully with many neural network problems that have proven very difficult for the standard back-propagation algorithm to solve [8,9,16,17]

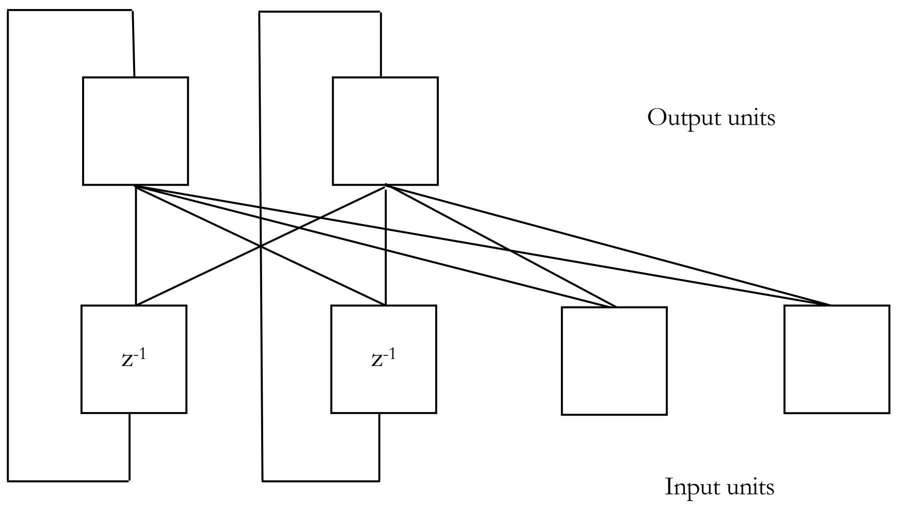


Fig 2.3. An example of a recurrent neural network with one layer. The function z^{-1} is the unit delay operator whose output is delayed with respect to the input by one time step i.e. $z^{-1}\left[x_j^{(k)}\right]=x_j^{(k-1)}$ where x_j is the jth input and k the time step.

Recurrent neural networks on the other hand distinguish themselves by comprising at least one feedback loop. For example, a recurrent neural network may consist of a single layer of neurons with each neuron feeding its output signal back to the inputs of the other neurons. The presence of a feedback loop has a profound effect on the learning capacity as well as its performance [6]. The feedback loops can involve the use of unit delay operators that can result in a non-linear dynamic behaviour. Thus, recurrent neural networks find greatest use in time series prediction problems [3,4,5].

2.2 Learning in artificial neural networks

A neural network learns about its environment by an adaptive process whereby a series of adjustments are made to the synaptic connections, or weights, and bias levels. Ideally, the network becomes more knowledgeable about its environment after each adaptive process. We define the learning process in the context of neural networks as follows [18]:

Learning is a process by which the free parameters of a neural network are adapted through a process of stimulation by the environment in which the network is embedded. The type of learning is determined by the manner in which the parameter changes take place.

A prescribed set of well-defined rules for the solution of a learning problem is called a learning algorithm. As one would expect, there is no single unique learning algorithm, but a diverse 'tool-kit' of algorithms each of which offers its own advantages. Basically, there are five different classes of learning rule: error-correction learning, memory-based learning, Hebbian learning, competitive learning, and Boltzmann learning. This thesis is primarily concerned with algorithms that belong to the error-correction class, specifically the tangent plane algorithm [2,14]. In error-correction learning, the input data is propagated forward through the network. The output of the network is then compared with the desired output and the error calculated. This error is then back-propagated through the network and used to adjust the weights so that the error decreases with each adaptive process.

2.2.1 First order methods and variants

The method of steepest descent is an iterative procedure for obtaining the values of the parameters that minimise the error (or cost) function. When applied to a neural network, this is equivalent to finding the values of the synaptic weights that connect the network units together. Geometrically, the function specifies an error surface defined over weight space. At each iteration of the steepest descent procedure, the weights are adjusted in the direction in which the error function decreases most rapidly. This direction is given by the negative gradient of the error function at the current point in weight space. The magnitude of the modification is proportional the magnitude of the error gradient. The procedure can be written

$$\Delta w_{ji}^{(n)} = -\eta \frac{\partial \varepsilon_k^{(n)}}{\partial w_{ji}^{(n)}} \qquad (2.1)$$

where Δw_{ji} is the change to the weight w_{ji} that regulates the connection from unit u_i to u_j, η is a positive constant called the learning rate, ε_k is the error function to be minimized, and n the time step. There are two main error functions, one is sum of square errors (SSE) and the other is the relative entropy function [19]

Whilst the steepest descent method can be an efficient means for obtaining the weight values that minimize the error function, it can be very slow to converge in practical situations. There are two reasons for this slow convergence [20]. First, the magnitude of the partial derivative of the error function may be such that modifying a weight will yield only a small change. This occurs where the error surface is fairly flat along a weight dimension, which will produce a small derivative. Alternatively, where the error surface is highly curved, the derivative is large in magnitude. Thus the value of the adjustment may overshoot the error minimum. Second, the direction of the negative gradient may not point towards the minimum of the error surface. This is illustrated in Fig. 2.4. The error surface is drawn topographically using constant error contours. The current weight vector is given by $\boldsymbol{w}^{(n)}$. Since the error surface is steeper along the $w_2^{(n)}$ weight dimension than the $w_1^{(n)}$ dimension, the derivative along this weight dimension will be larger.

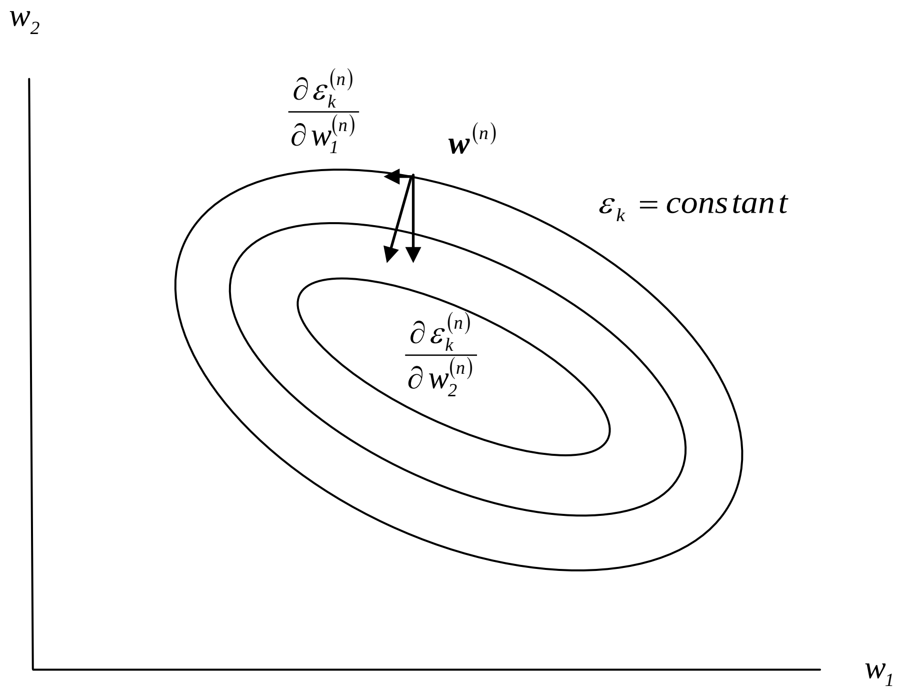

w_2
$\frac{\partial \varepsilon_k^{(n)}}{\partial w_1^{(n)}}$
$\boldsymbol{w}^{(n)}$
$\varepsilon_k = constant$
$\frac{\partial \varepsilon_k^{(n)}}{\partial w_2^{(n)}}$
w_1

Fig 2.4. Error surface defined over two dimensional weight space

A simple method of reducing the oscillations due to a large learning rate is to modify equation (2.1) as follows

$$\Delta w_{ji}^{(n)} = -\eta \frac{\partial \varepsilon_k^{(n)}}{\partial w_{ji}} + \alpha \Delta w_{ji}^{(n-1)} = -\eta \sum_{t=0}^{n} \alpha^t \frac{\partial \varepsilon_k^{(t)}}{\partial w_{ji}} \qquad (2.2)$$

where α is a positive constant called the momentum term, and $\Delta w_{ji}^{(n-1)}$ the change applied to weight w_{ji} during the $(n-1)th$ step.

According to Jacobs [20], the back-propagation algorithm with momentum is an exponentially weighted sum of the current and past partial derivatives of the error function. For this algorithm to be convergent, the momentum constant must be restricted to the range $0 \le \alpha < 1$. Note that when α is zero the back-propagation algorithm operates without momentum. When consecutive derivatives of a weight have the same sign, the weight w_{ji} is adjusted by a large amount, and when consecutive derivatives possess opposite signs, this sum is adjusted by a small amount as above. Thus, the inclusion of a momentum will either accelerate learning in a downhill direction, or have a stabilizing effect in directions that oscillate. The momentum term may also have the benefit of preventing the learning process from terminating in a shallow minimum of the error surface.

In view of the poor performance of the back-propagation steepest descent algorithm, it has been suggested that the value of the learning rate η be adapted according to the contours of the error function [20,21,22,23]. The learning rate η is then treated as another factor to alter the step size of the weight change on the error surface. At present a number of strategies for adapting the learning rate can be found in the literature. These strategies can be divided into two broad classes; global and local learning rate adaptation. Global learning rate adaptation involves finding the proper value for the learning rate [21,24,25,26,27]. Local learning rate adaptation involves using independent learning rates for each adjustable weight in the network [20,28,29,30,31,32]. Two examples are discussed below

In the first example, the learning rate is adapted for every training pattern. Schmidhuber's algorithm [27] calculates a new learning rate for each training pattern and does not use a momentum term. The new weight values are found by calculating the point of intersection of a line drawn in the steepest descent direction from the current position to the zero error plane. This is equivalent using a tangential hyper-plane to locally approximate the error function. Let ε_k represent the error caused by some particular pattern, so that ε_k is a function of the weights $\boldsymbol{w}$. Linearising the dependence of ε_k on $\boldsymbol{w}$ about some operating point $\boldsymbol{w}^{(n)}$

$$\varepsilon_k^{'}(\boldsymbol{w}) = \varepsilon_k(\boldsymbol{w}^{(n)}) + (\boldsymbol{w} - \boldsymbol{w}^{(n)}) \ \boldsymbol{.}\ \nabla\varepsilon_k(\boldsymbol{w}^{(n)}) \tag{2.3}$$

where $\nabla\varepsilon_k = (\partial\varepsilon_k / \partial w_{j,i}),\ \forall\, j, i$ represents the gradient vector, and $\boldsymbol{a} \,.\, \boldsymbol{b}$ represents the inner product of vectors $\boldsymbol{a}$ and $\boldsymbol{b}$. We wish to find a zero of the error function ε_k, so

$$0 = \varepsilon_k(\boldsymbol{w}^{(n)}) + (\boldsymbol{w} - \boldsymbol{w}^{(n)}) \ \boldsymbol{.}\ \nabla\varepsilon_k(\boldsymbol{w}^{(n)}) \tag{2.4}$$

Let $\Delta\boldsymbol{w}^{(n)} = \boldsymbol{w}^{(n+1)} - \boldsymbol{w}^{(n)}$. From the method of steepest descent, the weights are adjusted according to $\Delta\boldsymbol{w}^{(n)} = -\eta\nabla\varepsilon_k^{(n)}$, so the value of η that sets the linearised error $\varepsilon_k^{'} = 0$ is given by

$$\eta^{(n)} = \frac{\varepsilon_k^{(n)}}{\sum_{j,i}\left(\partial\varepsilon_k^{(n)} / \partial w_{ji}\right)^2} \tag{2.5}$$

where j, i range over all the weight indices. For practical reasons it is necessary to define an upper limit η_{max} for a single step. There may also be some error surfaces that never reach the zero plane. For these surfaces, a small constant value ε_{off} is subtracted to make sure that the zero point exists. Schmidhuber emphasized that his algorithm was able to escape from a local minima $(\varepsilon_k \neq 0 \wedge \|\nabla\varepsilon_k\| \approx 0)$.

Unfortunately it is likely to result in very big updates that may corrupt the whole network in one step. This strategy also can't handle very big datasets where some wrongly classified training examples might exist.

In the second example, each individual weight has a corresponding learning rate that is allowed to vary over time. The Rprop (resilient back-propagation) algorithm [28] differs from other first order techniques in that the individual step-sizes Δ_{ji} are independent of the magnitude of the partial derivatives $\partial\varepsilon_k / \partial w_{ji}$. For each weight w_{ji}, an individual step-size Δ_{ji} is adjusted according to

$$\Delta_{ji}^{(n)} = \begin{cases} \eta^{+}\Delta_{ji}^{(n-1)} & if \ \dfrac{\partial\varepsilon_k^{(n)}}{\partial w_{ji}}\dfrac{\partial\varepsilon_k^{(n-1)}}{\partial w_{ji}} > 0 \\ \eta^{-}\Delta_{ji}^{(n-1)} & if \ \dfrac{\partial\varepsilon_k^{(n)}}{\partial w_{ji}}\dfrac{\partial\varepsilon_k^{(n-1)}}{\partial w_{ji}} < 0 \\ \Delta_{ji}^{(n-1)} & otherwise \end{cases} \tag{2.6}$$

After adjusting the step-sizes, the weight updates Δw_{ji} are determined. The weight update rule can be written

$$\Delta w_{ji}^{(n)} = -sgn\left(\frac{\partial\varepsilon_k^{(n)}}{\partial w_{ji}}\right)\Delta_{ji}^{(n)} \tag{2.7}$$

where $0 < \eta^{-} < 1 < \eta^{+}$, and $sgn(\,.\,)$ is the sign function. The principal benefit of this update scheme is that it removes the harmful influence of the size of the partial derivatives on the weight step. Only the sign of the partial derivatives are used to find the proper update direction. Further, the step sizes are adapted according to the signs of the current and previous derivatives. When the signs of successive derivatives are opposite, this means that the algorithm has jumped over a minimum and that the step size is too large. On the other hand if two successive signs are equal, then the step size is not big enough and could be increased. Finally, local back-tracking is usually applied to those weights when a change in the sign of the corresponding derivatives are detected

Schiffmann et al [33] have made comparisons of different global and local adaptation techniques to accelerate learning, which are fixed learning rate adaptation, learning rate adaptation for each pattern [27], angle driven learning rate adaptation [21], the conjugate gradient method [36], Delta-bar-Delta [20], Rprop [28], Quickprop [32] and Cascade Correlation [7]. The benchmark dataset used was thyroid [34]. The results show that algorithms using local adaptation strategies outperform all global adaptive learning algorithms both in terms of training time and network performance. On the other hand the back-propagation algorithm performing pattern by pattern updating outperforms all global adaptive learning algorithms. Moreover, Rprop was the fastest algorithm that used a fixed topology. According to training speed only Quickprop is comparable to Rprop. The results also show that Rprop is robust with respect to its own internal parameters. The cascade correlation algorithm outperforms all the other algorithms but is not directly comparable to them because it does not use a fixed topology.

2.2.2 Second-order optimization techniques

In the sequential mode of learning, weight updating is performed after the presentation of each training example. This mode of learning is also referred to as on-line, pattern, and stochastic mode. In the batch mode of learning, weight updating is performed after the presentation of all the training examples. There are several advantages in favour of each type of learning mode as outlined by Battiti [37]. One of the reasons in favour of sequential learning is that it possesses a degree of randomness that may help in escaping from a local minimum. The fact that many large datasets contain redundant data has also been cited in favour of sequential learning, because many of the contributions to the gradient are similar, so waiting to collect all the gradient information together can be wasteful. On the other hand, collecting all the gradient information together before the weights are updated can help to avoid the mutual interference of weight changes that occur with large learning rates. Sequential methods may because of their degree of randomness, miss a perfectly good local minimum. Even if the training data is redundant, sequential methods may be slow in comparison to batch methods that use second-order information. Some second-order batch techniques show superior performance with respect to the standard back-propagation algorithm on problems with a limited number of weights (< 100), especially if high precision mappings are required.

Newton's method can be considered as the basic locally convergent method using second-order information. It is based on using a second order Taylor expansion of the error function ε_k about the current operating point $\boldsymbol{w}$

$$\varepsilon_k(\boldsymbol{w}+\Delta\boldsymbol{w}) \approx \varepsilon_k(\boldsymbol{w}) + \nabla\varepsilon_k^T\Delta\boldsymbol{w} + \frac{1}{2}\Delta\boldsymbol{w}^T\nabla^2\varepsilon_k\Delta\boldsymbol{w} \tag{2.8}$$

and solving for the step $\Delta\boldsymbol{w}$ that brings $\boldsymbol{w}$ to a point where the gradient is zero. This corresponds to solving the following linear system

$$\nabla^2 \varepsilon_k \Delta\boldsymbol{w} \quad = \quad -\nabla\varepsilon_k \tag{2.9}$$

Generally speaking, Newton's method converges quickly to a solution and does not exhibit the zigzagging behaviour that characterises the steepest descent method. The main problems that can arise with Newton's method is when the Hessian $\nabla^2\varepsilon_k$ is not positive definite (i.e. the directional derivative $\Delta\boldsymbol{w}^T\nabla^2\varepsilon_k\Delta\boldsymbol{w} < 0$), or when the Hessian is singular or ill-conditioned. If the Hessian is not positive definite, there exists directions $\Delta\boldsymbol{w}$ of negative curvature that would suggest an infinite number of steps to minimise the error function. This behaviour is not uncommon in neural networks: in some cases large steps push units into saturation resulting in very small second order derivatives.

When the Hessian matrix is not positive definite and well conditioned, Newton's method cannot be used without modifications. This can be explained by examining the eigenvalues of the Hessian. Writing the Hessian using a spectral decomposition, we have

$$\nabla^2\varepsilon_k \quad = \quad \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^T \quad = \quad \sum_{i=1}^{n}\Lambda_{ii}\,\boldsymbol{u}_i\boldsymbol{u}_i^T \tag{2.10}$$

where $\boldsymbol{\Lambda}$ is a diagonal matrix whose diagonal elements Λ_{ii} are the eigenvalues of the Hessian, and $\boldsymbol{U}$ is a matrix whose columns are the orthogonal set of eigenvectors associated with the eigenvalues. It is easy to see that, if some of the eigenvalues are close to zero, the inverse matrix will have eigenvalues close to infinity, a sure source of numerical problems. If one of the eigenvalues is negative, the error function does not have a minimum because large movements in the direction of the corresponding eigenvector decrease the error value to arbitrary negative values.

A recommended strategy for changing the Hessian in order to avoid these difficulties is that of summing it to a diagonal matrix of the form $\mu_k \boldsymbol{I}$ so that $\nabla^2 \varepsilon_k + \mu_k \boldsymbol{I}$ is positive definite and well-conditioned. A proper value for μ_k can be found using the modified Cholesky factorisation found in Gill et al [38]. The Cholesky factors of a positive definite matrix can be considered as a sort of square root of the matrix. The original matrix is expressed as the product $\boldsymbol{L}\boldsymbol{D}\boldsymbol{L}^T$, where $\boldsymbol{L}$ is a lower triangular matrix with 1's on the leading diagonal, and $\boldsymbol{D}$ is a diagonal matrix with positive diagonal elements. Taking the square root of the diagonal elements using them to form the matrix $\boldsymbol{D}^{\frac{1}{2}}$, the original matrix can be written as $\boldsymbol{L}\boldsymbol{D}^{\frac{1}{2}}\boldsymbol{D}^{\frac{1}{2}}\boldsymbol{L}^T = \hat{\boldsymbol{L}}\hat{\boldsymbol{L}}^T$, where $\hat{\boldsymbol{L}}$ is a general lower triangular matrix. If the original matrix is not positive definite, the factorization can be modified in order to obtain factors $\boldsymbol{D}$ with all diagonal elements positive. The factorization corresponds to the original factors of the Hessian, and differing from it by adding a diagonal matrix with non-negative elements.

If the error function to be minimised is the sum of error squares, $\varepsilon_k = \frac{1}{2}\sum_{p=1}^{m} e_p^2$, where e_p is the error of the pth input pattern, learning a set of examples consists of solving a non-linear least squares problem for which special methods have been devised. Two of these methods are now described: the Gauss-Newton method, and the Levenberg-Marquardt method.

Let the error signal e_p be a function of the weight vector $\boldsymbol{w} \in R^n$. The gradient and Hessian matrix of ε_k are given by

$$\nabla \varepsilon_k = \sum_{p=1}^{m} e_p \nabla e_p = \boldsymbol{J}_e^T \boldsymbol{e} \tag{2.11}$$

and

$$\begin{aligned} \nabla^2 \varepsilon_k &= \sum_{p=1}^{m} \left[\nabla e_p \nabla e_p^T + e_p \nabla^2 e_p \right] \\ &= \boldsymbol{J}_e^T \boldsymbol{J}_e + \boldsymbol{S} \end{aligned} \tag{2.12}$$

where $\boldsymbol{J}_e$ is the Jacobian matrix [$\partial \boldsymbol{e} \big/ \partial w_j$], $\boldsymbol{e}$ is a vector of errors e_p, and $\boldsymbol{S}$ is that part of the Hessian containing the second derivatives of $\boldsymbol{e}$, that is, $\boldsymbol{S} = \sum_p e_p \nabla^2 e_p$. For small residual problems (i.e. small values of e_p), the second order part $\boldsymbol{S}$ is negligible, and Newton's method can be written

$$\Delta \boldsymbol{w} = -[\boldsymbol{J}_e^T \boldsymbol{J}_e]^{-1} \boldsymbol{J}_e^T \boldsymbol{e} \tag{2.13}$$

It can be shown that this step is completely equivalent to minimizing the error obtained using a first order Taylor expansion of the error, $\boldsymbol{e}'$. The updated weight vector is then defined by

$$\boldsymbol{w} = \min_{\boldsymbol{w}} \left\{ \tfrac{1}{2} \boldsymbol{e}'^T \boldsymbol{e}' \right\} \tag{2.14}$$

Equation (2.13) defines the Gauss-Newton method. The term $\boldsymbol{J}_e^T \boldsymbol{J}_e$ in the Gauss-Newton method is a low computational approximation of the Hessian matrix. It is sufficiently accurate for small residual problems. Therefore, the Gauss-Newton method has quadratic or second order convergence as the minimum on the error surface is approached. Meyer [39] has shown that the convergence of the Gauss-Newton method is superlinear (i.e. $\| \boldsymbol{e}^{(t+1)} \| / \| \boldsymbol{e}^{(t)} \| \rightarrow 0$, $t \rightarrow \infty$) whenever $\| \boldsymbol{S} \| \rightarrow 0$, otherwise it is only first order.

The only problem that can arise with equation (2.12) is the Jacobian matrix $\boldsymbol{J}_e$ being rank deficient, and hence $\boldsymbol{J}_e^T \boldsymbol{J}_e$ is singular. The Levenberg-Marquardt (LM) method [40] incorporates a technique for dealing with a rank deficient $\boldsymbol{J}_e$, and is effective for small residual problems. In this method a diagonal matrix $\mu \boldsymbol{I}$ is added to $\boldsymbol{J}_e^T \boldsymbol{J}_e$, where μ is a small positive constant and $\boldsymbol{I}$ the unit matrix. When $\mu = 0$, $\Delta \boldsymbol{w}$ is given by $[\boldsymbol{J}_e^T \boldsymbol{J}_e]^{-1} \boldsymbol{J}_e^T \boldsymbol{e}$. As $\mu \rightarrow \infty$, the effect of the term $\mu \boldsymbol{I}$ increasingly dominates that of $\boldsymbol{J}_e^T \boldsymbol{J}_e$ so that $\Delta \boldsymbol{w} \rightarrow \mu^{-1} \boldsymbol{J}_e^T \boldsymbol{e}$, which represents an infinitesimal step in the steepest descent direction.

Fletcher [41] has improved the LM method by providing a strategy for selecting μ. To decide whether to change μ in the next iteration, he compares the actual reduction in the cost function with that predicted by assuming that the cost function is quadratic. If this ratio is in the region of 1 then the cost function is behaving quadratically and μ should be reduced. On the other hand, if it is close to zero (a sign of poor progress), it should be increased. Wilamowski et al [42] have further improved the LM method by introducing a modified cost function. The LM method requires the inversion of a $\boldsymbol{J}_e^T \boldsymbol{J}_e$ square matrix, which is not practical in large networks. Utilising a modified cost function and using the matrix inversion lemma produces a substantial reduction in complexity and memory usage

2.3 Generalization capabilities of artificial neural networks

A neural network is said to generalize well when the output computed by the network is correct for test data not used in training the network. Here it is assumed that the test data is drawn from the same population as the training data. If the learning process is viewed as a curve fitting problem then the neural network itself may be considered as a non-linear input-output mapping. Such a viewpoint then permits us to look on generalization simply as the effect of good non-linear interpolation. Neural networks can perform useful interpolation because multilayer neurons with continuous activation functions lead to output functions that are also continuous [43].

A neural network that generalizes well will produce a correct input-output mapping even when the test data is slightly different from the examples used to train the network. However, when a neural network learns too many input-output examples the network may end up memorising the training data. It may do so by finding a feature in the training data such as noise that is not present in the underlying function. Such a phenomenon is referred to as overfitting or overtraining. When a network is overtrained it loses its ability to generalize between similar input-output patterns

Good generalization performance in a neural network is influenced by a number of different factors; the size of the training set, the size of the network and the complexity of the problem at hand [6].

- The training set size must be sufficiently large to provide enough information to learn the underlying function. Generalization may fail if there are hidden variables affecting the training data that are not shown to the network or if noise has swamped the available information.

- A neural network that is too small may fail to learn the underlying function, that is, it will underfit the training data. However, a network that is too large may tend to overfit the training data and thus generalize poorly to new data. Thus, there is a trade-off between underfitting and overfitting.
- The underlying function must be sufficiently smooth. A network can learn functions with a finite number of discontinuities but not totally chaotic or random functions.

A number of different strategies can be found in the literature for improving generalization in neural networks. The Vapnik-Chervonenkis (VC) dimension provides theoretical worst case estimates for the size of the training set required for a good generalization [44]. To limit the effect of size on a neural network, you can either use additive, subtractive, or regularization methods. Additive methods start with a small network and insert new units until it can represent the required function [7,8,9]. Subtractive methods remove superfluous weights from a fully trained network [10,11]. Regularization methods use a network with a large number of weights, and impose constraints on each weight in addition to error minimization. Examples of regularization are weight sharing, and weight decay [12,13]. The onset of overfitting can be identified using a validation set of unseen data, similar in some sense to the data used to train the network. Training is then stopped before convergence occurs to avoid overfitting.

2.3.1 Constructive techniques

Constructive methods start out with a small network and then add new units and connections until the network can represent the required function. Perhaps the most notable example of constructive methods is the Cascade Correlation algorithm proposed by Fahlman and Lebiere [7]. The cascade correlation algorithm increases the size of the network by adding new units and layers. There are two distinctive features to the cascade correlation algorithm. First, the cascade architecture. This means that all the hidden units are added to the network one at a time, each on a separate hidden layer. The cascade structure leads to the creation of powerful high-order feature detectors and to very deep networks. Second, the objective function used to train the new hidden units. For each new hidden unit, the cascade-correlation process aims to maximize the magnitude of the correlation between the hidden units output and the residual network error signal. The cascade-correlation architecture thus has several advantages over conventional back-propagation; it learns very quickly, the network determines its own size, it preserves its structure even if the training set changes, it does not require back-propagating error signals, and only one layer of weights are trained at a time.

There are many constructive methods described in the literature for building Radial Basis Function (RBF) networks. In the procedure proposed by Chen et al [45], the training data points are considered as candidate RBF centres. The RBF centres are selected one by one using the orthogonal least squares (OLS) method. The OLS method has the property that each selected centre maximizes the contribution of the RBF network to desired output variance, and it does not suffer from numerical ill-conditioning. In [46], a constructive method was described named the Resource Allocation Network (RAN) which adds hidden units sequentially based on the novelty of the input data. A new input pattern is considered as novel if that sample is far away from existing centres and if the output error is large. If the input pattern does not pass the criteria for novelty, then no hidden unit is added and the network is trained using the OLS method. In [47] a new method for RBF networks named GAP-RBF was described which adds and prunes hidden units based on a simple estimate of the significance of centres. The significance of a unit is the error that results from removing that unit from the network over all inputs seen so far. Results for GAP-RBF show it can achieve a smaller network realized by RAN, and that it achieves higher classification accuracies and better generalization.

Zhang et al [22] proposed an orthogonal sequential technique for single hidden layered neural networks based on the OLS method. The procedure starts with a single hidden unit and sequentially increases the number of hidden units in a single hidden layer until the error is sufficiently small. When adding a new hidden unit, it is the component of the output that is perpendicular to the space spanned by the outputs of previously added hidden units that is used to train the network. The Gram-Schmidt method is used at each step to construct a set of orthogonal bases for the space spanned by the outputs of the hidden units. Two training examples were used to test the sequential training technique. In the first example, a function approximation problem was modelled using an RBF network. The results indicate that the RBF network with centres selected through optimization performs better than with centres selected using OLS method proposed by Chen et al [45]. In the second example, time-series data obtained from [48] was modelled using a single hidden layer neural network having different activations; linear, sigmoid, and Gaussian. The results indicate that the orthogonal sequential training technique constructs small parsimonious networks capable of good generalization.

2.3.2 Network pruning techniques

An alternative method to adjusting the size of a neural network is to start with a large network that overfits the input data and remove the subset of synaptic weights that result in the smallest increase in error. Sietsma and Dow [49] analyzed this by examining all neurons under the presentation of the entire training set and removing only those neurons that did not change during training. Although this technique produced good generalization results in small networks it was far too exhaustive an approach for training large neural networks. Mozer and Smolensky [50] introduced the idea of estimating the sensitivity of the network to the removal of a synaptic weight. This was achieved by associating a sensitivity value with each weight calculated from the gradient of the error function. They reasoned that weights with the smallest sensitivity values had the least effect on the network output and could therefore be removed from the network. LeCun [10] proposed an alternative approach called Optimal Brain Damage (OBD) for calculating the sensitivity values of the synaptic weights. He used a Taylor series to expand the error function about the error minimum and then applied a quadratic approximation to remove any high order terms from the resulting expansion. This produced saliency values dependent on the second order derivatives of the error function. Once again the synaptic weights with the smallest saliency values were the likeliest candidates for network pruning. The main disadvantage of pruning techniques are that they require training down to the error minimum before pruning can occur. This frequently produces massive overfitting which often cannot be repaired by subsequent pruning. The autoprune method [50] avoids this problem. Its weight importance coefficients are defined by a test statistic for the assumption that a weight becomes zero during the training process. Connections with a small test statistic can be pruned. Finnoff et al [52] has shown that autoprune is superior to OBD.

2.3.3 Other techniques for improving generalization

Overfitting can also be avoided by reducing the complexity of the data in the input data set. A technique called Principal Component Analysis (PCA) can be used to project the input vectors onto a vector space whose basis is described by the eigenvectors of an input correlation matrix. The reduction is achieved by choosing the principal components of the input vectors that have the largest variance. There are many examples of architectures performing PCA in the literature [54,55,56,57,58]. For example, the well-known generalized hebbian algorithm (GHA) proposed by Sanger [56], and the adaptive principal component extractor (APEX) proposed by Kung and Diamantaras [57]. Both are extensions of Oja's principal component neuron [55]. Fiori [58] applied these algorithms together with the new ψ-APEX to 20 datasets containing 5,000 samples. The results show that ψ-APEX and GHA give the best results when the eigenvalues of the correlation matrix are spread wide apart, otherwise they are the same.

Excessive training also contributes to poor generalization in neural networks. A technique called cross validation [59] can be used to detect the onset of overfitting during the training stage, training can then be terminated to prevent overfitting from developing. Prechelt [53] investigated cross validation using three different stopping criteria; the generalization loss on the validation data, the progress on the training set, and the ratio of the generalization loss to the progress made. He found that the slowest stopping criteria produced best generalization results but this was at the expense of much longer training times. Levin [60] investigated the effect of using different generalization measures on validation data in thin-plate spline RBF networks. When the mean squared error was used he found that the generalization error did not increase as expected when the network started to overfit the data. This was because the error measure used penalized networks that produced only a few erroneous results. He proposed the use of a new error measure based on the median of the absolute error. The median tends to reduce the effects of a few erroneous results on the overall sample average.

A novel approach to obtaining the right size network for good generalization was proposed by Lee [14]. Lee suggested growing the weights of the network from almost zero initial values with the expectation that only the necessary number of weights would be activated. One of the difficulties of starting the training under these initial values is that the direction of the gradient vector at a point close to the origin in weight space is approximately along the direction of the axis defined by the output unit's bias weight. Iterative methods based on steepest descent would therefore adapt only the output unit bias weight of the network. Lee suggested using the tangent plane algorithm as a starting point [10]. The tangent plane algorithm defines not just a single direction to move on being presented with a item of training data, but a whole plane of directions to move towards. The efficiency of the algorithm should not be impaired too much if a nearby point on the tangent plane is chosen, displaced somewhat in the direction away from the origin. Lee applied this approach to the Wisconsin State breast cancer diagnosis problem [34], and the Nettalk data [62]. The results were very promising and comparable to those previously obtained on these datasets. Further, the second tangent plane algorithm gives improved generalization independent of network size whist retaining the fast convergence speed and high classification accuracy of the original tangent plane method. A detailed description of the second tangent plane algorithm is presented in chapter 3 together with suggestions for improvements. The results of simulation tests will also be presented.

2.4 Multi-category classification using gene expression data

Bioinformatics is defined as the storage and manipulation of biological information [63]. Luscombe [64] defines bioinformatics as conceptualising biology in terms of molecules and applying informatics techniques to understand and organise the information associated with these molecules on a large scale. One aspect of bioinformatics is the analysis of biological data. This involves gene identification and prediction, gene structure prediction, and the investigation of macro structures such as secondary and tertiary protein structures, and examining protein geometries using distance and angle measures. Another aspect of bioinformatics is the biological data itself. One property of biological data is the extremely large amount. For example a DNA sequence of genes comprises strings of four base letters, each gene 1,000 bases long. The GenBank [65] repository holds more than 12.5 million bases in 115 million entries.

The intensive interest in bioinformatics has been driven by the emergence of experimental techniques that generate a great amount of data, such as DNA sequencing, mass spectrometry and micro-array expression analysis [66]. These problems are so large that they are impossible to analyse manually. Micro-array technologies [67,68] allow the monitoring of gene expression levels of thousands of genes simultaneously in any given cell, cell line or human tissue. The study of micro-array technology has attracted more interest and has been an important factor in bioinformatics research during recent years [69,70,71]. There are many classification methods being used for cancer classification both from statistical and machine based learning, but some characteristics of gene expression make this task very difficult. First, gene expression data usually has a high dimensionality, which often contains thousands of genes. This can cause a great computational overload. Second, the size of the samples is usually very small, often below 100. A low ratio between training sample size and number of genes results in a very sparse input space, which makes accurate classification very difficult. Third, most of the genes are irrelevant to cancer classification, and simply add back-ground noise to any analysis carried out.

There has been a number of classification methods used for cancer classification both from statistical and machine learning. These methods include k-nearest neighbour [66,72], linear discriminant analysis [68,70,72], and support vector machines [73,74,75]. Nearest neighbour methods are based on some distance function of a pair of patterns, such as the Euclidean distance. The k-nearest neighbour rule proceeds as follows. For each pattern in the test set, find the k-nearest patterns in the training set, and predict the class of the pattern by majority vote, that is choose the class that is most common among the k-nearest neighbours. Linear discriminant analysis is a method that finds the linear combinations of features which best separate two or more classes. The resulting combination can be used as a linear classifier or, more commonly, for dimensionality reduction. Fisher linear discriminant analysis is based on finding linear combinations of pattern vectors with large ratios of between-class to within-class sum of squares. This measure is in some sense a measure of the signal to noise ratio for the class labelling. One of the first applications of discriminant methods to gene expression data was the weighted voting scheme [70]. In this method a sample is assigned to a particular class according to the weighted distance between it and the nearest class mean vector. Support vector machines map the input space into a higher dimensional feature space so that the data is linearly separable into two classes. This separation is achieved by constraining the Euclidean norm of the weight vector.

Over the past few years binary classification methods using gene expression data has been studied intensively [66,68,70,73,74,76,77,78]. These studies indicate that multi-class problems are far more difficult than binary ones and classification accuracies drop off sharply as the number of classes increases. Ramaswamy et al [74] applied an SVM algorithm for the analysis of gene expression data on 14 different tumours in the GCM dataset. For a c-category classification problem, c binary classifiers were used each to discriminate one class from all others. This method has potential drawbacks when there is considerable overlap between classes in pattern space. The results are very promising in relation to k-nearest neighbour and weighted voting methods. Yeang et al [69] have made a comparison of three binary classification methods, k-nearest neighbour, weighted voting and support vector machines. Three combinatory schemes were used, one-versus-all, one-versus-one, and hierarchical partitioning. The results show that all the support vector machines produced the best results when all the genes were used. For the k-nearest neighbour and weighted voting methods, one-versus-one tended to outperform one-versus-all when a fixed number of genes were used. Dudoit et al [72] made a comparison of linear discriminant methods, nearest neighbour classifiers, decision trees and aggregation methods. Three datasets were used, lymphoma, leukaemia, and NCI60. The results show that the k-nearest neighbour method and diagonal linear discriminant method had the lowest test set errors, and that the Fisher discriminant method had the highest test set error. Stratnikov et al [78] presents a comprehensive evaluation of several multi-category classification methods including SVM, k-nearest neighbour, weighted voting and a back-propagation neural network. The study used nine multi-category datasets and two binary datasets; GCM dataset, brain tumour dataset, leukaemia dataset, MLL dataset, lung cancer dataset, SRBCT dataset, prostrate tumour dataset and DLBCL dataset. The results show that SVM classifiers were the best performers with and without gene selection, and that weighted voting and decision tree methods were the worst; the back-propagation neural network ranked in the middle.

Neural network classifiers are well established for their unique capability to map the input space non-linearly into a higher dimensional feature space so that the data is linearly separable into numerous different classes. Compared with SVM, neural networks can map the input space directly into a number of different classes, while SVM maps the input data so that it is separable into 2 classes. This particular property of neural networks to accommodate the non-linear features of expression data might actually reduce the number genes required for accurate classification, which is reducing the dimensionality of the classification problem. For linear algorithms, more genes are required to form a higher dimensional space for the separation. The first application of ANN for cancer classification using gene expression data is presented in [79]. Khan et al. used a two hidden layered feed-forward neural network to classify small, round blue-cell tumours into four categories. The ANN method correctly classified all the samples that present difficulties in clinical diagnosis methods. In Linder et al [77] a new neural network algorithm was developed for multi-category classification using gene expression data. This method uses a simple ANN to perform a pre-selection at the first stage. At this stage a simple ANN narrows the choice down to two preferred classes with the highest activities in the output neurons. A subsequent ANN (SANN) is applied for the final decision on these two selected classes. Linder have applied to the GCM dataset with very good results. The results show that SANN beats the best classifier as described in [74] but it causes a great increase in network complexity and very slow convergence. Therefore, faster and more efficient neural network algorithms are needed that are capable of high classification accuracy. In the next chapter we investigate a fast sequential algorithm for training neural networks.

Chapter 3

COMPREHENSIVE EVALUATION OF THE TANGENT PLANE ALGORITHM

In Lee [14], an algorithm is described for supervised learning in multilayered feed-forward neural networks. This second tangent plane algorithm uses the target values of the training data to define a surface in the weight space of the network. The weights are updated by moving to the tangent plane to this surface. It differs to more conventional gradient descent based learning methods by accepting almost zero initial conditions with the expectation that only the minimum number of weights will be activated. It has been shown to give improved generalization and significantly faster convergence relative to the standard back-propagation algorithm on benchmark classification problems.

In this chapter, the performance of the second tangent plane algorithm is evaluated for classification and function approximation tasks and compared with the back-propagation algorithm. Two limitations of the second tangent plane algorithm are identified. First, the performance of the second tangent plane algorithm is no better than the backpropagation algorithm in small parsimonious networks where generalization is found to be best. Second, the second tangent plane algorithm frequently fails to converge when the training data is corrupted by adding a small amount of random noise to the teaching values. Finally, the differentiation and evolution of weights in neural networks trained by the second tangent plane algorithm is investigated. Histograms of weight importance coefficients are used to evaluate the effectiveness of growing the weights from small initial values as a method for improving generalization.

3.1 Description of the tangent plane algorithm

A neural network must have the right size for good generalization. Networks that are too small cannot fit the required function, whereas networks that are too large are prone to overfitting [82]. There are several approaches to determine the correct size for a network. In Lee [14] a method was described that grows the weights from almost zero initial conditions in the expectation that only the necessary number of weights would be activated. Lee used the first tangent plane algorithm [2] as a starting point, for instead of determining a single direction to move on in weight space, it determines a plane of suitable points to move to. The first tangent plane algorithm is a fast method of training a feed-forward neural network. It avoids inappropriate step sizes by treating each training value as a constraint that defines a surface in weight space. The weights are then adjusted by moving from the current position to the tangent plane to this surface. The second tangent plane algorithm adjusts the weights by moving from the current position to a point close to the foot of the perpendicular, but displaced somewhat in the direction away from the origin. This directional component of movement helps to push the network weights away from the origin where the convergence speed of the tangent plane algorithm, and other steepest descent methods, is known to be very slow [2]. In the region of weight space close to the origin the axis defined by the weight from the constant output bias is very nearly perpendicular to all the constraint surfaces. Thus the tangent plane algorithm gives movement up and down this axis satisfying the constraints on the weights by adjusting this weight only.

In the next section we describe the derivation of the tangent plane algorithm and the steps involved in the training procedure. In the following section we describe the weight elimination procedure used to measure weight importance values and in the last section the simulation results.

3.1.1 Derivation of the tangent plane algorithm

The basic structure of a feed-forward neural network is shown in Fig 3.1. It consists of an input layer of units that supply information, or activations, to the inputs of units in the first hidden layer. These in turn supply activations to inputs of units in the next layer, and so on. Typically the units in each layer receive inputs from the output of the units in the preceding layer. Let w_{ji} denote the weight between unit u_i and u_j. ϕ_j and θ_j will be the input and output of u_j, so that $\theta_j = f(\phi_j)$ and $\phi_j = \sum_i w_{ji}\theta_i$ for some monotonic function f.

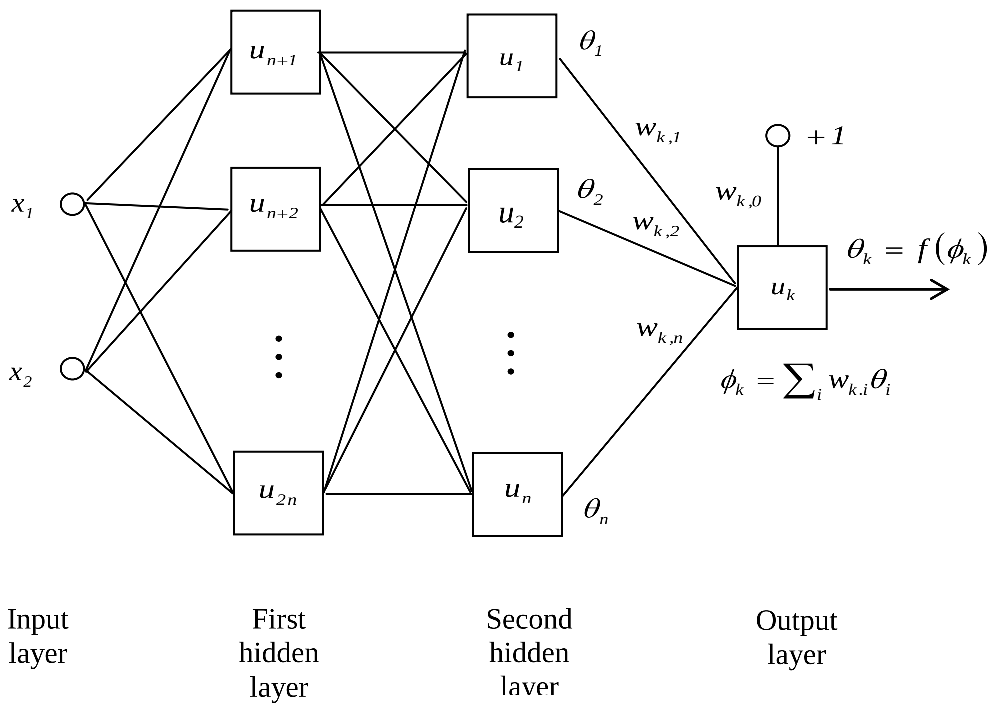


Fig. 3.1. The structure of a feed-forward neural network

Let u_k be trained to mimic the target value y_k. The tangent plane algorithm adjusts the weights by moving at an angle β to the perpendicular from $\boldsymbol{a}$ to the tangent plane to the surface $\phi_k = f^{-1}(y_k)$, taken at a point where a line dropped from $\boldsymbol{a}$

and parallel to the axis defined by the output unit's bias weight w_{k0} meets this surface (see Fig. 3.2).

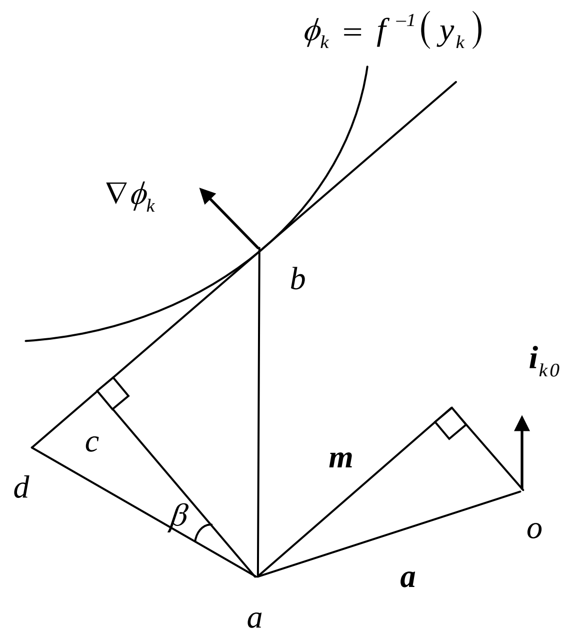


Fig. 3.2. Movement from the present position **a** to the foot of the perpendicular to the tangent plane of constraint surface $\phi_k = f^{-1}(y_k)$ to position **d**

Let $\boldsymbol{a} = \sum_{j,i} w'_{ji}\boldsymbol{i}_{ji}$, where $\boldsymbol{i}_{ji}$ is a unit vector in the direction of the w_{ji} axis. Use the equation $f^{-1}(y_k) = w_{k0} + \sum_{i\neq 0} w_{ki}\theta_i$ to find a value, w''_{k0}, for the bias weight w_{k0} from the values w_{ji} of the other weights, so that the surface $\phi_k = f^{-1}(y_k)$ contains the point $\boldsymbol{b} = w''_{k0}\,\boldsymbol{i}_{k0} + \sum_{j,i\neq k,0} w'_{ji}\boldsymbol{i}_{ji}$. Now, if we use the equation $f^{-1}(y_k) = w''_{k0} + \sum_{i\neq 0} w'_{ki}\theta_i$ and $f^{-1}(\theta_k) = w'_{k0} + \sum_{i\neq 0} w'_{ki}\theta_i$, and note that $\boldsymbol{b}$ differs from $\boldsymbol{a}$ only in the value of w_{k0} , we get

$$\begin{aligned} \boldsymbol{b} - \boldsymbol{a} &= \left(w''_{k0} - w'_{k0}\right)\boldsymbol{i}_{k0} \\ &= \left(f^{-1}(y_k) - f^{-1}(\theta_k)\right)\boldsymbol{i}_{k0} \end{aligned} \tag{3.1}$$

Let $\hat{\boldsymbol{n}}$ be the unit normal to the surface at $\boldsymbol{b}$, so $\hat{\boldsymbol{n}} = \nabla\phi_k / \|\nabla\phi_k\|$. The length of the perpendicular from $\boldsymbol{a}$ to the tangent plane at $\boldsymbol{b}$ is $(\boldsymbol{b} - \boldsymbol{a}).\hat{\boldsymbol{n}}$. If $\boldsymbol{c}$ is the foot of the perpendicular from $\boldsymbol{a}$ to the tangent plane at $\boldsymbol{b}$,

$$\begin{aligned} \boldsymbol{c} - \boldsymbol{a} &= \left(f^{-1}(y_k) - f^{-1}(\theta_k)\right)\boldsymbol{i}_{k0} \cdot \frac{\nabla\phi_k}{\|\nabla\phi_k\|}\frac{\nabla\phi_k}{\|\nabla\phi_k\|} \\ &= \frac{f^{-1}(y_k) - f^{-1}(\theta_k)}{\|\nabla\phi_k\|}\frac{\nabla\phi_k}{\|\nabla\phi_k\|} \end{aligned} \tag{3.2}$$

The vector parallel to the tangent plane and directed away from origin at $\boldsymbol{a}$ is $\boldsymbol{m} = \boldsymbol{a} - (\boldsymbol{a}.\hat{\boldsymbol{n}})\hat{\boldsymbol{n}}$. Thus, if $\boldsymbol{d} \in R^n$ is the point of intersection with the tangent plane of a line from $\boldsymbol{a}$ inclined at angle β to the perpendicular, then

$$\begin{aligned} \boldsymbol{d} - \boldsymbol{a} &= (\boldsymbol{d} - \boldsymbol{c}) + (\boldsymbol{c} - \boldsymbol{a}) \\ &= \|\boldsymbol{c} - \boldsymbol{a}\| \; tan\,\beta \, \frac{\boldsymbol{m}}{\|\boldsymbol{m}\|} + (\boldsymbol{c} - \boldsymbol{a}) \end{aligned} \tag{3.3}$$

Let $\delta = f^{-1}(y_k) - f^{-1}(\theta_k)$ be the error in the input to final unit. Hence, using equations (3.2) and (3.3), we obtain

$$\begin{aligned} \boldsymbol{d} - \boldsymbol{a} = \; & \frac{1}{\|\nabla\phi_k\|^2}\delta\nabla\phi_k + \frac{|\delta|}{\|\nabla\phi_k\|} tan\,\beta \frac{1}{\|\boldsymbol{m}\|}\Big(\boldsymbol{w} - \frac{1}{\|\nabla\phi_k\|} \\ & \times \sum_{lm} w_{lm} \frac{\partial\phi_k}{\partial w_{lm}} \frac{\nabla\phi_k}{\|\nabla\phi_k\|}\Big) \end{aligned} \tag{3.4}$$

So, to adjust a given weight w_{ji}

$$\Delta w_{ji} \;=\; \frac{1}{\left\|\nabla \phi_k\right\|^2}\,\delta\,\frac{\partial \phi_k}{\partial w_{ji}} + \frac{|\delta|}{\left\|\nabla \phi_k\right\|}\,tan\,\beta\,\frac{1}{\left\|\boldsymbol{m}\right\|}\left(w_{ji} - \frac{1}{\left\|\nabla \phi_k\right\|^2} \times \sum_{lm} w_{lm}\,\frac{\partial \phi_k}{\partial w_{lm}}\,\frac{\partial \phi_k}{\partial w_{ji}} \right) \tag{3.5}$$

where

$$\left\|\boldsymbol{m}\right\|^2 \;=\; \sum_{j,i}\left(w_{ji} - \frac{1}{\left\|\nabla \phi_k\right\|^2}\sum_{l,m}\left(w_{lm}\,\frac{\partial \phi_k}{\partial w_{lm}}\right)\frac{\partial \phi_k}{\partial w_{ji}}\right)^2$$

The term $\partial\phi_k \big/ \partial w_{ji}$ is the partial derivative of the net input to the output unit. This derivative is evaluated at point $\boldsymbol{w}''$ on the constraint surface, not at the current position $\boldsymbol{w}'$ in weight space. The treatment of this term follows from the back-propagation rule e.g.

$$\frac{\partial \phi_k}{\partial w_{ji}} \;=\; \frac{\partial \phi_k}{\partial \phi_j}\,\theta_i \tag{3.6}$$

and

$$\frac{\partial \phi_k}{\partial \phi_j} \;=\; \begin{cases} 1\,, & if\;\; j = k \\[2ex] f_j'\!\left(\phi_j\right) \displaystyle\sum_{m \in M_j} \frac{\partial \phi_k}{\partial \phi_m}\, w_{mj}\;, & if\;\; j \neq k \end{cases} \tag{3.7}$$

where M_j is the set of units to which u_j passes its output

The second tangent plane algorithm requires a parameter $tan\,\beta$ that needs to be set manually. This parameter is the tangent of the angle between the movement vector and the perpendicular from the current position to tangent plane. Preliminary tests showed that the algorithm was not particularly sensitive to the exact value chosen. The only difficulty with $\tan\beta$ is that very large values may result in units being forced into saturation which will slow down the learning. Lee suggests a modification to the second tangent plane algorithm that involves reversing the outwards push when the average of the absolute values of the weights $\overline{w}$ increases above 1.0. This was achieved by multiplying $\tan\beta$ by the term $(1 - \overline{w})$. This may give the second tangent plane algorithm an advantage over the standard back-propagation algorithm as there is no mechanism in this method for reducing large weight sizes should they occur. However, there exists a potential danger that sign changes in the term $(1 - \overline{w}) \times \tan\beta$ will cause oscillatory movement across the boundary of the region $\overline{w}$ < 1.0, which will slow down convergence

3.2 Implementation of the procedure

The following section is included to clarify the procedure for updating the weights of a network trained using the tangent plane algorithm

1. For each unit u_j,

$$x_j = \begin{cases} 1 \quad if \;\; j = k \\ \\ f^{-1}(\phi_j)\sum_m x_m w_{mj} \quad otherwise \end{cases}$$

2. For each weight w_{ji}, calculate $\partial\phi_k / \partial w_{ji}$ using

$$\frac{\partial\phi_k}{\partial w_{ji}} = x_j\theta_i$$

3. Calculate

$$\left\| \nabla\phi_k \right\|^2 = \sum_{j,i}\left(\frac{\partial\phi_k}{\partial w_{ji}}\right)^2$$

4. Calculate the components of the vector $\boldsymbol{m}$

$$m_{ji} = w_{ji} - \sum_{l,m}\left(w_{lm}\frac{\partial \phi_k}{\partial w_{lm}}\right)\frac{1}{\left\|\nabla \phi_k\right\|^2}\frac{\partial \phi_k}{\partial w_{ji}}$$

5. Calculate

$$\|\boldsymbol{m}\|^2 = \sum_{j,i} m_{ji}^2$$

6. Calculate

$$\delta = f^{-1}(y_k) - f^{-1}(\theta_k)$$

7. For each weight w_{ji}, add

$$\Delta w_{ji} = \frac{1}{\left\|\nabla \phi_k\right\|^2}\delta\frac{\partial \phi_k}{\partial w_{ji}} + \frac{|\delta|}{\left\|\nabla \phi_k\right\|}\tan\beta\frac{1}{\|\boldsymbol{m}\|}\left(w_{ji} - \frac{1}{\left\|\nabla \phi_k\right\|^2}\right.$$

$$\left.\times\sum_{lm} w_{lm}\frac{\partial \phi_k}{\partial w_{lm}}\frac{\partial \phi_k}{\partial w_{ji}}\right)$$

Note: the second tangent plane algorithm reverts to the first tangent plane algorithm when the angle parameter β is set to zero

3.3 Estimating weight sensitivity values

Despite the success of the tangent plane algorithm there is strong evidence to suggest that large weight values can harm generalization. Excessively large weights feeding into output units can cause wild outputs far beyond the range of the data if an output activation function is not included. To put it another way, large weights can cause excessively large variances in the output. According to Bartlett [83], the size of the weights is more important than the number of weights in determining good generalization. This poses the following question: is the strategy of growing weights actually harmful to generalization. One approach might be to measure the significance or importance of each weight, as the magnitude of the weights is not the best measure of their contribution to the training process [11].

There are several methods suggested for calculating the importance of connection weights. Karnin [12] measures the sensitivity S_{ji} of each weight by monitoring the sum of all the changes to the weights during training. Thus the saliency of a weight is given as $\hat{S}_{ji} = -\sum_t \partial \varepsilon_k^{(t)} \big/ \partial w_{ji} \, \Delta w_{ji}^{(t)} \; w_{ji}^f \; / \; (w_{ji}^f - w_{ji}^0)$, where t are the number of epochs trained, w_{ji}^f and w_{ji}^0 are the final and initial values of weight w_{ji}. Le Cun et al [10] measure the saliency of a weight by estimating the second derivative of the error. They also reduce the network complexity by constraining certain weights to be equal. Low saliency means low importance of the weights. A more sophisticated approach avoids the drawbacks of approximating the second derivatives by computing them exactly [11].

The last two methods have the disadvantage of requiring training down to the error minimum. The autoprune method [15] avoids this problem. It uses a statistic T to allocate an importance coefficient to each weight based upon the assumption that a weight becomes zero during the training process

$$T\left(w_{ji}\right) \quad = \quad log\left(\frac{\left| \sum_t w_{ji} - \Delta w_{ji}^{(t)} \right|}{\sum_t \left(\Delta w_{ji}^{(t)} - \left(\Delta \overline{w}_{ji} \right) \right)^2} \right) \qquad (3.8)$$

In the above formula, sums are over all training examples *t* of the training set, and the overline means arithmetic mean over all examples. A large value of T indicates high importance of weight w_{ji}.

3.4 Simulations and results

Comparative tests were performed on the first and second tangent plane algorithm and the back-propagation algorithm under a variety of initial conditions and using different network sizes. The benchmark datasets sets used were regression and classification problems. Classification problems involve a decision making task where the output fits into well-defined categories. The classification tasks chosen were N-bit parity and cancer. Regression problems involve the approximation of a continuous valued function. The regression problem chosen was hearta. All the datasets are from the UCI machine learning repository and made available in the Proben1 collection [34], except N-bit parity which has already been used in the paper on the tangent plane algorithm [2].

The N-bit parity problem was used to analyse the convergence behaviour of the first tangent plane algorithm. Tests were performed under a variety of conditions with respect to the network size for this purpose. Further tests were performed by adding noise to the teaching variables to analyse the ability of the tangent plane algorithm to converge to a compromise solution with fuzzy data. Network training was terminated after the error was reduced to below a preset value or the maximum number of epochs was reached

The hearta and breast cancer problems were used to determine the degree to which the second tangent plane algorithm would generalize from the given data. Tests were performed using different sized networks for this purpose. Network training was terminated using the method of early stopping as this method is known to help avoid overfitting.

3.4.1 Network initialization

The algorithms require parameters to be set manually. Preliminary tests showed that the best results were obtained with the parameters set as follows. First, the tangent plane algorithm. For N-bit parity, $\tan\beta = 0$. The input weights were set to random values in [-1,1]. For the hearta and breast cancer problems, $\tan\beta = 0.1$ and 0.02 respectively. The input weights were set to random values in the range [-0.01,0.01]. Next, the gradient descent back-propagation algorithm. For N-bit parity, $\eta = 0.01$, and $\alpha = 0.7$. For the breast cancer problem, $\eta = 0.01$, and $\alpha = 0.7$. For the hearta problem, $\eta = 0.01$, and $\alpha = 0.3$. The input weights were set to random values in the range [-1,1].

3.4.2 Simulation problems

The N-bit parity problem has N inputs and one output. The inputs are now data bits (a data word) and the output is the parity bit. The parity bit is set to be +1 if the total number of high bits in the data word is odd; otherwise it is set to -1. In the simulation there are 6 data bits (N = 6). All possible combinations are gone through ($2^6 = 64$). Thus, the number of training examples is 64. Since there were no testing examples available, the generalization properties of the network cannot be tested quantitatively.

The hearta problem is an analogue version of the heart disease diagnosis problem from the UCI machine learning repository [61]. The single continuous output predicts heart disease and decides the number of major vessels which are reduced in diameter by less than 50%. The decision is made based upon 13 input attributes which include age, sex, smoking habits, and subjective pain descriptions, and so on. The hearta dataset comprises 690 training examples and 230 testing examples.

The cancer problem contains some diagnosis results for breast cancer. Based on cell descriptions gathered by microscopic examination, a tumour is classified as benign or malignant. The dataset was created based upon the breast cancer Wisconsin problem dataset from the UCI machine learning repository [61]. The output represents the classification result for the purpose of breast cancer diagnosis. The decision is based on nine input attributes which include cell thickness, the uniformity of cell size, and cell shape. The number of training samples is 200 and the number of testing samples is 167.

3.4.3 Error metrics used to determine convergence

The error metrics used in the simulations were CERR (Classification ERRor) for classification problems, and NMSE (Normalized Mean Square Error) for regression problems e.g.

$$CERR = \frac{1}{2m}\sum_i \left| sgn\left(y_{ki}\right) - sgn\left(\theta_{ki}\right)\right| \tag{3.9}$$

and

$$NMSE = \frac{1}{m\sigma^2}\sum_i \left(y_{ki} - \theta_{ki}\right)^2 \tag{3.10}$$

where m is the number of training patterns, y_{ki} is the target output of the ith input pattern, θ_{ki} is the ith network output, sgn is the sign function of a number (i.e. if the number if negative, then the sgn function returns -1, otherwise it returns +1), and σ^2 is the variance of the target output data.

3.4.4 Discussion of results

N-bit parity. The first test is a generalization of Xor with *N* set to 6. In this test the convergence behaviour of the first tangent plane algorithm was compared with the back-propagation algorithm. A standard two hidden layer network was used with 6 inputs, and one output. The number of units in each hidden layers was increased from 10 to 40 units in steps of 10. 20 trials were carried out with the mean number of steps to converge, standard deviation, and number of successful trials recorded. Network training was terminated when the number of presentations of the entire dataset exceeded 1,000, or when the classification error was reduced to 5 x 10^{-2}. A further test was carried out with training terminated when the error was reduced to zero to give a harder measure of convergence

The results are tabulated in Table 3.1a and 3.1b. It was found that the tangent plane algorithm converged faster than the back-propagation algorithm, except in the smallest network where the convergence speed was slower and there were more failed trials. The most significant gains were made by the tangent plane algorithm in the largest networks when reducing the classification error to zero, the slow asymptotic behaviour of the back-propagation algorithm producing very slow convergence. Many of the trials carried out using the tangent plane algorithm in the smallest network would get stuck in local minima, which had a deleterious effect on the success rate. The same behaviour was observed in the back-propagation algorithm, but this algorithm appeared to be far more robust with regards to the size of the network.

Fig. 3.3 and 3.4 show typical convergence behaviour for the tangent plane algorithm and the back-propagation algorithm. A standard 6-15-15-1 network was used with training terminated after 200 epochs. The training curves for the tangent plane algorithm are very steep with convergence occurring rapidly within 50 epochs. Two curves contain small hills indicating the turbulent nature of network training. The training curves for the back-propagation algorithm are much smoother which suggests smaller weight updates. Convergence occurs within 100 epochs, with the exception of one curve, which appears to get stuck on a flattish plateau before converging. This is probably due to oscillatory behaviour caused by using a momentum term to accelerate the learning

(a)

	Avg. number of epochs to reduce CERR below 5.0 x 10^{-2}			Avg. number of epochs to reduce CERR to zero		
HU	Mean	Std Dev	Succ	Mean	Std Dev	Succ
10	314	268	8	235	84	6
20	32	12	20	56	29	20
30	20	6	20	27	8	20
40	13	2	20	18	6	20

(b)

	Avg. number of epochs to reduce CERR below 5.0 x 10^{-2}			Avg. number of epochs to reduce CERR to zero		
HU	Mean	Std Dev	Succ	Mean	Std Dev	Succ
10	85	35	20	500	290	13
20	34	9	20	68	30	20
30	21	6	20	33	9	20
40	15	3	20	20	7	20

Table 3.1. Mean number of steps to converge, standard deviation and number of successful trials (Succ) for standard networks with different numbers of hidden units HU on the 6-bit parity problem: (a) tangent plane algorithm, and (b) back-propagation algorithm with momentum ($\eta = 0.01$, $\alpha = 0.7$)

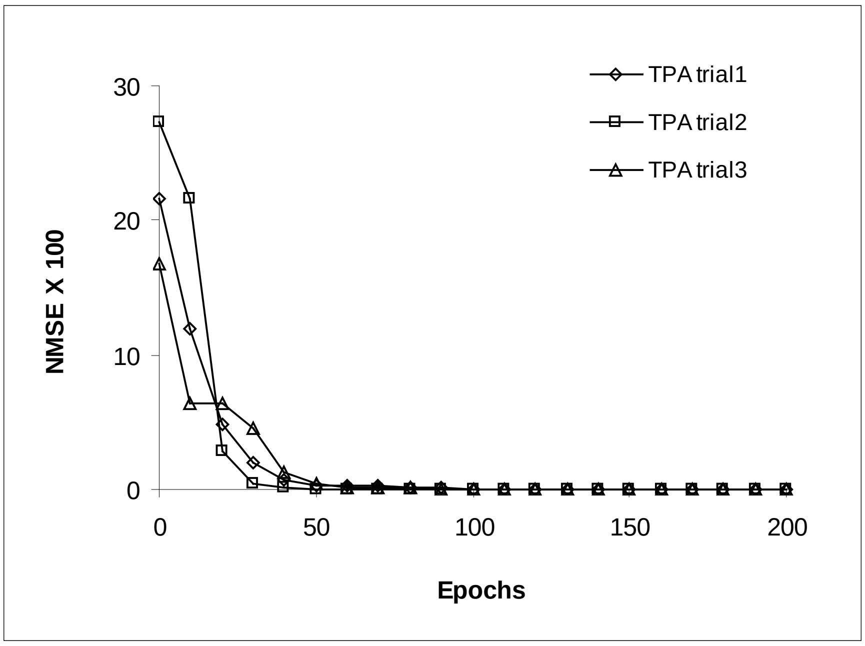


Fig. 3.3. Typical convergence behaviour of the first tangent plane algorithm on the 6-bit parity problem

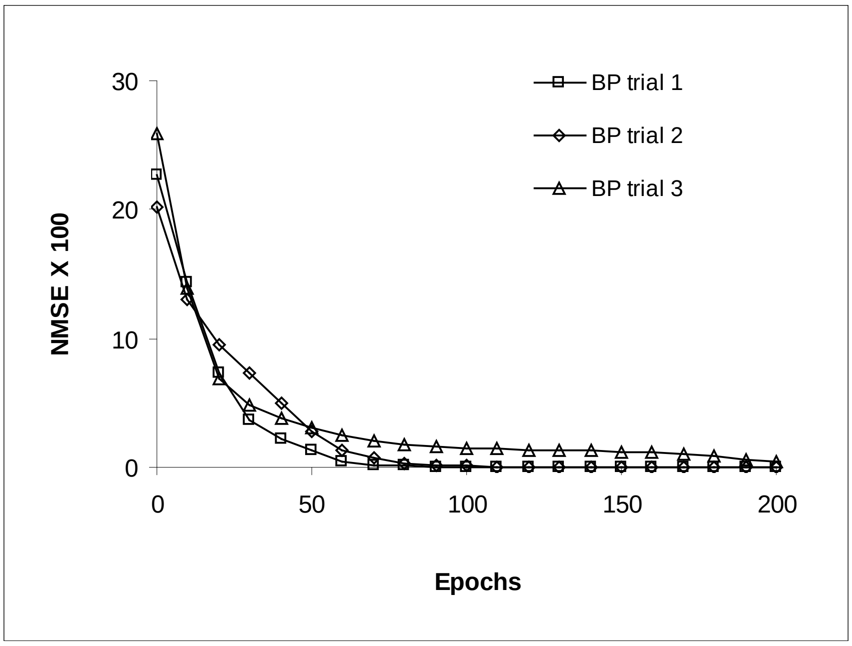


Fig. 3.4. Typical convergence behaviour of the back-propagation algorithm on the 6-bit problem (η = 0.01, and α = 0.7)

Training with inexact data. A further test was performed with data that had been partially corrupted in order to assess the robustness of the first tangent plane algorithm. A standard 6-15-15-1 network is assumed with 6 inputs, one output and 15 hidden units in two hidden layers. Two different types of inexact data were used. The first type of inaccurate data involved adding small random fluctuations to the teaching values. Each teaching variable was randomised to a value in the range $\left(y_k - \delta, y_k + \delta \right)$. The second type of inaccurate data involved using single items of rogue data presented occasionally into the network. The training data was generated in the usual way, but at epochs 4, 8, 12, …, 196, 200, the corresponding teaching values were given a 0%, 2% and 5% probability of being randomised in the range [-1,1]

Table 3.2a and 3.2b give the results for the first type of inexact data. The results are averaged over 20 trials. It was found that the tangent plane algorithm was tolerant of low levels of noise. All trials succeeded with the noise set at δ = 0.01, and 0.05. The situation with the back-propagation algorithm was the same. Increasing the level of noise had a deleterious effect on convergence speed with the tangent plane algorithm completely failing to converge for the highest level of noise. This behaviour is expected as the tangent plane algorithm uses the target data as a constraint to be satisfied. Clearly where the input patterns change from one presentation to the next so that no exact solution exists, the tangent plane algorithm will find it difficult to converge on a solution, and instead continue to hop around weight space.

Fig 3.5 and 3.6 shows the results for the second type of inexact data. The same set of initial weights was used in each figure. The training curves for the tangent plane algorithm drop sharply at first showing that the ability of the algorithm to converge upon a solution has not been impaired by the presence of noise. Thereafter the training curves exhibit turbulent behaviour that persists without diminution. The recovery time of the network after the presentation of the randomised data is typically four epochs. Increasing the error rate increased the occurrence of large spikes in the training curves. The noisy training curves for the back-propagation algorithm show slight perturbations about the curve with no noise added. Clearly the smaller steps taken have averaged out the fluctuations in the training data leading to good asymptotic behaviour

(a)

	Avg. number of epochs to NMSE below 10^{-2}			Avg. number of epochs to NMSE below 10^{-3}		
δ	Mean	Std Dev	Succ	Mean	Std Dev	Succ
0.00	31	10	20	51	11	20
0.01	32	9	20	52	12	20
0.05	33	12	20	62	12	20
0.10	33	8	20			

(b)

	Avg. number of epochs to NMSE below 10^{-2}			Avg. number of epochs to NMSE below 10^{-3}		
δ	Mean	Std Dev	Succ	Mean	Std Dev	Succ
0.00	38	13	20	59	15	20
0.01	42	25	20	60	19	20
0.05	37	9	20	101	45	20
0.10	43	11	20	336	111	11

Table 3.2. Mean number of steps to converge, standard deviation and number of successful trials (Succ) on the 6-bit parity problem for networks trained using fuzzy data: (a) tangent plane algorithm, and (b) back-propagation algorithm ($\eta = 0.01$, and $\alpha = 0.7$)

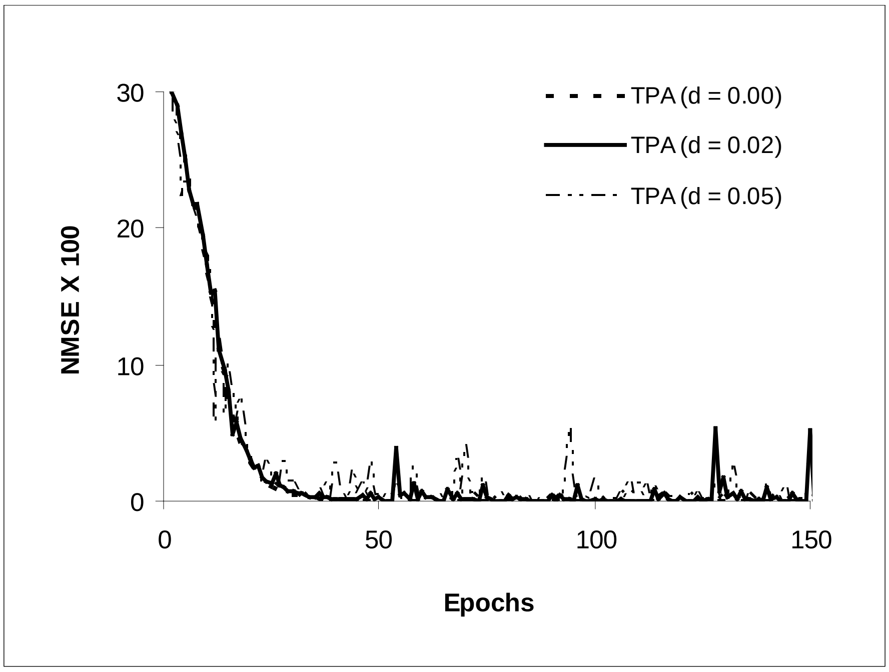


Fig. 3.5. Typical convergence behaviour of the tangent plane algorithm with the error rate for randomising the teaching values set at 0%, 2%, and 5%

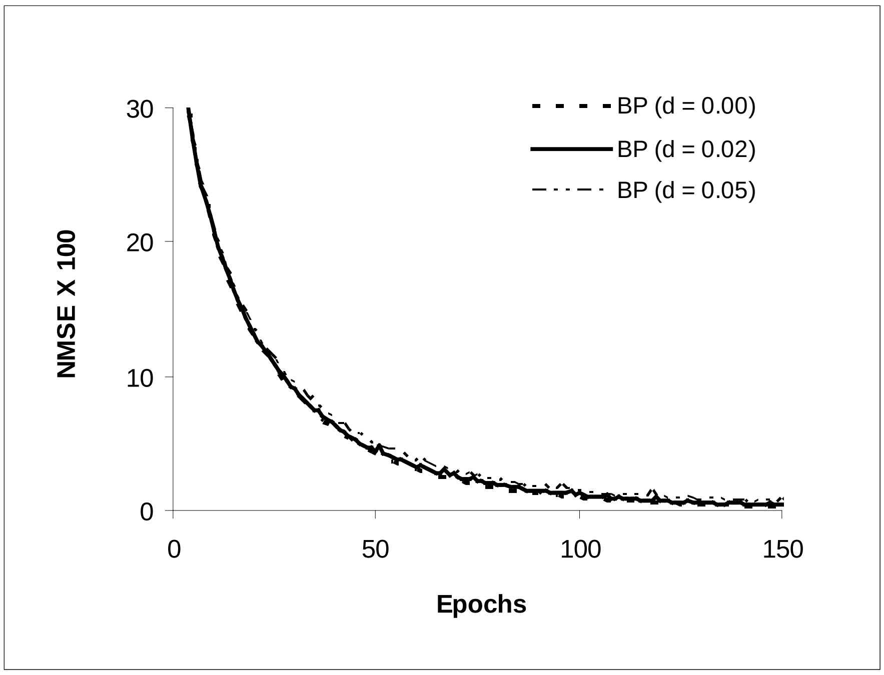


Fig. 3.6. Typical convergence behaviour of the back-propagation algorithm with momentum with the error rate for randomising the teaching values set at 0%, 2%, and 5%

Hearta problem. The second test is an analogue version of the heart disease problem from the Proben1 collection [34]. In this test the generalization performance of the second tangent plane algorithm was compared with the back-propagation algorithm. A standard feed-forward neural network was used with 9 inputs, one output, and two hidden layers. The number of units in each hidden layer was increased from 5 to 20 in steps of 5. 20 trials were carried out with the mean square error on the training set and the test set recorded together with the mean number of steps to converge. Network training was terminated using the method of early stopping.

In preliminary tests it was found that the convergence behaviour of the second tangent plane algorithm could be significantly improved by introducing a progressive stiffening of the step size. An exponential schedule $exp\left(-t/\tau\right)$ was used for this purpose with the time constant set at τ = 5,000. The results are tabulated in Table 3.3a. It was found that the generalization capability of the second tangent plane algorithm was comparable with the back-propagation algorithm, except in the smallest network where it was worse. Generalization was found to be independent of network size. Decreasing the value of the time constant had a beneficial effect on generalization but this was at the expense of the convergence speed, which was much slower

Fig. 3.7 and 3.8 show the generalization behaviour of the second tangent plane algorithm and the back-propagation algorithm. A standard 13-20-20-1 network was used with 13 inputs, 20 hidden units in each of two hidden layers and one output. The generalization curves of the second tangent plane algorithm are fairly smooth. One of the curves dips to a clearly defined minimum after 50 epochs. Two curves rise fairly steadily showing mild overtraining. The generalization curves of the back-propagation algorithm show the same behaviour. One curve dips to a local minimum after 50 epochs. There is evidence of mild overtraining in all the curves. Increasing the momentum rate had a deleterious effect on generalization, and produced mild turbulence in all the curves

(a)

	Second tangent plane algorithm Avg. validation set error using early stopping (NMSE)			Back-propagation algorithm Avg. validation set error using early stopping (NMSE)		
HU	Err	Err*	Steps	Err	Err*	Steps
10	0.27	0.35	121	0.21	0.31	59
20	0.16	0.34	88	0.18	0.32	43
30	0.12	0.34	70	0.16	0.34	39
40	0.11	0.34	52	0.17	0.37	25

(b)

	Second tangent plane algorithm Avg. validation set error using early stopping (CERR x 10^2)			Back-propagation algorithm Avg. validation set error using early stopping (CERR x 10^2)		
HU	Err	Err*	Steps	Err	Err*	Steps
10	4.35	5.14	135	4.37	4.30	82
20	3.19	4.27	62	3.97	5.79	72
30	2.70	3.78	46	4.23	6.99	37
40	2.21	3.65	41	4.03	7.51	36

Table 3.3. Training set error (Err) and test set error (Err*) for different sized networks with training terminated using early stopping: (a) hearta1 problem, (b) the breast cancer problem

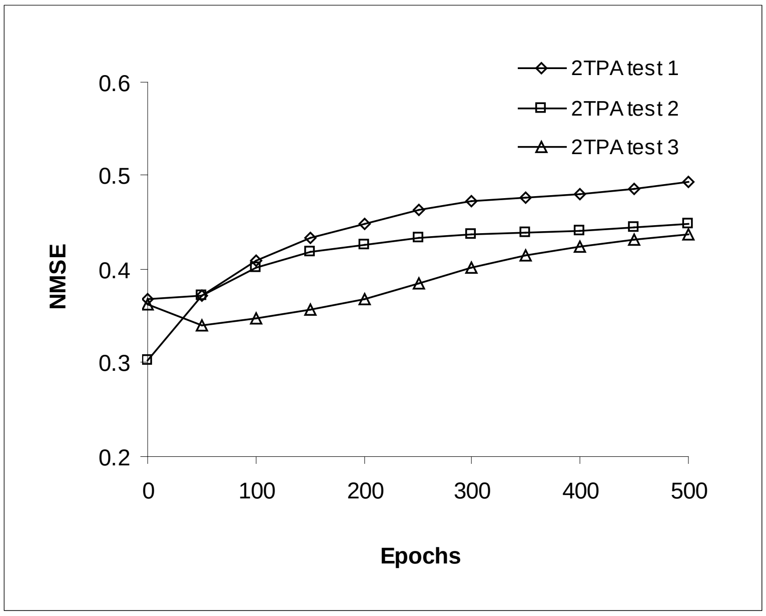


Fig. 3.7 Typical generalization behaviour of the second tangent plane algorithm on the hearta problem (*tan*β = 0.1, and τ = 5,000)

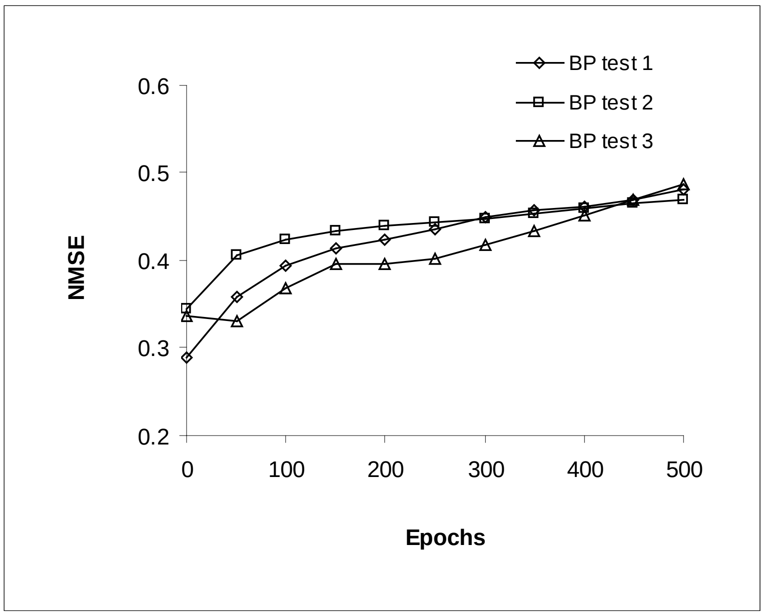


Fig. 3.8 Typical generalization behaviour of the back-propagation algorithm on the hearta problem (η = 0.01, and α = 0.3)

Cancer problem. The final test utilized data from the Wisconsin state breast cancer dataset, which is a real-world problem from the Proben1 collection [34]. In this test the generalization performance of the second tangent plane algorithm was compared with the back-propagation algorithm. A standard feed-forward neural network was used with 9 inputs, one output, and two hidden layers. The number of units in each hidden layer was increased from 5 to 20 in steps of 5. 20 trials were carried out with the classification error on the training set and test set recorded together with the number of steps taken to converge. Network training was terminated using the method of early stopping.

In preliminary tests it was found that the performance of the second tangent plane algorithm could be considerably improved by introducing a progressive stiffening of the step size. An exponential annealing schedule was used for this purpose (τ = 10,500). The results are tabulated in Table 3.3b. It was found that generalization was significantly better in networks trained by the second tangent plane algorithm, except in the smallest network where it was slightly worse. Increasing the size of the angle parameter ($tan\beta$ = 0.05, 0.10) had a deleterious effect on the convergence speed and generalization performance.

Fig. 3.9 and 3.10 show the generalization behaviour of the second tangent plane algorithm and the back-propagation algorithm. A standard 9-20-20-1 network was used with 9 inputs, 20 hidden units and one output. The generalization curves of the second tangent plane algorithm dip to a local minima at 50 epochs. Thereafter the curves rise to a flattish plateau. Very little learning occurs after 200 epochs. The generalization curves of the back-propagation algorithm dip to a local minimum at 50 epochs. Two curves contain slight undulations that persist without diminution. There is mild evidence of overtraining in one curve.

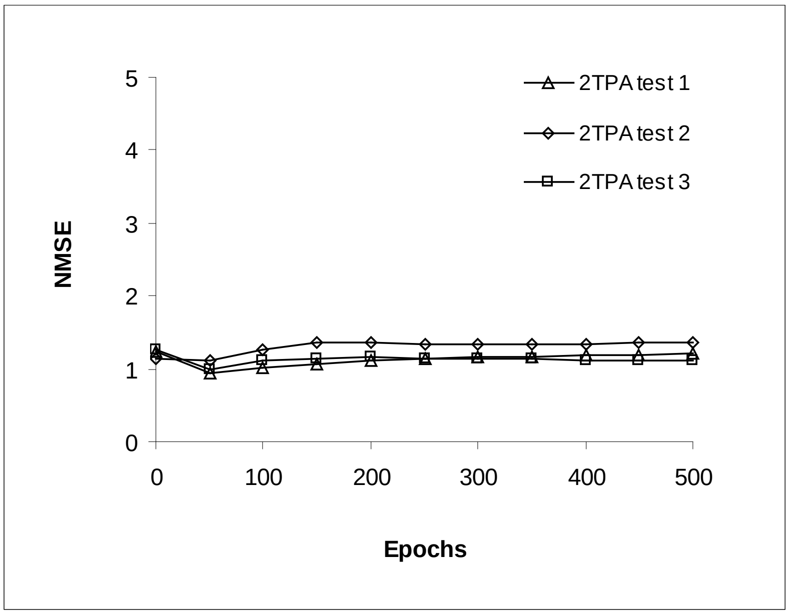


Fig. 3.9 Typical generalization behaviour of the tangent plane algorithm on the cancer problem (*tanβ* = 0.02, and τ = 10,500)

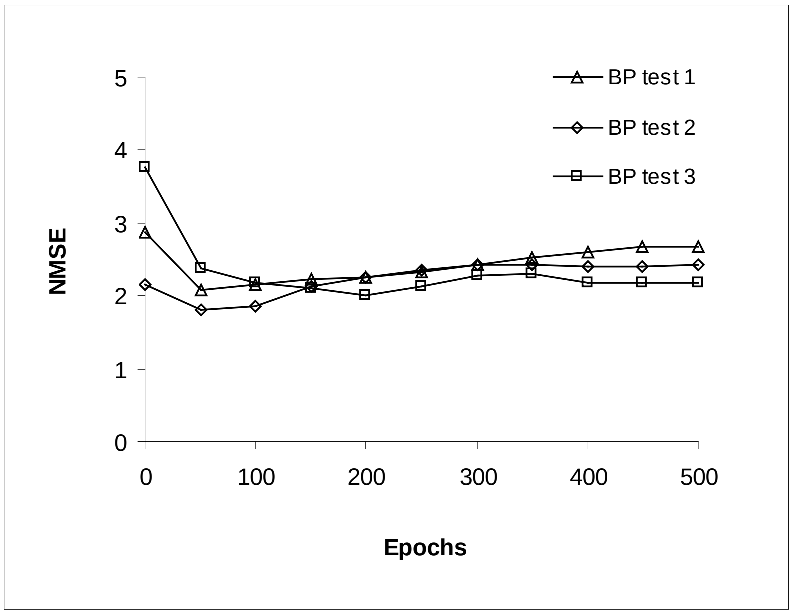


Fig. 3.10 Typical generalization behaviour of the back-propagation algorithm on the cancer problem (η = 0.01, and α = 0.7)

3.4.5 Discussion of weight sensitivity values

Next, we turn our attention to the differentiation and evolution of the weights in networks trained by the second tangent plane algorithm. The matter of growing weights from small initial values raises an important question, namely does the algorithm activate only the necessary number of weights, or does it activate all the weights leading to a large network structure? To answer this question, we will examine the importance of weights in the network. There are many methods to calculate the importance of weights such as assuming that the importance is proportional to the magnitude of the weights [52]. More sophisticated methods include optimal brain damage [10], and optimal brain surgeon [11]. Both of these methods require training down to the error minimum. We will use the autoprune method [51,52] which uses a statistic T to allocate an importance coefficient to each weight during the training process.

Comparative tests were carried out using the second tangent plane algorithm and the standard back-propagation algorithm. In our experiment we utilise data from two benchmark datasets; the breast cancer problem, and the hearta problem obtained from the UCI machine learning repository [61]. A standard feed-forward network was assumed with two hidden layers and 25 units in each hidden layer. The importance coefficients of the weights were recorded from the same trial at different epochs. The coefficient sizes were grouped in classes of width one and histograms plotted to show the distribution of the T values at different stages of training. Network training was terminated after a preset number of epochs.

Breast cancer problem. Fig 3.11 and 3.12 show the histograms of the importance coefficients for both algorithms. The histograms were plotted at epochs 20, 60, and 100. The average values of the histograms were calculated using the arithmetic mean. The tangent plane algorithm gave histogram averages of 5.47, 6.10, and

7.01 respectively. The back-propagation algorithm gave 8.45, 8.05, and 8.16. Notice the drift to the right of the histograms produced by the tangent plane algorithm (see Fig 3.11). Notice also the skewness of the histogram after 20 epochs. Initially the weights are densely packed about the origin with only a few of the weights responding to the training data. However, as the training continues, the algorithm activates more of the weights by pushing the weight values further away from the origin. In contrast the histograms produced by the backpropagation algorithm are fairly symmetric with constant mean (see Fig 3.12). Prechelt [51] suggests pruning all the weights with coefficients less than 0.5 times the arithmetic mean of the coefficient sizes. After 20 epochs, this is approximately 9% of the weights in the network trained by the tangent plane algorithm and 4% of the weights in the network trained by the backpropagation algorithm.

Hearta problem. Fig 3.13 and 3.14 show the histograms of the importance coefficients for both algorithms. The histograms were plotted at epochs 20, 60, and 100. The average values of the histograms were calculated using the arithmetic mean. The tangent plane algorithm gave histogram averages of 4.09, 5.48, and 6.22 respectively. The standard back-propagation algorithm gave 6.59, 6.14, and 6.05. Once again, the histograms produced by the tangent plane algorithm show a drift to the right as training continues. Notice that the means and variances of the histograms produced by the backpropagation algorithm remain essentially the same. This suggests that the backpropagation algorithm learns the problem very quickly, so that after 20 epochs there is no learning taking place. Using Prechelt's pruning criteria, after 20 epochs approximately 14% of the weights in the network trained by the tangent plane algorithm are eligible for pruning, and 7% of the weights in the network trained by the backpropagation algorithm

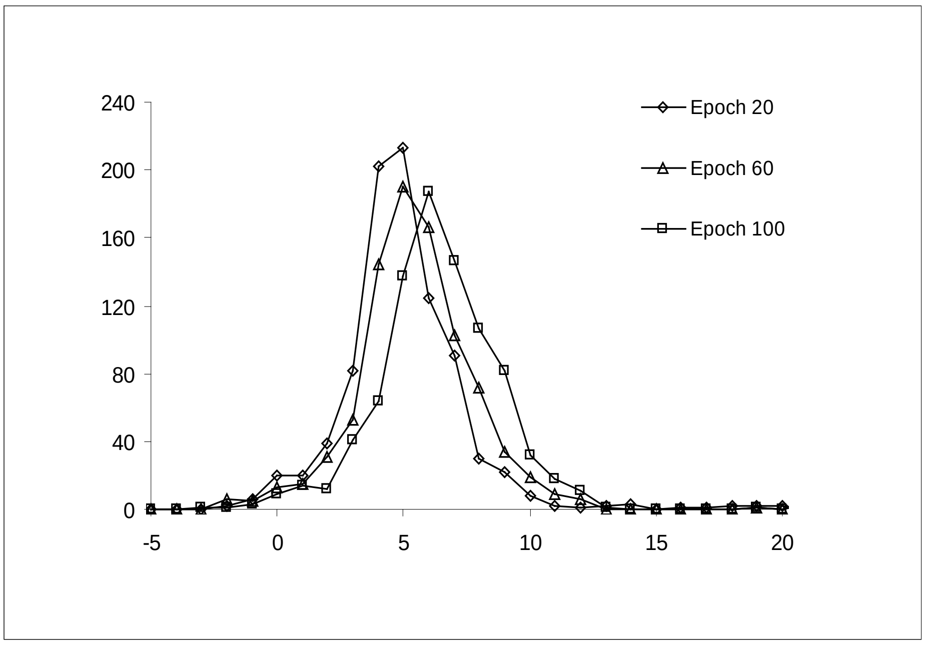


Fig. 3.11 Importance coefficient histogram for the tangent plane algorithm (cancer problem). Horizontal axis: coefficient size grouped in classes of width 1. Vertical axis: absolute frequency of weights with this coefficient size

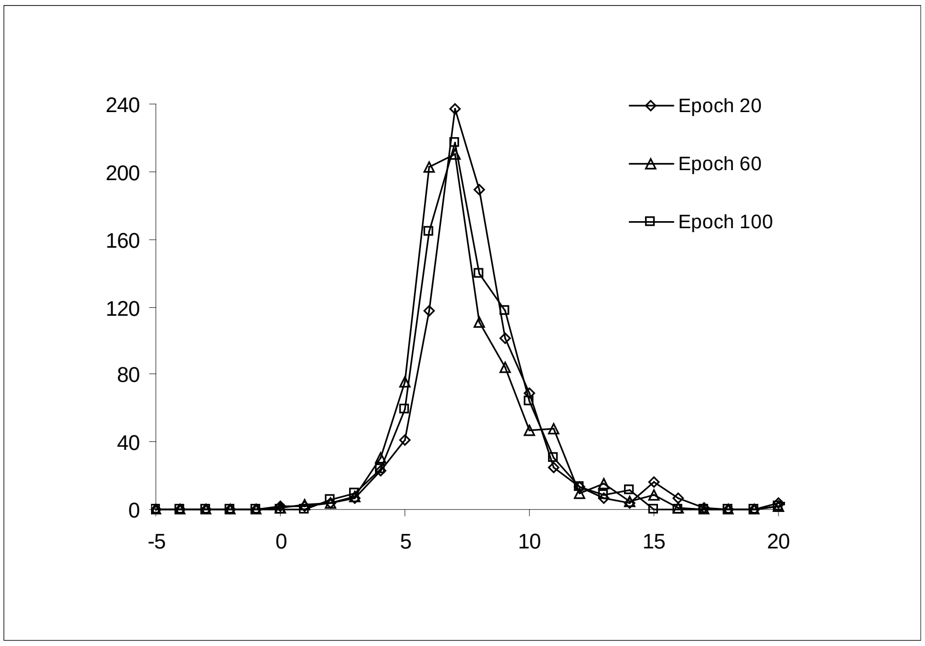


Fig. 3.12 Importance coefficient histogram for the back-propagation algorithm (cancer problem). Horizontal axis: coefficient size grouped in classes of width 1. Vertical axis: absolute frequency of weights with this coefficient size

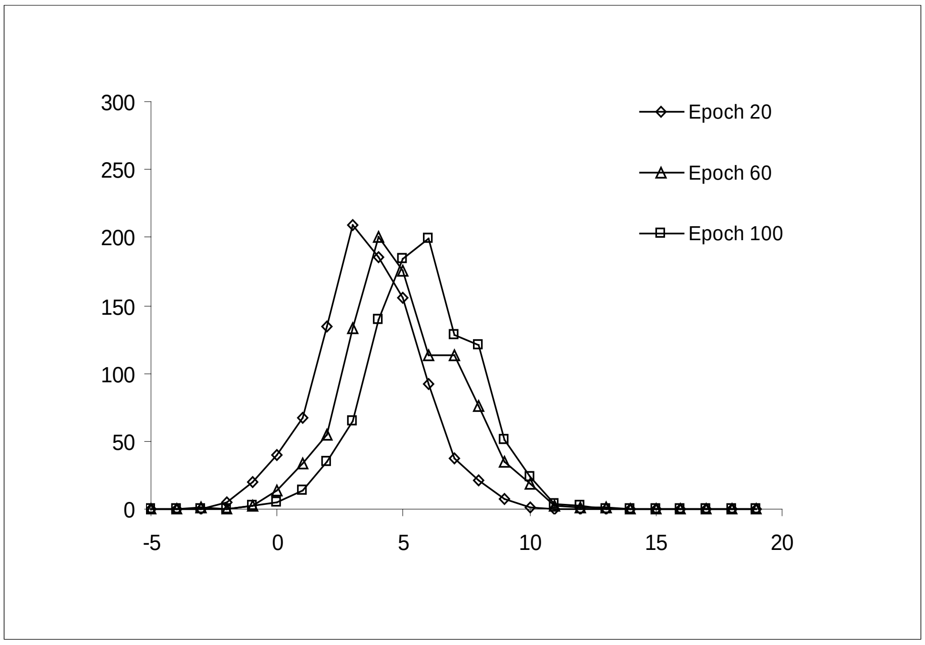


Fig. 3.13 Importance coefficient histogram for the tangent plane algorithm (hearta problem). Horizontal axis: coefficient size grouped in classes of width 1. Vertical axis: absolute frequency of weights with this coefficient size

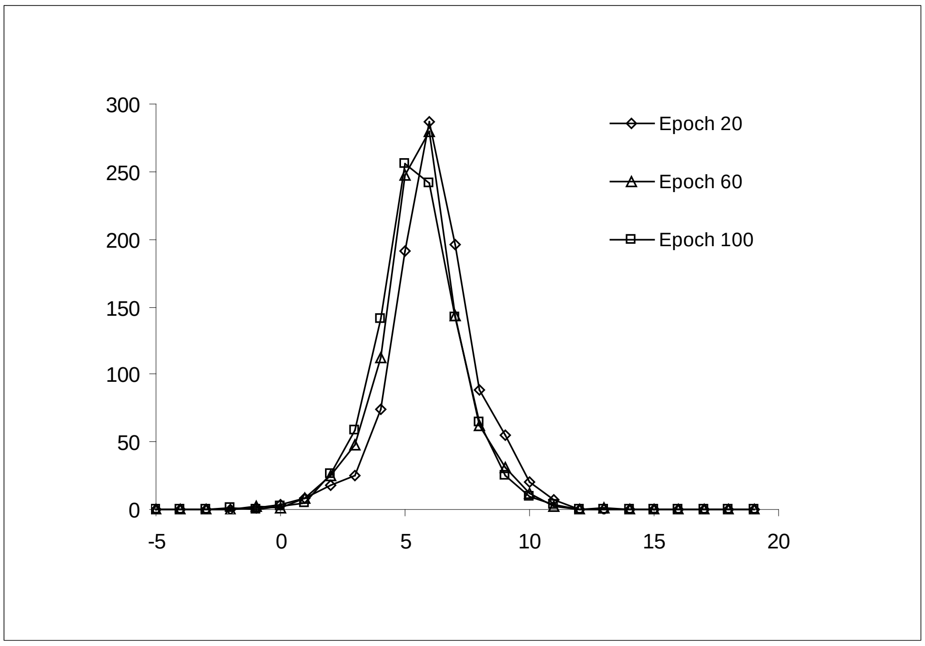


Fig. 3.14 Importance coefficient histogram for the back-propagation algorithm (hearta problem). Horizontal axis: coefficient size grouped in classes of width 1. Vertical axis: absolute frequency of weights with this coefficient size

3.5 Problems with the tangent plane algorithm

There are a number of difficulties with the tangent plane algorithm. These difficulties are summarised as follows

- For certain starting conditions of the weights, the tangent plane algorithm would get stuck in local minima that slow down the convergence. This behaviour is worse in small parsimonious networks where generalization is known to be best. Investigation into this stucking behaviour showed that it occurred whenever the tangent planes of two or more consecutive input patterns were very nearly parallel. The tangent plane algorithm would then zigzag between these surfaces for much longer resulting in very slow convergence. This problem disappeared in larger networks. As noted by Lee [2], the convergence speed of the tangent plane will be faster in situations where the number of free parameters in the network is greater than the number of patterns to be learned. The rationale behind this idea is that the tangent plane algorithm will have more directions to move on in a higher dimensional weight space

- The tangent plane algorithm is extremely sensitive to errors in the training data. Two different types of erroneous data were investigated. The first type involved adding small random fluctuations to the teaching values, and the second type adding single items of false data to the training set. It was found that the tangent plane algorithm would not tolerate large fluctuations in the teaching values (e.g. $y_k \pm \delta$, where $\delta \in [-0.1,0.1]$), or a large percentage of erroneous data (typically > 5%). These results are to be expected in a supervised teaching algorithm that uses the teaching values as constraints to be satisfied by adjusting weights. Schiffmann [33] noted a similar problem with Schmidhuber's algorithm [27] when a small number of incorrect examples were present in the dataset.

- Starting the training with weights initialised to small values and then pushing the weights in the direction away from the origin tends to produce large network structures having wide distributions of weight values. Histograms of weight importance coefficients show a general drift of the weights towards higher importance values. This suggests a differential activation of the weights with only a few weights activated at first. However, as the training continues, more of the weights engage in the learning which results in an oversized network structure. This means that any advantage gained by setting the weights to small initial values is lost if the algorithm does not find a solution quickly

Chapter 4

A NEW SEQUENTIAL TANGENT PLANE ALGORITHM FOR RECURRENT NEURAL NETWORKS

In chapter three a comparative evaluation of the tangent plane algorithm and the back-propagation algorithm was performed for regression and classification tasks. The benchmark datasets used were 6-bit parity [61], hearta [34] and cancer [34]. All the datasets were obtained from the UCI machine learning repository [61]. The results show that the tangent plane algorithm gives good generalization relative to the backpropagation algorithm. Generalization was found to be independent of network size. However, the tangent plane algorithm finds it difficult to converge on a compromise solution when the data is inexact or contains a small number of erroneous patterns. In these circumstances the tangent plane algorithm will continue to hop around weight space, adjusting the weights of the network to satisfy each new constraint in turn. In contrast, the backpropagation algorithm gave improved convergence, smaller steps taken effectively smoothing out the fluctuations in the training data. In this chapter, a new tangent plane variant is developed for fully recurrent neural networks (FRNNs). FRNNs use feedback connections and state units to learn the relationships between temporal sequences. The new algorithm is based upon the real time recurrent learning algorithm (RTRL), which is a gradient descent based method for training FRNN [3]. RTRL has been used in many application areas such as real-time process control and speech enhancement. It is shown that that learning temporal sequences can improve the stability of the tangent plane algorithm when handling inexact data.

4.1 Improvement in the stability of the tangent plane algorithm

We have already seen that the tangent plane method cannot handle datasets that contain a small amount of erroneous data, or datasets where the teaching values are fuzzy so that they vary from one presentation to the next. To overcome this difficulty Lee [14] has suggested using a progressive stiffening of the step size as the network becomes trained. This stiffening was implemented using a scaling factor s^n applied to each weight change, where s is a parameter set to 1 / 1.0002 and n the time step. Although the ability of the tangent plane algorithm to converge was restored using this method, progressively stiffening the step size may have a dramatic slowing down effect if not properly tuned, the parameter s being dependent on the size of the training set. Moreover, the same quality of solution can easily be achieved using the backpropagation algorithm with a fixed learning rate. Fully recurrent neural networks (FRNNs) are powerful tools for learning temporal sequences. FRNNs have been used in a variety of applications that involve time varying signals e.g. process control [84], speech recognition [85,86], and removing artefacts from electroencephalogram (EEG) signals [87]. Thus it seems worthwhile investigating a tangent plane variant for FRNNs as a practical alternative to using feedforward neural networks on problems with inexact data

4.1.1 A brief introduction to recurrent neural networks

Fully recurrent neural networks (FRNN) are powerful computational models that can learn temporal sequences, either in online or batch mode. A diagram of an FRNN is shown in Fig 4.1. The FRNN consists of two layers, an input layer of linear units and an output layer of non-linear units. The input layer is fully connected to the output layer by adjustable weights. Furthermore, the FRNN features unit delay feedback connections that feed back the activations of the output units to the input layer units. The output units thus have some knowledge of their prior activations, which enables them to perform learning that extends over time. FRNNs accomplish their learning task by mapping input sequences, and delayed activations, to another set of output sequences. Due to the nature of feedback around the output units, these units may continue to cycle information through the network over multiple time steps, and thereby discover abstract representations of time.

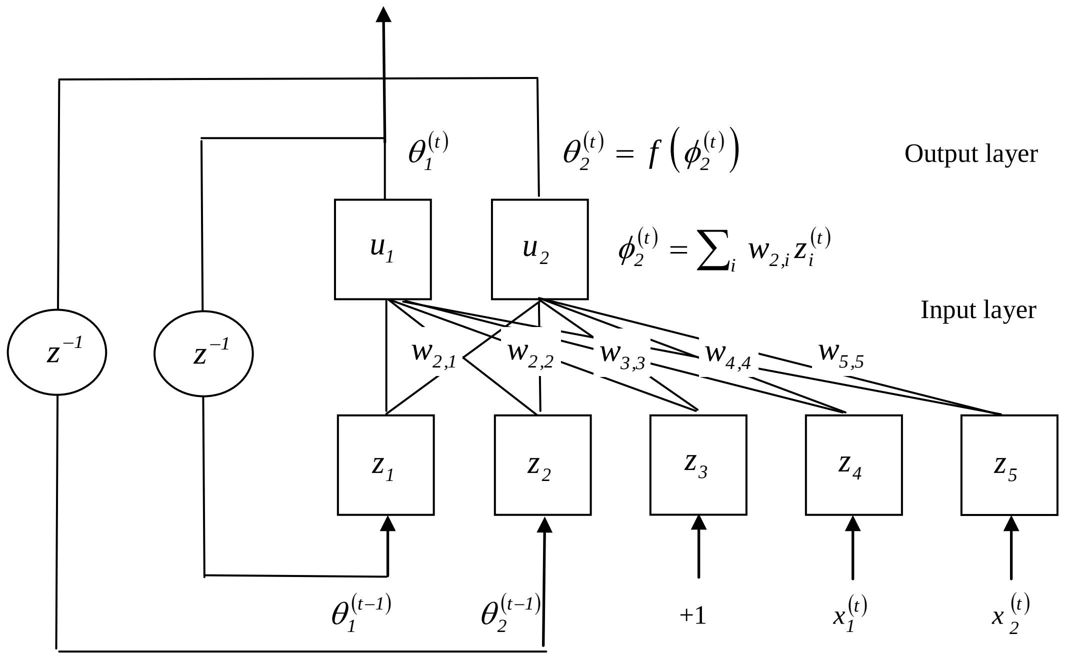


Fig 4.1. An example of a fully recurrent neural network with one output unit, one hidden unit, and two input units. The function z^{-1} is the unit delay operator whose output is delayed with respect to the input by one time step.

One of the most popular algorithms for training FRNNs is the real time recurrent learning (RTRL) algorithm [3]. RTRL is a gradient descent based algorithm for adjusting the output layer weights in a fully recurrent neural network. In the seminal paper by Williams and Zipser, two variations are presented, one for online and one for off-line (batch) learning. In both forms, RTRL has been used to train applications in a variety of areas such as speech enhancement [85,86], and real-time process control [84], where the output of the system is the response to current and previous input. RTRL has also been used to train FRNNs for next symbol prediction in an English text processing application [88]. Li et al [89] have used RTRL to train FRNNs for adaptive pre-distortion linearization of RF amplifiers. Finally RTRL has been used to train FRNNs that are capable of removing artefacts in EEG (electroencephalograms) signals [90].

There are a great many variants of RTRL present in the literature that are aimed at enhancing different aspects of the algorithm such as its computational complexity, convergence speed, and its sensitivity to the choice of initial weights. In Catfolis [91] a technique is presented to increase the performance of the RTRL algorithm by re-initializing it after specific time periods so that the gradient vector is dependent on fewer past values and the weights follow the true steepest descent direction more accurately. Also, the relationship between the slope of the activation function and the learning rate in the RTRL algorithm is explored in order to decrease the number of degrees of freedom on non-linear optimization problems in Mandic [92]. In Lu et al [93,94], a new mode exchange RTRL (MERTLR) algorithm was proposed which was designed to work in two modes, static and dynamic. In static mode some of the elements of the gradient matrix are fixed so that they do not change. This will reduce the time complexity to $O(n^3)$ in static mode. In dynamic mode the gradient matrix is computed as normal.

In this section we present a new RTRL variant based upon the tangent plane algorithm. While the original RTRL algorithm utilizes gradient information to guide the search towards the minimum training error, the new variant trains an FRNN by approaching the tangent planes to constraint surfaces defined in the weight space of the network. The motivation behind this new idea was to develop a more stable tangent plane algorithm capable of handling inexact data. Due to the presence of feedback connections, the FRNN will continue to recycle information over many time steps and thereby learn abstractions that extend over time. This may lead to a more robust tangent plane algorithm capable of predicting the correct response and not simply memorising an item of erroneous data.

4.1.2 Derivation of the new TPA-RTRL algorithm

Consider a FRNN of units $\{u_j\}$ (see Fig 4.1). For unit u_j, $\mathbf{w}_j^T$ = $[w_{j1}, w_{j2}, \ldots,$ $w_{j,(n+m+1)}]$ denotes a $(n+m+1)\times 1$ vector of weights, where n are the number of feedback connections and m the number of external inputs, one remaining input being the bias input weight. Let ϕ_j and θ_j denote the net input and output of u_j, and f the unit's activation function, typically $tanh(x)$. The following equations describe the FRNN at time instant t

$$\theta_j^{(t)} = f\left(\phi_j^{(t)}\right),\ j = 1, 2, \ldots, n \tag{4.1}$$

$$\phi_j^{(t)} = \sum_{l=1}^{n+m+1} w_{jl}^{(t)} z_l^{(t)} \tag{4.2}$$

$$[z_l^{(t)}]^{\mathrm{T}} = [\theta_1^{(t-1)}, \ldots, \theta_n^{(t-1)}, 1, x_1^{(t)}, \ldots, x_m^{(t)}] \tag{4.3}$$

The method assumes a FRNN with a single output unit. Let this output unit be denoted by u_1, with $\theta_1^{(t)}$ at time step t being trained to mimic the teaching value $y_1^{(t)}$. For a given set of inputs $x_i^{(t)}$, $i = 1, \cdots, m$, we can consider $\phi_1^{(t)}$ to be a function of the weights, $\phi_1^{(t)} : R^{n\times(n+m+1)} \to R$. Thus the equation $\phi_1^{(t)} = f^{-1}\left(y_1^{(t)}\right)$ defines a $n\times(n+m+1)-1$ surface in $R^{n\times(n+m+1)}$. The aim of the training procedure is to move from the current position in weight space to the nearest point on a tangent plane of this surface (see Fig 4.2)

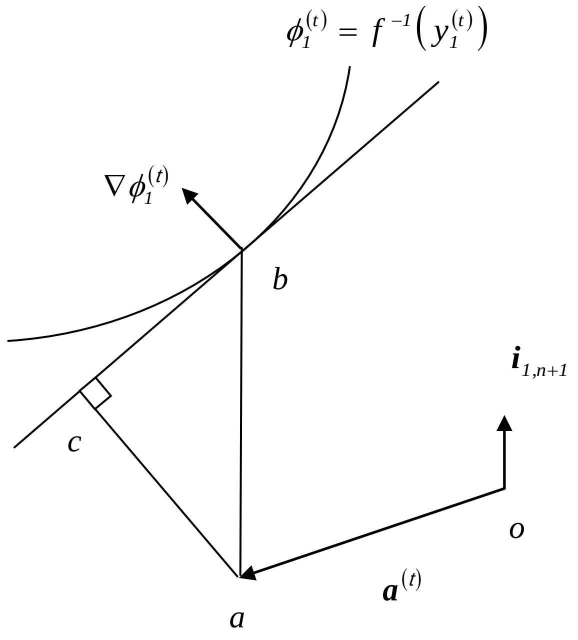


Fig 4.2. Movement from the present position **a** to the foot of the perpendicular to the tangent plane of constraint surface $\phi_1^{(t)} = f^{-1}\left(y_1^{(t)}\right)$ to position **c**

Let $\boldsymbol{a}^{(t)} = \sum_{j,i} w_{ji}^{'(t)}\boldsymbol{i}_{ji}$, where $\boldsymbol{i}_{ji}$ is a unit vector in the direction of the w_{ji} axis. Use the equation $f^{-1}\left(y_1^{(t)}\right) = w_{1,n+1}^{(t)} + \sum_{i \neq n+1} w_{1i}^{(t)} z_i^{(t)}$ to find a value, $w_{1,n+1}^{''(t)}$, for the bias weight $w_{1,n+1}$ from the values $w_{ji}^{'(t)}$ of the other weights, so that the surface $\phi_1^{(t)} = f^{-1}\left(y_1^{(t)}\right)$ contains the point $\boldsymbol{b}^{(t)} = w_{1,n+1}^{''(t)}\,\boldsymbol{i}_{1,n+1} + \sum_{j,i\neq 1,n+1} w_{ji}^{'(t)}\boldsymbol{i}_{ji}$.

Let $\hat{\boldsymbol{n}}^{(t)}$ be the unit normal to the surface at $\boldsymbol{b}^{(t)}$, so $\hat{\boldsymbol{n}}^{(t)} = \nabla\phi_1^{(t)} \big/ \left\|\nabla\phi_1^{(t)}\right\|$. The length of the perpendicular from $\boldsymbol{a}^{(t)}$ to the tangent plane at $\boldsymbol{b}^{(t)}$ is $\hat{\boldsymbol{n}}^{(t)} \boldsymbol{.}\, [\boldsymbol{b}^{(t)} - \boldsymbol{a}^{(t)}]$. Now, if we use the equation $f^{-1}\left(y_1^{(t)}\right) = w_{1,n+1}^{(t)} + \sum_{i \neq n+1} w_{1i}^{(t)} z_i^{(t)}$, we get

$$\begin{aligned} \boldsymbol{b}^{(t)} - \boldsymbol{a}^{(t)} &= [\, w_{1,n+1}^{''(t)} - w_{1,n+1}^{'(t)} \,]\ \boldsymbol{i}_{1,n+1} \\ &= [\, f^{-1}\left(y_1^{(t)}\right) - f^{-1}\left(\theta_1^{(t)}\right)]\ \boldsymbol{i}_{1,n+1} \qquad (4.4) \end{aligned}$$

Let $\delta^{(t)} = f^{-1}\left(y_1^{(t)}\right) - f^{-1}\left(\theta_1^{(t)}\right)$. If $\boldsymbol{c}^{(t)}$ is the foot of the perpendicular from $\boldsymbol{a}^{(t)}$ to the tangent plane at $\boldsymbol{b}^{(t)}$, then $\boldsymbol{c}^{(t)} - \boldsymbol{a}^{(t)} = ([\boldsymbol{b}^{(t)} - \boldsymbol{a}^{(t)}] \,\boldsymbol{.}\, \hat{\boldsymbol{n}}^{(t)})\ \hat{\boldsymbol{n}}^{(t)}$, and from equation (4.4) we have

$$\begin{aligned} \boldsymbol{c}^{(t)} - \boldsymbol{a}^{(t)} &= \delta^{(t)} \left(\boldsymbol{i}_{1,n+1} \boldsymbol{\cdot} \frac{\nabla \phi_1^{(t)}}{\left\| \nabla \phi_1^{(t)} \right\|} \right) \frac{\nabla \phi_1^{(t)}}{\left\| \nabla \phi_1^{(t)} \right\|} \\ &= \frac{\delta^{(t)} \nabla \phi_1^{(t)}}{\left\| \nabla \phi_1^{(t)} \right\|^2} \end{aligned} \tag{4.5}$$

Thus, the adjustment applied to weight w_{pq} can be written

$$\Delta w_{pq}^{(t)} = \frac{\delta^{(t)}}{\left\| \nabla \phi_1^{(t)} \right\|^2} \frac{\partial \phi_1^{(t)}}{\partial w_{pq}^{(t)}} \tag{4.6}$$

Using the chain rule of differentiation, the derivative $\partial \phi_1 \big/ \partial w_{pq}$ in equation (4.6) can be re-written as

$$\begin{aligned} \frac{\partial \phi_1^{(t)}}{\partial w_{pq}^{(t)}} &= \frac{\partial}{\partial w_{pq}^{(t)}} \sum_{j=1}^{n+m+1} w_{1j}^{(t)} z_j^{(t)} \\ &= \sum_{j=1}^{n} \frac{\partial \theta_j^{(t-1)}}{\partial w_{pq}^{(t)}}\ w_{1j}^{(t)} + \delta_{p1} z_q^{(t)} \\ &= \sum_{j=1}^{n} f'\left(\phi_j^{(t-1)} \right) \frac{\partial \phi_j^{(t-1)}}{\partial w_{pq}^{(t)}} w_{1j}^{(t)} + \delta_{p1} z_q^{(t)} \end{aligned} \tag{4.7}$$

where

$$\delta_{p1} = \begin{cases} 1 & \text{if } p = 1 \\ 0 & \text{otherwise} \end{cases} \tag{4.8}$$

Under the assumption, also used in the RTRL algorithm [9], that when the step size is sufficiently small, we have

$$\frac{\partial \phi_j^{(t-1)}}{\partial w_{pq}^{(t)}} \approx \frac{\partial \phi_j^{(t-1)}}{\partial w_{pq}^{(t-1)}} \tag{4.9}$$

A triple index set of variables π_{pq}^{j} can be used to denote the partial derivatives $\partial \phi_j / \partial w_{pq}$, $1 \le j,\, p \le n$, $1 \le q \le n+m+1$. We compute the values π_{pq}^{j} for every time step t and appropriate j, p, q as follows

$$\left[\pi_{pq}^{j}\right]^{(t)} = \sum_{m=1}^{n} f'\left(\phi_m^{(t-1)}\right)\left[\pi_{pq}^{m}\right]^{(t-1)} w_{jm}^{(t)} + \delta_{pj} z_q^{(t)} \tag{4.10}$$

Because we assume that the initial state of the FRNN has no functional dependence on the weights, we also have $[\pi_{pq}^{j}]^{(0)} = 0$.

The computational complexity of the TPA-RTRL algorithm is $O(n^4)$, where n is the number of processing units. This feature of the TPA-RTRL algorithm implies a heavy computational burden, especially when the network is scaled up. According to equation (4.10), π_{pq}^{j} is calculated by adding n products of the terms w_{jm}, π_{pq}^{m}, and $f'(\phi_m)$. Therefore, the number of operations involved for each π_{pq}^{j} is $2n$. There are $n^2(n+m+1)$ of π_{pq}^{j} in the network. Thus the total number of computations in equation (4.10) is $2n^3(n+m+1)$. The TPA-RTRL algorithm also requires the calculation of a global weighting term, $\sum_{p.q}\left(\pi_{pq}^{1}\right)^2$. This term is calculated by adding $n(n+m+1)$ squares of π_{pq}^{1}, increasing the computational burden by a further $2n(n+m+1)$ operations. The computational complexity of the

TPA-RTRL algorithm could be reduced by setting some of the π^{j}_{pq} to zero. The MERTRL algorithm [93,94] is a useful starting place for this, as it fixes the recurrent connections w_{pq} with $p \neq q$ (i.e. sets π^{j}_{pq} to zero). This will reduce the computational complexity to $O(n^3)$ on n^3 of the w_{pq}. An alternative approach might be to fix a preset number of randomly selected w_{pq} during each time step, the selection of w_{pq} changing from one time step to the next. This approach avoids the problem of constraining the weight change to a limited number of directions, which will prevent the algorithm from attaining a low MSE.

The method of adjusting weights by approaching the tangent planes could potentially lead to some very big weight updates. Large activations based on big weight updates fed back into the input layer may become a source of negative feedback and instability in the network [44]. In order to improve stability, it may be helpful to re-initialise the algorithm after a preset time period. Catfolis [91] suggested an improvement to the basic RTRL algorithm that involves using some a priori knowledge about the temporal requirements of the problem. The method involves setting the partial derivatives π^{j}_{pq} to zero every τ cycles so that the weight changes are based upon accumulated information over a time interval of τ. The reason for re-initialising π^{j}_{pq} is that some inputs will only have an influence on the network for a specific number of cycles, so accumulating information over a longer period may take the trajectory in weight space further away from the true trajectory taken by the gradient descent method. This is equivalent to adding noise to the negative gradient vector so that its direction may not point directly towards the minimum on the error surface

4.2 Implementation of the procedure

The following section is included to clarify the procedure for updating the weights of a network trained using the TPA-RTRL algorithm

1. Initialize

$$w_{pq} \in [-1,1],\ \pi^{j}_{pq} = 0,\ \forall\ j \in U,\ p \in U,\ q \in U \cup I \text{ s.t.}$$

$$U = \{1, \dots, n\},\ I = \{n+1, \dots, m\}$$

2. For each unit u_j, calculate θ_j using

$$\theta_j^{(t)} = \sum_{q \in U \cup I} w_{jq}^{(t)} z_q^{(t)},\ \forall\ j \in U$$

3. For each unit u_j, calculate π^{j}_{pq} using

$$\left[\pi^{j}_{pq}\right]^{(t)} = \sum_{m \in U} f'\left(\phi_m^{(t-1)}\right)\left[\pi^{j}_{pq}\right]^{(t-1)} w_{jm}^{(t)} + \delta_{pj} z_q^{(t)},\ \forall\ j, p \in U,\ q \in U \cup I$$

where δ_{pj} denotes the Kronecker delta

4. Calculate

$$\left\|\left[\boldsymbol{\pi}^{1}\right]^{(t)}\right\|^{2} = \sum_{p \in U,\, q \in U \cup I}\left(\left[\pi^{1}_{pq}\right]^{(t)}\right)^{2}$$

5. Calculate

$$\delta^{(t)} = f^{-1}\left(y_1^{(t)}\right) - f^{-1}\left(\theta_1^{(t)}\right)$$

6. For each weight w_{pq}, add

$$\Delta w_{pq}^{(t)} = \frac{\delta^{(t)}}{\left\|\left[\boldsymbol{\pi}^{1}\right]^{(t)}\right\|^{2}}\left[\pi^{1}_{pq}\right]^{(t)},\ \forall\ p \in U,\ q \in U \cup I$$

4.3 Simulations and results

Comparative tests were performed with the TPA-RTRL algorithm and the original GD-RTRL algorithm under a variety of initial conditions and using different network sizes. Three different datasets were used; pipelined Xor [3], simple sequence recognition [3], and the henon map time series [95]. The pipelined Xor problem was used to analyse the convergence behaviour of each algorithm on a simple pattern classification task. The simple sequence recognition and henon map time series problems were used to determine the ability of the network to configure itself so that it stores important information from an earlier stage in the input stream to help determine the output at a later time. For all the tests described here a fully recurrent neural network was used with the same unit trained to match specific target values at specified times. The tests were carried out using an Intel Pentium IV (2.67 GHz).

4.3.1 Network initialization

Both algorithms require parameters to be set manually. Preliminary tests showed that the best results were obtained with the parameters set as follows. First, the original GD-RTRL algorithm. For the pipelined Xor and simple sequence recognition problems, the learning rate was set to η = 4.0. For the henon map problem, the learning rate was set to η = 0.01. The input weights were set to random values in the range [-1.0, 1.0]. Next, the new TPA-RTRL algorithm. The input weights were set to random values in the range [-1.0,1.0].

4.3.2 Simulation problems

The Exclusive OR (XOR) problem is an example of a pattern classification task that cannot be solved using a single neuron. The input patterns are (-1,-1), (-1,1), (1,1) and (1,-1). The first and third patterns are in class -1, and the second and fourth patterns in class 1. The training examples were presented to the network in a random order, one each time step. Thus the epoch length is four cycles. The network was operated in a continuous mode, meaning that all the epochs were presented to the network after each other without telling the network something had happened. Tests were performed by varying the time delay between the presentation of an input pattern and training the network to match the corresponding teaching value. For each test, 50 trials were made with the mean number of steps to converge, the standard deviation, and the number of successful trials recorded. Network training was terminated when the error was reduced to below 10^{-2} for at least 20 epochs or 1,000 epochs elapsed

The simple sequence recognition problem has four inputs and one output. Two of the input lines, labelled *a* and *b*, serve a special purpose. The other two input lines, *c* and *d*, serve as distractors. At each time step one input line carries a 1, and all other input lines a -1. The network is trained to output a 1 when a 1 on the *b* line is immediately followed by a 1 on the *a* line, otherwise the output is -1. Once such a *b* occurs, its corresponding *a* is considered to be used up. An additional constraint was imposed that two 1s should be output every 16 time steps. Thus the epoch length is 16 cycles. Tests were carried out using different sized networks, and data that had been partially corrupted in order to determine the robustness of the TPA-RTRL algorithm. For each test, 50 trials were made with the mean number of steps to converge, standard deviation and number of successful trials recorded. Network training was terminated when the error was reduced to below 10^{-3} for at least 20 epochs or 1,000 epochs elapsed.

The henon map problem is a chaotic time-series prediction problem. The time series is computed by

$$x^{(t+1)} = 1 - c\left(x^{(t)}\right)^2 + b\,x^{(t-1)} \qquad (4.11)$$

where b= 0.3, c= 1.4, $x^{(0)}$= -0.361938, and $x^{(1)}$= 0.896601. The objective of the simulation is to train a network with one input and one output and various processing units to model the chaotic series generated by (4.11). Since x_{max} = 1.272967 and x_{min} = -1.284657, the input values were scaled in the range [-0.5, 0.5]. Tests were carried out using different sized networks. For each test, 5 trials were made with the mean square error and CPU time of the learning algorithms recorded. The mean square error (MSE) was measured by averaging the output square errors between 4,999,000 and 5,000,000 time steps. Network training was terminated after 5,000,000 time steps had elapsed.

4.3.3 Error metrics used to determine convergence

The error metric used in the simulations is MSE (Mean Square Error). The MSE is given by

$$MSE = \frac{1}{m} \sum_{i} \left(y_{ki} - \theta_{ki} \right)^2 \qquad (4.12)$$

where m is a predefined time interval (typically one epoch), y_{ki} is the target value of the ith input, and θ_{ki} is the ith model output

4.3.4 Discussion of results

Exclusive Or (Xor) with time delay. The first test is a simple non-linearly separable problem requiring at least two processing cycles to complete. The test was carried out using a FRNN network with three processing units. One of the processing units was trained to match the teaching signal at time t corresponding to the inputs presented to the network at time $t-\tau$, where the computational time delay τ was chosen to be one or two cycles (time steps). The results are tabulated in Table 4.1. It was found that the new TPA-RTRL algorithm gave significantly faster convergence than the original algorithm when the target values were delayed by one cycle relative to the inputs being Xored. It was also found that increasing the time delay between the inputs and corresponding target values had a deleterious effect on convergence, and that additional units had to be added to the network for convergence to occur. Generally speaking for longer delays than one cycle the network has to configure itself to have more than two hidden layers in order to match the required delay.

The final weight values of a typical FRNN network trained by the TPA-RTRL algorithm are given in Table 4.2. Notice the weights providing a feed-forward solution have become large, whilst the recurrent weights have become small. This suggests that the FRNN network has organised itself into a single hidden layered feed-forward neural network with outputs delayed by one cycle relative to the inputs. The function of the recurrent weights has become one of providing additional pathways during the learning stage.

Fig 4.3 and 4.4 show some typical training curves for both algorithms on the Xor problem. The training examples were split into groups of four with the network operated in continuous mode. Thus the epoch length is four cycles. The MSE was calculated over a strip length of 5 epochs. The training curves for the new TPA-RTRL algorithm show that convergence occurs rapidly, typically within 80 epochs. One of the curves (test 2) contains a slight dip which is probably due to the presence of turbulence caused by large weight updates, averaged over by the coarse sampling rate in Fig 4.3. The training curves for the original GD-RTRL algorithm show the same general trend, but this time convergence occurs typically within 200 epochs.

(a)

	New TPA-RTRL algorithm Training terminated after 1,000 epochs trained or MSE < 10^{-2}			Original GD-RTRL Training terminated after 1,000 epochs trained or MSE < 10^{-2}		
Units	Mean	Std	Succ	Mean	Std	Succ
3	88.38	66.98	40	357.5	138.62	40

(b)

	New TPA-RTRL algorithm Training terminated after 1,000 epochs trained or MSE < 10^{-2}			Original GD-RTRL Training terminated after 1,000 epochs trained or MSE < 10^{-2}		
Units	Mean	Std	Succ	Mean	Std	Succ
3	806.67	166.15	3			0
4	484.41	268.86	17	726.33	194.79	15
5	342..09	218.53	43	588.78	156.77	37

Table 4.1. Mean number of steps to converge, standard deviation and number of successful trials (Succ) for both algorithms on Xor problem: (a) one cycle delay, and (b) two cycle delay

U	B	I		R			T
1	1.88	0.00	0.00	-0.01	-1.98	1.99	+
2	1.49	-1.98	1.98	-0.02	0.15	-0.16	-
3	-1.97	-2.12	2.13	-0.01	0.15	-0.19	-

Table 4.2. Weight matrix for Xor with one-cycle delay. The columns labelled U, B, I, R, T indicate respectively: unit number, the bias weight, input weights, recurrent weights, and the teaching status where '+' indicates the presence of teaching value, and '-' no teaching value present

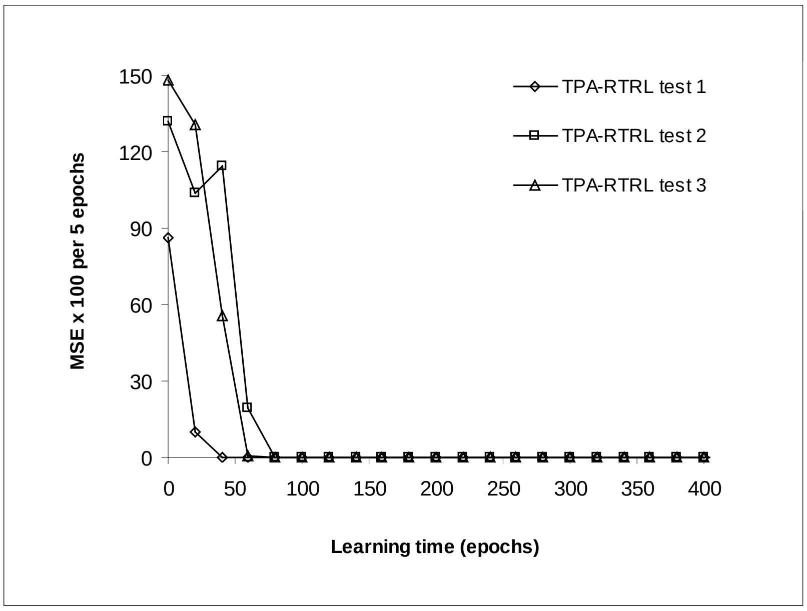


Fig. 4.3. Typical convergence behaviour of the new TPA-RTRL algorithm on the Xor problem with one unit trained to match the teaching signal of the inputs one cycle ago

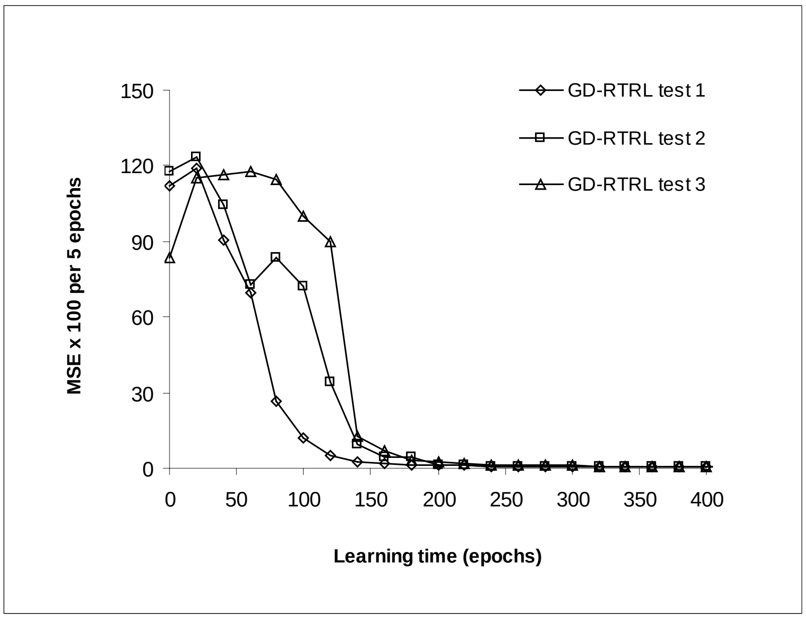


Fig. 4.4. Typical convergence behaviour of the original GD-RTRL algorithm on the Xor problem with one unit trained to match the teaching signal of the inputs one cycle ago

Simple sequence recognition. The second test is a simple sequence recognition task. The results are tabulated in Table 4.3. From the table we can see that that the new TPA-RTRL algorithm converges to a solution faster than the original RTRL algorithm, and that the convergence speeds of both algorithms improves with the size of the network. The improvement in the performance of the TPA-RTRL algorithm in larger networks can be explained as follows. If there are N patterns to be learned and $n \times (n+m+1)$ free parameters in the network, the probability of a pair of normals $\nabla\phi_1$ being nearly parallel, or a set of normals in weight space being nearly linearly dependent, will be reduced if $n \times (n+m+1) >> N$. Therefore a set of patterns should be learned more quickly by a network with a greater number of connections. Fig 4.5 and 4.6 show typical learning curves for both algorithms. The training examples were split into groups of 16. Thus the epoch length is 16 cycles. The learning curves for the TPA-RTRL algorithm show that convergence occurs rapidly, typically within 15 epochs. One curve (test 3) gets trapped in a local minimum. Convergence was restored by increasing the time delay between the presentation of an input pattern and the response of the network. The learning curves for the GD-RTRL algorithm show slow asymptotic behaviour with convergence occurring within 30 epochs.

A further test was carried out with the training data generated as normal, but at each presentation of an item of data the corresponding teaching value was given a 1%, 2% and 5% probability of being set at +1. The results are tabulated in Table 4.4. It was found that the TPA-RTRL algorithm is far more tolerant of noisy data, the slow convergence of the GD-RTRL algorithm producing a sluggish response that resulted in a higher proportion of failed trials. Fig 4.7 and 4.8 show some typical learning curves for a different type of inaccurate data. Once again, the training examples were split into groups of 16, so the epoch length is 16 cycles. The training data was generated as normal, but this time a pattern selected at random had its teaching value set at +1 during 12, 24 and 36 epochs. The learning curves for the TPA-RTRL algorithm show a pronounced peak at 12 epochs that corresponds to the first item of corrupted data. Subsequent responses to noisy data at epoch 24 and 36 diminish in amplitude. In each case, the network recovers quickly within two epochs after being presented with an item of corrupted data. The learning curves for the GD-RTRL algorithm show small peaks occurring at 24 and 36 epochs, the peak at 12 epochs being averaged over by the higher training error.

	New TPA-RTRL algorithm Training terminated after 5,000 epochs trained or MSE < 10^{-3}			Original GD-RTRL Training terminated after 5,000 epochs trained or MSE < 10^{-3}		
Units	Mean	Std	Succ	Mean	Std	Succ
2	47.50	53.45	48	215.84	43.76	50
4	21.04	11.01	50	127.70	21.55	50
6	18.66	8.49	50	102.74	13.90	50
8	18.00	13.15	50	90.56	11.74	50

Table 4.3. Mean number of steps to converge, standard deviation and success rate (Succ) for the new TPA-RTRL algorithm and original GD-RTRL algorithm on the simple sequence problem for different sized networks

	New TPA-RTRL algorithm Training terminated after 5,000 epochs trained or MSE < 10^{-3}			Original GD-RTRL Training terminated after 5,000 epochs trained or MSE < 10^{-3}		
n (%)	Mean	Std	Succ	Mean	Std	Succ
0	26.16	42.45	50	128.22	16.52	50
1	26.01	15.61	50	164.42	28.80	50
2	23.00	11.32	50	322.33	196.86	50
5	44.00	28.15	50			

Table 4.4. Mean number of steps to converge, standard deviation and success rate (Succ) for the new TPA-RTRL algorithm and original GD-RTRL algorithm on the simple sequence problem with n (%) of teaching values randomised

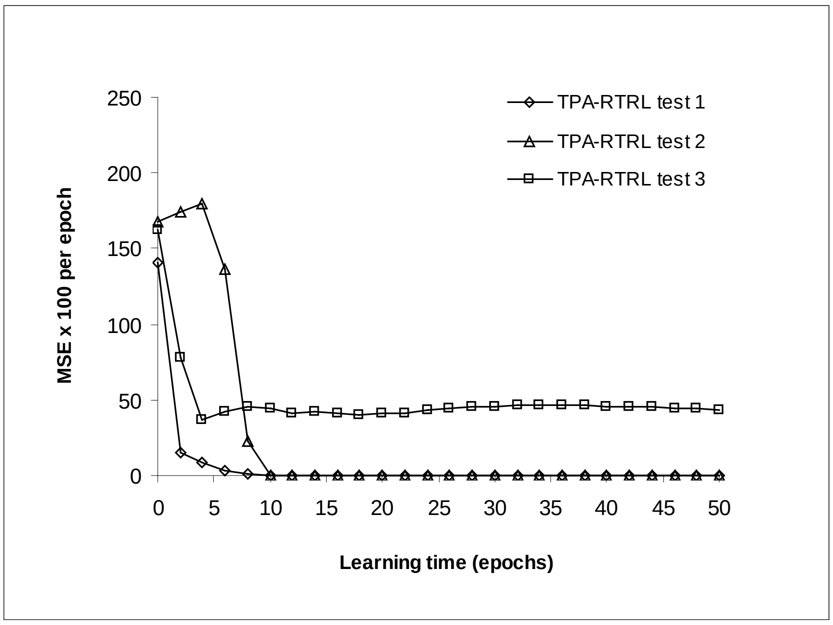


Fig. 4.5. Typical learning curves of the new TPA-RTRL algorithm on the simple sequence problem for a network with four processing units. Note that 1 epoch = 16 cycles

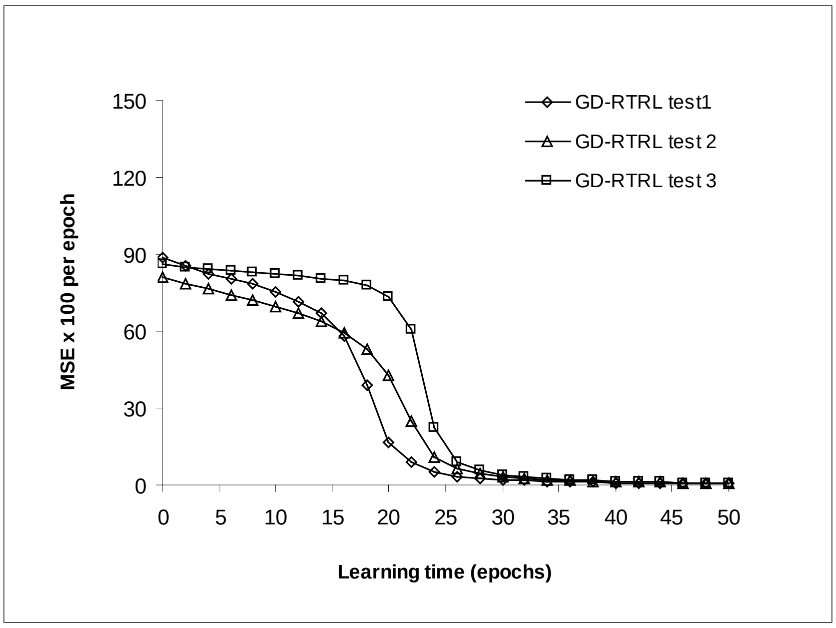


Fig. 4.6. Typical learning curves of the original GD-RTRL algorithm on the simple sequence problem for a network with four processing units. Note that 1 epoch = 16 cycles

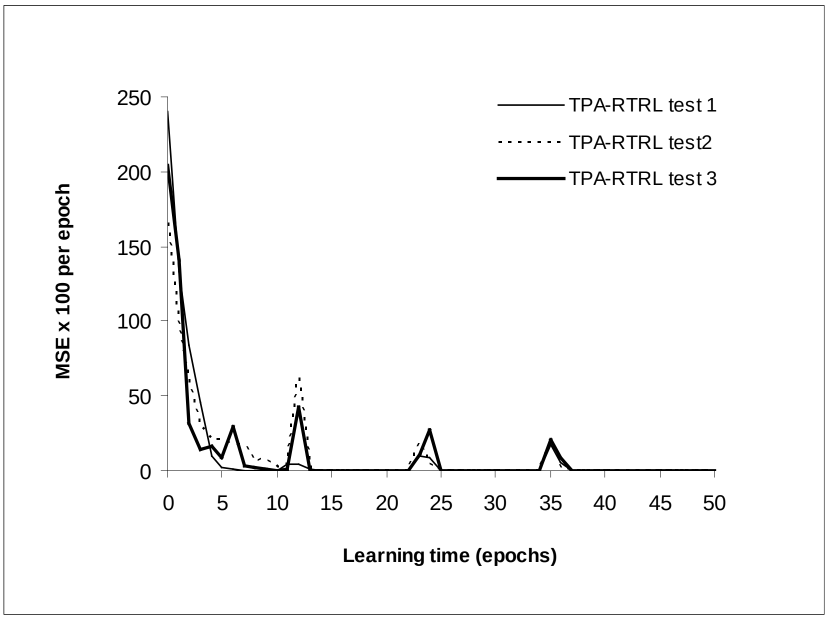


Fig. 4.7. Learning curves of the new TPA-RTRL algorithm for a network with four processing units. One teaching value has been corrupted at epoch 12, 24 and 36

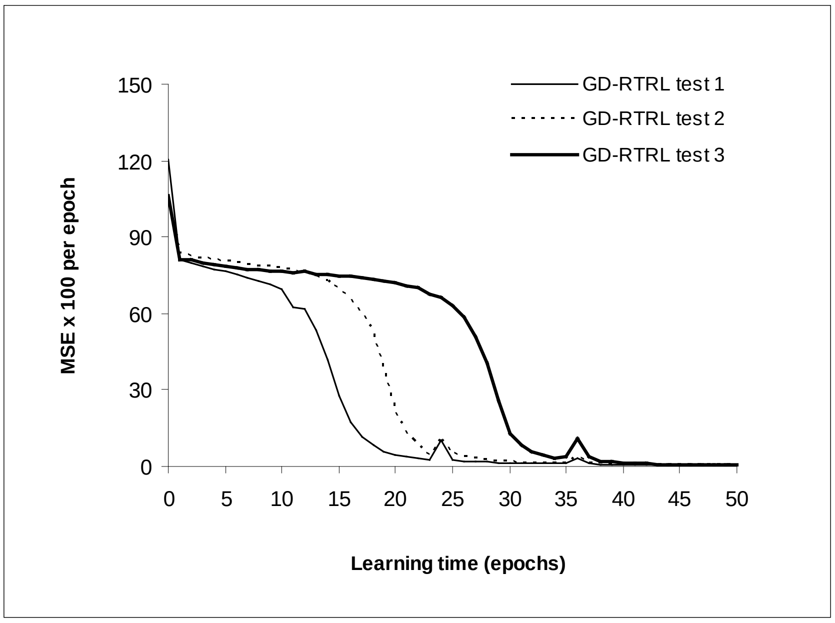


Fig. 4.8. Learning curves of the original GD-RTRL algorithm for a network with four processing units. One teaching value has been corrupted at epoch 12, 24 and 36

Henon map time series. The third test is a classical one-step-ahead prediction problem. Table 4.5 shows the mean square error and CPU time in seconds of the learning algorithms for various processing units. The mean square error was obtained by averaging the error over 1,000 time steps. It was found that the TPA-RTRL algorithm gave faster convergence relative to the original algorithm, and that the convergence speed improved with the number of processing units. The slower convergence of the original algorithm can be explained by very slow asymptotic behaviour, and the tendency of the network to get trapped in local minima. In contrast the much larger steps taken by the TPA-RTRL algorithm made this method of learning an effective global minimizer. However, the faster convergence of the new algorithm was paid for at the expense of the CPU time used. The training error is very similar to those given by Mak, Lu and Ku [93]. The mean square error was RTRL (6 units = 0.008, 9 units = 0.0008, 12 units = 0.0006), and MERTRL (6 units = 0.001, 9 units = 0.003, 12 units = 0.001). The poor performance of the MERTL algorithm is because some of the weights were seldom adapted, which prevents the algorithm from attaining a low mean square error. Figure 4.9 and 4.10 show some typical convergence curves for both algorithms. The learning curves for the TPA-RTRL algorithm show good asymptotic behaviour, whilst the curves for the original algorithm get trapped in a stucking state after a few epochs.

	Units = 6		Units = 9		Units = 12	
	MSEx10^2	CPU Time	MSEx10^2	CPU Time	MSEx10^2	CPU Time
TPA-RTRL	0.021	440	0.007	862	0.004	2147
GD-RTRL	0.018	208	0.013	694	0.022	1804

Table 4.5. Number of epochs trained and test set error for the henon map time series prediction problem. Note that 1 epoch = 1,000 cycles

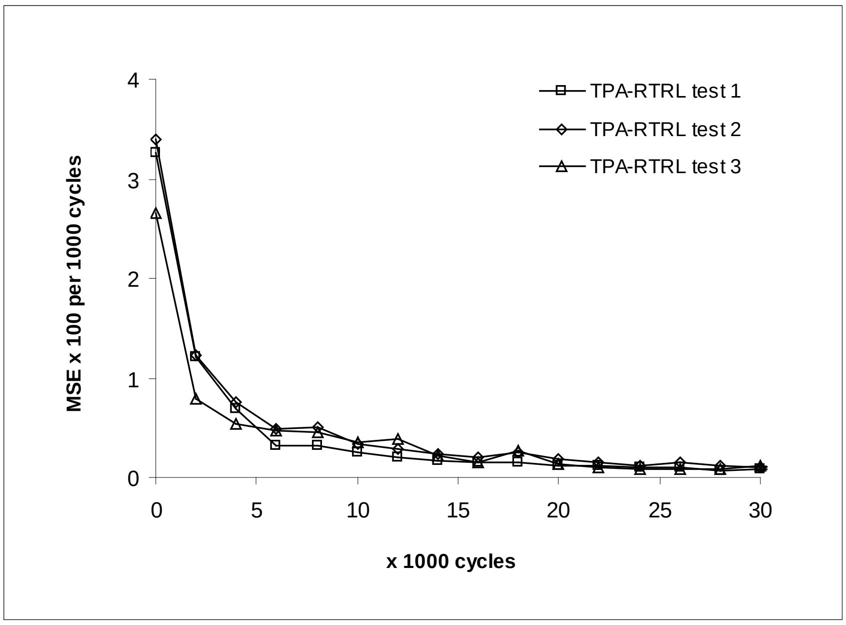


Fig. 4.9. Typical convergence behaviour of the new TPA-RTRL algorithm on the henon map time series prediction problem

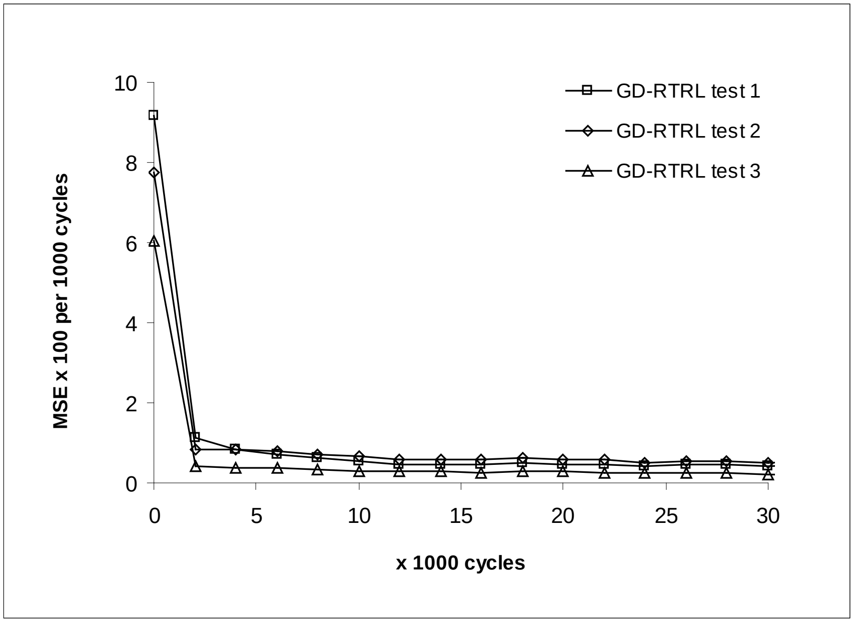


Fig. 4.10. Typical convergence behaviour of the original GD-RTRL algorithm on the henon map time series prediction problem

4.4 Summary

In this chapter, a new algorithm referred to as TPA-RTRL is proposed for training fully recurrent neural networks (FRNN). Recurrent neural networks contain feedback connections and state units to encode the temporal relationships between input data sequences. The new algorithm is based upon the real time recurrent learning (RTRL) algorithm proposed by Williams and Zipser [3], which has been used in many application areas such as real-time process control and speech enhancement. It is shown that learning to predict temporal sequences improves the stability of the tangent plane algorithm when the network is presented with a small amount of erroneous data. Several suggestions are made to improve the computational efficiency of the TPA–RTRL algorithm which include fixing the off-diagonal recurrent connections, and fixing a preset number of randomly selected connections at each time step

Comparative tests were carried out using the TPA-RTRL algorithm and the original GD-RTRL algorithm. The benchmark datasets used were pipelined Xor, and the simple sequence recognition problem, and the henon map time series. The results show that the new TPA-RTRL algorithm is very fast and stable. It can operate in feed-forward mode by organising a fully recurrent neural network into a conventional feed-forward neural network. It can also recover quickly when presented with small amounts of erroneous data. The results also show that the new algorithm is capable of producing high model accuracies on a non-trivial deterministic chaotic time series and that it outperforms RTRL and MERTRL in terms of accuracy. However, the runtimes may be prohibitively long in large networks

In the next chapter, a new variant of the tangent plane algorithm is proposed for feed-forward neural networks. This new algorithm includes two modifications to the original algorithm. Firstly, a directional movement vector is introduced into the training process to push the movement in weight space towards the origin. This movement vector simulates weight decay, which is known to have a beneficial effect on generalization in back-propagation learning. The directional movement vector is modified to give a heavier weighting to weights with small weight values. Secondly, a random sideways movement along tangent planes is introduced into the training process. This improves the likelihood of finding a good solution with small weights values (which can help generalization)

Chapter 5

A NEW SEQUENTIAL TANGENT PLANE ALGORITHM FOR FEED-FORWARD NEURAL NETWORKS

In chapter three, the second tangent plane algorithm was evaluated for function approximation and classification tasks on three neural network benchmark problems. This second tangent plane algorithm introduces a novel way of activating the weights in a neural network for good generalization to occur. It accepts almost zero starting conditions and moves away from the origin in weight space in a direction indicated by the training data. Compared with the back-propagation algorithm, the second tangent plane algorithm gives good generalization across a range of network sizes; the back-propagation algorithm generalizes well but only in small networks. However, the second tangent plane algorithm did not produce the expected separation of weights into active and inactive groups. Histograms of weight importance coefficients show that both the mean and variance of the distributions were observed to increase. This suggests that the tangent plane algorithm activates an increasing number of weights, each taking on more important roles within the network. In this chapter, two modifications to the tangent plane algorithm are suggested to overcome this difficulty. Firstly, a directional movement vector is introduced into the training process to push the movement in weight space towards the origin. This movement vector will encourage weight decay, which is known to have a beneficial effect on generalization. The directional movement vector is modified to give a heavier weighting to weights with small weight values. Secondly, a random sideways movement along tangent planes is introduced into the training process. This improves the likelihood of finding a good solution with smaller weight values (which can help generalization).

5.1 Improvement in the generalization of the tangent plane algorithm

Previously, we have investigated the evolution and differentiation of weights in networks trained by the second tangent plane algorithm. In order to determine whether a weight was active or not, the importance of the weight was calculated using the autoprune method. Autoprune [51,52] uses a statistic to allocate each weight an importance coefficient for the assumption that a weight becomes zero. Examination of the values of importance coefficients in networks trained by the second tangent plane algorithm show that both the mean and variance of the distribution of coefficient values tends to increase. This suggests that each weight takes on increasingly important roles in the network, so any advantage gained by starting the training with almost zero initial conditions is soon lost. It is well known that weight decay has a beneficial effect on generalization [12,13,52]. In the weight decay procedure the network itself removes superfluous weights by penalizing weights with small values. Thus an alternative strategy improve generalization in the tangent plane algorithm might be to start the training from arbitrary initial conditions, and then push the weights in a direction that encourages weight decay. The introduction of a directional component of movement along the tangent planes and towards the origin would have this effect

5.1.1 A brief introduction to pruning and weight decay

The principal idea of pruning is to reduce the number of free parameters in the network by removing superfluous weights from the network. If applied properly, it often reduces overfitting and improves generalization. The key to pruning is estimating the importance of a connection. Several such methods have been suggested. The simplest method estimates the importance of a weight based upon its magnitude [12]. More sophisticated methods include optimal brain damage (OBD) and optimal brain surgeon (OBS). OBD [10] uses a diagonal approximation of the Hessian of the error with respect to each weight to determine the saliency of the removal of that weight. Weights with low saliency are removed from the network. OBS [11] avoids the drawbacks of estimating the Hessian by computing the second derivatives almost exactly. Both methods require that the network is trained to the error minimum

Another approach is to have the network remove the superfluous weights itself. This can be achieved by giving each connection a tendency to decay to zero so that connections disappear unless they are reinforced. The simplest method is to subtract a small proportion of a weight after it has been updated [13]. This is equivalent to adding a penalty term $\gamma \sum_{j,i} w_{ji}^2$ to the original error function ε_k and performing gradient descent on the resulting total error. While this method clearly penalizes more w_{ji}'s than necessary, it overly discourages large weights. This can be cured by using a different penalty term $\mu \sum_{ji} w_{ji}^2 \;/\; (1 + w_{ji}^2)$ such that the small w_{ji}' s decay faster than the larger ones [101]. Simulations [102,103] using this penalty term show that no overtraining was observed and that the network was reduced to the optimum number of hidden units. Other regularisation methods may involve not only the weights but various derivatives of the output function [104], and sensitivity measures based on the significance of hidden units [105].

In this section we present a new sequential learning algorithm referred to as iTPA based upon the tangent plane algorithm. Whilst the original algorithm accepts almost-zero initial conditions and moves away from the origin, the new algorithm starts the training with weights initialized to arbitrary values and moves in a direction that encourages weight elimination. The motivation behind this new idea was to develop a tangent plane algorithm capable of building small economical networks by removing superfluous weights. A further motivation was to avoid the large weight values caused by moving in the direction away from the origin in weight space. Large weight values are known to have a harmful effect on the generalization capabilities of neural networks.

5.1.2 Derivation of the new iTPA algorithm

The basic structure of a feed-forward neural network is shown in Fig 5.1. It consists of an input layer of units that supply information, or activations, to the inputs of units in the first hidden layer. These in turn supply activations to units in the next layer, and so on. Typically the units in each layer receive inputs from the output of the units in the preceding layer. Let w_{ji} denote the connection between unit u_i and u_j. ϕ_j and θ_j will be the input and output of u_j, so that $\theta_j = f(\phi_j)$ and $\phi_j = \sum_i w_{ji}\theta_i$ for some monotonic function f.

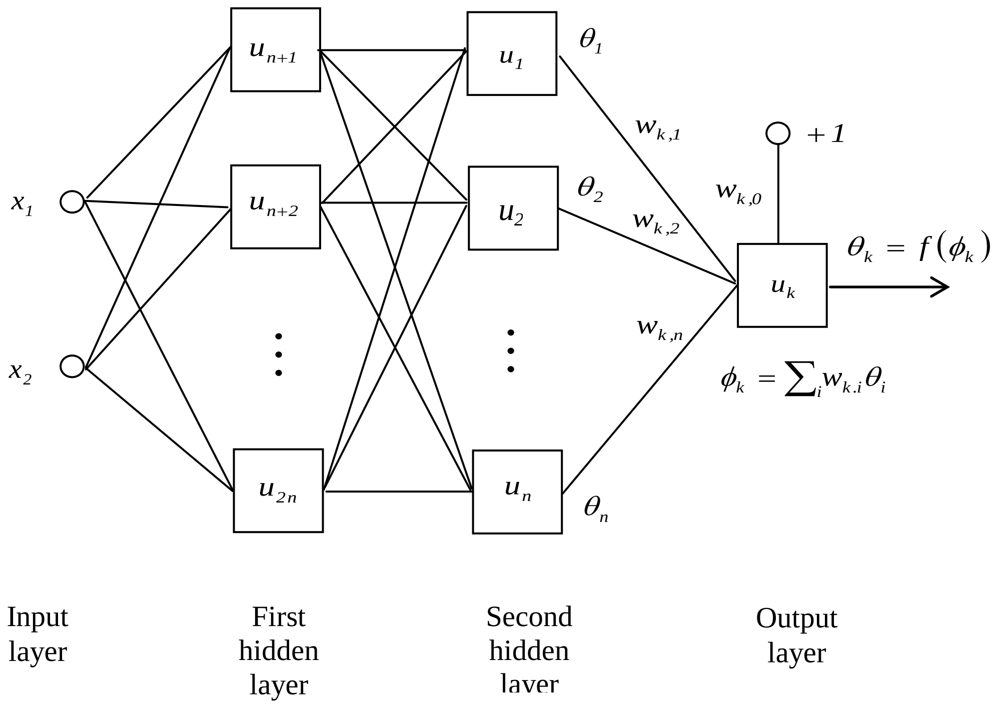


Fig. 5.1. The structure of a feed-forward neural network

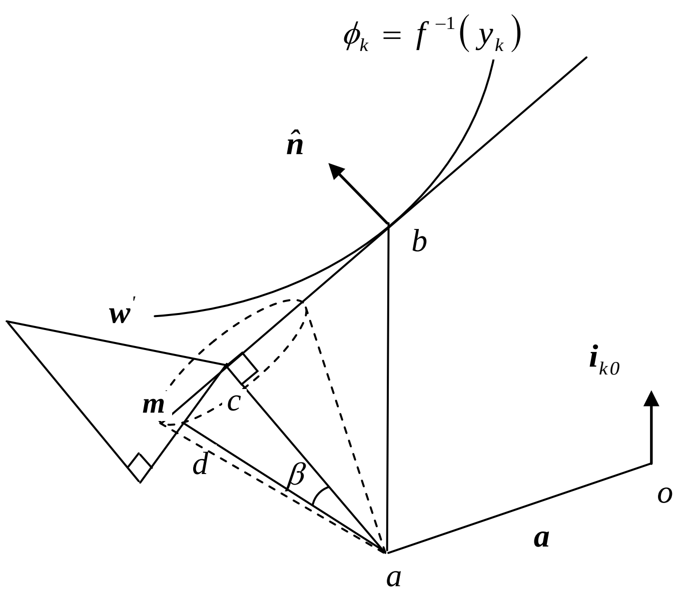


Fig. 5.2. Movement from the present position ***a*** to the point ***d*** inclined at an angle β to the perpendicular from ***a*** to the tangent plane to the constraint surface $\phi_k = f^{-1}(y_k)$ at point ***b*** in the weight space R^n. The vector ***m*** represents the orthogonal projection of the weight elimination vector ***w'*** orthogonally onto the normal ***n*** to the constraint surface at point ***b***

Let the single output unit u_k be trained to mimic the target value y_k, and let u_0 be the constant output unit, with θ_0 = 1. Let n denote the number of weights in the network. The current state of the weights is represented by $\boldsymbol{a} \in R^n$. For a given set of inputs we can consider ϕ_k to be a function of the weights $\phi_k : R \to R^n$. The second tangent plane algorithm adjusts the weights by moving along the line at an angle β to the perpendicular from the current position ***a*** to the $(n-1)$ tangent plane to the surface $\phi_k = f^{-1}(y_k)$, on the side of the perpendicular away from the origin (see Fig 5.2).

Let $\boldsymbol{a} = \sum_{j,i} w'_{ji}\boldsymbol{i}_{ji}$ be the current values of the weights, where $\boldsymbol{i}_{ji}$ is a unit vector in the direction of the w_{ji} axis. Use the equation $f^{-1}(y_k) = w_{k0} + \sum_{i\neq 0} w_{ki}\theta_i$ to find a value, w''_{k0}, for the bias weight w_{k0} from the values w_{ji} of the other weights, so that the surface $\phi_k = f^{-1}(y_k)$ contains the point $\boldsymbol{b} = w''_{k0}\,\boldsymbol{i}_{k0} + \sum_{j,i\neq k,0} w'_{ji}\boldsymbol{i}_{ji}$. Now, if we use the equation $f^{-1}(y_k) = w''_{k0} + \sum_{i\neq 0} w'_{ki}\theta_i$ and $f^{-1}(\theta_k) = w'_{k0} + \sum_{i\neq 0} w'_{ki}\theta_i$, and note that $\boldsymbol{b}$ differs from $\boldsymbol{a}$ only in the value of w_{k0}, we get

$$\begin{aligned} \boldsymbol{b} - \boldsymbol{a} &= \left(w''_{k0} - w'_{k0} \right)\boldsymbol{i}_{k0} \\ &= \left(f^{-1}(y_k) - f^{-1}(\theta_k) \right)\boldsymbol{i}_{k0} \end{aligned} \tag{5.1}$$

Let $\hat{\boldsymbol{n}}$ be the unit normal to the surface at $\boldsymbol{b}$, so $\hat{\boldsymbol{n}} = \nabla\phi_k / \|\nabla\phi_k\|$. The length of the perpendicular from $\boldsymbol{a}$ to the tangent plane at $\boldsymbol{b}$ is $(\boldsymbol{b} - \boldsymbol{a}).\hat{\boldsymbol{n}}$. If $\boldsymbol{c}$ is the foot of the perpendicular from $\boldsymbol{a}$ to the tangent plane at $\boldsymbol{b}$,

$$\begin{aligned} \boldsymbol{c} - \boldsymbol{a} &= \left(f^{-1}(y_k) - f^{-1}(\theta_k) \right)\left(\boldsymbol{i}_{k0} \,.\, \hat{\boldsymbol{n}} \right)\hat{\boldsymbol{n}} \\ &= \frac{f^{-1}(y_k) - f^{-1}(\theta_k)}{\|\nabla\phi_k\|} \frac{\nabla\phi_k}{\|\nabla\phi_k\|} \end{aligned} \tag{5.2}$$

and

$$\|\boldsymbol{c} - \boldsymbol{a}\| = \frac{f^{-1}(y_k) - f^{-1}(\theta_k)}{\|\nabla\phi_k\|} \tag{5.3}$$

The vector that is directed towards the origin and biased along the axes of the weights w_{ji} that have small weight values relative to some small positive constant

w_a is $\boldsymbol{w}' = -\sum_{j,i} \left(w_{ji} \middle/ w_a \right) \boldsymbol{i}_{ji} / \left(1 + w_{ji}^2 \middle/ w_a^2 \right)$. The projection of $\boldsymbol{w}'$ onto the tangent plane is given by

$$\begin{aligned} \boldsymbol{m} &= \boldsymbol{w}' - \left(\boldsymbol{w}' . \hat{\boldsymbol{n}} \right) \hat{\boldsymbol{n}} \\ &= \boldsymbol{w}' - \frac{1}{\| \nabla \phi_k \|} \sum_{l,m} \left(w'_{lm} \frac{\partial \phi_k}{\partial w_{lm}} \right) \frac{\nabla \phi_k}{\| \nabla \phi_k \|} \end{aligned} \qquad (5.4)$$

where

$$\boldsymbol{w}' = -\sum_{j,i} \frac{w_{ji} \big/ w_a}{1 + w_{ji}^2 \big/ w_a^2} \boldsymbol{i}_{j,i} \qquad (5.5)$$

Thus, if $\boldsymbol{d}$ is the point of intersection with the tangent plane of a line from $\boldsymbol{a}$ inclined at angle β to the perpendicular, then

$$\boldsymbol{d} - \boldsymbol{a} = \| \boldsymbol{c} - \boldsymbol{a} \| \tan \beta \frac{\boldsymbol{m}}{\| \boldsymbol{m} \|} + \left(\boldsymbol{c} - \boldsymbol{a} \right) \qquad (5.6)$$

Let $\delta = f^{-1}\left(y_k \right) - f^{-1}\left(\theta_k \right)$ be the error in the input to final unit. Hence using equations (5.2), (5.3) and (5.4) in (5.5) yields

$$\begin{aligned} \boldsymbol{d} - \boldsymbol{a} = \; & \frac{1}{\| \nabla \phi_k \|^2} \delta \nabla \phi_k + \frac{| \delta |}{\| \nabla \phi_k \|} \tan \beta \frac{1}{\| \boldsymbol{m} \|} \Big(\boldsymbol{w}' - \frac{1}{\| \nabla \phi_k \|} \\ & \times \sum_{lm} w'_{lm} \frac{\partial \phi_k}{\partial w_{lm}} \frac{\nabla \phi_k}{\| \nabla \phi_k \|} \Big) \end{aligned} \qquad (5.7)$$

Thus, to adjust a given weight w_{ji}

$$\Delta w_{ji} = \frac{1}{\left\| \nabla \phi_k \right\|^2} \delta \frac{\partial \phi_k}{\partial w_{ji}} + \frac{|\delta|}{\left\| \nabla \phi_k \right\|} \tan\beta \frac{1}{\| \boldsymbol{m} \|} \left(w'_{ji} - \frac{1}{\left\| \nabla \phi_k \right\|^2} \times \sum_{lm} w'_{lm} \frac{\partial \phi_k}{\partial w_{lm}} \frac{\partial \phi_k}{\partial w_{ji}} \right) \tag{5.8}$$

where

$$\| \boldsymbol{m} \|^2 = \sum_{j,i} \left(w'_{ji} - \frac{1}{\left\| \nabla \phi_k \right\|^2} \sum_{l,m} \left(w'_{lm} \frac{\partial \phi_k}{\partial w_{lm}} \right) \frac{\partial \phi_k}{\partial w_{ji}} \right)^2 \tag{5.9}$$

The term $\partial \phi_k / \partial w_{ji}$ is the partial derivative of the net input to the output unit. The treatment of this term follows from Lee [2,11]

$$\frac{\partial \phi_k}{\partial w_{ji}} = \frac{\partial \phi_k}{\partial \phi_j} \theta_i \tag{5.10}$$

and

$$\frac{\partial \phi_k}{\partial \phi_j} = \begin{cases} 1, & if \ j = k \\ f'_j(\phi_j) \displaystyle\sum_{m \in M_j} \frac{\partial \phi_k}{\partial \phi_m} w_{mj}, & if \ j \neq k \end{cases} \tag{5.11}$$

where M_j is the set of units to which u_j passes its output

Let us consider the different terms of (5.8) separately. The first term represents movement to the foot of the perpendicular at $\boldsymbol{c}$ from the current position $\boldsymbol{a}$ to the tangent plane to the constraint surface $\phi_k = f^{-1}(y_k)$. The second term determines the complexity of the network. The constant factor $tan\beta$ gives the angle between the movement vector and the perpendicular from the current position to the tangent plane. Its value is preferred to be small (e.g. typically ~ 0.05) so that a point nearby the foot of the perpendicular is chosen to move toward. A large value for $tan\beta$ would introduce inaccuracy into the weight update. The term in the brackets represents the projection of $\boldsymbol{w}^{'}$ onto the tangent plane. The directional vector $\boldsymbol{w}^{'}$ targets specific weights for removal according to the value of w_a. Those weights having an absolute value $\left| w_{ji} \right| >> w_a$ will receive a smaller push and thereby decay less rapidly to zero.

Let us next consider the computational efficiency of the new algorithm. A simple cost saving can be made by replacing the norm $\left\| \boldsymbol{m} \right\|$ in equation (5.9) with the norm of the directional vector $\boldsymbol{w}^{'}$. $\left\| \boldsymbol{w}^{'} \right\|$ is greater than or equal to $\left\| \boldsymbol{w}^{'} - \left(\boldsymbol{w}^{'}.\hat{\boldsymbol{n}} \right) \hat{\boldsymbol{n}} \right\|$. Its use will result in a reduction in the size of $\boldsymbol{m}$, but this term is scaled by $tan\beta$ anyway. Let n denote the total number of weights in the network. According to (5.9) $\left\| \boldsymbol{m} \right\|$ involves the expansion of the inner product $\boldsymbol{w}^{'}.\nabla\phi_k$, which requires $2n$ operations. The term $\boldsymbol{w}^{'}.\nabla\phi_k \big/ \left\| \nabla\phi_k \right\|^2$ is used to scale n partial derivatives, $\partial\phi_k \big/ \partial w_{ji}$, which requires a further $3n$ operations. Thus the total computational saving is $5n$ operations

The inclusion of a tendency to move towards the origin can be a disadvantage in the later stages of training. In cases where convergence does not occur quickly, the weight decay term $\boldsymbol{w}^{'}$ may penalize more of the weights than necessary giving average weight sizes small enough to trap the network in the region of weight space nearby the origin. A second improvement can be made by adding a small randomising term $\boldsymbol{x}$, $x_{ji} \in [-1,1]$, to the directional weight vector $\boldsymbol{w}'$. When projected onto the tangent plane it will take the movement laterally along tangent planes in random directions. Now, we only want this term to dominate $\boldsymbol{w}'$ when all the weights are small. We can achieve this goal by scaling $\boldsymbol{x}$ according to some monotonically decreasing function of the weights, say $\alpha_0 = w_b \, / \, |\overline{w}|$, where w_b is a small positive constant

5.2 Implementation of the new procedure

The following section is included to clarify the procedure for updating the weights of a network using the new iTPA algorithm

1. For each unit u_j,

$$x_j^{(t)} = \begin{cases} 1 \quad if \;\; j = k \\ \\ f^{-1}(\phi_j) \sum_m x_m w_{mj} \quad otherwise \end{cases}$$

2. For each weight w_{ji},

$$\frac{\partial \phi_k}{\partial w_{ji}} = x_j \theta_i$$

3. Calculate the squared norm of $\nabla \phi_k$

$$\| \nabla \phi_k \|^2 = \sum_{j,i} \left(\frac{\partial \phi_k}{\partial w_{ji}} \right)^2$$

4. Calculate the average of the absolute values of the weights

$$|\overline{w}| = \frac{1}{n} \sum_{j,i} |w_{ji}|$$

5. Generate a random vector $[x_{ji}]$ with $|x_{ji}| < 1$

6. For each weight w_{ji},

$$w'_{ji} = -\frac{\left(w_{ji} / w_a\right)}{\left(w_{ji}^2 / w_a^2\right) + 1} + \alpha_0 \, x_{ji}$$

7. Calculate the components of the vector $\boldsymbol{m}$

$$m_{ji} = w'_{ji} - \sum_{l,m} \left(w'_{lm} \frac{\partial \phi_k}{\partial w_{lm}} \right) \frac{1}{\left\| \nabla \phi_k \right\|^2} \frac{\partial \phi_k}{\partial w_{ji}}$$

8. Calculate the squared norm of $\boldsymbol{m}$

$$\left\| \boldsymbol{m} \right\|^2 = \sum_{j,i} m_{ji}^2$$

9. For each weight w_{ji}, add

$$\Delta w_{ji} = \frac{1}{\left\| \nabla \phi_k \right\|^2} \left(\delta \frac{\partial \phi_k}{\partial w_{ji}} + \left| \delta \right| \left\| \nabla \phi_k \right\| \tan \beta \frac{1}{\left\| \boldsymbol{m} \right\|} m_{ji} \right)$$

where $\delta = f^{-1}\left(y_k\right) - f^{-1}\left(\theta_k\right)$

5.3 Simulations and results

Comparative tests were performed on the new iTPA algorithm and original tangent plane algorithm under a variety of different initial conditions. The datasets used were regression and classification problems. Classification problems involve a decision making task where the output fits into well-defined categories. The classification task chosen was the two spiral problem as given in Fahlman [7]. Regression problems involve the approximation of a continuous valued function. The regression tasks chosen are the henon map time series [9], which were made artificially by computer simulation, and the housing price estimation problem obtained from the UCI data repository [61]

The two spiral problem was used to determine the effect of the weight sensitivity parameters on the convergence properties of the new iTPA algorithm. Since the tangent plane algorithm converges faster in oversized networks, a 2-30-30-1 architecture was used rather standard architectures quoted elsewhere e.g. the 2-5-5-5-1 architecture in [106]. For each test, 10 trials were performed with the classification error on the training set and the test set, mean number of epochs to converge, and number of successful trials recorded. Network training was terminated using Fahlman's 40-20-40 criteria [32] (e.g. for each pattern the output had to be less than 0.4 if the desired value was 0, or greater than 0.6 if the desired output was 1) for the purpose of comparison with [107]

The henon map and house price estimation problem were used to determine the effect of the directional movement vector on the distribution of weight sensitivities in networks trained by the new iTPA algorithm. The method used to estimate the importance of the weights was the autoprune [51]. Once again, an oversized network was used with the expectation that the weight decay term would automatically prune the network. For each test, 20 trials were performed with the mean square error on the training set and test set recorded together with the mean number of steps to converge. Network training was terminated when the error on the training set was reduced to below a preset value or the maximum number of permissible epochs was exceeded.

5.3.1 Network initialization

Both algorithms require parameters to be set manually. Preliminary tests showed that the best results were obtained with the parameters set as follows. First, the iTPA algorithm. For the two spiral problem, $tan\,\beta = 0.05$. The weight sensitivity parameters w_a and w_b were varied according to a grid search. The input weights were set to random values in the range [-2, 2]. For the henon map and house price estimation problems, $tan\,\beta$ = 0.05, w_a = 0.05, and w_b = 0.2. The input weights were set to random values in the range [-1,1]. Next, the second tangent plane algorithm. The angle parameter $tan\,\beta$ = 0.05. The input weights were set to random values in the range [-0.01, 0.01]

5.3.2 Simulation problems

The two spiral problem consists of two interlocking spirals, each made up of 97 data points. The network must learn to discriminate the two spirals. Traditionally this is known to be a very difficult problem for the back-propagation algorithm to solve. There are two inputs and one output. The inputs are the x and y co-ordinates, and the output notifies which spiral the point belongs to. For the points in the first spiral the output is set to +1, and for points on the other spiral the output is set to -1. The number of training samples is 194. A test set of 192 samples was generated by rotating the two spirals by a small angle.

The henon map problem is a chaotic time-series prediction problem. The time series is computed by

$$x^{(t+1)} = 1 - c\left(x^{(t)}\right)^2 + bx^{(t-1)} \qquad (5.12)$$

where $x^{(t)}$ is the value at taken time t, and the parameters $b = 0.3$, and $c = 1.4$. Initial values for the time series are $x^{(1)} = x^{(0)} = 0.63133545$. This point is called the fixed point of the time series. In neural network simulations, four successive values of the time series are used in predicting the next value. Thus, the number of inputs is four and the number of outputs in one. The number of training samples is 100, and testing samples is 100. Data values were taken from the range [31,230]

The house price estimation problem is a real-world problem that estimates the price of houses in the suburbs of Boston based on some attributes of houses (e.g. location, crime rate, level of air pollution, etc.). The number of inputs is 13, and the number of outputs is one. The number of training and testing examples is 253. The data sets used in the simulation were sampled randomly from the dataset provided by the UCI data repository with the outputs scaled down in the range [-1,1].

5.3.3 Error metrics used to determine convergence

The error metrics used in the simulations were CERR (Classification ERRor) for classification problems, and NMSE (Normalized Mean Square Error) for regression problems e.g.

$$CERR = \frac{1}{2m} \sum_i \left| sgn\left(y_{ki}\right) - sgn\left(\theta_{ki}\right) \right| \qquad (5.13)$$

and

$$NMSE = \frac{1}{m\sigma^2} \sum_i \left(y_{ki} - \theta_{ki} \right)^2 \qquad (5.14)$$

where m is the number of training patterns, y_{ki} is the target output of the ith input pattern, θ_{ki} is the ith network output, sgn is the sign function of a number (i.e. if the number if negative, then the sgn function returns -1, otherwise it returns +1), and σ^2 is the variance of the target output data.

5.3.4 Discussion of results

Two spiral problem. The first test is a difficult non-linearly separable problem where a set of co-ordinates (*x,y*) is classified as belonging to one of two interwoven spirals. For the iTPA algorithm, the average number of steps to converge varied from 370 to 810 epoch (mean = 566, std. dev. = 168) with the weight sensitivity parameters set at $w_a = 0.02$, and $w_b = 0.2$. One failed trial was excluded from the results. The classification error on the test set was 1.69 x 10^{-2} (e.g. % test set learned is 98.3) with all the points on the training set correctly classified. Using the original tangent plane algorithm, all trials converged in less than 710 epochs (mean = 529, std. dev. = 149). The classification error on the test set was 1.63 x 10^{-2} (e.g. % test set learned is 98.4) with all the training points correctly classified. The results compare favourably with those given by Linder et al [107] (Aprop: epochs = 67, % test set learned = 96.6; Rprop: epochs = 246, % test set learned = 65.6). The network architecture used with Aprop was 2-100-100-1, giving a total of 31,000 weights.

Table 5.1 demonstrates the effects of changing the weight sensitivity parameters w_a and w_b of the new iTPA algorithm. It was found that the classification error was not particularly sensitive to the exact value chosen for w_a. However, increasing the value of w_a had a deleterious effect on the convergence speed of the new algorithm and resulted in more failed trials. Clearly larger values for w_a have driven more of the weights down to zero resulting in problems with local minima. It was also found that increasing the value of w_b improved the speed of convergence speed. Including random movement along tangent planes into the weight update equation was sufficient to break out of local minima and permits the network to move in directions that are not available to the original algorithm

w_a	w_b	Cerr	Cerr*	Steps	Succ
0.02	0.5	0.0093	0.0017	566	10
0.05	0.5	0.0096	0.0019	1101	10
0.10	0.5	0.0096	0.0020	1716	10
0.20	0.5	0.0097	0.0018	1604	5
0.05	0.02	0.0097	0.0021	1319	9
0.05	0.05	0.0095	0.0023	1030	10
0.05	0.10	0.0095	0.0016	960	9
0.05	0.50	0.0096	0.0019	1101	10

Table 5.1. Classification error on the training set (Cerr) and test set (Cerr*), mean number of steps to converge, and number of successful trials (Succ) for different values of the weight sensitivity parameters w_a and w_b

Fig 5.3 and 5.4 show some typical test curves for both algorithms on the two spiral problem. Different sets of initial weights were used in each test. The test curves of the new iTPA algorithm show some variation (Fig 5.3). In many of the curves generated the convergence speed was found to be slow at the start of the training run with intermittent rises in the test error fairly typical (test 3). When the new algorithm was close to a solution, the convergence speed was usually rapid (test 1 and 3). The test curves of the original algorithm also show wide variation in the error (Fig 5.4). In the best curves the convergence speed was rapid with the network learning all the points on the test set (test 3). Generally speaking the original algorithm had fewer problems with local minima. There was very little evidence of overfitting observed in any of the test curves

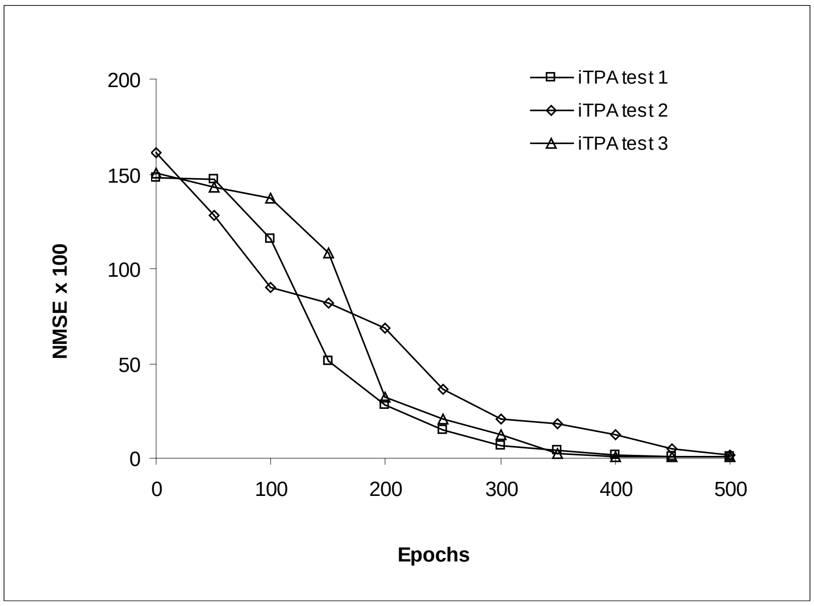


Fig. 5.3. Typical generalization behaviour of the new iTPA algorithm on the two spiral problem

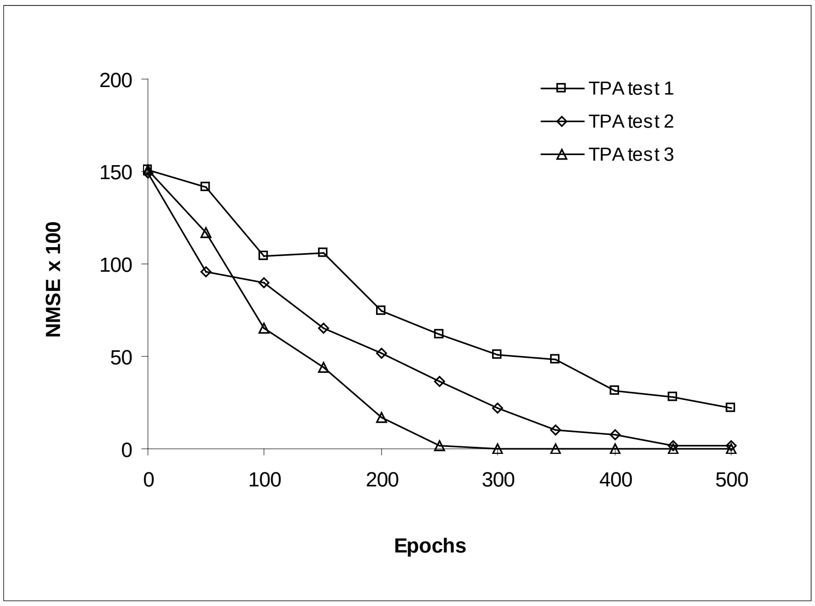


Fig. 5.4. Typical generalization behaviour of the second tangent plane algorithm on the two spiral problem

Henon map time series. The second test is a classical deterministic one-step-ahead prediction problem. Once again, an oversized 2-20-20-1 network architecture was chosen in order to determine the degree to which the new algorithm would remove redundant weight connections. For the iTPA algorithm, the number of steps to converge varied from 310 to 1150 (mean = 676, and std. dev. = 321) when the weight sensitivity parameters were set at w_a = 0.02, and w_b = 0.2. The mean square error on the test set was 0.003, with standard deviation 0.001. There was little evidence of overtraining. Increasing the value of the parameter w_a had a beneficial effect on generalization behaviour but resulted in much slower convergence. Using the original tangent plane algorithm, all trials converged in less than 1110 epochs (mean = 431, std. dev. = 259). The final error on the test set was 0.004 with standard deviation 0.006. Gross overfitting was observed in many trials, which accounts for the large variance in the final error.

Fig 5.5 and 5.6 show histograms of the importance coefficients of the weights for both algorithms on the henon map problem. The importance coefficients were recorded from the same trial at epochs 100, 300 and 500. The coefficient sizes were grouped in classes of width one and histograms plotted to show the distribution of the T_{ji} values at three different stages of training. The new iTPA algorithm gave average coefficient sizes of 1.48, 1.46, and 1.46. The original algorithm gave 1.70, 1.96, and 2.25. Notice the lengthening of the right tail of the histograms produced by the new algorithm (see Fig 5.5). This result suggests that a small proportion of the weights have taken on increasingly important roles in the network as the learning continues. Notice also the peak to the left of the main distribution (see Fig 5.5). This suggests that the weights of the network trained by the new algorithm are separating into two distinct groups.

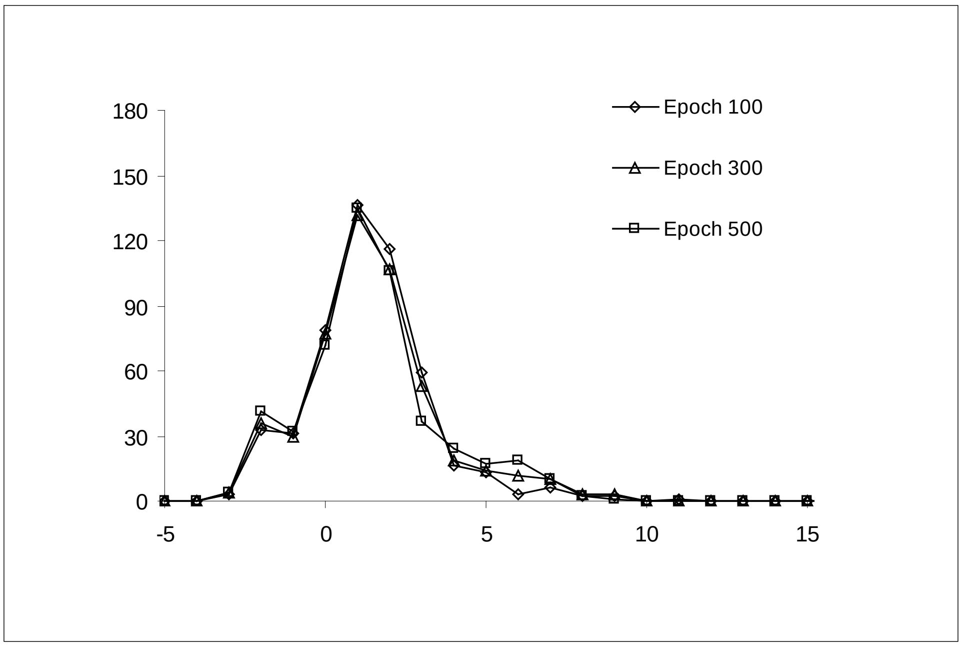


Fig. 5.5 Importance coefficient histograms for the new iTPA algorithm (henon map problem). Horizontal axis: coefficient size grouped in classes of width 1. Vertical axis: absolute frequency of weights with this coefficient size.

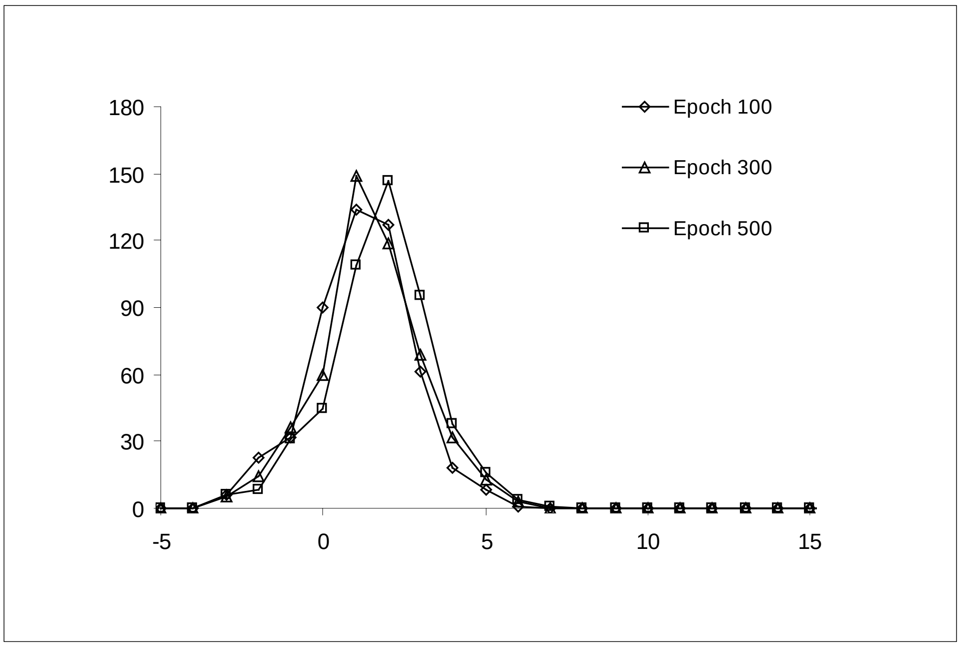


Fig. 5.6 Importance coefficient histograms for the second tangent plane algorithm (henon map problem). Horizontal axis: coefficient size grouped in classes of width 1. Vertical axis: absolute frequency of weights with this coefficient size

Housing price estimation problem. The final test is a real world problem that aims to estimate the price of housing in the suburbs of Boston. Once again, an oversized 13-20-20-1 network was used to determine the degree to which both algorithms could generalize in a network that contains redundant connections. For the iTPA algorithm, the number of steps to converge varied from 310 to 1190 (mean = 519, and std. dev. = 334) when the weight sensitivity parameters were set at w_a = 0.01, and w_b = 0.1. The mean square error on the test set was 0.13, with standard deviation 0.02. There was little evidence of overtraining. Using the original tangent plane algorithm, all trials converged in less than 1110 epochs (mean = 389, std. dev. = 305). The final error on the test set was 0.17 with standard deviation 0.05. Gross overfitting was observed in most trails. These results compare favourably with the results given by Lahnajärvi et al [17] (CasCor: epochs = 496, generalization = 0.22; Rprop: epochs = 603, generalization = 0.23).

Fig. 5.7 and 5.8 each show histograms of the importance coefficients for both algorithms on the housing price estimation problem. The importance coefficients were recorded from the same trial at epochs 100, 300 and 500. The coefficient sizes were grouped in classes of width one and histograms plotted to show the distribution of the T_{ji} values at three different stages of training. The new iTPA algorithm gave average coefficient sizes of 2.24, 2.68, and 2.79 respectively. The original algorithm gave 2.56, 3.33, and 3.81. The histograms produced by the new algorithm show the same kind of behaviour as before, namely the lengthening of the right tail and the small peak to the left of the distribution (see Fig 5.7). The histograms produced by the original algorithm show a distinctive drift to the right. This result shows that an increasing number of weights have evolved from the initial distribution about the origin (Fig 5.8)

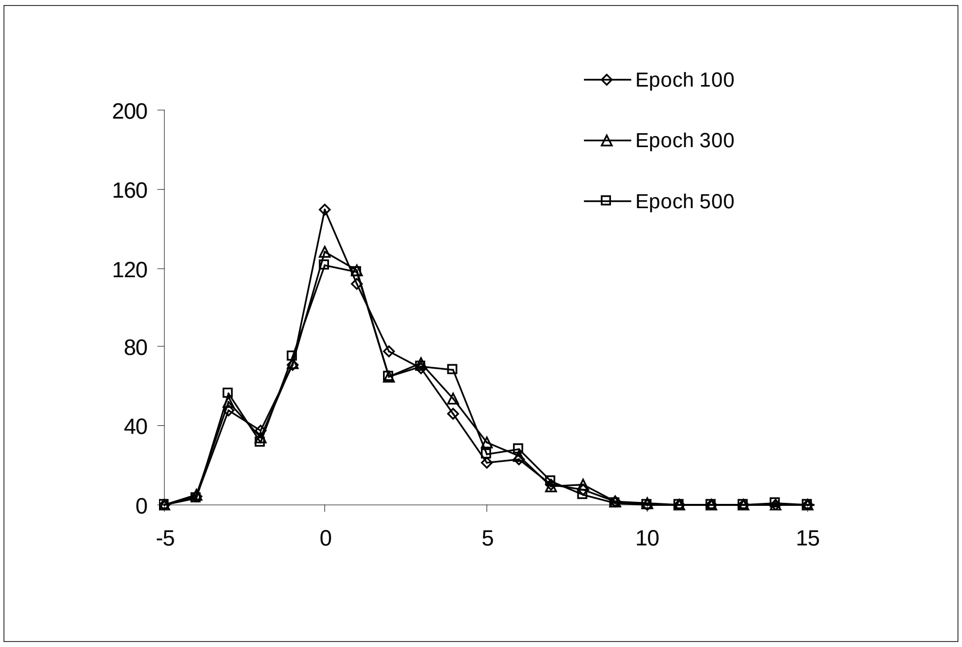


Fig. 5.7 Importance coefficient histograms for the new iTPA algorithm (housing price problem). Horizontal axis: coefficient size grouped in classes of width 1. Vertical axis: absolute frequency of weights with this coefficient size

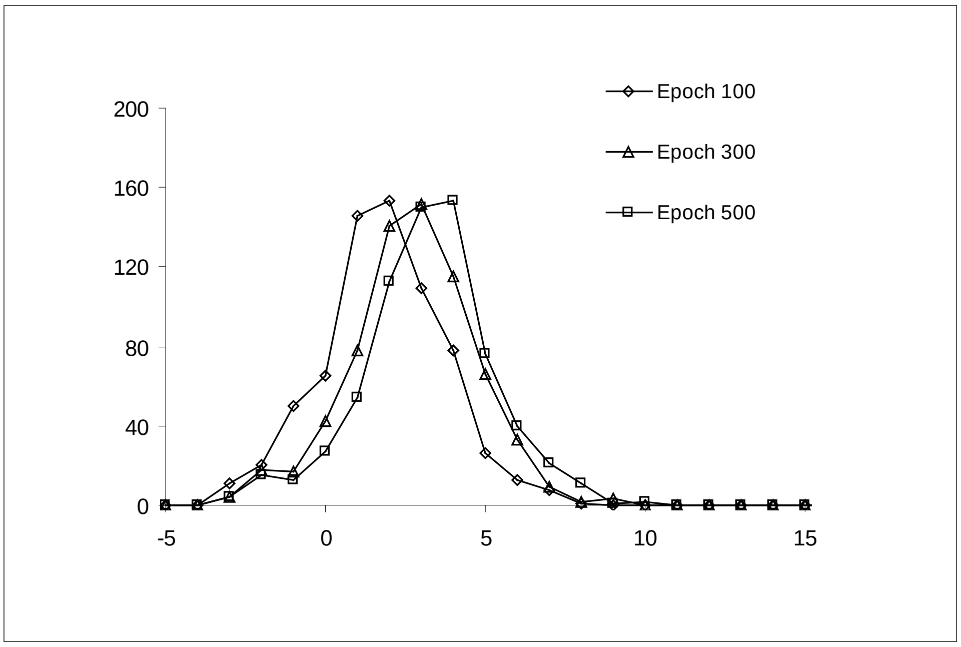


Fig. 5.8 Importance coefficient histograms for the second tangent plane algorithm (housing price problem). Horizontal axis: coefficient size grouped in classes of width 1. Vertical axis: absolute frequency of weights with this coefficient size

5.3.5 Comparison of the different algorithms

In order to determine whether the difference in the results is statistically significant, we perform some hypothesis tests. The test used was a standard *t*-test with the iTPA sample of 20 test errors compared with the corresponding sample from the original tangent plane algorithm for each dataset used in the study. A second test was carried out by comparing the results of weight growth and weight decay using the tangent plane algorithm to training a static network with early stopping. The backpropagation algorithm (learning rate $\eta = 0.01$, momentum term $\alpha = 0.7$) was used for this purpose. For the correct application of the *t*-test, it was necessary to take the logarithm of the test errors (since the test errors have log-normal distribution) and remove any outliers, following the same procedure in [51]. The resulting samples were tested for normality using the Kolmogorov-Smirnov test.

Problem	Training samples	Test samples	Inputs	(a)	(b)	(c)
Spiral	194	192	2	-	B 3.03	B 1.89
Henon	100	100	4	-	-	-
Housing	150	103	13	I 2.76	B 1.97	B 4.60

Table 5.2. Results of a t-test comparing the mean test errors of the different algorithms. The entries show differences that are statistically significant on a 10% level and dashes mean no significance found. Column (a): iTPA algorithm (“I”) vs. second tangent plan algorithm (“T”). Column (b): iTPA algorithm vs. backprop algorithm (“B”). Column. (c): second tangent plane algorithm vs. backprop algorithm

The results are tabulated in Table 5.2. Dashes mean differences that are not significant at the 10% level i.e. the probability that the differences are purely accidental. Other entries indicate the superior algorithm (e.g. iTPA algorithm - I, second tangent plane algorithm – T, backpropagation algorithm - B), and the value of the *t* statistic. Column (a) gives a comparison between the new iTPA algorithm and the second tangent plane algorithm. The results show that there is no significant difference in most of the benchmarks datasets used in the study. This would suggest that the new iTPA algorithm produces networks with a similar proportion of inactive weights as the original second tangent plane algorithm. The result in the housing estimation problem might be accounted for by a larger weight sensitivity parameter w_a, which gives the cut-off value below which weights are forced to zero in the iTPA algorithm. Column (b) and (c) gives a comparison with the backpropagation algorithm using early stopping. The results show that early stopping is superior to weight growth and weight elimination using the tangent plane algorithm. However this success was gained at the expense of prohibitively long runtimes (e.g. two spiral: epochs = 1600; henon: epochs = 1544; and housing: epochs = 449)

5.4 Summary

In this chapter, a new variant of the tangent plane algorithm referred to as iTPA is proposed for feed-forward neural networks. This new algorithm includes two modifications to the existing algorithm. Firstly, a directional movement vector is introduced into the training process to push the movement in weight space towards the origin. This movement vector simulates weight decay, which is known to have a beneficial effect on generalization in back-propagation learning. The directional movement vector is further modified to give a heavier weighting to weights with small weight values. Secondly, a random sideways movement along tangent planes is introduced into the training process. This improves the likelihood of finding a good solution with small weights values (which can help generalization).

Comparative tests were carried out using the new iTPA algorithm and the second tangent plane algorithm. The benchmark datasets used were two spiral, henon map, and housing price. The results indicate that the new iTPA algorithm retains the fast convergence speed of the original method. Including a small amount of random movement along tangent planes into the weight update often helps the network break out of local minima that can slow down the convergence. The results also show that the new algorithm gives improved generalization relative to the original algorithm in some problems (e.g. housing price), and comparable generalization in yet other problems (e.g. henon map). Finally, the new iTPA algorithm does not appear to give any tangible benefits in terms of improved generalization relative to the backpropagation algorithm with early stopping

In the next chapter, a new batch tangent plane algorithm is developed for training small parsimonious networks. This new algorithm uses the gradient information and target values to construct a linear system, and solves this system by finding a least squares solution. The newly developed algorithm is evaluated and compared with Rprop [28] on two benchmark datasets. Rprop is a very fast locally adaptive learning algorithm that is very robust relative to the selection of its internal parameters. The results show that the new batch tangent plane algorithm is very fast relative to Rprop. Some limitations of the new algorithm are also identified.

In chapter 7, two improvements are suggested to overcome the difficulties of the batch tangent plane algorithm. The newly developed algorithm is evaluated and compared with two popular network constructive techniques on three neural network benchmark problems

Chapter 6

A NEW BATCH TANGENT PLANE ALGORITHM FOR FEED-FORWARD NEURAL NETWORKS

In chapter three, a sequential learning algorithm called the tangent plane algorithm was evaluated for function approximation and classification tasks. This algorithm modifies the weights of a feed-forward neural network in the direction in which the error function decreases most rapidly. Unlike the gradient descent backpropagation algorithm, it does not require a learning rate parameter to be set to adjust the step size. Instead it uses the target data to define a surface in weight space. The weights are then updated by moving to the tangent plane to this surface, taken at a convenient point. The results show that the tangent plane algorithm is very fast relative to the standard backpropagation algorithm. However this improvement in speed was observed in large network structures. In small economical networks the convergence speed was found to be very slow and there were more failures to converge. Collecting all the gradient information together before the weights are updated can help to avoid the mutual interference of weight changes that slow down the convergence speed. Further, sequential methods may be slow in comparison to batch methods that use second-order information. In this chapter, a batch implementation of the tangent plane algorithm is developed for training small parsimonious feed-forward networks. This new algorithm uses the Gauss-Newton vector to guide the search toward the solution of a system of tangent plane equations. It is shown that the new batch tangent plane algorithm is fast compared with the best first order learning algorithm

6.1 Improving convergence in small economical networks

Previously, we have evaluated the tangent plane algorithm for three neural network benchmark tasks. This tangent plane algorithm treats each target value as a constraint which defines a surface in weight space. The weights are updated by moving to the tangent plane to this surface. Unlike the backpropagation algorithm, it automatically calculates the correct step to be taken. This is the principal strength of the tangent plane algorithm. Compared with the back-propagation algorithm, the tangent plane algorithm is very fast and avoids problems like local minima. However, this improvement was observed in large networks that have a high dimensional weight space. In small networks the convergence speed at best was comparable with more failures to converge.

Another difficulty with the tangent plane algorithm is locally instability due to the large steps taken in weight space. This was particularly noticeable with large datasets that contain one or more incorrect patterns. In order to address this difficulty a progressive stiffening factor was introduced whereby the step size was progressively decreased as the weights become trained. However, this strategy produced slower convergence relative to the backpropagation algorithm. Further it requires setting a parameter that can be difficult to tune

Collecting all the gradient information together before the weights are updated can help to avoid the mutual interference of weight changes that occur with large learning rates. Furthermore, sequential methods may be slow in comparison to batch methods that use second-order information. Thus it seems worthwhile investigating a batch implementation of the tangent plane algorithm for small economical networks. In the next section we describe the derivation of the new algorithm referred to as GN-TPA

6.1.1 Derivation of the new GN-TPA algorithm

A multi-layered feed-forward neural network of $\{u_j\}$ units is assumed where the connection from u_i to u_j is regulated by weight w_{ji}. ϕ_j and θ_j will be the input and output of u_j so that $\theta_j = f(\phi_j)$ for some monotonic function f, and $\phi_j = \sum_i w_{ji}\theta_i$. Let u_k be the single output neuron with θ_k trained to mimic a target value y_k. The tangent plane algorithm determines a plane of suitable points to move to within weight space. This suggests an attractive possibility for training in batch mode, which is to move to the intersection of each plane after the presentation of all the training patterns.

Our starting place is a general equation for the movement from the present position to a point on the tangent plane to the surface $\phi_k = f^{-1}(y_k)$. Let w'_{ji} be the current point in weight space, and let w''_{ji} be a point on the constraint surface such that w''_{ji} differs from w'_{ji} only in the value of the bias weight w_{k0}. It follows that $f^{-1}(y_k) - f^{-1}(\theta_k)$ is the distance along the axis corresponding to w_{k0} from w'_{ji} to w''_{ji}. Let $\nabla\phi_k$ be the gradient vector at w''_{ji}. The perpendicular distance Δw from w'_{ji} to the tangent plane at w''_{ji} is given by

$$\begin{aligned}
\Delta w &= \frac{\nabla\phi_k^T}{\left\|\nabla\phi_k\right\|}\cdot\left(f^{-1}(y_k) - f^{-1}(\theta_k)\right)\boldsymbol{i}_{k0} \\
&= \frac{f^{-1}(y_k) - f^{-1}(\theta_k)}{\left\|\nabla\phi_k\right\|}\frac{\partial}{\partial w_{k0}}\left(w_{k0} + \sum_{i\neq 0} w_{ki}\theta_i\right) \qquad (6.1) \\
&= \frac{f^{-1}(y_k) - f^{-1}(\theta_k)}{\left\|\nabla\phi_k\right\|}
\end{aligned}$$

If $\Delta \boldsymbol{w} \in R^n$ denotes a vector from the current point to a point on the tangent plane, then the projection of $\Delta \boldsymbol{w}$ onto $\nabla \phi_k$ is given by

$$[\nabla \phi_k]^T \Delta \boldsymbol{w} \quad = \quad f^{-1}(y_k) - f^{-1}(\theta_k) \tag{6.2}$$

Equation (6.2) defines a $(n-1)$ plane of suitable points to move to, within the weight space R^n. In order to develop a batch tangent plane algorithm we construct a system of equations using the entire training set and solve for a suitable point to move towards. For a set of input-output data $\{(\boldsymbol{x}^{(i)}, y_k^{(i)})\}_{i=1}^{m}$, equation (6.2) can be written as an $m \times n$ system as follows

$$\begin{aligned}
\left[\nabla \phi_k^{(1)}\right]^T \Delta \boldsymbol{w} &= f^{-1}\left(y_k^{(1)}\right) - f^{-1}\left(\theta_k^{(1)}\right) \\
\left[\nabla \phi_k^{(2)}\right]^T \Delta \boldsymbol{w} &= f^{-1}\left(y_k^{(2)}\right) - f^{-1}\left(\theta_k^{(2)}\right) \\
&\cdots\cdots\cdots\cdots\cdots\cdots \\
\left[\nabla \phi_k^{(m)}\right]^T \Delta \boldsymbol{w} &= f^{-1}\left(y_k^{(m)}\right) - f^{-1}\left(\theta_k^{(m)}\right)
\end{aligned} \tag{6.3}$$

Alternatively, equation (6.3) can be written more concisely as

$$\boldsymbol{J}_\phi(\boldsymbol{w})\Delta \boldsymbol{w} \quad = \quad f^{-1}(\boldsymbol{y}_k) - f^{-1}(\boldsymbol{\theta}_k)$$

where $\boldsymbol{J}_\phi = [\nabla \phi_k^{(1)}, \cdots, \nabla \phi_k^{(m)}]^{\mathrm{T}}$, $\nabla \phi_k^{(i)} \in R^n$, $m \geq n$, is an $R^{m \times n}$ Jacobian matrix, $\boldsymbol{y_k} = [y_k^{(1)}, \cdots, y_k^{(m)}]^T$ is a vector of target values, and $\boldsymbol{\theta_k} = [\theta_k^{(1)}, \cdots, \theta_k^{(m)}]^T$ is a vector of model outputs

If the number of training samples m equals the dimension of weight space n, then $\boldsymbol{J}_\phi$ is square and the system (6.3) can be solved by matrix inversion provided that $\boldsymbol{J}_\phi$ is non-singular (e.g. invertible). However, for most real neural network problems $m >> n$, which makes the system overdetermined. For such problems $\boldsymbol{J}_\phi$ will not be square, so there may not exist a set of Δw_{ji} such that (6.3) is satisfied exactly. Instead, one may need to find specific $\Delta \hat{w}_{ji}$ such that

$$\left\| \boldsymbol{J}_\phi(\boldsymbol{w})\Delta\hat{\boldsymbol{w}} - \boldsymbol{e}_k(\boldsymbol{w}) \right\|^2 = \min_{\Delta \boldsymbol{w}} \left\| \boldsymbol{J}_\phi(\boldsymbol{w})\Delta\boldsymbol{w} - \boldsymbol{e}_k(\boldsymbol{w}) \right\|^2 \qquad (6.4)$$

where $\boldsymbol{e}_k(\boldsymbol{w}) = [\, f^{-1}(\boldsymbol{y}_k) - f^{-1}(\boldsymbol{\theta}_k)] \in R^m$ represent a vector of errors. Expanding the norm squared of the residuals, we have

$$\begin{aligned}\left(\boldsymbol{J}_\phi\Delta\boldsymbol{w} - \boldsymbol{e}_k\right)^T\left(\boldsymbol{J}_\phi\Delta\boldsymbol{w} - \boldsymbol{e}_k\right) &= \left(\boldsymbol{J}_\phi\Delta\boldsymbol{w}\right)^T\left(\boldsymbol{J}_\phi\Delta\boldsymbol{w}\right) - \\ &\quad \boldsymbol{e}_k^T\boldsymbol{J}_\phi\Delta\boldsymbol{w} - \left(\boldsymbol{J}_\phi\Delta\boldsymbol{w}\right)^T\boldsymbol{e}_k + \boldsymbol{e}_k^T\boldsymbol{e}_k\end{aligned} \qquad (6.5)$$

The two middle terms on the right-hand side of (6.5) are equal. Differentiating and setting the result equal to zero, we arrive at the normal equations

$$\boldsymbol{J}_\phi^T\boldsymbol{J}_\phi\Delta\boldsymbol{w} = \boldsymbol{J}_\phi^T\boldsymbol{e}_k \qquad (6.6)$$

Equation (6.6) defines the new GN-TPA algorithm. The training procedure is iterative and proceeds as follows. Starting with some initial guess $\boldsymbol{w}^{initial}$ for the minimum, the weight update proceeds according to the iteration $\boldsymbol{w}^{new} = \boldsymbol{w}^{old} + \Delta\boldsymbol{w}$, where $\Delta\boldsymbol{w}$ is the weight increment at time step n. The iteration is terminated when the error vector $\boldsymbol{e}_k = f^{-1}(\boldsymbol{y}_k) - f^{-1}(\boldsymbol{\theta}_k)$ is sufficiently small or the matrix $\boldsymbol{J}_\phi^T \boldsymbol{J}_\phi$ becomes rank deficient

Let us next consider the convergence properties of the GN-TPA algorithm. If $\boldsymbol{e}_k = f^{-1}(\boldsymbol{y}_k) - f^{-1}(\boldsymbol{\theta}_k)$, then $\nabla \boldsymbol{\varphi}_k$ equals $-\nabla \boldsymbol{e}_k$ or $-\boldsymbol{J}_e$. Using this in the right-hand side of (6.6) yields $-\boldsymbol{J}_e^T \boldsymbol{e}_k$, which is the gradient of the error surface $\varepsilon_k = \sum_i e_{ki}^2$. As far as the generation of a descent direction goes, the square matrix $\boldsymbol{J}_\phi^T \boldsymbol{J}_\phi$ is always at least semi-positive definite, so $\Delta \boldsymbol{w}^T \boldsymbol{J}_\phi^T \boldsymbol{e}_k \geq 0$ or $\Delta \boldsymbol{w}^T \boldsymbol{J}_e^T \boldsymbol{e}_k \leq 0$ (e.g. $\Delta \boldsymbol{w}$ is a descent direction for ε_k). However this does not guarantee that $\varepsilon_k^{new} < \varepsilon_k^{old}$ as $\Delta \boldsymbol{w}$ might be too large locating $\boldsymbol{w}$ well beyond the minima. This problem is exacerbated by the choice of $f^{-1}(x)$. For example $f^{-1}(x)$ = $tanh^{-1}(x)$ blows up as x approaches ± 1

The only difficulty that can arise is $\boldsymbol{J}_\phi$ being rank deficient and hence $\boldsymbol{J}_\phi^T \boldsymbol{J}_\phi$ is singular. The customary practice for dealing with a rank deficient $\boldsymbol{J}_\phi$ is to add a diagonal matrix $\mu \boldsymbol{I}$ to the term $\boldsymbol{J}_\phi^T \boldsymbol{J}_\phi$, where $\mu \geq 0$ is a constant and $\boldsymbol{I} \in R^{n \times n}$ the unit matrix. When $\mu = 0$, $\Delta \boldsymbol{w}$ is a least squares step. As $\mu \to \infty$, the term $\mu \boldsymbol{I}$ increasingly dominates that of $\boldsymbol{J}_\phi^T \boldsymbol{J}_\phi$ so that $\Delta \boldsymbol{w} \to \mu^{-1} \boldsymbol{J}_\phi^T \boldsymbol{e}_k$. Finally, there remains the problem of finding the proper value of μ. One approach might be use a line search, but the main objection here is that is it prohibitively slow. Another approach might be to use a region of trust model. In Fletcher [41], if the error surface is approximately quadratic, then the model is operating optimally and μ is halved; otherwise μ incremented by a factor $\nu > 2$

6.1.2 Solving the tangent plane normal equations

The explicit formulation of the matrix $[\boldsymbol{J}_\phi^T\boldsymbol{J}_\phi]^{-1}\boldsymbol{J}_\phi^T$ from equation (6.6) is undesirable as it is prone to considerable round-off errors during its computation. Instead we can solve the equation $\boldsymbol{J}_\phi\,\Delta\boldsymbol{w} = \boldsymbol{e}_k$ using a singular value decomposition (SVD) which is numerically more stable [109]. This method involves factorizing the matrix $\boldsymbol{J}_\phi$ into an $m\times n$ orthogonal matrix $\boldsymbol{U}$, an $n\times n$ orthogonal matrix $\boldsymbol{V}$, and an $n\times n$ diagonal matrix $\boldsymbol{\Sigma}$ comprising the singular values of $\boldsymbol{J}_\phi^T\boldsymbol{J}_\phi$ along the leading diagonal with zero's elsewhere

$$\boldsymbol{J}_\phi = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T \qquad (6.7)$$

If the columns $\boldsymbol{v}_i$ of the right orthogonal matrix $\boldsymbol{V}$ are treated as eigenvectors of the symmetric matrix $\boldsymbol{J}_\phi^T\boldsymbol{J}_\phi$, then the elements along the leading diagonal of $\boldsymbol{\Sigma}$ are the positive square roots of the corresponding eigenvalues. The columns $\boldsymbol{u}_i$ of the left orthogonal matrix $\boldsymbol{U}$ correspond to the eigenvectors of $\boldsymbol{J}_\phi\boldsymbol{J}_\phi^T$. Solving (6.3) using a singular value decomposition, we arrive at the standard result

$$\Delta\boldsymbol{w} = \sum_{i=1}^{n}\frac{\boldsymbol{u}_i^T\boldsymbol{e}_k}{\sigma_i}\boldsymbol{v}_i \qquad (6.8)$$

where $\boldsymbol{e}_k$ is a vector of residuals (errors), which is this case are given by $f^{-1}(\boldsymbol{y}_k)-f^{-1}(\boldsymbol{\theta}_k)$, and σ_i are the singular values. In the summation above the terms that correspond with relatively small singular values can be omitted to improve the robustness in the calculation of $\Delta\boldsymbol{w}$. This situation arises whenever $\boldsymbol{J}_\phi$ is close to being rank deficient

6.1.3 Implementation of the procedure

The following section is included to clarify the procedure for updating the weights of a network trained using the batch tangent plane algorithm. For m distinct training samples ($\boldsymbol{x}^{(i)}, y_k^{(i)}$), where $\boldsymbol{x}^{(i)} \in R^N$ and $y_k^{(i)} \in R$

1. Compute the unit outputs $\boldsymbol{\theta}_j \in R^m$
2. Compute the components of the Jacobian $\boldsymbol{J}_\phi \in R^{m\times n}$
3. Compute the eigenvalues and orthonormal eigenvectors of the

   symmetric matrix $\boldsymbol{J}_\phi^T \boldsymbol{J}_\phi \in R^{n\times n}$
4. Construct $\boldsymbol{\Sigma} \in R^{n\times n}$ as a square matrix whose diagonal elements σ_{ii}

   are the singular values of $\boldsymbol{J}_\phi$
5. Set $\boldsymbol{V} = [\ \boldsymbol{v}_i] \in R^{n\times n}$ where the columns $\boldsymbol{v}_i \in R^n$ are the eigenvectors

   identified in step 3
6. Calculate $\boldsymbol{U} = \boldsymbol{J}_\phi \boldsymbol{V}\ \boldsymbol{\Sigma}^{-1}$
7. Calculate $\Delta \boldsymbol{w} = \sum_{i=1}^{n} \left(\boldsymbol{u}_i^T \boldsymbol{e}_k / \sigma_i \right) \boldsymbol{v}_i$
8. Test on the training set
9. If model adequate, then Stop

   else Goto step 1

6.2 Simulations and results

Comparative tests were performed using the new GN-TPA algorithm and a fast first order learning algorithm, Rprop (resilient back-propagation) [28], under a variety of initial conditions and different network sizes. The training sets used were regression and classification problems. Classification problems involve a decision making task where the output fits into well-defined categories. The classification task chosen was the breast cancer diagnosis problem obtained from the Proben1 collection [34]. Regression problems involve the approximation of a continuous valued function. The regression task was the additive function problem which was created artificially by computer simulation [22].

For each test carried out a single hidden layer feed-forward neural network was trained 20 times using different starting values for the weights. The number of steps to converge, and the normalised square error on the training set and test set were averaged over the 20 trials. An upper limit was placed on the maximum permissible number of steps. Network training was terminated using the method of early stopping as this method is known to help avoid overfitting [52].

6.2.1 Network initialization

The GN-TPA algorithm does not require any parameters to be set manually. In preliminary tests it was found that the performance of the algorithm was sensitive to the initialization of the weights. For the breast cancer problem, the input weights were set to random values in the range [-2, 2]. For the additive problem, the input weights were set to random values in the range [-1, 1]

The parameters used with the Rprop algorithm are the step increment factor η^{+}, the step decrement factor η^{-}, the initial step size $\Delta^{(0)}$, the maximum step size Δ_{max}, and the minimum step size Δ_{min}. The step increment and decrement factors were chosen to be the same as in the original paper, i.e. $\eta^{+} = 1.2$, $\eta^{-} = 0.5$. The initial step size is not critical, and was set to $\Delta^{(0)} \in \left[0.05, 0.2\right]$. The maximum step size was also chosen to be the same as in the original paper, i.e. $\Delta_{max} = 50$. Finally, the minimum step size was set to $\Delta_{min} = 1x10^{-8}$ to avoid overflow/underflow problems with floating point variables. The input weights were set to random values in the range [-0.1, 0.1]

6.2.2 The error metric used to determine convergence

The error metrics used in the simulations were CERR (Classification ERRor) for classification problems, and NMSE (Normalized Mean Square Error) for regression problems e.g.

$$CERR = \frac{1}{2m} \sum_i \left| sgn\left(y_{ki}\right) - sgn\left(\theta_{ki}\right) \right| \tag{6.9}$$

and

$$NMSE = \frac{1}{m\sigma^2} \sum_i \left(y_{ki} - \theta_{ki} \right)^2 \tag{6.10}$$

where m is the number of training patterns, y_{ki} is the target output of the ith input pattern, θ_{ki} is the ith network output, sgn is the sign function of a number (i.e. if the number if negative, then the sgn function returns -1, otherwise it returns +1), and σ^2 is the variance of the target output data.

6.2.3 Simulations problems

The cancer problem contains some diagnosis results for breast cancer. Based on cell descriptions gathered by microscopic examination, a tumour is classified as benign or malignant. The dataset was created based upon the breast cancer Wisconsin problem dataset from the UCI machine learning repository [61]. The output represents the classification result for the purpose of breast cancer diagnosis. The decision is based on nine input attributes which include cell thickness, the uniformity of cell size, and cell shape. The number of training samples is 200 and the number of testing samples is 167.

The additive problem is a non-linear function approximation problem that was obtained from [22]. The function is computed by

$$z = -0.5x_1^2 - x_1x_2 + 0.5\,x_2^2 + \frac{x_1}{1 + x_2^2} + 0.5\,e^{x_1 - x_2} + 0.05\,u \qquad (6.11)$$

A small signal u is added to the output with values uniformly distributed in the range [-1, 1]. Four hundred data points are generated. The first 200 data points are used as training data whilst the remaining are used as test data. The input values are uniformly distributed in the range [-1, 1]. All functional values or outputs are scaled down in the range [-1, 1]

6.2.4 Discussion of results

The first test utilized the additive function data. The results are tabulated in Table 6.1a. It was found that the batch tangent plane algorithm gave significantly faster convergence relative to Rprop, and that the convergence speed improved with network size. Both methods reached the minimum training error. It was also found that the new GN-TPA algorithm gave comparable generalization relative to Rprop, except in the smallest network where it was slightly worse. When failed trials were removed from the results, the performance of the batch tangent plane algorithm in the smallest network was significantly better (test error = 2 x 10^{-4}, epochs = 18). The improvement in the Rprop algorithm in comparison was relatively small (test error = 1 x 10^{-4}, epochs = 1873)

Fig 6.1 and 6.2 show the training curves for both algorithms. The curves for the GN-TPA algorithm drop quickly at first. Thereafter two of the curves exhibit sharp bumps. Convergence occurs rapidly within 15 epochs. The sharp bumps suggest oscillatory behaviour caused by the step size overshooting the linear minimum. The curves for Rprop show a sharp initial bump after one epoch. Thereafter the curves are relatively flat illustrating linear convergence. The first sharp bump is probably due to Rprop increasing the step size far too quickly in the initial weight space. The subsequent bump illustrates the tendency of Rprop to overcompensate the step size when moving in a descent direction

(a)

	Batch tangent plane algorithm (GN-TPA) Avg. validation set error using early stopping (NMSE x 10^2)			Resilient back-prop (Rprop) Avg. validation set error using early stopping (NMSE x 10^2)		
Size	Err	Err*	Steps	Err	Err*	Steps
10	0.03	0.04	811	0.02	0.02	1975
15	0.01	0.02	25	0.02	0.02	1865
20	0.01	0.01	30	0.02	0.02	1724

(b)

	Batch tangent plane algorithm (GN-TPA) Avg. validation set error using early stopping (CERR x 10^2)			Resilient back-prop (Rprop) Avg. validation set error using early stopping (CERR x 10^2)		
Size	Err	Err*	Steps	Err	Err*	Steps
10	3.71	2.81	10	3.91	2.12	20
15	3.54	2.01	12	3.91	2.69	17
20	3.66	2.66	10	3.83	2.78	23

Table 6.1. Training set error (Err), test set error (Err*) and steps to converge for different size networks with training terminated using early stopping: (a) additive function approximation problem, (b) the breast cancer problem

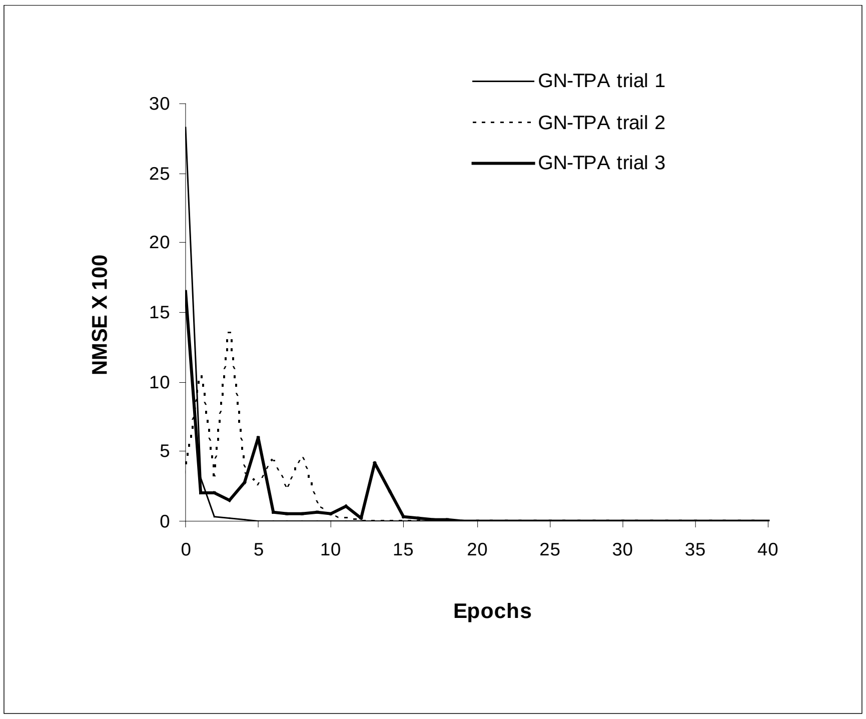


Fig.6.1. Typical training curves generated by the new GN-TPA algorithm on the additive function approximation problem

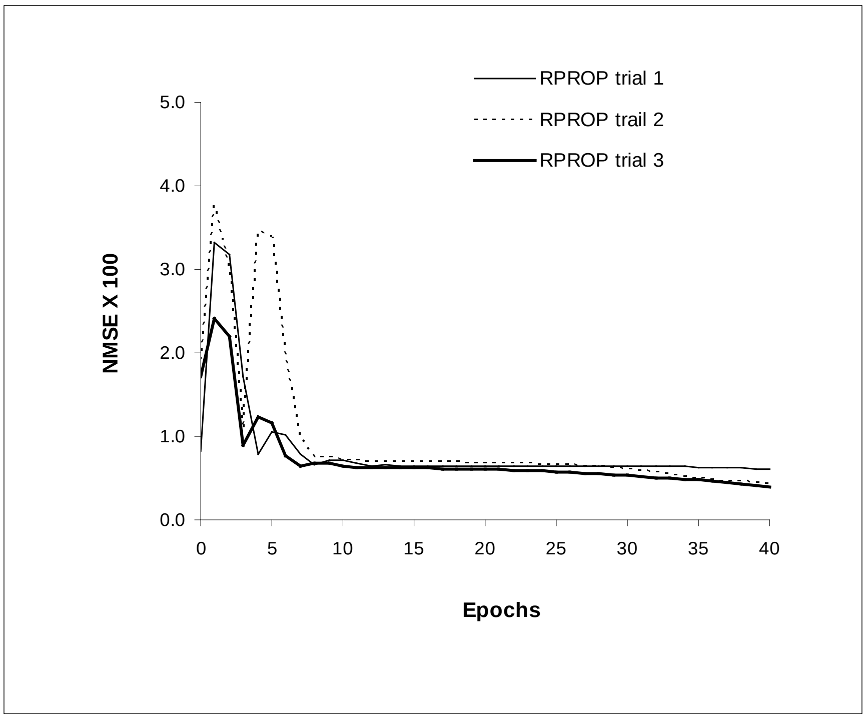


Fig.6.2. Typical training curves produced by the Rprop algorithm on the additive function approximation problem

The second test utilized the breast cancer dataset. The results are tabulated in Table 6.1b. It was found that the GN-TPA algorithm gave faster convergence relative to Rprop across a range of network sizes. However, this improvement was paid for by much longer training times. It takes approximately 5 minutes for the batch tangent plane algorithm to train a network with 20 units 5 times using a Pentium IV (2.67 GHz). Rprop takes approximately 10 seconds to finish the training process. Clearly the computational burden of performing an SVD operation makes the batch tangent plane algorithm suitable only for small to medium sized networks. It was also found that the batch tangent plane algorithm gave comparable generalization relative to the Rprop algorithm, and that generalization was independent of network size.

Figures 6.1 and 6.2 show some typical training curves for both algorithms. The training curves for the new GN-TPA algorithm are very different. The first curve drops fairly sharply within the first five epochs. Thereafter it is relatively flat illustrating linear convergence. The second curve exhibits sharp bumps during the first 8 epochs, and then the behaviour is similar to the first curve. The sharp bumps suggest oscillatory behaviour due to the large steps taken overshooting a solution point. The last curve drops sharply and then stalls at 10 epochs. This is probably an effect of an (almost) singular Jacobian matrix. The curves for the Rprop algorithm drop rapidly within the first 5 or so epochs. However, these curves differ by converging to a deep minimum. One curve performs steep gradient ascent before converging on a solution. Note that these curves follow the same general trend. This result is not surprising given the starting values of the weights are within a much smaller region in the weight space

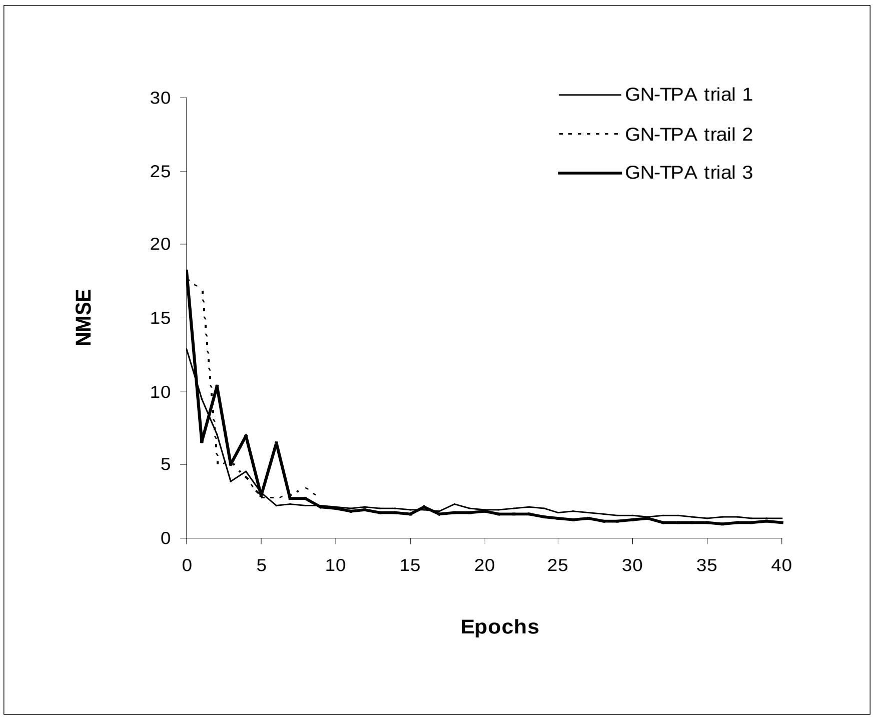


Fig.6.3. Typical training curves produced by the new GN-TPA algorithm on the breast cancer problem

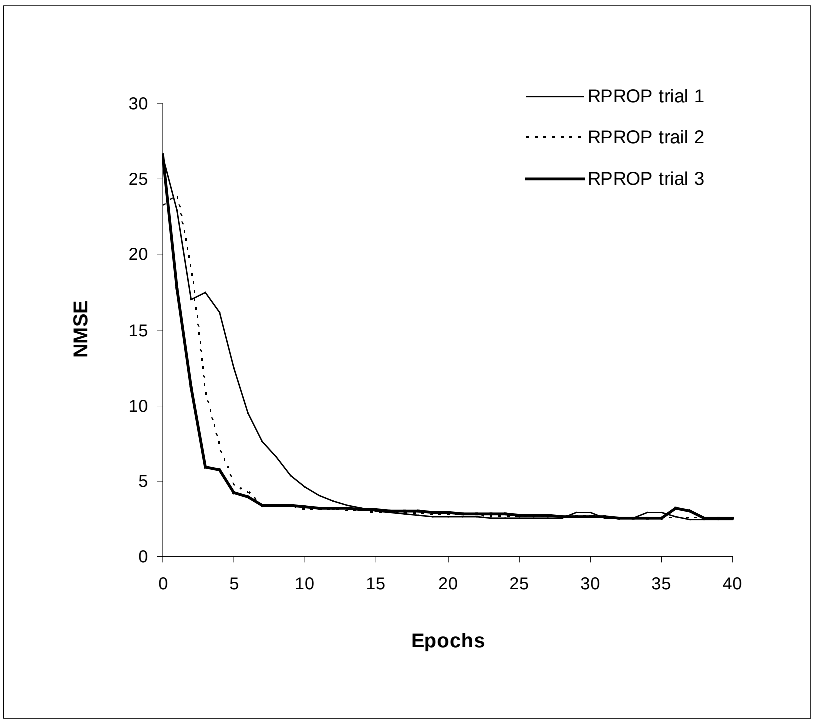


Fig.6.4. Typical training curves produced by the Rprop algorithm on the breast cancer problem

6.2.5 Problems with the new GN-TPA algorithm

There are a number of difficulties with the GN-TPA algorithm. These difficulties can be summarised as follows

- First, a good initial guess is required if convergence is to occur. The GN-TPA algorithm is a local minimizer of the error (or cost) function ε_k. A necessary condition for local convergence is that the initial guess must be sufficiently close to a solution $\boldsymbol{w}^*$, that is $\varepsilon_k(\boldsymbol{w}^*) \leq \varepsilon_k(\boldsymbol{w})$ for $\left\| \boldsymbol{w} - \boldsymbol{w}^* \right\| < \delta$ where δ is a small positive constant. Local convergence is likely to cause problems with stability and convergence, and where the error function has more than one minimum.
- Secondly, the computational overhead of performing a generalized inverse operation causes the computer to crash. The computational complexity of a generalized inverse operation of a $m \times n$ matrix will vary depending on the method used. Under an SVD operation this is equal to $4m^2n + 8mn^2 + 9n^3$, where m is the number of patterns to be learned, and n the number of weights. For a given training set with m patterns to be learned, the computational cost increases as n^3, which can be quite significant in large networks with many weights.
- Finally, the computation of the generalized inverse operation is prone to numerical errors. This is due to the Jacobian matrix $\boldsymbol{J}_\phi$ becoming rank deficient. For example, at the minimum $\boldsymbol{w}^*$ the vector $\boldsymbol{J}_\phi^T \boldsymbol{e}_k$, being proportional to the gradient vector, must be zero. Therefore, if $\boldsymbol{e}_k(\boldsymbol{w}^*) \neq 0$, then it follows that $\boldsymbol{J}_\phi(\boldsymbol{w}^*)$ is rank deficient.

Chapter 7

BATCH LEARNING BY APPROACHING TANGENT PLANES IN THE EXTREME LEARNING MACHINE

In chapter six, a new batch tangent plane algorithm was developed for small parsimonious networks. This new algorithm utilizes the Gauss-Newton vector to guide the search toward the error minimum. Comparative tests were performed using the new batch tangent plane algorithm and a fast locally adaptive learning algorithm, Rprop (resilient back-propagation) [28] under a variety of different initial conditions and network sizes. The benchmark datasets used were breast cancer obtained from the UCI machine learning repository [61], and the additive function problem obtained from [22]. The results show that the new batch tangent plane algorithm gives improved convergence speed and comparable generalization performance relative to the Rprop algorithm. However, the batch tangent plane algorithm suffers from a number of problems. Firstly, the algorithm is locally convergent, meaning that a good initial starting condition is required for convergence to a local good minimum to occur. Secondly, the computational overhead of performing a generalized inverse operation after each learning step is very large making runtimes very long. Finally, the computation of the generalized inverse matrix is prone to numerical errors. This is due to the Jacobian matrix becoming rank deficient. In this chapter, the newly developed batch tangent plane algorithm is applied to a novel network structure called an extreme learning machine to overcome the difficulties with this algorithm, namely local convergence and the high computational overhead. Studies [76,110,111] have shown that the extreme learning machine (ELM) is very fast and efficient

7.1 Improving the convergence behaviour and efficiency of the new batch tangent plane algorithm

Previously, we have described a new learning algorithm based on tangent plane algorithm targeted at small parsimonious networks. This new algorithm relaxes the requirement of the original method for movement to the nearest point on the tangent plane to a constraint surface, but instead moves to a general point on the tangent plane. Further a system of equations is constructed for the entire training set and solved using a singular value decomposition. The resulting movement is a Gauss-Newton (GN) step toward the minimum training error. Unfortunately this strategy can produce some big weight updates leading to oscillatory behaviour for certain starting conditions of the weights. In Huang et al [110,111], a new learning algorithm was described for single hidden layer feed-forward neural networks (SLFN) called the Extreme Learning Machine (ELM). In this method, the input weights of the hidden units are initialized to random values and fixed. The learning process is then treated as a linear problem with the weights of the output unit optimized through a generalized inverse operation. Studies [111,112] have shown that ELM is very fast and efficient. Thus a strategy to improve the convergence of the batch tangent plane algorithm would be to use a SLFN with input weights set to random values

7.1.1 A brief introduction to the ELM algorithm

Traditionally gradient descent based learning methods and variations such as the back-propagation algorithm have been used to train the weights of feed-forward neural networks. It is generally known that these methods are very slow due to the improper choice of step size and problems with convergence to local minima. Also, many epochs or presentations of the entire dataset are required to learn the training data making gradient descent based methods very slow.

Recently it has been shown that single hidden layered neural networks (with N hidden units) with input weights chosen arbitrarily can learn N distinct observations [113,114]. Unlike more traditional methods that tune all the parameters, the input weights of the first hidden layer do not have to be adjusted at all. Study results [110] show that this method not only makes learning extremely fast but also produces good generalization. Recently it has been further proved that single hidden layered neural networks with arbitrarily assigned input weights can approximate any continuous function on any compact dataset. After the input weights of the hidden layer units are randomly chosen, a single hidden layered neural network can be treated as a linear system and the output weights determined analytically through a generalized inverse operation of the hidden layer outputs. From the foregoing discussion, we show that a SLFN with m units can learn m distinct samples

For m arbitrary distinct samples $(\boldsymbol{x}_i, y_i)$, where $\boldsymbol{x}_i = [x_{i1}, \cdots, x_{iN}]^T \in R^N$, and $y_i \in R$, the output of a SLFN is $\theta_k = \sum_i w_{ki}\theta_i(\boldsymbol{x}, \boldsymbol{w}_i)$, where w_{ki} is ith weight of output unit u_k, θ_i is the output of hidden unit u_i, $\boldsymbol{w}_i = [w_{i1}, \cdots, w_{iN}]^T \in R^N$ are the input weights of u_i. If hidden unit u_i has activation f, then $\theta_i = f(\boldsymbol{w}_i^T \boldsymbol{x})$. For a set of inputs $\boldsymbol{x}_i$, $i = 1, 2 \cdots m$, the output of u_i can be treated as an m dimensional vector $\boldsymbol{\theta}_i \in R^m$. If the input weights $\{w_{ij} : i \neq k\}$ are set to arbitrary values and fixed, then the outputs $\boldsymbol{\theta}_i \in R^m$, $i = 1, 2 \cdots n$ form a random set of vectors that span a subspace of R^m. Provided that $n < m$, then with probability one these vectors are linearly independent [111]. As $n \to m$, the outputs $\boldsymbol{\theta}_i \in R^m$, $i = 1, 2, \cdots, n$, will be extended to form a spanning set of R^m. Thus, for a specific set of weights $\{\hat{w}_{ki}\}$, any point $\boldsymbol{y}_k \in R^m$ can be reached by a single hidden layered neural network with input weights fixed at arbitrary values.

Based on the foregoing analysis, we evaluate the newly developed GN-TPA algorithm for the Extreme Learning Machine.

7.1.2 Derivation of the new GN-TPA algorithm

For m distinct samples ($\boldsymbol{x}^{(i)}, y_k^{(i)}$), where $\boldsymbol{x}^{(i)} = [x_1^{(i)}, \cdots, x_N^{(i)}]^T \in R^N$ and $y_k^{(i)} \in R$, a single layer feed-forward neural network with n hidden units and activation function f is given by

$$f\left(w_{k0} + \sum_{i=1}^{n} w_{ki}\, \theta_i^{(j)} \right) = \theta_k^{(j)}, \quad j = 1, 2, \cdots, m \tag{7.1}$$

$$\theta_i^{(j)} = f\left(w_{i0} + \sum_{l=1}^{N} w_{il}\, x_l^{(j)} \right) \tag{7.2}$$

The method assumes a single layer feed-forward neural network with input weights $\boldsymbol{w}_i$, $i = 1, 2, \cdots, n$ chosen arbitrarily. For such a system, the batch tangent plane algorithm is defined by the following matrix equation

$$[\,\boldsymbol{\theta}^{(t)}]^T \Delta \boldsymbol{w}_k^{(t)} = f^{-1}\left(\boldsymbol{y}_k\right) - f^{-1}\left(\boldsymbol{\theta}_k^{(t)}\right) \tag{7.3}$$

In equation (7.3), $\boldsymbol{\theta}^{(t)} = [\,\boldsymbol{\theta}_1^{(t)}, \cdots, \boldsymbol{\theta}_n^{(t)}]^T$, $\boldsymbol{\theta}_i^{(t)} \in R^m$, is a matrix of outputs from hidden layer units u_i at time step t, $\Delta \boldsymbol{w}_k^{(t)} \in R^n$ is a vector of weight changes to the final unit, $\boldsymbol{y}_k \in R^m$ is a vector of desired outputs, and $\boldsymbol{\theta}_k^{(t)} \in R^m$ a vector of outputs from the neural network.

Expanding the left-hand side of (7.3),

$$[\,\boldsymbol{\theta}^{(t)}]^T[\,\boldsymbol{w}_k^{(t+1)}-\boldsymbol{w}_k^{(t)}] \quad = \quad f^{-1}(\,\boldsymbol{y}_k) - f^{-1}\left(\boldsymbol{\theta}_k^{(t)}\right) \qquad (7.4)$$

But $f^{-1}\left(\boldsymbol{\theta}_k^{(t)}\right) = [\,\boldsymbol{\theta}^{(t)}]^T\boldsymbol{w}_k^{(t)}$, so we arrive at

$$\begin{aligned}[\,\boldsymbol{\theta}^{(t)}]^T\boldsymbol{w}_k^{(t+1)} \quad &= \quad [\,\boldsymbol{\theta}^{(t)}]^T\,\boldsymbol{w}_k^{(t)} + f^{-1}(\boldsymbol{y}_k) - [\,\boldsymbol{\theta}^{(t)}]^T\boldsymbol{w}_k^{(t)} \\ &= \quad f^{-1}(\boldsymbol{y}_k) \qquad (7.5)\end{aligned}$$

For convenience, we drop the notation for the time step t. If $m >> n$, then $\boldsymbol{\theta}$ will not be square, so there may not exist a set of w_{ki} such that (7.5) is exactly satisfied. Instead, one may need to find specific $\hat{w}_{ki}$ such that

$$\left\| \boldsymbol{\theta}^T\hat{\boldsymbol{w}}_k - f^{-1}(\,\boldsymbol{y}_k)\right\| \quad = \quad \min_{w\in R^n} \left\|\boldsymbol{\theta}^T\boldsymbol{w}_k - f^{-1}(\,\boldsymbol{y}_k)\right\| \qquad (7.6)$$

Let $\boldsymbol{J}$ represent a $m \times n$ matrix with columns $\boldsymbol{\theta}_i \in R^m$, so $\boldsymbol{J} = [\,\boldsymbol{\theta}_1, \cdots, \boldsymbol{\theta}_n\,]$. For the general case where $m >> n$, the solution of the above linear least squares problem is given by

$$\hat{\boldsymbol{w}}_k \quad = \quad \left[\boldsymbol{J}^T\boldsymbol{J}\right]^{-1}\boldsymbol{J}^T f^{-1}(\boldsymbol{y}_k) \qquad (7.7)$$

Equation (7.7) defines the GN-TPA algorithm applied to the ELM paradigm. It solves the optimization problem in a single step.

Consider the different terms in (7.7). The term $[\boldsymbol{J}^T\boldsymbol{J}\,]^{-1}\boldsymbol{J}^T$ is recognised as the Moore-Penrose generalized inverse of $\boldsymbol{J}$, that is $\boldsymbol{J}^+ = [\boldsymbol{J}^T\boldsymbol{J}\,]^{-1}\boldsymbol{J}^T$. A theorem [108] tells us that if there exists a matrix $\boldsymbol{B}$ such that $\boldsymbol{B\,Y}$ is a minimum norm least squares solution of the linear system $\boldsymbol{X\,A} = \boldsymbol{Y}$, then it is necessary and sufficient that $\boldsymbol{B} = \boldsymbol{A}^{+}$, the Moore-Penrose generalized inverse of $\boldsymbol{A}$. Thus the special solution $\hat{\boldsymbol{w}}_k$ is the solution with the minimum norm least squares, meaning that the minimum training error can be reached by this solution.

Next, the term $f^{-1}(\boldsymbol{y}_k)$ is the target vector mapped backward to give the desired input to the final unit. $f^{-1}(\boldsymbol{y}_k)$ will tend to favour solutions with a large norm of weights, which is known to be harmful to generalization. Introducing a regularisation term $\lambda \boldsymbol{I}$ $(\lambda \geq 0)$ into the pseudo-inverse $[\boldsymbol{J}^T\boldsymbol{J}]^{-1}\boldsymbol{J}^T$ will help to discourage overtraining in the network [135]

7.1.3 Solving the tangent plane normal equations

Using a singular decomposition to compute the pseudo-inverse can be computationally expensive, especially when the network is scaled up. An alternative method, solving the tangent plane normal equations $\boldsymbol{J}^T\boldsymbol{J}\ \boldsymbol{w}_k = \boldsymbol{J}^T f^{-1}\left(\boldsymbol{y}_k\right)$ using a thin QR factorization, is computationally simple [109]. The method is based on the orthogonal decomposition of $\boldsymbol{J}$ into

$$\boldsymbol{J} = \begin{bmatrix} \boldsymbol{R}_1 & \boldsymbol{R}_2 & \cdots & \boldsymbol{R}_n \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ & a_{22} & \cdots & a_{2n} \\ & & \ddots & \vdots \\ & & & a_{nn} \end{bmatrix} \tag{7.8}$$

where $\boldsymbol{R} = [\boldsymbol{R}_1, \cdots, \boldsymbol{R}_n] \in R^{m\times n}$ is orthogonal and $\boldsymbol{A} = [a_{ji}] \in R^{n\times n}$ is upper right triangular. If $\boldsymbol{J}$ has full column rank, then the columns of $\boldsymbol{R}$ will form an orthonormal basis for $span\,(\boldsymbol{J}_1 \ldots \boldsymbol{J}_\mathrm{n})$.

From equation (7.6), we need to find a specific $\boldsymbol{w}_k$ such that the norm $\left\| \boldsymbol{J}\,\boldsymbol{w}_k - f^{-1}(\,\boldsymbol{y}_k) \right\|$ is minimised. To proceed, we note that an arbitrary vector $\boldsymbol{y}$ is invariant under multiplication by an orthogonal matrix $\boldsymbol{Z}$ e.g.

$$\left\| \boldsymbol{Z}\,\boldsymbol{y} \right\| \;=\; \left(\boldsymbol{y}^T \boldsymbol{Z}^T \boldsymbol{Z}\,\boldsymbol{y} \right)^{\frac{1}{2}} \;=\; \left(\boldsymbol{y}^T \boldsymbol{y} \right)^{\frac{1}{2}} \;=\; \left\| \boldsymbol{y} \right\| \qquad (7.9)$$

Therefore, from equation (7.8)

$$\begin{aligned} \left\| \boldsymbol{J}\,\boldsymbol{w}_k - f^{-1}(\boldsymbol{y}_k) \right\| &= \left\| \boldsymbol{R}^T (\,\boldsymbol{J}\,\boldsymbol{w}_k - f^{-1}(\boldsymbol{y}_k)\,) \right\| \\ &= \left\| \boldsymbol{R}^T \boldsymbol{R}\,\boldsymbol{A}\,\boldsymbol{w}_k - \boldsymbol{R}^T f^{-1}(\boldsymbol{y}_k) \right\| \qquad (7.10) \end{aligned}$$

But $\boldsymbol{R}$ is orthogonal, so

$$\left\| \boldsymbol{J}\,\boldsymbol{w}_k - f^{-1}(\boldsymbol{y}_k) \right\| \;=\; \left\| \boldsymbol{A}\,\boldsymbol{w}_k - \boldsymbol{R}^T f^{-1}(\boldsymbol{y}_k) \right\| \qquad (7.11)$$

If $\boldsymbol{A}$ has non-zero elements in the leading diagonal (e.g. $\boldsymbol{J}$ is non-singular), the least squares solution $\hat{\boldsymbol{w}}_k$ that minimises the norm $\left\| \boldsymbol{J}\,\boldsymbol{w}_k - f^{-1}(\,\boldsymbol{y}_k) \right\|$ can be found by back substitution in

$$\boldsymbol{A}\;\boldsymbol{w}_k \;=\; \boldsymbol{R}^T f^{-1}(\,\boldsymbol{y}_k) \qquad (7.12)$$

A great computational saving can be made if we choose a method that computes the QR factorisation of $\boldsymbol{J}$ iteratively so that the jth column of $\boldsymbol{R}$ is generated after the jth step. If the hidden units are added one by one, then any increase in computational complexity is solely due to the next $\boldsymbol{R}_j$. Since the Gram-Schmidt algorithm [116] generates each $\boldsymbol{R}_j$ as a linear combination of $\boldsymbol{\theta}_1 \cdots \boldsymbol{\theta}_{j-1}$, $\boldsymbol{\theta}_i \in R^m$, it makes sense to choose this algorithm. When the Gram-Schmidt process is implemented on a computer rounding-off errors can cause significant loss of orthogonality. For this reason, the process is said to be unstable. The process can be stabilised with a small modification. The nth iteration of the stabilized Gram-Schmidt process can be summarized as follows

$$\begin{aligned} & a_{jn} = \boldsymbol{\theta}_n^T \boldsymbol{R}_j, \quad \boldsymbol{\theta}_n \leftarrow \boldsymbol{\theta}_n - a_{jn}\boldsymbol{R}_j, \quad j = 1, 2, \cdots n-1 \\ & a_{nn} = \| \boldsymbol{\theta}_n \|, \quad \boldsymbol{R}_n = (1/a_{nn})\boldsymbol{\theta}_n \end{aligned} \tag{7.13}$$

According to equation (7.13), a_{1n} is calculated by multiplying two m-dimensional vectors, $\boldsymbol{\theta}_n$ and $\boldsymbol{R}_1$, which requires $2m$ computations. The projection of $\boldsymbol{\theta}_n$ orthogonally onto $\boldsymbol{R}_1$ also requires $2m$ computations. Thus, the number of operations is $4m$. There are totally $(n-1)$ of $\boldsymbol{R}_j$. Therefore, the number of operations is $4m(n-1)$. Finally, $\boldsymbol{R}_n$ is calculated by normalizing $\boldsymbol{\theta}_n$, which requires $2m$ computations. Therefore, the total number of operations at is $4m(n-1) + 2m$. Summing over all $\boldsymbol{R}_j$, there are totally $4m\sum_{j=1}^{n} j - 4m\sum_{j=1}^{n} 1$ $+ 2m\sum_{j=1}^{n} 1$ = $2mn^2$ operations

A simple mechanism to avoid numerical ill-conditioning in the matrix $\boldsymbol{J}$ can be built into the Gram-Schmidt procedure. A very small a_{nn} implies that the vector $\boldsymbol{\theta}_n$ is a linear combination of $\boldsymbol{\theta}_1 \cdots \boldsymbol{\theta}_{n-1}$. Therefore, if a_{nn} is less than a certain threshold value, a new $\boldsymbol{\theta}_n$ should be generated by randomizing the weights w_{ni}. The procedure is then repeated for the nth step.

7.1.4 The stopping criteria

The stopping criteria used to terminate training is the Akaike Information Criteria (AIC) [117]. The AIC is an effective measure of the trade-off between the model accuracy and model complexity, and has been used elsewhere as stopping criterion in related models [22,45]. The AIC criteria can be written as

$$AIC(\chi) = N\,log(e) + k\,\chi \tag{7.14}$$

where N is the number of training data, e is the variance of the model residuals (errors), k is the number of hidden units, and χ is the critical value of the χ^2 distribution with one degree of freedom for a given significance level.

7.1.5 Implementation of the procedure

The following section is included to clarify the procedures for updating the weights of a network using the algorithm outlined in this chapter. Given a set of training data $\{(\boldsymbol{x}_i, y_{ki}) : \boldsymbol{x}_i \in R^N,\ y_{ki} \in R,\ i = 1, 2, \cdots, m\}$

Step 1

Initialise the constant output vector $\boldsymbol{\theta}_1 \in R^m$

Compute the norm $a_{11} = \|\boldsymbol{\theta}_1\|$

Compute first orthonormal vector $\boldsymbol{R}_1 = (1/a_{11})\boldsymbol{\theta}_1$

Next n

Step n

Initialise the weight vector $\boldsymbol{w}_n \in R^N$

Compute the output vector $\boldsymbol{\theta}_n \in R^m$

```
    For $j = 1$ to $n-1$
        Compute the inner product $a_{jn} = \boldsymbol{\theta}_n^T \boldsymbol{R}_j$
        Set $\boldsymbol{\theta}_n \leftarrow \boldsymbol{\theta}_n - a_{jn}\boldsymbol{R}_j$
    Compute the norm $a_{nn} = \|\boldsymbol{\theta}_n\|$
    Compute $nth$ orthonormal vector $\boldsymbol{R}_n = (1/a_{nn})\boldsymbol{\theta}_n$
    If $a_{nn} < \delta$, then Goto Step $n$
    Solve $\sum_{i=1}^{n} a_{ji} w_{ki} = \boldsymbol{R}_j^T f^{-1}(\boldsymbol{y}_k),\ j = 1, 2 \cdots n$
    If model adequate, then Stop
    Next $n$
    Goto Step $n$
```

7.1.6 Relationship with existing methods

The GN-TPA algorithm presented in this thesis shares some similarities with the orthogonal sequential training technique developed by Zhang and Morris [22]. The orthogonal sequential training technique adds new hidden units one by one to a single hidden layer feed-forward neural network. When adding a new unit the new information introduced by this unit is caused by that component of its output that is perpendicular to the space spanned by the outputs from the previously added units. The new units are trained to minimise the error in the contribution they make to the overall error. Learning is terminated when the network error is sufficiently small. The model robustness was improved by modifying the cost function to include a novel regularisation term

Both the GN-TPA algorithm and the orthogonal sequential training technique build neural networks in a constructive way. They eliminate the need to guess in advance the size of the network. However the GN-TPA differs from the orthogonal sequential training technique in two respects. Firstly, the GN-TPA algorithm sets the input weights and biases of new hidden units to arbitrary values and fixes them; only the output layer weights are trained. The orthogonal sequential training technique uses error minimization to train the new hidden units and output connections. This suggests that the orthogonal sequential training technique is likely to produce smaller more economical networks. Secondly, the solution of the GN-TPA algorithm is the solution with the smallest least squares, meaning that the smallest training error can be reached by this solution. The orthogonal sequential training technique attempts to minimise the cost function using a gradient descent based algorithm which is prone to getting stuck in local minima.

7.2 Simulations and results

Comparative tests were performed using three network-building techniques; the GN -TPA algorithm, the orthogonal sequential training technique [22], and the cascade algorithm [8,9]. These are methods that start with a small network and insert additional units and connections until the network can represent the required function. The orthogonal sequential training technique inserts new units and connections one by one into a single hidden layer feedforward neural network. The new hidden units are trained to minimise the contribution they make to the overall network error. The learning rule used to train the new units is the standard back-propagation algorithm. The cascade algorithm inserts new units and connections one by one, each into a different hidden layer of a cascade neural network. After a new hidden unit is inserted, its output connections are trained by error minimization. The learning rule used for candidate training is the Rprop (resilient back-propagation) algorithm [28]. Rprop is a very fast gradient descent based learning rule that uses the signs of the current and previous partial derivatives of the error function to adapt the weights.

The results presented in this investigation were averaged over 20 trials. For each trial carried out, the error on the training set and test set were recorded together with the number of units inserted into the network. The best results on the test set were also recorded. The method of early stopping was used to terminate each stage of network training. Overall network training was terminated when the Akaike Information Criterion reached its minimum. Hidden unit training was terminated when the generalization loss on the test set was significant, and there was little progress made on the training set [44].

7.2.1 Network initialization

The GN-TPA algorithm and the sequential orthogonal training technique are methods that build single hidden layer neural networks. In all simulations, the weights of the new units were initialised to random values in the range [-2, 2]. The choice of network architecture for the cascade algorithm is not critical. Prechelt [8] investigated the effect of cascading hidden units versus not cascading hidden units for six members of the CasCor family. He concluded that in most cases there was no significant difference. We will use a cascade network in this investigation as the performance of the cascade algorithm has been well documented for this type of network. The weights of the hidden units were initialised to random values in the range [-0.1, 0.1], which according to Lahnajarvi [9] gives the best results for the problems used in this investigation.

The Rprop algorithm and the back-propagation algorithm require parameters that need to be set manually. First, the Rprop algorithm. The parameters used with Rprop are the step increment factor η^{+}, the step decrement factor η^{-}, the initial step size $\Delta^{(0)}$, the maximum step size Δ_{max}, and the minimum step size Δ_{min}. The step increment and decrement factors were chosen to be the same as in the original paper, i.e. $\eta^{+} = 1.2$, $\eta^{-} = 0.5$. The initial step size is not critical, and was set to $\Delta^{(0)} = 5x10^{-3}$. The maximum step size was also chosen to be the same as in the original paper, i.e. $\Delta_{max} = 50$. Finally, the minimum step size was set to $\Delta_{min} = 1x10^{-8}$ to avoid overflow/underflow problems with floating point variables. Second, the back-propagation algorithm. The learning rate η was set to $\eta = 0.1$. A small value for the learning rate was chosen to avoid oscillatory behaviour, the training stage of candidate units being very turbulent. In any event the convergence speed of the learning algorithm is not critical

7.2.2 The error metric used to determine convergence

The error metric used in the simulations is NMSE (Normalized Mean Square Error). NMSE is given by

$$NMSE = \frac{1}{m\sigma^{2}} \sum_{i} \left(y_{ki} - \theta_{ki} \right)^{2} \qquad (7.15)$$

where m is the number of training patterns, y_{ki} is the target value of the ith input pattern, θ_{ki} is the ith network output, and σ^{2} the variance of the target data.

7.2.3 Simulations problems

The training sets used in all the simulations were regression problems, which are problems that involve the approximation of a continuous valued target function. The regression tasks chosen are the additive problem [22], the housing price estimation problem [61], and the Henon map chaotic time series [9].

The house price estimation problem is a real-world problem that estimates the price of houses in the suburbs of Boston based on some attributes of houses (e.g.

location, crime rate, level of air pollution, etc.). The number of inputs is 13, and the number of outputs is one. The number of training and testing examples is 253. The data sets used in the simulation were sampled randomly from the dataset provided by the UCI data repository with the outputs scaled down in the range [-1,1].

The additive problem is a non-linear function approximation problem that was obtained from [22]. The function is computed by

$$z = -0.5x_1^2 - x_1x_2 + 0.5\,x_2^2 + \frac{x_1}{1 + x_2^2} + 0.5\,e^{x_1 - x_2} + 0.05\,u \qquad (7.16)$$

A small signal u is added to the output with values uniformly distributed in the range [-1,1]. Four hundred data points are generated. The first 200 data points are used as training data whilst the remaining are used as test data. The input values are uniformly distributed in the range [-1,1]. All functional values or outputs are scaled down in the range [-1,1]

The henon map is a time series prediction problem. The chaotic time series data is computed as follows

$$x^{(t+1)} = 1 - c\left(x^{(t)}\right)^2 + b\,x^{(t-1)} \qquad (7.17)$$

where $x^{(t)}$ is the value at time t, while $b = 0.3$ and $c = 1.4$ are parameters. Initial values for the time series are $x^{(1)} = x^{(0)} = 0.63133545$ [95]. In neural network simulations, four successive values are used to predict the next value. Thus, the number of inputs is four, and the number of outputs is one. The number of training and testing samples is 200, and the used data values are taken from the time step range [31, 230]. Since x_{max} = 1.272967 and x_{min} = -1.284657, the input values were scaled in the range [-1,1].

7.2.4 Discussion of results

Additive function. The first test is a non-linear function approximation problem obtained from [22]. The results are tabulated in Table 7.1a. It was found that the GN-TPA algorithm gave the best results on the test set (Err* = 1.4×10^{-4}), and the cascade algorithm the worst (Err* = 3.5×10^{-4}). However, the best results in terms of the network size were obtained by the cascade algorithm which constructed the smallest networks (e.g. between 15 and 32 units), and the GN-TPA algorithm the worst (e.g. between 21 and 40 units). This result is not surprising as the GN-TPA algorithm does not optimize the new hidden units, so the amount of new information they introduce into the network may be very small. Fig. 7.1, 7.2 and 7.3 display the variation in the error on the training and test sets unit by unit, for each of the three algorithms. The training curves for both the GN-TPA algorithm and cascade algorithm are quite smooth whereas the curve for the orthogonal sequential training technique contains two small kinks. There is very little evidence of overtraining in any of the generalization curves. This suggests that each of the three algorithms is perfectly capable of constructing a network that can perform an accurate mapping of the simulation data. Note that the generalization curve produced by the orthogonal sequential training technique is a good match with Zhang's results in [22]. Figure 7.4 shows the variation in the AIC, unit by unit, for the GN-TPA algorithm. It was found that the AIC stopped decreasing after using 30 units. This result is in good agreement with the average size of the network constructed by the GN-TPA algorithm (e.g. 29.1 units). Note that the general trend of the AIC curve is similar to the results reported by Zhang [22].

(a)

	Best test set error using early stopping (NMSE x 10^2)			Avg. test set error using early stopping (NMSE x 10^2)		
Problem	Size	Err	Err*	Min-Max	Err	Err*
Additive	21	0.001	0.001	21-40	0.009	0.014
Henon	19	1.070	1.347	14-30	1.063	2.680
Housing	17	4.871	14.030	06-21	9.147	19.417

(b)

	Best test set error using early stopping (NMSE x 10^2)			Avg. test set error using early stopping (NMSE x 10^2)		
Problem	Size	Err	Err*	Min-Max	Err	Err*
Additive	27	0.001	0.012	13-29	0.012	0.035
Henon	12	1.559	2.131	09-23	3.190	4.880
Housing	13	4.838	11.322	07-16	5.842	15.617

(c)

	Best test set error using early stopping (NMSE x 10^2)			Avg. test set error using early stopping (NMSE x 10^2)		
Problem	Size	Err	Err*	Min-Max	Err	Err*
Additive	25	0.003	0.005	15-32	0.026	0.024
Henon	40	0.467	1.476	10-40	2.975	4.046
Housing	15	8.793	10.552	05-22	9.667	12.252

Table 7.1. Training set error (Err) and test set error (Err*) for three problem domains using: (a) GN-TPA algorithm, (b) Cascade algorithm, and (c) Orthogonal sequential training technique

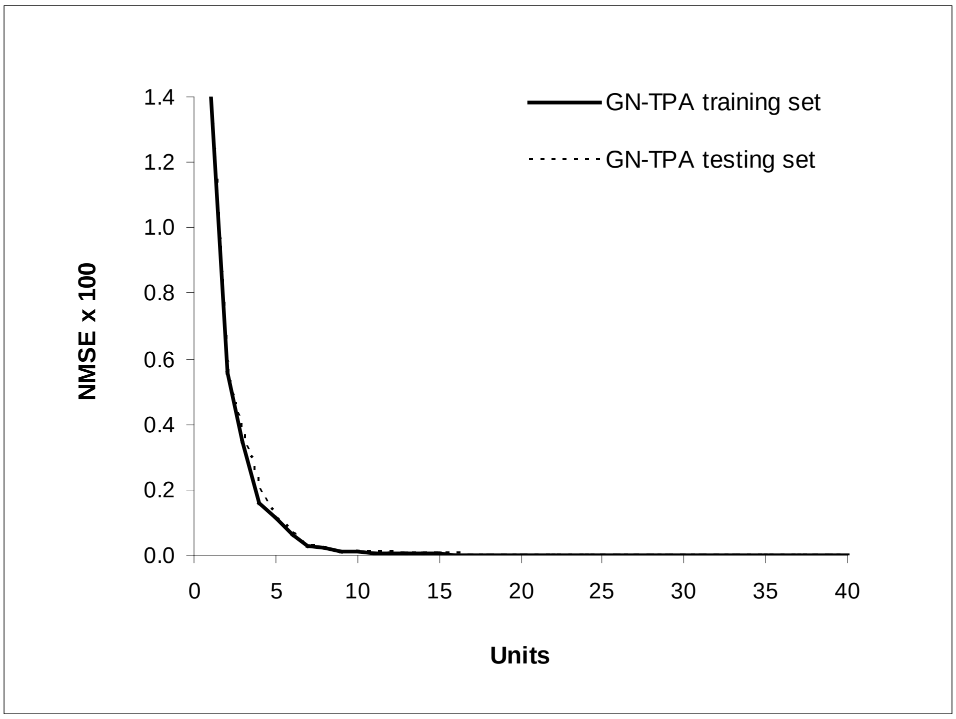


Fig.7.1. Typical training and generalization curves produced by the GN-TPA algorithm on the additive problem

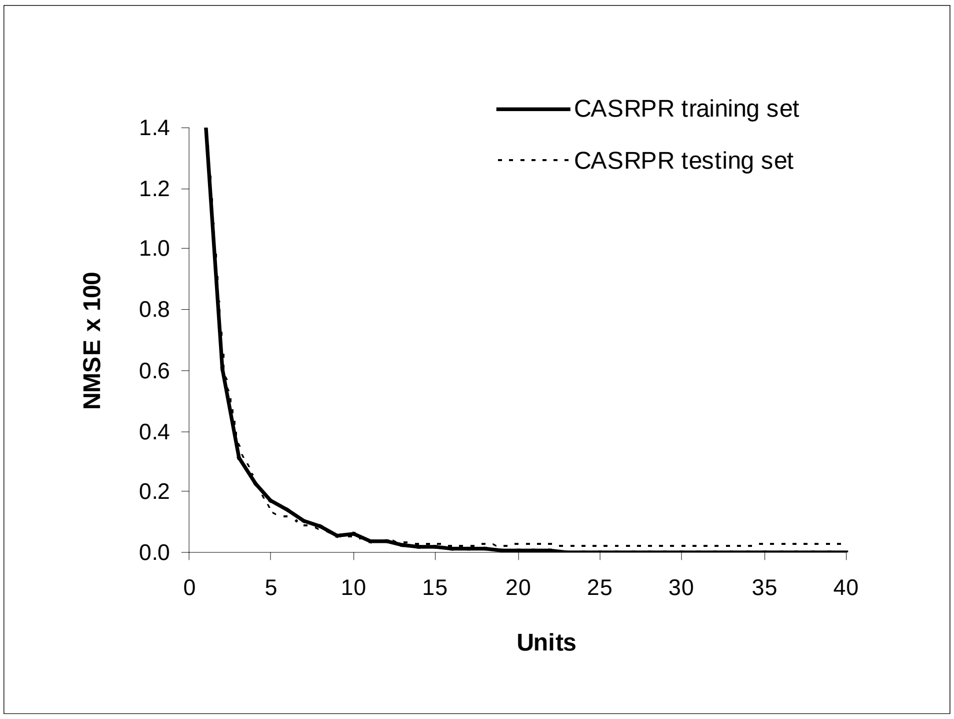


Fig.7.2. Typical training and generalization curves produced by the Cascade algorithm on the additive problem

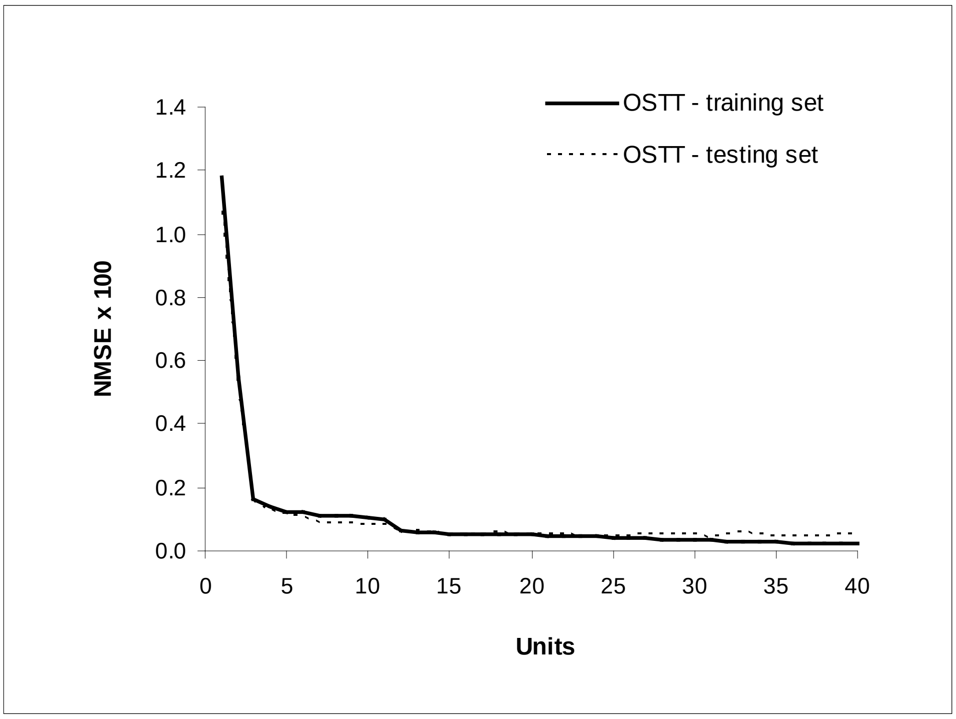


Fig.7.3. Typical training and generalization curves produced by the sequential training technique on the additive problem

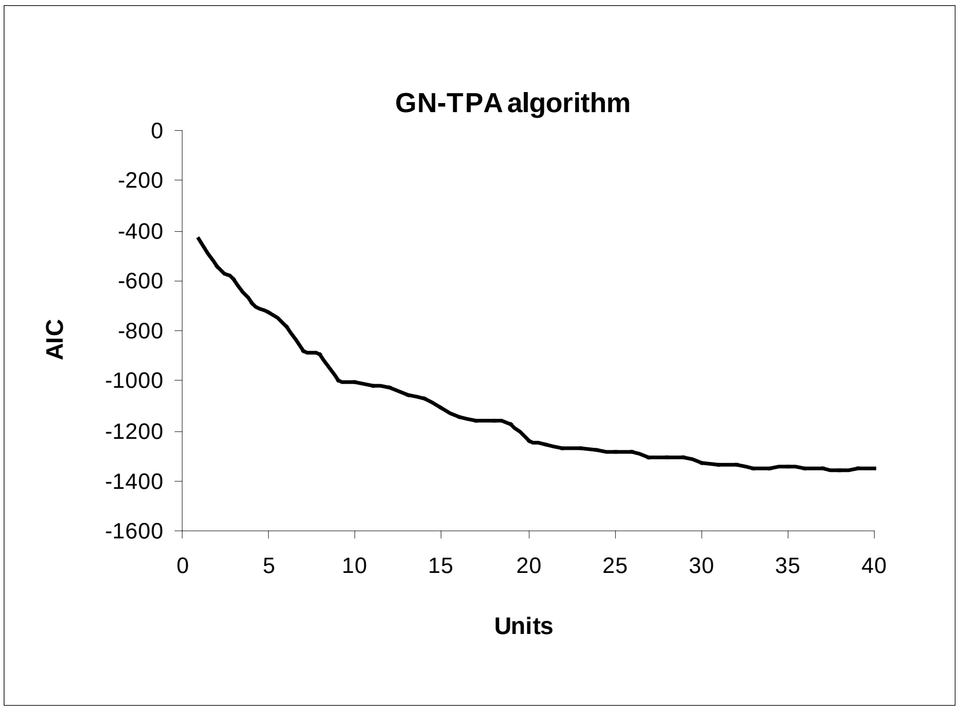


Fig.7.4. The AIC at each training step for the GN-TPA algorithm on the additive function problem

Henon map time series. The second test is a deterministic one-step-ahead prediction problem. The results are tabulated in Table 7.1b. It was found that the GN-TPA algorithm gave the best results on the test set (Err* = 2.68 x 10^{-2}), and the Cascade algorithm the worst (Err* = 4.88 x 10^{-2}). The best results in terms of the network size were obtained by the Cascade algorithm, which constructed the smallest networks (e.g. between 9 and 23 units). The orthogonal sequential training technique constructed the largest networks (e.g. between 10 and 40 units) and not the GN-TPA algorithm as expected. The results for the cascade algorithm are similar to those obtained by Lahnajarvi [9]. Figure 7.5 shows the model output produced by the GN-TPA algorithm for the Henon map data. The test data used was taken from the time step [100,130]. As can be seen, the network produces a good representation of the target data. Figures 7.6, 7.7 and 7.8 show the training and test curves for the GN-TPA algorithm, cascade algorithm, and orthogonal sequential training technique respectively. The results for the GN-TPA algorithm and the orthogonal sequential training technique are very similar. Both methods produce very smooth training curves. Mild overtraining occurred in both test curves after 25 units. In the case of the orthogonal sequential training technique this is probably due to poorly conditioned $\boldsymbol{R}$ and $\boldsymbol{\theta}$ matrices used to train the output unit. As pointed out by Zhang [22], the likelihood of two or more columns in $\boldsymbol{\theta}$ being very nearly parallel increases as more units are inserted into the network and trained to minimise the residual model error. The Cascade algorithm performed less well as seen from Figure 7.7. The training curve gradually tapers off after five units resulting in long runtimes. It was during the later stages of training that gross overfitting occurred. This result is typical of the generalization behaviour found in cascade networks [8]

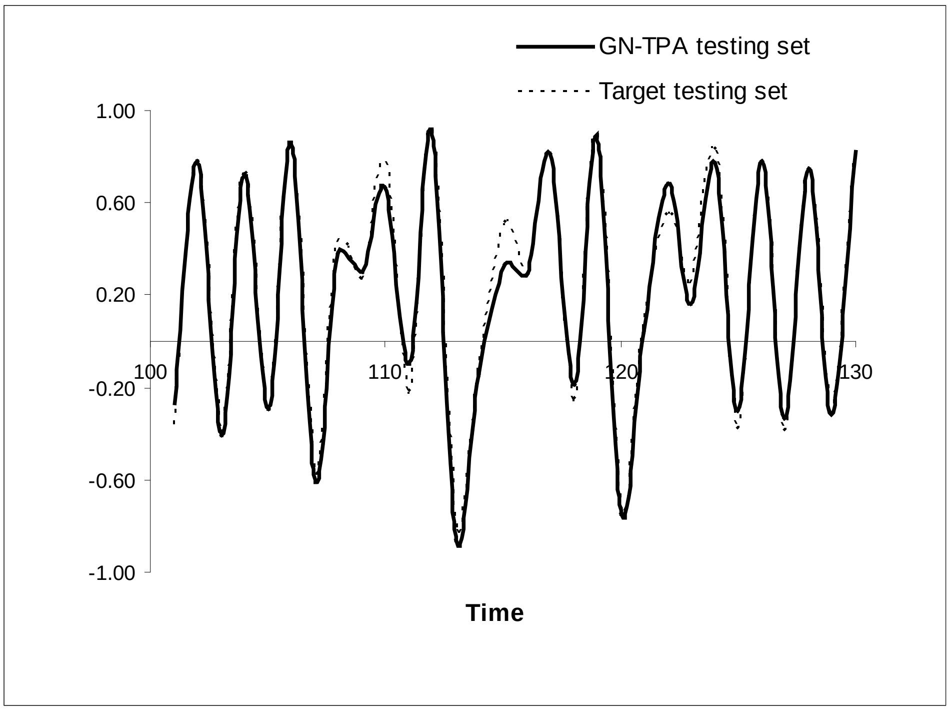


Fig.7.5. Plot of the model output generated by the GN-TPA algorithm on the Henon map problem

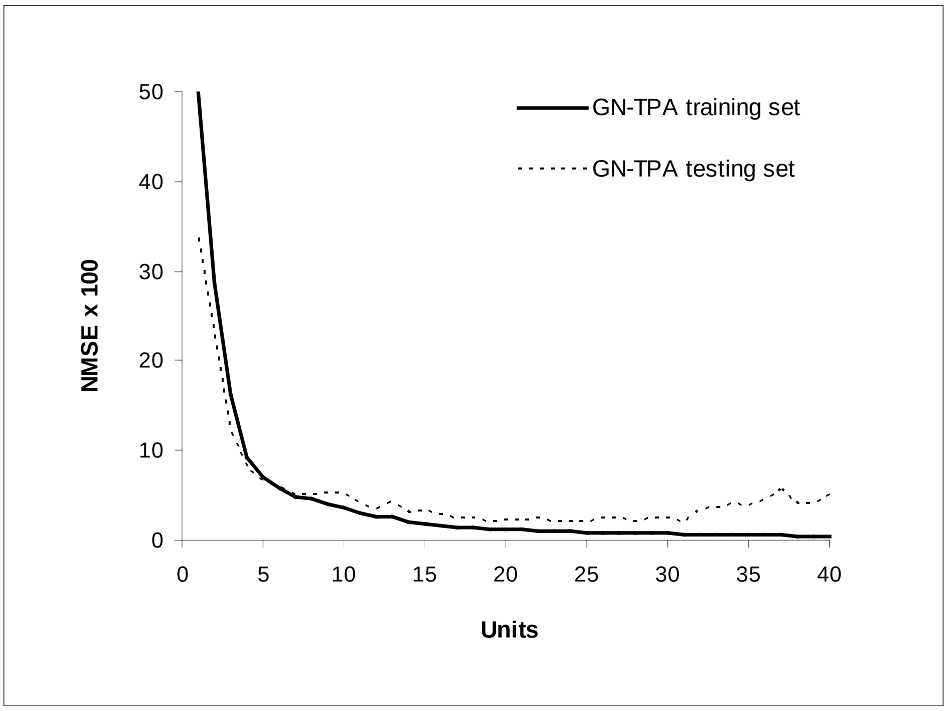


Fig.7.6. Typical training and generalization curves produced by the GN-TPA algorithm on the Henon map problem

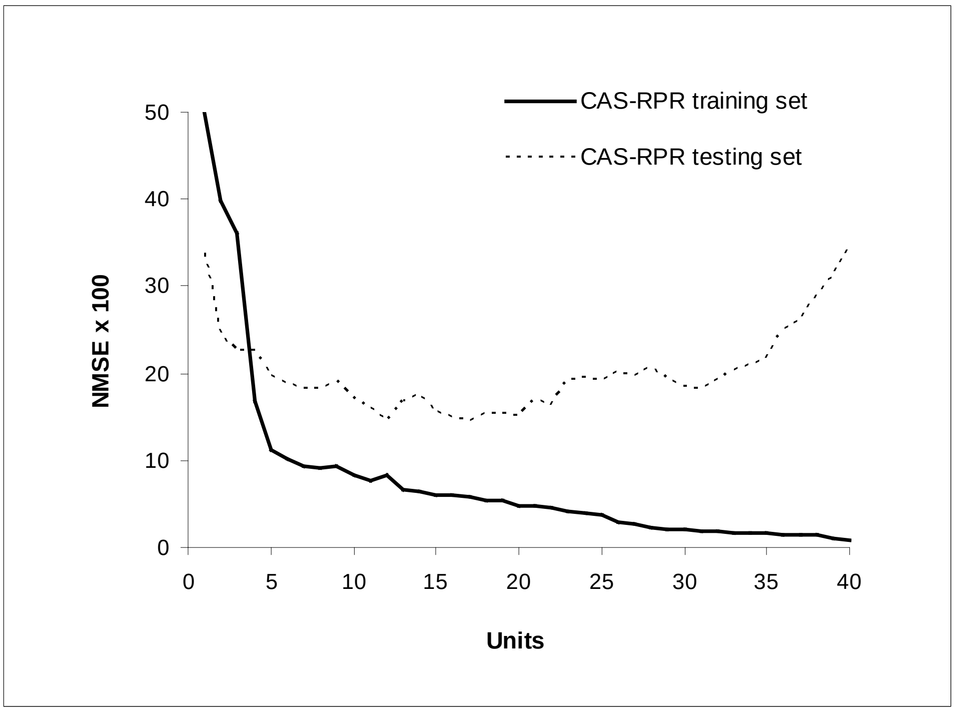


Fig.7.7. Typical training and generalization curves produced by the Cascade algorithm on the Henon map problem

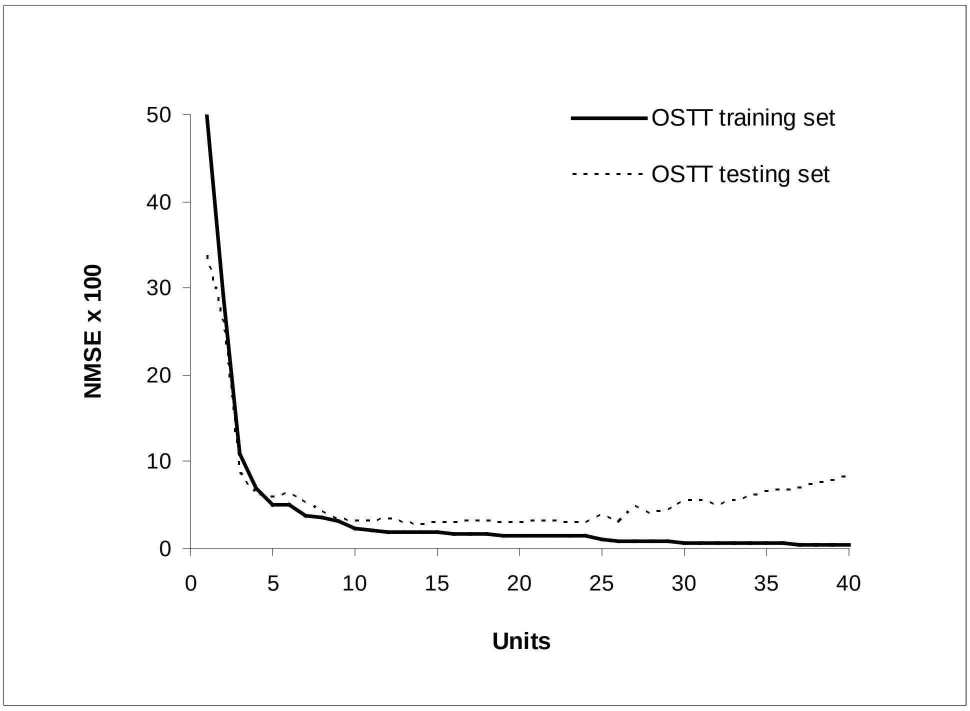


Fig.7.8. Typical training and generalization curves produced by the sequential training technique on the Henon map problem

Housing price estimation. The third test is a regression problem that predicts the price of houses in the suburbs of Boston based on some attributes. The results are tabulated in Table 7.1c. This was a far more challenging test for the GN-TPA algorithm (Err = 9.15 x 10^{-2}, Err* = 19.42 x 10^{-2}). The size of the networks constructed contained between 6 and 21 units. The cascade algorithm performed much better giving Err = 5.84 x 10^{-2}, Err* = 15.62 x 10^{-2}. As expected, the cascade algorithm constructed the smallest networks containing between 7 and 16 units. These results are similar to those obtained by Lahnajarvi [9]. Finally, the orthogonal sequential training technique gave Err= 9.67 x 10^{-2}, Err* = 12.25 x 10^{-2}. The size of the networks constructed contained between 5 and 22 units. Figures 7.9, 7.10 and 7.11 show the training and test curves for the GN-TPA algorithm, cascade algorithm, and orthogonal sequential training technique respectively. The training curve for the orthogonal sequential training technique tapers off gradually resulting in very long runtimes. The test curve exhibits mild overtraining over the full range of hidden units. This suggests that the individual contributions of new hidden units must be very small resulting in slow convergence. Very small $\left\| \boldsymbol{R}_j \right\|$ will lead to ill conditioned $\boldsymbol{R}$ and $\boldsymbol{\theta}$ matrices, which in turn will contribute to poor generalization performance. The training curve for the cascade algorithm drops off steeply at first giving good convergence. The test curve dips quickly to a clearly defined minimum after seven units and thereafter rises sharply. This is fairly typical behaviour for the cascade algorithm and has been observed elsewhere (e.g. Henon map problem). The training curve for the GN-TPA algorithm also drops off steeply at first giving good convergence. The test curve quickly dips to a flat plateau after 5 units. There is very little evidence of overtraining in the test curve. This is the principal strength of the GN-TPA algorithm.

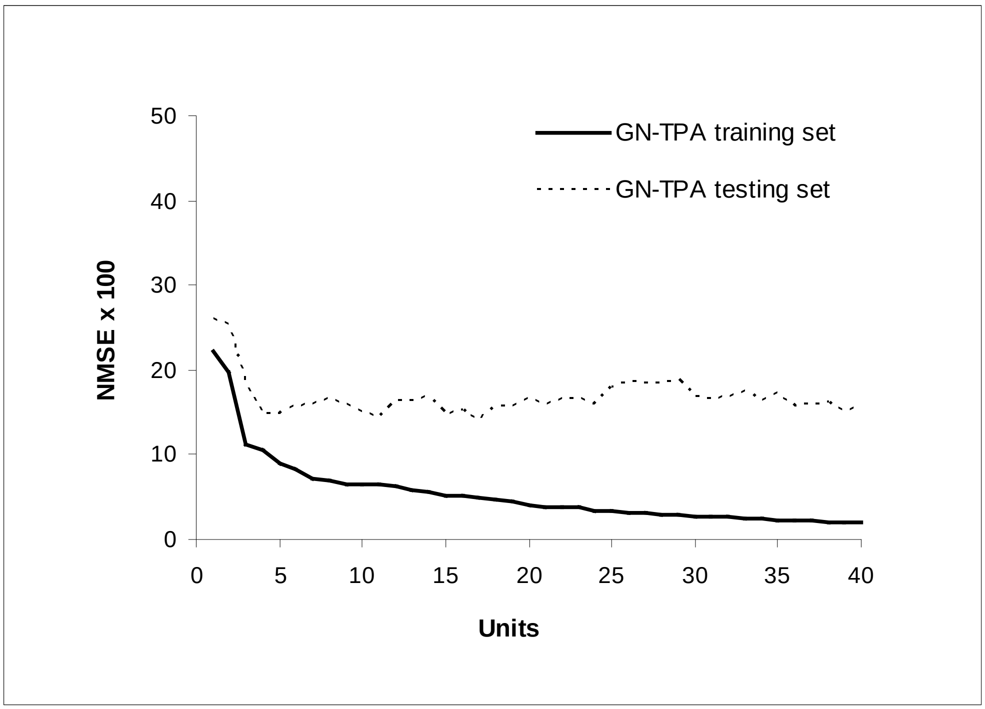


Fig.7.9. Typical training and generalization curves produced by the GN-TPA algorithm on the housing estimation problem

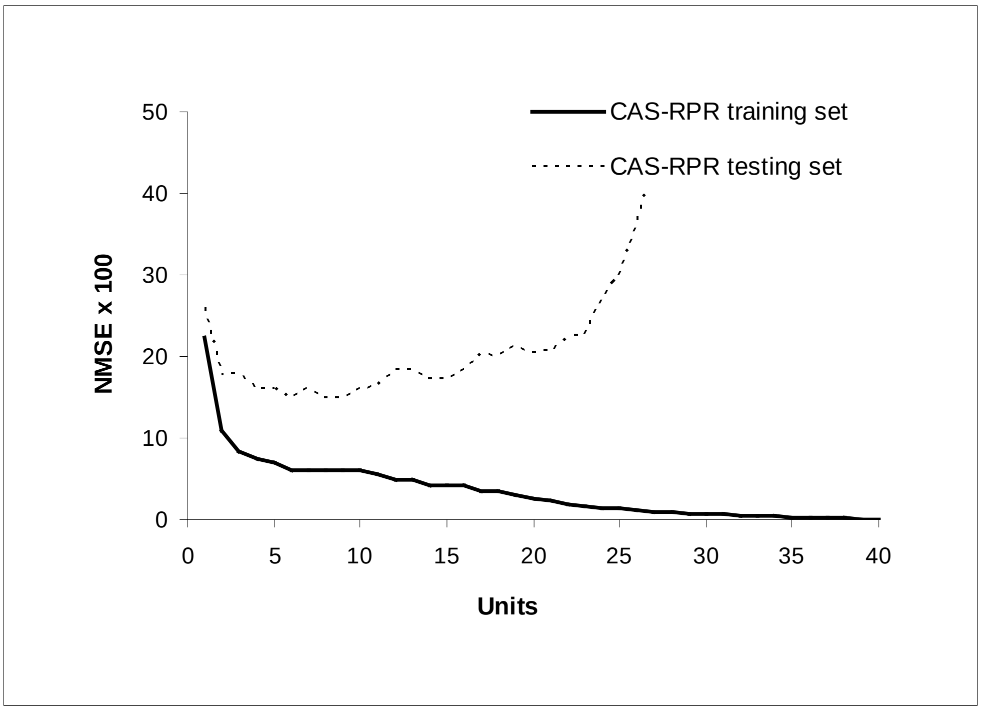


Fig.7.10. Typical training and generalization curves produced by the Cascade algorithm on the housing estimation problem

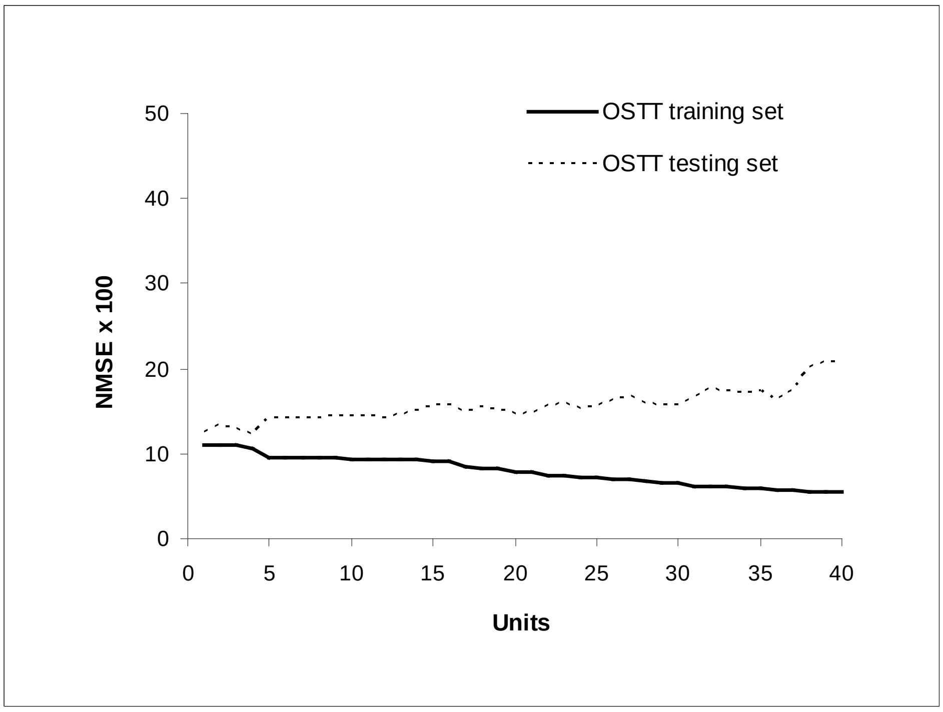


Fig.7.11. Training and generalization curves produced by the sequential training technique on the housing estimation problem

7.3 Summary

In this chapter, the newly developed batch tangent plane algorithm referred to as GN-TPA is evaluated for a novel network structure called the extreme learning machine. This extreme learning machine is a single hidden layer neural network (SLFN) with input weights fixed at arbitrary values. The SLFN is then treated as a linear system with the output layer weights determined analytically. The smallest training error can be achieved using this method. The new algorithm is modified in order to improve its computational efficiency by using the QR decomposition. The outputs of the hidden units are projected one by one orthogonally onto the hidden layer output space. This means that any increase in the computational cost is solely due to the next hidden unit. Network training is terminated when the model performance is satisfactory. The procedure is very fast and stable and avoids problems like ill conditioning

Comparative tests were carried out using the GN-TPA algorithm, the cascade algorithm, and the orthogonal sequential training technique. The benchmark datasets used in the study were the additive function, henon map, and housing price. The results show that the GN-TPA algorithm gives the best training and test set errors relative to the other algorithms on two of the datasets. The GN-TPA algorithm does not suffer from the same computational difficulties as the orthogonal sequential training technique, namely numerical ill-conditioning in the orthogonal matrix. Generalization appears to be independent of network size. The principal weakness of the GN-TPA algorithm is that it constructed the large networks compared with the other algorithms; although this does not appear to degrade generalization performance.

In the next chapter we investigate multi-classification problems in the bioinformatics area using gene expression data. Two cancer classification problems are investigated that have proven difficult for conventional neural network techniques to solve, GCM [118] and Lymphoma [119]. The new iTPA and GN-TPA tangent plane algorithms developed in this thesis are applied to multi-category cancer classification problems, and compared with other current classification methods, a support vector machine (SVM) and two newly developed algorithms called subsequent ANN (SANN) and FGAP-RBF.

Chapter 8

MULTI-CATEGORY CANCER CLASSIFICATION USING THE TANGENT PLANE ALGORITHM

The artificial neural network (ANN) has been well established as a classifier for its unique capability to represent non-linear mappings between the input and output data. It can perform complex non-linear mappings by encoding the input patterns into a high dimensional feature space. In this feature space the input patterns can be mapped directly into a number of different classes. It is this ability of ANN to map the input data directly into a number of classes that has seen their increasing use as intelligent alternatives to more established classifiers such as support vector machines in the area of cancer classification.

The first application of ANN for diagnostic classification of cancer using gene expression data was done by Khan et al [79]. In this paper a two hidden layered neural network was used to classify small, round blue-cell tumours into 4 diagnostic categories. The ANN method correctly classified all the samples which often present difficulty in clinical diagnosis. Stratnikov et al [78] presents a comparison of multi-classification methods for gene expression cancer diagnosis problems. In this paper several methods are compared such as SVM, k-nearest neighbour, weighted voting and back-propagation neural networks. The benchmark datasets used were 11-Tumours [121], GCM [118], 9_Tumours [122], Brain_Tumour1 [123], Brain_Tumour2 [124], Leukemia [70], MLL [125], Lung cancer [126], SRBCT [79], Prostrate_Tumour [127], and DLBCL [128]. The results show that the SVM based classifiers are the best performers, whilst k-nearest neighbour and weighted voting are the worst. Neural networks rank in the middle.

In order to improve classification accuracy, several ANNs can be combined either by using ensembles of networks or cascading ANNs. When ANNs are trained for different subtasks instead of the same task, those approaches are combined into a mixture of experts. For example, Qian and Sejnowski [129] have used a two-level ANN to predict the secondary structure of protein. In this scheme the output of the first ANN was used as the input for the second ANN. Employing a consecutive structure network obtained a 2% increase in prediction accuracy. Linder et al [77] have developed a novel neural network algorithm for multi-category classification using micro-array gene expression data. This subsequent ANN (SANN) uses a simple ANN to perform a pre-selection. At the first stage the two most preferred classes are selected. After that a subsequent ANN stage makes the final decision based upon the two most preferred classes. The benchmark dataset used was GCM [118]. The results show that the SANN approach displayed higher classification accuracy than a simple ANN for the range of selected genes [74]. However, this improvement was paid for by a big increase in network complexity causing a great computational burden and very long run-times in terms of the number of epochs trained and the computation time.

In this chapter we present two newly developed algorithms based on the tangent plane algorithm for multi-category cancer classification. The problems used in the investigation are described in the first section. The second and third sections describe the two algorithms and the fourth section the simulation results. Finally a comparison of the two algorithms is presented in section five

8.1 The multi-category classification problems

The first dataset comes from a study of microarray data for snap-frozen human tumour and normal tissue samples, spanning 14 different tumour classes were obtained from four different hospitals. Initial diagnosis was made at university hospital referral centres by using available clinical and microscopic histopathological information. All tumours were biopsy specimens were obtained from primary sites obtained before any treatment and were enriched in malignant cells. Normal tissue RNA was taken from snap-frozen autopsy specimens. Hybridization targets were prepared with RNA taken from whole tumours by using published methods [70]. Targets were hybridized sequentially by using oligonucleotide micro-arrays containing a total of 16,063 genes. Expression values for each gene were calculated by using Affymetrix GENCHIP analysis software. Of 314 tumour samples, and 98 normal tissue samples processed, 218 tumours and 90 normal tissue samples passed quality control criteria and were used for subsequent data analysis. Ramaswamy et al. [74] made this dataset available as a reference for micro-array gene expression profiling at [118] offering several files for download.

- GCM_Training.res (training set: 144 primary tumour samples)
- GCM_Test.res (test set: 46 primary, 8 metastatic)
- GCM_PD.res (poorly differentiated adenocarcinomas: 20 samples)
- GCM_Total.res (training set + test set + normals (90): 280 samples)

In each dataset above, columns represent gene profiles, rows represent samples, and the values are raw averaged real number values output from the Affymetrix package

The data from the above mentioned file GCM_Training.res contains gene expression profiles comprising 16,063 genes and 144 primary tumour samples spanning 14 common tumours. The detailed information of the number of patterns in each class of tumour used for training and testing is given in Table 8.1. All the input attributes were normalised to remove any bias in the mean values

Tumour Type	Abbr.	Sample Size	Training Set	Test Set
Breast	BR	8	6	2
Prostrate	PR	8	6	2
Lung	LU	8	6	2
Colorectal	CO	8	6	2
Lymphoma	LY	16	13	3
Bladder	BL	8	6	2
Melanoma	ML	8	6	2
Uterus-Adeno	UT	8	6	2
Leukaemia	LE	24	19	5
Renal	RE	8	6	2
Pancreas	PA	8	6	2
Ovary	OV	8	6	2
Mesothelioma	ME	8	6	2
CNS	CNS	16	13	3

Table 8.1. Partitioning of the GCM_training.res dataset into training and test samples.

For the purpose of comparison, we use the same gene selection method as in Zhang [110], Linder et al. [77] and Ramsawamy et al [74], which is the recursive feature elimination method. In Ramsawamy et al [74], an SVM-OVA classifier was used for gene expression profiling on multi-category classification micro-array data. Each SVM-OVA classifier produces a hyper-plane in the input space defined by a weight vector $\boldsymbol{w}$, which is a vector of n elements each corresponding to a particular gene.

The absolute magnitude of each element in $\boldsymbol{w}$ can be considered as a measure of the importance of the corresponding gene. In the recursive feature elimination method, each SVM-OVA classifier is first trained with all the genes, and then the bottom 10% genes with the smallest $|w_i|$ are removed. Each classifier is then re-trained. The process is repeated iteratively and a rank of all genes obtained for each class of tumour. The most significant 14, 28, 42, 56, 70, 84, and 98 genes selected by this method can be found in the file OVA_MARKERS.xls. It also gives the most significant genes for each of the 14 classes of tumour.

The second dataset comes from a study of gene expressions for the most prevalent adult lymphoid malignancies: B-cell chronic lymphocrytic leukaemia (B-CLL), follicular lymphoma (FL), and diffuse large B-cell lymphoma (DLBCL) obtained from [119]. Gene expression levels were measured using a special cDNA micro-array containing genes preferentially expressed in lymphoma cells. In each hybridization, fluorescent cDNA targets were prepared from tumour mRNA samples (fluorescent dye Cy5) and a reference sample derived from a pool of nine different lymphoma cell lines (fluorescent dye Cy3). The cell lines in the common pool were chosen to represent diverse expression patterns. The lymphoma dataset contains 62 samples each consisting of 4,062 genes which include 42 cases of DLBCL, 9 cases of FL, and 11 cases of B-CLL. The gene expression data is summarised by an 62 x 4062 matrix $\boldsymbol{X} = [x_{ji}]$ where x_{ji} denotes the base 2 logarithm of the Cy5/Cy3 background corrected fluorescence intensity ratio for gene j in lymphoma sample i.

Tumour Type	Abbr.	Sample Size	Training Set	Test Set
Diffuse Large B-cell Lymphoma	DLBCL	42	35	7
Follicular Lymphoma	FL	9	7	2
B-cell Chronic Lymphoma	B-CLL	11	8	3

Table 8.2. Partitioning of the Lymphoma dataset into training and test samples.

For the Lymphoma dataset, the microarray data contained a number of genes with fluorescent intensity too low to be recorded and flagged as missing. The mean percent of missing data in the array is 6.6%. The missing data was imputed using a *k* nearest neighbour algorithm as in [72,76]. For each gene expression profile with missing data, *k* other gene expression profiles that are most similar are found and then the weighted average of the corresponding attributes used to estimate the missing attribute. The metric used to determine the *k* nearest neighbours is the Euclidean distance. A value of *k* = 5 was used as in [72].

Many genes exhibit near constant expression levels across tumour samples. In this study a preliminary selection of genes based on the ratio of their between-group to within-group sum of squares is used to sort the genes in a descending order of importance as in [72,76] e.g.

$$BW_j = \frac{\sum_i \sum_k I(y_i = k)\left(\hat{x}_{kj} - \hat{x}_j\right)^2}{\sum_i \sum_k I(y_i = k)\left(x_{ij} - \hat{x}_{kj}\right)^2}$$

where $\hat{x}_j$ and $\hat{x}_{kj}$ represent the average expression level of gene j across all tumour samples i, and across tumour samples belonging to class k. The function I returns the value 1 when the sample class label y_i equals k, otherwise it returns 0.

8.2 The sequential learning algorithm - iTPA-OVA

In chapter five, a new algorithm referred to as iTPA algorithm was developed to overcome the drawbacks of the second tangent plane algorithm. Compared with the original algorithm, this new iTPA algorithm produces the desired separation of active and inactive weights for good generalization to occur. However, the new iTPA algorithm may face difficulties in applications that have multiple outputs as the weights need to be adjusted to satisfy several constraints. Consider a network with r outputs, each of which is trained to recognise a target output. On the presentation of an input pattern, each target output would define a ($n-1$) surface in the weight space R^n. A powerful convergence method would be to adjust the weights by moving to the intersection of the tangent planes to these r surfaces. However, this would produce a great computational overload as it requires a system of r equations to be solved for each input pattern.

Recently support vector machines (SVM) have been widely used for cancer classification problems [73,74,75]. Some combinatory schemes have been used to modify SVM for multi-category classification. Ramsawamy [74] have used a one-versus-all (OVA) scheme to perform multi-classification using SVM. For a k classification problem, k binary classifiers would be used to distinguish one class from all others. Ramaswamy applied this method to the GCM dataset. The results show that SVM using an OVA scheme is best suited for classification. The method adopted here uses a similar approach. Instead of training a single ANN on one task, several ANN are trained on separate sub-tasks

The method assumes a modular network structure. Within the modular architecture each module represents a single hidden layer ANN. Each ANN is trained to discriminate one particular class from all others (OVA). 1-of-c encoding is used for the target outputs. Upon the presentation of an input pattern, the ANN corresponding to the input class has its output set to +1; all other outputs are set to -1. The new algorithm referred to as iTPA-OVA uses the target output encodings to define a set of $(n-1)$ surfaces in the weight space of the modular ANN. The weights are adjusted by moving to a point on the tangent planes to these surfaces, each taken at a convenient point.

The principal benefit of using a modular network is that it dichotomises a multi-class problem into a set of two class problems. Binary classification is a much easier task for ANN than multi-category classification and classification accuracy is much higher. A further benefit is that the one-versus-all scheme is scalable. A k class problem will require only k modular ANN to be trained. An alternative method, which is to use a one-versus-one scheme, requires $k(k-1)/2$ ANN to be trained which will produce a great computational overload as k increases

8.3 The batch learning algorithm - ELM-TPA

The new GN-TPA algorithm is attempted for multi-category classification. 1-of-c encoding is used for the target outputs. The number of output units is equal to the number of classes in the problem. The index of the output unit with the highest output activity indicates the class label of the corresponding input. In the ELM architecture, the input weights are chosen arbitrarily and fixed so that they do not change. Huang et al [108,109] have shown that the outputs from the hidden units will form a spanning set in R^m as the number of outputs n approaches m, the number of training examples. Each output unit is trained to discriminate one class from all others using the new GN-TPA algorithm. Since the inputs to the final units form a linear combination of the spanning set, we expect the outputs from the final units to match exactly the target values of the corresponding inputs.

8.4 Discussion of results for the individual algorithms

The classification performance of the iTPA-OVA algorithm and the GN-TPA algorithm are evaluated for multi-category classification problems using micro-array gene expression data, namely the GCM dataset [118] and the Lymphoma dataset [119]. The results obtained were compared with the best results for other classification methods found in the literature; SANN, SVM OVA, SVM OVO [74], and FGAP-RBF [110]. All the results for SANN, SVM OVA, SVM OVO, and FGAP-RBF are taken from the literature

In our simulations, 10 trials were carried out with the mean classification accuracy recorded for the most significant genes in each dataset. For the GCM dataset the most significant 14, 28, 42, 56, 70, 84 and 98 genes can be found in the file OVA_MARKERS. For the Lymphoma dataset the genes were sorted according to the ratio of their “between group” to “within group” sum of squares and the top 10, 20, 50, 100, 200, 400 and 800 genes selected as in [110]. The Lymphoma dataset also required missing genes to be calculated. A *k*-nearest neighbour algorithm was used for this purpose with $k = 5$. Multi-fold cross-validation was used on the sample data as the number of training samples is relatively small. The sample data was split into two subsets, a training set and test set, according to the ratio 4:1 as in Linder et al [77] with different shuffles of the data used in each trial.

8.4.1 Error metrics used in the simulations

A simple measure is used to determine the classification accuracy. For each problem, the number of output units in the network corresponds to the number of classes of the problem. The index position of the output unit with the highest output corresponds to the class of the input data. Thus the classification accuracy is the percentage of correct responses on the test data.

8.4.2 The iTPA-OVA algorithm

In our simulations, all the input attributes were normalised. 1-of-*c* encoding was used to label each class category. The number of outputs is equal to the number of class categories. The output of the neuron with the highest value indicates class membership of the corresponding input. The size of each module was restricted to 10 units to save on computation size. The weights were initialized to random values in the range [-1,1]. iTPA requires three parameters to be set. Preliminary tests showed that the best results were obtained with the parameters set as follows.

For the Lymphoma problem; $tan\beta$ = 0.5, w_a = 0.5, and w_b = 0.5 respectively. The angle parameter $tan\beta$ is preferred to be small so that it does not disturb the training process too much. However, a few trials of values in [0.0, 0.2] would cause the network to overfit the training data, so a larger value had to be used. w_a and w_b depend upon the initial values of the weights. The learning algorithm was not particularly sensitive to the exact value chosen

For the GCM problem; $tan\beta$ = 0.2, w_a = 0.5, and w_b = 0.5. The learning algorithm was not particularly sensitive to the exact value chosen for the angle parameter $tan\beta$. w_a and w_b were preferred to be large so that the weights were pushed quickly towards the origin before convergence occurred. In that way the algorithm could search more thoroughly for a solution with small weight values.

8.4.2.1 The lymphoma problem

The performance of iTPA was evaluated for the lymphoma dataset. The results are presented and compared with other classification methods found in the literature, namely SVM-OVO [76]. All the results are tabulated in Table 8.3. It was found that iTPA performed well when fewer genes were selected. However, the classification accuracy tended to decline with increasing number of genes. With 400 selected genes, misclassifications were due to one sample, DLBCL-0009. This sample tended to be classified as an FL case, perhaps reflecting tissue sampling. The FL cases were generally harder to classify. This was due to the larger classes (DLBCL, and B-CLL) being chosen in preference to the smaller classes (FL). Clearly as the number of genes was increased, less important genes perhaps not relevant to cancer distinction have added noise to the training data. These genes will compromise classification accuracy and increase the computational burden. This suggests that the real issue to be addressed is the selection of marker genes. A better choice for the number of genes might be achieved by imposing a cut-off on BW [72]

Genes	iTPA OVA	SVM OVO
10	100.0	98.3
20	99.6	99.7
50	100.0	100.0
100	100.0	100.0
200	100.0	100.0
400	99.2	100.0
800	98.3	100.0

Table 8.3. Classification accuracy on the Lymphoma dataset for different algorithms

Convergence occurred rapidly for smaller gene numbers, typically within 2 epochs, but declined slightly for the largest gene number. This result is to be expected as there are more patterns to be learned. The results are comparable with SVM OVO which also achieves high classification accuracies across the range of selected genes.

8.4.2.2 The GCM problem

The performance of iTPA was evaluated for the GCM benchmark dataset. The results are presented and compared with other classification methods found in the literature; SANN [77], and FGAP-RBF [110]. All the results are presented in Table 8.4. It was found that iTPA out performs SANN and FGAP-RBF for each selected gene number. The most significant gains were made when fewer genes were selected. The poor performance of FGAP-RBF on fewer gene numbers may be due to the decoupling effect of the DEKF method used for parameter selection which undermines model accuracy [110].

Genes	iTPA OVA	FGAP RBF	SANN
14	81.2	65.5	68.6
28	85.8	69.1	71.5
42	86.1	75.2	72.9
56	88.8	79.4	79.2
70	89.7	80.3	76.4
84	89.1	82.1	80.6
98	88.8	82.4	77.1

Table 8.4. Classification accuracy (%) on the GCM dataset for different algorithms

Convergence occurs rapidly, typically within 15 epochs. The convergence speed improves with more genes selected. Runtimes take approximately 5 minutes on a Pentium IV (2.67 GHz). The situation with SANN is far worse. SANN requires up to 1000 epochs to finish the training process. Here runtimes can take up to two hours. FGAP-RBF converges in one epoch. Runtimes take only a few seconds, making this algorithm the most efficient with iTPA a good second best. No doubt the poor performance of SANN results from the network complexity. In SANN there are one ANN and 91 SANN to be trained, each ANN consisting of 10 hidden units. Networks are organised into ensembles of 5 modules making a total of 4600 units to be trained.

The classification accuracy of iTPA on each individual tumour class was investigated. The dataset used was GCM with the most significant 84 genes selected as in [8]. The classification accuracy is taken as the number of hits on the test data using 5-fold cross validation. The results are displayed in Fig. 8.1. It was found that iTPA performed equally well in most tumour classes. The tumour classes with the largest samples were all classified correctly; Lymphoma, Leukaemia, and CNS. The number of misclassifications in the other tumour classes was less than 25% of the sample size (e.g. 1 in 4) except in Breast, Bladder and Ovary where it was worse. SANN performed the worst in most tumour classes, showing distinct preference for some classes than others. Classification accuracies for the different algorithms were particularly poor in the Ovary class making Ovary a difficult class to predict.

Table 8.5 shows the confusion matrix for iTPA on the GCM problem for each individual tumour class. The elements on the leading diagonal give the percentage of correct classifications whilst those in the off-diagonal positions give the percentage of misclassifications. It can be seen that the samples in the largest classes were all correctly classified. This shows that increasing the sample size will improve classification accuracy. The classification accuracy for Colorectal and Mesothelioma was also high. Regarding misclassifications, it can be seen that the worst class was Ovary with iTPA showing a distinct preference for Breast, Uterus, and Bladder. The classification accuracy for Bladder was also quite low with iTPA showing a distinct preference for Melanoma and Ovary

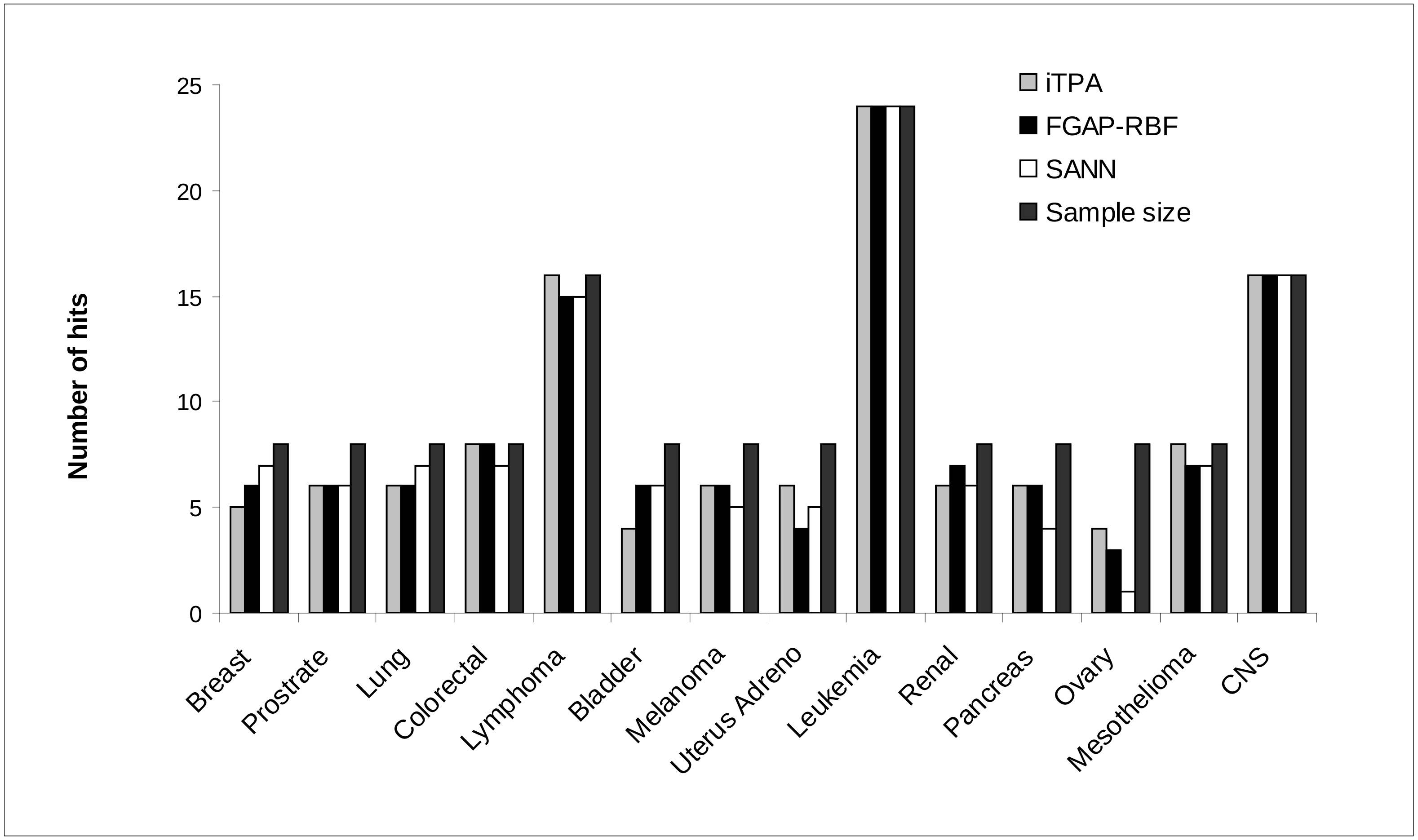


Fig 8.1. Comparison of classification accuracy for different categories on the GCM dataset; breast, prostrate, lung, colorectal, lymphoma, bladder, melanoma, uterus adreno, leukaemia, renal, pancreas, ovary, mesothelioma, and cns

	BR(%)	PR(%)	LU(%)	CO(%)	LY(%)	BL(%)	ML(%)	UT(%)	LE(%)	RE(%)	PA(%)	OV(%)	ME(%)	CNS(%)
BR	65	10	20	0	0	0	0	0	0	15	5	10	5	0
PR	5	70	0	0	0	0	5	0	0	0	0	0	0	0
LU	5	20	70	0	0	5	0	0	0	0	5	0	0	0
CO	0	0	0	95	0	5	0	0	0	0	0	0	0	0
LY	0	0	5	0	100	0	0	0	0	0	0	0	0	0
BL	0	0	0	0	0	60	5	0	0	5	5	10	0	0
ML	5	0	5	0	0	15	75	10	0	0	0	0	5	0
UT	5	0	0	0	0	0	10	75	0	0	0	15	0	0
LE	0	0	0	0	0	0	0	0	98	0	0	5	0	0
RE	5	0	0	0	0	0	0	0	0	65	5	10	0	0
PA	5	0	0	0	0	0	0	5	2	0	80	0	0	0
OV	5	0	0	0	0	10	0	10	0	15	0	45	0	0
ME	0	0	0	5	0	5	0	0	0	0	0	0	90	0
CNS	0	0	0	0	0	0	5	0	0	0	0	5	0	100

Table 8.5. Confusion matrix obtained for iTPA on the GCM problem. The elements on the leading diagonal represent percentage correct classifications, and the elements on the off-diagonal positions percentage of misclassifications

8.4.3 The GN-TPA algorithm

In our simulations, all the input attributes were normalised to remove any bias in the mean values. 1-of-*c* coding was used to label each class category. The number of outputs is equal to the number of class categories. The GN-TPA algorithm was implemented using an SVD operation. GN-TPA requires two parameters to set manually. Firstly, a gain parameter λ was introduced into the activation function of the hidden units, which decides the flatness of the output. According to Zhang et al [76,110], a flatter activation function gives better generalization on problems where the input data is sparsely distributed. Observation of a few trials showed that small values of λ caused a great improvement in generalization. Secondly, the number of hidden units *N* also has to be specified. Large values of *N* close to the input dimension of the sample data would cause gross overfitting on the training set. The exact value of λ and *N* was determined by grid search; $N \in$ {5, 10, 15, …, 100}, and $\lambda \in$ {1.0, 0.1, 0.01, 0.001}. Finally, the input weights and hidden unit biases were set to random values in the range [-1,1]

8.4.3.1 The lymphoma problem

The performance of the GN-TPA algorithm was evaluated for the Lymphoma benchmark dataset. The results are presented and compared with other classification methods found in the literature; ELM and SVM-OVO in [76]. All the results are tabulated in Table 8.6. It was found that GN-TPA achieved very high classification accuracies, except when the number of selected genes was 800. With 800 genes, misclassifications were due to samples in the smaller FL class being classified as DLBCL cases. Once again, one particular DLBCL sample, DLBCL-0009, tended to be misclassified as an FL case, perhaps reflecting tissue sampling. It is clear that less important genes not relevant to cancer distinction have added noise to the training data. These genes will compromise classification accuracy and increase the computational burden. This is probably due to the BW criteria being unable to identify genes that discriminate between all classes. It is also possible that the initialization method used has rendered some genes useless to cancer distinction. This would suggest further research is required on different initialization techniques and smarter gene selection methods.

Genes	GN-TPA	ELM	SVM OVO
10	100.0	98.3	100.0
20	100.0	99.7	99.2
50	100.0	100.0	99.2
100	100.0	100.0	100.0
200	100.0	100.0	100.0
400	100.0	100.0	100.0
800	99.2	100.0	100.0

Table 8.6. Classification accuracy on the Lymphoma dataset for different algorithms

Convergence occurs rapidly in one epoch. Runtimes take only a few seconds on a Pentium IV (2.67 GHz). The situation with ELM and SVM OVO is the same. Runtimes take only a few seconds for a C++ implementation of SVM and a Matlab implementation of ELM on a similar platform.

Genes	10	20	50	100	200	400	800
N	12	6	8	11	21	10	9
λ	1.0	0.1	0.1	0.001	0.001	0.001	0.001

Table 8.7. Optimum values for the parameters of GN-TPA for each selected gene number

8.4.3.2 The GCM problem

The performance of GN-TPA was evaluated for the GCM dataset. The results are presented and compared with other classification methods found in the literature; SVM-OVO and ELM [76]. The most significant 14, 28, 42, 56, 70, 84, and 98 genes selected as in [74]. Different combinations of (N,λ) were used for each gene number. The maximum number of hidden units was set to 100 as there are only 111 samples used at any time in the training data. The results for the best (N,λ) are recorded in Table 8.8. The optimum network size and gain parameter for each selected gene number is given in Table 8.9. It was found that the classification accuracy of the GN-TPA algorithm grows with the number of genes selected. The results are better than SVM OVO, which is the best SVM classifier, but slightly worse than ELM (i.e. typically < 1%). The inferior performance of the GN-TPA algorithm relative to ELM across the gene numbers may be due to the preparation of the input data, which was different in [76]. In the present study, all the input attributes were normalized, whereas in [76] they were scaled in the range [0,1]. Scaling the input values is equivalent to scaling the columns in the hidden layer output matrix. Pre-conditioning strategies such as scaling, reordering and shifting are known to affect the condition number of linear equations [130]

Genes	GN-TPA	ELM	SVM OVO
14	74.5	74.3	70.2
28	77.6	78.5	74.5
42	79.7	80.6	75.1
56	80.9	81.9	75.7
70	80.9	83.4	77.9
84	83.6	84.1	77.9
98	82.6	83.4	79.2

Table 8.8. Classification accuracy on the GCM dataset for different algorithms

Genes	14	28	42	56	70	84	98
N	15	25	25	35	40	40	45
λ	0.1	0.1	0.1	0.1	0.1	0.001	0.001

Table 8.9. Optimum values for the parameters of GN-TPA for each selected gene number

In the second test the classification accuracy of the GN-TPA for each individual tumour category was investigated. The dataset used was GCM with the most significant 84 genes selected. The results are illustrated in Fig 8.2. It was found that the GN-TPA performed equally well in most tumour categories. The tumour classes with the most samples were classified correctly, namely lymphoma, leukaemia, and CNS. For lung, colorectal, bladder, melanoma, uterus, renal, pancreas, and mesothelioma, misclassifications were less than 25% (i.e. 2 in 8). For bladder, prostrate and ovary, misclassifications were less than 50% (i.e. 1 in 2). These results differ to ELM in colorectal, bladder, melanoma and pancreas, which were better than expected, and in prostrate, ovary, and mesothelioma, which were slightly worse. Clearly an improvement in classification accuracy in one class has been paid for by more misclassifications in another class

Table 8.10 shows the confusion matrix for GN-TPA on the GCM problem. It can be seen that the samples in the largest classes were all correctly classified. This shows that increasing the sample size will improve classification accuracy. The classification accuracy for Colorectal and Mesothelioma was also high. Regarding misclassifications, it can be seen that Ovary was the worst class with GN-TPA showing a distinct preference for Bladder, Uterus, Lymphoma and Renal

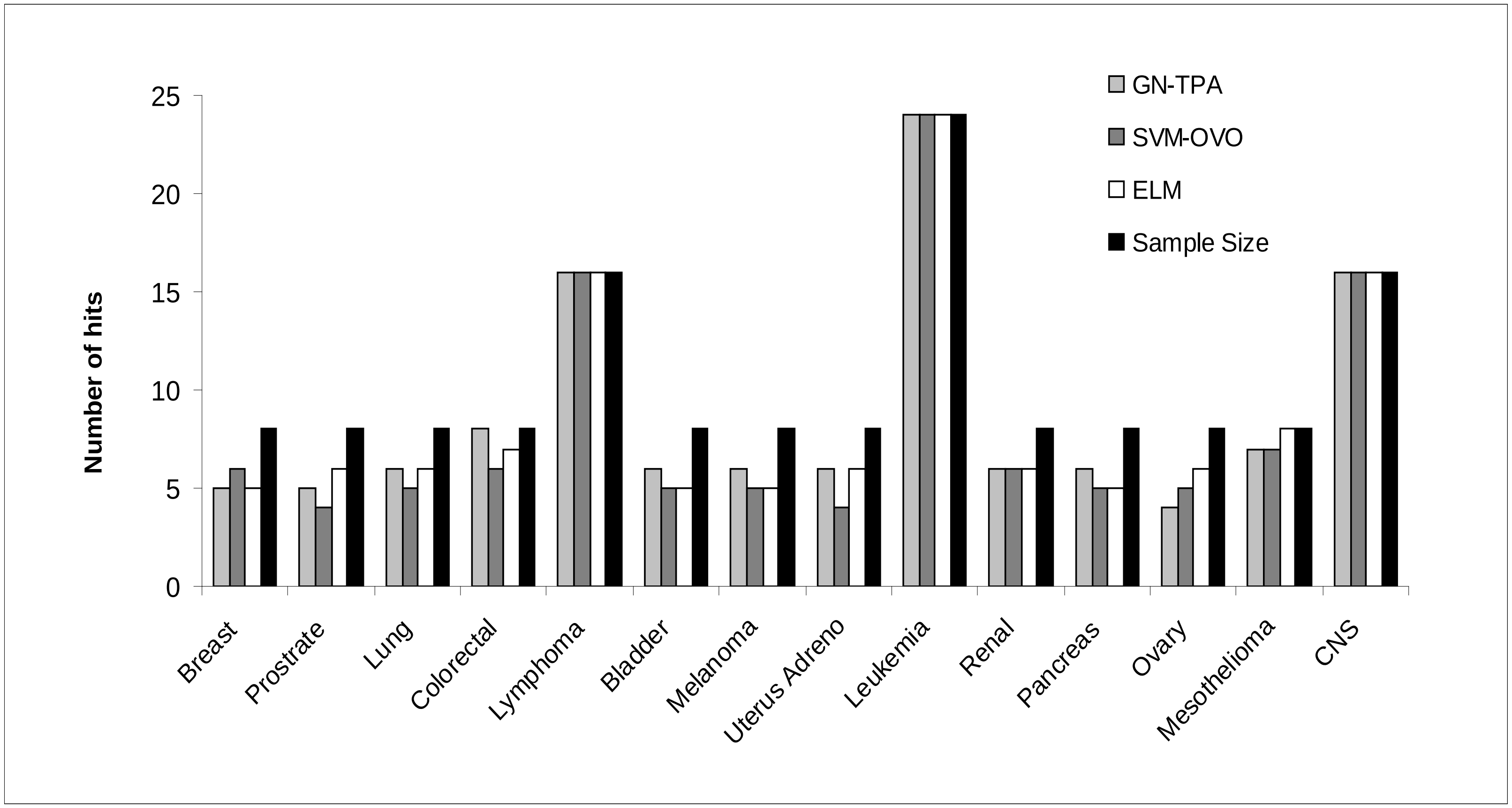


Fig 8.2. Comparison of classification accuracy for different categories on the GCM dataset; breast, prostrate, lung, colorectal, lymphoma, bladder, melanoma, uterus adreno, leukaemia, renal, pancreas, ovary, mesothelioma, and cns

	BR(%)	PR(%)	LU(%)	CO(%)	LY(%)	BL(%)	ML(%)	UT(%)	LE(%)	RE(%)	PA(%)	OV(%)	ME(%)	CNS(%)
BR	65	15	0	0	0	5	5	0	0	0	5	5	0	0
PR	0	70	0	0	0	0	0	0	0	0	0	0	0	0
LU	5	15	85	0	0	0	0	0	0	5	0	0	0	0
CO	0	0	0	90	0	0	5	0	0	0	0	0	0	0
LY	5	0	0	0	100	0	0	0	0	0	0	10	0	0
BL	5	0	15	0	0	70	0	0	0	0	5	10	0	0
ML	0	0	0	0	0	10	70	5	0	0	20	5	0	0
UT	0	0	0	0	0	0	15	80	0	5	0	15	0	0
LE	5	0	0	0	0	0	0	0	100	10	0	0	0	3
RE	5	0	0	0	0	0	5	0	0	70	0	10	0	0
PA	5	0	0	0	0	0	0	0	0	0	65	5	0	0
OV	0	0	0	0	0	10	0	15	0	10	5	40	0	0
ME	5	0	0	10	0	5	0	0	0	0	0	0	100	0
CNS	0	0	0	0	0	0	0	0	0	0	0	0	0	97

Table 8.10. Confusion matrix obtained for GN-TPA on the GCM problem. The elements on the leading diagonal represent percentage correct classifications, and the elements on the off-diagonal positions percentage of misclassifications

8.5 Comparison of the different algorithms

We have already seen that the GN-TPA algorithm creates larger network structures compared with other network building techniques. However, the number of adjustable weights n in the network is small; only the output weights are trained. This raises the following question; "Does the tangent plane algorithm operating in batch mode actually produce better generalization results than in sequential mode?" In order to answer this question, we carry out a statistical hypothesis test on the simulation results for the Lymphoma dataset in the last section. The most significant genes were selected as in [74]. The values tested were the mean square error on the test set. The statistical hypothesis test used was a two-tailed *t*-test with Cochran-Cox approximation for the case of unequal variances. The results were tested for normality using a Kolmogorov-Smirnov (KS) test. For normality, the p-value associated with KS test should exceed 0.15. A small number of tests (overall 14%) deviate substantially from a normal distribution, which under-estimates the significance of differences.

8.5.1 Network initialization

1-of-*c* encoding was used for each class label. The number of output units is equal to the number of class categories. The output activation with the highest activity indicates class membership of the corresponding input. The parameters for GN-TPA were set as follows. The number of hidden units was fixed at N = 10, which is the same as the number of hidden units per module in the sequential learning algorithm, iTPA-OVA. A gain parameter λ *w*as introduced into the activation function giving a flatter output, which gives better generalization results when the ratio of the input dimension to the number of samples is high. Different values of the gain parameter λ were used for each selected gene number. The exact values for λ were determined by grid search: $\lambda \in \{1.0, 0.1, 0.001\}$. The parameters for iTPA were set as follows;

$tan\,\beta$ = 0.5, w_a = 0.5, and w_b = 0.5. *tan β* is preferred to be large so that the network does not overfit the training data. w_a and w_b depend upon the initial values of the weights. The learning algorithm was not particularly sensitive to the exact value chosen for these parameters

10 trials were carried out with the mean classification accuracy on the test set recorded. Multi-fold cross-validation was used to test the network as the number of training samples per class is relatively small. The mix of training and test data were kept fixed in the ratio 4:1 as in Linder et al [77]. Different shuffles between training and test data were used according to each trial. Early stopping was used to terminate network training, which helps to prevent overtraining.

8.5.2 Results and discussion

We performed a statistical hypothesis test to answer our question; "Does GN-TPA generalize better than iTPA-OVA in small networks?" The results are tabulated in Table 8.11. Dashes mean differences that are not significant at the 5% level i.e. the probability that the differences are purely accidental. Parenthesis means that the results are not precise because one test deviated from a normal distribution. Other entries indicate the superior algorithm (e.g. GN-TPA - E, and iTPA-OVA - M), and the value of the *t* statistic. For DLBCL: 7 times no significant difference. For FL: 4 times no significant difference, 3 times iTPA-OVA better than GN-TPA. For B-CLL: 7 times iTPA-OVA better than GN-TPA.

iTPA-OVA significantly out-performs GN-TPA with one exception, the DLBCL dataset. This dataset contains the most training examples making it the easiest dataset to learn. Otherwise, the results suggest that iTPA-OVA is the superior algorithm. This result is not surprising as there are significantly more adjustable weights per module in a SLFN than in the corresponding ELM network. Fig. 8.3, 8.4, and 8.5 show the test curves of GN-TPA and iTPA-OVA for each tumour class. The results suggest that although different algorithms are used, the trends of the curves are very similar for each tumour class.

Genes	10	20	50	100	200	400	800
DLBCL	(0.64)	(0.32)	-	-	-	-	-
FL	(0.10)	-	-	M 3.24	M 4.19	M 4.42	-
B-CLL	M 7.00	M 6.56	M 4.02	M 2.67	M 5.64	M 4.88	M 3.35

Table 8.11. Results of a t-test comparing the mean test set errors for the two algorithms. Dashes mean that differences are not significant at the 5% level, parenthesis mean that the results are imprecise. Other entries indicate the superior algorithm: iTPA-OVA – M, GN-TPA – E

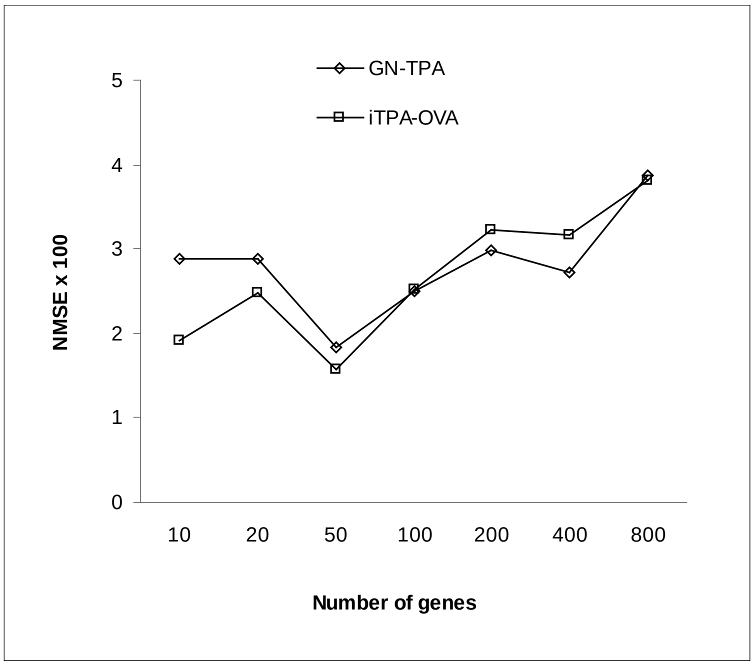


Fig 8.3. Comparison of classification errors on the DLBCL tumour class for each selected gene number

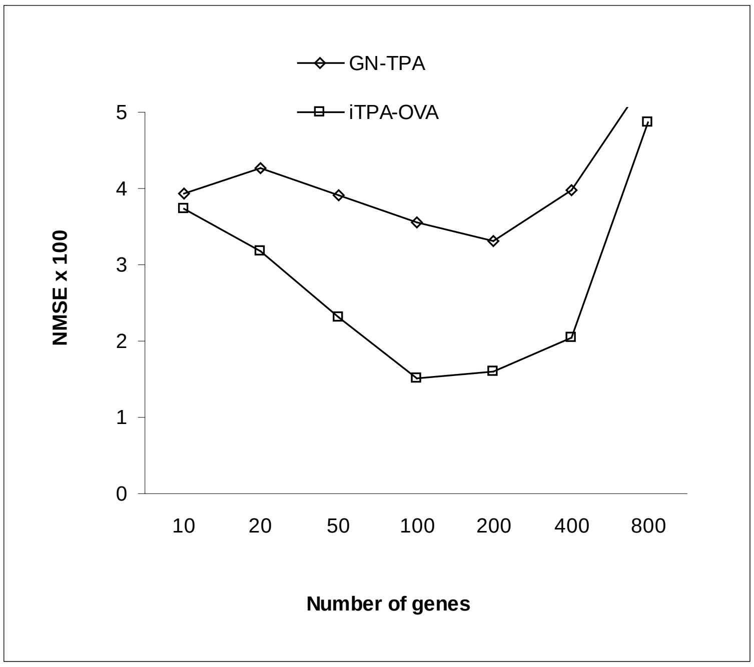


Fig 8.4. Comparison of classification errors on FL tumour class for each selected gene number

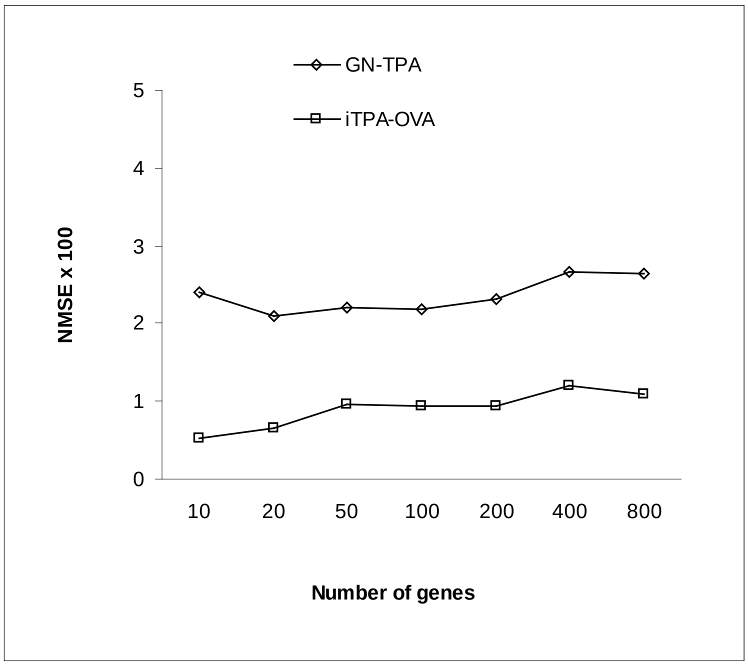


Fig 8.5. Comparison of classification errors on B-CLL tumour class for each selected gene number

8.6 Summary

Artificial neural networks have been well established for their unique capability to represent any non-linear input-output mapping. Compared with SVM, neural networks try to map the input data directly into separate classes in feature space, while SVM tries to separate the data into two classes. However, studies show that the classification accuracy of neural networks drops quickly as the number of classes increases. Various combinatory schemes have been proposed for combining neural networks, but these schemes produce larger and more complex network structures, longer runtimes, and heavy computational burdens. Therefore there is a need for fast and efficient algorithms to train them. In this chapter, two neural network algorithms are investigated for multi-category classification using gene expression data, namely iTPA and GN-TPA. Both algorithms are fast and efficient. Studies show that they have good generalization properties on benchmark neural network tasks

The iTPA algorithm is attempted for multi-category classification and compared with the latest results in the literature. The method adopted here referred to as iTPA-OVA uses a modular network. Within the modular network architecture, each module represents a single hidden layered ANN. Each ANN is trained to discriminate one particular class from all others. The benchmark datasets used were GCM [118] and lymphoma [119]. The results show that iTPA-OVA gives better overall classification accuracy relative to the FGAP-RBF and SANN algorithms, and comparable classification accuracy relative to SVM-OVO. The classification accuracy in individual tumour categories is better than SANN, and at least as good as FGAP-RBF. iTPA-OVA does not favour one class over another, but gives a good overall balance among the various classes

The GN-TPA algorithm is compared with the latest results in the literature on two benchmark multi-category cancer classification problems, the GCM dataset [118] and the lymphoma dataset [119]. The results show that the GN-TPA gives comparable classification accuracy relative to the best SVM classifier, SVM OVO, and slightly worse classification accuracy relative to the ELM algorithm. The performance of ELM depends to a large extent on the generation of a set of inputs to the final layer units. The larger the subspace spanned by these vectors, the better the classification accuracy. Unfortunately these input vectors are not optimised to minimise the training set error, but generated randomly. Thus, any improvement in classification accuracy is gained by utilizing more hidden units.

The classification accuracy of iTPA-OVA and GN-TPA are sensitive to the number of genes required for accurate cancer classification. Experimental results show that the overall classification accuracy drops when the number of genes approaches 1,000. This is probably due to the method of gene selection used for accurate cancer classification. Many genes are irrelevant to cancer classification. These genes increase computational complexity and introduce noise into the training data. This problem is exacerbated by the relatively small sample sizes. It is recognised that the higher the ratio of the training samples to the number of free parameters in the network, the better the generalization will be. Micro-array data typically comprises very few samples (< 30) of many thousands of genes. This suggests that smarter gene selection methods are needed to identify marker genes

Chapter 9

CONCLUSIONS AND FUTURE WORK

9.1 Conclusions

In this thesis, we have investigated and developed sequential and batch learning algorithms based upon the tangent plane algorithm for artificial neural networks with applications from the bioinformatics area.

In the first part of this thesis, the performance of sequential and batch learning algorithms based on the tangent plane algorithm are investigated

- The convergence behaviour of the tangent plane algorithm is investigated and compared with the gradient descent back-propagation algorithm. The results indicate that the tangent plane algorithm gives fast convergence relative to the back-propagation algorithm, except in small networks where the convergence speed was slower and there were more failed trials. The principal strength of the tangent plane algorithm is that it does not require manually tuning a learning rate parameter, but instead automatically adjusts the learning rate to give the correct step size.

- The stability of the tangent plane algorithm is investigated and compared with the gradient descent back-propagation algorithm using two different types of inexact data. First, varying amounts of random “white” noise were added to the teaching values of the training data so that they vary from pattern presentation to pattern. Second, single items of rogue data were occasionally fed into the network during training. The results indicate that the tangent plane algorithm is a relatively robust method of training neural networks. However, the big weight updates caused by noisy data frequently disrupt the training when the level of noise is high resulting in longer recovery times and more failed trails

- The generalization performance of the second tangent plane algorithm is investigated and compared with the gradient descent back-propagation algorithm. The results indicate that the second tangent plane algorithm gives improved generalization relative to the back-propagation algorithm in some task and comparable generalization in the others, except in the smallest networks where it was the same. Generalization was found to be independent of network size. This is the principal strength of the second tangent plane algorithm.

- A number of limitations leading to slightly inferior performance are identified in the tangent plane algorithm

  - The convergence speed of the tangent plane algorithm is no better than the steepest descent back-propagation method in small parsimonious networks where generalization is known to be best. The probability of a set of normals to constraint surfaces being nearly linearly dependent will be much higher in small networks with few adjustable weights leading to slow convergence.
  - The tangent plane algorithm frequently fails to converge when the training set is inexact or fuzzy. The convergence of the algorithm was frequently disrupted by big weight updates in response to rogue data patterns that were occasionally fed into the network. Further the

algorithm would not converge to a compromise solution when the teaching values were corrupted by a small amount of artificial noise. In this case the algorithm would continue to oscillate between approximate solutions

  - The distribution of weight importance coefficients in networks trained by the second tangent plane algorithm was investigated. The results indicate that the tangent plane algorithm does not produce a separation of active and inactive weights as expected, but the weights continue to grow from small initial values producing large network structures with wide distributions of weight values.

- A new sequential learning algorithm referred to as TPA-RTRL is developed for fully recurrent neural networks. It is shown that recycling information around the network can improve the stability of the tangent plane algorithm when the training set contains a small percentage of rogue data. This is because the network learns to predict the correct response to an item of data in advance of receiving the input. The new algorithm is investigated and compared with the original GD-RTRL algorithm on two sequence recognition tasks. The results show that the new algorithm learns to predict temporal sequences faster than the original GD-RTRL algorithm. The results also show that the new algorithm recovers quickly after the network is presented with spurious items of rogue data. However the runtimes are significantly longer and increase with the size of the network.

- A new sequential learning algorithm referred to as iTPA is developed to improve the generalization performance of the second tangent plane algorithm. In the new algorithm, an additional term is included that pushes the weights along tangent planes in a direction that encourages weight elimination. The new algorithm is investigated and compared with the original algorithm on two real world neural network problems. The results show that the new algorithm gives improved generalization relative to the original algorithm in some problems, and comparable generalization in others. The results also show that the new algorithm retains the fast convergence speed of the original method. Including a small amount of random movement along tangent planes often helps the network break out of local minima that can slow down the convergence.

- A new batch learning algorithm referred to as GN-TPA is developed to overcome the problem of slow convergence in small parsimonious networks where generalization is known to be best.

  - In the new algorithm, a system of tangent plane equations is constructed and solved using the method of least squares, which is an optimization technique used to find an approximate solution to a system of equations where no exact solution exists. The new algorithm is investigated and compared with the Rprop algorithm on two neural network benchmark problems. The results show that the new algorithm is very fast. However some trials stalled due to oscillatory behaviour, showing that the step size was too large and that a good initial guess is needed for fast convergence to occur. Also, some trials failed due to rank deficiency in the Jacobian matrix.
  - Two modifications are suggested to overcome the difficulties with the new batch algorithm, viz. convergence to local minima, and the computational cost of performing an SVD operation. Firstly, a novel

network architecture called an extreme learning machine (ELM) is utilised, which is a single hidden layer feed-forward neural network (SLFN) with input weights chosen arbitrarily. Secondly, the tangent plane normal equations are solved iteratively by using an orthogonal transformation. This improves the computational efficiency of the algorithm and gives improved generalization when combined with early stopping. Comparative tests were performed using the new GN-TPA algorithm, the cascade algorithm, and the orthogonal sequential training technique. The results show that the GN-TPA gives improved generalization on some problems relative to the cascade algorithm and the orthogonal training technique. There was little evidence of overtraining in any of the networks trained by the GN-TPA algorithm. The principal weakness of the GN-TPA algorithm is that it builds large networks. However, this is only a minor issue as generalization performance does not appear to deteriorate with increasing network size

In the later part of this thesis, a significant contribution is made to multi-category classification problems in the bioinformatics area

- A new sequential learning algorithm referred to as iTPA-OVA is investigated and compared with SANN and FGAP-RBF on two cancer classification problems using gene expression data. In order to improve classification accuracy several simple ANNs are combined using a one-versus-all (OVA) combinatory scheme. The results show that the new iTPA-OVA algorithm gives better overall accuracy across a range of selected marker genes relative to SANN, and FGAP-RBF, and an overall accuracy at least as good as the best SVM classifier. Classification accuracy in individual tumour categories is better than SANN, and at least as good as FGAP-RBF. The new sequential algorithm does not favour one class over another, but gives a good overall balance among the various classes. Although the new algorithm gives better classification accuracy relative to the best classifier, FGAP-RBF, the training times are slightly longer, so an improvement in one process must be paid for by a deterioration in another

- The new batch leaning algorithm GN-TPA is investigated and compared with SVM-OVO and the original ELM algorithm on two cancer classification problems using gene expression data. In order to improve classification accuracy a gain parameter was introduced into the hidden unit activations giving a flatter output. The results show that the new batch algorithm gives comparable overall accuracy across a range of selected marker genes relative to the best SVM classifier, SVM-OVO, and slightly worse performance relative to the original ELM algorithm. The classification accuracy of the new algorithm in individual tumour categories is at least as good as the best SVM classifier. The new batch algorithm does not favour one class over other classes.

- Study results show that the new GN-TPA algorithm constructs slightly larger networks than other popular network building techniques. This raises the following question, "Does the new batch tangent plane algorithm generalize better than the new sequential tangent plane algorithm is smaller networks?" In order to answer this question, a statistical hypothesis test is carried out using the gene expression data. The results show that there is a significant difference at the 5% level. The new sequential algorithm gives better classification accuracy in smaller tumour classes across a range of selected marker genes. Both algorithms give comparable accuracy on the largest tumour class.

- The classification accuracy of the new sequential and batch learning algorithms tends to drop with more selected marker genes (> 100). This is probably due to the inability of the gene selection method to identify genes that discriminate between different classes rather than the performance of the individual algorithms

9.2 Recommendations for future work

Possible areas for future work that emerge from this thesis include

- Study results show that the new TPA-RTRL algorithm takes more time to finish the learning process due to the high computational complexity of the algorithm. Therefore methods to reduce the computational complexity of the algorithm can be investigated such as fixing some of the recurrent connections in MERTRL [93,94] or fixing a preset number of weights that are randomly chosen during each time step.

- Speech enhancement remains a developing area in neural network research. Several researchers have used neural networks for speech enhancement [85,86]. Conventional ANNs cannot easily model the temporal behaviour of speech signals by using a windowed input [62]. One way to address this issue is using recurrent neural networks to deal with the varying length of speech. Therefore the new TPA-RTRL algorithm will be further developed for speech enhancement applications

- Traditionally the Extreme Learning Machine requires a high number of hidden units which may lead to ill conditioning on some problems. Fei Han et al [130] have applied a modified particle swarm optimization method to select input weights and biases of hidden layer units giving better generalization and condition than other ELM methods. Particle swarm optimization methods are known to converge rapidly in the initial stages of training around a global minimum [131]. However this strategy still requires the number of hidden units to be selected a-priori. This suggests there is scope to further modify the sequential GN-TPA algorithm for the extreme learning machine using particle swarm optimization methods so that the most favourable new hidden units are selected for insertion into the network

- Gene selection is of vital importance in molecular cancer classification using high dimensional gene expression data. Carrying out feature selection reduces the curse of the dimensionality problem and improves prediction accuracy. Wang et al [132] have compared established feature selection methods using eight gene expression datasets; Colon tumour, CNS tumour, DLBCL, Leukaemia 1, Lung cancer, Prostrate cancer, Leukaemia 2, and breast cancer. The classification methods used were a probabilistic classifier (NB), K-NN and SVM. Therefore a further development of this work is to compare the performance of ANN trained using the GN-TPA algorithm to benchmark the performance of neural network classifiers on the microarray datasets used in this study

- Cancer cells possess traits similar to normal stem cells. It is unclear however whether these similarities reflect the activity of common molecular pathways. Ittah Ben-Porath et al [133] all have analysed the enrichment patterns of genes associated with embryonic stem (ES) cell identity in the expression levels of various human cancer types. Recently developed gene expression analysis methods [134] were used to determine whether the expression signatures that define human ES cell identity are also active in human tumours. The results indicate a novel link between ES cell identify and microscopic histopathelogical traits of tumours. Therefore a further development of this work is to use neural network methods to benchmark the performance of neural networks in determining the link between human ES cells and tumour cells

BIBLIOGRAPHY

01 Rumelhart, D.E. and McClelland, J.L. *Parallel distributed processing: Explorations in the microstructure of cognition*. Cambridge, MA: MIT Press, 1986.

02 Lee, C.W. Learning in neural networks by using tangent planes to constraint surfaces. *Neural Networks*, vol. 6. pp. 385-392, 1993.

03 Williams, R.J, and Zipser, D. A learning algorithm for continually running recurrent neural networks. *Neural Computation*, vol.1, no. 2, pp.270-280, 1989

04 Jordan, M.I. Attractor dynamics and parallelism in a connectionist sequential machine. *Proceedings of the eighth annual conference of the cognitive science society*, 531-546, 1986

05 Pineda, F.J. Dynamics and architecture for neural computation. *Journal of complexity,* 4, 216-245, 1988

06 Haykin, S. Neural networks: a comprehensive foundation. New Jersey : Prentice Hall, 1999.

07 Fahlman, S. and Lebiere, C. The cascade correlation learning architecture. *Advances in neural information processing systems*, vol. 2. pp. 524-532, 1990.

08 Prechelt, L. Investigation of the CasCor family of learning algorithms. *Neural Networks,* vol.10, no. 5, pp. 885-896, 1997.

09 Lahnajärvi, J.J.T, Lehtokangas, M.I. and Saarinen, J.P.P. Evaluation of constructive neural networks with cascade architectures. *Neurocomputing,* vol. 48, pp. 573-607, 2002.

10 LeCun, Y.L, Denker, J.S., and Solla, S.A. Optimal brain damage. *Advances in neural information processing systems,* vol.2, pp. 598-605, 1990.

11 Hassibi, B. and Stork, D.G. Second order derivatives for network pruning. *Advances in neural information processing systems,* vol.5, pp. 164-171, 1993.

12 Krogh, A. and Hertz, J.A. A simple weight decay can improve generalization. *Advances in neural information processing systems,* vol.4, pp. 950-957, 1992.

13 Nowlan, S.J., and Hinton, G.E. Simplifying neural networks by soft weight-sharing*. Neural Computation*, 4(4) : 473-493, 1992.

14 Lee, C.W. Training feedforward neural networks: an algorithm giving improved generalization. *Neural Networks*, vol. 10. pp. 61-68, 1997.

15 Lapedes, A. and Faber, R. How neural networks work. Los Alamos National Laboratory, NM 87545.

16 Lehtokangas, M. Fast initialization for fast cascade correlation learning. *IEEE transactions on neural networks*, 10(2) : 410-414, 1999.

17 Lahnajärvi, J.J.T, Lehtokangas, M.I. and Saarinen, J.P.P. Fixed cascade error : a novel constructive neural network for structure learning. Proceedings of the artificial neural networks in engineering conference, Nov 7-10, pp. 25-30, 1999

18 Mendel, J.M. and McLaren, R.W. Reinforcement-learning control and pattern recognition systems. *Adaptive learning and recognition systems: theory and applications,* vol. 66., pp. 287-318, 1970.

19 Solla, S.A, and Levin, E., and Fleisher, M. Accelerated learning in layered neural networks. *Complex Systems*, **2**, pp. 625-639, 1988

20 Jacobs, R.A. Increased rates of convergence through learning rate adaptation methods*. Neural Networks*, vol. 1, pp. 295-307, 1988.

21 Chan, L. W., and Fallside, F. An adaptive training algorithm for back-propagation networks. *Computer speech and language,* 2: 205-218, 1987.

22 Zhang, J and Morris, A.J. A sequential learning approach for single hidden layer neural networks. *Neural Networks,* vol. 11, no. 1, pp. 65-80, 1998.

23 Haffner, P., Waibel, A., Sawai, H. and Shikano, K. Fast back-propagation learning methods for neural networks in speech. *ATR Interpreting telephony research laboratories,* TR-1-0058, 1988.

24 Darken, C. and Moody, J. Note on learning rate schedules for stochastic optimization. *Neural information processing systems*, pp. 832-838, 1991

25 Saloman, R. Improved convergence ate of back-propagation with dynamic adaptation of the learning rate. Lecture notes in computer Science, PPSN1, Dortmund, pages 269-273, 1990

26 Hertz, J., Krogh, A., and Palmer, R.G. Introduction to the theory of neural computation. Addison-Wesley, ISBN 0-201-51560-1, 1991

27 Schmidhuber, J. Accelerated learning in back-propagation nets. Connectionism in perspective, Elsevier Science Publishers, pp. 439-445, 1989

28 Reidmiler, M, and Braun, H. A direct adaptive method for faster backpropagation learning: The RPROP algorithm. *Proceedings of the IEEE International conference on Neural Networks*, pages 589-591, IEEE Press, 1993.

29 Silva, F.M, and Almeida, L.B. Speeding up back-propagation. *Advanced neural computers*, pp. 151-158, 1990

30 Tollenaere, T. Fast adaptive back-propagation with good scaling properties.

Neural networks, vol. 3, pp. 561-573, 1990

31 Igel, C., and Husken, M. Empirical evaluation of the improved Rprop learning algorithms. *Neurocomputing*, 50(C), pp. 105-123, 2003

32 Fahlman, S. An empirical study of the learning speed in back-propagation networks. Technical Report CMU-CS-88-162, 1988

33 Schiffmann, W., Joost, M., and Werner, R. Comparison of optimized back-propagation algorithms, Proc. of the European Symposium of Artificial Neural Networks, Brussels, pp. 97-104, 1993

34 Prechelt, L. *PROBEN1* – A set of benchmarks and benchmarking rules for neural network training algorithms. Technical report 21/94, Fakultät für Informatik, Universität Kaxlsruhe, Germany. ftp:/pub/papers/techreports/1994/1994-21.ps.gz

35 Press, W., Teukolsky, S., Vetterling, W., Flannery, B. *Numerical recipes in C.* Cambridge University Press, 2nd Edition, 1994

36 Kramer, A.H., Vincentelli, Sanglovanni, A. Efficient parallel learning algorithms for neural networks. *Advances in neural information processing systems I*, pp. 40-48, 1989

37 Battiti, R. First and second order methods of learning: between steepest descent and Newton's method. *Neural Computation,* 4, 141-166, 1992.

38 Gill, P.E., Murray, W., and Wright, M.H. Practical optimization. London : Academic Press, 1981

39 Meyer, R.R. Theoretical and computational aspect of nonlinear regression. *Nonlinear Programming,* pp. 465-486. Academic Press, New York, 1970

40 Levenberg, K. A method for the solution of certain problems in least squares. *Quart. Appl. Math,* vol. 2, pp. 164-168, 1944.

41 Fletcher, R. A modified Marquardt subroutine for nonlinear least squares. Atomic energy establishment report R6799, Harwell, England, 1971

42 Wilamowski, B.M., Kaynak, O., Iplikei, S., Őnder Efe, M. An algorithm for fast convergence in training neural networks. *Proceedings of the international joint conference on neural networks,* 2, 1778-1782, 2001

43 Poggio, T. and Girosi, F. Networks for approximation and learning. *Proceedings of the IEEE* , vol. 78, pp. 1481-1497, 1990

44 Vapnik, V.N., and Chervonenkis, A.Ya. On the uniform convergence of relative frequencies of events to their probabilities. *Theoretical probability and its applications.* vol. 17, pp.264-280, 1971.

45 Chen, S, Cowan, C.F.N and Grant, P.M. Orthogonal least squares learning algorithm for radial basis function networks. *IEEE Transactions on Neural Networks*, vol. 2, no. 2, pp. 302-309, 1991.

46 Platt, J. A resource allocating network for function interpolation. *Neural Computation*, 3(2) : 213-255, 1991

47 Guang-Bin Huang, Saratchandran, P, and Sundarajan, N. An efficient sequential learning algorithm for growing and pruning RBF (GAP-RBF) networks. *IEEE transactions on systems, man and cybernetics, Part B,* 34(6) : 2284-2292, 2004

48 Narenda, K.S. and Parthasaranthy, K. Identification and control of dynamic systems using neural networks. *IEEE Transactions on Neural Networks*, vol. 1, no. 1, pp. 4-27, 1990.

49 Sietsma, J. and Dow, R. Neural network pruning – why and how. *Proc. IEEE int. conf. neural networks* I, 325-333, 1988

50 Mozer, M.C. and Smolensky, P. Skeletonization: A technique for trimming the fat from a network via relevance assessment. Advances in Neural Information Processing, pp. 107-115, 1989.

51 Prechelt, L. Connection pruning with static and adaptive pruning schedules. *Neurocomputing*, Volume 16, Issue 1, pp. 49-61, 1997

52 Finnoff, W, Hergert, F, Zimmermann, H.G. Improving model selection by non-convergent methods. *Neural Networks,* vol.6, 771-783, 1993.

53 Prechelt, L. Automatic early stopping using cross validation: quantifying the criteria. *Neural Networks,* vol.10, no. 5, pp. 885-896, 1997.

54 Schittenkopf, C, Deco, G. and Brauer, W. Two strategies to avoid overfitting in feedforward networks. Neural Networks, vol. 10, no. 3, 1997.

55 Oja, E. Principal components, minor components, and linear networks. *Neural networks* 5, pp. 927-935, 1992

56 Sanger, T.D. Optimal unsupervised learning in a single layered neural network. *Neural networks*, vol. 2, pp. 459-473, 1989

57 Diamantaras, K.I, and Kung, S.Y. Principal component networks: theory and applications, Wiley and Sons, New York, 1996

58 Fiori, S. An experimental comparison of three PCA neural networks. *Neural processing letters*, vol. 11, pp. 209-218, 2000

59 Stone, M. Cross validation choice and assessment of statistical predictions. Royal Statistical Society, B36, pp. 111-113, 1974.

60 Levin, R.I, Lieven, N.A.J. and Lowenberg, M.H. Measuring and improving neural network generalization for model updating. Journal of Sound and Vibration, 238 (3), pp. 401-424, 2000.

61 The UCI Learning Machine Repository WWW site (http://www.ics.uci.edu/~mlearn/MLRepository.html).

62 Sejnowski, T.J. and Rosenberg, C.R. Parallel networks that learn to pronounce English text. *Complex Systems.* 1, 145-168, 1987.

63 Hanock, J.M, and Zvelebil, M.J. *Dictionary of bioinformatics and computational biology*. Wiley, 2004

64 Luscombe, N.M., Greenbaum, D., and Gerstein, M. What is bioinformatics? A proposed definition and overview of the field. *Int medical informatics association*, pages Yearbook, pages 83-99, 2001

65 Benson, D.A., Boguski, M., Lipman, D.J., and Ostel, J. *Nucleic acids research,* 29(1):15-18, 2000

66 Bourne, P.E. and Weissig, H. *Structural Bioinformatics*. Wiley, 2003

67 Brown, P.O. and Botstein, D. Exploring the new world of the genome with dna microarrays. *Nature genetics*, 21:33-37, 1999

68 Lander, E.S. Array of hope. *Nature genetics*, 21:3-4, 1999

69 Chen-Hsiang Yeang, Sridhar Ramaswamy, Pablo Tamayo, Sayan Mukherjee, Ryan Rifkin, Michael Angelo, Michael Reich, Eric Lander, and Todd Golub. Molecular classification of multiple tumor types. *Bioinformatics,* 17:316-322, 2001

70 Golub, T.R., Slonim, P., Tamayo, P., Huard, M., Gaasenbeek, J.P., Mesirov, H., Coller, H., Loh, M.L, Downing, J.R., Caligiuri, C.D., and Lander, E.S. Molecular classification of cancer: class discovery and class prediction by gene expression monitoring. *Science,* 286(5439):531-537, 1999

71 Alon, U., Barkai, D.A, Notterman, K., Gish, K., Ybarra, D.M.S, and Levine, A.J. Broad patterns of gene expression revealed by clustering analysis of tumor and normal colon tissue probed by oligonucleotide arrays. *Proceedings of National academy of Sciences,* 96(12):6745-6750, 1999

72 Dudoit, S., Fridlyand, J., Speed, T.P. Comparison of different discrimination methods for classification of tumours using gene expression data. *Journal of the American statistics association,* 97 (457):77-87, 2002

73 Furey, T., Cristianini, N., Duffy, N., Haussler, D. Support vector machine classification and validation of cancer tissue samples using microarray

expression data. *Bioinformatics,* 16(10):906-914, 2000

74 Ramaswamy, S., Tamayo, P., Golub, T.R. Multiclass cancer diagnosis using tumor gene expression signatures. *Proceedings of National Academy Sciences, USA,* 98(26):15149-15154, 2002

75 Yoonkyung Lee and Cheol-Koo Lee. Classification of multiple cancer types by multi-category support vector machines using gene expression data. *Bioinformatics*, 18:1132-1139, 2003

76 Zhang, R, Guang-Bin Huang, Saratchandran, P, and Sundarajan, N. Multi-category classification using an extreme learning machine for microarray gene expression cancer diagnosis*. IEEE transactions on computational biology and bioinformatics*, Vol. 4, No. 3, pp. 485-495, 2007

77 Linder, R., Dew, D. Sudhoff, H, Theegarten, D, Remberger, K. The subsequent artificial neural network (sann) approach might bring more classificatory power to ann-based dna microarray analysis. *Bioinformatics,* 20(18):3544-3552, 2004

78 Statnikov, A. Aliferis, C.F., Tsamardinos, I., Hardin, D., and Levy, S. A comprehensive evaluation of multi-category classification methods for microarray gene expression cancer diagnosis. *Bioinformatics,* 21(5):631-643, 2005

79 Khan, J., Wei, J.S., Ringner, M., Saal, L.H., Ladanyi, M., Westermann, F., Berthold, F., Schwab, M., Antonescu, C.R., Petersen, C., and Meltzer, P. Classification and diagnosis of cancers using gene expression profiling and artificial neural networks. *Nature Medicine,* 7(6): 673-679, 2001

80 Alizadeh, A., Eisen, M., Davis, E., Chi Ma, Lossos, I.S., Rosenwald, A., Boldrick, J.C., Hajeer Sabet, Truc Tran, Xin Yu, Powell, J.I., Liming Yang, Marti, G.E., Moore, T., Hudson, J., Byrd, J.C., Botstein, D., Brown, P.O., and Staudt, L.M. Distinct types of diffuse large b-cell lymphoma identified by gene expression profiling. Nature, 403(6760) : 503-511, 2000

81 Vogl, T.P., Mangis, J.K., Rigler, A.K., Zink, W.T, and Alkon, D.L. Accelerating the convergence of the back-propagation method. *Biological cybernetics*, 59 : 257-263, 1988

82 Geman, S., Bienenstock, E. and Doursat, R. Neural networks and the bias/variance dilemma*. Neural Computation*, 4 : 1-58, 1992.

83 Bartlett, P.L. The same complexity of pattern classification with neural networks: the size of the weights is more important than the size of the network. *IEEE transactions on information theory*, 44(2) : 42–61, 1998.

84 Hambaba, A. Robust hybrid architecture to signals from manufacturing and machine monitoring. . *Intell. Fuzzy Syst.* 9(1-2), pp. 29-41, 2000

85 Juang, C.F., and Lin, C.T. Noisy speech processing by recurrently adaptive fuzzy filters. *IEEE trans. Fuzzy systems.* 9(1), pp. 139-152, 2001

86 Parveen, S, and Green, P. Speech enhancement with missing data techniques using recurrent neural networks. *IEEE trans. Fuzzy systems.* 1, 733-736, 2004

87 Selvan, S, Srinivasan, R. Recurrent neural network based efficient adaptive filtering technique for removal of ocular artefacts from EEG. *IETE Tech Review*, 17 (1-2), 73-78, 2000

88 Perez-Ortez, J.A., Calera-Rubio, M.L, Forcada, M.L. Online symbolic sequence prediction with discrete time recurrent neural networks. *Proceedings of the International Conference on Artificial Intelligence,* vol. 2130, pp.719-724, 2001

89 Li, C.G., He, S.B., Liao, X.F. Yu, J.B. Using recurrent neural network for adaptive predistortion linearization of RF amplifiers. *Int. Rf Microwave computer aided eng.* 12(1), pp.125-130, 2002

90 Selvan, S. Srinivasan, R. Adaptive filtering techniques using neural networks. IETE Tech Rev. 17(1-2), pp.73-78, 2000

91 Catfolis, T. A method for improving the real-time recurrent learning algorithm. *Neural Networks*, 6(6), 807-821, 1993

92 Mandic, D.P. Chambers, J.A. Relating the slope of the activation function and learning rate within recurrent neural networks. *Neural computation*, 11(5), 1069-1077, 1999

93 Lu, Y.L., Mak, M.W., and Siu, W.C. Application of a fast recurrent learning algorithm to text-to-phoneme conversion, to appear in *Proc ICNN'95,* Australia, Dec 1995

94 Lu, Y.L., Mak, M.W., and Siu, W.C. Diminish the computational burden of real time recurrent learning algorithm by constrained sensitivity, to appear in *Proc ICNNSP'95*, China, Dec 1995

95 Mak, M.W., Ku, K.W., and Lu, Y.L. On the improvement of the real time recurrent learning algorithm for recurrent neural networks. *Neurocomputing*, vol 24, issues 1-3, pp. 13-36, Feb 1999

96 Hush, D.R, and Horne, B.G. Progress in supervised neural networks: What's new since Lippmann?" *IEEE Signal Processing,* 10, 8-39, 1993.

97 Burton, R.M., and Mpitsos, G.J. Event-dependent control of noise enhances learning in neural networks. *Neural Networks*, vol. 5, pp. 627-637, 1992

98 Heskes, T. Stochastics of on-line backpropagation. *Proceedings of the European symposium on artificial neural networks*, pages 223-228, 1994

99 Rögnvaldsson, T. On Langevin updating in multilayered perceptrons. *Neural Computation,* **6**, 916-926, 1994

100 An, G. The effect of adding noise during back-propagation training on a generalization performance. *Neural Computation,* **8**, 643-674, 1996

101 Weigend, A., Rumelhart, D., and Huberman, B. Generalization by weight elimination with application to forecasting. *Advances in neural information processing systems 3 (NIPS 90),* pp. 875-882, 1991

102 Chauvin, Y. Dynamic behaviour of constrained back-propagation networks. *Advances in neural information processing* (2), pp. 642-649, 1990

103 Chauvin, Y. Generalization performance of overtrained back-propagation networks. *Neural Networks, Proc EUROSIP Workshop.* pp. 46-55, 1990

104 Chauvin, Y. A back-propagation algorithm with optimal use of hidden units. *Advances in neural information processing systems* I, pp. 519-526, 1989

105 Ji, C. Snapp, R., and Psaltis, D. Generalizing smoothness constraints from discrete samples. Neural Computation, vol. 2, no. 2, 188-197, 1990

106 Lang, K.L., and Michael J, Witbrock, M.J. Learning to Tell Two Spirals Apart. Proceedings of the 1988 Connectionist Models Summer School, Morgan Kaufmann, 1988

107 Linder, R., Wirtz, S., Poppl, S.J. Speeding up backpropagation learning by the APROP algorithm. *Proceedings of the second international ICSC symposium on neural computation,* ICSC Academic Press, pp. 122-128, 2000

108 Serre, D. Matrices: Theory and applications. Springer-Verlag New York, Inc 2002, pp. 147

109 Teoh Eu Jin. Training issues and learning algorithms for feedforward and recurrent neural networks. PhD thesis, National University of Singapore, 2009

110 Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Slew. Extreme learning machine: A new learning scheme of feedforward neural networks. *Proceedings of the international joint conference of neural networks (IJCNN2004),* 25-29, 2004

111 Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Slew. Extreme learning machine: Theory and applications. *Neurocomputing,* 70, 489-501, 2006

112 Zhang, R. Efficient sequential and batch learning artificial neural network methods for classification problems. PhD thesis, Nanyang Technological

University, 2005

113 Guang-Bin Huang. Learning capacity and storage capacity of two hidden layer feedforward networks. *IEEE transactions on neural networks.* 14(2):274-281, 2003

114 Shi'ichi Tamura and Masahiko Tateish1. Capabilities of a four-layered feedforward neural network: four layers versus three. *IEEE transactions on neural networks*: 8(2):251-255, 1997

115 Businger and Golub, 1965. Linear least squares reduction by householder transformations. *Numerische Maths*, 7, 265-276, 1965

116 Björk, A. Solving least linear squares problems by Gram-Schmidt orthogonalization. *Nordisk Tidskr: Information-Be-handling,* vol. 7, pp. 1-21, 1967.

117 Akaike, H. A new look at the statistical model identification. *IEEE Transactions on Automatic Control.* 19(6), 716-723, 1974.

118 The Broad Institute WWW site (http://www.broad.mit.edu/cgi-bin/cancer/publications/pub)

119 WWW site (http://llmpp.nih.gov/lymphoma/data.shtml)

120 Ringner, M, Peterson, C., Khan, J. Analysing array data using supervised methods. *Pharmacogenomics,* 3(3) : 403-415, 2002

121 Su, A., Welsh, J.B., Sapinoso, L.M., Kern, S.G., Dimitrov, P., Schultz, P.G., Powell, S.M. Molecular classification of human carcinomas by use of gene expression signatures. *Cancer research*, 61:7388-7393, 2001

122 Staunton, J., Slomin, D.K., Coller, H.A., Tomayo, P., Angelo, M.J., park, J., Scherf, U., Lee, J.K, Reinhold, W.O., Weinstein, J.N., Mesirov, J.P., Lander, E.S., Golub, T. Chemosensitivity prediction by transcription profiling. *Proceedings of National Academy Sciences, USA,* 98(19):10787-10792, 2001

123 Pomeroy, S., Tamayo, P., Gassenbeek, M., Lisa, M.A., Sturla, M., McLaughlin, M.E., Kirn, J.Y.H., Gournnerova, L.C., Black, P. Lander, E.S., and Golub, T.S. Prediction of central nervous system embryonal tumour outcomes based on gene expression. *Nature,* 415:436-442, 2002

124 Nutt, C.L., mani, D.R., Betensky, R.A., Tamayo, P., Carincross, G., Ladd, C., Pohl, U., Hartmann, C., McLaughlin, M.E., Batchelor, T.T., Black, P.M., Andreas von Deimling, Pomeroy, S., Golub, T., Louis, D.N. Gene expression based classification of malignant glicomas correlates better with survival that histological classification. *Cancer research,* 63:1602-1607, 2003

125 Armstrong, S.A., Staunton, J.E., Silverman, L.B., Pieters, R., den Boer, M.L., Minden, M.D., Sallan, S.E., Lander, E.S., Golub, T.R. Mll translocations specify a distinct gene expression profile that distinguishes a unique leukemia. *Bioinformatics,* 30(16):41-47, 2002

126 Arindarn Bhattacharjee, Richards, W.G., Staunton, J., Cheng Li, Monti, S., Vasa, P., Ladd, C., Javad Beheshti, Bueno, R., Gillette, M., Loda, M., Weber, G., Mark, E.J., Lander, E.S., Wing Wong, Meyerson, M. Classification of human lung carsimonas by mma expression profiling reveals distinct adrenocarcinoma subclasses. *Proceedings of national Academy Sciences*, USA.98(24):13790-13795, 2001

127 Ship, M.A., Ross, K., Tamayo, P. Weng, A.P., Kutok, J.L., Aguiar., R.C.T., Gaasenbeek, M., Angelo, M., Reich, M., Pinkus, G.S., and Golub, T.R. Gene expression correlates of clinical prostrate cancer behaviour. *Nature Medicine,* 8:68-74, 2002

128 Dinesh Singh, Febbo, P.G., Ross, K., Jackson, D.G., Manola, J., ladd, C., Tamayo, P., Renshaw, A., Golub, T.R., Sellers, W. Diffuse large b-cell lymphoma outcome prediction by gene expression profiling and supervised learning machine. *Cancer cell*, 1(2), 2002

129 Qian, N, and Sejnowski, T. Predicting the secondary structure of globular proteins using neural network models. *J. Mol. Biol.,* 202, 865-884, 1988

130 Fei Han, hei-Fen Yao, Qing Hua Ling. An improved extreme learning machine based on particle swarm optimization. *Bio-inspired computing and applications*, volume 6840/2012, 699-704, 2012

131 Jing-Ru Zhang, Jun Zhang, Tat Ming Lok, Michael R. Lyu. A hybrid particle swarm optimization – backpropagation algorithm for feedforward neural network training. *Applied mathematics and computation*, 1026–1037, 2007

132 Xiasheng Wang, Osamu Gotoh. A robust gene selection method for micro-array based cancer classification. *Cancer inform*, 9, 15-30, 2010

133 Ben-Porath I, Thomson, M.W., Carey, V.J., Ruping Ge, Bell, G.W., Regev A, Weinberg, R.A. An embryonic stem cell-lie gene expression signature in poorly differentiated aggressive human tumours. Net Genet, 40(5): 499-507, 2008

134 Segal E, Friedman N, Koller D, Regev A. A module map showing conditional activity of expression modules in cancer. Net Genet, 36:1090-1098, 499-507, 2004

APPENDIX A - PROGRAM DEVELOPMENT

Hardware, software and platform

The implementation of the programs used in the simulations was carried out on a Hewlett Packard Pavilion Pentium 4 2.67 CPU GHz with 512 MB of RAM running Windows XP Professional SP2. The programs were written in Object Pascal using Borland Delphi Professional 6. Delphi is a rapid application development tool (RAD) for developing Windows applications, dynamic linked libraries (DLLs), system control modules, console applications and web server applications. Microsoft Excel was used to plot the graphs and charts included the thesis

Modular structure

A classical procedural paradigm was adopted to implement the algorithms. The program design was broken down into discrete modules implementing strong cohesion and low coupling between modules. Typical modules include

build_net. This procedure accepts the numbers of layers in the neural network and number of inputs and builds the corresponding network structure

add_neuron. This procedure accepts the layer and unit position and adds a neuron to the network

init_weights. This procedure accepts the layer and unit position and initialises the weights in the weight table to random values

set_inputs. This procedure reads a row of data from the input buffer into the input layer of the network

forward_prop. This procedure forward propagates the activations from the input layer to the output layer

backward_prop. This procedure calculates the local gradients of each individual unit in the network

calc_grads. This procedure calculates the gradient contribution of each individual weight in the network

Update_weights. This procedure calculates the adjustments to the weights in the neural network

Data structures

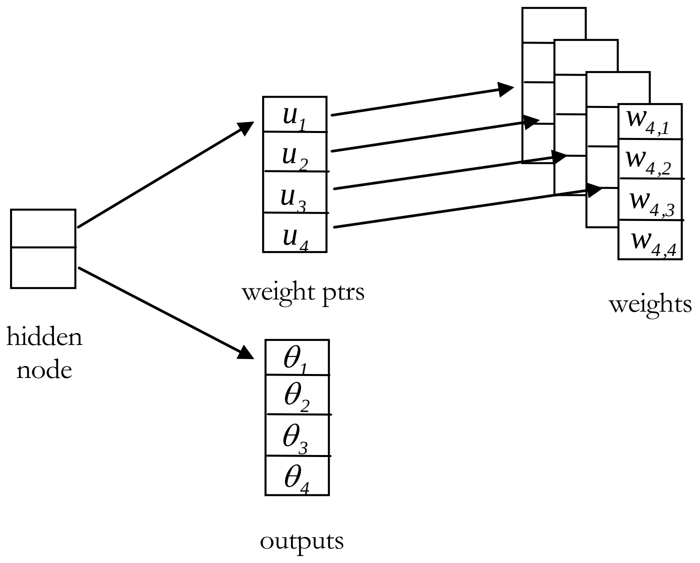


Fig A.1. The figure shows the layer structure used to model a collection of nodes with similar function. In this example the weight values of all input connections to the fourth unit u_1 are stored sequentially in the $w_{1,j}$ array, connections to the second unit u_2 in the $w_{2,j}$ array, and so on, enabling rapid sequential access to these values

The programs used in the simulations utilised many advanced features of a modern programming language. Specifically, dynamic data structures were used to store to store the unit activations, gradient information and weight values. Using a data structure permitted units to be added to the network at runtime so that the memory requirements of the application were optimised to the runtime environment

The network structure consisted of a network node with pointers or references to layer nodes. The layer notes in turn contained pointers or references to output tables, gradient and weight pointer nodes. Finally the gradient and weight pointer nodes contained references to the data tables. Fig A.1 shows part of the data structure used to implement a single layer in the neural network.

Data transformation algorithms

The individual programs used in this study share common routines for data processing, namely procedures to set the inputs, forward propagate activations, backward propagate local gradients, calculate the gradient vector and update the weights of the network. Other modules specific to individual programs include functions to perform the SVD and QR decomposition

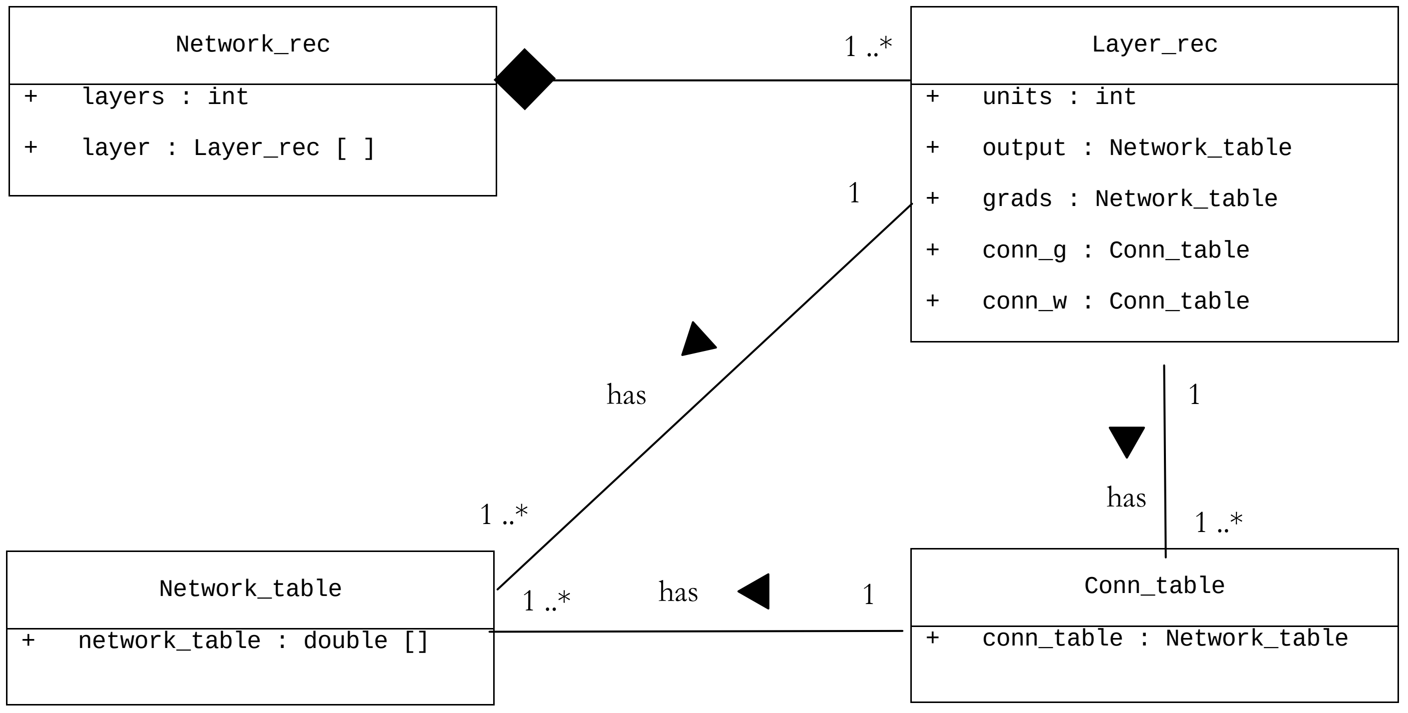
```
Module: set_inputs(j)
  FOR i <- 0 TO inputs
    IN_layer^.output^[i] <- input_buffer[j].pattern[i]
  target <- input_buffer[j].target
END set_inputs

Module: forward_prop
    Module: forward_prop_layer (upper_layer, lower_layer)
      FOR j <- 1 TO upper_layer^.units
          FOR i <- 1 TO lower_layer^.units
              a <- lower_layer^.output^[j] * upper_layer^.conn_w^[j]^[i]
          upper_layer^.output^[j] <- tanh(a)
    END forward_prop_layer
    FOR k <- layers – 1 DOWNTO 1
      upper_layer <- network^.layer [k]
      Lower_layer <- network^.layer [k + 1]
      forward_prop_layer (upper_layer, lower_layer)
END forward_prop

Module: back_prop
    Module: back_prop_layer (upper_layer, lower_layer)
      FOR j <- 1 TO lower_layer^.units
```

```
            FOR i <- 1 TO upper_layer^.units
               lower_layer^.grads^[j] <- lower_layer^.grads^[j] +
                  upper_layer^.grads^[j] * upper_layer^.conn_w^[j]^[i]
               lower_layer^.grads^[j] <- lower_layer^.grads^[j] *
                  (1 – sqr(lower_layer^.output^[j]))
   END back_prop_layer
FOR k <- 1 TO layers – 1
      upper_layer <- network^.layer [k]
      lower_layer <- network^.layer [k + 1]
      back_prop_layer (upper_layer, lower_layer)
END back_prop

Module: calc_grad
   FOR k <- 1 TO layers – 1
      upper_layer <- network^.layer [k]
      lower_layer <= network^.layer [k + 1]
      FOR j <- 1 TO upper_layer^.units
         FOR i <- TO lower_layer^.units
            upper_layer^.conn_g^[j]^[i] <- upper_layer^.grads^[j] *
               lower_layer^.output^[i]
END calc_grad

Module: adjust_weights
   Module: adjust_layer (upper_layer, lower_layer)
      FOR j <- 1 TO upper_layer^.units
         FOR i <- 1 TO lower_layer^.units
            upper_layer^.conn_w^[j]^[i] <- upper_layer^.conn_w^[j]^[i] +
               nu * upper_layer^.conn_g^[j]^[i]
   END adjust_layer
   nu <- calc_learning_rate
   FOR k <- 1 TO layers – 1
      upper_layer <- network^.layer [k]
      lower_layer <- network^.layer [k + 1]
      adjust_layer (upper_layer, lower_layer)
END adjust_weights
```

File organisation and record structure

Most of the benchmark data used in the simulations has been made publically available on websites as text files. Thus the file organisation is sequential. This means that input attributes are accessed sequentially starting from the first item in the file, and reading the attributes one by one until the last item was reached. The input attributes are either floating point or integer. A typical text file would have corresponding data attributes organised in columns, the last column containing the target values. Similarly the results of computations used to determine the performance of the algorithms was written item by item to a text file. The output data was either floating point or integer

APPENDIX B – CLASS DIAGRAM OF DATA STRUCTURE



APPENDIX C – TESTING STRATEGY AND TEST DATA

Testing strategy and test data

The neural network benchmark datasets used to evaluate the newly developed algorithms have all been made publicly available on the UCI machine learning repository unless otherwise stated. These datasets are divided up into training, validation and testing examples. The training data is used to train the network. The performance of the network on the test data estimates its effectiveness in practice. Therefore, the network should not see the test data during the training stage. The training data of some datasets is further subdivided into actual training examples and validation examples. The validation dataset is used to determine the performance of the network during training. A detailed description of the datasets used in this thesis is given below

cancer. The cancer problem contains some diagnosis results for breast cancer. The output represents the classification result for the diagnosis. The decision is based on 9 continuous valued input attributes. The number of training examples is 200 and test examples is 167

hearta1. The hearta problem is an analogue version of the heart disease diagnosis problem. The single continuous output predicts heart disease. The decision is based on 13 continuous valued input attributes. The number of training examples is 690 and test examples is 230

housing. The housing problem is real world problem that estimates the price of housing in the suburbs of Boston. The number of inputs is 13 and the number of outputs one. The number of training and test examples is 253

henon map. This problem is an artificially generated deterministic time series prediction task. Four successive values are used to predict the next value. Thus there are four inputs and one output. The number of training examples is 100 and test examples is 100

two spiral. The two spiral problem is an artificially generated dataset containing the (x, y) coordinates of two interlocking spirals. For points in the first spiral the output is set to +1, and for points on the second spiral -1. The number of training examples is 194 and test examples in 192

additive. The additive problem is an artificially generated nonlinear regression task. There are two continuous valued inputs uniformly distribute in the range [-1,1]. The single continuous output scaled in the range [-1,1]. The number of training examples is 200 and test examples in 200

APPENDIX D – IMPLEMENTATION OF iTPA ALGORITHM

```
//******************************************************************************
// Improved tangent plane algorithm                                             *
// Version: 1.0                                                                 *
// Author: Paul May                                                             *
// Date:                                                                        *
//******************************************************************************
// Description:                                                                 *
// The following program implements the improved tangent plane algorithm (iTPA)   *
// in Object Pascal. Specific libraries used include UN012 which contains the     *
// data structures and UN022 which builds the neural network                    *
//******************************************************************************
// Usage:                                                              *
// The program runs in batch mode                                               *
//   Input files - hearta1.txt                                                  *
//   Output files -                                                             *
//******************************************************************************
program M304;
{$APPTYPE CONSOLE}
uses
  sysUtils,
  UN012 in '..\Units\UN012.pas',
  UN022 in '..\Units\UN022.pas';

const
  b = 1.00;
  c = 1.30;
  n0 = 6.0;
  w0 = 0.05;
  w1 = 0.5;
  tan_b1 = 0.05;
  nu_max = 1.00;
  nu_min = 1.00;
  nu_zero = 1.00;
  sig2_tr = 0.55;
  sig2_val = 0.55;
  err_min = 5.00;
  layers = 4;
  outputs = 01;
  hidden_L1 = 20;
  hidden_L2 = 20;
  inputs = 13;
  pk_max = 1.0;
  vl_max = 1.0;
  epoch_max = 5000;
  epoch_min = 30;
  start  = 001;
  train  = 690;
  pattern   = 920;
  max_trial = 010;
  strip_len = 010;
  high_values = $FFFF;
  low_values = $0000;
  file_IN   = '..\..\data\hearta1.txt';
  file_OUT_1 = 'G:\hearta101.txt';

type
  net_rec_type = record
     pattern : array [1..inputs] of real;
     target  : real;
  end;
  net_file_Type = file of net_rec_type;

var
  OUT_layer : layer_ptr;
  IN_layer  : layer_ptr;

  network : network_ptr;
  net_record : net_rec_type;
  net_IN     : Text;
  net_OUT_1  : Text;

  data : array[1..50] of integer;
  data_buffer : array[1..pattern,1..15] of real;
  input_buffer : array[1..pattern] of net_rec_type;

  pk, vl, err_tr, err_val, err_val_min, target, cerr_tr, cerr_val, perr_tr, perr_val,
  err_strip, err_tr_tmp, err_val_tmp, err_tr_avg, err_tr_last, err_val_avg, epoch_avg,
  nu, wavg : real;
  epoch, epoch_tmp, neurons, i, n : integer;

//attr0, attr1, attr2, attr3, attr4, attr5, attr6, attr7, attr8, attr9, attr10, attr11,
//attr12, attr13 : integer;
  attr0, attr1, attr2, attr3, attr4, attr5, attr6, attr7, attr8, attr9, attr10, attr11,
  attr12, attr13 : real;

function tanh(a : real) : real;
begin
  tanh := (exp(a) - exp(-a)) / (exp(a) + exp(-a));
end;

function inv_tanh(a : real) : real;
begin
  inv_tanh := 0.5 * ln ((1 + a) / (1 - a));
end;

function norm_t : real;
var
  UPPER_layer, LOWER_layer : layer_ptr;
  temp : real;
  i, j, k : integer;
begin
  temp := 0;
  for k := 1 to layers - 1 do
    begin
      UPPER_layer := network^.layer[k];
      LOWER_layer := network^.layer[k + 1];
      for j := 1 to UPPER_layer^.neuron do
        begin
          temp := temp + sqr(UPPER_layer^.conn_t^[j]^[0]);
          for i := 1 to LOWER_layer^.neuron do
            temp := temp + sqr(UPPER_layer^.conn_t^[j]^[i]);
        end;
    end;
  norm_t := sqrt(temp);
end;

function norm_s : real;
var
  UPPER_layer, LOWER_layer : layer_ptr;
  temp : real;
  i, j, k : integer;
begin
  temp := 0;
  for k := 1 to layers - 1 do
    begin
      UPPER_layer := network^.layer[k];
      LOWER_layer := network^.layer[k + 1];
      for j := 1 to UPPER_layer^.neuron do
        begin
          temp := temp + sqr(UPPER_layer^.grads^[j]);
          for i := 1 to LOWER_layer^.neuron do
            temp := temp + sqr(UPPER_layer^.grads^[j] * LOWER_layer^.output^[i]);
        end;
    end;
  norm_s := sqrt(temp);
end;

function norm_r : real;
var
  UPPER_layer, LOWER_layer : layer_ptr;
  temp : real;
  i, j, k : integer;
begin
  temp := 0;
  for k := 1 to layers - 1 do
    begin
      UPPER_layer := network^.layer[k];
      LOWER_layer := network^.layer[k + 1];
      for j := 1 to UPPER_layer^.neuron do
        begin
          temp := temp + sqr(UPPER_layer^.conn_r^[j]^[0]);
          for i := 1 to LOWER_layer^.neuron do
            temp := temp + sqr(UPPER_layer^.conn_r^[j]^[i]);
        end;
    end;
  norm_r := sqrt(temp);
end;

function calc_wavg : real;
var
```

```
  UPPER_layer, LOWER_layer : layer_ptr;
  temp : real;
  i, j, k, num  : integer;
begin
  temp := 0; num := 0;
  for k := 1 to layers - 1 do
    begin
      UPPER_layer := network^.layer[k];
      LOWER_layer := network^.layer[k + 1];
      for j := 1 to UPPER_layer^.neuron do
        begin
          num := num + 1;
          temp := temp + abs(UPPER_layer^.conn_w^[j]^[0]);
          for i := 1 to LOWER_layer^.neuron do
            begin
              num := num + 1;
              temp := temp + abs(UPPER_layer^.conn_w^[j]^[i]);
            end;
        end;
    end;
  calc_wavg := temp / num;
end;

function calc_ws : real;
var
  UPPER_layer, LOWER_layer : layer_ptr;
  temp : real;
  i, j, k : integer;
begin
  temp := 0;
  for k := 1 to layers - 1 do
    begin
      UPPER_layer := network^.layer[k];
      LOWER_layer := network^.layer[k + 1];
      for j := 1 to UPPER_layer^.neuron do
        begin
          temp := temp + UPPER_layer^.conn_w^[j]^[0] * UPPER_layer^.grads^[j];
          for i := 1 to LOWER_layer^.neuron do
            temp := temp + UPPER_layer^.conn_w^[j]^[i] * UPPER_layer^.grads^[j] *
              LOWER_layer^.output^[i];
        end;
    end;
  calc_ws := temp;
end;

procedure calc_r;
var
  UPPER_layer, LOWER_layer : layer_ptr;
  temp1 : real;
  temp2 : real;
  i, j, k : integer;
begin
  temp1 := calc_ws;
  temp2 := norm_s;
  for k := 1 to layers - 1 do
    begin
      UPPER_layer := network^.layer[k];
      LOWER_layer := network^.layer[k + 1];
      for j := 1 to UPPER_layer^.neuron do
        begin
          UPPER_layer^.conn_r^[j]^[0] := UPPER_layer^.conn_w^[j]^[0] - temp1 *
            UPPER_layer^.conn_s^[j]^[0] / sqr(temp2);
          for i := 1 to LOWER_layer^.neuron do
            UPPER_layer^.conn_r^[j]^[i] := UPPER_layer^.conn_w^[j]^[i] - temp1 *
              UPPER_layer^.conn_s^[j]^[i] / sqr(temp2);
        end;
    end;
end;

procedure init_g;
var
  UPPER_layer, LOWER_layer : layer_ptr;
  i, j, k, sgn : integer;
begin
  wavg := calc_wavg;
  for k := 1 to layers - 1 do
    begin
      UPPER_layer := network^.layer[k];
      LOWER_layer := network^.layer[k + 1];
      for j := 1 to UPPER_layer^.neuron do
        begin
          for i := 0 to LOWER_layer^.neuron do
```

```
            begin
              if random > 0.5 then
                sgn := +1
              else
                sgn := -1;
              UPPER_layer^.conn_g^[j]^[i] := (UPPER_layer^.conn_w^[j]^[i] / w0) /
                (power((UPPER_layer^.conn_w^[j]^[i] / w0),n0) + 1) + sgn * random / (wavg / w1);
            end;
        end;
    end;
end;

function norm_g : real;
var
  UPPER_layer, LOWER_layer : layer_ptr;
  temp : real;
  i, j, k : integer;
begin
  temp := 0;
  for k := 1 to layers - 1 do
    begin
      UPPER_layer := network^.layer[k];
      LOWER_layer := network^.layer[k + 1];
      for j := 1 to UPPER_layer^.neuron do
        begin
          temp := temp + sqr(UPPER_layer^.conn_g^[j]^[0]);
          for i := 1 to LOWER_layer^.neuron do
            temp := temp + sqr(UPPER_layer^.conn_g^[j]^[i]);
        end;
    end;
  norm_g := sqrt(temp);
end;

function calc_gs : real;
var
  UPPER_layer, LOWER_layer : layer_ptr;
  temp : real;
  i, j, k : integer;
begin
  temp := 0;
  for k := 1 to layers - 1 do
    begin
      UPPER_layer := network^.layer[k];
      LOWER_layer := network^.layer[k + 1];
      for j := 1 to UPPER_layer^.neuron do
        begin
          temp := temp + UPPER_layer^.conn_s^[j]^[0] * UPPER_layer^.conn_g^[j]^[0];
          for i := 1 to LOWER_layer^.neuron do
            temp := temp + UPPER_layer^.conn_s^[j]^[i] * UPPER_layer^.conn_g^[j]^[i];
        end;
    end;
  calc_gs := temp;
end;

function calc_gr : real;
var
  UPPER_layer, LOWER_layer : layer_ptr;
  temp : real;
  i, j, k : integer;
begin
  temp := 0;
  for k := 1 to layers - 1 do
    begin
      UPPER_layer := network^.layer[k];
      LOWER_layer := network^.layer[k + 1];
      for j := 1 to UPPER_layer^.neuron do
        begin
          temp := temp + UPPER_layer^.conn_r^[j]^[0] * UPPER_layer^.conn_g^[j]^[0];
          for i := 1 to LOWER_layer^.neuron do
            temp := temp + UPPER_layer^.conn_r^[j]^[i] * UPPER_layer^.conn_g^[j]^[i];
        end;
    end;
  calc_gr := temp;
end;

procedure calc_g;
var
  UPPER_layer, LOWER_layer : layer_ptr;
  temp1, temp2, temp3, temp4 : real;
  i, j, k : integer;
begin
  init_g;
  temp1 := calc_gs;
  temp2 := calc_gr;
  temp3 := norm_s;
  temp4 := norm_r;
  for k := 1 to layers - 1 do
    begin
      UPPER_layer := network^.layer[k];
      LOWER_layer := network^.layer[k + 1];
      for j := 1 to UPPER_layer^.neuron do
        begin
          UPPER_layer^.conn_g^[j]^[0] := UPPER_layer^.conn_g^[j]^[0] - temp1 *
            UPPER_layer^.conn_s^[j]^[0] / sqr(temp3);
          for i := 1 to LOWER_layer^.neuron do
            UPPER_layer^.conn_g^[j]^[i] := UPPER_layer^.conn_g^[j]^[i] - temp1 *
              UPPER_layer^.conn_s^[j]^[i] / sqr(temp3);
        end;
    end;
end;

procedure calc_grad;
var
  UPPER_layer, LOWER_layer : layer_ptr;
  temp : real;
  i, j, k : integer;
begin
  temp := 0;
  for k := 1 to layers - 1 do
    begin
      UPPER_layer := network^.layer[k];
      LOWER_layer := network^.layer[k + 1];
      for j := 1 to UPPER_layer^.neuron do
        begin
          UPPER_layer^.conn_s^[j]^[0] := UPPER_layer^.grads^[j];
          for i := 1 to LOWER_layer^.neuron do
            UPPER_layer^.conn_s^[j]^[i] := UPPER_layer^.grads^[j] * LOWER_layer^.output^[i];
        end;
    end;
end;

procedure set_inputs(j : integer);
var
  i : integer;
begin
  IN_layer^.output^[0] := 1;
  for i := 1 to inputs do
    IN_layer^.output^[i] := input_buffer[j].pattern[i];
  target := input_buffer[j].target;
end;

procedure forward_prop;
var
  UPPER_layer, LOWER_layer : layer_ptr;
  k : integer;
  a : real;

  procedure forward_prop_layer(var UPPER_layer, LOWER_layer : layer_ptr);
  var
    j, i : integer;
    a : real;
  begin
    for j := 1 to UPPER_layer^.neuron do
      begin
        a := UPPER_layer^.conn_w^[j]^[0];
        for i := 1 to LOWER_layer^.neuron do
          a := a + LOWER_layer^.output^[i] * UPPER_layer^.conn_w^[j]^[i];
        UPPER_layer^.output^[j] := tanh(a);
      end;
  end;

begin
  for k := layers - 1 downto 1 do
    begin
      UPPER_layer := network^.layer[k];
      LOWER_layer := network^.layer[k + 1];
      forward_prop_layer(UPPER_layer, LOWER_layer);
    end;
end;

procedure backward_prop;
var
  UPPER_layer, LOWER_layer : layer_ptr;
  k : integer;
  procedure backward_prop_layer(var UPPER_layer, LOWER_layer : layer_ptr);
  var
    i, j : integer;
  begin
    for j := 1 to LOWER_layer^.neuron do
      begin
        LOWER_layer^.grads^[j] := 0;
        for i := 1 to UPPER_layer^.neuron do
            LOWER_layer^.grads^[j] := LOWER_layer^.grads^[j] + UPPER_layer^.grads^[i] *
              UPPER_layer^.conn_w^[i]^[j];
        LOWER_layer^.grads^[j] := LOWER_layer^.grads^[j] * (1 - sqr (LOWER_layer^.output^[j]));
      end;
  end;

begin
  for k := 1 to layers - 1 do
    begin
      UPPER_layer := network^.layer[k];
      LOWER_layer := network^.layer[k + 1];
      backward_prop_layer(UPPER_layer, LOWER_layer);
    end;
end;

procedure adjust_weights;
var
  UPPER_layer, LOWER_layer : layer_ptr;
  eta, alpha1, temp1, temp2, temp3, temp4, temp5, temp6 : real;
  k, sgn : integer;

  procedure adjust_layer(var UPPER_layer, LOWER_layer : layer_ptr);
  var
    i, j : integer;
  begin
    for j := 1 to UPPER_layer^.neuron do
      begin
        for i := 0 to LOWER_layer^.neuron do
          begin
            UPPER_layer^.conn_t^[j]^[i] := nu * (eta * UPPER_layer^.conn_s^[j]^[i] / temp1 –
              alpha1 * TAN_B1 * (UPPER_layer^.conn_g^[j]^[i] / temp5));
            UPPER_layer^.conn_w^[j]^[i] := UPPER_layer^.conn_w^[j]^[i] +
              UPPER_layer^.conn_t^[j]^[i];
          end;
      end;
  end;

begin
  temp1 := norm_s;
  temp2 := norm_r;
  temp5 := norm_g;
  eta := (inv_tanh(target) - inv_tanh(OUT_layer^.output^[1])) / temp1;
  alpha1 := abs(inv_tanh(target) - inv_tanh(OUT_layer^.output^[1])) / temp1;
  for k := 1 to layers - 1 do
    begin
      UPPER_layer := network^.layer[k];
      LOWER_layer := network^.layer[k + 1];
      adjust_layer(UPPER_layer, LOWER_layer);
    end;
end;

procedure train_net;
var
  j : integer;
begin
  for j := start to train do
    begin
      set_inputs(j);
      forward_prop;
      backward_prop;
      calc_grad;
      calc_r;
      calc_g;
      adjust_weights;
    end;
end;

procedure test_net;
var
  t_max, t_min : real;
  j, x, y : integer;
begin
  err_tr := 0;
```

```
//err_tr := 1;
  cerr_tr := 0;
  t_max := LOW_VALUES;
  t_min := HIGH_VALUES;
  for j := start to train do
    begin
      set_inputs(j);
      forward_prop;
      if target > t_max then
        t_max := target;
      if target < t_min then
        t_min := target;
      err_tr := err_tr + sqr(target - OUT_layer^.output^[1]);
//    err_tr := err_tr * sqr(target - OUT_layer^.output^[1]);
      if target >= 0 then
        y := 1
      else
        y := -1;
      if OUT_layer^.output^[1] >= 0 then
        x := 1
      else
        x := -1;
      cerr_tr := cerr_tr + abs(y - x);
    end;
// percentage sum of square error
  perr_tr := 100 * err_tr / (train * sqr(t_max - t_min));
// Classification error CERR x 100
  cerr_tr := 100 * cerr_tr / (2 * train);
// Normalised mean square error NMSE x 100
  err_tr := 100 * err_tr / (train * sig2_tr);
//err_tr := 100 * power(err_tr, 1 / train) * sig2_tr;
//err_tr := err_tr / (train * sig2_tr);
end;

procedure validate_net;
var
  t_max, t_min : real;
  j, x, y : integer;
begin
  err_val := 0;
  cerr_val := 0;
  t_max := LOW_VALUES;
  t_min := HIGH_VALUES;
  for j := train + 1 to pattern do
    begin
      set_inputs(j);
      forward_prop;
      if target > t_max then
        t_max := target;
      if target < t_min then
        t_min := target;
      err_val := err_val + sqr(target - OUT_layer^.output^[1]);
      if target >= 0 then
        y := 1
      else
        y := -1;
      if OUT_layer^.output^[1] >= 0 then
        x := 1
      else
        x := -1;
      cerr_val := cerr_val + abs(y - x);
    end;
// Percentage sum of square error
  perr_val := 100 * err_val / ((pattern - train) * sqr(t_max - t_min));
// Classification error CERR x 100
  cerr_val := 100 * cerr_val / (2 * (pattern - train));
// Normalised mean square error NMSE x 100
  err_val := 100 * err_val / ((pattern - train) * sig2_val);
//err_val := err_val / ((pattern - train) * sig2_val);
end;

procedure net_input;
var
  i, j : integer;
begin
  assign(net_IN, file_IN);
  reset(net_IN);
  for j := start to pattern do
    begin
      for i := 1 to inputs do
        begin
          read(net_IN, attr1);
          input_buffer[j].pattern[i] := attr1 / b;
        end;
      readln(net_IN, attr0);
      input_buffer[j].target := attr0 / c;
    end;
  close(net_IN);
end;

procedure train_strip;
var
  err_strip, err_strip_min : real;
  j : integer;
begin
  err_strip := 0;
  err_strip_min := high_values;
  for j := 1 to strip_len do
    begin
      train_net;
      test_net;
      if err_strip_min > err_tr then
        err_strip_min := err_tr;
      err_strip := err_strip + err_tr;
    end;
  pk := 1000 * (err_strip / (strip_len * err_strip_min) - 1);
end;

procedure validate_val;
var
  j : integer;
begin
  validate_net;
  if err_val_min > err_val then
    err_val_min := err_val;
  vl := 100 * ((err_val / err_val_min) - 1);
end;

begin

  assign(net_OUT_1, file_OUT_1);
  rewrite(net_OUT_1);

  n := 0;
  err_val_tmp := high_values;
  err_val_avg := low_values;
  err_tr_avg := low_values;
  epoch_avg := low_values;
  for i := 1 to max_trial do
    begin
      epoch := 0;
      nu := nu_zero;
      err_val_min := high_values;
      err_tr_last := low_values;
      new(network);
      build_network(network, layers, inputs);
      add_neuron(network, 1, outputs);
      for neurons := 1 to hidden_L1 do
        add_neuron(network, 2, neurons);
      for neurons := 1 to hidden_L2 do
        add_neuron(network, 3, neurons);
      OUT_layer := network^.layer[1];
      IN_layer  := network^.layer[layers];
      OUT_layer^.grads^[1] := 1.0;
      net_input;
      init_weights(network, 1, outputs);
      for neurons := 1 to hidden_L1 do
        init_weights(network, 2, neurons);
      for neurons := 1 to hidden_L2 do
        init_weights(network, 3, neurons);
      repeat
        train_strip;
        validate_val;
        writeln(i:6, epoch:6, perr_tr:12:3, perr_val:12:3, vl:12:3, wavg:12:3);
        epoch := epoch + strip_len;
      until (epoch >= EPOCH_MAX) or ((err_tr <= ERR_MIN) and
            (vl > VL_MAX));
      if (err_val < err_val_tmp) then
        begin
          err_val_tmp := err_val;
          err_tr_tmp := err_tr;
          epoch_tmp := epoch;
        end;
      dispose(network);
      if (epoch < EPOCH_MAX) then
        begin
          err_val_avg := err_val_avg + err_val;
          err_tr_avg := err_tr_avg + err_tr;
          epoch_avg := epoch_avg + epoch;
        end;
    end;

  writeln('Training error   ', err_tr_tmp:9:2);
  writeln('Validation error ', err_val_tmp:9:2);
  writeln('Epochs           ', epoch_tmp:9);

  writeln('Avg training error   ', err_tr_avg / n:9:2);
  writeln('Avg validation error ', err_val_avg / n:9:2);
  writeln('Avg epochs           ', epoch_avg / n:9:2);

  close(net_OUT_1);  readln;

end.
```