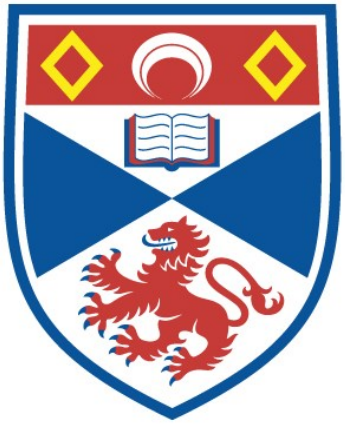
ROBUSTNESS AND GENERALISATION: TANGENT HYPERPLANES AND CLASSIFICATION TREES

Antonio Ramires Fernandes

A Thesis Submitted for the Degree of PhD
at the
University of St Andrews



1997

Robustness and Generalisation: Tangent Hyperplanes and Classification Trees

A thesis submitted to the

UNIVERSITY OF ST. ANDREWS

for the degree of
DOCTOR OF PHILOSOPHY

By

Antonio Ramires Fernandes

July 1997

I, Antonio Jose Borba Ramires Fernandes, hereby certify that this thesis, which is approximately 60000 words in length, has been written by me, that is the record of work carried out by me and that it has not been submitted in any previous application for a higher degree.

Date: 14/2/97 Signature of candidate:

I was admitted as a research student under Ordinance No. 12 in 1991 and re-registered as a candidate for the degree of Doctor of Philosophy in 1992; the higher degree study for which this is a record was carried out in the University of St. Andrews between 1991 and 1996.

Date: 14/2/97 Signature of candidate:

In submitting this thesis to the University of St. Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker.

Date: 14/2/97 Signature of candidate:

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the Degree of Doctor of Philosophy in the University of St. Andrews and that the candidate is qualified to submit the thesis in application for that degree

Date: 24/2/97 Signature of supervisor:

Abstract

The issue of robust training is tackled for fixed multilayer feedforward architectures. Several researchers have proved the theoretical capabilities of Multilayer Feedforward networks but in practice the robust convergence of standard methods like standard backpropagation, conjugate gradient descent and Quasi-Newton methods may be poor for various problems. It is suggested that the common assumptions about the overall surface shape break down when many individual component surfaces are combined and robustness suffers accordingly.

A new method to train Multilayer Feedforward networks is presented in which no particular shape is assumed for the surface and where an attempt is made to optimally combine the individual components of a solution for the overall solution. The method is based on computing Tangent Hyperplanes to the non-linear solution manifolds. At the core of the method is a mechanism to minimise the sum of squared errors and as such its use is not limited to Neural Networks. The set of tests performed for Neural Networks show that the method is very robust regarding convergence of training and has a powerful ability to find good directions in weight space.

Generalisation is also a very important issue in Neural Networks and elsewhere. Neural Networks are expected to provide sensible outputs for unseen inputs. A framework for hyperplane based classifiers is presented for improving average generalisation. The framework attempts to establish a trained boundary so that there is an optimal overall spacing from the boundary to training points closest to this boundary. The framework is shown to provide results consistent with the theoretical expectations.

Acknowledgements

This thesis would not have been possible without the help of Mike Weir, my supervisor. I would like to thank Margarida Fernandes, my mother, for lending me her Macintosh for what turned out to be a long period of time, and Margaret and Helen, from the Computer Science Division.

Last, but definitely not least, I would like to thank Carmen Roriz, my wife, and Tiago Samuel, my son, for putting up with me and for encouraging me unconditionally.

This thesis was supported by JNICT, Junta Nacional de Investigação Cientifica e Tecnologica, under the grant BD/1593/91-IA.

1. Introduction

1.1. Brains and Computers

Although nowadays computers of the digital symbolic type are said to be very powerful machines the human brain still outperforms them very easily in a variety of tasks. In fact, brain and symbolic computers seem to complement each other, or as Caudill and Butler (1989) put it, "brains and computers are handy things to have". They are good at fundamentally different things.

Caudill and Butler (1989) present two examples that show a fundamental difference between a symbolic computer and a person. The first is dividing a seven digit number by another number. This task is extremely difficult for the average human person, it requires following a complicated symbolic algorithm and there is plenty room for errors to occur. On the other hand an electronic calculator can do it very rapidly and without any errors. The second example involves recognising a human face in a crowded room. In this case, the task is fairly simple for a human but extremely hard even for the most advanced symbolic computer running the most advanced software.

It is the awareness of this discrepancy in abilities that today leads scientists to research on how the brain operates and ultimately how to emulate the brain's non-symbolic behaviour in a computer.

The brain is a fascinating 'machine'. Its capabilities are astonishing by any standards. Hertz, Krogh and Palmer (1991) list some of the features that the brain possesses:

- It is robust and fault tolerant. Nerve cells die everyday without affecting its performance significantly;

- It is flexible. It can easily adjust to a new environment by 'learning' - It doesn't have to be programmed in Pascal, FORTRAN or C;
- It can deal with information that is fuzzy, probabilistic, noisy or inconsistent;
- It is highly parallel.

These features are highly desirable in computational systems. Neurocomputing is the science that attempts to incorporate these features in a computational system. In §1.3 a more detailed analysis is done comparing the classical computational systems based on the symbolic paradigm with neurocomputing.

Although in the early ages of neurocomputing researchers inside the community were careful to make their neurocomputing models plausible from a biological point of view, nowadays neurocomputing is no longer strongly attached to biological factors. A rough and simple modelling may be sufficient to extract at least some of the previously mentioned properties. If initially neurocomputing was used mainly as a computational model for the brain, today neurocomputing is used for many practical applications like forecasting, pattern classification, etc., and therefore this detachment from biology is justifiable.

However since the initial inspiration for neurocomputing was the brain, some biological terms remain in use. For instance, a neurocomputing system is a network of units usually referred to as neurons, and some researchers still talk about synapses when referring to the connections between neurons.

1.2. The Neurocomputing vs. The Symbolic Approach

When solving a problem using the classical approach one has to specify explicitly a set of symbolic rules for the computer to solve the problem. As an

example of the classical approach expert systems are presented. Expert systems are based on a knowledge base, an inference engine and a user interface. The knowledge base is a problem specific set of IF-THEN rules that describes the particular problem being dealt with. The inference engine is the module that consults the knowledge base and acts upon the information in it. The user interface links the inference engine to the external environment.

To create a knowledge base for an expert system the knowledge needed to solve the problem must be incorporated into the knowledge base. However such knowledge is in some situations incomplete. The factors that influence the problem might not be fully known. Furthermore, according to Gallant (1993), in many cases the knowledge is very hard to translate into rules. For instance, in the general case humans are very good at recognising faces. However writing a set of IF-THEN rules to distinguish two persons based on their photographs is by no means a trivial task.

Gallant (1993) says that the process of construction and debugging a knowledge base is the main problem in building expert systems. Gallant goes even further saying that once the knowledge is extracted from the expert(s), the knowledge base is almost certain to be incomplete or inconsistent.

The neurocomputing approach couldn't be more different. A neurocomputing system adapts itself to solve a problem without being told symbolic rules for how the problem is solved. For instance, in a face recognition problem, a neurocomputing system can learn to distinguish persons based on their photographs without being told specifically what are the differences between the persons. There is no need for an expert to tell the system how each person's face differs from the others.

In some cases an expert is still needed; however the expert's role is different from the classical symbolic approach. For instance in forecasting problems, the expert does not need to specify symbolic rules that model the process as in the traditional approach. Instead, the role of the expert is mainly to identify the variables and targets which may be important to the forecast and provide a significant set of examples of how the system should behave. It is up to the neurocomputing system to learn how to relate the variables in order to obtain a correct forecast.

A neurocomputing system learns by example; a large enough set of examples must be created to teach the system. The neurocomputing system extracts the rules in the asymbolic sense outlined by Denker et. al. (1987) needed to solve the problem during the learning process based on the examples presented.

The structure of the examples, called patterns from now on, depends on the neurocomputing paradigm being used. Some paradigms require the pattern to have two components, an input vector, or input pattern, and an output vector or target. These patterns can be seen as the desired behaviour for the neurocomputing system to learn, i.e. they specify what the system should respond, the target, when prompted with a query, the input pattern. Other paradigms require only the input pattern. In these cases it is up to the system to learn how to separate the different examples into classes.

The learning paradigms can be broadly classified as unsupervised, supervised, or reinforcement learning. In the unsupervised learning case only the input pattern is presented. It is up to the neural net to divide the patterns into classes based on their similarities and differences. Reinforcement learning uses only the input patterns as in the unsupervised learning but a grade is given to the neural net telling how its performance has improved in general terms since the last time it was graded. The supervised learning strategy considers a pattern as having both the input

pattern and a respective specific target output pattern. In this way, each time the neural net is presented with an input pattern an error can be computed based on the difference between the output given by the net and the specific target. The error is then fed to the system so that the system can evaluate its performance and correct its behaviour accordingly.

Another important feature of neurocomputing is the capability of dealing with incomplete or even partially incorrect inputs. This further enhances the performance of neurocomputing systems when compared with classical computing approaches. Furthermore, in the general case, a neurocomputing approach is capable of providing a sensible answer in geometric terms to inputs it has never seen before. For example, in the face recognition problem, when a neurocomputing system receives a slightly blurred photograph it may still recognise the face in it.

The above mentioned features give an edge to neurocomputing systems over symbolic systems such as expert systems for some problems. There are however areas in which expert systems, for example, are clearly better suited than neurocomputing. Expert systems are able to tell the user how they reached a certain output from an input state in meaningful terms. The symbolic rules used to achieve the output can be listed, reassuring the user, or at least justifying why the expert system achieved a certain output. This is not always possible with the neurocomputing approach. The knowledge base in a neurocomputing approach is not as easy to consult as the set of IF-THEN rules that forms the knowledge base of expert systems. The knowledge base in a neurocomputing approach is defined in more detail in §1.3.

There are some interesting approaches to deal with this problem of explanation for the neurocomputing approach. Casimir Klimasaukas (1991) analyses how the outputs of a neurocomputing system vary when the input state is slightly modified. Doing this analysis for each variable of the input

example enables a conclusion to be drawn as to which inputs from the input example had more influence on the output obtained.

In order to combine the best features of both systems a hybrid approach is possible. For an example, Gallant (1993) shows how a knowledge base of a neurocomputing system can serve as a knowledge base for an expert system that performs classification tasks.

1.3. General Structure of Feedforward Neural Nets

A neurocomputing system in its most simple form is a single neural network. From now on, unless stated otherwise, this thesis deals only with one class of neural networks: feedforward networks.

A multilayer feedforward net can be defined as a box with a set of inputs and a set of outputs. Inside this box are a number of simple processing units called neurons that communicate with each other through weighted links. Signals are sent to a neuron either from the inputs or from other neurons through the weighted links, hereafter called weights. These signals are combined in some way to become the excitation of the neuron. The excitation may be then further processed to obtain an activation which in turn may be further processed to obtain an output for the neuron. The output of a neuron is then sent to other neurons or to the outputs of the box. Since the network must be feedforward there must be no cyclic paths in the network.

The set of weights specifies not only which neurons connect to which but the strength of the respective connection. The outputs obtained when the box is presented with an input pattern are a function of these weights. Therefore the set of weights can be seen as the internal state of the box.

Initially, when the box is constructed, an internal state is selected randomly, therefore it is highly unlikely that the box will perform the desired

input/output relation. In such situations the box has to change its internal state to perform the correct input/output mapping. A learning rule is used for this effect, gradually changing the internal state of the box so that the initial input/output relation performed by the neural net becomes the desired input/output association.

To illustrate how learning occurs the supervised learning mode is presented since this is the learning mode used in this thesis.

The learning process is often an iterative process. A set of patterns, each pattern made of an input vector and a target, is presented to the box. An error can then be computed for the set of patterns comparing the outputs obtained for each pattern with the targets for the respective pattern. Afterwards the learning rule adjusts the internal state of the box so that the error is decreased. The set of patterns is presented again and again, and the internal state is modified each time, until a sufficiently low error is obtained.

The internal state of a neural net can be seen as representative of the knowledge a neural net contains. Therefore it makes sense to talk about the knowledge base of a neural net when referring to its internal state. Hence the knowledge base of a neural net is the set of connections present in the neural net as well as their strengths. The knowledge is represented in a numerical fashion as opposed to the symbolic approach used in expert systems. This is the main reason why it is so hard to extracts reasons for action from the knowledge base of a neural net.

1.4. Issues in Neural Nets

As mentioned before this thesis deals with a particular class of neural nets, the multilayer feedforward nets. These nets use the supervised learning mode in which a pattern has both an input component and a target component. The internal structure of these nets is discussed in more detail later in §2.2.

The issues discussed here are the ones relevant to this type of net. Researchers in Neural Networks face three main issues: the internal structure of the neural net, called the architecture, the generalisation ability, and the robustness of the learning regimes. In the following subsections each of these issues is discussed.

1.4.1. The Architecture

The first issue that faces someone trying to use neural nets to solve a particular task is how to define the architecture of the net, i.e. what is the number of neurons needed and how to connect them to perform a certain task.

One possible approach to avoid having to determine the precise number of neurons needed is based on algorithms that consider the architecture to be a variable. This approach is capable of finding an appropriate architecture as part of the learning process, therefore eliminating to a certain extent the problem of knowing a priori the number of neurons needed to solve a problem. These algorithms, in the general case, won't find the minimal necessary architecture to solve a problem but rather an architecture that is sufficient to solve the problem of realising the training set. Hertz, Krogh and Palmer (1991) divide these algorithms into three broad categories : Pruning, weight decay, and growing algorithms.

Pruning algorithms start with a large enough number of neurons to realise the training set and try to eliminate unnecessary neurons either during or after learning has occurred, see Siestma and Dow (1988) for an example. If the elimination is done after training, then it is necessary to retrain the network but according to Hertz, Krogh and Palmer (1991), the retraining is usually rather fast. A disadvantage of this technique is how to determine what is a large number of neurons for a particular task without falling into

extremes because the larger the number of neurons the more computationally expensive the training process will be.

Regarding the topology, Weight Decay, proposed by Lang and Hinton (1990), removes unused connections from the network. The principle is to diminish the weight values after each iteration so that weights that are not reinforced will tend to zero. Weights below a threshold value are removed. As mentioned before for the pruning algorithms, with this technique having a priori knowledge about what is a large enough architecture to solve the problem, without falling into extremes, is also requested for good performance.

Growing algorithms start with few neurons and add neurons to the network as needed to realise the desired I/O relation. For examples of these techniques see Mezard and Nadal (1989), Fahlman and Lebiere (1990), and Frean (1990). A possible disadvantage of these methods is that they may end up with a much larger architecture than needed to solve the problem.

It is also possible to combine these methods, for instance one can use a growing algorithm to construct an initial architecture that solves the problem and afterwards use a pruning algorithm or weight decay to remove the unnecessary hidden units.

1.4.2. Generalisation

The ability to generalise is very important in a neural network. If one wanted a lookup table for a particular training set then using a neural network wouldn't provide any major advantages over lookup tables, besides compactness, considering that training a neural net is by no means a trivial task. Therefore, if compactness isn't the main issue, one can only justify using a neural network in problems where there is a need for generalisation and this is why generalisation is so important.

A feedforward neural net can be seen as a model to perform an I/O mapping in which the weights are the free parameters. As in any interpolative method, an excessive number of free parameters results in overfitting (Hertz, Krogh and Palmer 1991). Although an overfitted model will provide an error free output for the training set, its ability to generalise, i.e. to give "sensible" answers when presented with inputs that are not present in the training set, is relatively poor in the general case.

An approach to solve this problem is called regularisation. Regularisation encourages smoother network mappings by adding an extra term to the error function, see Bishop (1995). The Weight Decay algorithm by Lang and Hinton (1990) mentioned before in §1.4.1 can also be used for regularisation.

According to the assumption that smoothness is desirable the best model tends to be the one that provides correct answers to the inputs without an excessive number of free parameters. If a network is overfitting the problem then reducing the number of units and weights reduces the number of free parameters, weights, and therefore tends to provide a better generalisation.

There is another side to the generalisation problem, the selection of the training set. If the training set is not truly representative of the underlying function from which it is extracted then there is no guarantee of sensible answers for inputs not belonging to the training set. In this case there is the possibility that the network will find a rule which the patterns obey but that isn't the desired underlying rule for the original problem (Denker et. al. 1987).

Turban (1992) reports such a situation. The objective was to construct a system that would detect the existence of tanks in a picture. The approach taken used a neural network. Two sets of pictures of a battlefield were taken, one with tanks and the other without tanks but containing rocks and other

objects. After an extensive training session, the network learned to distinguish between the two sets of pictures providing an accurate response to each member of the training set for almost 100% of the pictures. Later someone discovered that two cameras were used to construct the training set. All the pictures with tanks had been taken with one camera and these photos were slightly darker than the photos from other camera. The neural network had learned to tell which camera had been used for a photograph and not if tanks were or were not present.

The above example shows that the selection of a representative training set is, as the choice of architecture, fundamental. Denker et. al. (1987) says that there will always be many underlying rules for any training set and therefore many valid generalisations for the network to choose. However odds can be improved by, for example, taking care when selecting the training set, for instance, to avoid situations like the one described in the above example.

1.4.3. Robustness

Robustness of the training regime for the training set is an issue that is twofold, one aspect lies with the probability that the system will find a solution that realises the training set if such a solution exists, the other aspect is related to the amount of time the neural net takes to reach the solution. This issue was first raised by Minsky and Papert (1969) and although neural nets have evolved considerably since then it still remains a major problem.

Neural nets, while theoretically very powerful in their mapping ability, see Hecht-Nielsen (1989) and Lippmann (1987), in practice take an unfeasible amount of time to learn to perform some tasks. Baum and Lang (1990) report a problem, see §5.3, where although an internal state that realises the desired

input/output mapping for a certain architecture exists, the learning rules tested were unable to find it in a feasible amount of time starting from several random internal states.

Minsky and Papert (1988) suggest that one way to deal with the robustness problem is to break the initial task into several smaller tasks and have one network for each task. They even go further suggesting the use of different neural net paradigms or other Artificial Intelligence techniques to solve each of the subtasks.

However, there are failures reported even for problems that are not particularly large, e.g. Baum and Lang (1990). Even in small toy problems, if the wrong learning parameters are set, the failure rate can become unacceptable. This suggests that breaking a large problem into smaller subtasks may not be the whole answer.

For this reason, it is fair to say that robustness is a fundamental issue in Neural Nets. When one considers that training a neural net is often a very lengthy process, even in fast computers, it is difficult to overstate the significance of robustness. If robustness isn't pursued then choosing an appropriate architecture and a representative training set can be wasted efforts.

It is therefore a priority to find a robust algorithm to train neural nets. This thesis explores a new approach to train multilayer feedforward neural nets that is more robust than standard techniques.

1.5. Thesis Structure

In the first part of chapter 2 a brief history is presented showing the path from the first computational model proposed for the neuron up to multilayer feedforward networks. In the remaining sections, two neural net models are

presented and several learning rules discussed. The presentation includes a graphical perspective whenever possible to help the understanding of the concepts being introduced. Some of the issues raised in chapter 1 are discussed for the learning rules presented.

A novel approach to train multilayer feedforward networks is presented in chapters 3 and 4. This new approach looks at the individual information provided by each pattern and combines it in a more fruitful way than existing comparable techniques.

The main objective is to create a robust algorithm to train multilayer feedforward nets. The results presented in chapter 5 show that this objective has been achieved. The robustness of the algorithm is clear from the success rates obtained as well as from the low number of epochs needed.

In chapter 6 a constructive type of algorithm is presented for classification problems. The results presented show that the algorithm is very fast in 2-D input spaces and that relatively low number of hidden units are used.

The issue of generalisation is tackled in chapter 7 with a new framework for generalisation being presented.

In chapter 8 conclusions are presented.

1.6. Notation

1.6.1. Equations

Upper case is used to represent vectors and matrices, while lower cases are assigned to individual variables. An element from a vector is represented by the same name as the vector or matrix it belongs to but in lower case and it is indexed for its position in the vector or matrix. For instance a_{ij} indicates the element on the i^{th} row, j^{th} column of matrix A. All variables are in italics.

The p-norm of a vector is represented with the symbol ||. For instance $\|V\|_2$ represents the 2-norm of vector V. A single bar is used for modulus operations, for instance $|a|$ represents the absolute value of a.

The equations are numbered according to chapter, section and their relative position in the section. Whenever a reference to an equation appears in the text, the reference appears in curved brackets. For instance (3.1.10) relates to the 10^{th} equation in the first section of chapter 3.

1.6.2. Figures and Tables

Figures representing Euclidean spaces are done in 2-D for reasons of simplicity, nevertheless they are readily generalised to n-D.

The numbering of figures and tables includes the number of the chapter and index of the figure in the chapter. For instance fig. 3.2 indicates the second figure to appear in chapter 3.

2. An Introduction to Artificial Neural Nets

As mentioned before in chapter 1, Artificial Neural Nets were first inspired by the workings of the brain. The first studies in this field attempted to create a model for the brain's neuron.

The landmark paper from McCulloch and Pitts (1943) is perhaps the first to propose a computational model of the neuron in the brain. However, learning was not contemplated in this early model. The power of this model was grounded on the fact that, although each neuron model was a very simple device, a network built with these models could perform very complex logical propositions. The power of these networks arises from the massive parallelism of the system. That is, since many neurons in the network are working at the same time the system is capable of high performance even if each individual neuron has a low performance.

The workings of these model neurons are very simple. A model neuron receives inputs either from the outside world or from other neurons. These inputs belong to one of two categories: excitatory or inhibitory. If a neuron receives an inhibitory input then it will not fire regardless of the amount of excitatory inputs it receives. If only excitatory inputs are present then the neuron sums the inputs and will fire if the sum exceeds some fixed threshold.

The McCulloch-Pitts model neuron performs like a threshold logical device in which no learning occurs. In fact according to Anderson and Rosenfeld (1988) "It is possible to buy McCulloch-Pitts neurons at your local Radio Shack store, in the form of logical circuits".

The book by Donald Hebb (1949) is the first to propose a physiological learning rule. Hebb's rule states that if a neuron contributes repeatedly or persistently to another neuron's firing then the connection between them is

somehow strengthened. In essence, what Hebb is proposing is that the connections between neurons are adaptive parts of the brain.

The perceptron, proposed by Rosenblatt (1958), is the first adaptive artificial neural network that is computationally oriented and is capable of 'learning', i.e. capable of adaptive behaviour. In the 1958 paper, the learning rules presented were largely based only on reinforcement of connections as proposed by Hebb. However, Rosenblatt proposes that, in a computational context, besides reinforcing the connections between neurons, the thresholds should also be adjusted.

In a later book, Rosenblatt (1961), the capabilities of the simplest class of perceptrons are discussed. Also presented in this book is The *Perceptron Convergence Theorem* that guarantees that the perceptron will converge to a solution in a finite amount of time if such a solution exists. The learning rule presented in the theorem involves reduction of the strength between connections as well as reinforcement.

Perceptrons were widely used for classification problems. Although the perceptron could theoretically learn to correctly perform classification on any linearly separable problem (see §2.1.1 for further details on linear separability), many learning rules took an unfeasible amount of time to converge to a solution.

Around the same time, Widrow and Hoff (1960) proposed the ADALINE, a network related to Rosenblatt's simplest class of perceptrons, with the same capabilities but with a faster and more accurate learning rule, the delta rule. (For an extended review of the ADALINE network architecture and learning rule see §2.1.)

Minsky and Papert in their book *Perceptrons* (1969) point out that the class of solvable problems with Perceptrons is very restricted in practical terms.

They do a very penetrating and methodical analysis of the potential and limitations of the perceptron. They draw the reader's attention to the fact that although the perceptron theorem proves that the perceptron will theoretically converge to a solution in a finite amount of time if a solution exists, in practice very long convergence times are needed for large problems in the general case. According to their analysis based on several learning rules, including the ADALINE learning rule, the perceptron performance deteriorates very rapidly when the problem size increases. Results in *Perceptrons* show that the scaling problem is a real theoretical issue.

Furthermore, Minsky and Papert (1969) considered that the extension to multilayer networks (see §2.2) would not overcome the limitations of the perceptron, so that as they put it "...we consider it to be an important research problem to elucidate (or reject) our intuitive judgement that the extension is sterile".

According to Anderson and Rosenfeld (1988), *Perceptrons* contributed largely to the dismissal of neural nets as a serious research subject. According to Hecht-Nielsen (1989) "The artificial intelligence community got all of the neural research money". The subject went underground. But Anderson and Rosenfeld also state that *Perceptrons* is a brilliant book and that it would be unfair to blame *Perceptrons* alone for the decline of interest in the area. Hecht-Nielsen also corroborates the view that *Perceptrons* is not solely responsible for the decay in interest in neural networks.

According to Bernard Widrow (1987), there had always been great resistance to the whole idea of actually building an artificial 'chunk of the brain'. Also, there was much hype in the media about the subject. Rosenblatt (1961) quotes a newspaper headline in his book: "Frankenstein Monster Designed by Navy Robot That Thinks". Hecht-Nielsen (1989) also points out the excessive hype

surrounding the subject as a factor to discredit the area and anger technical people from other fields.

Furthermore, Anderson and Rosenfeld (1988) say that interest in the early network models was in decline for several years before *Perceptrons* appeared; according to them "the perceptron failed to achieve much beyond the initial success". *Perceptrons* was the last stroke in an already moribund field.

Nevertheless, some researchers still continued to try to find a way to overcome the limitations of the perceptron. The solution was found in an extension to the ADALINE network architecture and learning rule. These nets are called multilayer feedforward networks (§2.2) and the new learning rule is now commonly called backpropagation (§2.3) or the generalised delta rule.

This rule seems to have been discovered independently several times. Werbos (1974) seems to be the first to present a successful rule to train multilayer feedforward networks. Parker (1985) rediscovered the learning rule but only when the rule was rediscovered for the third time by Rumelhart, Hinton and Williams (1986), did it become popular.

Multilayer feedforward networks have been found both theoretically and practically to be much superior to the perceptron or the ADALINE. Lippmann (1987) shows that a multilayer feedforward network can solve any classification problem. Also, in function approximation, multilayer feedforward networks have been found to be powerful tools. There are a number of theorems quoted by Hecht-Nielsen (1989) that prove that, for any L_2 function f, there is a multilayer feedforward network (§2.2.1) that can implement f to any desired degree of accuracy. The set of functions L_2 includes, for example, continuous functions, and it includes discontinuous functions that are piecewise continuous on a finite number of subsets of the

domain. According to Hecht-Nielsen (1989) "L2 includes any function that could ever arise in a practical problem". Similar work was done around the same time by Lapedes and Farber (1988), Yan Le Cun (1987), Hornik, Stinchcombe and White (1988), Moore and Poggio (1988), and Irie and Miyake (1988).

Despite these improvements, some of the initial criticism from Minsky and Papert (1969) directed at the perceptron still applies to multilayer feedforward networks. In the revised edition of *Perceptrons* (1988) Minsky and Papert use Rumelhart, Hinton and Williams' (in Rumelhart, McClelland et. al., 1986) own results to point out that the scaling problem described earlier in this section is still present in multilayer feedforward networks trained with back-propagation. It is important to focus on the fact that both Lippmann (1987) and Hecht-Nielsen (1989) only say that a solution exists for some multilayer feedforward network:- they don't prove that some learning algorithm will reach that solution.

The backpropagation algorithm is based on steepest gradient descent. More powerful Numerical Analysis techniques have been applied more successfully to train multilayer feedforward nets. In §2.4 one of such techniques, conjugate gradient descent, is presented. Nevertheless, scaling up is still a problem as Baum and Lang (1990) and Lang and Witbrock (1988) show.

2.1. The ADALINE and the Delta Rule

The ADALINE (ADAptive LINear Element) is used in this section to illustrate the workings of a single layer net. The ADALINE is selected rather than the Perceptron by Rosenblatt because, although less popular, it provides a more direct path to the understanding of multilayer feedforward networks and the backpropagation rule. The neurons of an ADALINE and a multilayer

feedforward network have a lot in common and the backpropagation rule is a generalisation of the delta rule.

The ADALINE is a network with a layer of neurons and a layer of input units where the input patterns are presented. The neurons are the only processing elements in the net; the input units do not process the inputs they receive. For this reason the input layer is not included in the layer counting and therefore the ADALINE is usually referred to as a single layer network. The input units are connected to the output layer neurons through weighted links, hereafter called weights.

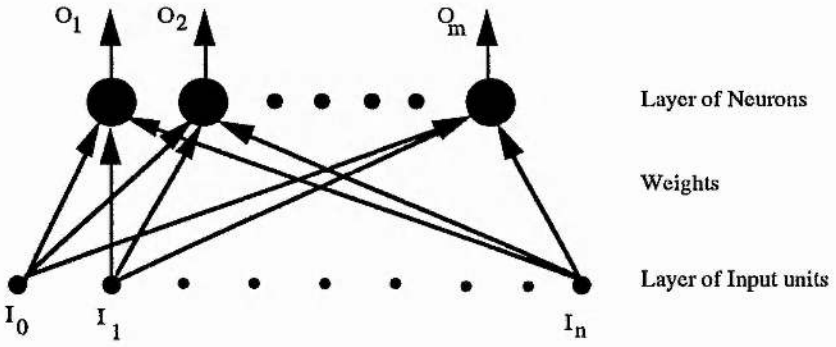


figure 2.1 - The architecture of the ADALINE network

Figure 2.1 depicts a general architecture of the ADALINE network with m neurons and $n+1$ input units. The next figure, 2.2, represents an individual neuron in more detail and is used to describe the functioning of the neuron in the ADALINE network.

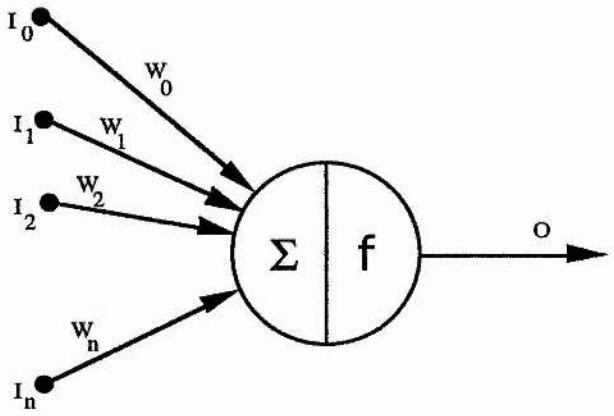


figure 2.2 - The ADALINE network with a single neuron.

An input to the network in figure 2.2 is a vector

$$I = (I_1,...,I_n) \qquad (2.1.1)$$

where $I_1,...,I_n$ are real values. There is an extra input in the input layer, I_0, usually referred to as the bias. The weight linking this extra input to the neuron acts as the adjustable threshold in the Perceptron networks. The extra input is constantly set to 1.

As a matter of notation a pattern is defined in here has having two components: the input vector as defined in (2.1.1), also referred to as the input pattern, and the targets or desired outputs.

The functioning of the net is very simple, the input pattern is presented at the input units, collected by the weights and sent to the neuron. An excitation which is a weighted sum of the inputs present in the input pattern is computed, and afterwards a function, usually referred to as activation function, is applied to this value to produce an output.

The excitation of a neuron when presented with an input pattern I is computed as in §2.1.2,

$$ex = \sum_{j=0}^{n} I_j w_j \qquad (2.1.2)$$

where I_j is the j^{th} component of the input vector and w_j is the weight connecting the input unit j to the neuron. The index j starts at 0 to include the bias input, usually set to 1 as mentioned before. The index n is the number of components of the input pattern.

The activation function f of a neuron, when used for learning in the ADALINE proposed by Widrow and Hoff (1960) is a linear function. Once the network has learned and is put to use, the activation function can be replaced by a threshold function if the problem is a classification one. However, during training the activation function selected must be differentiable.

The output can be for example computed as

$$O = f(ex) = ex \qquad (2.1.3)$$

where ex is as defined in (2.1.2) and O is the output.

Once the output is computed for an input pattern one can compare it with a target output for the respective pattern. If the output obtained is equal to the target output then nothing is done. Otherwise the weights must be adjusted so that a correct output is obtained.

The error for a pattern can be defined as

$$E = \frac{1}{2}(O - T)^2 \qquad (2.1.4)$$

where O is the output obtained for an input pattern using (2.1.3), and T is the target output for the respective pattern. The fraction is introduced to simplify the calculus used elsewhere.

If one considers a net with m neurons in the output layer then the total error becomes

$$TE = \frac{1}{2}\sum_{j}^{m}(O_j - T_j)^2 \qquad (2.1.5)$$

where O_j is the output obtained at output unit j and T_j is the target for output unit j.

The delta rule presents a way to change the weights so that the error decreases. The more general formula for the error presented in (2.1.5) is used here to describe the learning rule. The delta rule uses gradient descent to update the weights after each pattern has been presented to the network and its error computed.

The weights are updated according to

$$\Delta w_{ij} = -\eta \frac{\partial TE}{\partial w_{ij}} \qquad (2.1.6)$$

where w_{ij} is the weight connecting from input unit i to neuron j and η is a real constant called learning rate. To compute the derivative present in (2.1.6) the definition of TE in equation (2.1.5) is used:

$$\frac{\partial TE}{\partial w_{ij}} = -(T_j - O_j)\frac{\partial O_j}{\partial w_{ij}} \qquad (2.1.7)$$

The derivative in the right side of (2.1.7) can be easily computed using the definition of O in (2.1.3), and (2.1.2).

$$\frac{\partial O_j}{\partial w_{ij}} = I_i \qquad (2.1.8)$$

So, collecting the results from (2.1.6) to (2.1.8), the weight update formula becomes

$$\Delta w_{ij} = \eta (T_j - O_j) I_i \qquad (2.1.9)$$

2.1.1. A graphical perspective of the ADALINE

Each input pattern can be represented as a point in Euclidean space. This space has as many dimensions as the number of input components of the pattern. For instance, let us consider a simple problem, the logical AND, with two inputs. A classification problem is selected for this section since the ADALINE was initially conceived as 'an adaptive pattern classifier' (Widrow and Hoff 1960). For simplicity reasons and without loss the targets presented in the example are -1 and 1 instead of 0 and 1 as in the real logical AND problem.

The set of patterns contains four patterns, each has two inputs and a target as described in table 2.1.

Input 1	Input 2	Target
0	0	-1
0	1	-1
1	0	-1
1	1	1

Table 2.1 - logical AND patterns

This set of patterns can be represented graphically as in figure 2.3. The Euclidean space where the patterns are represented is usually referred to as the input space.

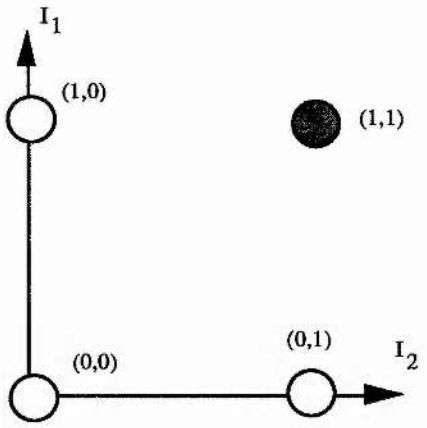


figure 2.3 - The logical AND patterns represented in input space. The black spot indicates that the target is 1, the target is -1 for the remaining patterns.

Since the logical AND is a classification problem there is no need to actually achieve targets of 1 and -1 since what is required is only to separate the outputs for the patterns in one class from the outputs obtained for the patterns in the other class. In the general case, using binary output values of -1 and +1, a pattern is considered to be correctly classified if the pattern's output sign is equal to the sign of the target for the respective pattern.

For a particular weight state there is an input space region in which the patterns will have negative inputs and an input space region where the patterns will have a positive output. Between these two regions are the points which have a 0 output. These latter points obey the following equation,

$$\sum_{j=0}^{n} w_j I_j = 0 \qquad (2.1.10)$$

where w_j is the weight state connecting the j^{th} input unit to the output neuron, I_j is the j^{th} component of the input pattern, and n is the total number of components in the input vector. Note that index j starts at 0 to include the bias term.

Equation (2.1.10) can be reformulated, by separating the bias from the other inputs, as

$$\sum_{j=1}^{n} I_j w_j = -w_0 \qquad (2.1.11)$$

where w_0 is the bias weight. The value I_0, being the bias, is a constant value set to 1 as mentioned before in §2.1 and therefore it has been omitted from the equation.

This equation, (2.1.11), defines a hyperplane in input space. Patterns to one side of the hyperplane have negative outputs while patterns to the other side of the hyperplane have positive outputs. If the bias term was absent from (2.1.10) then the hyperplane would have to go through the origin of the input space and the logical AND problem would be unsolvable using the ADALINE.

Within this context, a weight state in the ADALINE network with a single neuron defines a partition of input space into two regions. A solution weight state can then be seen as defining a partition in input space that realises for each pattern an output with the same sign as its target, i.e. a complete partition. Therefore learning in an ADALINE network can be interpreted geometrically as finding the hyperplane that completely partitions the input space. Figure 2.4 shows an example of a hyperplane that performs a complete partition of input space.

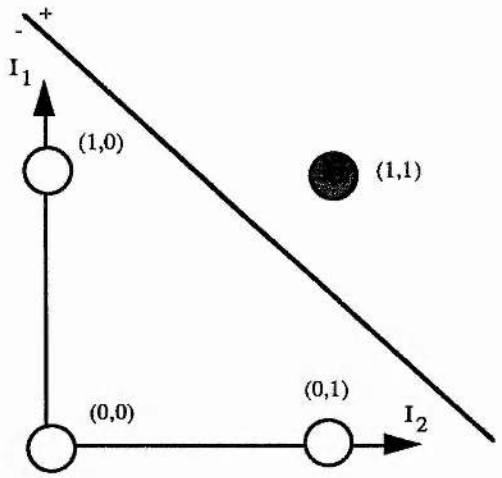


figure 2.4 - A hyperplane dividing the input space.

In this case it is possible to divide the input space in two regions in such a way that in each side of the hyperplane only patterns of the same class are to be found. When such a situation occurs the problem is said to be linearly separable. Both ADALINE and the Perceptron are limited in only being able to solve linearly separable problems. There are though, many problems which are not linearly separable. The logical exclusive-OR, for example, is not a linearly separable problem and therefore can not be solved by either ADALINE or the Perceptron. For a more detailed analysis of the logical exclusive-OR problem refer to §2.2.3.

2.1.2. Conclusion

The ADALINE network is a very simple learning device which was invented originally for pattern classification. It can learn to perform all classification tasks where the property of linear separability is present.

However, simple problems like the logical exclusive-OR are not linearly separable and therefore cannot be solved with an ADALINE network. There is a need for a more powerful architecture.

2.2. Multilayer Feedforward Networks

2.2.1. Architecture Definition

As mentioned before, a single layer network is only capable of solving linearly separable problems. This makes this device a very limited one, simple problems like the logical exclusive-OR cannot be solved using the ADALINE.

If, however, one could use more than one neuron to partition the input space and combine these partitions for input into other neurons then more complex problems could be solved.

Neurons organised in this way are said to form a multilayer perceptron network or multilayer feedforward network. The term feedforward is used because the signal is propagated from the input layer to the output layer without recurrent connections, i.e. connections that form cyclic paths in the network. As a matter of notation, an index is attributed to each layer relative to its position in the network, for instance the input layer is layer 0, the output layer has an index n where n is the total of layers excluding the input layer. In this thesis, it is further assumed that, unless specifically stated otherwise, a unit in layer i connects to all the units in layer i+1 and only to these units.

As in the ADALINE a Bias unit is also present as an extra input unit in the input layer. In a multilayer feedforward network the Bias unit is connected to all the units in the net from layer 1 until the output layer. This unit has a constant output of 1. Note that the Bias unit is an exception to the assumption in the previous paragraph that each unit is connected to all units in the succeeding layer and only to these units.

The layers between the input units and the output layer are called hidden layers. A network can have as many hidden layers as desired.

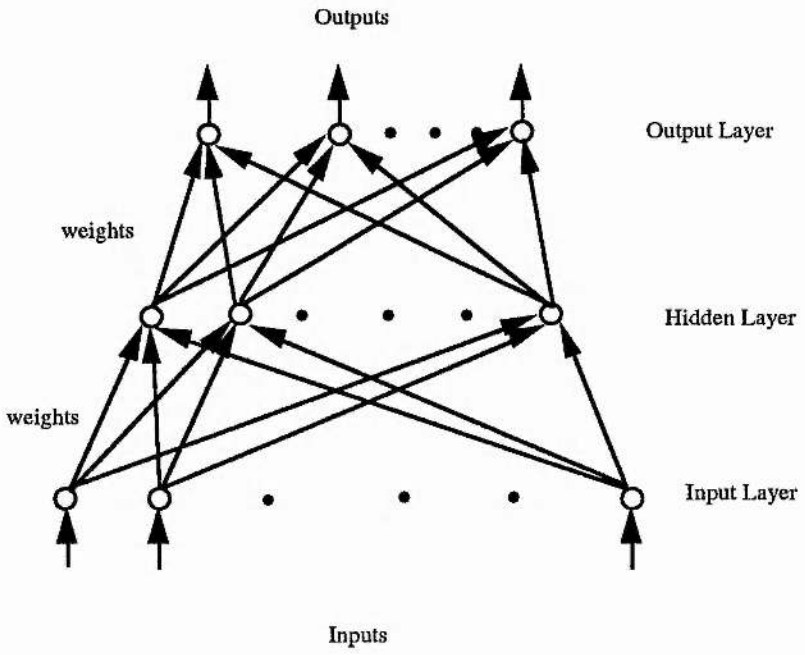


figure 2.5 - An example of a multilayer feedforward network.

In figure 2.5 an example of a multilayer feedforward network with a single hidden layer is presented. Notice that in figure 2.5 the bias unit is not present. This unit must be added as an extra input, set permanently to 1, and is linked to all neurons in the hidden and output layers.

Some authors (Hecht-Nielsen, 1989) include the input layer in layer counting so this would be a three layer network. Others (Hertz, Krogh and Palmer, 1991) count only the hidden layers and output layer so this would be a two layer network. In order to avoid confusion and since the input and output layers have to be present in all networks, only the number of hidden layers will be counted. Hence the net in figure 2.5 is described in this thesis as being a single hidden layer network.

2.2.2. Feeding a Pattern to the Net

In this section the formulas to compute the output of a pattern based on its inputs are presented. Initially a description of a neuron is presented to introduce the formulas needed.

A neuron in a multilayer feedforward network is similar to the ADALINE neuron, the only difference being in the activation function. Rumelhart, Hinton and McClelland (in Rumelhart, McClelland et. al., 1986) explain why a linear activation function for the hidden units will provide no advantage to multilayer feedforward networks when compared with the ADALINE network.

If a multilayer feedforward network uses linear activation functions then a solution weight state would satisfy the following equation:

$$T = I * (W_1 * W_2 * ... * W_n) \qquad (2.2.1)$$

where T is a matrix with rows t_i that stand for the target values for pattern i, I is a matrix where each row is an input pattern, and W_j is the matrix of weights connecting from layer j-1 to layer j. The components w_{jik} from matrix W_j are the weights connecting from neuron i in layer j-1 to neuron k in layer j. The matrix W_1 has k rows where k is the number of input units, the matrix W_n has j columns where j is the number of output units.

Equation (2.2.1) can be transformed into

$$O = I * W \qquad (2.2.2)$$

where

$$W = W_1 * W_2 * ... * W_n \qquad (2.2.3)$$

Since matrix W_1 has k rows and the matrix W_n has j columns, the matrix W will have k rows and j columns.

If a solution exists for (2.2.1) then a solution exists for (2.2.2). However the matrix W in equation (2.2.2) can be seen as the weight matrix for a single layer network with k input units and j neurons in the output layer. Therefore if there is a solution for a multilayer feedforward network with linear activation functions then a solution exists for a network with a single layer with the same number of input and output units. But if a solution exists for a single layer network then the problem must be linearly separable and therefore the capabilities of multilayer feedforward networks using linear activation functions would be restricted to linearly separable problems.

To circumvent this problem, Rumelhart, Hinton and Williams (in Rumelhart, McClelland et. al. 1986) use instead a differentiable semilinear activation function for the neurons in the hidden layer and output layer. For the input units a linear activation function is used, see equation (2.1.3).

The excitation of a neuron is computed as in the ADALINE using equation (2.1.2). The activation function proposed by Rumelhart, Hinton and Williams (in Rumelhart, McClelland et. al., 1986) to compute the output of a neuron is

$$f(ex) = \frac{1}{1 + e^{-ex}} \qquad (2.2.4)$$

where ex is the excitation of the neuron as defined in (2.1.2).

This function represents a sigmoid which produces outputs in the range [0,1] as shown in figure 2.6.

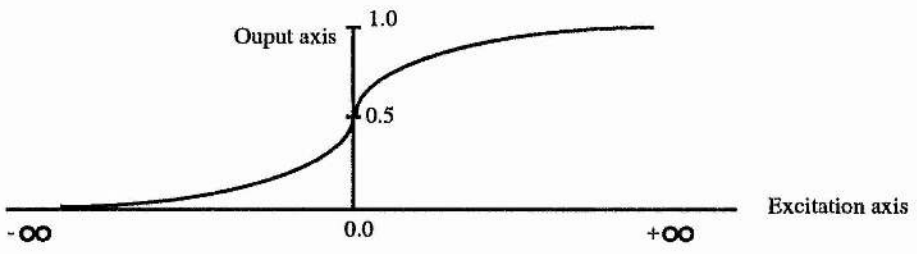


figure 2.6 - Graphical representation of (2.2.4)

Up to this point in this section the formulas to compute the output of a neuron were presented. From this point forward the process of computing the net's output for a given input pattern is presented.

To compute the output of a net when an input pattern is fed in, it is necessary to propagate the input signal throughout the net, layer by layer starting from the input layer. To compute the output of a neuron it is necessary to compute its excitation first. The excitation of a neuron depends on the outputs of the neurons in the previous layer which are connected to it. It is therefore necessary, in order to compute the outputs of the neurons in layer i, to have the outputs from neurons in layer i-1 previously computed. To compute the net's output, i.e. the outputs of the neurons in the output layer, one has to compute first the outputs for the neurons in layer 1, then the outputs for neurons in layer 2, and so on until the output layer is reached.

For instance, in a single hidden layer net like the one in figure 2.5, one has first to compute the outputs of the hidden layer neurons. Only afterwards can the outputs of the neurons in the output layer be computed.

2.2.3. A Graphical Perspective for Multilayer Feedforward Nets

In §2.1.1 it is shown how the ADALINE network solves a linear separable problem, in this section two examples of classification problems which are not linearly separable are presented. These examples will be shown within the context of a general graphical perspective.

First let us analyse the case where the hidden units have threshold output functions. The threshold function used is defined in the following equation.

$$f(ex) = 0 \qquad \text{if } ex \leq 0$$

$$f(ex) = 1 \qquad \text{if } ex > 0 \tag{2.2.5}$$

where ex stands for excitation as defined in (2.1.2).

In §2.1.1 it was mentioned that for the ADALINE network achieving separation in a problem, i.e. obtaining the desired outputs for all patterns, was equivalent to find a complete partition of input space, i.e. a partition in which in each of the regions there were only patterns from one class. In this section the relation between separation and complete partitioning is analysed for multilayer feedforward networks.

The logical exclusive-OR problem is a natural first choice since it is widely known throughout the community as being one of the simplest linearly inseparable problems. The set of patterns for the logical exclusive-OR problem contains four patterns, where each has two inputs and a target as described in table 2.2.

Pattern	Input 1	Input 2	Target
1	0	0	0
2	0	1	1
3	1	0	1
4	1	1	0

Table 2.2 - logical exclusive-OR patterns

Notice that the targets are now set at either 0 or 1 as opposed to -1 and 1 as in the ADALINE, table 2.1. This arises from the output range provided by the activation function described in (2.2.4) which produces outputs only belonging to [0,1]. Therefore, when using targets of 0 and 1 for a classification problem, a weight state can be considered a solution if it provides outputs above 0.5 for patterns with targets of 1 and outputs below 0.5 for patterns with targets of 0. In this sense the 0.5 value for multilayer feedforward nets is equivalent to the value 0 for the ADALINE.

As in the ADALINE case, each pattern can be seen as a point in input space. This set of patterns can be represented graphically as in figure 2.7.

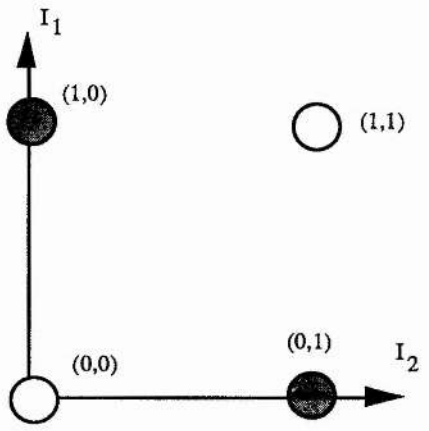


figure 2.7 - logical exclusive-OR patterns in input space. Black points refer to targets of 1, otherwise the target is 0

A network with a single hidden layer is now considered to solve this problem. The network must contain two inputs and an output neuron since that is directly defined by the problem. The main problem here is finding a sufficient number of hidden neurons to solve the problem.

The weights connecting to each neuron in the hidden layer, called a hidden neuron hereafter, define a hyperplane in input space as in the ADALINE case. The set of those hyperplanes, one for each hidden neuron, defines a partition of the input space.

In the exclusive-or case, using a strictly layered architecture, a complete partition of input space can be defined with two hyperplanes as shown in figure 2.8. As mentioned before, by complete partition is meant an input space partition where in each region there are only patterns whose desired output corresponds to one class.

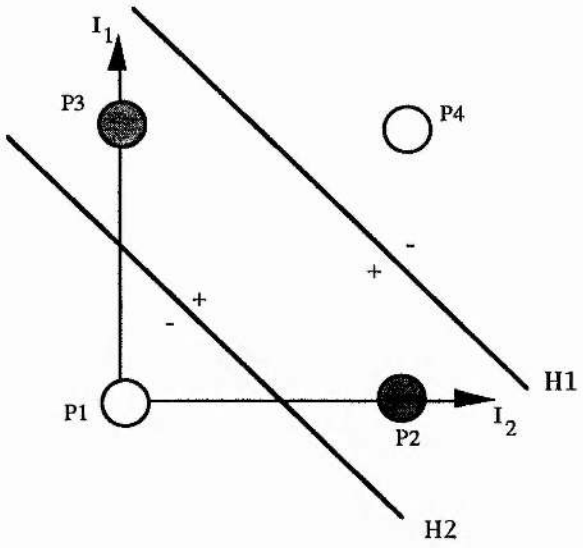


figure 2.8 - A complete partition of input space for the exclusive-OR problem. The labels on the hyperplanes indicate which hidden neuron they refer to. The plus and minus signs close to the hyperplanes indicate if the output is 1 or 0 for the respective hidden unit for patterns on either side of the hyperplane. These signs play no part in the determination of a complete partition.

For the partition in figure 2.8 the following set of outputs are obtained for the hidden units with threshold functions.

Pattern	H1	H2	Target
1	1	0	0.0
2	1	1	1.0
3	1	1	1.0
4	0	1	0.0

Table 2.3 - Outputs for the hidden units using threshold function

If we map the values in table 2.3 in an Euclidean space, see figure 2.9, it becomes clear that it is possible for the output neuron to separate the outputs.

This is because the problem presented to the output neuron is linearly separable.

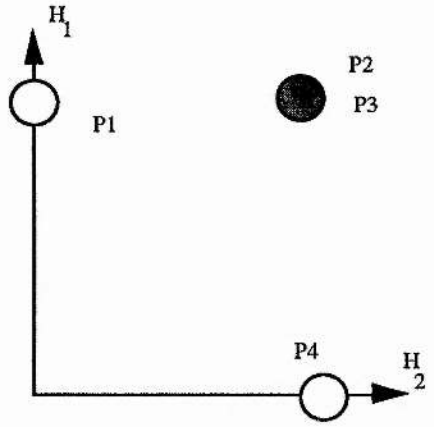


figure 2.9 - Graphical representation of figure 2.3

However, in contrast with the ADALINE case, achieving a complete partition does not imply being able to solve the problem. For instance let us look at the partition on figure 2.10. In this case a complete partition is also present however separation is not possible.

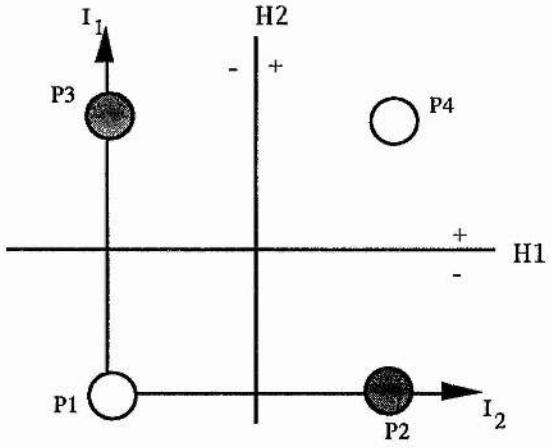


figure 2.10 - A complete partition for the XOR problem

The outputs for the hidden units with threshold functions are presented in table 2.4.

Pattern	H1	H2	Target
1	0	0	0.0
2	0	1	1.0
3	1	0	1.0
4	1	1	0.0

Table 2.4 - Outputs for the hidden units using threshold function base on figure 2.10.

If one compares the outputs from the hidden units with threshold function in figure 2.10 with the inputs of the training set on table 2.2 one can see that the hidden units are doing nothing to solve the problem which remains that of XOR. The outputs from the hidden units with threshold functions are still not linearly separable.

A complete partition through the hidden layer is therefore not sufficient to achieve separation on the output unit.

Is complete partitioning necessary for achieving separation on the output unit? The answer is affirmative, see Nilsson (1965). Assume an incomplete partition, i.e. a partition so that in at least one of the regions there are patterns from both classes. This implies that there will be patterns from different classes with the same hidden unit outputs because all patterns in a region have the same hidden unit outputs and there is at least one region with patterns from different classes. As far as the output unit is concerned there is no difference between patterns in the same region. Therefore a correct classification for all patterns in the training set is impossible without a complete partition.

As a summary it is possible to say that achieving a complete partition with hyperplanes is not a sufficient condition for separability using threshold

hidden neurons. It is however a necessary condition to achieve separation as shown above.

There is however a theorem by Nilsson (1965) that relates more strongly hyperplane partitioning to separation:

> Given P hyperplanes which form a nonredundant[1] partition of two classes with a finite number of elements, if exactly $P+1$ cells[2] formed by the partition are occupied by patterns then the problem is solvable with a single hidden layer architecture using P hidden neurons.

Going back to the XOR problem we can see that the input space partition in figure 2.8 satisfies the theorem conditions because :

- P = 2, i.e. there are 2 hyperplanes and 3 regions occupied;
- The partition is nonredundant, i.e. it is impossible to remove a hyperplane without having patterns from both classes in the same region;
- The training set is finite.

Therefore, and based only on the theorem, it is possible to conclude that the XOR problem is solvable with a single hidden layer feedforward architecture with two threshold hidden neurons. The above theorem provides a useful visual way to check if a certain partition can provide separation with single hidden layer feedforward networks before training in 2-D.

---

[1] A nonredundant partition is a complete partition with the property that if any one of the separating hyperplanes is removed, at least two nonempty regions will merge into one region

[2] Nilsson's cell is the equivalent of a region from a partition

Another example, the chequerboard problem, Weir, Lansley and Clark (1993), is now presented to show a need for using more than one hidden layer. Notice that this problem has an infinite set of patterns, the finite version of the problem will be dealt with later in this section.

The desired input space classification for the chequerboard problem is presented in figure 2.11. Assume that the net used to try to solve the problem is a single hidden layer net with two threshold hidden neurons. Furthermore, assume that the threshold hidden neurons produce the hyperplanes labelled in the figure as H1 and H2, and that the signs presented for each hyperplane indicate if the output is 1 (positive sign) or 0 (negative sign).

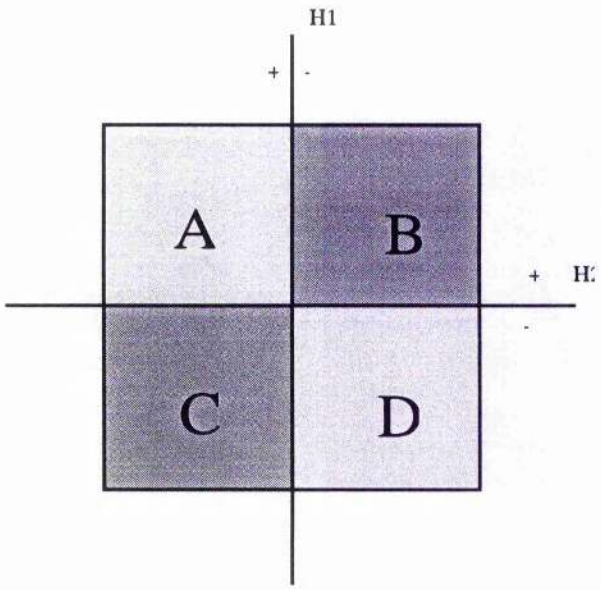


figure 2.11 - A 2-D input space with a chequerboard desired classification. The darker areas represent desired targets of 0 and the lighter ones desired targets of 1. The labels A through D indicate regions of input space. The hyperplanes selected produce a complete partition of input space.

Nilsson's theorem can't be applied to this problem since the training set is infinite and there are $P = 2$ hyperplanes and 4 regions are occupied by patterns, as opposed to 3 as required by the theorem.

However, it can at least be seen that the given single hidden layer is inadequate. The signs either side of each hyperplane indicate if the output is 1 (positive sign) or 0 (negative sign).

Table 2.5 shows the I/O mapping required of the given net's output unit. This mapping, like that of table 2.4, is XOR, even though the sides of the hyperplanes the signs are on are different. In fact it doesn't matter which sides the signs are on, the residual problem for the output unit is linearly inseparable. Hence the net requires a second hidden layer to solve the problem.

Region	H1	H2	Target
A	0	0	0
B	1	0	1
C	0	1	1
D	1	1	0

Table 2.5 - Output for the threshold hidden neurons for the chequerboard problem for the 4 regions present and the region's targets.

The finite version is now analysed. In Nilsson (1965) there is a corollary for his theorem mentioned previously in this section that says that it is always possible to find a single hidden layer feedforward network that separates two finite sets of patterns.

Therefore for the finite version of the chequerboard problem it is possible to find a partition that obeys the conditions on Nilsson's theorem. This implies

that a single hidden layer network can succeed at solving the finite version of the chequerboard problem. However studies by Gibson (1994) suggest that the number of single hidden layer hyperplanes needed for a relatively large and uniformly distributed training set approximating an infinite problem such as the chequerboard problem is much larger than the number of double hidden layer hyperplanes as the infinite limit is approached.

It is therefore preferable to use a two hidden layer architecture, with two hidden neurons in each hidden layer, to solve the chequerboard problem since this architecture is capable of solving the problem completely in theory and more feasibly in practice.

Up to this point only hidden units with threshold output functions were catered for. Now sigmoidal units will be dealt with.

In every problem where the training set is finite a sigmoidal unit can approximate a threshold function to an arbitrarily close degree, i.e. to provide only outputs of 0 or 1 for the patterns in the training set. This is because, since the training set is finite there is a minimum gap between patterns at opposing sides of the hyperplane at the activation function of the unit.

Let us suppose that patterns A and B are those which define the minimum gap. Since A is on the opposing side of the hyperplane to B, then the activations for the patterns must have opposite signs as well. By scaling up the weights connecting to any unit the absolute value of the excitation can be made arbitrarily large, which means that the sigmoidal outputs for these two patterns will be either close to 1 if the excitation is positive, or 0 if the excitation is negative. All other patterns which are further away from the hyperplane will also have outputs of effectively 0 or 1 after scaling depending on the respective sign of the excitation because their absolute

excitation values will be larger than those for the patterns which are closest to the hyperplane.

So all problems using finite training sets which can be solved with threshold function can also be solved in theory by sigmoidal units with the same topology.

The XOR problem is now revisited to show one of the differences between sigmoidal or threshold functions.

Earlier in this section it was mentioned that complete partitioning was necessary to achieve separation on the output unit. This is true for threshold units as shown above. However when using sigmoidal units there is no need to have a complete partitioning.

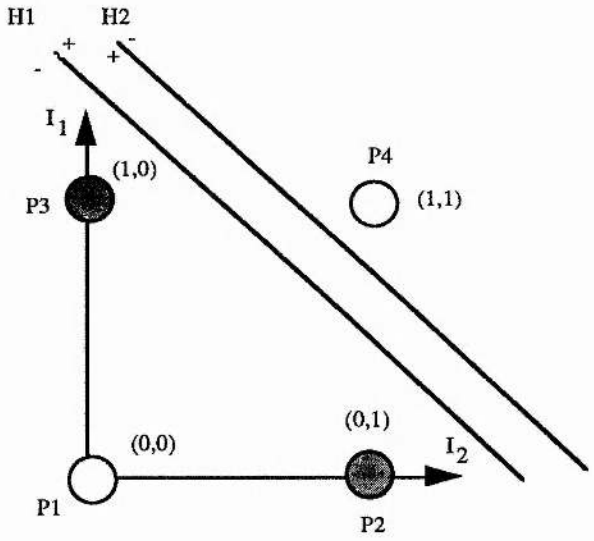


Figure 2.12- An incomplete partition for the XOR problem. The use of sigmoidal activation functions means that the plus and minus signs close to the hyperplanes indicate if the output is below or above 0.5 for the respective hidden unit for patterns on either side of the hyperplane as opposed to 0 outputs of 0 or 1.

Looking at figure 2.12 it is clear that it represents an incomplete partition. Table 2.6 presents a set of possible output values for the hidden neurons according to figure 2.12.

Pattern	H1	H2	Target
1	0.05	0.75	0.0
2	0.45	0.60	1.0
3	0.45	0.60	1.0
4	0.60	0.45	0.0

Table 2.6 - A set of possible output values for the hidden neurons from an incomplete partition for the XOR-problem combined with their targets.

Mapping these values on a Euclidean Space, see figure 2.13, shows that separation is achievable even when the hyperplanes do not form a complete partition as in the example depicted in figure 2.12.

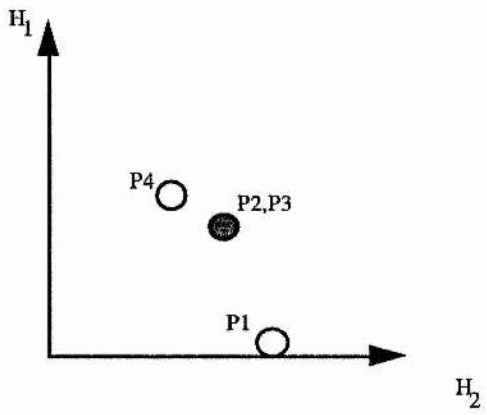


figure 2.13- Graphical representation of table 2.6

As a summary it is possible to say that achieving a complete partition with hyperplanes is not a necessary condition for separability using sigmoid activation neurons as opposed to threshold units where complete partitioning is necessary.

The chequerboard problem is now revisited in its infinite version. As for the threshold functions there is no evidence that there is a solution using a single hidden layer. There is an approximate solution using two layers by approximating threshold functions with the sigmoidal activation functions.

The finite version however can be solved both using a single hidden layer or a two hidden layer architecture. Nevertheless the objections raised for threshold functions concerning the number of hyperplanes remain for the single hidden layer architecture with sigmoidal functions.

Based on Nilsson's corollary to his theorem one can conclude saying that all finite problems are theoretically solvable by both threshold or sigmoidal functions using single hidden layer networks. However in some cases it is probably better to consider a second hidden layer to save on the number of hidden units.

2.3. Standard Backpropagation

2.3.1. The Backpropagation Rule

Once a pattern is fed to the network then its output can be compared to its target and an error generated. If this is done for all patterns then an overall error is obtained. The backpropagation rule establishes a way of computing how each weight should be changed to decrease the overall error. The rule is based on steepest gradient descent, hence the weight transitions are proportional to the partial derivatives of the error function with respect to the weights.

$$\Delta wij = - \eta \frac{\partial E}{\partial wij} \qquad (2.3.1)$$

where η is some real constant, w_{ij} is the weight connecting unit i to unit j and E is the overall error. The overall error, E, is defined as being the sum of the individual errors found for each pattern.

$$E = \sum_{r} E_r \qquad (2.3.2)$$

in which E_r stands for the error computed for pattern r, with r ranging over the number of patterns. The error for each pattern, E_r, is defined as being the sum of the errors found for each output unit.

$$E_r = \sum_{k} E_{rk} \qquad (2.3.3)$$

in which E_{rk} stands for the error found for pattern r on the k^{th} neuron of the output layer. E_{rk}, is defined by equation (2.3.4).

$$E_{rk} = \frac{1}{2} (t_{rk} - o_{rk})^2 \qquad (2.3.4)$$

in which t_{rk} stands for the target defined for the k^{th} component of the output pattern for pattern r, o_{rk} stands for the output obtained at output neuron k when pattern r was fed to the network.

The partial derivatives in equation (2.3.1) can be rewritten using equation (2.3.2).

$$\frac{\partial E}{\partial w_{ij}} = \sum_{p} \frac{\partial E_p}{\partial w_{ij}} \qquad (2.3.5)$$

Although this equation can be solved directly it is simpler to partition the calculus on the right side into smaller expressions using the chain rule. In this thesis this is done following the presentation by Rumelhart, Hinton and Williams (in Rumelhart, McClelland et. al., 1986).

Applying the chain rule to each of the elements of the sum on the right side of equation (2.3.5) will produce (2.3.6).

$$\frac{\partial E_r}{\partial w_{ij}} = \frac{\partial E_r}{\partial ex_{rj}} \frac{\partial ex_{rj}}{\partial w_{ij}} \qquad (2.3.6)$$

where E_r is the error obtained for pattern r, and ex_{rj} is the excitation for pattern r at unit j. The excitation of neuron r for pattern j, ex_{rj}, is defined as the weighted sum of the outputs units connected to neuron r, see (2.3.7).

$$ex_{rj} = \sum o_{ri} * w_{ij} \tag{2.3.7}$$

where o_{ri} represents the output from unit i for pattern r and w_{ij} stands for the weight linking unit i to unit j.

The first term establishes how the alteration of the activation of an unit affects the error, whereas the second term refers to how the change in the weights themselves affect the excitation level of a unit.

The second term of the right side of equation (2.3.6) can now be easily computed.

$$\frac{\partial ex_{rj}}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \sum_h w_{ih}\, o_{rh}$$

$$= \sum_h \frac{\partial w_{ih}}{\partial w_{ij}}\, o_{rh}$$

$$= o_{rj} \tag{2.3.8}$$

For the moment let us define a new variable for each unit.

$$\delta_{rj} = - \frac{\partial E_r}{\partial ex_{rj}} \tag{2.3.9}$$

The weight change proposed with backpropagation can then be computed as

$$\Delta w_{ij} = \sum_p \eta * \delta_{pj} * o_{pi} \tag{2.3.10}$$

In the following equations (2.3.11) to (2.3.16) the computation of δ for each unit will be shown. To compute the right side of equation (2.3.9) the chain rule is used again to simplify the calculus.

$$\delta_{rj} = -\frac{\partial E_r}{\partial o_{rj}} \frac{\partial o_{rj}}{\partial ex_{rj}} \tag{2.3.11}$$

The second term of the right side of (2.3.11) can be computed based on equation (2.2.4) which defines the activation function.

$$\frac{\partial o_{rj}}{\partial ex_{rj}} = f'(ex_{rj}) \tag{2.3.12}$$

All that remains is the computation of the first term of the right side of equation (2.3.11). If the unit j is an output unit then the derivative is straight forward based on equations (2.3.3) and (2.3.4).

$$\frac{\partial E_r}{\partial o_{rj}} = -(t_{rj} - o_{rj}) \tag{2.3.13}$$

If the unit j is an output unit then δ_{rj} can be defined as in (2.3.14).

$$\delta_{rj} = f'(ex_{rj}) * (t_{rj} - o_{rj}) \tag{2.3.14}$$

However, if the unit is a hidden unit then the chain rule needs to be applied again.

$$\begin{aligned}\frac{\partial E_r}{\partial o_{rj}} &= \sum_h \frac{\partial E_r}{\partial ex_{rh}} \frac{\partial ex_{rh}}{\partial o_{rj}} \\ &= \sum_h \frac{\partial E_r}{\partial ex_{rh}} \frac{\partial}{\partial o_{rj}} \sum_i w_{ik} * o_{ri} \\ &= -\sum_h \delta_{rh} * w_{jh}\end{aligned} \tag{2.3.15}$$

where the variable h covers all units to which unit j is connected.

Joining the results from equation (2.3.15) and (2.3.12) the δ for hidden units can now be computed as in equation (2.3.16)

$$\delta_{rj} = f'(ex_{rj}) * \sum_{h} \delta_{rh} * w_{jh} \qquad (2.3.16)$$

Equation (2.3.17) shows the derivative of the activation function in §2.2, equation (2.2.4).

$$f'(ex_{rj}) = ex_{rj} (1 - ex_{rj}) \qquad (2.3.17)$$

All the formulae needed to compute a weight transition in weight space according to the standard version of backpropagation have been presented.

It is clear from equation (2.3.16) that first one has to compute the δ for the output units. A hidden unit can only have its δ computed once all the units it connects to have their δ's computed. In other words, the δ's are back-propagated through the net from the output units, hence the name of the algorithm.

Two possibilities were presented by Rumelhart, Hinton and Williams (in Rumelhart, McClelland et. al., 1986) for computing a weight change. One updates the weight state after the presentation of each pattern, the other sums the proposed updates from each pattern and only performs a transition in weight space when the whole set has been presented to the net. The former is known as the online mode and the latter as the batch mode. Both are a generalisation of the delta rule used by Widrow and Hoff (1960) in the ADALINE.

As a summary, the relevant formulae needed to update the weights in a multilayer feedforward network are condensed here.

For the online mode the weights are updated after each pattern's presentation. Therefore the weight connecting neuron i to neuron j is updated according to

$$w_{ij} = w_{ij} + \Delta w_{ij} \tag{2.3.18}$$

where

$$\Delta w_{ij} = \sum_{p} \eta * \delta_{pj} * o_{p\,i} \tag{2.3.19}$$

where δ_{pj} is defined as

$$\delta_{rj} = f'(ex_{rj}) * (t_{rj} - o_{rj}) \tag{2.3.20}$$

if neuron j is in the output layer, or

$$\delta_{rj} = f'(ex_{rj}) * \sum_{h} \delta_{rh} * w_{j\,h} \tag{2.3.21}$$

if neuron j is in a hidden layer.

For the batch mode the weights are updated only after the set of patterns is presented to the net. After each pattern's presentation Δw_{ij} is computed. The weight connecting neuron i to neuron j is updated after the last pattern has been presented according to the formula

$$w_{ij} = w_{ij} + \sum_{k} \Delta w_{kij} \tag{2.3.22}$$

where Δw_{kij} is the weight's change computed using (2.3.19) after pattern k is presented to the net.

2.3.2. A Graphical Perspective of Backpropagation

In this section it is explained graphically how backpropagation works in the batch mode. This mode is selected because then the backpropagation rule is equivalent to steepest gradient descent.

The output provided for each pattern depends upon the inputs and the weights. The inputs are fixed for each pattern, hence, in essence, the outputs depend only upon the weights.

Since the error is directly related to the output it is possible to construct an individual error/weight surface for each pattern where for each weight state there is an error associated with it. If one thinks of the error as a height measure then the lowest point in each individual surface is the one with lowest error for the respective pattern.

Furthermore, for any network there is at least one weight state that has zero error for any particular pattern. For instance, let us consider an arbitrary net and set all the weights to zero except those which connect the bias input to the output units. Then, for any particular pattern, the weights connecting the bias input unit to the output units can be set so that the outputs obtained are equal to the targets for that pattern.

These individual error/weight surfaces have been found empirically to be very simple for steepest gradient descent, i.e., no local minimum has ever been reported and a global minimum is usually found in a few iterations with standard backpropagation. Weir and Chen (1990), also report the same findings.

An overall error/weight surface can be built through superposition of the individual error/weight surfaces. It is in this surface that backpropagation travels in the batch mode. However the superposition of many individual surfaces, no matter how simple they are, is bound to originate a very poor surface for steepest gradient descent to find a global minimum in a feasible amount of time. In fact steepest gradient descent only points to the goal when the surface is a circular bowl. The performance of this technique deteriorates

rapidly when the surface to which it is being applied differs from the circular bowl.

Ravines, or narrow valleys, are a common feature in overall error/weight surfaces. Silva and Almeida (1990), for example, point out the tendency for steepest gradient descent to oscillate between the sides of ravines as a cause for poor convergence rates.

Local minima, defined here as regions in weight space in which to escape from them one has to increment the error, and flat plateaux, defined here as regions with zero derivatives in every direction, add further obstacles for backpropagation to find a global minimum of the overall error/weight surface.

There are many claims that local minima in these surfaces are rare, see for instance Hecht-Nielsen (1989) and Rumelhart, Hinton and Williams (in Rumelhart McClelland et. al., 1986). In practice though, an empirical criterion of failure to converge within a certain amount of time is more useful than stopping only when a minimum, local or global, has been reached to evaluate an algorithm's performance. In this sense, and for the reasons mentioned above, the performance of backpropagation in overall error/weight surfaces has been found to be very poor, see Minsky and Papert (1988) and Baum and Lang (1990).

If the online method is used then the weights are updated after each pattern's presentation. This implies that for each update, the gradient is computed only for a pattern's individual error/weight surface. As mentioned before the individual error/weight surfaces are very simple even for steepest gradient descent. However, the goal remains the same, to achieve the global minimum of the overall error/weight surface.

Backpropagation in the online mode doesn't follow the steepest gradient on the overall error/weight surface. Nevertheless Rumelhart, Hinton and Williams (in Rumelhart McClelland et. al. 1986) point out that when using a small enough learning rate, the departure from the steepest gradient will be negligible. Hence the criticism of the batch mode stands for the online mode with a small learning rate.

When using large learning rates with the online mode the weights' update for the last pattern presented will tend to spoil whatever has been achieved for the remaining patterns. Consequently, this mode tends to create an unfeasible zig-zagging path which goes from a weight space region which satisfies the last pattern presented to a region of weight space that satisfies the current pattern being presented.

2.3.3. Momentum

The use of momentum was proposed initially by Rumelhart, Hinton and Williams (in Rumelhart, McClelland et. al., 1986) in order to speed up backpropagation.

A momentum term α can be add to the weight update equation (2.3.19)

$$\Delta w_{ij}(t+1) = \sum_{p} (\eta * \delta_{pk} * o_{pi}) \quad + \quad \alpha \, \Delta w_{ij}(t) \qquad (2.3.23)$$

where the constant α determines by how much the last iteration at time t influences the new weight change at time $t+1$. This new equation specifies that the weight update is proportional to the steepest gradient descent vector from the current weight state plus a fraction of the last update. Rumelhart, Hinton and Williams (in Rumelhart, McClelland et. al., 1986) set α to be 0.9 for most of their simulations.

Momentum can be seen as having an accelerating effect if the direction computed at time t agrees with the direction at time $t+1$, and as having a damping effect otherwise.

The inclusion of a momentum term speeds up considerably the learning process. However, backpropagation with momentum still fails to provide satisfactory results when applied to large problems, see Baum and Lang (1990), and Minsky and Papert (1988).

2.3.4. Conclusion

Backpropagation in the batch mode is based on steepest gradient descent in the overall error/weight surface. Superposition can originate very nasty surfaces for steepest gradient descent. For instance, ravines and flat plateaux are reported as being a common feature present in overall error/weight surfaces. Although many authors claim that local minima are not a problem theoretically, in practice the backpropagation algorithm can take an unfeasible amount of time to converge when applied to anything but toy problems.

In the online mode the weights are updated after each pattern's presentation, therefore it doesn't follow exactly the steepest gradient on the overall error/weight surface. However, with a small enough learning rate, the departure from the steepest gradient will be negligible. Hence the criticism for the batch mode stands for the online mode with a small learning rate.

If a large learning rate is used with the online mode then there is a tendency to create an unfeasible path which zig-zags between the weight space regions that satisfy individual patterns.

The inclusion of momentum speeds up backpropagation but the improved algorithm still fails to converge to an existing solution in large problems.

2.4. Conjugate Gradient Descent

Without the momentum term, if line search is done for each iteration along the steepest gradient descent method until a minimum is found, then the new direction has to be orthogonal to the previous one (Press et al., 1992). Including the momentum term and still doing the line search, a compromise is being made for the new direction to be between the old and the orthogonal directions.

Even in a simple surface it is clear that the steepest gradient descent method is not a very practical idea. For instance, consider a 2-D quadratic surface. In such a surface the method will, in the general case, perform many small steps to reach the global minimum. Momentum helps to decrease the number of iterations but it still takes a large number of iterations.

Conjugate gradient descent methods achieve the global minima in a quadratic surface in n dimensions in at most n iterations. This represents a clear improvement over both steepest gradient descent and the momentum variation techniques. This performance is achieved through the use of n mutually conjugate directions.

Considering a direction taken at time t, $\Delta w(t)$, the direction at time $t+1$, $\Delta w(t+1)$, is conjugate to the direction at time t if

$$\Delta w(t) * H * \Delta w(t+1) = 0 \qquad (2.4.1)$$

where H is the Hessian matrix of the error function in (2.3.2).

Press et. al. (1992) say, in an attempt to define the conjugate direction concept intuitively, that the direction taken at time $(t+1)$ is conjugate if it spoils as little as possible the minimisation along the direction taken at time(t).

A conjugate gradient descent method is described in (2.4.2) where line search is done for each direction computed until a minimum is reached.

$$\Delta w(t+1) = - \frac{\partial E}{\partial W} + \beta \Delta w(t) \qquad (2.4.2)$$

for some appropriate β.

The above equation, (2.4.2), has the same form as the momentum equation in (2.3.23) the difference being that in the latter case the variable β is made constant for all values of t.

The Polak-Ribiere rule (in Polak 1971) determines the coefficient β so that the directions obtained between time $(t+1-n)$ and $(t+1)$ are all mutually conjugate. Equation (2.4.3) achieves this without requiring the knowledge of H.

$$\beta = \frac{(\nabla E(t+1) - \nabla E(t))\, \nabla E(t+1)}{(\nabla E(t))^2} \qquad (2.4.3)$$

where

$$\nabla E(t) = \frac{\partial E}{\partial W(t)} \qquad (2.4.4)$$

where $W(t)$ is the current weight state at time t.

As mentioned before in this section, on a strictly quadratic surface in n dimensions the Polak-Ribiere rule will converge in at most n steps. If the surface is not quadratic then, in the general case, the minimum will not have been achieved in the first n steps. Nevertheless, conjugate gradient descent methods should perform better than steepest gradient descent for irregular surfaces because they use more information to compute each new direction.

Johansson, Dowla and Goodman (1992) tested several conjugate gradient descent methods and show that conjugate gradient methods outperform backpropagation in various problems.

The results obtained by the method used for this thesis are consistent with those reported in the literature. For instance, Smagt (1994) reports a success rate for the XOR problem with initial weight states randomly picked from [-1,1] of 82.1%. Webb and Lowe (1988) report a convergence rate of 67.4% for the same initial problem. With the thesis method a percentage between these two results, 75.4, was obtained. As for the average number of epochs needed to achieve convergence, Smagt reports 79.2, Webb reports 40, and the thesis result is 70.5. The differences in the values reported might be attributed to different line search techniques, as well as to the fact that Smagt uses Powell's restarts which were not used in the thesis method. According to Smagt (1994) the use of Powell's restarts can improve convergence success.

The values obtained for the thesis method with a smaller initial weight interval, [-0.1,0.1], are relatively worse than those for the [-1,1] interval which shows that CGD can also be affected by initial conditions. The convergence rate dropped to 57.1% and the average number of epochs went up to 167.75.

A function approximation problem was also tested in this thesis. This problem is a modified version of a problem in Smagt's 1994 paper. The modification consists of squashing the outputs within the range 0.05, 0.95, instead of the original -1,1 range as used in Smagt. As mentioned in §2.2 the output function used in this thesis only produces outputs between 0 and 1. The stopping criterion is also different from Smagt's. In this thesis a maximum linear error per pattern of 0.025 was used as a stopping criterion, whereas in Smagt's paper an average square error of 0.025 was used. The criterion used in this thesis is therefore much more demanding. Nevertheless in both versions of the problem, the CGD techniques proved to be very robust achieving 100% of success. The number of epochs is different but this is to be expected. Smagt reports an average below 2500 and the average obtained using the method for

the thesis is 10072.71. The difference can be justified in two ways: the more demanding stopping criterion and the different output functions.

What will be important for the thesis is that although conjugate gradient methods lead to faster training than backpropagation, they still have robustness problems. This has been described above for the XOR problem. Furthermore, Baum and Lang (1990) still report that no convergence was ever achieved when they applied conjugate gradient descent to a specific problem with a predetermined architecture for which a solution exists, i.e. the 2 spirals problem, see §5.3. This implies that the overall error/weight surfaces are in fact too complex even for conjugate gradient descent methods.

2.5. Other Minimisation Techniques

Other minimisation methods have been applied to train Neural Nets. These include the Quasi-Newton and Levenberg-Marquardt methods, see Press et. al. (1992).

In conjugate gradient descent methods implicit use was made of the Hessian matrix. Quasi-Newton methods make explicit use of the Hessian matrix.

Quasi-Newton's methods are based on Newton's method. The weight transition in Newton's method is computed as in equation (2.5.1).

$$\Delta W = H^{-1} * G \qquad (2.5.1)$$

where H is the Hessian matrix and G is the gradient vector.

In a quadratic surface the weight transition presented in equation (2.5.1) points directly at the minimum with the appropriate step size.

Newton's method requires the computation of the Hessian matrix and its inverse. Quasi-Newton methods are alternative approaches which compute an approximation to the inverse Hessian over a number of steps. As with

conjugate gradient descent, Quasi-Newton methods find the minimum of a quadratic surface in at most n steps, where n is the number of dimensions.

Quasi-Newton methods have been tested by Webb et al (1988), Møller (1993) and Smagt (1994). The results by Webb et al (1988) suggest that in some problems Quasi-Newton techniques such as David-Fletcher-Powell (DFP) or Broyden-Fletcher-Goldfarb-Shanno (BFGS) perform better than Conjugate Gradient Descent techniques. In other problems it is Conjugate Gradient Descent which performs best, see Smagt (1994) and Møller (1993).

The Levenberg-Marquardt technique is a hybrid between steepest descent and an inverse-Hessian method. For an implementation of this technique for multilayer feedforward networks see Webb et al (1988). The main idea behind the technique, as implemented by Webb et al, is that the surface tends to be more quadratic as the goal region is approached. The direction obtained with the Levenberg-Marquardt algorithm when the error is repeatedly high is close to that obtained with steepest descent. As the error decreases, the direction becomes closer to the one obtained with the inverse Hessian method. A potential problem with this method is the fact that having a repeatedly small output error does not necessarily mean that goal weight state is near the current weight state. It is possible to have a weight state with low error with the surface between it and the goal being far from quadratic. Also, if the error takes too long to decrease to a significantly small value then steepest descent will have an excessively large influence on the direction for a large number of epochs.

2.6. Conclusion

Neural nets have come a long way since the McCulloch and Pitts nets. However, as Minsky and Papert pointed out in their revised edition of

Perceptrons, some of the criticism initially directed at the perceptrons and single layer nets in general still applies to multilayer feedforward nets.

Multilayer feedforward nets are theoretically much more powerful than single layer nets. They can solve problems that single layer nets can't, at least from a theoretical point of view.

In practice though, things turn out still to be frustrating in the sense that the existing algorithms used to train multilayer feedforward nets can still take a large amount of time to train real world problems.

Baum and Lang (1990) tried to solve the two spirals problem (§5.3) with a single hidden layer net with 2 inputs, 50 hidden units and one output unit using both backpropagation with momentum and conjugate gradient descent methods and in both situations they failed to obtain a solution although a solution exists. The same authors report that conjugate gradient descent was unable to find a solution even when the network was expanded to include 10 additional hidden units.

Lang and Witbrock (1989) confirm these results since they also report failure to achieve a solution with single hidden layer architectures. Lang and Witbrock (1989) did manage to achieve a solution with a three hidden layer architecture with connections between a layer and all succeeding layers using variable learning rates. When experimenting with a two hidden layer architecture they concluded that backpropagation with momentum wasn't robust enough to provide a low failure rate.

The 2 spirals problem is really only a bridge between toy problems and real world problems. Consequently, these results illustrate that the overall error/weight surfaces for multilayer feedforward nets may be likely to be too tortuous for either backpropagation or conjugate gradient descent in real world problems as well. In short, these techniques would appear to not be

robust enough for reliably and straightforwardly training multilayer feedforward networks in real world problems.

Quasi-Newton methods seem to perform better than Conjugate Gradient descent methods on some problems, see Webb et al (1988). However, on other problems the inverse situation occurs, i.e. Conjugate Gradient Descent performs best, see Møller (1993) and Smagt (1994). This suggests that Quasi-Newton methods too are not robust across a variety of problems.

Excluding steepest descent, all the methods above attempt to make progress by assuming to some degree that the surface is quadratic in selecting the next transition. This assumption is only made for weight states where the error is repeatedly low in the implementation of the Levenberg-Marquardt technique by Webb et al (1988). The Levenberg-Marquardt technique tries to combine steepest descent and quadratic descent at the most appropriate stages. However the assumptions about the overall surface shape are still likely to be limited in providing robust transitions. This is because the overall error-weight surface is highly irregular.

The results for the 2 spirals problem suggests that these approaches which rely on a particular shape of the surface may not be robust for all problems.

3. Tangent Hyperplanes

3.1. An Alternative View of Goal Weight States

The common view of a goal weight state is as a global minimum of an overall error/weight surface. The overall error/weight surface is built through superposition of the individual surfaces. However superposition for large problems is bound to create very irregular surfaces in the neural case (§2.3.2).

Finding a global minimum of the overall error/weight surface with various gradient descent based methods has been proven unfeasible in many reports, for instance see Baum and Lang (1990), and Lang and Witbrock (1988).

The approach taken here is to look at the individual error/weight surfaces and try to combine them in a more fruitful way. For each individual error/weight surface there is a set of weight states that have zero error. Such a set will be referred to as a solution manifold. A solution manifold for a pattern can be described in terms of individual error/weight surfaces as being the error/weight contours that have zero error associated with them.

The alternative view presented here is of a goal weight state as being a point in weight space which is closest to all solution manifolds in Euclidean distance terms. The solutions for both approaches coincide in the exact solution case, see figure 3.1. In this case there is a common intersection to all the solution manifolds producing a global minimum of zero error in the overall error/weight surface. Any point that belongs to the common intersection has a null distance to all the solution manifolds and therefore is also a solution weight state for the solution manifold view.

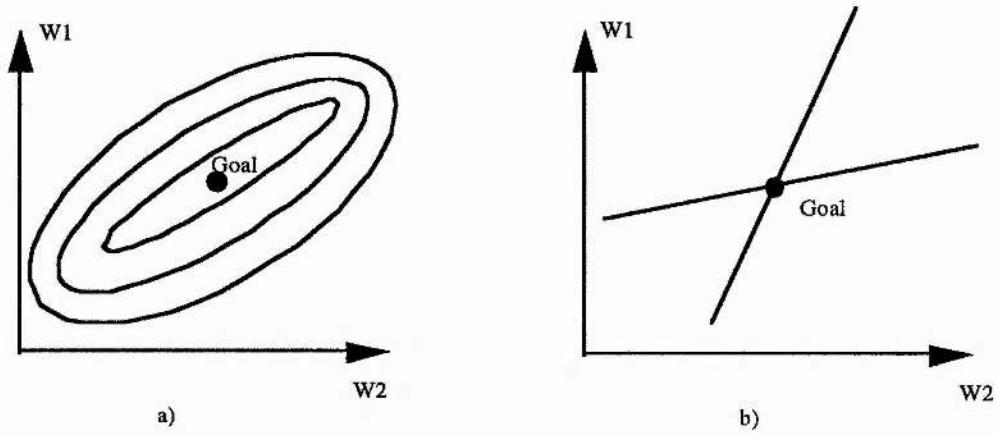


figure 3.1 - shows two views of a goal state in weight space. In a), the classical view of the goal as the minimum of the overall surface is presented. The ellipses represent idealised error/weight contours, the outer ellipses have higher error than the inner ones. In b), the goal is presented as being the intersection of the solution manifolds, these are idealised as straight lines.

In the inexact solution case the solution for the alternative solution manifold approach is the set of weight states that minimise the Euclidean distance to all solution manifolds.

The solutions for two approaches, the global minimum and the solution manifold view, may differ in the inexact solution. The global minimum does not necessarily coincide with the point which is closest to all the solution manifolds in cases where the error/distance ratio is not linear, i.e. the closest point to all the solution manifolds does not necessarily minimise the sum of squared output errors as defined in §2.3.1. However in the general case of multi-layer feedforward nets the difference between the two approaches has been found empirically to typically be negligible in the output solution states obtained. For the cases where the differences may be significant, an analysis will be done later in chapter 4.

3.1.1. The Solution Manifold View for Single Layer Nets

In single layer nets (§2.1) the solution manifolds are linear. This will be shown in §3.1.1.1. Also, in this case, steepest gradient descent along the individual error/weight surfaces produces a vector with an accurate direction in the sense that it is orthogonal to the respective solution manifolds. However, steepest gradient descent along the overall error/weight surface produces a proposed weight jump that has a direction and step size which are arbitrary. Arbitrary in the sense that, in the general case, the vector produced by steepest gradient descent for the overall error/weight surface does not have an accurate direction or magnitude for a goal weight state, i.e., the weight space transition does not update the current weight state to coincide with or move in the direction of a goal weight state. An example is presented in figure 3.2. This is because the steepest gradient descent direction does not coincide with the goal direction in all but the cases of linear or circular error/weight contours.

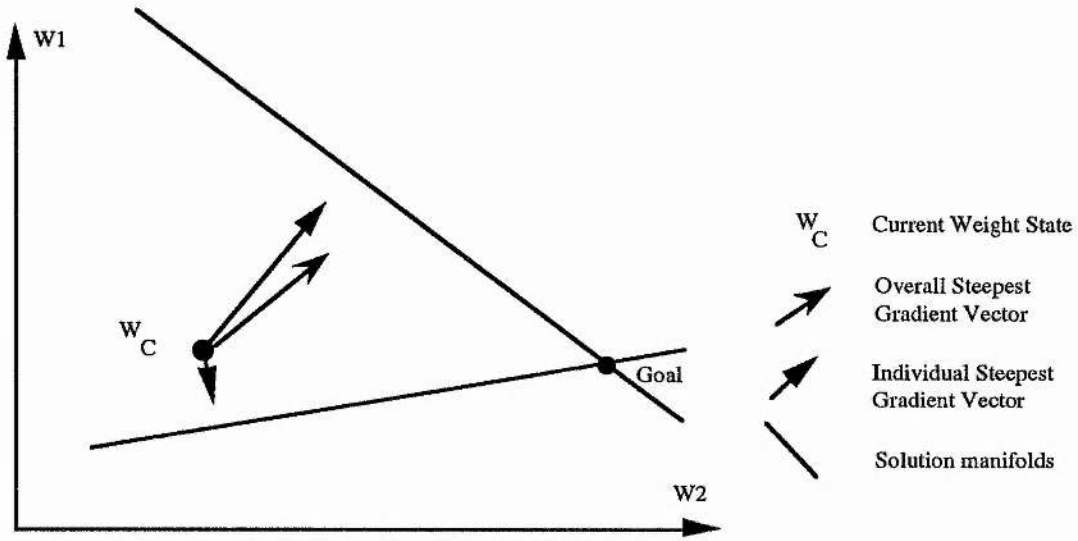


figure 3.2 - The overall steepest gradient, being the sum of individual steepest gradients for individual surfaces, produces poor predictions of the whereabouts of the goal.

As mentioned before, in the solution manifold view a goal is considered to be the closest point to all solution manifolds. Since the solution manifolds are linear it is possible to use linear system techniques to solve the problem without resorting to minimisation techniques. This enables the user to get a solution in one step.

In section 3.1.1.1 the problem of finding a goal for the solution manifold view is formalised. A version of least squares to solve the problem is presented in 3.1.1.2. Finally conclusions are drawn in 3.1.1.3.

3.1.1.1. Minimising Euclidean Distance to a Set of Solution Manifolds

The problem can be summarised informally as follows :

> given a set of solution manifolds find a point which is closest to all solution manifolds in Euclidean distance terms.

If one considers a single layer net with a single output unit then the equation of the solution manifolds for pattern i is

$$\sum_j d_{ij} w_j = c_i \qquad (3.1.1)$$

where d_{ij} is the j^{th} component of the input pattern i, w_j is the weight that multiplies the j^{th} component of the input pattern into the neuron and c_i is constant. Note that the bias input, see §2.1, is considered to be component index 0 of every input pattern.

If the output function being used is linear then c_i is the target for pattern i. If another 1-1 output function is used then c_i is defined as

$$c_i = f^{-1}(t_i) \qquad (3.1.2)$$

where t_i is the target for pattern i and f^{-1} is the inverse of the activation function (see §2.1).

Equation (3.1.1) defines a solution manifold as a hyperplane in weight space. The Euclidean distance r_i from a weight state W to the solution manifold for pattern i is therefore defined by equation (3.1.3).

$$r_i = \frac{\left| \sum_j d_{ij} w_j - c_i \right|}{\sqrt[2]{(\sum_j d_{ij}^{\,2})}} \qquad (3.1.3)$$

A solution weight state is one that minimises the sum of the distances r_i defined in equation (3.1.3) for all patterns.

The problem can now be stated more formally as being :

$$\text{minimise} \sum_i r_i^{\,2} \qquad (3.1.4)$$

where the square is introduced to simplify the transformation of this problem into one of least squares as shown in the next section.

3.1.1.2. A Least Squares Approach to Minimise Euclidean Distance

The set of hyperplanes can be written in matrix form as

$$D * W = C \qquad (3.1.5)$$

where D is the matrix with the hyperplanes' independent coefficients, d_{ij}, as defined in equation (3.1.1), C is the vector of constants c_i as defined in (3.1.2) and W is an unknown weight state vector.

In this section it is shown how to use a least mean squares approach to solve the problem defined in the previous section. The objective function to minimise in a least squares approach is

$$\| D * W - C \|_2 \tag{3.1.6}$$

which means in terms of solution manifolds to minimise

$$\sum_i \left(\sum_j d_{ij} * w_j - c_i \right)^2 \tag{3.1.7}$$

Comparing (3.1.7) with (3.1.4) and (3.1.3), where the objective function for the original problem is defined, it is clear that they do not correspond to the same problem. To make the problems identical it is necessary to normalise the coefficients of the hyperplanes according to (3.1.8).

$$nd_{ij} = \frac{d_{ij}}{\sqrt[2]{(\sum_j d_{ij}^{\,2})}}$$

$$nc_i = \frac{c_i}{\sqrt[2]{(\sum_j d_{ij}^{\,2})}} \tag{3.1.8}$$

where nd_{ij} and nc_i are the normalised coefficients. The normalised system can be written as

$$ND * W = NC \tag{3.1.9}$$

where ND and NC have the coefficients defined by (3.1.8). The problem based on this system can be seen as a type of weighted least squares.

If $\overline{W}$ is a solution then

$$ND * \overline{W} = P \tag{3.1.10}$$

where P is such that it minimises

$$\| P - NC \|_2 \tag{3.1.11}$$

Replacing P in equation (3.1.11) by the left side of (3.1.10), the quantity that is being minimised becomes as in (3.1.12).

$$\| ND * \overline{W} - NC \|_2 \tag{3.1.12}$$

This in turn means minimising

$$\sum_i \left(\sum_j n d_{ij} * w_j - n c_i \right)^2 \tag{3.1.13}$$

which is equivalent to

$$\sum_i \left(\frac{\sum_j d_{ij} * w_j - c_i}{\sqrt[2]{\sum_j d_{ij}^{\,2}}} \right)^2 \tag{3.1.14}$$

Minimising this last equation is identical to minimising the sum of Euclidean distances from $\overline{W}$ to the set of solution manifolds. Hence solving the normalised system of linear equations in the least squares sense is equivalent to determining the closest point to a set of hyperplanes.

To solve the system of equations described in (3.1.9) in a least squares approach it is necessary to cater for the following cases:

- Exact solution cases;
- Inexact solution cases;
- Multiple solutions cases.

The last two features listed above are probably the most important in the neural case. It is unrealistic to assume that all neural systems will have a single exact solution. A multiple solution case is more likely to occur than a single exact solution since it occurs when the number of linearly

independent hyperplanes is less than the number of weights. Incidentally, it will be shown in §3.3 that in a multiple solution case it is important to know the properties of the solution obtained. An inexact solution case occurs when the neural net architecture (§2.2.1) chosen is insufficient to solve the problem exactly. It is therefore necessary to select a system that caters for these situations.

Besides the features listed above it is also important that the system selected is able to deal with ill-conditioned systems from a numerical analysis point of view.

The system selected was Singular Value Decomposition, hereafter called SVD, for the following reasons :

- Golub and Loan (1983) claim that in cases of ill-conditioning, the reliability of SVD is unsurpassable:
- The solution given by SVD in a multiple solution case is the one of lowest Euclidean norm:
- In the inexact solution case SVD provides a solution in the least squares sense.

A guideline for the number of operations needed to solve the least squares problem with SVD using the algorithm is presented in Golub and Loan as being

$$4m\ n^2 + 8n^3 \qquad (3.1.15)$$

where m is the number of rows of the matrix and n is the number of columns of matrix NA. An operation is considered to be a floating point addition plus a floating point multiplication.

It is clear from (3.1.15) that the number of rows, i.e. patterns, has a much smaller impact on performance than the number of columns, i.e. weights. This feature is important when considering that SVD is responsible for the vast majority of computational expense in an epoch for the linear approximation approach which will be described later in §3.3.

Traditional algorithms like gradient descent methods have a computational expense per iteration directly proportional to the number of patterns, i.e., duplicating the training set implies twice the computational expense of the original training set. As far as SVD is concerned this relation is not as direct. Although adding patterns to the training set increases the computational expense, the example below shows that, when considering a training set which is four times the size of the original training set, the computational expense for SVD does not necessarily increase by four.

For instance let us consider a net with 20 weights and two training sets corresponding to the same problem. One training set has 20 patterns and a much richer training set has 80 patterns. The 20 patterns of the first training set implies that, according to (3.1.15), the computational expense for SVD per iteration is 72000 operations. The richer training, with four times more training patterns, requires only 96000 operations, i.e. an increase of only approximately 33%.

3.1.1.3. Conclusion

As far as single layer nets are concerned the alternative view of a goal based on the solution manifolds enables the user to find a solution in one step. This solution will coincide with the minimum of the overall error/weight surface if there is a common intersection to all the solution manifolds. Otherwise there will be a difference in the solutions for both methods. These differences will be dealt with more detail in chapter 4.

Using the new approach guarantees a weight jump with optimal direction and step size as opposed to gradient descent methods. This ability to find a good direction and step size will be most fruitful when the extension to multi-layer feedforward networks is considered.

Lastly it is worth referring to the fact that the new method scales up much better than standard gradient methods when the training set is enlarged.

3.1.2. Non-Linear Systems of Equations

In the general case a single layer net is insufficient to solve the problem and there is a need for multilayer feedforward nets as mentioned in §2.2. Furthermore, in this case the solution manifolds for individual patterns are not linear since in the equation for the net's output there is no linear relation between the weights and the output and hence between the weights and the error.

The set of solution manifolds therefore corresponds to a system of non-linear equations. The steepest descent method of backpropagation (§2.3) is well known for solving such systems for neural networks. There are also a number of classical techniques, besides steepest gradient descent, and some of them have been applied to multilayer feedforward neural nets with various degrees of success. See, for instance, Johansson, Dowla and Goodman (1992), or Møller (1993) for examples of conjugate gradient descent, Parker (1987), or Smagt (1994) on quasi-Newton methods, and Webb et al (1988) on the Levenberg-Marquardt technique.

All the approaches above, except steepest descent, assume to some degree that the surface is quadratic.

In this chapter a novel approach tailored especially for least squares problems and inspired by the linear case will be explored. In this method no

assumptions are made about the shape of the surface. This method uses linear approximation to non-linear solution manifolds to build a system of linear equations that can be solved as shown in §3.1.1. The solution obtained is a linear approximation to the solution of the system of non-linear equations. From §3.2 until the end of the chapter, several aspects of the method are detailed.

3.1.3. Conclusion

An alternative solution manifold view of a goal weight state was presented. In this view the goal is considered to be a point which is closest to all the solution manifolds in Euclidean distance terms.

Using the solution manifold view enables the user to compute the solution of a single layer net in one step.

In multi-layer feedforward networks the solution manifolds are non-linear. A novel linear approximation is taken in the next section to explore the potential of the solution manifold view.

3.2. Linear Approximation to Non-linear Solution Manifolds

The approach taken here is to compute linear approximations to the non-linear solution manifolds. The optimal solution of the system of linear approximations is an approximation to the optimal solution of the system of non-linear equations. A solution for a set of solution manifolds will be optimal within this approach if it minimises the squared sum of Euclidean distances from the solution to the set of solution manifolds. To find the optimal solution of the system of linear approximations the least squares approach illustrated in the section for the single layer net will be used.

The exact solution case will be used to demonstrate the features of a linear approximation for reasons of simplicity.

In the inexact solution case a difference between a goal weight state according to the global minimum view and a closest point to all the solution manifolds in Euclidean distance terms is to be expected. As mentioned before further details on this subject are presented in chapter 4. The principles present in the examples and diagrams to follow also apply to the inexact solution case as long as the difference mentioned is taken into account. Multiple solution cases are discussed later in §3.3.1.

If the current weight state coincides with the goal weight state then the solutions for both linear and non-linear systems are equal in the exact solution case, figure 3.3.a). In this case, the intersection of the non-linear solution manifolds is the same weight state as the intersection of the linear approximations. The linear approximations are taken at the closest points from the solution manifold to the current weight state. These points will be referred to as tangency points.

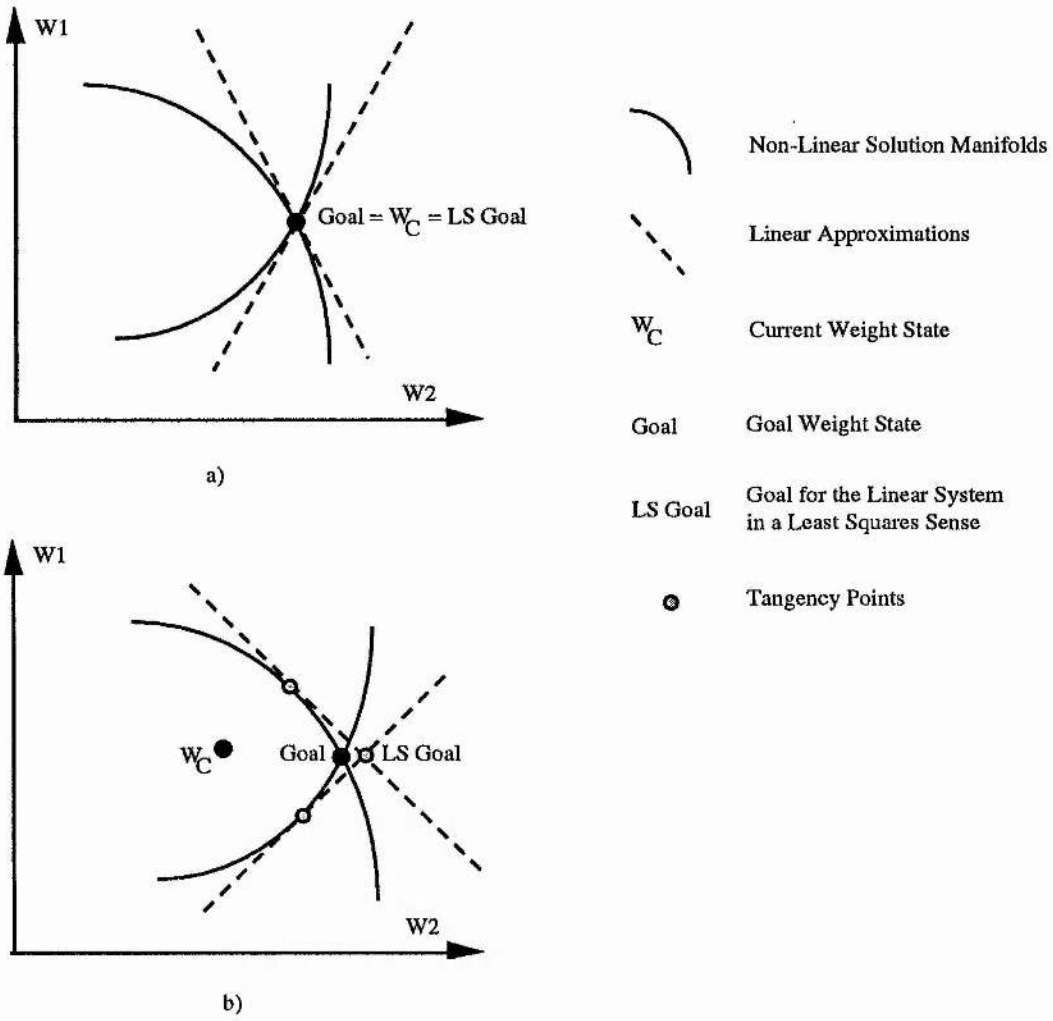


figure 3.3 a) The current weight state coincides with the goal hence the two solutions coincide. b) The current weight state is close enough to the goal and therefore the linear approximation is still good.

If the current weight state is placed close enough to the goal, the linear approximation is still good, see figure 3.3.b). However as the distance between the goal and the current weight state increases the linear approximation gets worse in the general case.

Note though that as the linear approximations deteriorate the problem with the linear approximation lies in the step size proposed more than with the direction. The optimal step size prediction tends to deteriorate much faster than the direction prediction. The direction obtained in the general case with

linear approximations has been empirically found to be more stable than step size when the distance increases.

In addition, the direction obtained with linear approximations has been found empirically to be much more accurate than the one obtained with steepest gradient descent. This is to be expected since using a system of linear approximations provides much more information to decide a direction in weight space than the overall steepest gradient descent direction. When compared to gradient descent methods, the linear approximations method combines the individual patterns' information in a more powerful way than superposition.

An intuitive graphical image as figure 3.4 exemplifies the behavioural difference in an overall error/weight surface between steepest gradient descent method and the linear approximations method. Steepest gradient descent is attracted strongly to the subset of solution manifolds associated with patterns of higher error. As one subset is neared, the other patterns are worsened in their error. Hence steepest gradient descent is attracted to varying subsets of solution manifolds thus creating an oscillatory path. By contrast, the linear approximations approach will create a steady trajectory along the solution manifolds towards the goal.

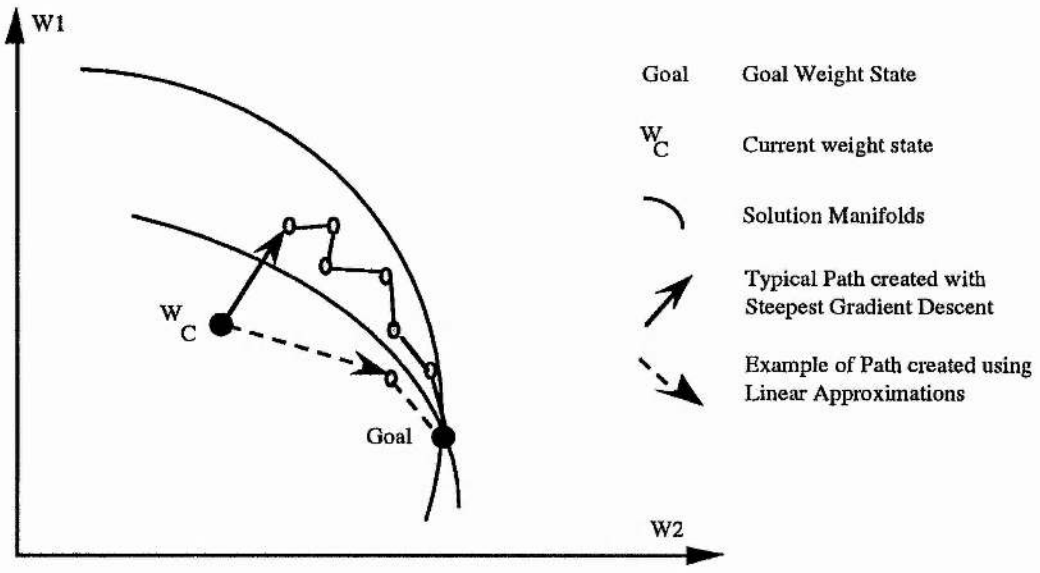


figure 3.4 - Examples of paths created with steepest gradient descent and with linear approximations.

Direction is the more critical of the two factors of direction and step size since step size is more easily corrected. The direction proposed by the linear approximations may be followed by a line search to find an optimal step size. Using line search is one option to cope with the deteriorated approximation.

Another option is to try to prevent the deterioration occurring in the first place. This approach is based on closeness since using linear approximations requires closeness to the goal to ensure a good approximation. The more curved the solution manifolds, the closer the current weight state will have to be to the goal.

In the general case the goal is not close enough to the current state to guarantee a good linear approximation both in terms of direction and step size. A possible solution is to select a sufficiently close temporary goal, i.e. a subgoal, so that a good linear approximation in both direction and step size towards the subgoal is guaranteed.

If the subgoal is closer to the goal than the current weight state in either output or weight spaces then achieving the subgoal will imply some progress towards the goal in the respective space.

However the goal position in weight space is unknown a priori, therefore it is not possible to recognise a progressive weight state. Hence it is not possible to select systematically a weight state to become a subgoal that will provide some degree of progress in weight space when achieved.

On the other hand, since the goal position in output space is known it is possible to select the target subgoal outputs to represent a lesser neural error than the current output state, so that the subgoal once achieved will bring some progress. The weight state that realises the targets selected must be close enough to the current weight state to be found by a good linear approximation.

Consequently, a trade-off between closeness and progress is implicitly present when selecting a subgoal. If too much progress is pursued in a subgoal then the weight state that realises it may be too far away from the current weight state to be found by a linear approximation. On the other hand, selecting a subgoal too close to the current state, while generating a very accurate linear approximation, will bring an insignificant amount of progress towards the goal once the subgoal is achieved.

To solve this dilemma, several candidates for subgoals are attempted for each current weight state to get the best possible trade-off between progress and closeness. From the set of candidate subgoals some will be close enough according to the criteria used while others won't. The main purpose of this selection process is then to select, between the achievable candidate subgoals, the most progressive one in output space.

A transition is made in weight space and another candidate subgoal may be then selected to become a temporary goal, ideally creating a chain of subgoals between the current state and a goal state.

3.2.1. Conclusion

A new method using linear approximations to non-linear solution manifolds to explore the potential of the solution manifold view of a goal weight state was presented.

Linear approximation is a way of combining the gradient vectors in an optimal geometrical fashion for linear systems. In non-linear systems however, linear approximations per se are insufficient to provide a good combination of direction and step size in the general case.

The direction provided has been considered empirically as being worthwhile to pursue in the general case. Using line search is a classical technique that can be used to determine the remaining feature of step size for a transition.

Another view of the problem, the subgoal approach, is based on the fact that closeness is one of the requirements of linear approximations for providing a correct step size. Using subgoals will guarantee a good direction and step size towards the subgoal. As long as a chain of subgoals sufficiently close together in weight space is established between the current state and a goal state this approach provides a path between the two states.

3.3. Tangent Hyperplanes as Linear Approximations

In this section the concepts of the previous section are presented in neural net terms. How to find a solution manifold and what the linear approximation to it is are two main points of focus during this section. Also presented is the construction of the system of linear approximations to be solved using least squares as shown in §3.1.1.2.

First the case of nets with a single output unit is presented. Afterwards an extension to general multilayer feedforward nets is described.

3.3.1. Nets with a Single Output Unit

The solution manifold for a pattern i in this type of net can be described by

$$N(W,I_i) = t_i \qquad (3.3.1)$$

where W is a solution weight state, I_i is the input vector from pattern i, t_i is the target for pattern i and N is the net's I-O function. As mentioned before in §2.3.2, input patterns are not really considered to be variables in the sense that they are constant throughout the training process. N may therefore be seen as a function of the weights that for the input training patterns provides the net's desired output. If there are k weights in the net then the solution manifold for pattern i, i.e. the set of weights that satisfies (3.3.1), is a surface with $(k - 1)$ dimensions.

In this case a linear approximation to a solution manifold is a linear surface with $(k - 1)$ dimensions, i.e. a hyperplane.

For each pattern, a hyperplane tangent to its solution manifold can be computed. The system of tangent hyperplanes is a linear approximation to the system of non-linear equations.

For each pattern, its solution manifold can be found doing line search along the direction proposed by the steepest gradient descent. The need for a line search arises from the fact that there is no linear relation between the error and distance to the solution manifold, i.e. the distance to the solution manifold following a certain direction cannot be predicted from two measures of the error at two different distances along that same direction. The use of steepest gradient descent in individual surfaces is justified here by the fact that the individual surfaces are much simpler than the overall surface so that the

direction proposed by the gradient in such surfaces is accurate enough for the purposes of this approach (see 2.3.2).

The point found in the solution manifold will be referred to as the tangency point. The vector that goes from the current weight state to the tangency point will be referred to as the gradient vector.

A tangent hyperplane can now be computed assuming that the gradient vector obtained at the current weight state is normal to the tangent at the tangency point on the solution manifold. Theoretically this occurs exactly when the error/weight contour that contains the current weight state is parallel to the contour that represents the solution manifold. However this is not the general case in neural nets for the individual error/weight surfaces, a small degree of non-parallelism can be found in the contours of individual error/weight surfaces. Nevertheless it was found empirically that the degree of non-parallelism present in neural individual error/weight surfaces does not affect significantly the proposed direction or step size of the linear approximation to the goal or subgoal in the sense that a good approximation is still obtainable if the goal, or subgoal, is close enough.

The tangent hyperplane is orthogonal to the gradient vector and contains the tangency point as shown in figure 3.5.

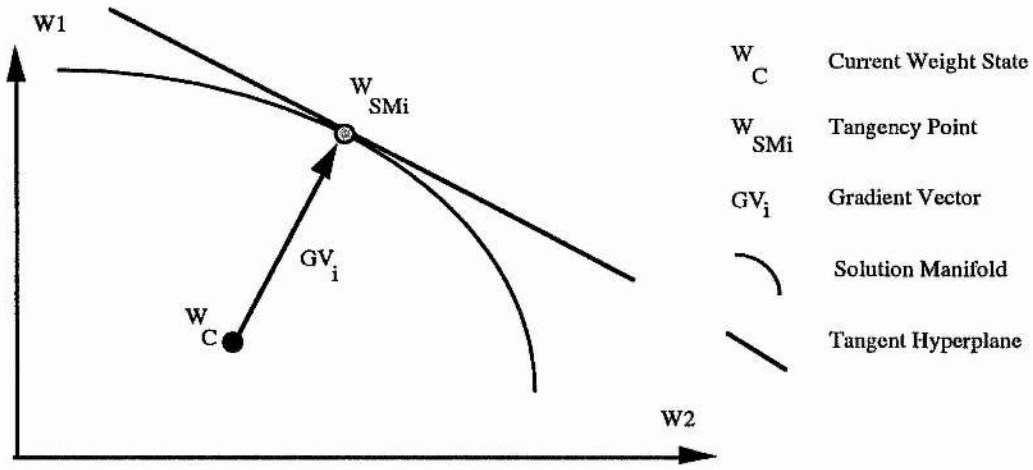


figure 3.5 - shows a hyperplane tangent to a solution manifold for pattern i.

The equation for a hyperplane is

$$a_{i1}w_1 + a_{i2}w_2 + \ldots + a_{ik}w_k = b_i \qquad (3.3.2)$$

where k is the number of weights and i is the index of the pattern to which the linear approximation is being computed. The set of equations obtained for the set of patterns can also be written in matrix notation as being

$$AX = B \qquad (3.3.3)$$

where A is a matrix containing the a_{ij} coefficients of the hyperplanes and B is a vector with coefficients b_i.

The points W_{SM_i} and W_C and the vector GV_i used in the equations that follow are those present in figure 3.5. The tangent hyperplane can be defined as being orthogonal to the gradient vector, GV_i, and containing the tangency point, W_{SM_i}, for pattern i. The gradient vector for pattern i, GV_i, is defined by the current weight state W_C and the point W_{SM_i}.

$$GV_i = W_{SM_i} - W_C \qquad (3.3.4)$$

The coefficients from matrix A, a_{ij}, are defined by

$$a_{ij} = GV_{ij} \tag{3.3.5}$$

and the b_i coefficients from vector B in equation (3.3.3) are defined by

$$b_i = \sum_j (a_{ij} W_{SM_{ij}}) \tag{3.3.6}$$

As mentioned for the single layer net case, the hyperplane equations need to be normalised before the set of equations is solved using the least squares approach described previously. The normalised coefficients are as defined in (3.3.7)

$$na_{ij} = \frac{a_{ij}}{\sqrt[2]{\sum_j a_{ij}^2}}$$

$$nb_i = \frac{b_i}{\sqrt[2]{\sum_j a_{ij}^2}} \tag{3.3.7}$$

The system of normalised equations will be

$$NA * W = NB \tag{3.3.8}$$

where NA and NB have their components defined by (3.3.7). The point in weight space W, when the system is solved in a least squares sense as described in §3.1.1.2, represents the new weight state.

As mentioned before, for the multiple solution case, the least squares approach used here provides the point of lowest Euclidean norm, i.e., the point which is closest to the origin. For linear solution manifolds this poses no problem since all solutions are equally close to all the solution manifolds. However for non-linear solution manifolds the situation is different.

For instance, let us consider a 3-D space and two 2-D solution manifolds. The 2-D tangent hyperplanes to the solution manifolds intersect in a straight line in the general case. This straight line can be seen as an approximation to the curve which, in the general case, is the intersection of the two 2-D solution manifolds. In this case the points in the line are not equally close to the curve. The approximation is optimal, in the general case, at the point in the line which is closest to the current weight state. However, the point provided by the least squares approach is the point which is closest to the origin (§3.1.1.2).

In order to solve this problem a translation is performed. The current weight state becomes the origin and all the computed hyperplanes have to be translated so as to keep their relative positions to the current weight state. The orientation of the hyperplane remains constant, only its position must be altered. The coefficients in matrix A, defined in (3.3.3), determine the hyperplane's orientation and therefore matrix A remains constant. Also since the normalisation defined in 3.3.7 depends only on matrix A, the matrix NA remains constant as well. The position of the hyperplanes, defined by the coefficients in vector B (3.3.6), is dependent on the matrix A coefficients and also on the tangency points. The tangency points have to be translated according to

$$W_{TSM_i} = W_{SM_i} - W_C \qquad (3.3.9)$$

where W_{TSM_i} is the point that represents W_{SM_i} in a new frame of reference where W_C is the origin.

A new vector B' defining the position of the hyperplanes in the new frame of reference where W_C is the origin has to be computed. The value of its component, b_i', is

$$b_i' = \sum_j (a_{ij} * W_{TSM_{ij}}) \tag{3.3.10}$$

which when normalised becomes

$$nb_i' = \frac{b_i'}{\sqrt{\sum_j a_{ij}^2}} \tag{3.3.11}$$

The new system looks like equation (3.3.12).

$$NA * X = NB' \tag{3.3.12}$$

where NB' is a vector with components defined in (3.3.11). The weight space point X given by least squares needs to be translated back to the original frame of reference. The value given by least squares in the new frame of reference can be seen as the vector that indicates the magnitude and direction of the jump in weight space towards the new weight state in the original frame of reference. The new weight state, W_{LS}, is defined as

$$W_{LS} = W_C + X \tag{3.3.13}$$

3.3.2. Nets with Multiple Output Units

In the case of a net with r output units the solution manifold for a pattern is defined by the system of equations

$$\begin{cases} N_1(W, I_i) = t_{i1} \\ N_2(W, I_i) = t_{i2} \\ \ldots\ldots\ldots\ldots\ldots \\ N_r(W, I_i) = t_{ir} \end{cases} \tag{3.3.14}$$

where N_j is the net's I-O function for the output unit j, I_i is the input vector of pattern i, W is a weight state and t_{ij} is the target for pattern i and output unit j.

Each N_j defines the output for an output unit based on a weight state and an input pattern. As mentioned before for the single output unit case, the input pattern is not considered to be a variable since it is fixed during training. Each N_j is therefore a function of the weights. If there are k weights in the net then the set of weights that satisfies the function is a surface with $(k-1)$ dimensions.

Each of these surfaces can be seen as a solution manifold for a particular output unit in the net. To be considered a solution a weight state has to produce correct outputs for all output units. Hence, the global solution manifold for a particular pattern is the intersection of all the solution manifolds for each output unit; therefore, in the general case, it has $(k - r)$ dimensions.

The approach taken here to compute a linear approximation to the global solution manifold of a pattern, as described in (3.3.14), is to compute a linear approximation to each of the output units' solution manifolds. The system of linear approximations obtained for a pattern is an approximation to the system in (3.3.14).

For each pattern such a linear system can be computed. The set of systems of linear approximations obtained is an approximation to the set of systems of equations found for the set of patterns.

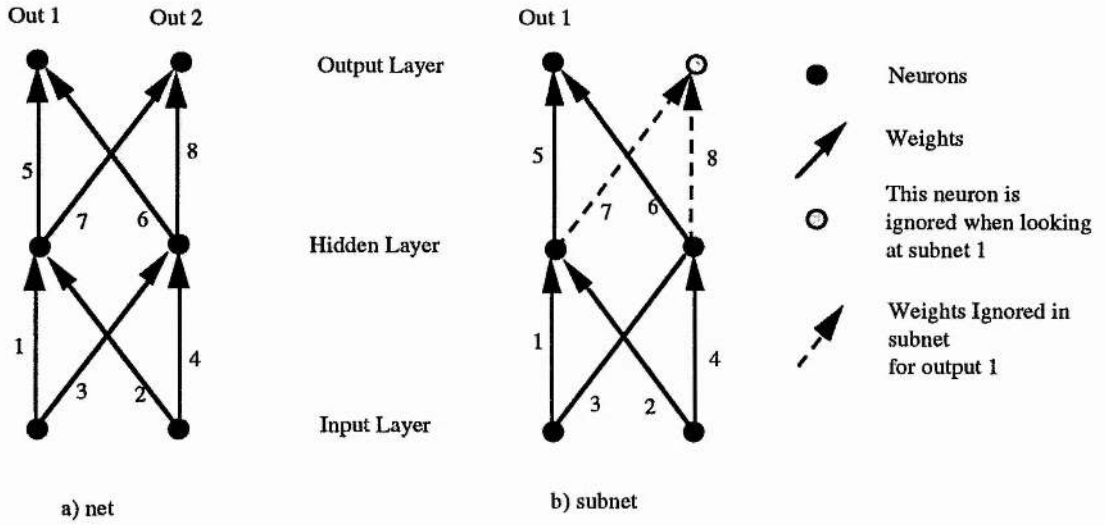


figure 3.6 - the concept of a subnet

In figure 3.6 the subnet concept is introduced. A subnet in the scope of this work is a part of a net in which only one output unit is present. For any particular net there are r subnets in which r is the number of output units. Figure 3.6.b) shows a subnet of the net in 3.6.a) where all output units but output unit 1 are ignored. The equations in (3.3.14) each relate to a subnet; in particular the solution manifolds for each output unit correspond to the solution manifolds of the subnets. This is because equation N_i defines the output in the subnet which contains the output unit i.

For each subnet a linear approximation to a pattern's solution manifold can be computed. To find the solution manifold of a subnet for a pattern, the pattern's steepest gradient direction is computed for that subnet, i.e. all output units in the net that don't belong to the subnet have no influence on the gradient's computation. The direction found is used in conjunction with line search to find the subnet solution manifold and a tangent hyperplane to it can then computed as described in §3.3.1.

For each pattern then, there is one tangent hyperplane per output unit. The system of r hyperplanes, r being the number of output units, is the linear

approximation to the net's global solution manifold described in (3.3.14) for the pattern.

In matrix NA in equation (3.3.12) each row represents a hyperplane. Hence $r*p$ (r stands for the number of output units and p for the number of patterns) rows will be needed in the general case of a feed-forward net with multiple output units. The same principle applies to vector NB' (3.3.12).

In figure 3.6 a net with two output units is shown. If, for example, the training set has only two patterns then matrix NA is as defined in equation (3.3.15).

$$NA = \begin{bmatrix} NA_{11} \\ NA_{12} \\ NA_{21} \\ NA_{22} \end{bmatrix} \qquad (3.3.15)$$

where NA_{ij} stands for the normalised gradient vector computed for pattern i and output unit j assuming that all other output units have zero error, i.e. the coefficients of the gradient vector corresponding to the weights connecting to those output units are zero. For instance NA_{11} is defined as

$$NA_{11} = \begin{bmatrix} na_{111}\ na_{112}\ na_{113}\ na_{114}\ na_{115}\ na_{116}\ 0.0\ 0.0 \end{bmatrix} \qquad (3.3.16)$$

where na_{11j} stands for the gradient vector for the subnet containing output unit 1 for pattern 1, weight j in figure 3.6. The zeros correspond to the weights that are not part of the subnet; in this case the weights 7 and 8 are not part of the subnet in figure 3.6.b).

3.4. Tangent Hyperplanes with Line Search

Line search is a classical technique to find the optimal step size. The use of line search will enable the method to follow the direction proposed by the least squares method without overshooting the goal.

There are two measures that can be minimised along the direction provided using tangent hyperplanes :

- Euclidean distance to the solution manifolds;
- Output error (§2.3).

The aim of using solution manifolds is to find the point which is closest to all solution manifolds so that Euclidean distance is the logical measure to minimise.

However, this might turn out to be computationally very expensive. For each point in the line search the individual solution manifolds would have to be found and the Euclidean distance recomputed. This involves, for each step of the line search, computing the steepest gradient descent for each pattern and doing another subsidiary line search to find each pattern's solution manifold.

Output error can also be used for the line search along the direction proposed using tangent hyperplanes, although it differs from minimising Euclidean distance. Output error is not proportional to Euclidean error in any case where the error/weight contours are not parallel, i.e. there may be cases where two different weight states at the same distance from a solution manifold have different output errors. The line search done using output error does not necessarily stop at the closest point to the solution manifolds.

Although the above shows disadvantages to minimising output error, it is a much cheaper option than minimising the Euclidean distance and therefore it will be kept as an option for now. Only a forward pass (see §2.2.2) is involved for each step of the line search in the output error approach.

There may be cases for both measures when significant progress cannot be achieved. These are:

- The direction proposed by the least squares method has the current weight state as being a local minimum for the measure which is being used for the line search;

- The weight state proposed by line search for either measure is extremely close to the current weight state.

When the first case occurs there is no possible progress along the direction proposed. Hence the method is stuck. The second case occurs due to a very accentuated curvature on the solution manifolds and a good linear approximation can only be obtained if the subgoal weight state is extremely close to the current weight state.

To avoid such situations, the line search includes an allowance in the cases where the error cannot be decreased for the respective error measure, or when the potential progress is too small to be worth pursuing a good linear approximation. A minimal weight transition is done, even when the error increases, if the heuristic criterion defined in (3.4.1) is satisfied. It has been found empirically that the criterion in (3.4.1) speeds up convergence.

$$\frac{|E_C - E_L|}{E_C} < P \qquad (3.4.1)$$

where E_C is the error associated with the current weight state, E_L is the error associated with the weight state being considered under the line search and P is some small constant defined a priori. If the left side of the inequality (3.4.1) was multiplied by 100 then P would represent the percentage that the error is allowed to increase in each iteration.

3.5. Tangent Hyperplanes with Subgoals

In this section, an alternative to using linear approximation for the goal direction alone will be presented. As discussed before in §3.2, a linear

approximation will be accurate in direction as well as step size if the goal weight state is close enough to the current weight state. However, it is not possible to choose the current weight state to be close to the goal.

This obstacle may be overcome though if a subgoal can be placed close enough to the current weight state in order to guarantee that a good linear approximation towards the subgoal is obtained. This method will be useful though, only if achieving the subgoal implies achieving some progress in either output or weight spaces.

To obtain progress in weight space it is necessary to know the goal position in weight space so that the subgoal can be placed systematically closer to the goal than the current weight state. However the goal position in weight space is not known a priori.

In output space terms though, the goal position is known so it is possible to select a subgoal in output space so that if the subgoal is achieved then progress is achieved. As mentioned before, a subgoal must be placed close enough to the current output state to be achievable, yet on the other hand it should be as far away towards the goal as possible to allow for as much progress as possible.

This interaction between closeness and progress implies attempting several candidate subgoals for each current weight state to obtain the best possible trade-off. Each time a subgoal is attempted a weight state is obtained. This weight state is the linear approximation to the system of non-linear solution manifolds. The weight state, and the respective output state it realises, can then be evaluated for progress according to the criteria described in §3.5.1.

There is no reason a priori to avoid attempting the goal output state as the first candidate subgoal. If the goal is close enough in weight space then the linear approximation will yield a good solution. To compute a linear

approximation to the goal as the first candidate subgoal, the steps described in §3.3.1 are followed.

If the goal is not close enough in weight space so that a good linear approximation is unobtainable, then a closer candidate subgoal in output space must be considered. Closer and closer candidate subgoals in output space are considered until one is found to be close enough and as a consequence the respective linear approximation is deemed to be good.

As mentioned before, a subgoal must be placed in output space, and two ways of selecting a closer subgoal candidate in this space are examined in §3.5.2 and §3.5.3. In §3.5.4 the pros and cons of each approach are discussed and the approach taken is revealed.

The computational effort to compute subgoals is analysed at §3.5.5. An algorithmic version of the method in which all the parts are put together is presented in §3.5.6.

3.5.1. Candidate Subgoal Evaluation

A set of heuristic criteria has been developed for choosing a candidate subgoal as the actual subgoal. If these criteria are met then the linear approximation is considered good enough and a transition in weight space is done. However, should these criteria fail there is a need to consider candidate subgoals which are closer to the current state.

Theoretically it is always possible to select a candidate subgoal to be close enough to the current state to make the solution manifolds highly linear locally and therefore guaranteeing a very accurate approximation. Having said this, a subgoal is only an intermediate step towards the goal so that achieving the candidate subgoal precisely is not necessary. Achieving a subgoal within a certain tolerance is a more sensible approach. The number

of candidate subgoals tested for each weight state will be smaller if a tolerance is used thus shortening the computation for each iteration.

The achievability of progress towards a subgoal can be assessed empirically using output information with (3.5.1).

$$\|O_C - O_{SM}\|_2 > \|O_{LS} - O_{SM}\|_2 \qquad (3.5.1)$$

where O_C is the current output state, O_{SM} is the subgoal output and O_{LS} is the output state that is realised by the weight state achieved using least squares as described in §3.3.1. Figure 3.7 depicts the criterion.

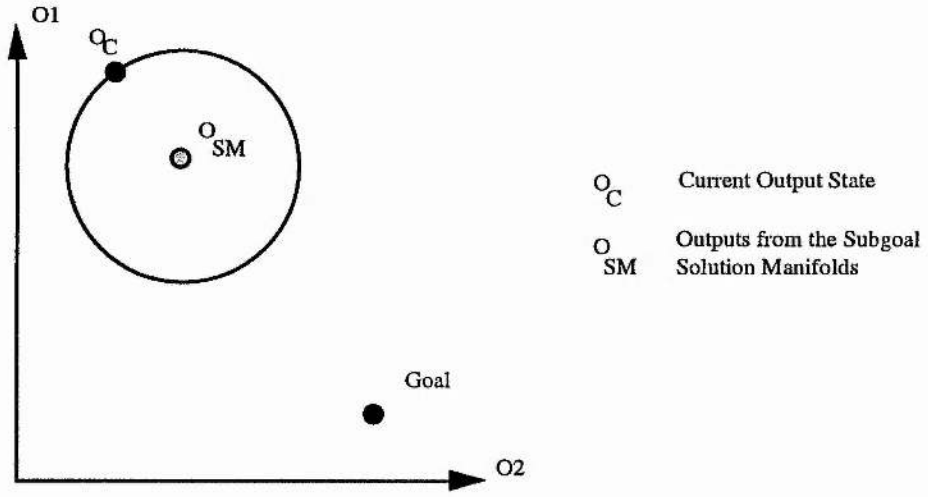


Figure 3.7- Illustration of the criterion in (3.5.1). If the output state associated with the weight state given by least squares, O_{LS}, is within the circle then the criterion is satisfied.

The criterion in (3.5.1) on its own may be inadequate for cases when the output subgoal state is far away from the current output state. This is because a small improvement in distance from the subgoal may then represent progress, but not *stable* progress, i.e. linear progress achieved through good linear approximation (see Figure 3.8).

The approach taken in this work is to keep the transitions under control and accept a candidate subgoal not just because progress in output space is obtained but because a good linear approximation is also obtained.

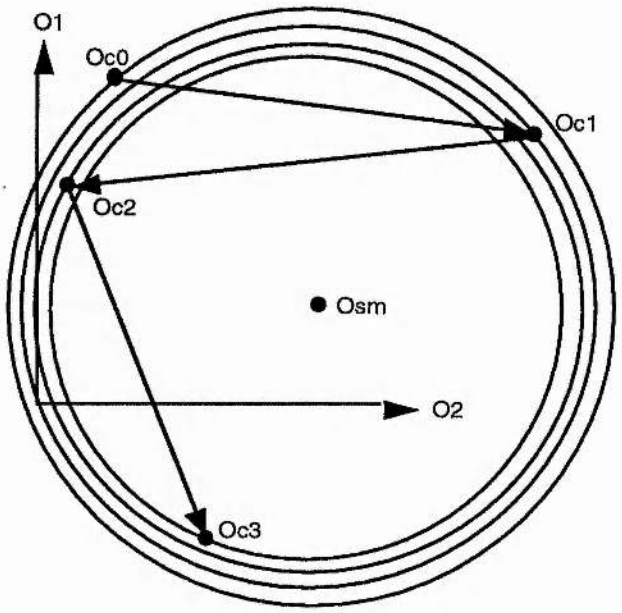


Figure 3.8. The circle criterion 3.5.1 may be insufficient on its own to yield stable progress towards the goal.

To ensure that a good linear approximation is made, two other heuristic criteria to select a subgoal were devised. The first one creates an upper bound for the weight's variation (3.5.3), and the second limits the acceptable output difference between the subgoal output and output achieved per pattern (3.5.4).

$$\forall i \; : |W_{Ci} - W_{LSi}| < L_w \qquad (3.5.2)$$

$$\forall p : |O_{SMp} - O_{LSp}| < L_O \qquad (3.5.3)$$

where W_{Ci} is the i^{th} component of the current weight state, W_{LSi} is the i^{th} component of the weight state proposed by least squares for the current candidate subgoal, O_{SMp} is the output for pattern p at the subgoal output state,

and O_{LSp} is the output for pattern p obtained at the weight state W_{LS}. L_W and L_O are small real positive constants set a priori by the user.

The a priori aspect to setting L_W and L_O may mean that the solution manifolds for some regions are such that L_W or L_O are in fact too high for stable approximation. Each acts a secondary safeguard in case the other is set too high, in essence providing two independent means of stability.

It might be thought that criterion (3.5.3), hereafter called the box criterion, supersedes criterion (3.5.1), hereafter called the circle criterion, since both assess progress in output terms. However, consider the case when the subgoal is relatively close to the current output state (Figure 3.9).

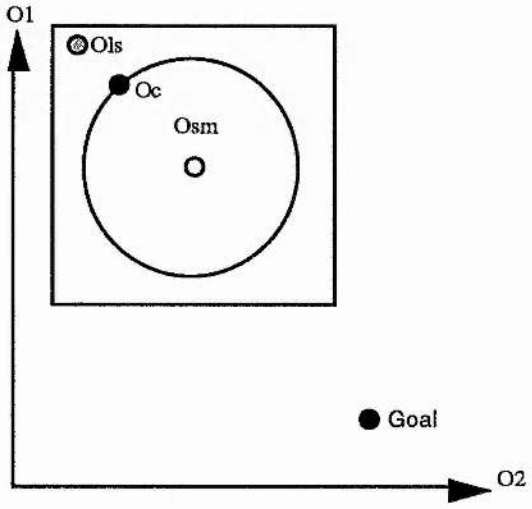


Figure 3.9. The relation of Criteria (3.5.1) and (3.5.3) when the subgoal is relatively close to the current output state. O_{LS} is shown at an unprogressive position.

In this case it is not sensible to use the box criterion on its own because it allows unprogressive jumps in output space towards the subgoal (see Figure 3.9). The circle criterion may be used in addition here to override the box criterion and prevent acceptance of the candidate step size.

Finally, a fourth criterion has proved useful. It may be the case on occasion that the degree of closeness required for a good linear approximation is excessive. The fourth criterion comes into its own when the potential progress is too small to be worth pursuing an acceptable match with closer candidate output subgoals according to the first three criteria. Consequently, as in the line search (§3.4) case, a minimal weight transition is done whenever the criterion in (3.5.4) is satisfied to ensure that the algorithm does not become stuck.

$$\|O_C - O_{LS}\|_2 < T \qquad (3.5.4)$$

The four criteria are combined as follows. The two criteria in (3.5.2) and (3.5.3) together ensure closeness so as to encourage a good linear approximation. Consequently, a candidate subgoal is only accepted if both the criteria specified in (3.5.2) and (3.5.3) are verified. If and only if a candidate subgoal satisfies these two criteria will the other two criteria be checked for the candidate subgoal. They are checked so that if there is progress towards the subgoal (3.5.1), or the candidate step size is small (3.5.4), then a transition is done.

In logical terms, a transition is made when the expression

((3.5.2) AND (3.5.3)) AND ((3.5.1) OR (3.5.4))

is TRUE, where the number of the criterion represents a logical value of TRUE if the criterion is satisfied and FALSE otherwise.

Restricting the step size by using Lo and Lw can cause an increase in the number of epochs needed for some problems relative to not using them. For instance, progress towards the goal is neglected when one is trying to achieve a subgoal which is not the goal itself. That is, progress towards the goal may be possible for larger sizes than the one eventually taken. The

relatively small step size is taken because it is the largest step size making progress towards the subgoal.

However, there are many cases where the use of Lo and Lw is beneficial. For example, in the function approximation problem described in §5.4 14 trials were done without using Lo. In these 14 trials, 5 needed over 1000 epochs to converge. Upon using Lo and setting it to 0.01, all 14 trials converged before 1000 epochs. Furthermore, in 11 of the cases the results using Lo = 0.01 were better, in terms of the number of epochs needed for convergence to be achieved, than the results when not using Lo.

In all the problems tested in this thesis, and for the values of Lo and Lw tested, the best results in terms of convergence rates were almost always obtained with a low, i.e. restrictive, setting. This shows that, at least in the problems tested, Lo and Lw can provide better convergence rates when small settings are used, thereby making the Tangent Hyperplanes method more robust.

3.5.2. Candidate Subgoal Setting Using Output Measures

Any output state which is between the goal and the current output state is a natural candidate subgoal. Output subgoals in this condition will, once achieved (§3.5.1), allow for some progress in output space in the general case.

Candidate subgoals can be set using output space information only. For each pattern an output which is between the current output and the goal output may be considered as a component of an output candidate subgoal.

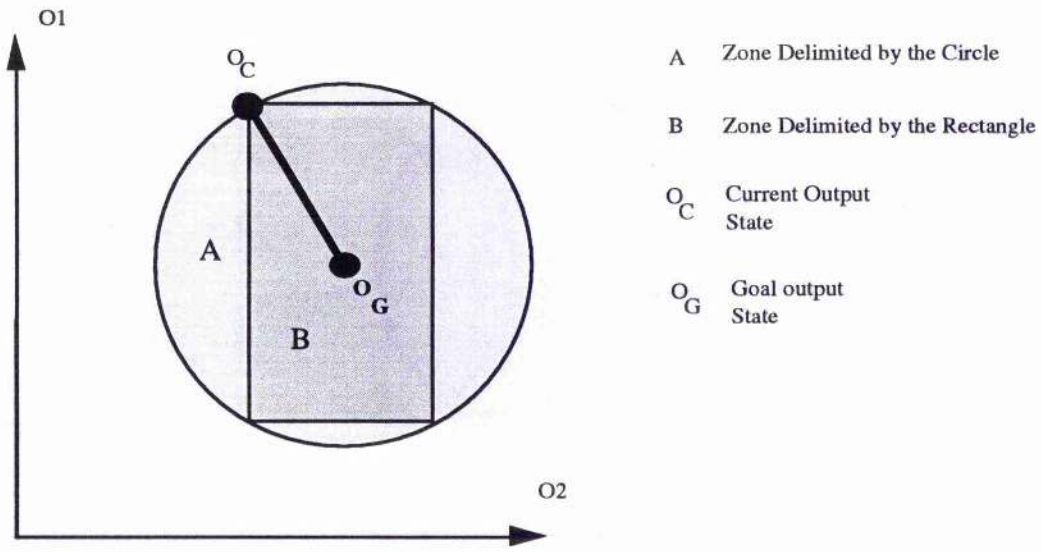


figure 3.10 - output space regions to select candidate subgoals

Figure 3.10 exemplifies three possibilities to place a candidate subgoal in 2-D output space in which the overall error is decreased once the subgoal is achieved. If a candidate subgoal is placed inside zone A, the region within the circumference, then once the subgoal is achieved it will decrease the overall error although some patterns may have their error increased. Any candidate subgoal inside zone B, the rectangle, will provide a decrease in error for all patterns. Finally candidate subgoals placed in the bold straight line segment connecting the current to the goal output states will try to satisfy all patterns proportionally to their initial linear error.

The approach taken here places the candidate subgoals on the bold line segment. If one was to select either the circumference, zone A, or the square, zone B, some criteria not based only on the error would be needed to specify which patterns require more attention. Such criteria requires more information than the presently available at each iteration. Without additional information giving them an advantage though, there is no reason to contemplate a less direct setting of candidate output subgoals than the bold line segment approach.

To create an output subgoal an individual output subgoal has to be defined for each pattern. The output subgoal for each pattern is set according to equation (3.5.5).

$$O_{Si} = O_{Ci} + [(O_{Gi} - O_{Ci}) * f] \qquad 0 < f \leq 1 \qquad (3.5.5)$$

where O_{S_i} stands for the output subgoal for pattern i, O_{C_i} for the current output for pattern i, and O_{G_i} for the goal output for pattern i. The variable f determines how close the subgoal is set in output space.

For each pattern, the error contour from the individual error/weight surfaces that corresponds to the output selected as subgoal can be found doing line search along the individual steepest gradient direction. The contours then become the candidate subgoal solution manifolds. Linear approximations can then be computed to each solution manifold. Afterwards the linear system constructed can be solved using the least squares approach described in §3.1.1.2. The weight state obtained is the candidate subgoal attempt that must be evaluated according to the criteria in §3.5.1.

In the subgoal approach, the goal is attempted as the first candidate subgoal. This is equivalent to having f equal to 1 in (3.5.5). If the candidate subgoal attempt passes the criteria in §3.5.1 then a weight transition is done and the whole process is repeated from the new weight state. Otherwise a closer candidate subgoal must be considered, i.e., a smaller value for f is used to define a new candidate subgoal. The variable f is decreased until a candidate subgoal is attempted successfully according to §3.5.1. A practical way of selecting a smaller f is using a bisection algorithm, i.e. if a candidate subgoal fails to meet the criteria in §3.5.1 then the variable f is halved in value.

To provide a better understanding of the subgoal approach an example is now presented. Let us suppose we want to train a net with two inputs and one output to classify two patterns P_1 and P_2. P_1 has a desired output of 0.95 and P_2

of 0.05. The initial weight state is obtained randomly as usual. Let us suppose that the initial weight state provides an output of 0.53 to P_1 and 0.64 to P_2. Classical approaches like backpropagation use 0.95 and 0.05 as targets to train the net until these values are actually achieved within a certain tolerance.

As mentioned before, the goal is considered to be the first candidate subgoal in the approach taken. If the candidate subgoal is attempted successfully then a transition in weight space is done otherwise a closer candidate subgoal must be considered. Let us assume that the goal failed to meet the criterion in §3.5.1, 0.5 is the next value selected for f. The candidate subgoal output for P_1 is computed as in (3.5.6).

$$OS_1 = 0.53 + [(0.95 - 0.53) * 0.5] = 0.74 \qquad (3.5.6)$$

The same reasoning applied to P_2 would yield an output subgoal of 0.345. These values would form the output candidate subgoal. If the candidate subgoal attempt fails the criteria in §3.5.1 then f must be reset to an even smaller value until the criteria in §3.5.1 are satisfied.

After satisfying a candidate subgoal another one is to be computed based on the new output state. Subgoals are to be computed repeatedly until the new output state coincides with the goal within a certain tolerance, i.e., the output subgoal for P_1 has to eventually be close to 0.95 and the output subgoal for P_2 has to eventually be close to 0.05.

3.5.3. Candidate Subgoal Setting Using Weight Space Measures

As mentioned before in §3.5.2, when using output space measures an output subgoal target is selected for each individual pattern, then the contours corresponding to these subgoal output targets must be found. This approach involves, for each individual pattern, a line search to find the appropriate

contours. These contours then become solution manifolds for the subgoal output targets selected.

Another possible approach is to select a contour for each pattern directly without line search, using the gradient vector computed for the goal solution manifold. The output state corresponding to the set of contours is considered to be the output candidate subgoal.

Contours between the current weight state and the goal contour are natural candidates to become candidate subgoal solution manifolds since the corresponding output state is closer to the goal output than the current output state.

A simple way of selecting the candidate subgoal solution manifold is using a vector which has the gradient vector's direction but with lesser magnitude, see figure 3.11. The contour which is intersected by the tip of this new vector, W_{SSM_i}, becomes the candidate subgoal solution manifold for pattern i.

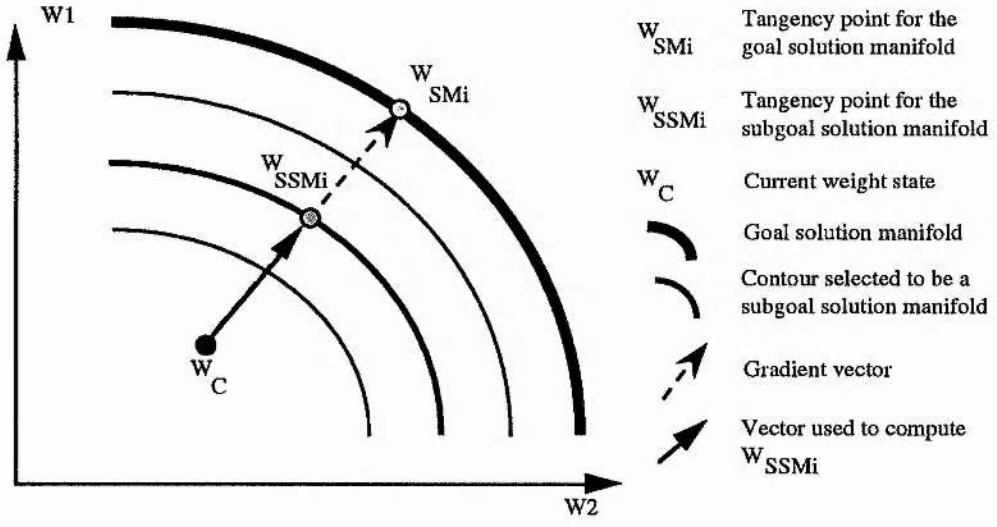


figure 3.11 - An error/weight contour is selected to become a subgoal solution manifold

A hyperplane orthogonal to the vector obtained and containing the tangency point for the subgoal solution manifold, W_{SSM_i}, is then computed as being

tangent to the candidate subgoal solution manifold as explained in §3.3.1. To compute W_{SSM_i} (3.5.7) can be used.

$$W_{SSM_i} = W_C + (W_{SM_i} - W_{C_i}) * f \qquad 0 < f < 1 \qquad (3.5.7)$$

The points referred to in (3.5.7) are those present in figure 3.11. The variable f is responsible for the distance at which the hyperplanes are set. The variable f can be made different for each pattern. However, as mentioned before in §3.5.2, without further information showing an advantage, there is no reason to contemplate less direct settings. Therefore the variable f is always set to the same value for all the patterns in the approach taken.

Selecting the candidate subgoals in this way causes the proposed weight transition vectors computed for the candidate subgoal attempts to be collinear, as shown in figure 3.12 (this figure is a more complete version of fig. 3.11 with two patterns and including the tangent hyperplanes). Furthermore, these vectors have a magnitude which is proportional to the variable f allowing for the candidate subgoal attempts to be computed directly. In graphical terms, only a rescaling of a linear diagram is involved repeatedly when going from the goal solution manifold to a closer candidate solution manifold. These properties will be demonstrated in §3.5.5.

The least squares system thus has to be solved just once for each current weight state for the first candidate subgoal, i.e. the goal, to provide an initial direction and step size.

For all other subgoal candidates, the weight transition proposed by least squares can be computed directly as follows without solving a linear system. Assume that X_{LS} is the weight space vector obtained when using a linear approximation to the goal solution manifold. Furthermore, assume that this approximation has failed the heuristic criteria in §3.5.1.

Since it is assumed that the linear approximation to the goal is not considered good enough, then a closer candidate subgoal must be considered. For this end the variable f is set at a value between 0 and 1 as imposed by (3.5.7). The linear approximation to the candidate subgoal for that value of f, W_{LSS}, can be computed directly as being

$$W_{LSS} = W_C + (X_{LS} * f) \qquad (3.5.8)$$

All the points mentioned in the equation are present in figure 3.12. X_{LS} is also implicitly present in figure 3.12 as being the vector from W_C to W_{LS}.

To test the linear approximation, the output state for any candidate subgoal besides the goal must be computed as well. For each pattern it is necessary to compute the tangency points for the candidate subgoal to determine vector NB' (3.3.12). Equation (3.5.7) can be used for this end. The output candidate subgoal can then be read for each pattern at the respective tangency point.

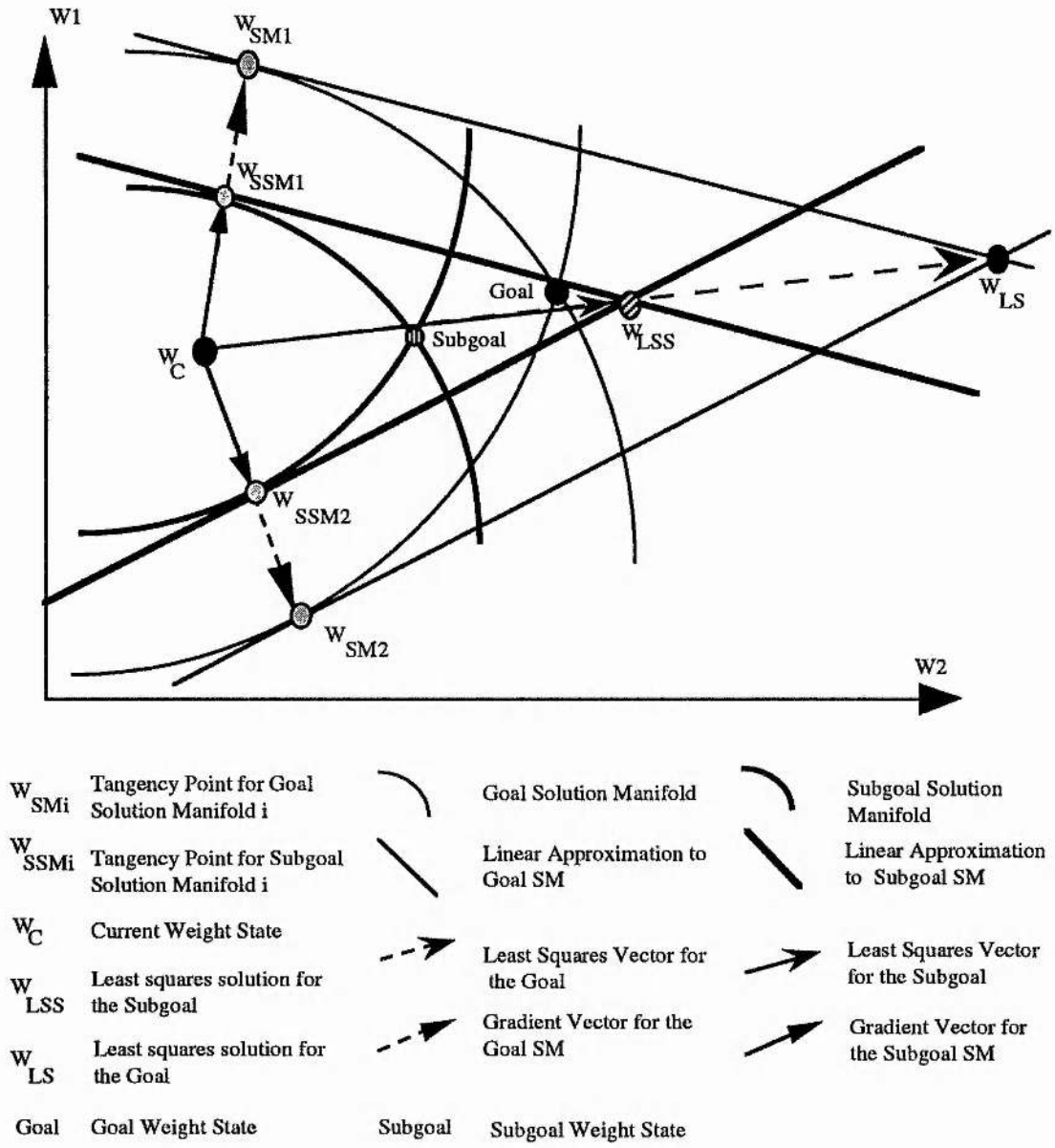


Figure 3.12 - The least squares vector for the subgoal is collinear with the vector for the subgoal

When the candidate subgoal fails to satisfy the heuristic criteria in §3.5.1, a closer candidate subgoal must be considered. The variable f is decreased each time a new candidate subgoal is attempted. Once a candidate subgoal is found that satisfies the criteria defined in §3.5.1, a transition in weight space is done and one epoch is then completed.

3.5.4. The Approach Taken

In the first approach, §3.5.2, output space is considered whereas in the second, §3.5.3, the focus is on weight space.

The output space approach is more error oriented than the weight space approach. This represents an advantage for the output space approach in the sense that the subgoal output targets are set proportionally to the error at the current state so that patterns with higher error are given more weight in the final decision. In the weight space approach only the distances between the current weight state and the solution manifolds are considered.

Although not catering directly for each individual pattern's error, the weight space approach was selected because it is much cheaper computationally than the output space approach. It requires the least squares problem to be solved only once per iteration. This will be demonstrated in detail in §3.5.5.

Also, the contour selection is direct in the weight space approach, whereas in the output approach it involves line search.

The weight space approach is based on the distances between the current weight state and the solution manifolds whereas the output approach is based on the distances in output space, i.e. output error. The objective function for minimisation proposed in this work is based on the distances in weight space and the weight space approach is theoretically closer to this purpose than the output space approach.

The fact that the weight space approach is cheaper and theoretically more close to the objective function makes it preferable as the approach to be taken.

3.5.5. Subgoal Computation

In §3.5.3 it was mentioned that the linear approximations to the candidate subgoals could be computed directly from the linear approximation to the goal without having to solve a linear system as described in §3.3.1. That is, there is a need to solve a linear system in the least squares approach only once per iteration. In this section these statements are proved in detail.

This has a major impact on performance. If it turned out to be necessary to solve a linear system in a least squares sense for each candidate subgoal then the method would be much less feasible due to the high amount of computations required per epoch.

It is worth remembering at this stage that a translation is done so that the current weight state coincides with the origin, §3.3.1. The computations below for each new candidate subgoal will be done in this frame of reference before normalisation occurs.

To compute a new candidate subgoal a set of new tangency points has to be evaluated. Due to rescaling by f, we have

$$W_{TSSM_i} = W_{TSM_i} * f \qquad 0 < f < 1 \qquad (3.5.9)$$

where W_{TSM_i} stands for the translated tangency point computed for the goal for pattern i, f represents the fractional distance at which the new tangency point, W_{TSSM_i}, computed for the candidate subgoal for pattern i is placed.

The orientation of the hyperplanes remains constant since the gradient vectors used to find the tangency points for the subgoal are collinear with the gradient vectors computed to find the tangency points for the goal, see §3.5.3. Therefore matrix A, equation (3.3.3), computed for the goal remains untouched for all the other candidate subgoals. Hence since only the tangency points are altered, only vector B', equation (3.3.6), will need to be

replaced. As mentioned before in equation (3.3.9) vector B' has each component, b_i', described as

$$b_i' = \sum_j (a_{ij} * W_{TSM_{ij}}) \qquad (3.5.10)$$

where a_{ij} is a coefficient of the matrix A defined in (3.3.3). The new vector, B_S, has its components, b_{Si} , defined as

$$b_{Si} = \sum_j (a_{ij} * W_{TSSM_{ij}}) \qquad (3.5.11)$$

The term $W_{TSSM_{ij}}$ in (3.5.11) can be replaced by the right side of (3.5.9), hence bs_i can be defined as

$$bs_i = \sum_j (a_{ij} * W_{TSM_{ij}} * f) = f * \sum_j (a_{ij} * W_{TSM_{ij}}) = f * b_i' \qquad (3.5.12)$$

So vector B_S can be defined as

$$B_S = f * B' \qquad (3.5.13)$$

It is necessary to normalise the system in (3.5.13) as mentioned in §3.1.1. The normalisation coefficient, presented in (3.1.7), being only dependent on the coefficients from matrix A, is the same for all the candidate subgoals' linear approximation since the matrix A is constant throughout the iteration. The new system for the candidate subgoal after normalisation looks like (3.5.14).

$$NA * X_S = NB_S = f * NB' \qquad (3.5.14)$$

where NA is the same matrix as defined in (3.3.10), NB_S is the vector B_S after normalisation has occurred and X_S is a weight space transition proposed by least squares.

Equation (3.5.14) can be rewritten as

$$NA * (\frac{1}{f} * X_S) = NB' \qquad (3.5.15)$$

The vector X computed for the goal solves this system in the least squares sense, therefore X_S can be computed directly according to equation (3.5.16)

$$X_S = f * X \qquad 0 < f < 1 \qquad (3.5.16)$$

Equation (3.5.16) shows that once a direction for the goal has been computed, then for each candidate subgoal the new solution, X_S, has the same direction and a proportional magnitude. Therefore it can be computed directly without having to use least squares for each candidate subgoal. As mentioned before this represents a major saving of computational effort when testing many candidate subgoals per iteration.

A line search is done along the direction computed for the goal to find the most progressive candidate subgoal that meets the criteria in §3.5.1.

The computational effort for an iteration involves solving a linear system once and testing one or more candidate subgoals. To solve a linear system using SVD, $4mn^2+8n^3$ operations are required, see §3.1.1.2. To test a candidate subgoal according to the criteria in 3.5.1, two forward passes, §2.2.2, are needed. One is to compute the candidate output subgoal and the other is to compute the output state associated with the rescaled weight space transition proposed by least squares.

It has been observed empirically that the number of subgoals to be tested per epoch decreases as the current state approaches the goal.

3.5.6. The Algorithm

In previous sections, the theory behind the new methodology was presented. In this section the theory is put together in an algorithmic version. The algorithm presented corresponds to a complete iteration. The results presented in chapter 5 are obtained using this algorithm.

Assume W_C to be the current weight state, O_C to be the output state obtained in W_C. Comments are presented in brackets to relate to text and equations from previous sections.

for each pattern i **do**

{

Find solution manifold;

[use steepest gradient direction and line search, §3.3]

Store point from the solution manifold as W_{SM_i};

Calculate $GV_i = W_{SM_i} - W_C$;

[Gradient vector definition in (3.3.4), §3.3.1]

[see figure 3.6]

Compute row i of matrix NA ; [eq. (3.3.7)]

Compute W_{TSM_i};

[Translation of W_{SM_i} according to equation 3.3.9)]

Compute nb'_i; [eq. (3.3.11)]

}

Use Least Squares to solve $NA * X = NB'$; [eq. (3.3.12)]

set $f = 1$;

set *end* = FALSE;

do

{

Compute least squares weight state : $W_{LS} = W_C + X$; [eq. (3.3.13)]

Test W_{LS} to check if the criteria for closeness are satisfied;

[§3.5.1]

if (criteria are satisfied) {

$W_C = W_{LS}$;

end = TRUE;

}

else {

$f = f * 0.5$;

$X = X * f$;

}

} **until** (*end* = TRUE).

3.6. Conclusion

In §3.1 a new way of combining the patterns' individual information was presented. The concept of solution manifold was introduced as being the error/weight contour with zero error for the individual error/weight surfaces. An alternative view of a goal weight state was proposed in which a goal weight state is considered to be a closest point to all the solution manifolds.

An approach to explore the potential of the solution manifold view using linear approximations to non-linear solution manifolds was presented in §3.2 and §3.3. In non-linear systems the use of linear approximations provides a good estimate of the goal position if the goal weight state is close enough. As the distance between a goal weight state and the current weight state is increased the linear approximations estimate tends to get worse in the general case. In this case it is the step size prediction that suffers the most. Direction has been found empirically to be good enough to pursue in §3.3. The direction proposed with linear approximations can be followed by line search to find the best optimal step size. Two possible measures to minimise using line search were presented in §3.4 for this end.

Also, based on the principle that close is good, the subgoal concept was introduced in §3.2. If a subgoal is placed close enough to the current state then a good linear approximation towards the subgoal for both direction and step size is obtainable in the general case. If a chain of subgoals sufficiently close together linking the current weight state to the goal weight state can be constructed then convergence is guaranteed. Two subgoal selection processes were presented in §3.5 in which the subgoal once achieved reduces the output error towards the goal.

Linear approximations coupled with either line search or subgoals provide a complete technique for both exact and inexact root-finding of a system of non-linear equations where first derivatives are available.

4. Normalisation Issues

In chapter 3 it was mentioned that, when using a least squares approach, in order to obtain a solution that minimised the sum of Euclidean distances in weight space terms from the current weight state to the solution manifolds it was necessary to normalise the coefficients of the matrix and vector that formed the linear system.

In this chapter it will be shown how different normalisations, or even the absence of normalisation, affects the solution proposed by least squares.

Initially the difference between normalised and non-normalised systems is analysed in a general linear systems context. This analysis is followed by sections showing the implication of normalisation in a neural networks context.

First the single layer net case is dealt with, afterwards the multilayer case is explored. Only in the case of inexact solution is there a difference amongst the several approaches to be explored. As far as the exact case goes, all approaches provide the same solution. For this reason all problems and examples in this chapter are inexact solution cases.

Finally a proposal for normalisation including an error term is presented and analysed.

4.1. The Effects of Normalisation in Linear Systems

In this section an analysis is done to see what the differences are when using SVD with normalised and non-normalised systems. As mentioned before this section deals only with inexact solution cases since in the exact solution case the solutions for both systems coincide.

Consider for example the following equation of a hyperplane in a 2-D space

$$-0.5w_1 - 0.5w_2 = 0.5 \qquad (4.1.1)$$

The same hyperplane can be written as

$$-10w_1 - 10w_2 = 10 \qquad (4.1.2)$$

One could say that equation (4.1.2) is equal to equation (4.1.1) times 20. Despite this scaled equivalence, the effects of the two equations are very different. The least squares approach is scaling dependent, i.e. if the coefficients of one of the hyperplanes is scaled by a constant then the least squares approach will provide a different solution although the set of hyperplanes represented is the same. This can be seen as a type of weighted least squares, i.e. the hyperplanes with higher coefficients are stronger attractors than the ones with smaller coefficients.

Normalisation solves this problem. For instance if one normalises both (4.1.1) and (4.1.2) the equations obtained are both identical to equation (4.1.3).

$$-0.707w_1 - 0.707w_2 = 0.707 \qquad (4.1.3)$$

In a normalised system all hyperplanes are equally strong attractors. In the neural net's case this means that all patterns have the same weighting in computing the new direction.

It is possible that it may be convenient to have different weightings for each hyperplane as it will be shown in §4.4. In this case one should normalise the hyperplanes coefficients first so that each hyperplane has unit weighting, and afterwards one can multiply the coefficients by the desired weightings.

4.2. The Single Layer Net Case

An example of an inexact solution case is now presented. The solutions for both the non-normalised and normalised systems are discussed.

A net with two input units and one output unit is used to show the differences obtained because this enables a graphical representation of the weight space in 2-D. Lets assume there are three patterns, each with two inputs and a target excitation. The patterns are as presented in table 1.

Pattern	I_1	I_2	T
1	1	1	1
2	1	-1	1
3	0	1	-1

Table 4.1 - Inputs and targets for patterns

In this case there is no exact solution, i.e. there is no weight state that satisfies all the patterns. In the single layer net case the hyperplane coefficients that represent the solution manifolds for each pattern are obtainable directly from the inputs and targets of the patterns as shown in §3.1.1.1. Figure 4.1 is a graphical representation in weight space of the solution manifolds for the three patterns. The numbers close to the lines refer to the pattern numbers in table 4.1.

Figure 4.1 also shows the two solutions obtained from using the normalised and the non-normalised systems. Each solution is optimal for the respective system.

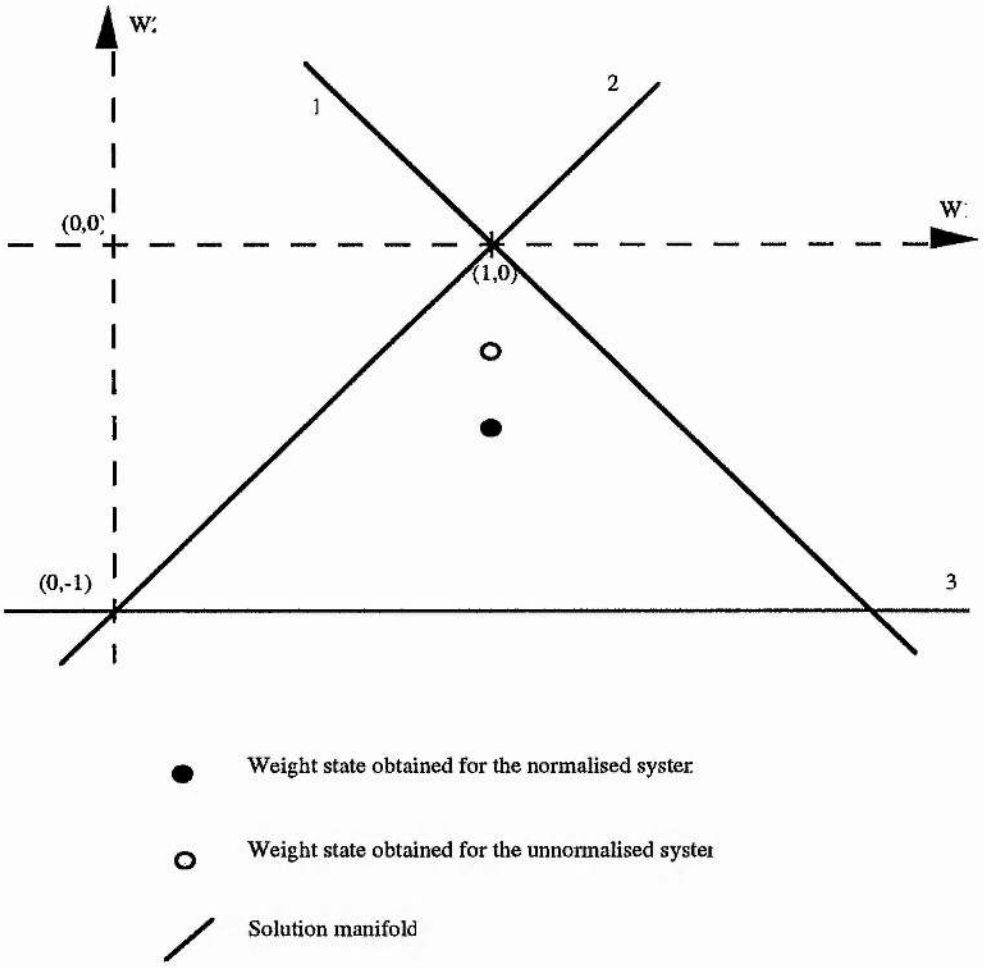


Figure 4.1 - The solution manifolds in weight space for the patterns in table 4.1 and the weight states obtained from both systems.

Which solution is better is a context dependent question and, as mentioned before, only applies to the inexact solution case. If the context is single layer networks with output functions as defined in equation (4.2.1) and error functions as defined in (4.2.2) then the non-normalised system gives an optimal solution because the function being minimised coincides with the output error function being minimised for the global minimum approach.

The output is defined as

$$o_j = f(ex_j) = ex_j = \sum_k i_{kj} w_k \qquad (4.2.1)$$

where i_{kj} is the k^{th} component of the input from pattern j to be fed into input unit k, and w_k is the weight connecting input unit k to the output unit.

Note that the output is a linear function of the input and that the relation is the same for every pattern. Although this last remark seems rather obvious in a neural network context it is important from a linear systems point of view.

The neural output error function is defined as

$$\sum_{j} (o_j - t_j)^2 \qquad (4.2.2)$$

where o_j is the output for pattern j and t_j is the target for the respective pattern. Equation (4.2.2) is equivalent to

$$\sum_{j} (\sum_{k} i_{kj} w_k - t_j)^2 \qquad (4.2.3)$$

Equation (4.2.3) coincides with

$$\| IW - T \|_2 \qquad (4.2.4)$$

which is the function being minimised in the non-normalised system.

If the output function does not obey the conditions stated above then the solution of the non-normalised system no longer coincides with the global minimum because in this case (4.2.2) is no longer equivalent to (4.2.3). The non-normalised system assumes the same linear relation between the output and the inputs for every pattern.

4.3. The Multilayer Net Case

In the multilayer net case the solution manifolds are not obtained directly from the patterns. For each pattern, line search must be performed along the steepest gradient descent direction to find the solution manifold.

As mentioned before in §3.2, the solution manifolds are not linear, the approach taken is to take linear approximations to the solution manifolds at the tangency points.

For reasons of simplicity the solution manifolds in the example that follows are idealised by straight lines. The same solution manifolds used in the single layer net case are used in this section but now the solution manifolds have to be found through line search along the steepest gradient descent direction.

The process of finding the solution manifolds is now illustrated. First it is necessary to find the tangency points for each pattern. Assuming the current weight state to be (1,1), the following tangency points are found :

- pattern 1 : (0.5,0.5)
- pattern 2 : (1.5,0.5)
- pattern 3 : (1,-1)

According to equation (3.3.4) the gradient vector for each pattern is :

- pattern 1 : (-0.5,-0.5)
- pattern 2 : (0.5,-0.5)
- pattern 3 : (0,-2)

Figure 4.2 shows the gradient vectors and tangency points found.

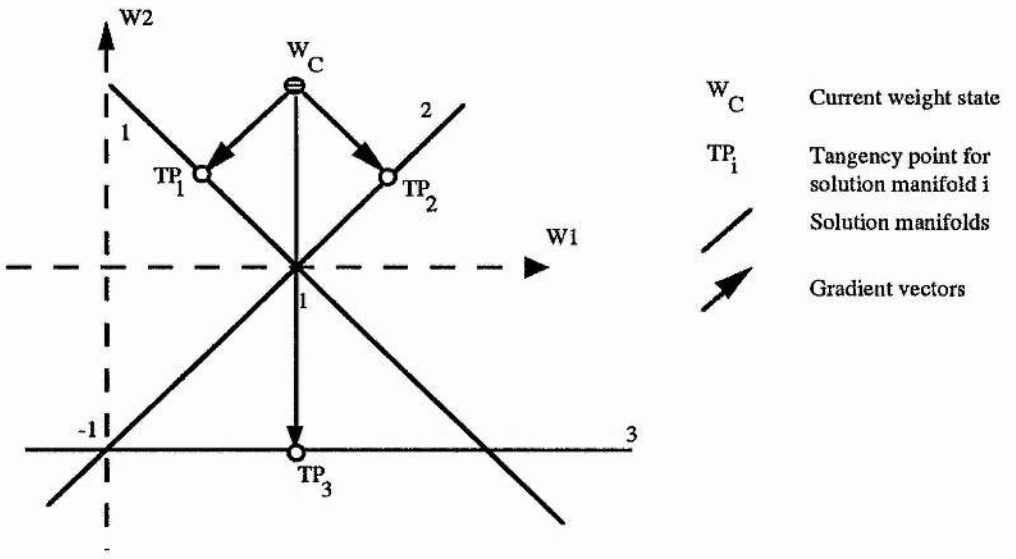


Figure 4.2 - Simplified solution manifolds for the multilayer net case.

According to §3.3, a translation needs to be made to a new frame of reference where the current weight state is the origin. The equations for the linear approximations in this new frame of reference without normalisation found for each pattern can be computed according to equations (3.3.5) and (3.3.10).

The equations obtained for the solution manifolds in this new frame of reference are :

- pattern 1 : $-0.5w_1 - 0.5w_2 = 0.5$
- pattern 2 : $0.5w_1 - 0.5w_2 = 0.5$
- pattern 3 : $-2w_2 = 4$

Solving this non-normalised system, using SVD and assuming that the current weight state is (1,1), equation (3.3.3) gives a solution weight state (1,-0.888889). If the system is normalised according to (3.3.7) and (3.3.11) then solving the system in equation (3.3.12) using SVD produces the solution weight state (1,-0.5).

Let us assume now that the current weight state is (1,0.1). The solution for the non-normalised system is now different, (1,-0.995885), yet the problem is the same. The solution for the normalised system however, remains constant at (1,-0.5). A solution for a normalised system is a fixed point for linear systems.

The reason for this discrepancy in the non-normalised system is based on the fact that the same hyperplane can be defined with different equations. In §4.1 it was explained why different coefficients affect the solution.

In the multilayer net case the equation of the hyperplane that represents the solution manifold for a pattern is a function of the current weight state. This is because different current weight states provide different tangency points and different gradient vectors. For this reason the non-normalised system provides different solutions for different current weight states.

The quality of the solution obtained for the non-normalised system depends on the current weight state. This is because as mentioned above, different current weight states will provide different solution weight states.

The quality of the normalised solution is optimal according to the error measure defined in §3.1.1.1, i.e. the error is the sum of Euclidean distances in weight space terms from the solution weight state to the solution manifolds.

4.4. Normalisation Including an Error Term

When using the normalised system described above the output error term is neglected. For each pattern the output error for each output unit is defined as

$$error_j = t_{ij} - o_{ij} \qquad (4.4.1)$$

where t_{ij} represents the target for pattern i and output unit j, and o_{ij} represents the output obtained with the current weight state for pattern i and output unit j.

In the normalised system each pattern is taken into account according to the Euclidean distance from the current weight state to the solution manifold. In function approximation problems, this may prevent the network from reaching a solution if there is no exact solution, i.e. if there is no weight state that is common to all the solution manifolds. It may be the case that despite the fact that the solution weight state is as close as possible in Euclidean distance terms to all the solution manifolds some patterns have a much higher output error than others, including having an output error above the desired output error tolerance.

Also if a large tolerance is in use then not paying attention to the output error term may slow performance. This is because in this case those patterns which have output errors higher than the output error tolerance are not given larger weightings in constructing the new direction over those patterns which have output errors below the specified tolerance. By giving larger weightings to the patterns with higher error then these patterns become stronger attractors and by consequence the goal region may be achieved earlier.

In these contexts it makes sense to pay attention to the output errors obtained for each pattern in the sense of trying to decrease the output errors for patterns with higher output error at the expense of patterns with lower output error.

A normalisation including an error term is now presented. The main objective is to give larger weightings to the hyperplanes from those patterns which have a higher neural error. For example, when two patterns have different output errors but their solution manifolds are at the same Euclidean distance in weight terms from the current weight state it is necessary to get closer in weight space to the pattern with higher output error in order to decrease the squared sum of output errors. This is a problem that the standard

normalisation can't solve since the current weight state is already as close as possible in Euclidean distance in weight space terms to all the solution manifolds which is the goal for standard normalisation.

As mentioned before, the normalisation gives all hyperplanes the same weightings in the linear system. If the normalised hyperplane coefficients are multiplied by the ratio (output error)/distance associated with them then the hyperplanes with higher ratio are stronger attractors than the ones with a lower ratio, i.e. they have a larger weighting and therefore influence the solution more.

Normalisation based only on Euclidean distance in weight space terms can be seen as the pure method according to the alternative view of the goal based on the solution manifolds. Normalisation with error is a practical version for minimising least squared errors.

The error normalised system can be written as

$$ENA * W = ENB' \tag{4.4.2}$$

in which ENA is a matrix whose components ena_{ij} are defined in equation (4.4.3) and ENB' is a vector whose components enb'_i are defined by equation (4.4.5) (see below). That is,

$$ena_{ij} = \frac{na_{ij} * e_i}{dist_i} \tag{4.4.3}$$

where na_{ij} is defined by equation (3.1.8), e_i represents the absolute value of the distance between the output obtained and the target for pattern i, and $dist_i$ representing the Euclidean distance in weight space terms from the current weight state to the solution manifold for pattern i is defined as

$$dist_i = \frac{| \sum_j a_{ij} * w_i - b'_i |}{\sqrt[2]{\sum_j a_{ij}^2}} \qquad (4.4.4)$$

where a_{ij} is defined as in (3.3.5) and b'_i is defined in (3.3.10).

Equation (4.4.5) defines the elements enb'_i from the vector ENB'.

$$enb'_i = \frac{nb'_i * e_i}{dist_i} \qquad (4.4.5)$$

where nb'_i is defined as in (3.3.11).

It is now proved that using a error normalisation as described above corresponds to a least squares output error problem, which is the standard measure for backpropagation.

Assume that W_S is a solution. Then

$$ENA * W_S = P \qquad (4.4.6)$$

where P is such that it minimises

$$\| P - ENB' \|_2 \qquad (4.4.7)$$

Replacing P in (4.4.7) by the left side of (4.4.6), the quantity being minimised becomes as in (4.4.8).

$$\| ENA * W_S - ENB' \|_2 \qquad (4.4.8)$$

This in turn means minimising

$$\sum_i \left(\sum_j \frac{na_{ij} * w_{sj} * e_i}{dist_i} - \frac{nb'_i * e_i}{dist_i} \right)^2 \qquad (4.4.9)$$

which is equivalent to

$$\sum_i \left(e_i \frac{\sum_j na_{ij} * w_{sj} - nb'_i}{dist_i} \right)^2 \qquad (4.4.10)$$

For simplicity reasons each of the fraction members will be dealt with separately. The bottom part of the fraction has already been dealt with in equation (4.4.4). The top part of the fraction is now dealt with in equation (4.4.11).

$$na_{ij} * w_{sj} - nb'_i = \frac{\sum_j (a_{ij} * w_{sj} - b'_i)}{\sqrt[2]{\sum_j a_{ij}^2}} \qquad (4.4.11)$$

Comparing equations (4.4.11) and (4.4.4) we can see that equation (4.4.9) can be simplified to equation (4.4.12). This is because the bottom and top part of the fraction in (4.4.9) are equal in absolute value.

$$\sum_i e_i^2 \qquad (4.4.12)$$

Equation (4.4.12) states that the quantity being minimised is actually the sum of squared errors as desired, so this concludes the proof.

Note that SVD being a linear system assumes a linear relation between error and Euclidean distance in weight space terms. This is not the case with sigmoidal units so the system becomes iterative. This is not a new dimension to the problem though since the methods presented in chapter 3 are iterative by nature.

4.5. Conclusion

Normalisation allows SVD to compute the point which is closest in Euclidean distance to all the solution manifolds in weight terms. However in some

situations this solution may not be good enough from an output error point of view.

An error normalisation was presented that allows SVD to minimise the output squared error instead of the Euclidean distance in weight terms to the solution manifolds. In this way SVD error function is consistent with the standard error function for neural nets, the least mean squares error presented in §2.3.1.

5. Experiments on Tangent Hyperplanes

In this chapter the various versions of Tangent Hyperplanes described in both chapters 3 and 4 are tested.

In section 5.1 the XOR problem is tested with all versions of Tangent Hyperplanes. The main objective is to determine which of the versions proposed performs best for a small benchmark.

In the remaining sections the best version will be tested in more demanding problems like the 5 bit parity problem in §5.2, and the 2-spirals problem in §5.3. In §5.4 a function approximation problem is presented.

Results for standard backpropagation with momentum are also presented where possible in order to provide a comparison. Where available reports on other techniques will also be provided. In §5.5 a conclusion is made about the results obtained in the previous sections.

A small glossary of terms is now introduced in order to help the understanding of the experiments:

- Epoch: a training of the net with each member of the training set presented once.
- Tolerance: the maximum acceptable linear difference between the output obtained and the desired target. When all patterns have a difference smaller than the tolerance, training is halted.
- Trial: a training of the net from an initial random weight state until either the tolerance condition is met or a maximum number of epochs occurs.

For each of the problems tested there is a common set of parameters to all algorithms. These parameters are the training set, initial weight states, tolerance, and number of trials.

Then there are the parameters which are algorithmic specific. For example, the learning rate is specific to backpropagation whereas the maximum weight variation per epoch, L_w in equation 3.5.3, is specific to Tangent Hyperplanes with Subgoals.

For each algorithm specific parameter a set of values was tested. Tests were done for each possible combination of the algorithm specific parameter values considered. By test is meant running a number of trials using one of the algorithms with a certain combination of parameters. A random seed determines the initial weight values which are taken randomly from an interval for each trial. For each problem the same random seeds were used for all tests in order to guarantee that the results were not affected by having different starting conditions for each method.

For each method or version several tests are made for each particular problem. The results presented for each test consist of the following data:

- Percentage of successes, i.e. the number of trials that converged in less than the maximum number of epochs allowed;
- Average number of epochs for successful trials;
- Standard Deviation from the average;
- Number of epochs for slowest successful trial;
- Number of epochs for fastest successful trial.

Note that an epoch for Standard Backpropagation is much cheaper from a computational point of view than one epoch for Tangent Hyperplanes. An

epoch in a Tangent Hyperplanes algorithm implies using SVD which makes it heavier from a computational point of view than an epoch for standard backpropagation. In §3.5.5 an indication of the number of operations needed for SVD is given.

However the number of epochs required to solve a problem provides more information besides allowing time comparison. For instance, given identical starting conditions, if a method takes 10000 epochs to perform a certain task then its ability to find a good direction and magnitude for the weight transition in each epoch is inferior to the ability of a method which takes only 10 epochs. It is this latter information that is more important here due to the concern of the thesis with robustness.

The standard deviation from the average provides useful information about the consistency of the algorithms in the sense that it tells about how an algorithm reacts to initial conditions. When considering a series of numbers, the number of epochs in this case, the standard deviation measures the variation of the data around the average. So if two algorithms have the same average but with different standard deviations this implies that the data for the algorithm with higher standard deviation is more varied in the general case than the data for the algorithm with the smaller standard deviation. A small standard deviation implies in the general case that all the numbers in the series are closer to the average. Considering that the number of epochs for a trial is a function of the initial conditions, if an algorithm has a small standard deviation from the average then the algorithm has a smaller sensitivity to initial conditions than an algorithm with a higher standard deviation. The standard deviation can be seen as a measure for the consistency of an algorithm.

The values of the longest and the shortest run are useful for interpreting the standard deviation values. For instance one can have a small standard

deviation for a series of numbers in which there are a few values that differ greatly from the average with low deviations for the remaining majority of values of the series which are very close to the average. Having the information of the maximum and minimum numbers of the series helps to understand more fully how the numbers are distributed around the average.

5.1. The XOR problem

The main objective of this section is to select a version of Tangent Propagation based on results for the XOR problem to use for the three other problems tested afterwards in the following sections.

The architecture used in this section is a strictly layered architecture with 2 inputs, 2 hidden units in a single layer and an output unit. All units in the hidden and output layers use sigmoidal activation functions.

The training set used is as presented in table 5.1.

Input 1	Input 2	Target
0	0	0.05
0	1	0.95
1	0	0.95
1	1	0.05

Table 5.1 - The training set for the XOR problem

Each subsection that follows relates to a particular algorithm or version, however all relate to the training set and architecture described in this section.

5.1.1. Standard Backpropagation with Subgoals

In this section a subgoal version of Standard Backpropagation is presented. The subgoal concept was implemented in output space because in backpropagation this option does not have the disadvantages mentioned in chapter 3 for Tangent Hyperplanes.

For each pattern the output at the initial weight state is computed. Then the distance between the initial output value and the target is worked out. The distance is divided into n equal parts where n is the number of subgoals. The i^{th} output subgoal is computed as being

$$t_{ij} = o_j - (t_j - o_j) * \frac{1}{n} * i \qquad (5.1.1)$$

where t_{ij} is the target for subgoal i and pattern j, o_j is the initial output state for pattern j, and t_j is the final target for pattern j.

The set of t_{ij}'s, one for each pattern, becomes a temporary target to be achieved before proceeding to the next subgoal, subgoal i+1. Ten random weight states were tested and the number of epochs to reach the first output subgoal compared with the number of epochs needed to reach the final target.

Before proceeding to the testing phase one needs to establish when to consider that subgoal i has been achieved and therefore continue training aiming at subgoal i + 1. Considering that the tolerance used for Standard Backpropagation is Tol, a subgoal is considered to be achieved when all the patterns have errors below a tolerance as defined in eq. (5.1.2).

$$tol = \frac{Tol}{n} \qquad (5.1.2)$$

where Tol is the tolerance used for Backpropagation without subgoals and n is the number of subgoals. Equation (5.1.2) says that the tolerance used for the subgoal version is in the inverse proportion to the number of subgoals.

For comparison, Standard Backpropagation without subgoals was also carried out. The standard training was conducted until a tolerance of 0.35 was achieved. According to eq. (5.1.2) the tolerance used for the subgoal approach is 0.035 given that 10 subgoals were used.

The common technical data for both cases is the learning rate set to 1.0, the momentum value set to 0.9, and the interval where the initial weight states were taken from: [-0.1,0.1].

A small test with twenty initial random weights was done, and the number of epochs needed to achieve tolerance are presented in table 5.2. For the subgoal version the number of epochs presented is the number of epochs needed to achieve only the first subgoal.

WS	SBP	SG
1	2114	55492
2	4488	58551
3	472	6352
4	1480	20895
5	1004	13214
6	767	27457
7	1161	19700
8	8953	95959
9	1109	18038
10	682	8374
11	3890	51860
12	3034	171700
13	819	11556
14	1773	35431
15	5952	26764
16	921	10182
17	2683	25701
18	1665	18864
19	1748	20006
20	2865	54027

Table 5.2 - Results for Standard Backpropagation with and without Subgoals. WS stands for weight state.

The average for Standard Backpropagation (SBP) is 2379 epochs, and the average for the subgoal version (SG) is 37506 epochs. The subgoal version needs more than 15 times the number of epochs Standard Backpropagation needs. Furthermore, the results for the subgoal version relate only to the first subgoal so further training would be needed to solve the problem using the subgoal approach.

Using subgoals for standard backpropagation as described above consists only of resetting the targets as far as achieving the first subgoal is concerned. The data obtained from the experiments suggests that resetting the targets in this way results in a decrease in performance when compared to the original targets.

It may be argued that achieving the first subgoal is harder because the errors for each pattern are smaller than for Standard Backpropagation

without subgoals. Smaller errors have smaller gradients which in turn cause smaller weight jumps. In fact the errors in the first epoch for the first subgoal are n times smaller than for Standard Backpropagation, where n is the number of subgoals.

The following test which sets the learning rate for the subgoal version n times greater than the learning rate for Standard Backpropagation shows that increasing the learning rate is not sufficient to make the subgoal approach worthwhile.

The following table shows results for Standard Backpropagation with a learning rate of 1.0 and for the subgoal version with a learning rate of 10.0.

WS	SBP	SG*10
1	2114	4727
2	4488	5936
3	472	154
4	1480	3472
5	1004	1114
6	767	2156
7	1161	1633
8	8953	14158
9	1109	1756
10	682	203
11	3890	4122
12	3034	19322
13	819	907
14	1773	5066
15	5952	1123
16	921	704
17	2683	1294
18	1665	3572
19	1748	834
20	2865	200

Table 5.3 - Results for Standard Backpropagation and the Subgoal version with a learning rate 10 times superior to the one used for Standard Backpropagation.

According to table 5.3, the average for the subgoal version is 3622, which is still higher than the Standard Backpropagation version which is 2379. Although there is a clear improvement in the subgoal version results in comparison with the previous test, the average is still significantly higher. Furthermore, note that, as mentioned before, the results presented for the subgoal version refer only to the number of epochs needed to achieve the first subgoal.

The results presented in tables 5.2 and 5.3 show that further theoretical study and/or more powerful heuristics are needed to successfully apply the subgoal concept to Standard Backpropagation.

5.1.2. Standard Backpropagation

In this section results for the XOR problem are presented using Standard Backpropagation and momentum with several learning rate values.

Technical Data :

- Possible Values for the learning rate : { 0.1, 0.5, 1.0};
- Momentum : 0.9;
- For each learning rate value, 1000 trials were done starting from different initial weight states;
- Initial weight values belong to [-0.1, 0.1];
- Tolerance : 0.35;
- Maximum number of epochs per trial : 50000;

Table 5.4 shows the results obtained.

	lr = 0.1	lr = 0.5	lr = 1.0
% of success	86.4	99.4	99.3
Average	19449.28	5742.23	2960.43
St. Deviation	11394.22	5804.53	3291.50
Maximum	49845	45456	36982
Minimum	2972	655	358

Table 5.4 - Results for the tests for the XOR problem with 3 different learning rates.

These results, presented in table 5.4, show that standard backpropagation can have good convergence rates on the XOR problem if trained for long enough. Nevertheless the number of epochs needed to achieve such convergence rates in this simple problem is extremely high.

Further tests were made with different values for the maximum number of epochs to see how the convergence rates are affected. Table 5.5 presents the values of the convergence rates for maximum number of epochs equal to 2500, 5000 and 1000 epochs.

Max. Epochs	lr = 0.1	lr = 0.5	lr = 1.0
25000	65.7	97.1	98.9
5000	2.3	64.4	87.6
1000	0	1.8	15.9

Table 5.5 - Additional results for the tests for the XOR problem.

The results in table 5.5 show that if the maximum number of epochs is set to 25000 and 5000 the convergence rates are severely affected especially for learning rates below 1.0. The results for the maximum number of epochs set to 1000 are extremely poor as expected since this setting of the maximum number of epochs is well below the averages reported in table 5.4.

5.1.3. Tangent Hyperplanes with Line Search

In this section results are presented from tests done with both versions of Tangent Hyperplanes with Line Search described in §3.4.

Other than the parameters in common to all algorithms in this chapter, there is just one algorithm specific parameter in this version which is P, where P determines how much the error is allowed to increase per epoch. This is because it may occur that when following the direction proposed by the line search the error cannot be decreased, see §3.4. P can be seen as a percentage value when multiplied by 100. High percentages are undesirable because they may lead to oscillatory paths. Extremely low percentages on the other hand, while keeping the oscillation under control, tend to have very small step sizes.

Having this in mind the values tested in this experiment were 10%, 5% and 1%. The complete technical data for this experiment is as follows :

- Possible Values for P : {0.01, 0.05, 0.1}.
- Number of trials : 1000
- Tolerance : 0.35
- Maximum number of epochs : 1000
- Initial Weights belong to [-0.1,0.1]

Note that the maximum number of epochs selected is the lowest value tested for Standard Backpropagation in §5.1.2.

For each type of line search tests were done with the normalised and error normalised systems. Table 5.6 presents the results obtained with each version

of Tangent Hyperplanes with Line Search using output error for the line search.

	Normalised			Error Normalised		
	P = 0.01	P = 0.05	P = 0.1	P = 0.01	P = 0.05	P = 0.1
% success	98.9	98.8	98.9	99.5	99.5	99.5
Average	10.53	9.63	10.06	8.98	8.98	8.98
St. Dev	37.44	24.74	28.29	8.38	8.37	8.37
Max	905	611	611	239	239	239
Min	4	4	4	4	4	4

Table 5.6. Results for Tangent Hyperplanes with Line Search using Output Error.

The first impression from table 5.6 is that both the Normalised and Error Normalised versions are very robust. The average number of epochs is similar for both techniques. The main difference is in the standard deviation results which indicate that the error normalised version is more consistent than the normalised version. This may be due to the fact that the line search measure is in more agreement with the error normalisation measure because both measures are based on output error.

The next table, 5.7, shows the results using Euclidean error for the line search.

	Normalised			Error Normalised		
	P = 0.01	P = 0.05	P = 0.1	P = 0.01	P = 0.05	P = 0.1
% success	98.9	99.1	99.4	99.3	99.2	99.6
Average	10.61	9.51	8.44	10.27	9.14	8.65
St. Dev	17.32	14.40	2.60	5.53	3.96	4.19
Max	542	452	23	107	69	93
Min	4	4	4	4	4	4

Table 5.7. Results for Tangent Hyperplanes with Line Search using Euclidean Error.

Table 5.7 shows that line search with Euclidean error is also very robust on this problem. Both the percentage of successes and the average number of epochs are similar. There is a significant difference in the standard deviations but this can be due to the setting of P. If P is set to 0.1 then the Normalised version has a lower standard deviation whereas for the other settings of P it is the Error Normalised version which has the lowest values.

In general the results obtained show that there are no significant differences for this problem between the Euclidean solution and the Standard output error based solution. However, it is possible that for problems with small output tolerances the differences could turn out to be significant.

5.1.4. Tangent Hyperplanes with Subgoals

This section reports on the results obtained when using the method described in §3.5 and variations to be found in chapter 4.

In all the versions reported on here there are three parameters involved: L_W, L_O and T. All these parameters are described in detail in chapter 3. A summary description of these parameters, is that L_W, stands for the maximum variation

in value per weight per epoch, L_O represents the maximum variation in output space, and T defines a minimal step size in output space.

For each of the parameters 3 different values were tested:

- L_W belongs to {1.0, 0.5, 0.1} ;
- L_O belongs to {0.1, 0.05, 0.01};
- T belongs to {0.1, 0.001, 0.00001}.

For each triple $\{L_W, L_O, T\}$ 1000 trials were done. The maximum number of epochs for each trial is 1000 epochs which is the lowest value tested for Standard Backpropagation in §5.1.2. Tolerance is set at 0.35.

The following table, 5.8, shows the percentage of successes obtained with different combinations of parameters for two versions of Tangent Hyperplanes with Subgoals. One version, called THSN, Tangent Hyperplanes with Subgoals and Normalisation, uses the normalisation of the hyperplanes coefficients as described in §3.5, and the other version, called THSE, Tangent Hyperplanes with Subgoals and Error normalisation, uses the linear error normalisation as described in §4.4.

	THSN			THSE		
T = 0.00001	Lo = 0.1	Lo = 0.05	Lo = 0.01	Lo = 0.1	Lo = 0.05	Lo = 0.01
Lw = 1.0	99.7	99.7	99.8	100.0	100.0	100.0
Lw = 0.5	99.7	99.7	99.8	100.0	100.0	100.0
Lw = 0.1	99.3	99.3	99.2	100.0	100.0	100.0
T = 0.001						
Lw = 1.0	99.8	99.8	99.9	99.9	99.9	100.0
Lw = 0.5	99.8	99.8	99.9	100.0	100.0	100.0
Lw = 0.1	99.3	99.3	99.4	100.0	100.0	100.0
T = 0.1						
Lw = 1.0	99.8	99.8	99.9	100.0	100.0	100.0
Lw = 0.5	99.8	99.8	99.9	100.0	100.0	100.0
Lw = 0.1	99.3	99.3	99.4	100.0	100.0	100.0

Table 5.8 - Success rates for the XOR problem using both the THSN version and the THSE version.

A first analysis of table 5.8 shows that both versions are extremely robust for the parameter combinations tested having success rates between 99.2% and 100%. Further analysis shows that the normalised version at least is more sensitive to parameter Lw than to Lo or T for the values tested.

When comparing both versions one can see that the version using the Error Normalisation performs better with a superior success rate. The next table, 5.9, provides more data on the same experiments for the Error Normalisation version.

T = 0.00001	Lo = 0.1	Lo = 0.05	Lo = 0.01	Data
Lw = 1.0	12.80	12.81	14.48	Average
	3.72	3.73	3.815	St. Deviation
	88	88	90	Max
	9	9	10	Min
Lw = 0.5	15.75	15.76	15.96	Average
	3.76	3.77	3.80	St. Deviation
	92	92	92	Max
	12	12	12	Min
Lw = 0.1	49.21	49.21	49.6	Average
	9.15	9.19	10.38	St. Deviation
	154	154	189	Max
	36	36	36	Min

T = 0.001	Lo = 0.1	Lo = 0.05	Lo = 0.01	Data
Lw = 1.0	9.17	9.18	11.48	Average
	2.82	2.82	2.74	St. Deviation
	39	39	34	Max
	6	6	7	Min
Lw = 0.5	12.70	12.71	13.04	Average
	2.41	2.41	2.40	St. Deviation
	30	30	34	Max
	8	8	9	Min
Lw = 0.1	48.56	48.53	48.92	Average
	6.38	6.37	6.77	St. Deviation
	102	102	105	Max
	36	36	36	Min

T = 0.1	Lo = 0.1	Lo = 0.05	Lo = 0.01	Data
Lw = 1.0	8.91	8.93	11.48	Average
	2.74	2.74	2.74	St. Deviation
	31	31	34	Max
	5	5	7	Min
Lw = 0.5	12.69	12.70	13.04	Average
	2.31	2.33	2.40	St. Deviation
	23	23	34	Max
	8	8	9	Min
Lw = 0.1	48.48	48.46	48.92	Average
	6.18	6.18	6.77	St. Deviation
	101	96	105	Max
	36	36	36	Min

Table 5.9 - Additional results for the XOR problem using the Error Normalised version.

Table 5.9 confirms that as observed for the Normalised version the Error Normalised version of the method is also more sensitive to the parameter Lw than to Lo or T. A smaller Lw implies more epochs on the average, this is to be expected since Lw controls the maximum variation per weight per epoch. The same effect is produced when reducing Lo or T although to a much lesser extent.

Note that the analysis being made is based on the tested values for Lw, Lo and T. From a more general point of view it looks like there is a floor effect with Lo not set low enough to make an impact.

The averages for the number of epochs obtained are in general very low when compared to Standard Backpropagation, being comparable with the ones obtained for the line search version. The standard deviation values obtained in general are also relatively very small compared to Standard Backpropagation, being better in general than those obtained for the line search versions.

5.1.5. Conclusion

Standard backpropagation can be made to converge almost 100% on the XOR problem if one lets the simulation run for long enough and with the right parameters. Nevertheless the number of epochs needed are extremely high when compared to any of Tangent Hyperplane versions. The number of epochs needed for Tangent Hyperplanes are two orders of magnitude lower than the best results for Standard Backpropagation.

This empirical evidence confirms the theoretical suppositions made in chapters 3 and 4 that the Tangent Hyperplane technique has a powerful ability to find a good direction as well as a good magnitude for weight transitions. Also the small values obtained for the standard deviation confirm the consistency of the Tangent Hyperplanes' methods.

Webb et al (1988) report a convergence rate of 95.4% with the Levenberg-Marquardt technique on this problem. Reported results with Conjugate Gradient Descent and Quasi-Newton techniques provide worse results than the results obtained in here with the best settings for Standard Backpropagation.

It remains to choose a version of Tangent Hyperplanes to test on harder problems. From the results available the error normalised system seems to be superior to the normalised system with either subgoals or line search.

Line search is on the average faster than subgoals. However, the standard deviation is higher for the line search approach and this means less consistency in the results. Also higher convergence rates were obtained with the subgoal approach, in particular the percentage of successes for the subgoal error normalised version is 100% in the vast majority of the cases.

Since the next problems should be harder to solve, the version selected is the more robust and more consistent of all tested, namely subgoals with error normalisation.

5.2. The 5 Bit Parity Problem

In this section another problem is tested, the 5 bit parity problem. The network used has a strictly layered architecture with 5 input units, 5 hidden units in a single layer and 1 output unit. All the hidden units and output units have sigmoid activation functions.

Each input pattern of the training set has 5 inputs and is a combination of 1's and 0's. The target for the pattern is 0.95 if the number of inputs in the input pattern equal to 1 is odd, and 0.05 when the number is even.

All possible combinations of inputs are present in the training set so the training set has $2^5 = 32$ patterns.

5.2.1. Standard Backpropagation

The learning rates selected for this problem were 1.0, 0.5, 0.1. Momentum was kept at 0.9. 100 trials were done for each value of the learning rate. the initial weights for each trial were taken from the interval [-0.1, 0.1]. The maximum number of epochs per trial is 500000 epochs. This number is very large, but is needed to achieve even a modest amount of success.

The following table shows the results obtained.

	lr = 0.1	lr = 0.5	lr = 1.0
% of success	0	17	11
Average	n a	304818.1	56915.55
St. Deviation	n a	93884.4	77792.4
Maximum	n a	479548	282926
Minimum	n a	90412	3286

Table 5.10 - Results for the tests for the 5 bit parity problem with 3 different learning rates (na - not available, i.e. no convergences were achieved within 500000 epochs).

The number of failures reported in table 5.10 is extremely high for the parameters tested. Another interval for the initial weights was tested to see how backpropagation's performance varies. The new interval is [-0.5,0.5]. The results are presented in table 5.11.

	lr = 0.1	lr = 0.5	lr = 1.0
% of success	89	39	9
Average	102763.5	32236	104221.4
St. Deviation	110340.2	66906	124164.4
Maximum	485307	217684	377726
Minimum	3447	648	6643

Table 5.11 - Results for the tests for the 5 bit parity problem with 3 different learning rates with the interval for the initial weights set at [-0.5,0.5].

Although there is a significant increase in performance with this new interval, see table 5.11, the results obtained are still relatively poor specially for learning rates of 0.5 and 1.0.

Note that there is no claim here that the intervals tested are the best ones for this problem using standard backpropagation and that good performances could not be obtained for this problem with different parameters.

However, this example is significant because it shows that standard backpropagation's performance is very dependent on the initial conditions and the learning rate selected. With the wrong parameter settings standard backpropagation's performance can yield extremely poor results.

5.2.2. Tangent Hyperplanes

The version tested in this section is the one mentioned in §5.1.4, namely Tangent Hyperplanes with Subgoals and Error Normalisation.

Three parameters are involved: Lw, Lo and T. As for the XOR problem, for each of the parameters 3 different values were tested:

- Lw belongs to {1.0, 0.5, 0.1} ;

- Lo belongs to {0.1, 0.05, 0.01};
- T belongs to {0.1, 0.001, 0.00001}.

For each triple $\{Lw,Lo,T\}$ 100 trials were done. The initial weight states for each trial were selected randomly in the interval [-0.1,0.1]. Tolerance is set at 0.35.

As mentioned previously in §5.1.4, for the XOR problem the Tangent Hyperplanes average number of epochs was two orders of magnitude better than Standard Backpropagation's average. The maximum number of epochs for each trial was computed accordingly, i.e. it was set to be two orders of magnitude lower than the maximum number of epochs set for Standard Backpropagation in §5.2.1. Therefore the maximum number of epochs is set to 5000 epochs.

The following table, 5.12, presents the percentage of successes obtained using the above possible values for each parameter.

	THSE		
$T = 0.00001$	$Lo = 0.1$	$Lo = 0.05$	$Lo = 0.01$
$Lw = 1.0$	100	100	100
$Lw = 0.5$	100	100	100
$Lw = 0.1$	100	100	100
$T = 0.001$			
$Lw = 1.0$	95	96	100
$Lw = 0.5$	99	99	99
$Lw = 0.1$	100	100	100
$T = 0.1$			
$Lw = 1.0$	90	90	100
$Lw = 0.5$	99	99	99
$Lw = 0.1$	100	100	100

Table 5.12 Percentage of successes for the 5 bit parity problem using the selected version of Tangent Hyperplanes with Error Normalisation.

Table 5.12 shows that when one is dealing with harder problems T becomes an important parameter together with Lw. Lo is also a significant parameter in this problem when T is greater than 0.00001.

The Tangent hyperplanes algorithm seems to prefer small values for the settings of parameters. Note that for the lowest settings of each parameter the convergence rates are always 100% except in two cases where the convergence rate is 99%. As mentioned in §3.5, Lo and Lw are used to reinforce closeness. The smaller the values for these settings the closer the subgoal will have to be from the current weight state. Since the subgoal approach is based on closeness it is to be expected that better convergence rates are obtained with small settings. The parameter T controls the minimal step size when good subgoals are not found. Having a small setting for T

prevents the method from producing large weight jumps in regions where the linear approximation does not yield a good approximation, therefore keeping the descent under control.

The following table, 5.13, shows additional results when T = 0.00001.

T = 0.00001	Lo = 0.1	Lo = 0.05	Lo = 0.01	Data
Lw = 1.0	415.56	416.15	425.44	Average
	288.54	288.56	277.29	St. Deviation
	1504	1504	1354	Max
	48	48	52	Min
Lw = 0.5	422.48	422.48	430.43	Average
	274.04	274.04	287.35	St. Deviation
	1241	1241	1278	Max
	83	83	83	Min
Lw = 0.1	661.55	661.55	661.55	Average
	470.33	470.33	470.33	St. Deviation
	2342	2342	2342	Max
	177	177	177	Min

Table 5.13 Additional results for T = 0.00001 for the 5 bit parity problem using the selected version of Tangent Hyperplanes.

For T = 0.00001, see table 5.13, Lw is a more influential parameter than Lo for the values tested. In particular if Lw = 0.1 in this problem it seems as if Lo is of no importance. Recall that Lw determines the maximum variation per weight per epoch and Lo determines the maximum variation in output space per output unit per epoch. In this context the results can be interpreted as suggesting that, for the values of Lw used, the variation in output space actually obtained is in the general case less than the values of Lo tested. Lo seems not be set low enough to make an impact, i.e. the floor effect reported for the XOR problem seems to apply as well for the 5 bit parity problem when

T = 0.00001. However, as mentioned before, Lo is an influential parameter for convergence rates when T is larger than 0.00001.

As in the XOR problem, smaller values of Lw imply higher averages. This can also be explained by the fact that Lw limits the magnitude of the weight jump in an epoch.

As for the standard backpropagation case a second interval, [-0.5,0.5], for the initial weight states was also tested. The following table, 5.14, presents the percentage of success for this new interval.

	THSE		
T = 0.00001	Lo = 0.1	Lo = 0.05	Lo = 0.01
Lw = 1.0	99	98	98
Lw = 0.5	95	95	95
Lw = 0.1	87	87	87
T = 0.001			
Lw = 1.0	96	96	98
Lw = 0.5	96	96	96
Lw = 0.1	87	87	87
T = 0.1			
Lw = 1.0	91	89	96
Lw = 0.5	98	98	95
Lw = 0.1	87	87	87

Table 5.14 Percentage of successes for the 5 bit parity problem using [-0.5,0.5] as the interval for the initial weights.

Although there is a decrease in performance, the results presented in table 5.14 are still much better than the ones obtained with standard backpropagation. Also note that the difference in the results for the two intervals tested is not as significant with the Tangent Hyperplanes method as with Standard Backpropagation in terms of convergence rates. This suggests

that the Tangent Hyperplanes method is less sensitive to initial conditions regarding the interval where the initial weights are taken from than Standard Backpropagation.

The following table, 5.15, shows more data when T = 0.00001.

T = 0.00001	Lo = 0.1	Lo = 0.05	Lo = 0.01	Data
Lw = 1.0	417.01	397.19	391.24	Average
	314.02	226.78	210.96	St. Deviation
	2612	1299	1299	Max
	84	84	83	Min
Lw = 0.5	578.09	577.52	583.09	Average
	435.67	432.30	442.68	St. Deviation
	2713	2648	2950	Max
	140	140	142	Min
Lw = 0.1	1938.01	1938.01	1939.17	Average
	985.82	985.82	984.23	St. Deviation
	4816	4816	4816	Max
	306	306	306	Min

Table 5.15 Additional results for T = 0.00001 for the 5 bit parity problem using the selected version of Tangent Hyperplanes using [-0.5,0.5] as the interval for weight initialisation.

Comparing table 5.15 with the one obtained for the interval [-0.1,0.1], table 5.13, it is evident that not only the averages are higher but also the standard deviations obtained are worse. This may be a sign that the Tangent Hyperplanes technique works best if the starting position in weight space is close to the origin. Nevertheless these data alone are insufficient to draw that conclusion. Further work would be needed to establish this for the general case.

The data presented also suggests that when $Lw = 0.1$ the low success rate may due to the fact that the number of epochs needed for the longest run, max, is very close to the maximum number of epochs. To verify this statement one of the less successful parameter settings was tested with a higher maximum number of epochs, this time set to 15000. The success rate increased from 87% to 93% and the longest run took 11329 epochs.

5.3. The 2 Spirals Problem

This problem by Alexis Wieland, appears in Lang and Witbrock (1988). This is a classification problem with two classes. Each class of patterns forms a spiral and the two spirals are intertwined. The main objective is to find a neural network to separate the two classes of patterns. The I/O map is presented in figure 5.1. The patterns represented by black squares have a target of 0.95 and the patterns represented by white triangles have a target of 0.05.

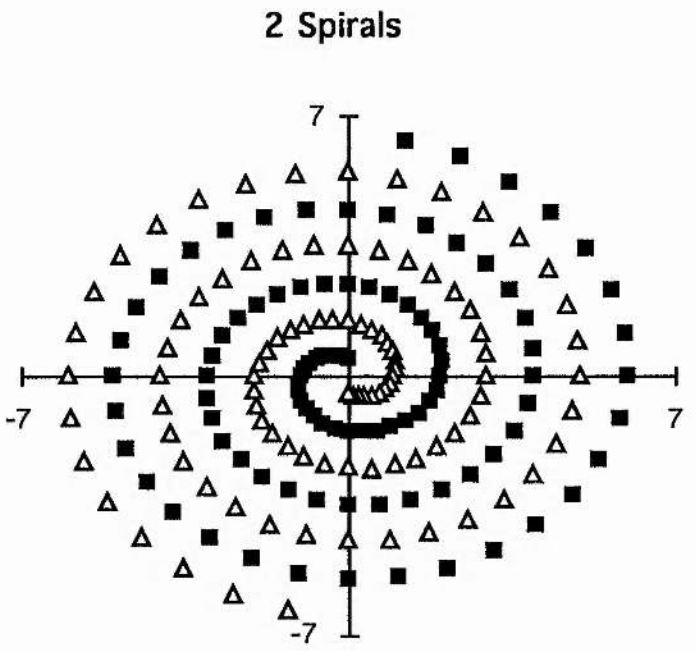


figure 5.1 - I/O Map for the 2 Spirals problem

This problem was classified as 'rather hard' in Ripley (1993). Baum and Lang (1990) found that there are many solutions for the 2 Spirals problem using a strictly layered architecture with 2 input units, 50 hidden units, and 1 output unit. However the method by Baum and Lang requires an 'oracle' able to provide the target classifications for input space points that do not belong to the given training set for the problem. In this section the net is trained only with the patterns present in the given training set.

The same authors claim that they were unable to reach a solution using either standard backpropagation or conjugate gradient descent methods. Baum and Lang (1990) tried to train a larger strictly layered net with 60 units in the hidden layer with conjugate gradient descent but still achieved no positive results.

Lang and Witbrock (1988) also claim the same failures for networks with strict Multilayer Feedforward networks as described in chapter 2. The networks that succeed in Lang and Witbrock's paper are jumped architectures, i.e. architectures where there are connections between the neurons in layer i, and all neurons in layers with indexes greater than i. Furthermore they didn't succeed at training with jumped architectures with less than 2 hidden layers and even with two hidden layers they didn't get robust convergence.

In this section the 2 Spirals problem was tackled using a strictly layered architecture with 50 hidden units in a single hidden layer. This architecture was selected, not because there is a belief that this architecture is more appropriate for the problem than the architecture selected by Lang and Witbrock, but simply because it is an example of a combination of training set and architecture where there were no reports of a solution being achieved with a fixed architecture using only the training set, e.g. without an oracle as in Baum and Lang (1990).

This is important for the work in the thesis. Up to now all the empirical evidence points to a better direction with Tangent Hyperplanes than with standard backpropagation. The question now is how much better is the Tangent Hyperplanes' direction? Can it converge in situations where neither standard backpropagation nor conjugate gradient descent have never converged? It is to answer these questions that this problem and this architecture have been selected.

Results from Tangent Hyperplanes with Subgoals and Error Normalisation are presented. Results from the gradient descent techniques are unavailable for the reasons given above.

A small preliminary test was made to see for which parameter values does the algorithm behaves best.

Three parameters are involved: Lw, Lo and T. As for the XOR and 5bit parity problems, for each of the parameters 3 different values were tested:

- Lw belongs to {1.0, 0.5, 0.1} ;
- Lo belongs to {0.1, 0.05, 0.01};
- T belongs to {0.1, 0.001, 0.00001}.

The technical data concerning the experiments is as follows :

- Tolerance : 0.35
- Number of trials : 1 for each different setting of parameters (more results will follow after this initial trial).
- Initial weights belong to [-0.1,0.1]
- Maximum number of epochs allowed : 10000

The following table, 5.16, presents the results obtained.

	THSE		
$T = 0.00001$	$Lo = 0.1$	$Lo = 0.05$	$Lo = 0.01$
$Lw = 1.0$	2269	2258	2379
$Lw = 0.5$	2292	2292	2387
$Lw = 0.1$	2340	2340	2351
$T = 0.001$			
$Lw = 1.0$	565	574	588
$Lw = 0.5$	641	648	634
$Lw = 0.1$	1548	1548	1545
$T = 0.1$			
$Lw = 1.0$	245	216	658
$Lw = 0.5$	403	327	737
$Lw = 0.1$	1629	1629	1843

Table 5.16 Numbers of epochs for the 2 spirals problem using the subgoal version of Tangent Hyperplanes with Error Normalisation.

No failures were found with the parameters tested in this case. This method has achieved a solution in each of the trials for a problem where both Standard Backpropagation and Conjugate Gradient Descent methods have reportedly failed, see Baum and Lang (1990), and Lang and Witbrock (1988).

More results are now presented with $T = 0.1$, the highest value tested, see table 5.17. Ten different initial weight states were tested for each possible setting of the remaining parameters, Lw and Lo.

$T = 0.1$	$Lo = 0.1$	$Lo = 0.05$	$Lo = 0.01$	Data
$Lw = 1.0$	242.5	217.6	604.6	Average
	46.93	24.72	107.42	St. Deviation
	320	266	673	Max
	175	175	455	Min
$Lw = 0.5$	367.7	345.9	650.5	Average
	62.30	33.92	94.27	St. Deviation
	471	387	743	Max
	261	265	465	Min
$Lw = 0.1$	1698.5	1683.3	1796.5	Average
	207.96	197.38	204.17	St. Deviation
	2116	2024	2167	Max
	1372	1372	1473	Min

Table 5.17 Additional results for $T = 0.1$ for the 2 spirals problem using the selected version of Tangent Hyperplanes.

Note that the lowest setting of Lo when Lw is set at 1.0 has a clear negative effect on performance. The effect is reduced as Lw is decreased. Lw controls the maximum weight jump per weight, per epoch. Therefore for lower values of Lw the variation in output space obtained is not sufficiently high to achieve the maximum variation allowed as determined by Lo . For larger values of Lw the maximum variation allowed in output space is higher and the criterion which determines the maximum variation in output space, Lo, comes into play.

The success rate was 100% for all tested parameter settings. This further confirms the robustness of the Tangent Hyperplane's version tested in here, for all the 108 tests on the 2 spirals problem no failures were reported.

This proves that, for this benchmark problem at least, this method is indeed extremely robust surpassing clearly both Standard Backpropagation and Conjugate Gradient Descent.

5.4. A Function Approximation Problem

All the previously tested problems were classification problems. In this section a function approximation problem is presented. Function approximation problems are common and so it would be useful to test the Tangent Hyperplanes algorithm on a problem of this type.

The function selected is

$$f(x) = 0.5 + (0.45 * sin(x) * cos(\ 2*x)) \qquad (5.4.1)$$

where x belongs to $[0,2*\pi]$.

A strictly layered network with one input unit, ten hidden units in a single layer and one output unit was used to train this problem. The training set consists of 20 patterns with inputs uniformly distributed in $[0,2*\pi]$ and targets computed according to (5.4.1).

For all tests the initial weight states are in the interval [-0.1,0.1]. The tolerance in a function approximation problem is usually much smaller than for classification problems. This is because whereas the latter one is only concerned with separating the patterns from both classes, in the former the network should give more precise answers, i.e. outputs very close to the targets. The tolerance used in this problem is 0.025.

For both methods, Tangent Hyperplanes and Standard Backpropagation 100 trials were done with the maximum number of epochs set to 100000.

5.4.1. Standard Backpropagation

For Standard Backpropagation the learning rates tested were 0.1,0.5 and 1.0. Momentum was set at 0.9. Table 5.18 presents the results obtained.

	lr = 0.1	lr = 0.5	lr = 1.0
% successful	98	43	1
Average	45462.56	17023.44	14519
St. Deviation	15325.65	16395.99	n a
Maximum	89172	83777	n a
Minimum	23384	5234	n a

Table 5.18. Results for the tests for the function approximation problem (na - not applicable).

The results in table 5.18 show that Standard Backpropagation is extremely sensitive to the learning rate used in this problem. Another interesting result is that the best convergence rate for this problem was obtained with a learning rate of 0.1 whereas in the 5 bit parity problem the best convergence was obtained with a learning rate of 1.0. This echoes the well-known fact that the best learning rate for a problem can be a very bad value for another problem. Møller (1993), for example, uses a learning rate of 0.2 for the 3 bit parity problem, and 0.01 for the 9 bit parity problem.

5.4.2. Tangent Hyperplanes

The report of the test done with Tangent Hyperplanes is now presented. For each of the parameters involved the following values were tested:

- L_W belongs to {1.0, 0.5, 0.1} ;
- L_O belongs to {0.01, 0.005, 0.001};

- T belongs to {0.1, 0.001, 0.00001}.

Note that the range of values for Lo is different from the ranges tested for all the previous problem although a common value is kept for comparison with the previous tests. The reasons for the new settings are:

- In this problem a smaller tolerance is used and it makes sense to be more restrictive in the maximum allowed variation in output space than in the previously tested problems which had a larger tolerance.
- In the previous problems lower settings of Lo seemed to have a positive effect on the convergence rates for training.

For each triple $\{L_w, L_o, T\}$ 100 trials were done. The maximum number of epochs for each trial is 100000 epochs as for Standard Backpropagation.

The following table shows the percentage success rates obtained with each combination of the parameters involved.

	THSE		
T = 0.00001	Lo=0.01	Lo=0.005	Lo=0.001
Lw = 1.0	99	97	99
Lw = 0.5	100	99	100
Lw = 0.1	100	100	100
T = 0.001			
Lw = 1.0	98	100	99
Lw = 0.5	99	100	100
Lw = 0.1	99	100	100
T = 0.1			
Lw = 1.0	100	100	99
Lw = 0.5	100	100	100
Lw = 0.1	100	100	100

Table 5.19. Percentage of successes for the function approximation problem.

Again the results in table 5.19 show that the method is very robust. As for the previously tested problems, using small values of Lo or Lw makes for

convergence rates of between 99% and 100%. The highest value for T provides the highest convergence rates as opposed to the 5 bit parity problem where the lowest setting for T provided the best results. This discrepancy can be explained by the different settings of Lo in the two problems. Note that in the 5 bit parity problem setting Lo at 0.01 also provided very high convergence rates.

The average number of epochs for Tangent Hyperplanes is one order of magnitude smaller than the average obtained for standard backpropagation.

5.4.3. Conclusion

The results for Standard Backpropagation show that this algorithm is extremely sensitive to the learning rate used in this problem. Although good results can be obtained for this algorithm they rely heavily on the choice of the learning rate.

Results for this problem using a Conjugate Gradient Descent method achieve 100% success. The average number of epochs for CGD is roughly 10 times the number needed for Tangent Hyperplanes.

The Tangent Hyperplanes algorithm gives robust convergence rates for all the tested settings of the parameters involved. In particular when T is set at 0.1 the convergence rate is 100% in 8 out of the 9 cases (the remaining case has a 99% convergence rate). When Lo or Lw are set with the minimum setting from the values tested the minimum convergence rate is also 99% with the majority of the cases having convergence rates of 100%.

5.5. Conclusion

In the XOR problem all versions of Tangent Hyperplanes were tested with several possible combinations of the relevant parameters. The results obtained show that all versions are extremely robust in this problem. The

extremely low number of epochs needed on average confirms the theoretical suppositions that the method has a very good ability to find good directions in weight space.

In all the other problems the best performing version of Tangent Hyperplanes was tested. The results show again a very high success rate combined with a low average numbers of epochs.

In the 5 bit parity problem the performance obtained with the interval [-0.1,0.1] is superior to the one obtained with the interval [-0.5,0.5]. This data may suggest that Tangent Hyperplanes works best with initial weight states closer to the origin but a larger set of tests involving more problems would be needed to confirm if this is the case.

An important result is the one relating to the 2 spirals. For this problem no convergences were obtained with the 2-50-1 architecture by Baum and Lang (1990) and single hidden layer architectures tested by Lang and Witbrock (1988) in a feasible amount of time with either standard backpropagation or conjugate gradient descent. The Tangent Hyperplanes version tested seems to be able to achieve a solution always regardless of the parameters tested.

In the function approximation problem very high success rates were also obtained regardless of the parameters used. Results on the same problem for Standard Backpropagation show that this method is extremely sensitive to the learning rate. Conjugate Gradient Descent is very robust in this problem but it takes approximately 10 times more epochs than the Tangent Hyperplanes algorithm.

In relation to the three parameters involved in the Tangent Hyperplanes algorithm a general analysis follows.

For the classification problems tested the best results overall for percentage of successes were found with T = 0.00001, i.e. the most restrictive setting used for this parameter provides the best percentage of successes.

In the function approximation problem it is the highest setting of T which gives best convergence rates. As mentioned before this discrepancy can be explained by the different settings of Lo in the two types of problems.

Overall, when Lw is set at 0.1 there are very good results in terms of percentage of successes, except in the 5 bit parity problem with [-0.5,0.5] as the interval for the initial weight states. In all the other cases the lowest value for the percentage of convergence rates is 99%. For Lw it is the smallest setting which provides better percentage of successes in both types of problems. Lo also follows this rule, providing the best results in terms of convergence rate success when the lowest setting is selected.

The fact that the smaller settings of Lw and Lo provide more robustness confirms the theoretical suppositions that closeness is required in order to achieve a good linear approximation. Recall that Lo and Lw were presented in §3.5.1 as a mean to ensure closeness.

The set of tests performed show that Tangent Hyperplanes is indeed a very robust algorithm and has a powerful ability to find good directions in weight space.

6. Classification Trees

The tangent hyperplanes algorithm in all its versions presented in chapters 3 and 4 is a function approximation type of algorithm. Although the tests reported in chapter 5 include classification problems these were tackled from a function approximation point of view. The adopted strategy was to minimise the amount of output error instead of minimising the number of misclassified patterns.

It might appear that the only difference between classification and function approximation problems is that in the former type one uses a much larger tolerance. However classification problems are inherently different from function approximation problems.

The notion of target or desired output is different for the two types of problem. Whereas from the function approximation point of view there is an interest in achieving outputs as close as possible to the respective targets, from the classification point of view the target only specifies the class to which the pattern belongs to. Two patterns with the same target belong to the same class.

From a classification point of view it seems more proper to define a target interval instead of an exact analogue target as in the function approximation point of view. For instance, considering a linear activation function, if a pattern has a target output of 1.0, then from a classification point of view the target output can be seen as the interval] 0 , $+\infty$]. For targets of -1.0 the target interval would be] $-\infty$, 0[.

The error function from a pure classification point of view is also different from the ones used in the function approximation point of view. A pattern in

a pure classification point of view is either correctly classified or misclassified. The amount of output error is not relevant.

In this chapter a classifier is presented. First an algorithm for linearly separable problems is presented in §6.1. For linearly separable problems the algorithm divides the training set into two subsets, each containing all the patterns from one class.

Linearly inseparable problems are tackled in §6.2. For linearly inseparable problems the main objective is to repeatedly divide the training set using a single hyperplane, i.e. a single layer network, each time in a fruitful way. Two criteria to divide the training set are presented.

In §6.3 one uses the hyperplanes found in §6.2 as building blocks of a larger structure, namely Classification Trees, in order to solve linearly inseparable problems. As the name suggests Classification Trees use a tree structure as opposed to a network structure. However Classification Trees can be converted to Neural Networks without any further training, see Sirat and Nadal (1990).

The 2 spirals problem tested previously for the Tangent Hyperplanes algorithm is again the problem selected to test the classifier in §6.4. Finally a conclusion is drawn in §6.5.

6.1. Linearly Separable Problems

In Neural Networks theory linearly separable problems can be solved with a single layer network.

A problem is considered to be linearly separable if there is a hyperplane in input space that separates the patterns from opposite classes.

Assume without loss of generalisation that the target for patterns in class A is positive and that the target for patterns in class B is negative. Then one can say that a problem is linearly separable if a vector W exists such that

$$\forall p \in \text{A}, w_0 + \sum_{j=1}^{n} w_j * p_j > 0.0$$

$$\forall q \in \text{B}, w_0 + \sum_{j=1}^{n} w_j * q_j < 0.0 \qquad (6.1.1)$$

where A is the set of patterns from one class, and B is the set of patterns from the opposite class. w_j is the j^{th} component of W, p_j and q_j are the j^{th} components of the respective input patterns p and q.

From a graphical point of view the required hyperplane in input space has all the patterns from A to one side and all the patterns from B in the opposite side of the hyperplane. The following figure presents a linearly separable training set and a hyperplane which separates the two classes.

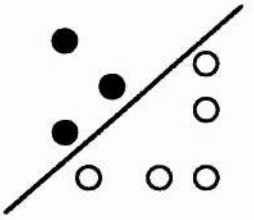


Figure 6.1. A hyperplane separating two classes, the black dots belong to one class, the white dots belong to the opposite class.

The definition presented in (6.1.1) implies that in order to solve a classification problem in n-D input space with a neural system one needs to solve a (n+1)-D system of linear inequalities.

From a formal point of view a classification problem is a system of linear inequalities, see (6.1.1). Yao (1990) reports an iterative algorithm using linear programming by Meggido (1983) which solves this type of system. Meggido's

algorithm transforms an n-D system of linear inequalities into a $(n-1)$-D problem, therefore achieving a reduction by 1 in the dimensionality of the original problem. Meggido also presents a technique to drop at least a quarter of the constraints, i.e. inequalities, in each epoch.

These are important results because in the general case the lower the number of dimensions the easier it is to solve the problem.

In this chapter a new algorithm is presented where a linearly separable problem posed in n-D input space is solved in $(n-1)$-D space. Considering that the neural version of the original n-D problem is $(n+1)$-D (see 6.1.1), this algorithm achieves a reduction in the dimensionality of the neural version by 2. Further work is needed to explore the possibility of dropping constraints in each epoch as in Meggido's algorithm.

The mechanism reduces the dimensionality by:

- decomposing the problem into one of finding a hyperplane parallel to a separating hyperplane and then translating the hyperplane into a separating position. In this way one weight dimension (the bias) is omitted from the new search problem.

- defining the search for a hyperplane in terms of the angles it makes in input space. This reduces the search by a further dimension.

The reduction of the number of dimensions of the search space is especially important for low values of n. For instance for a 2-D input space problem this method has a search space in 1-D, i.e. a line search is enough to find a separating hyperplane. As n grows the possible benefit of applying this mechanism decreases. Bishop (1995) suggests the use of Principal Component Analysis, discussed at length in Jollife (1986), or Fisher's Discriminant Analysis, Fisher (1936), in order to reduce the dimensionality of the original

problem. However if the dimensionality can't be significantly reduced to a low number then the technique may not bring significant benefits from a computational point of view.

The method is introduced for 2-D input space and then the generalisation to n-D, $n > 2$, is presented.

6.1.1. 2-D Input Space

The system that represents the problem of linear separability is:

$$\forall p \in A,\ w_0 + w_1 * p_1 + w_2 * p_2 > 0.0$$

$$\forall q \in B,\ w_0 + w_1 * q_1 + w_2 * q_2 < 0.0 \qquad (6.1.2)$$

In this section first it is shown how to reduce the dimensionality of the problem stated in (6.1.2), and afterwards a method to reduce the search space is presented.

6.1.1.1. Reducing the Dimensionality of the Problem

The system in (6.1.2) can also be represented as

$$\forall p \in A,\ w_1 * p_1 + w_2 * p_2 > k$$

$$\forall q \in B,\ w_1 * q_1 + w_2 * q_2 < k \qquad (6.1.3)$$

considering $k = -w_0$.

The linear separability problem can now be redefined as:

Definition 6.1

Find (w_1, w_2) such that there is a k that satisfies the system in (6.1.3).

Assuming that one has a pair (w_1,w_2) such that k exists, one knows that the outputs for patterns in class B must be smaller than k. Similarly, patterns in class A must have outputs larger than k.

This means that, considering the two variables $maxB$ and $minA$ defined in equations (6.1.4) and (6.1.5), inequality (6.1.6) must be verified for some (w_1,w_2).

$$maxB = \text{maximum } \{ out_p : p \in \text{B} \} \tag{6.1.4}$$

$$minA = \text{minimum } \{ out_q : q \in \text{A} \} \tag{6.1.5}$$

where out_x is the output for pattern x for a given pair (w_1, w_2) using a linear activation function.

$$maxB < minA \tag{6.1.6}$$

Therefore one can say for a given pair (w_1,w_2) that a value k that satisfies definition 6.1 exists by evaluating $maxB$ and $minA$ and checking if inequality (6.1.6) is verified.

Given a pair $(maxB, minA)$ the value k in 6.1.3 must lie in the interval $]maxB, minA[$. A possible selection for k is to select the midpoint of the interval, i.e.

$$k = (minA + maxB) * 0.5 \tag{6.1.7}$$

Therefore given a pair (w_1, w_2) such that inequality (6.1.6) is verified, the computation of the bias w_0 is trivial and can be done according to equation (6.1.7).

The above analysis allows the problem to be decomposed from finding a triple (w_0, w_1, w_2) that satisfies condition (6.1.1) to first finding a pair (w_1,w_2) that satisfies condition (6.1.6) and then computing a value of k (= $-w_0$) to provide

the triple (w_0, w_1, w_2). The decomposition means in practice that, first of all, an α can be sought to establish a hyperplane passing through the origin that provides (w_1, w_2). Secondly, this hyperplane can then be translated by k to separate the classes and yield (w_0, w_1, w_2).

The above concepts can be exemplified graphically.

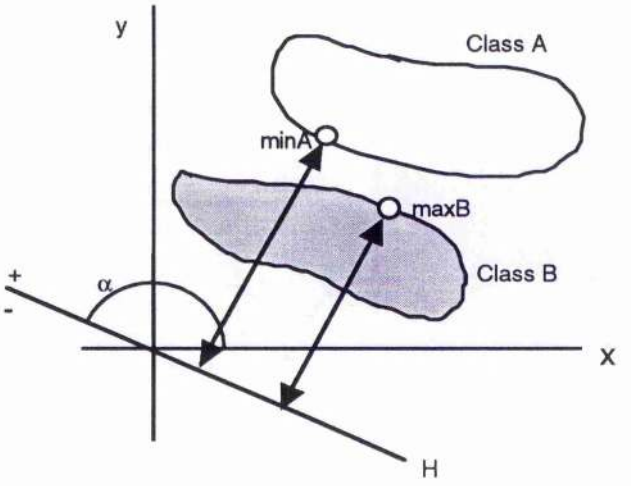


Figure 6.2. Graphical perspective of the concepts involved in this section.

From Figure 6.2 it can be seen that $minA\alpha > maxB\alpha$. This means that (6.1.6) is satisfied, i.e. there exists a bias value which corresponds to a shift of the hyperplane H defined by α, such that the hyperplane correctly separates the two classes.

The next figure uses the same training set, but a different angle α.

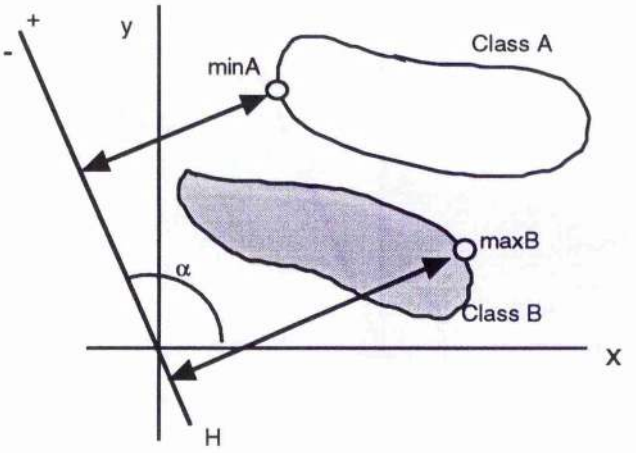


Figure 6.3. The same training set as in figure 6.2 but a different hyperplane.

The hyperplane in Figure 6.3 does not obey condition (6.1.6) and there is no bias value which will shift the hyperplane defined by α to become a separating hyperplane.

The angle finding part of the method is now described in more detail. The pair (w_1, w_2) defines a hyperplane which has the following equation:

$$w_1 * x + w_2 * y = 0 \qquad (6.1.8)$$

From a graphical point of view, this hyperplane passes through the origin, therefore this hyperplane can be defined by the angle it makes with one of the axes, see the figure below.

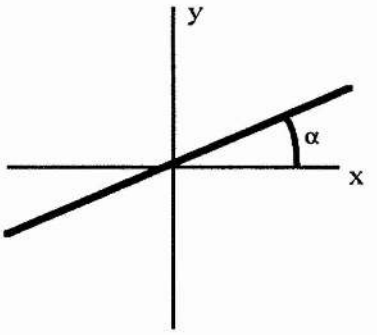


Figure 6.4. A hyperplane which crosses the origin can be defined by the angle it makes with one of its axes, in the figure the angle α of the hyperplane is relative to the x axis.

Given an angle α the values of (w_1, w_2) can be computed according to the following equations:

$$w_1 = \cos(\alpha)$$

$$w_2 = \sin(\alpha) \qquad (6.1.9)$$

This is equivalent to converting the point (w_1, w_2) from Cartesian coordinates to Polar coordinates considering that

$$\sqrt{x^2 + y^2} = 1 \qquad (6.1.10)$$

The problem of linear separability can now be defined in terms of the angle α, i.e. find an angle α which defines a hyperplane which in turn satisfies definition 6.1.

This can be done with a search on the possible values of α. Since the angle α must be in [0,360[the search has clear boundaries which is a major benefit. Searches with no clearly defined boundaries can become very lengthy.

There are at least two criteria to check if an angle defines a hyperplane which satisfies the definition 6.1. One is simply to check if inequality (6.1.6) is satisfied for the pair (w_1,w_2) computed according to equations (6.1.9).

A start to doing this can be made by rewriting the inequality (6.1.6) as

$$minA\alpha - maxB\alpha > 0 \qquad (6.1.11)$$

where $minA\alpha$ is the minimum of the outputs from patterns in class A obtained at angle α, and $maxB\alpha$ is the maximum of the outputs from class B obtained at angle α.

Therefore the search for an angle α could have as its main goal to maximise the following function:

$$r(\alpha) = minA\alpha - maxB\alpha. \qquad (6.1.12)$$

The following example shows that this function has local maxima even in linearly separable problems. Therefore if one used this function for the search of the angle α then a simple line search procedure wouldn't work, i.e. the entire search space between 0 degrees and 360 degrees would have to be searched.

Consider for instance the following training set:

Class	In1	In2
A	1	1
	1	-1
	2	2
	3	3
	5	1
B	-1	3
	-1	-2
	-3	-1
	-5	-2
	-4	-5

Table 6.1. Training set with two classes

From a graphical point of view the training set is represented according to figure 6.5.

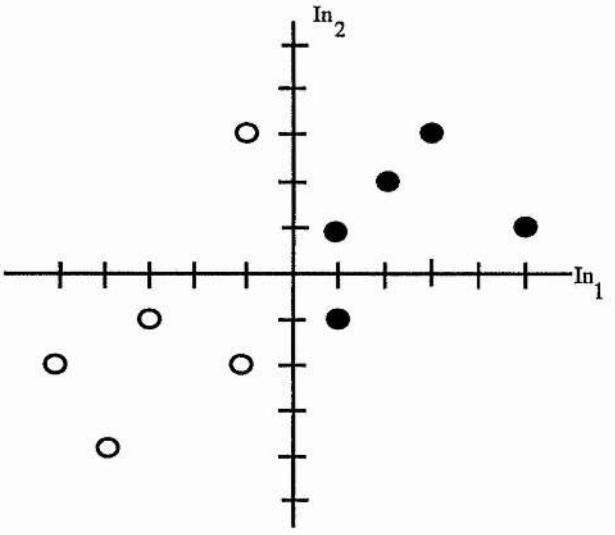


Figure 6.5. Graphical representation of the training set in table 6.1.

The following figure presents a graphic showing the value of $minA\alpha - maxB\alpha$ for α in [0,360[.

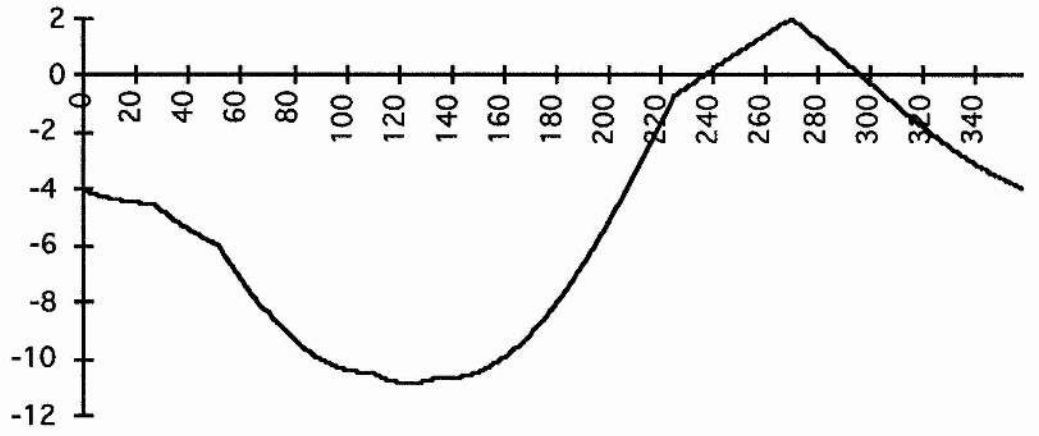


Figure 6.6. The value of minAα - maxBα for α in [0,360[.

In the next figure a closer look is taken at the interval [100,160].

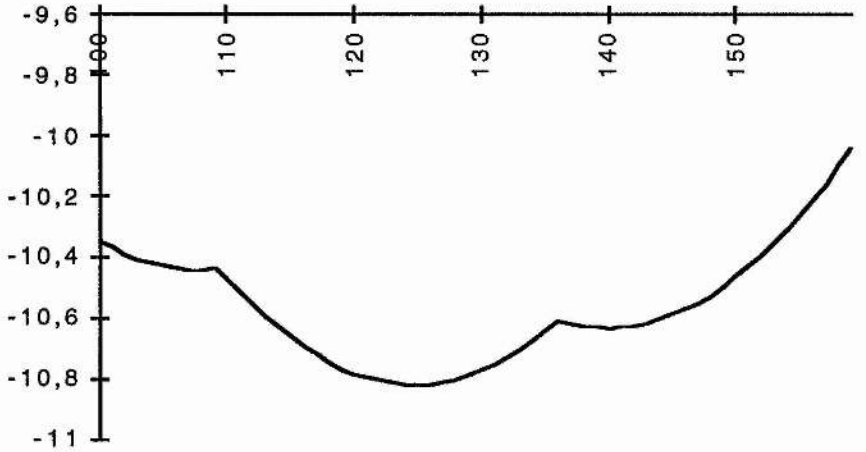


Figure 6.7. The value of minAα- maxBα for α in [100,160].

From figure 6.7 it is clear that the function $minA\alpha - maxB\alpha$ can have several local maxima in the interval [0,360[.

The problem can be simplified since this chapter is dealing only with classification problems, i.e. any angle which verifies inequality (6.1.11) would solve the problem. The objective of the search for an angle doesn't have to be to maximise the function r in (6.1.12), finding a value which

satisfies (6.1.11) is enough. The search for a solution can therefore stop as soon as (6.1.11) is satisfied.

Another possible criterion to find an angle α which defines a hyperplane parallel to a separating hyperplane is based on the number of correctly classified patterns.

A criterion to check if the angle α satisfies definition 6.1 is to verify if all patterns from class B satisfy (6.1.13), i.e. to maximise the following function f:

$$f(\alpha) = \#\ \{\ i \in \mathrm{B} : out_i\alpha < minA\alpha\} \qquad (6.1.15)$$

where $out_i\alpha$ is the output obtained for pattern i for angle α, and $minA\alpha$ is the minimum of the outputs for patterns in A obtained at angle α.

The maximum of this function, #B, is obtained when all patterns from B are correctly classified.

The following figure shows a graphic where the number of correctly classified patterns from class B is shown for α in [0,360[. Note that there are five patterns in class B so the maximum is five.

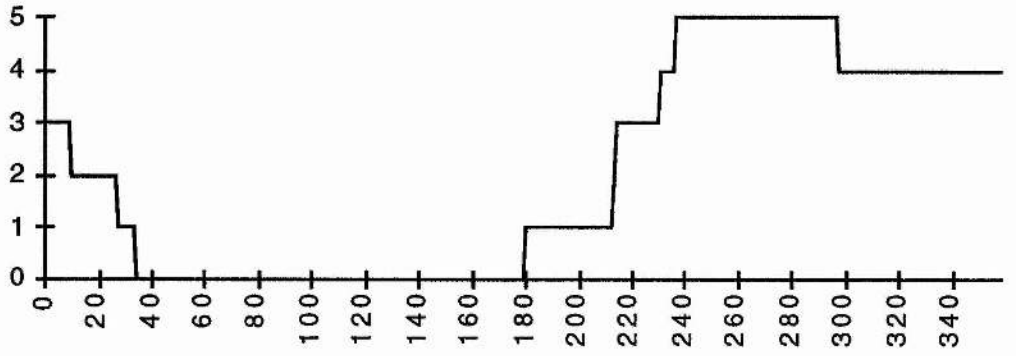


Figure 6.8. Shows the number of correctly classified patterns in B for α in [0,360[.

This function has no local maxima if the training set is linearly separable as shown next.

If there is a local maximum then there is at least one pattern from B which is correctly classified in the disjoint intervals containing the local and global maxima and misclassified inbetween. However a pattern cannot be correctly classified in disjoint intervals by a rotating hyperplane. Therefore there can be no local maxima for function f in (6.1.15) for linearly separable problems.

Since there are no local maxima a line search procedure can be applied to search for an angle α which satisfies definition 6.1. For this reason this latter criterion is preferable to the former criterion presented for linear separable problems.

Whichever the criterion applied, the neural version of the separability problem, which is a 3-D problem, has been transformed into a 1-D problem. With this mechanism the dimension of the original problem is reduced by two.

6.1.1.2. Reducing the Search Space by Half

Let's consider the two hyperplanes defined by the angles α and β where

$$\alpha = \beta + 180 \qquad (6.1.16)$$

From a graphical point of view the two angles represent the same hyperplane orientation. From a classification point of view the pattern's outputs obtained with angles α and β are the negatives of each other. This is because the weights are computed according to 6.1.9.

The following figure shows the two angles defining the same hyperplane orientation, although, as mentioned before, the outputs obtained for each angle are the negatives of each other.

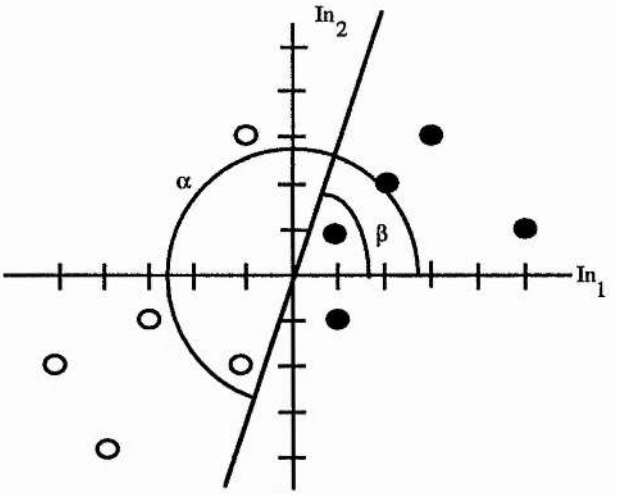


Figure 6.9. A hyperplane which separates the patterns from the training set in table 6.1.

In this section it is implicitly assumed that the targets are used only to identify the classes to which the patterns belong, i.e. two patterns with the same target belong to the same class. The targets are used to test for separability only as opposed to necessarily correct classification values.

The implications of the property described above are now presented for both criteria in the previous section.

First the criterion which is based on the positive solutions of $minA\alpha - maxB\alpha$ is explored. Afterwards, the criterion based on the maximisation of the number of correctly classified patterns from class B is tackled.

If the outputs obtained for angle α are the negatives to the outputs obtained for angle β then

$$minA\alpha = -maxA\beta$$

$$maxB\alpha = -minB\beta \qquad (6.1.17)$$

where $minA\alpha$ and $maxA\beta$ are respectively the minimum and maximum outputs for patterns in class A obtained for angles α and β, and $maxB\alpha$ and $minB\beta$ are

respectively the maximum and minimum outputs for patterns in class B obtained for angles α and β.

Therefore if the hyperplane defined by angle α satisfies

$$minA\alpha - maxB\alpha > 0 \qquad (6.1.18)$$

then the hyperplane defined by β satisfies

$$minB\beta - maxA\beta > 0 \qquad (6.1.19)$$

As shown in figure 6.9 any of the two angles solves the problem, in the sense that they both define a hyperplane parallel to a separating hyperplane.

If both criteria, (6.1.18) and (6.1.19), are tested for each angle, angles greater than or equal to 180 degrees don't have to be tested. This is because if an angle θ greater than 180 degrees satisfies one of the criteria specified in (6.1.18) and (6.1.19) then the angle ϕ defined by

$$\phi = \theta - 180 \qquad (6.1.20)$$

will satisfy the other criterion.

This means that according to the first criterion, maximising either (6.1.18) or (6.1.19), the search for an angle can be restricted to the interval [0,180[, instead of the interval [0,360[. In this way the search interval is reduced to half of its original size.

If one considers the second criterion, maximising the number of correctly classified patterns from class B, the same reasoning applies. As mentioned before in the previous section, it is assumed that all patterns from class A are correctly classified.

Since the sign of the target is used only to indicate the class to which the pattern belongs to, then the hyperplane defined by an angle α is parallel to a separating hyperplane if one of the two conditions are verified:

$\forall i \in \mathrm{B}, out_i\alpha < minA\alpha$ or

$\forall i \in \mathrm{B}, out_i\alpha > maxA\alpha$ (6.1.21)

where $out_i\alpha$ is the output obtained for pattern i from class B at angle α, $maxA\alpha$ is the maximum of the outputs obtained for patterns in class A at angle α, and $minA\alpha$ is the minimum of the outputs obtained for patterns in class A.

Consider the following functions:

$f(\alpha) = \#\{ i \in \mathrm{B} : out_i\alpha < minA\alpha\}$ (6.1.22)

$g(\alpha) = \#\{ i \in \mathrm{B} : out_i\alpha > maxA\alpha\}$ (6.1.23)

A hyperplane is parallel to a separating hyperplane if either (6.1.24) or (6.1.25) are verified.

$f(\alpha) = \#\mathrm{B}$ (6.1.24)

$g(\alpha) = \#\mathrm{B}$ (6.1.25)

By restricting the search for the angle which defines a hyperplane parallel to a separating hyperplane the search space for the angle has been reduced from [0,360[to [0,180[. The number of tests to be made for the reduced interval is the same as the number of tests for the interval [0,360[since two tests are being made for each angle tested. However the number of times the training set is feed in the network for the interval [0,180[is half the number needed for the interval [0,360[. Therefore the computational effort when using the reduced interval is smaller than when using the interval [0,360[.

As mentioned before, throughout this section it was assumed that the targets were only used to identify the classes to which the patterns belong. However there may be situations where the targets are significant in the sense that one is really interested to have positive outputs for a particular class and negative outputs for the opposite class. In this situation, assuming that the hyperplane defined by angle α provides outputs which are the negative to the desired outputs, the hyperplane defined by angle β in equation (6.1.17) will provide the desired outputs.

$$\beta = \alpha + 180 \qquad (6.1.26)$$

6.1.1.3. Summary and Geometrical Interpretation

The original linear separation problem is initially divided into two sub problems:

- Find a hyperplane crossing the origin that is parallel to one that separates the two classes;
- Translate the hyperplane so that the patterns are separated, i.e. all patterns from one class have positive outputs and all the patterns from the other class have negative outputs.

The first subproblem is dealt with in terms of the angle the hyperplane makes with one of the axes. Using Polar coordinates a hyperplane in 2-D input space crossing the origin can be defined by a single angle. The problem can therefore be redefined in terms of finding this angle.

From a geometrical point of view the coefficients which define a hyperplane crossing the origin computed according to (6.1.9) are points in a circle with centre in the origin and radius equal to 1. To perform a search over the points on the circle the angle is used.

A further simplification of the problem arises from the reduction of the search space by half. This is accomplished due to the fact that two hyperplanes with angles which differ in 180 degrees have outputs negative to each other.

From a geometrical point of view this represents only searching on a semicircle instead of the whole circle.

The computation of k can be seen as a translation of the hyperplane so that all patterns from one class have positive outputs and all patterns from the opposite class have negative outputs.

Considering that the neural version of the original 2-D input space problem is 3-D this algorithm achieves a reduction in the dimensionality of the neural version by 2. This method has the additional benefit of having clear boundaries for the search of the angle. This result is particularly important for problems with a 2-D input space since line search procedures are in general extremely fast when compared with minimisation methods for 2 or more dimensions, which involve finding a minimum in a surface.

6.1.2. n-D linear separable problems

In this section it is shown how to generalise the concepts from the previous section to the n-D case. The 3-D case is presented in detail; generalisation to higher dimensions is discussed afterwards.

The system that represents the problem of linear separability in 3-D is:

$$\forall p \in \text{A}, w_0 + w_1 * p_1 + w_2 * p_2 + w_3 * p_3 > 0$$

$$\forall q \in \text{B}, w_0 + w_1 * q_1 + w_2 * q_2 + w_3 * q_3 < 0 \qquad (6.1.27)$$

The same system could be represented as

$$\forall p \in A,\ w_1 * p_1 + w_2 * p_2 + w_3 * p_3 > k$$

$$\forall q \in B,\ w_1 * q_1 + w_2 * q_2 + w_3 * q_3 < k \qquad (6.1.28)$$

considering $k = -w_0$.

The linear separability problem in 3-D can then be defined as:

Definition 6.2

> Find (w_1, w_2, w_3) such that there is a k that satisfies the system in (6.1.28).

The problem is now reduced to find a triple (w_1, w_2, w_3) which satisfies the definition given above. The computation of k is similar to the 2-D input space case, that is once the maximum output of one class and the minimum output for the opposite class are computed, k can be set to be midway between these two values.

The triple (w_1, w_2, w_3) defines a hyperplane which has the following equation:

$$w_1 * x + w_2 * y + w_3 * z = 0 \qquad (6.1.29)$$

As in the 2-D case, this hyperplane crosses the origin. The point (w_1, w_2, w_3) can be converted to Spherical coordinates, resulting in a point (ρ, α, β) according to the following formulas:

$$w_1 = \cos(\alpha)\cos(\beta)$$

$$w_2 = \cos(\alpha)\sin(\beta)$$

$$w_3 = \sin(\alpha) \qquad (6.1.30)$$

assuming that

$$\sqrt{w_1^2 + w_2^2 + w_3^2} = 1 \qquad (6.1.31)$$

Note that the assumption in equation (6.1.31) may be made without loss since all realisable I/O mappings lie on the surface of the hypersphere defined by eq. 6.1.31.

The search for (w_1, w_2, w_3) can therefore be replaced by a search for α and β which has clearly defined boundaries. Testing the angles in [0,360] provides all possible combinations for the hyperplane's coefficients.

As in the 2-D case the search space can be reduced to half the size, i.e. the search for α can be restricted to the interval [0,180[as shown in the next paragraph.

Suppose that α is in the interval [180,360[. The hyperplane defined by the angles $(\alpha + 180, \beta)$ will define a hyperplane which has the same orientation but produces outputs negative to the hyperplane defined by (α, β). As mentioned in the previous section these hyperplanes do not have to be considered. This implies that the search for the angle α can be restricted to the interval [0,180[which reduces the search space to half its original size.

As for the 2-D case two criteria are available to decide if a hyperplane separates both classes. The criteria are now presented in the 3-D version.

Two angles, α and β, define the hyperplane. The values of $maxA\alpha\beta$, $minA\alpha\beta$, $maxB\alpha\beta$, and $minB\alpha\beta$ are computed as follows.

$maxA\alpha\beta$ = maximum $\{ out_p\alpha\beta : p \in A \}$

$minA\alpha\beta$ = minimum $\{out_p\alpha\beta : p \in A \}$

$maxB\alpha\beta$ = maximum $\{ out_q\alpha\beta : q \in B \}$

$minB\alpha\beta$ = minimum $\{out_q\alpha\beta : q \in B \}$ (6.1.32)

where $maxX\alpha\beta$ is the maximum output obtained for patterns in class X with the hyperplane defined by (α, β), $minX\alpha\beta$ is the minimum output obtained for patterns in class X with the hyperplane defined by (α, β), and $out_i\alpha\beta$ is the output for pattern i with the hyperplane defined by (α, β).

Similarly the following functions are defined:

$$f(\alpha, \beta) = \#\ \{\ i \in B : out_i\alpha\beta < minA\alpha\beta\ \}$$

$$g(\alpha, \beta) = \#\ \{\ i \in B : out_i\alpha\beta > maxA\alpha\beta\ \} \qquad (6.1.33)$$

As in the 2-D case linear separation can be interpreted in terms of correctly classified patterns or in terms of separating outputs. According to the first criterion the following functions are used:

- $f(\alpha,\beta) = \#B$
- $g(\alpha,\beta) = \#B$ (6.1.34)

According to the second criterion the inequalities in (6.1.35) are used.

- $maxB\alpha\beta < minA\alpha\beta$
- $maxA\alpha\beta < minB\alpha\beta$ (6.1.35)

A simple heuristic to find a hyperplane which correctly partitions the training set is to select sampling weight states/angles at regular intervals until a separating hyperplane is found. If a separating hyperplane can't be found with a particular angle range then the range should be subdivided. Note however that the efficiency of this heuristic is severely penalised as the number of dimensions increases.

The neural version of the linear separation problem stated in (6.1.27) is a 4-D problem. The new interpretation of the problem in terms of spherical

coordinates transformed the neural version in a 2-D problem. A reduction in the dimensionality by two is achieved with this new interpretation.

In the general case of linear separability in n-D input space (n > 1) the same reasoning applies. The problem is initially divided into two sub problems:

- Find a hyperplane crossing the origin that is parallel to a separating hyperplane;
- Translate the hyperplane so that all patterns from one class have positive outputs and all patterns from the opposite class have negative outputs.

Since a hyperplane in n-D input space which crosses the origin can be defined by $(n-1)$ angles, using the generalisation of the formula for converting spherical coordinates to cartesian coordinates in n-D, the problem of linear separability becomes the problem of finding the values of those $(n-1)$ angles which define a separating hyperplane, i.e. there are only $n-1$ variables in the problem.

6.2. Linearly inseparable problems

If a problem is linearly inseparable then there is no complete solution using single layer nets, i.e. there is no hyperplane which separates both classes. Only a partial solution is achievable using single layer nets.

However single layer nets can be used as building blocks of a larger architecture to solve linearly inseparable problems. It is in this light that this section explores single layer nets for linearly inseparable problems.

The main purpose of this section is to propose a criterion for selecting a partial solution which is appropriate for the method described in §6.1. By appropriate is meant a criterion which will bring little or no additional computational effort to test each angle as described in §6.1.

A common criterion, which will be referred to as C1, is to select the hyperplane that minimises the total number of misclassifications. In this way one is trying to maximise the usefulness of the hyperplane in terms of classification error reduction. Mezard and Nadal (1989) use this criterion in their constructive algorithm. However, how does one know if the hyperplane is actually minimising the number of misclassification? The fact that the hyperplane is changing very little between adjacent epochs does not necessarily mean that a minimum has been found, only that a shallow gradient has been found. In order to overcome this problem variants of this criterion, used by Gallant (1986) and Romaniuk and Hall (1993), limit the time the algorithm spends dividing the training set by setting a maximum number of epochs.

The minimisation of classification error criterion, C1, when applied to the method described in §6.1 brings a considerable additional computational effort because for a given angular state the total number of misclassified patterns also depends on the value of the bias weight. This implies that for each angle tested a line search would have to be performed to find the bias value which misclassifies the least number of patterns.

Another possible criterion, hereafter referred to as C2, which avoids the above search for the bias value, is to first select the set of hyperplanes each of which in one of the sides have only patterns from one class. A further selection within this set is then made of a hyperplane that maximises the number of patterns on this side. On the other side of the hyperplane will be all the patterns from the training set that belong to the other class together with some, possibly none, of the other class. This implies that there are no misclassifications in one of the classes.

If one thinks of a single layer network as a building block of a larger network then this latter approach divides the training set in a useful way in the sense outlined below.

Using criterion C2 one knows that in one of these subsets there are only patterns from one class which means that there is no further work needed for this subset. Furthermore, as mentioned before, on this subset there is maximisation of the number of patterns.

Assuming a 2-D input space the following four functions must be evaluated:

$$f(\alpha) = \#\ \{\ i \in \mathrm{B} : out_i\alpha < minA\alpha\}$$

$$g(\alpha) = \#\ \{\ i \in \mathrm{B} : out_i\alpha > maxA\alpha\}$$

$$h(\alpha) = \#\ \{\ i \in \mathrm{A} : out_i\alpha < minB\alpha\}$$

$$l(\alpha) = \#\ \{\ i \in \mathrm{A} : out_i\alpha > maxB\alpha\} \qquad (6.2.1)$$

Note that in comparison with §6.1, specifically (6.1.22) and (6.1.23), two functions have been added, the two at the bottom. The two new functions are equivalent to the two functions which were inherited from §6.1, the two at the top, in the sense that only the classes change places.

The reason for the new functions can be better appreciated by looking at the following diagram:

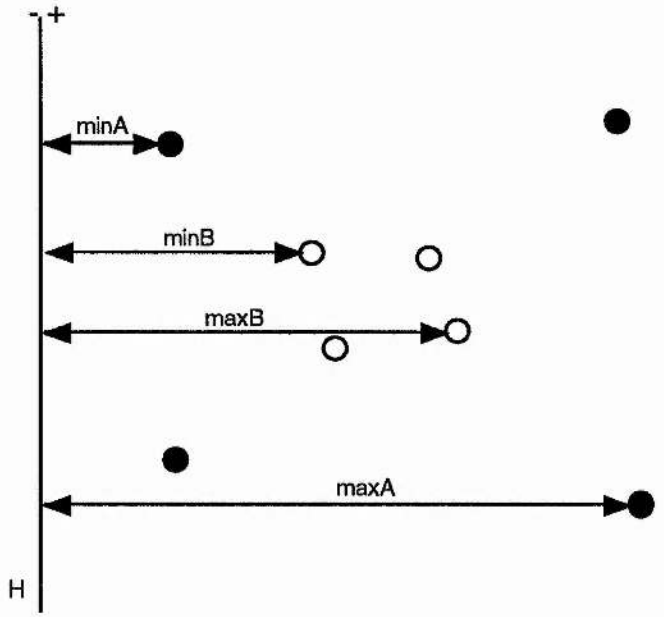


Figure 6.10. An example showing a hyperplane H and the maximum and minimum distances from the patterns of each class for a linearly inseparable problem. Black dots represent patterns for class A, white dots represent patterns for class B.

For the angle α which defines the hyperplane, the values for the four functions appearing in (6.2.1) are:

$$f(\alpha) = 0$$

$$g(\alpha) = 0$$

$$h(\alpha) = 2$$

$$l(\alpha) = 2$$

In fact, for the training set in Figure 6.10, any value of α will provide $f(\alpha)=g(\alpha)=0$. This is because the training patterns of class A completely surround the patterns from class B, i.e. there are never patterns from B which have outputs either smaller than minAα or larger than maxAα, for any α. This is the reason why one needs to use four functions instead of two as in §6.1.

The angle which satisfies the criterion is the angle which maximises function m in (6.2.2).

$$m(\alpha) = \text{maximum } \{ f(\alpha), g(\alpha), h(\alpha), l(\alpha) \} \qquad (6.2.2)$$

From a computational point of view each function only requires a comparison between two floating point values and a possible sum. This computational effort is relatively small when compared to, for instance, a floating point multiplication, which is required to feedforward a pattern.

Thus criterion C2 is applicable to the method described in §6.1 without any significant additional computational effort required and therefore this criterion is the one selected for linearly inseparable problems.

The two criteria, C1 and C2, are now compared from a graphical point of view. An example of a training set in input space is presented in the following figure. Two hyperplanes are also displayed. Hyperplane P1 misclassifies only one pattern, minimising the number of misclassifications according to criterion C1. Hyperplane P2 follows criterion C2 by having patterns of only one class on one of its sides and maximising the number of such patterns.

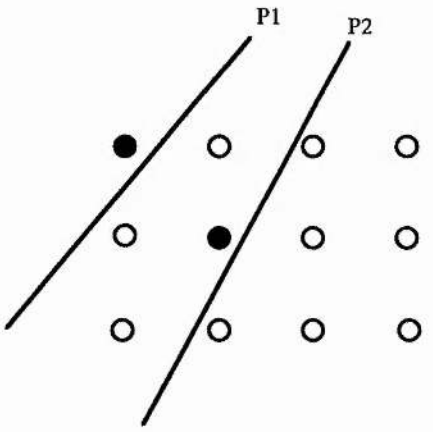


Figure 6.11. A set of patterns in input space with two possible solutions, one for each criterion C1 and C2, are represented by hyperplanes P1 and P2 respectively.

The strategy to find the solution according to the C2 criterion is similar to the one presented for finding the solution in the linearly separable problems. The only fundamental difference is in the guarantee the strategy provides. In linearly separable problems one could easily identify the solution as the hyperplane which separates the two classes. In linearly inseparable problems there is usually no way to guarantee that a hyperplane maximises a function because one typically doesn't know what the maximum is. The method used here is a relatively simple heuristic for finding a value relatively close to the maximum. See Polhill (1996) for more complex heuristics.

As described in §6.1.2, the strategy divides the training set into two non empty sets by sampling weight states/angles at regular intervals. It then selects, from the hyperplanes that have all patterns of one class on one side, a hyperplane that maximises the criterion function in (6.2.2). The function is maximised in order to reduce the number of hyperplanes for the sake of training speed.

Note that, as opposed to linearly separable problems, for linearly inseparable problems all that is required for the construction to terminate is that the hyperplane divides the training set into two non-empty sets. A successful separation of the training set into two non-empty sets can be guaranteed by subdividing the range upon finding two successive hyperplanes with all the patterns on a different side of each hyperplane.

6.3. Classification Trees

In this section a method to compose single layer networks into a larger structure is presented. The main objective is to use the theory from the previous sections in order to solve linearly inseparable problems.

Let's assume one has a linearly inseparable problem. Using a single layer network as described in §6.2 will divide the training set into two non-empty subsets: one with the patterns which have a positive output and another with the patterns which have a negative output. If in one of these subsets there are only patterns from one class then the problem is solved for the respective subset. If in a subset there are patterns from both classes then another single layer network can be used considering only the respective subset of the training set.

As mentioned in §6.2, for a linearly inseparable problem the method attempts to divide the training set into two non-empty sets. As for linearly separable problems the method always finds a solution. Therefore the partitioning will always terminate.

A high level neural structure, a Classification Tree, may be built using these principles. This is similar to a Decision Tree as will be shown.

The divide and conquer strategy adopted is similar to the one presented by Draghici (1995) for the Constraint Based Decomposition algorithm, CBD. In CBD the training set is also divided in two subsets and each subset is trained separately. The main difference between the two approaches is how the input space is divided in each stage.

The final structure which combines the hyperplanes found at each stage in CBD is also different. Draghici (1995) builds a network using a two hidden layer architecture, the first hidden layer has the hyperplanes found, a second hidden layer implements the logical AND, and the output layer implements the logical OR. Both structures are identical in behaviour in the sense that they both provide the same classifications given the same set of hyperplanes.

Note that a Classification Tree can also be converted to a Neural Network using the same scheme as Draghici used but this conversion may have disadvantages when patterns are feedforward. Unless using a parallel hardware architecture, Draghici's approach requires more computational effort than the Classification Tree approach for the feed forwarding of a pattern. In CBD all units have to be computed in order to achieve a classification, whereas in the Classification Tree this is not the case as it will be shown.

The divide and conquer strategy present in Classification and Regression Trees (CART), Breiman et. al. (1983), and the algorithm ID3, Quinlan (1983), is identical to the Classification Tree approach. The tree structure is also identical, only the non-terminal nodes differ. In both ID3 and CART the hyperplanes in each non-terminal node have to be parallel to one of the axes, see Bishop (1995). This implies that a large number of hyperplanes, compared to the minimum required may be needed. For instance, a linearly separable problem requiring a single hyperplane not parallel to any of the axes will require a larger number of hyperplanes for complete separation than the minimal number required. For implementations of ID3 see Dietterich et al (1990), for CART see Atlas et. al. (1989). In the Classification Tree approach proposed here, the hyperplanes are computed according to the algorithm in §6.2, i.e. the hyperplanes can have any orientation, not being restricted to being parallel to one of the axes.

A node of a Classification Tree can be either:

- A single layer network
- A terminal node

A single layer network is used when there are patterns in the training set from both classes being fed into the node. In this case the training set is split

into two subsets, one with the patterns which have a positive output and another with the patterns which have a negative output. These subsets will be fed into two other child nodes.

A terminal node is used when the training set has only patterns from one class. In this case the node just provides a classification.

As a matter of notation and in agreement with the notation used in binary trees in computer science, the first node of the Classification Tree shall be referred to as root node.

The XOR problem is now presented as an example. The criteria used for constructing the single layer networks is the one selected in §6.2.

Figure 6.12 shows the training set in input space and a possible hyperplane selected from the ones that obey the criterion selected in §6.2. The hyperplane divides the input space in two regions and the training set is divided accordingly.

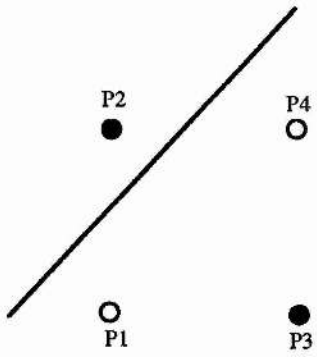


Figure 6.12. The XOR problem plus a hyperplane that maximises the function in (6.2.2).

Using the hyperplane in figure 6.12 the training set is divided into the following subsets:

- { P2 }

- { P1, P3, P4}

The first subset has only patterns from one class so a terminal node is created with the classification for P2. In the second subset there are patterns from both classes. This second subset is linearly separable so a single layer network is capable of linear separation using the method presented in §6.1. The next figure presents the second subset of the training set plus the hyperplane that solves this subproblem.

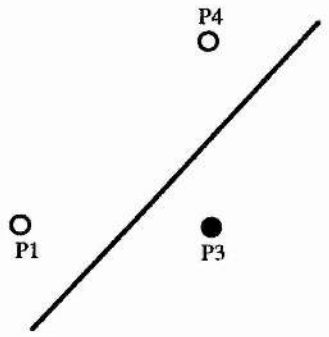


Figure 6.13. A subset of the original XOR training set in input space. A hyperplane that separates both classes is also present.

The hyperplane in figure 6.13 correctly separates the patterns from both classes present in the subset. Since in each side of the hyperplane there are only patterns from one class two terminal nodes are created each with the classification for the respective subset of patterns.

The Classification Tree for this example can be graphically displayed as shown in the next figure.

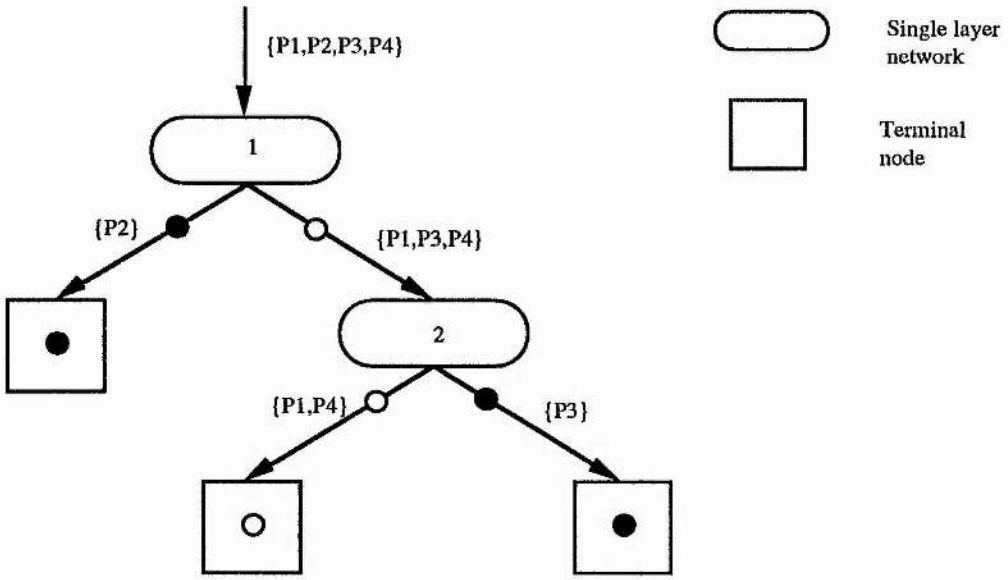


Figure 6.14. Classification Tree to solve the XOR problem

The single layer network 1 corresponds to the one presented in figure 6.12, and the single layer network 2 refers to the one depicted in figure 6.13. The training set that is fed into each node of the tree is presented in parentheses. The classification attributed is shown inside the terminal node. The circles over the links show which link is used when feedforwarding a pattern after the Classification Tree is built depending on the classification of the pattern provided by the node where the link starts from. For instance all patterns classified as white by the root node will follow the right link, whereas all patterns classified as black will follow the left link.

After the Classification Tree is built, in order to compute the classification of a pattern and starting from the root of the network the following algorithm, which is similar to the algorithm used for Decision Trees, can be used:

1. If the current node is a terminal node then provide a classification and stop, otherwise compute the excitation of the single layer network for the presented pattern;

2. Follow the link to the child which matches the classification obtained (the child node becomes the current node) and jump to 1.

In the beginning of this section it was mentioned that using Draghici's CBD one had to compute all units in order to obtain a classification for a pattern. For the Classification Tree approach, in the general case, only a subset of the single layer nodes of the tree are actually used when feed forwarding a pattern as described in the algorithm above. As mentioned before, this only represents an advantage in speed when the algorithm is running in a conventional hardware architecture as opposed to a parallel architecture.

6.4. Solving the 2 Spirals Problem with Classification Trees

The 2 spirals problem tested in this section is the same as in §5.3. In order to solve this problem Classification Trees were used where each node is found using the algorithms in §6.1 and §6.2.

As mentioned before in §6.2, for linearly inseparable problems one needs to test the whole interval to search for the best angle. In this test 36 angles were tested for each node in the tree, i.e. every angle which is multiple of 5 in [0,180[was tested.

The test took less than 0.3 seconds on a Pentium 90 machine and 34 nodes, i.e. 34 hyperplanes, were needed to build a Classification Tree which correctly classified all patterns in the training set.

Draghici (1995) presents an average time of 56.35 seconds on a Sparcstation 10/41. Romaniuk and Hall (1993) using the Divide and Conquer Network, DCN, report an average time of 772.1 seconds on a Sparcstation 2. Note that the time reported with these two methods is an average time because they are dependent on initial conditions as opposed to the Classification Tree method.

Since the results were obtained in different machines comparison is not straightforward. However since all these algorithms spent most of their time doing floating point operations it is possible to get an estimate of the time which would take if they were all done on a Pentium 90 using the floating point specifications (SpecFP92) of each machine. The SpecFP92 for the Pentium 90 tested is 68.3, for a Sparcstation 10/41 is 67.8, and for a Sparcstation 2 is 17.0.

This means that if an algorithm takes 1 second in a Pentium 90, it will take approximately 1.007 seconds on a Sparcstation 10/41 and 4.047 seconds on a Sparcstation 2. Applying this correction factor, the results as if all tests were done on a Pentium 90 are as follows:

- Draghici's CBD: 55.95 seconds;
- Romaniuk and Hall's DCN: 190.78 seconds.

If a direct comparison were to be made based on these estimates for the 2 spirals problem then the Classification Tree would be 186.5 times faster than CBD and 635 times faster than DCN.

It must be stated that the use of a correction factor based on SpecFP92 does not provide an exact comparison between the methods. There are other factors involved which can affect performance, for instance the way the source code for the simulator was written, which compiler was used, etc.

These comparisons should only be taken as an indication of the relative speed of the methods for solving the two spirals problem and not as exact figures. Furthermore there is no claim that these comparative results would hold if the input space had a higher number of dimensions. As mentioned in §6.1 the method used for computing the non terminal nodes of the Classification Tree is more efficient when the dimension of input space is low.

As for the number of hyperplanes used, Draghici's CBD uses in average 53.65 hyperplanes, and Romaniuk and Hall's DCN uses in average 34.6 units. The Classification Tree uses 34 hyperplanes which is roughly the same number as Romaniuk and Hall's average for DCN, but a significantly lower number than Draghici's CBD.

6.5. Conclusion

In this chapter a classifier was presented. The method proposed reduces the dimension of the neural version of the problem by 2, i.e. for instance a 2-D input space problem the method reduces the original linear separability problem, which is a set of inequalities in 3-D, to a 1-D problem.

This result is particularly important for problems with a 2-D input space since line search procedures are in general extremely fast when compared with minimisation methods for 2 or more dimensions, which involve finding a minima in a surface. As the dimension of input space grows the potential benefits of the method will tend to disappear.

The main purpose of the method is to divide the training set into two subsets. If the problem is linearly separable then the method can find a separating hyperplane.

For linearly inseparable problems, §6.2, the method divides each existing training set using a single hyperplane in order to find a best solution. A criterion to define possible best solutions was presented.

If a problem is linearly inseparable then the training set gets divided in two subsets. For each subset either there are only patterns from one class or there are patterns from both classes. In the first case the problem is solved for the subset. For the second case the method is applied again for the subset until in

each of the subsets there are only patterns from one class. A Classification Tree, §6.3, is built using the hyperplanes found at each stage.

The results reported in §6.6 show that this method is extremely fast to solve the two spirals problem requiring less than 0.3 seconds on a Pentium 90. Using a correction criterion to accommodate for the hardware differences this method should perform two orders of magnitude faster than Draghici's CBD and Romaniuk and Hall's DCN given identical hardware configurations.

A note must be made regarding these comparisons. As mention in §6.1 the benefits of this method are greater when the dimension of the input space is small which is the case for the 2 spirals problem. There is no claim that these comparisons would hold if the input space had a higher number of dimensions.

7. A Mechanism for Generalisation

In this chapter a mechanism is proposed to improve the average generalisation ability for hyperplane based classifiers. The mechanism attempts to do this by establishing a trained boundary so that there is a maximal overall spacing from the boundary between training points closest to this boundary.

Any solution, i.e. any hyperplane which correctly separates the training set, for any particular training set provides a potentially valid generalisation. The mechanism presented in §7.1 attempts to improve generalisation on average. Relation to other work is presented in §7.2.

An implementation of the mechanism provided for the Classification Trees algorithm from chapter 6 is presented in §7.3. However note that the mechanism is algorithm independent and can be applied in principle to any classifier which is hyperplane based. Results for this implementation are presented in §7.4.

Conclusions are drawn in §7.5.

7.1. The Mechanism

For a given training set it is likely that for each class in the training set the patterns aren't actually at the boundaries of the regions defined by the underlying problem for each class.

Therefore it is likely that for each class there will be patterns from the underlying problem between the closest training points of the same class to the boundary defined by the underlying problem, which is referred to as the underlying boundary from now on, and the boundary itself.

An equivalence relation can be defined on the set of possible partitions of a training set. Two partitions would be in the same equivalence class if both have the same training patterns in each region. The argument which follows applies to each single equivalence class of partitions.

It is assumed that the patterns from the underlying problem between the closest training points of the same class to the underlying boundary and the underlying boundary itself are the ones more likely to get misclassified by any training boundary. This is a reasonable assumption since it is a subset of these patterns that lie between any trained boundary and the actual underlying boundary. Figure 7.1 illustrates this.

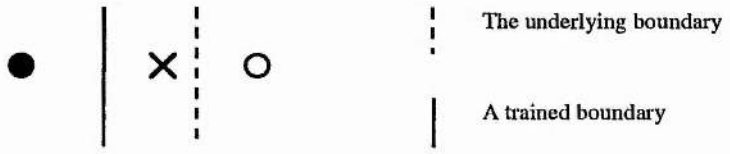


Figure 7.1. The cross indicates a pattern from the underlying problem which does not belong to the training set and is misclassified by the trained boundary.

An argument now follows for improved generalisation when there is the maximal overall spacing from the trained boundary between training points closest to this boundary. The argument directly involves only a relative small subset of the possible test patterns, i.e. those on lines between pairs of training points of opposite classes closest to the hypothesised boundary.

For the sake of simplicity, linearly separable problems are presented to illustrate the concepts which are about to be introduced. Consider the simple underlying problem of a binary partition. The figure 7.2 shows possible trained boundaries which separate the two patterns. The trained boundary b) in figure 7.2 is the maximal spacing boundary.

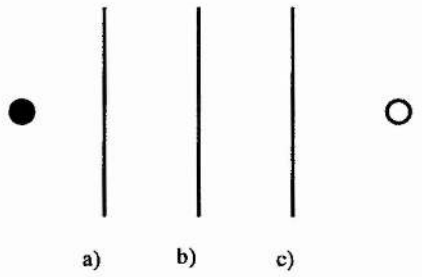


Figure 7.2. Three trained boundaries which separate the training set.

Any of the trained boundaries presented has an equal probability of being the underlying boundary assuming of course that there is no a priori knowledge of the underlying problem.

Consider now four test patterns each nearer to the underlying boundary than the patterns in figure 7.2, see figure 7.3. Note that the four test patterns have no classification associated with them because one doesn't know where the underlying boundary is.

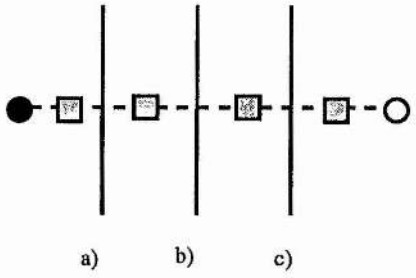


Figure 7.3. Four test patterns were added to figure 7.2.

Assume now that the underlying boundary coincides with the trained boundary a) from figure 7.3. Then the test patterns will be classified as in figure 7.4.

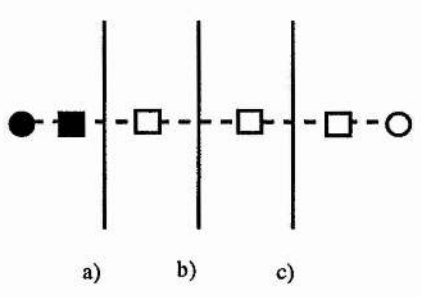


Figure 7.4. The trained boundary a) coincides with the underlying boundary.

Under this assumption the trained boundary a) would correctly classify all four test patterns, the trained boundary b) would correctly classify three out of the four test patterns, and finally the trained boundary c) would correctly classify two out of the four test patterns.

As mentioned before any of the trained boundaries has an equal probability of being the actual underlying boundary. Applying the same reasoning to the other trained boundaries the following results are obtained:

UB	TB	Correct
a	a	4
	b	3
	c	2
b	a	3
	b	4
	c	3
c	a	2
	b	3
	c	4

Table 7.1. Number of correct classifications provided by the three trained boundaries assuming different underlying boundaries.

In table 7.1 the first column shows the position of the underlying boundary, the second column shows the trained boundary, and the third column shows the number of test patterns correctly classified using the trained boundary in the second column and assuming an underlying boundary as specified in the first column.

In the worst case situation both trained boundaries a) and c) only correctly classify two out of the four patterns whereas the trained boundary b) correctly classifies three out of four patterns. On average 3.33 patterns are correctly classified by using the maximal spacing boundary whereas only 3 patterns are correctly classified by the other two trained boundaries.

In the general case, considering n trained boundaries and $n+1$ test patterns the average misclassification rate for the maximal spacing boundary is defined as follows considering n to be an odd number

$$maxAv(n) = \frac{2}{n} * \sum_{x=1}^{p} x \qquad (7.1.1)$$

where p is defined as

$$p = \frac{n-1}{2} \qquad (7.1.2)$$

The average misclassification rate for any other trained boundary i is defined as follows

$$rate_i(n) = \frac{2 * \sum_{x=1}^{i-1} x + \sum_{x=i}^{n-i} x}{n} \qquad (7.1.3)$$

where the index i is an integer in the interval $[1,p]$, see figure 7.5.

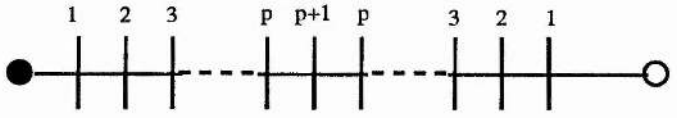


Figure 7.5. A set of n training boundaries. The numbers above the boundaries represent the boundaries indexes which appear in equation 7.1.3. The trained boundary p+1 is the maximal spacing boundary.

In order to prove that the maximal spacing boundary improves the average generalisation ability it is necessary to show that

$$maxAv(n) < rate_i(n) \tag{7.1.4}$$

where i is in the interval $[1,p]$.

Considering the expressions in the right side of (7.1.1) and (7.1.3), (7.1.4) becomes

$$\frac{2}{n} * \sum_{x=1}^{p} x < \frac{2 * \sum_{x=1}^{i-1} x + \sum_{x=i}^{n-i} x}{n} \tag{7.1.5}$$

since $p > i$ it follows from (7.1.5) that

$$\frac{2}{n} * \sum_{x=i}^{p} x < \frac{\sum_{x=i}^{n-i} x}{n} \tag{7.1.6}$$

and hence

$$\sum_{x=i}^{p} x < \sum_{x=p+1}^{n-i} x \tag{7.1.7}$$

Both sides of the inequality (7.1.7) are sums of a series of consecutive numbers with the same numbers of elements because

$$p - i = n - i - p - 1 \qquad (7.1.8)$$

due to the relation between p and n stated in equation (7.1.2). Since the series on the left side of the inequality (7.1.7) starts with a lower value the inequality (7.1.7) is verified. This proves that the maximal spacing boundary produces a lower average number of misclassifications than any other boundary.

Hence the probability of correctly classifying patterns which are between the training points and the underlying boundary will be increased by maximising the overall spacing between the patterns closest to the trained boundary and the trained boundary itself.

As mentioned before the argument directly involves only a subset of the possible test patterns, i.e. those on lines between pairs of training points of opposite classes closest to the hypothesised boundary. As the number of training points either side of each boundary tends to infinity, more and more test points lie between a pair of training points. So, provided the actual training points are representative of what happens in the infinite limit, the average generalisation ability should be improved over all possible test points.

An analysis is now presented on how the maximal spacing approach can influence the average generalisation ability for different types of underlying boundaries.

Enforcing a maximal spacing partition can cause the use of a larger number of hyperplanes than the minimal number needed to separate the patterns in the training set. The figure below shows an example.

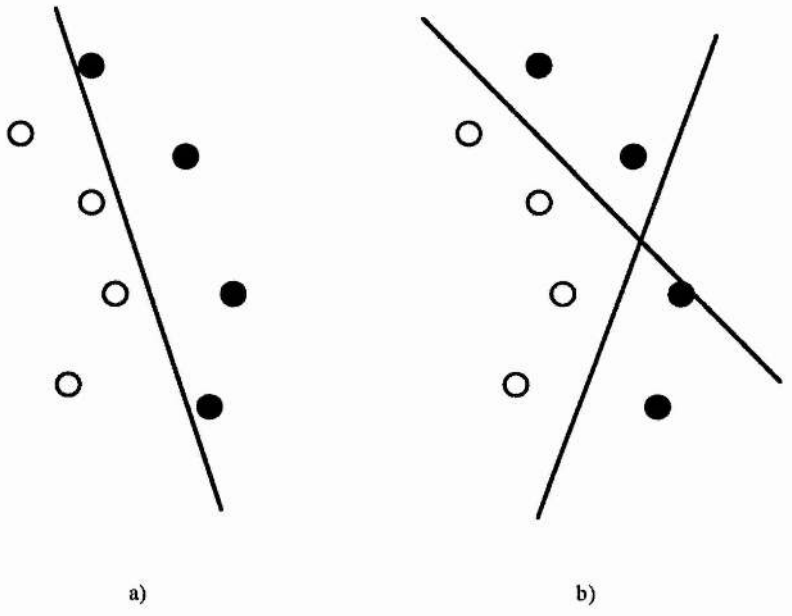


Figure 7.6. Two partitions for the same training set. a) A single hyperplane partition. b) The maximal spacing partition.

For underlying problems requiring a large number of hyperplanes to produce a good approximation of the underlying boundary, using a larger number of hyperplanes than the minimal number actually needed to separate the training set may be advantageous. Some curved boundaries are an example of such an underlying boundary provided that there are enough training points to be representative.

For instance for the training set for the 2 spirals problem used by Baum and Lang (1990) it is clear that the trained boundary with maximal spacing should improve the generalisation results. This is because the underlying boundary is regular and evenly spaced either side of the training points.

However for underlying boundaries requiring a low number of hyperplanes relative to other ones consistent with the training set the maximal spacing boundary will provide poor generalisation results.

For example, because each hyperplane forces maximal spacing locally, one may end up with a zig-zag I/O partition as the figure below illustrates.

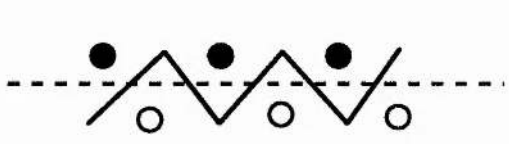


Figure 7.7. Two possible partitions for the same training set.

The dotted line represents a possible linear underlying boundary. The plain line represents the trained boundaries constructed with the maximal spacing approach.

If the underlying boundary is in fact the linear one, the zig-zag boundary created with the maximal spacing approach will lead to poor generalisation results.

The approach taken here to this problem is to explore various spacings besides the maximal spacing. Suppose the hyperplanes are required to be at a minimal distance g from any pair of training points of opposite classes.

Setting g at 0.0, i.e. no minimal spacing is required, one gets the same partition as with the standard approach. With g set to the maximal value for the training set one obtains the partition for the maximal spacing approach.

Low values of g relative to the maximum value of g tend to use a lower number of hyperplanes in average. This is because with lower values of g more training patterns are correctly classified by each hyperplane relative to the maximal value of g. The following figure presents three partitions obtained with different values of g.

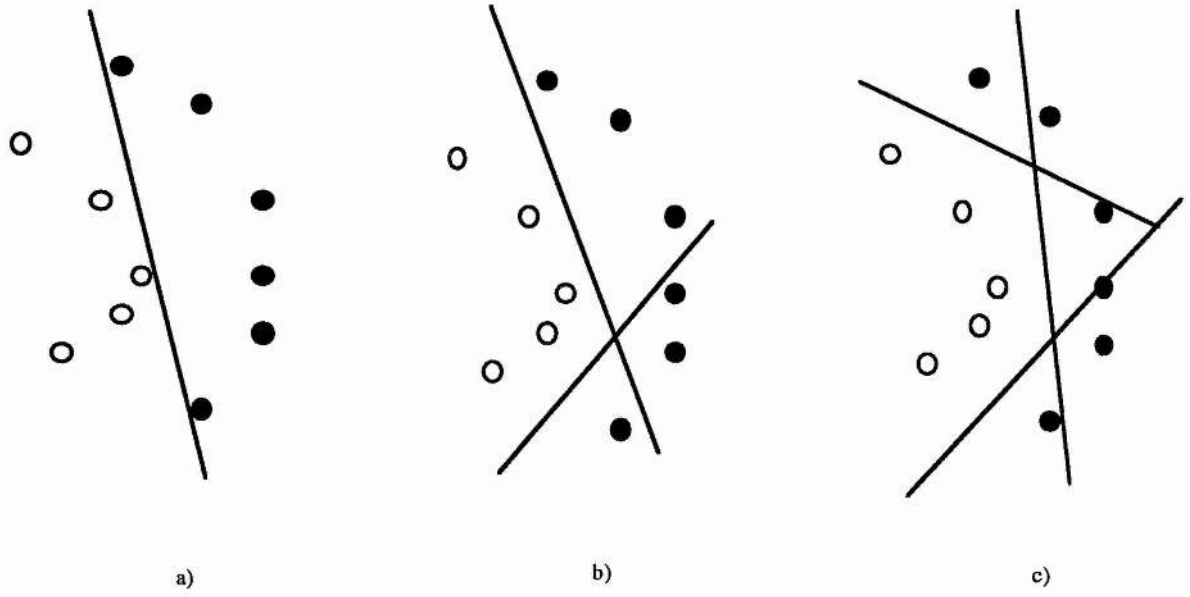


Figure 7.8. Three partitions created with different values of g. Partition a) has g = 0; partition c) has g set to the maximal value; partition b) uses an intermediate value for g.

Varying g between 0.0 and the maximal value of g enables us to explore multiple partitions including as a subset those accessible through standard methods. Within each partition explored the value of g is maximised to obtain the best average generalisation ability for that partition.

Not all possible partitions are explored since more than one partition may be consistent with any particular value of g. Nevertheless the variety of partitions given by this method provides improved generalisation relative to methods using a single partition not known to be the best one.

Testing the partitions obtained by varying g requires a test set. Using the test set for each partition, the number of misclassified patterns can be computed and the partition with the lowest misclassification rate is then selected.

7.2. Relation to Other Work

For a general review of generalisation see Denker et. al. (1987) or Draghici (1995). In here the most closely related work will be discussed.

Fahlman's empirical study (1988) is perhaps the first to attempt a related approach.

From a pure classification point of view a pattern is either correctly classified or misclassified. Therefore as soon as all patterns are correctly classified training could in theory be halted.

Fahlman proposes that training should continue until a no man's land is achieved. For instance if the outputs are in the range [0,1] Fahlman proposes that training should stop only when all patterns are correctly classified and there are no patterns with outputs in the interval [0.4,0.6].

This can be viewed as an attempt to guarantee a minimal spacing between the patterns from opposite classes closest to the underlying boundary.

The difference in Fahlman's approach and the approach presented in here is that Fahlman's minimal spacing boundary is achieved in output space whereas the approach presented in §7.1 is based in input space.

The disadvantage of the output space approach is that a no man's land in output space can be artificially created by scaling up the weights. By scaling up the weights of any trained boundary which separates the two classes a no man's land of any size within the limits of the possible output values can be artificially created.

This implies that in practice Fahlman's approach does not guarantee a minimal distance in input space, it just guarantees a minimal distance in output space.

A possible solution to overcome this difference would be to have the weights normalised. The distance in output space is then equal to a set distance in input space. However in this case minimising the output error using least squares, which is the context of this study, may in the extreme case not lead to a solution where all the patterns are correctly classified.

This is because with normalised weights a fixed gap of 0.2 in output between patterns of opposite classes may not be realisable in conjunction with completely correct classification.

7.3. An Implementation using Classification Trees

Consider the following training set and two possible partitions in figure 7.9.

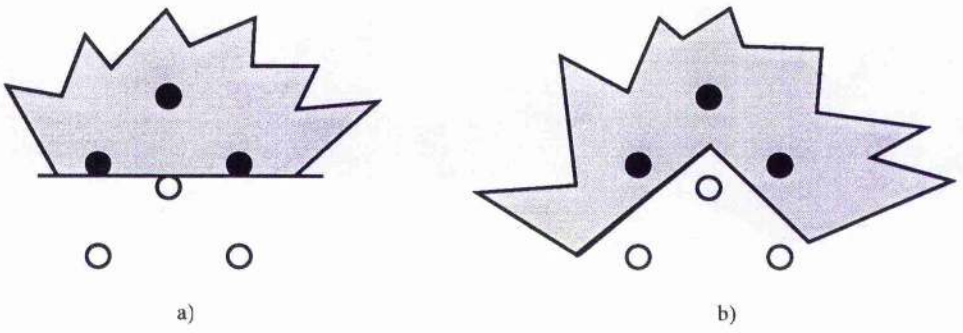


Figure 7.9. a) The region defined by the hyperplane without the generalisation framework. b) The region found using the generalisation framework.

In both cases the regions defined by the I/O mapping offer valid generalisation in the sense that both regions realise the training set. However, as mentioned in §7.1, the hyperplanes in figure 7.9.b) provide better generalisation on average.

The main reason for the difference in the average generalisation ability of the classifiers in fig. 7.9 from a graphical point of view is that in a) the hyperplane is set too close to the patterns in the training set, whereas in b)

the hyperplanes are set at a larger distance from the patterns in the training set.

The partitions in figure 7.9 correctly classify all patterns because all patterns from B obey the following condition:

$$\forall\, i \in \mathrm{B}: out_i\alpha < minA\alpha \qquad (7.3.1)$$

In order to guarantee a minimal distance it makes sense to say that a pattern from B will only be correctly classified if

$$out_i\alpha + 2*g < minA\alpha \qquad (7.3.2)$$

where g is a positive constant.

Inequality (7.3.2) says that, assuming that all patterns from A are correctly classified, in order for a pattern from B to be considered correctly classified it needs not only to have an output below the minimal output for A but also that the difference in outputs must be greater than 2 * g. The hyperplane can then be positioned so that it is at least at a distance of g from any of the patterns which it correctly classifies. This means that instead of maximising the distance in outputs between the closest patterns from opposite classes to the hyperplane one is actually defining a threshold as being the minimum acceptable distance.

Note that on the method described in chapter 6 the weights are normalised, i.e. the distance in output space is equal to a set distance in input space. Also the method in chapter 6 does not minimise the sum of squared errors, so the problems mentioned in the previous section when normalising the weights do not apply.

Relating back to figure 7.9, for a large enough g there is no hyperplane which correctly classifies all patterns. In fact there are values for g for

which the respective hyperplane misclassifies all patterns according to (7.3.2). The constant g is training set dependent, i.e. it is possible that g is set to a value which is too large for the training set, meaning that for every angle there are no correctly classified patterns.

The approach taken to implement the maximal spacing is to select a large value of g to start with. If there are no patterns from B correctly classified according to (7.3.2) for that value of g then the value of g is decreased. The value of g is repeatedly decreased until there are patterns which are correctly classified according to (7.3.2)

Assuming a good g for the training set in figure 7.9, i.e. a large enough value so that the hyperplane is not positioned to close to any of the patterns, and small enough to avoid misclassifying all patterns from B for every angle tested, then for the orientation of the hyperplane in figure 7.9.a) there would only be one correctly classified pattern from B.

The following figure shows a possible hyperplane which correctly classifies 2 patterns from B using the definition in (7.3.2).

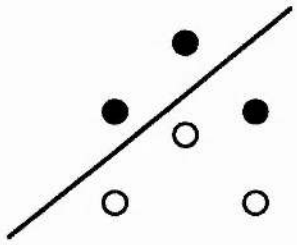


Figure 7.10. A hyperplane found using equation (7.3.2) to compute the number of correctly classified patterns.

The training set is divided into two subsets using the hyperplane in figure 7.10. In one of the subsets there are only patterns from one class. In the other subset there are patterns from both classes so a further hyperplane is needed.

Figure 7.11 shows a possible hyperplane which separates both classes using inequality (7.3.2).

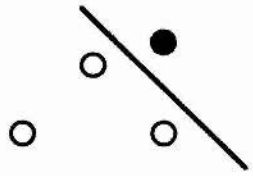


Figure 7.11. A hyperplane which separates the remaining patterns.

As seen in this example, using the proposed framework for generalisation it is possible that more hyperplanes are needed to successfully solve the problem than the minimal number.

7.4. Experiments

In order to test the generalisation mechanism the Classification Tree was built using a training set. Afterwards the Classification Tree obtained is tested on a test set with different patterns.

Hinton (1989) criticises the use of test sets in the sense that they only provide an informal demonstration that the classifier is capable of nontrivial generalisation. Hinton says that test sets alone do not provide any insight as to why the classifier is able to generalise. While this is true, in this particular case the theoretical foundations have also been laid, §7.1, and the test set is used to backup the theory and not the other way around.

7.4.1. The 2 Spirals Problem

The training set used here is the same as in §5.3. The test set selected for this problem is the same as in Baum and Lang (1990) and Lang and Witbrock (1988) to allow comparisons to be made.

The test set is constructed by adding three points between each pair of adjacent training points of the same class. The test set used has 576 patterns being three times as large as the training set.

The results using the implementation for the generalisation framework in §7.1 are presented for several initial values of g in table 7.2. Although in practice one could start with a large value of g and let the algorithm reduce it if needed, the approach taken in this section shows how varying the spacing, up to the largest spacing could influence generalisation.

The four columns present the following information:

- g - The initial value of g used in each experiment;
- Errors - The number of misclassified patterns from the test set;
- Nodes - the number of non-terminal nodes in the Classification Tree, i.e. the number of hyperplanes needed to solve the problem;
- Seconds - the number of seconds needed to build the Classification Tree using only the training set on a Pentium 90.

g	Errors	Nodes	Seconds
0	85	34	0.267
0.05	42	29	0.248
0.10	26	33	0.270
0.15	21	44	0.373
0.20	4	35	0.309
0.25	2	46	0.399
0.30	2	49	0.426
0.35	2	52	0.415
0.40	0	61	0.511
0.45	0	80	0.691

Table 7.2. Results for several values of g.

Values for g equal to or greater than 0.5 are too large for the 2 spirals training set. This is because of the spacing between the two spirals. When using initial values of g greater or equal to 0.5 the value of g was reduced by the mechanism as described in §7.3 until a value smaller than 0.5 was achieved and the training set was divided into non-empty sets.

As expected, the results show that as g is increased the errors in the test set decrease. For g equal to 0.40 or 0.45 there are 0 errors in the test set.

The results also show that using g, i.e. setting g to a value different from 0.0 does not monotonically increase the number of nodes. In the Classification Tree, setting g at 0.05 or 0.10 actually uses less hyperplanes than when g is set to 0.0.

Comparing with results from Baum and Lang (1990) and Lang and Witbrock (1988) this method shows a superior generalisation capability. The best result for generalisation obtained by Baum and Lang (1990) is 15 errors in the test

set. The best result for generalisation in Lang and Witbrock's paper (1988) is 56 errors in the test set.

In this particular problem very good results were obtained. This is because both the training and the test set patterns are evenly spaced from the underlying boundaries.

7.4.2. Randomly Generated Training Sets

The following problems have randomly generated training sets. To construct each training set n points in the square (0,0) (1,1) were randomly picked. A pattern would have as input the point and as target the result of the membership function presented for each problem. Three different values of n were used to build classification trees: 50, 200, 1000. For each value of n, 26 different training sets were constructed.

In order to evaluate the generalisation ability, a test set with 10000 patterns was used. The test set patterns are positioned on a grid where the distance between two adjacent points is 0.01.

For each problem the following results are presented for each value of n and for each value of g:

- Average number of nodes;
- Average number of test set errors;
- Ratio of the number of misclassified patterns for the current g value and the number for $g = 0$.

7.4.2.1. The Quarter Circle Problem

The class membership function for this problem is

$$x^2 + y^2 > 0.75 \qquad (7.4.1)$$

where x and y are input components of the pattern.

If a pattern satisfies inequality (7.4.1) it belongs to one class, if it doesn't it belongs to the opposite class. The following figure presents the I/O map of the underlying function.

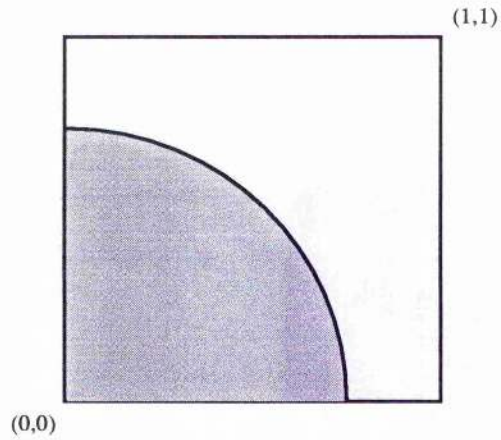


Figure 7.12. The I/O map for the Quarter Circle Problem

For n = 50 the following results were obtained:

g values	Nodes	Errors	Ratio
0.000	2.69	816.38	
0.005	3.11	810.65	0.99
0.012	3.42	770.50	0.94
0.025	3.84	760.00	0.93
0.050	7.80	628.76	0.77
0.100	16.75	575.11	0.70
0.250	29.61	573.84	0.70

Table 7.3. Results for the Quarter Circle Problem with n = 50.

As expected, the ratio is decreased as g is increased. The number of hyperplanes also increases with g. Values of g greater than 0.25 are too large for the training set. The largest spacing, g = 0.25 provided better generalisation in 22 out of 26 cases relative to g = 0.

For n = 200 the following results were obtained:

g values	Nodes	Errors	Ratio
0.000	4.50	354.30	
0.005	5.07	295.07	0.83
0.007	5.09	278.46	0.78
0.012	8.46	266.11	0.75
0.025	16.23	241.50	0.68
0.050	34.15	231.26	0.65

Table 7.4. Results for the Quarter Circle Problem with n = 200.

Values of g greater than 0.05 are too large for the training set. The maximal spacing approach gave better generalisation results compared to g = 0 in 24 out of 26 cases.

For n = 1000 the following results were obtained:

g values	Nodes	Errors	Ratio
0.000	8.92	129.92	
0.0005	9.15	117.30	0.90
0.0025	11.07	100.11	0.77
0.0050	17.07	88.65	0.68
0.0075	26.46	86.88	0.66
0.0100	36.19	80.15	0.61

Table 7.5. Results for the Quarter Circle Problem with n = 1000.

Maximal spacing when compared with g = 0 provided better generalisation results in all cases. g = 0.01 is the maximal g for these training sets.

In all cases the ratio of misclassified patterns decreases as g increases. The number of hyperplanes increases with g. The ratio for the maximal spacing approach is between 0.61 and 0.70 for the different values of n.

7.4.2.2. The Line Problem

The class membership function for this problem is

$$x + y < 1.0 \qquad (7.4.2)$$

where x and y are the input components of the pattern.

If a pattern satisfies inequality (7.4.2) it belongs to one class, if it doesn't it belongs to the opposite class. The following figure presents the I/O map of the underlying function.

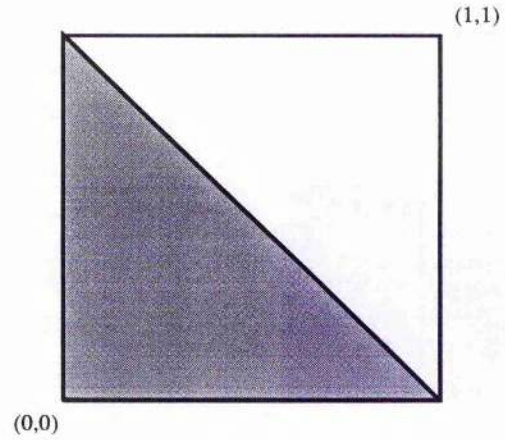


Figure 7.13. The I/O map for the Line Problem

The next three tables present the results obtained.

g values	Nodes	Errors	Ratio
0.000	1	398.15	
0.001	1	382.80	0.96
0.005	1	360.00	0.90
0.010	1.03	331.23	0.83
0.025	1.84	372.50	0.93
0.050	3.30	337.23	0.84
0.100	7.11	453.26	1.13
0.5	29.11	520.23	1.30

Table 7.6. Results for the Line Problem with $n = 50$.

g values	Nodes	Errors	Ratio
0.000	1	80.50	
0.001	1	77.07	0.95
0.005	1.34	73.46	0.91
0.010	2.46	80.42	0.99
0.025	6.84	134.61	1.67
0.050	15.57	207.26	2.57
0.100	31.53	227.11	2.82

Table 7.7. Results for the Line Problem with $n = 200$.

g values	Nodes	Errors	Ratio
0.000	1	49.50	
0.001	1	49.5	1.00
0.005	5.30	43.73	0.88
0.010	13.42	48.26	0.97
0.025	33.80	75.50	1.52
0.050	57.30	77.92	1.57
0.100	77.19	78.46	1.58

Table 7.8. Results for the Line Problem with $n = 1000$.

This problem has a linear underlying boundary, i.e. it requires only one hyperplane for a good approximation between the underlying boundary and the trained boundary. As mentioned in §7,1 bad generalisation results are expected in this case using the maximal spacing approach. This is because the maximal spacing approach tends to use a larger number of hyperplanes, relative to those needed to actually separate the training set, producing in this case a zig-zag trained boundary. Such a boundary is not representative of what occurs in the infinite limit which is a straight line.

Nevertheless, for all values of n, there was a partition which provided an improvement in average generalisation relative to the partition obtained with $g = 0.0$, with the best ratios for the different values of n varying between 0.83 and 0.91.

7.5. Conclusions

A mechanism was presented to improve the average generalisation ability. The method attempts to establish a trained boundary so that there is a maximal spacing from the boundary between training points closest to this boundary for a given partition.

This mechanism tends to produce solutions with a large number of hyperplanes relative to the minimum needed to partition the training set. This may or may not be advantageous depending on the underlying boundaries.

The results confirm the theory as to when improvement may be expected. The results of the 2 spirals problem with zero misclassification error on the test set came as no surprise since both the training and test set patterns are equally and evenly spaced from the underlying boundary. In particular, the training set in conjunction with the mechanism is representative of what happens in the infinite limit.

The quarter circle also shows an improvement in the average generalisation ability using the maximal spacing approach. This is to be expected since again, the training patterns in conjunction with the mechanism are representative of what happens in the infinite limit. The improvement is not as large as in the two spirals problem because the training patterns in this case are not evenly spaced from the underlying boundary.

For the line problem the results confirm the theory about when the mechanism will degrade generalisation performance. In this case, the training patterns in conjunction with the mechanism produce a zig-zag boundary which is not representative of the straight line occurring in the infinite limit.

The mechanism can be used to generate a set of partitions which include as a subset those generated by the standard approach. In all tests the mechanism provided partitions with better average generalisation ability than the partitions obtained with the standard approach.

8. Conclusion

This thesis addresses two issues in Neural Nets research: robustness in convergence, and generalisation. Robustness is tackled for Multilayer Feedforward Networks. A framework for generalisation is presented for Classifiers. From a theoretical point of view the generalisation framework presented can be applied to Multilayer Feedforward Networks although it has not been done here.

The issue of robust training is addressed for fixed architectures. Several researchers have proved the theoretical capabilities of Multilayer Feedforward networks, (§2.2.3), but in practice the robustness of standard methods like standard backpropagation, conjugate gradient descent and Quasi-Newton methods is poor for various problems, (§2.6).

Except for standard backpropagation, all the methods mentioned above assume to some degree that the error/weight surface is quadratic, (§2.4) and (§2.5). It was suggested that the common assumptions about the overall surface shape break down when many individual component surfaces are combined and robustness suffers accordingly.

In chapter 3 a new method to train Multilayer Feedforward networks was presented in which no particular shape is assumed for the overall surface and where an attempt is made to optimally combine the individual components of a solution for the overall solution. The method is based on computing Tangent Hyperplanes to the non-linear solution manifolds. The method presented is basically a mechanism to minimise the sum of squared errors and therefore its use is not limited to Neural Networks.

It was shown that the prediction of the goal's whereabouts using linear approximations to non-linear solution manifolds is good only if the current

weight state is close enough to the goal weight state, (§3.2). This notion of closeness is deeply related to the curvature of the solution manifolds. The more curved the solution manifolds, the closer the current weight state has to be to the goal weight state in order to get a good linear approximation.

In the general case, the prediction of the goal's whereabouts using linear approximations tends to get worse as the distance between the current weight state and the goal weight state increases.

The two components of a prediction are the direction and the step size. The direction of the prediction obtained in the general case with linear approximations has been empirically found to be more stable than step size when distance increases.

Based on this empirical evidence, line search, being a classical technique to find the optimal step size, was one additional option used to construct a version of the algorithm, (§3.4). Two error measures were proposed for the line search: output error and Euclidean distance in weight space to the solution manifolds. The results presented show that when using large output error tolerances the line search using an output error measure performs marginally better.

Another approach used to deal with the closeness problem was based on preventing deterioration from occurring in the first place, (§3.5). This was achieved with the notion of a subgoal. In this thesis, two methods to construct subgoals were considered. The first was set in output space and the second in weight space. The second approach was selected because it provides a better direction and is much cheaper.

The method was shown to be very robust regarding convergence of training when compared to other common methods for these types of nets, see

chapater 5. These results confirm the theoretical suppositions that the method is able to perform good weight space transitions.

In the second part, a new divide and conquer type of algorithm using hyperplanes was proposed based on Classification Trees, see chapter 6. The algorithm used to divide the training set reduces the dimensionality of weight space by 2. The benefits of this technique are greater when the dimension of weight space is low. For instance considering a 2-D input space, a 3-D weight space, the reduction implies that the problem is solved in a 1-D space. Note however that as the dimension of weight space increases the potential benefits of this technique grow weaker.

The algorithm solves linearly separable problems with a single hyperplane, (§6.1). For linearly inseparable training sets, the algorithm divides the training set into two non-empty sets, (§6.2). The algorithm is applied recursively to each subset which contains patterns from more than one class. The partitioning of the training set stops when the partition correctly separates the patterns in the training set.

A Classification Tree is built with the hyperplanes found at each stage, (§6.3). If desired, the Classification Tree obtained can be transformed without further training into a Neural Network with two hidden layers.

The results show that training is extremely fast when dealing with 2-D input problems. However, as mentioned before, the gains reported cannot be generalised to the n-D input space case, $n > 2$. The experiments in (§6.4) and (§7.4) also show that the algorithm uses a relatively low number of hyperplanes.

In chapter 7 a mechanism to improve the average generalisation ability was proposed. The mechanism attempts to establish a trained boundary so that

there is an optimal overall spacing from the trained boundary to training points closest to this boundary.

The results show that for underlying boundaries which require a large number of hyperplanes for the trained boundary to achieve a good approximation to the underlying boundary a maximal spacing approach is capable of improving the average generalisation ability relative to the standard approach.

For underlying boundaries which require a low number of hyperplanes relative to others consistent with the training set, guaranteeing a minimal spacing strategy is a better option.

In all tests performed, either the minimal or the maximal spacing approach managed to improve the average generalisation ability when compared to the standard approach. An analysis was given as to the overall merit of the implemented approaches.

8.1. Future Work

8.1.1. Robust Training

The algorithm presented has three parameters which as shown by the results can affect performance. While good convergence rates were obtained in general, the speed of convergence does have significant variations depending on the parameter settings. A mechanism designed to automatically fine tune the parameters during training is desirable.

In the experiments it was shown that for the 5 bit parity problem the results achieved were better if the initial random weight states selected were in an interval closer to the origin. This suggests that the Tangent Hyperplanes algorithm may work better when starting from an initial weight state close to the origin. Further tests are needed to verify if this is the case.

8.1.2. The Classifier

Meggido's algorithm not only reduces dimensionality by a factor of one, he also devised a mechanism to drop constraints in every epoch. Such a mechanism significantly reduces the computational effort of Meggido's algorithm. Further work is needed to investigate if dropping constraints is a possibility using the classifier presented.

Also studying more powerful heuristics, or developing an algorithm, to find the best orientation for a hyperplane according to the criteria presented is a subject for further work when the number of input components is greater than 2.

8.1.3. Generalisation Framework

From a theoretical point of view, the application of the framework proposed in §7.1 is not restricted to the method described in chapter 6.

It would be interesting to implement the generalisation framework using other hyperplane based classifiers, such as Tangent Hyperplanes applied to Multilayer Feedforward Networks, and see how the generalisation results would compare with and without the implementation. Further work is needed to define an implementation for this case.

Appendix A. Thesis Software Manual

The thesis software manual is divided in three parts. The first part, §A.1, deals with fixed Multilayer feedforward architectures. Included in this part is Tangent Hyperplane's implementation for all the versions described in chapters 3 and 4, as well as an implementation of Standard Backpropagation. The second part, §A.2, deals with the constructive technique for classification including the generalisation framework, chapters 6 and 7. Finally in §A.3 the format of weight and pattern files is presented.

A.1. Fixed Multilayer Feeforward Architectures

This appendix describes how to use the software with which all the results on chapter 5 were obtained. Two types of programs are used, interactive and batch programs.

The interactive simulator is ideal to run small scale experiments, testing individual trials with both Tangent Hyperplanes and Standard Backpropagation.

The batch programs are designed to perform extensive testing and have been designed so that they can run in background mode.

Together with the software a set of pattern files is presented for the 4 problems tested in chapter 5. The pattern's file format is described so that new pattern files for other problems can be constructed.

A.1.1. Simnet : Interactive Simulator

This simulator lets you run experiments in all variations of Tangent Hyperplanes and Standard Backpropagation.

To build the simulator make sure you have the following files in the working directory :

- nn.c (neural net's module)
- command.c (commands module)
- main.c (interface module)
- svd.c (SVD module)
- svd.h (external definitions for svd.c)
- nn.h (external definitions for nn.c)
- nrutil.h (definitions used in svd.c)
- nnconst.h (constants for every module)
- command.h (external definitions for command.c)
- makefile (instructions for make utility)

If you do have all the files in the working directory type "make simnet" followed by CR (CR stands for Carriage Return or Enter) at the prompt and the make utility of UNIX will create the executable file for the simulator. The executable file is called simnet, so if you want to run the simulator type "simnet" followed by CR.

The following list of options making up the main menu should appear :

- A - BackPropagation
- B - Tangent Hyperplanes
- C - Init Weights
- D - Quick Net
- E - Report Network
- F - Context Menu
- G - Load Patterns
- H - Reset
- I - Load Weights
- J - Save Weights
- L - Init TH
- M - I/O Map
- X - Quit

Note that although some care has been put for validating the options there is no safety mechanism preventing you from crashing the system. This is not a commercial software package and therefore is not fool proof. Please follow the steps carefully in order to avoid crashing the system.

The first thing you must do is to create an architecture, i.e. a neural network. Press "D" followed by CR. The simulator will now ask how many layers there are in the net including the input and output layer. If for instance you want a net to train the XOR problem press "3" followed by CR.

Next the simulator will ask you to specify the number of units in each layer. Press the desired number followed by CR for each layer. For instance in the XOR problem you should type 2 CR 2 CR 1 CR. If by any chance you get it wrong just introduce a "0" in any of the remaining layers and the simulator will consider the net as invalid. Note that you must introduce some number for every remaining layer. Once an architecture is defined there is no going back. If you want to change the architecture you'll have to quit the simulator and start all over again.

The next thing you must do is to load a set of patterns, make sure that the pattern files are in your working directory. The following pattern files are part of the package:

- xor.pp (patterns for the XOR problem)
- bitpar.pp (5 bit parity problem)
- spiral.pp (2 spirals problem)
- fa.pp (function approximation problem)

To load a set of patterns press "G" followed by CR. The simulator will ask you for the filename where the pattern set is. Assuming you want the XOR problem training set loaded type "xor.pp" followed by CR. At this stage if

you made a mistake there is no need to exit the simulator, just press "G" followed by CR and type the desired file name again also followed by CR.

From now on the options differ depending on whether you're using Standard Backpropagation or Tangent Hyperplanes. However before actually starting training is perhaps useful to introduce the context menu.

A.1.1.1. The Context Menu

The context menu is the menu where the training parameters like learning rate, momentum, etc., are defined. Press "F" followed by CR to access this menu. The following options are available although not necessarily relevant for both algorithms, Tangent Hyperplanes and Standard Backpropagation.

- 2 - Learning Rate
- 3 - Momentum
- 4 - Max. Epochs
- 5 - Final Error Tolerance
- 6 - Random Seed
- 7 - Limits
- 9 - BIAS
- B - TH mode
- C - TH params
- D - TH mode2
- X - Quit

This is the list of parameters for either Tangent Hyperplanes or Standard Backpropagation. You'll have probably noticed that numbers "1", "8", and option "A" are absent from the list. This is because some options have become useless and therefore have been removed from the simulator. However the original numbering of the options has been left in place so that operating the simulator continued to be a familiar process.

Note that, extra to the above list, the current value for the parameter is also presented in front of each item in the list. Only if you want to change the currently displayed value should you enter the option.

To access each of the options just press the relevant digit followed by CR. Then the simulator will ask you for new values for the parameter being altered. There are some validation constraints implemented, meaning that in some cases if an invalid value is introduced there will be no change from the current value displayed in the screen. However, as mentioned before, you should not rely on such safeguards to prevent you from crashing the system. Extreme care should be taken each time new values are introduced to avoid problems further ahead when training.

Once the system starts asking for values just type the desired values and press CR for each value introduced. Note that some options may require more than one value, these are options B, C and D.

The meaning, where needed, and scope of each option is now presented:

- **Learning Rate** : Only relevant to Backpropagation.
- **Momentum** : Only relevant to Backpropagation.
- **Max. Epochs** : The maximum number of epochs that the simulator will perform before halting. The system may stop before this number is actually achieved, see Final Error Tolerance. All versions.
- **Final Error Tolerance** : Defines the linear output error tolerance. When all patterns in the training set have reached the tolerance defined training will stop. All versions.

- **Random Seed** : Indicates the random seed used to initialise the weights. Different random seeds will, in principle, generate different initial weight states. All versions.

- **Limits** : Defines the limits within which the weights will be randomly initialised. Only one value is requested, the initial weight state interval is [-value,value].

- **BIAS** : By default the BIAS unit is active with value 1. However, you can turn it off if you want to by pressing "0" followed by CR. All versions.

- **TH params** : These parameters are for TH only. Their meaning is explained in the chapter 3 in full detail. The Subgoal version of TH needs all 3 parameters (L_W, L_O and T), the Line Search only needs the last parameter, T. The classifier version also only needs the last parameter.

- **TH mode** : TH only. Specifies if TH is run under the Subgoal or the Line Search versions. The options are displayed on the screen.

- **TH mode2** : TH only. This option allows you to select between the Normalised or Error Normalised versions.

- **Quit** : Select this option to go back to the main menu.

A.1.1.2. Starting Training

The material in this section assumes that there is already a network and a set of patterns in memory and the right parameters are set through the Context Menu. If this is not the case go back to the beginning and follow the instructions until you reach this section again.

After building the network, loading the training set, and setting the right parameters, one is almost ready to start. First press "C" followed by CR to

have the weights randomly initialised (or press "I" to load your own weights (see §A.1.1.3)). Otherwise the initial weight state will be the zero weight state. Now you can select your training algorithm to start training:

- Backpropagation;
- Tangent Hyperplanes.

If you are going to run Backpropagation, for example, you can now press "A" followed by CR to start training. On the screen, on each line of output the number of the last epoch done will be displayed as well as the linear error for each of the patterns.

If instead you want to train with Tangent Hyperplanes in any of the versions you must press "L" followed by CR before you start training. This option initialises the data structures used in Tangent Hyperplanes. Now to start training you can press "B". The data displayed for each epoch is as follows :

it = jump : dist : cos : Ok ; ME ; max ; min

where:

- it - number of current epoch;
- jump - size of jump using Euclidean distance measure;
- dist - total Euclidean distance to solution manifolds before the epoch occurred;
- cos - cosine between previous epoch and present epoch transitions in weight space;
- Ok - represents the number of patterns correctly classified;

- ME - displays the error pattern with the maximum linear error;
- max - displays the output for the pattern with the highest output value for a target below 0.5 (only relevant for classification problems);
- min - displays the output for the pattern with the lowest output value for a target above 0.5 (only relevant for classification problems).

There is no way to stop a training session until it halts either by achieving tolerance or the maximum number of epochs. That is unless you press CTRL C to exit the simulator.

A.1.1.3. The Remaining Options Available in the Main Menu

In this section the operation of each of the remaining options available from the main menu is described. It is assumed that a network has been built and a training set loaded.

- Report Network : This option asks you for a file name. A report on the network displaying the weight values and all connections is sent to a file named after the string you introduced.
- Reset : Reinitialises the weights and resets the 'epochs done' counter.
- Save Weights : Asks you for a file name. Saves the current weight state in the named file.
- Load Weights : Ask for a file name and loads a previously saved weight state from the named file.
- I/O Map : Only valid for nets with 2 inputs, e.g. it can't be used with the 5 bit parity problem. The result of this operation is a file named "io.com" that you can visualise graphically following the instructions described in §A.1.1.4 of this appendix.

- Quit : Quits the simulator, however first you'll be asked to confirm your intentions.

A.1.1.4. I/O Maps : How to visualise them

The software needed to convert the files produced by the simulator to raster files was produced by Gary Polhill during his work for a Ph.D. in the University of St. Andrews. He kindly agreed to let me use it for my own thesis for which I'm in debt to him.

First one must build the converter program. Make sure that in your working directory the following files are present:

- rasdraw.h
- gtop.h
- rasterdraw.h
- fgraph.c
- gtop.c
- rasterdraw.c
- makefile

If this is the case type "make fgr" followed by CR to build the converter.

Now it is described how to view the I/O map files created with the interactive simulator. To create a I/O map file see option "M" on the main menu. This produces a file called io.com. To visualise the results you must execute the following commands:

"fgr io.com io.ras" followed by CR, and

"loadscrn io.ras &" followed by CR.

The utility loadscrn allows one to visualise raster files. It should be available in your directory. If it isn't contact your system administrator and most likely he or she will be able to provide you with a solution.

You can also print the I/O map by typing

"lpr -v -PrinterName io.ras" followed by CR.

A.1.2. Batch Testing

There are five programs that do tests using Tangent Hyperplanes and one for Backpropagation:

thstest : Tests Tangent Hyperplanes with Subgoals.

thstest2 and thstest3 : Tests Tangent Hyperplanes with Subgoals extensively.

thlstest : Tests Tangent Hyperplanes with Line Search.

bptest : Tests Standard Backpropagation.

These programs take parameters on the command line as shown in each of the next subsections. Should it be the case that you forget the parameters just press the relevant name of the program and the list of parameters needed is displayed on the screen. Don't forget however that before you do this you should create the executables for the programs as described below.

A.1.2.1. Testing TH with Subgoals

Before making the executable file, thstest, make sure that the following files are in your working directory:

- makefile (instructions for make utility)
- command.c (commands module)
- nn.c (neural net's module)

- svd.c (SVD module)
- svd.h (external definitions for svd.c)
- nrutil.h (definitions used in svd.c)
- thstest.c (main module)
- nnconst.h (constants for every module)
- command.h (external definitions for command.c)
- nn.h (external definitions for nn.c)

To make the executable file type at the prompt "make thstest" followed by CR.

This program, thstest, is not interactive and therefore all options must be introduced when the program is called.

To use the program type the following command at the prompt :

"thstest p mode Lw Lc T wl tol met trials"

followed by CR. The options after the program's name are now discussed :

- p : defines the problem. the valid values for p are

  1 : XOR problem ;

  2 : 5 bit parity problem ;

  3 : 2 Spirals problem;

  4 : The function approximation problem

- mode : Mode defines if the test is made using Standard Normalisation or Error Normalisation. Valid values for mode are :

  0 : Normalised

  1 : Error Normalised

- Lw : defines the value for Lw as described in chapter 3. Any real positive value is valid.
- Lo : defines the value for Lo as described in chapter 3. Any real positive number smaller than 1 is valid.
- T : defines T as described in chapter 3. Any positive real number is valid.
- wl : defines the interval to initialise the weights. Any positive real number is valid. The weights will be initialised between [-value,value] where value represents the number introduced.
- tol : defines the final error tolerance. Any real positive value in [0,1[is valid.
- met : Maximum number of epochs per trial. Any positive integer number is valid.
- start stop : start with trial 'start' until trial 'stop' is achieved.

For example suppose you want to test Tangent Hyperplanes with Subgoals and Error Normalisation on the XOR problem with the following parameters {Lw = 0.5, Lo = 0.1, T = 0.00001}, having the weights initialised in [-0.1,0.1], with a tolerance of 0.35, for 100 trials having a maximum of 1000 epochs per trial. The respective command should be

"thstest 1 1 0.5 0.1 0.00001 0.1 0.35 1000 0 100"

followed by CR. The results are sent to a file named :

"TP*p_Lw_Lo_T_start_stop*"

If you want the process to run on the background then you should type :

"nohup thstest 1 1 0.5 0.1 0.00001 0.1 0.35 1000 100 &"

followed by CR.

A.1.2.2. Testing TH with Subgoals Extensively

These programs, thstest2 and thstest3, differ from the one described in §2.1 of this manual because they don't take the values of Lw, Lo and T as parameters when calling the program. These program tests all combinations of the following values for each of the parameters:

- Lw $\in$ {1.0, 0.5, 0.1}
- Lo $\in$ {0.1, 0.05, 0.01} (thstest2) Lo $\in$ {0.01, 0.005, 0.001} (thstest3)
- T $\in$ { 0.1, 0.001, 0.00001}

Otherwise these programs are equivalent to the one described in §2.1 called 27 times, i.e. one for each possible combination of parameters.

Before making the executable file, thstest2 or thstest3, make sure that the following files are in your working directory:

- makefile (instructions for make utility)
- command.c (commands module)
- nn.c (neural net's module)
- svd.c (SVD module)
- svd.h (external definitions for svd.c)
- nrutil.h (definitions used in svd.c)
- thstest2.c or thstest3.c (main modules)
- nnconst.h (constants for every module)
- command.h (external definitions for command.c)
- nn.h (external definitions for nn.c)

To make the executable file type at the prompt "make thstest2" or "make thstest3" followed by CR.

These programs, thstest2 and thstest3, are not interactive and therefore all options must be introduced when the program is called.

To use the thstest2 program type the following command at the prompt :

"thstest2 p mode wl tol met trials"

followed by CR. To use thstest3 replace thstest2 in the command line by thstest3. The options after the program's name are now discussed :

- p : p defines the problem. the valid values for p are

  1 : XOR problem ;

  2 : 5 bit parity problem ;

  3 : 2 Spirals problem;

  4 : Function approximation problem

- mode : Mode defines if the test is made using Standard Normalisation or Error Normalisation. Valid values for mode are :

  0 : Normalised

  1 : Error Normalised

- wl : defines the interval to initialise the weights. Any positive real number is valid. The weights will be initialised between [-value,value] where value stands for the value introduced.

- tol : defines the final error tolerance. Any real positive value from [0,1[is valid.

- met : Maximum number of epochs per trial. Any positive integer number is valid.

- trials : Number of trials. Any positive integer number is valid.

This program produces no output to the screen. Instead 27 files are produced, one for each combination of parameters with the following format.

In the first line the values for Lw, Lo, and T are presented. The remaining lines have the number of trial, number of epochs until separation is achieved and number of epochs until tolerance is achieved. There should be as many of these lines as the number of trials requested.

The names of the files holding the results is as follows :

"THres$ip.a.b.c$"

where i relates to the version being used (2 for thstest2, 3 for thstest3), p relates to the problem being trained, and a, b or c are integers with values between 0 and 2. a concerns the value of T, b the value of Lw, and c the value of Lo. The correspondences are the following :

- (a = 0, T = 0.1); (a = 1, T = 0.001); (a = 2, T = 0.00001)
- (b = 0, Lw = 1.0); (b = 1, Lw = 0.5); (b = 2, Lw = 0.1)
- (c = 0, Lo = 0.1); (c = 1, Lo = 0.05); (c = 2, Lo = 0.01) for thstest2 or

  (c = 0, Lo = 0.01); (c = 1, Lo = 0.005); (c = 2, Lo = 0.001) for thstest3

So if for instance, a file named "TPres21.1.1.1" means that this file contains the results of the experiments on the XOR problem obtained with thstest2 made with the following parameters :

- T = 0.001
- Lw = 0.5

- Lo = 0.05

If you want the process to run on the background then you could type, for example :

"nohup thstest2 1 1 0.1 0.35 1000 100 &"

followed by CR.

A.1.2.3. Testing TH with Line Search

Before making the executable file, thlstest, make sure that the following files are in your working directory:

- nn.c (neural net's module)
- command.c (commands module)
- thlstest.c (main module)
- svd.c (SVD module)
- svd.h (external definitions for svd.c)
- nn.h (external definitions for nn.c)
- nrutil.h (definitions used in svd.c)
- nnconst.h (constants for every module)
- command.h (external definitions for command.c)
- makefile (instructions for make utility)

To make the executable file type at the prompt "make thlstest" followed by CR.

This program, thlstest, is not interactive and therefore all options must be introduced in the command line when the program is called.

To use the program type the following command at the prompt :

"thlstest p mode norm T wl tol met trials"

followed by CR. The options after the program's name are now discussed :

- p : p defines the problem. the valid values for p are

  1 : XOR problem ;

  2 : 5 bit parity problem ;

  3 : 2 Spirals problem

- mode : Defines the error function to be use for the line search. Valid values for mode are :

  1 : Output Error

  2 : Euclidean Error

- norm : Defines the normalisation used. Valid values for mode are :

  0 : Standard Normalisation

  1 : Error Normalisation

- T : defines the value for parameter T as described in chapter 3. Any positive real number is valid.

- wl : defines the interval to initialise the weights. Any positive real number is valid. The weights will be initialised between [-value,value].

- tol : defines the final error tolerance. Any real positive value from [0,1[is valid.

- met : Maximum number of epochs per trial. Any positive integer number is valid.

- trials : Number of trials. Any positive integer number is valid.

For example suppose you wanted to test Tangent Hyperplanes with Line Search using Euclidean Error and Standard Normalisation on the XOR problem with parameter T = 0.00001, having the weights initialised in [-0.1,0.1], with a tolerance of 0.35, for 100 trials having a maximum of 1000 epochs per trial. The respective command should be

"thlstest 1 2 1 0.00001 0.1 0.35 1000 100"

followed by CR. The following results are displayed in the screen : number of trial and number of epochs needed to achieve tolerance. If the number of epochs displayed for a trial is equal to the maximum number of epochs then the respective trial has not been successful.

To send the results to a file do :

"thlstest 1 2 1 0.00001 0.1 0.35 1000 100 > filename"

followed by CR. Where filename stands for the file you want to send the data into. If you want the process to run on the background then you should type :

"nohup thlstest 1 2 1 0.00001 0.1 0.35 1000 100 > filename &"

followed by CR.

A.1.2.4. Testing Backpropagation

Before making the executable file, bptest, make sure that the following files are in your working directory:

- nn.c (neural net's module)
- command.c (commands module)
- bptest.c (main module)

- svd.c (SVD module)
- svd.h (external definitions for svd.c)
- nn.h (external definitions for nn.c)
- nrutil.h (definitions used in svd.c)
- nnconst.h (constants for every module)
- command.h (external definitions for command.c)
- makefile (instructions for make utility)

To make the executable file type at the prompt "make bptest" followed by CR.

This program, bptest, is not interactive and therefore all options must be introduced in the command line when the program is called.

To use the program type the following command at the prompt :

"bptest p lr m wl tol mne start stop"

followed by CR. The options after the program's name are now discussed :

- p : p defines the problem. the valid values for p are

  1 : XOR problem ;

  2 : 5 bit parity problem ;

  4 : Function approximation problem.

- lr : defines the value for the learning rate value.

- m : defines the momentum value.

- wl : defines the interval to initialise the weights. Any positive real number is valid. The weights will be initialised between [-value,value].

- tol : defines the value for parameter T as described in chapter 6. Any positive real number is valid.

- mne : Maximum number of epochs per trial. Any positive integer number is valid.

- start stop : Do trials from trial 'start' until trial 'stop'. Any positive integer number is valid for both 'start' and 'stop'.

For example suppose you wanted to test Backpropagation on the XOR problem with a learning rate of 1.0, momentum of 0.90, a tolerance of 0.35, having the weights initialised in [-0.1,0.1], for 1000 trials having a maximum of 1000 epochs per trial. The respective command should be

"bptest 1 1.0 0.9 0.1 0.35 1000 0 1000"

followed by CR.

The results are sent to a file named:

BPmne_lr_wl

where mne stands for the maximm number of epochs, lr stands for the learning rate, and wl defines the interval to initialise the weights. If you want the process to run on the background then you should type :

"nohup bptest 1 1.0 0.9 0.1 0.35 1000 0 1000 &"

followed by CR.

A.2. Classification Trees and Generalisation Framework

The software for Classification Trees has two programs. One program builds the tree for a particular training set using an introduced value for g. The

other program was developed to test the Classification Tree in a test set to check its generalisation ability.

A.2.1. Building the Classification Tree

To create the program make sure that the following files are in your working directory:

- class.c (C code for the program)
- makefile (instructions for compiling and linking all files)

If you do have all the files in the working directory type "make class" followed by CR (CR stands for Carriage Return or Enter) at the prompt and the make utility of UNIX will create the executable file for the simulator. The executable file is called class, so if you want to run the simulator type "class" followed by CR.

The following list of options making up the main menu should appear :

- 1 - Load Pattern File
- 2 - Set Constant g
- 3 - Solve it
- 4 - Save Tree
- 0 - Exit

The first thing you must do is to load a pattern file. Three pattern files with classification problems are available in the package, however the simulator only works in 2-D input space. Therefore the pattern files you can test the simulator with are:

- xor.pp (the XOR problem)
- spiral.pp (the 2 spirals problem)

To load a pattern file press "1" followed by the name of the file WITHOUT the extension. For instance to load the two spirals problem just input "spiral" instead of "spiral.pp".

You can set the constant g by selecting option 2. The default value for g is 0.0.

The Classification Tree can now be build, press "3" and the program will build the tree reporting the amount of time elapsed during the construction of the Tree. Note that the time shown is NOT CPU time. CPU time is generally smaller because it only counts the amount of CPU a process has taken disregarding the time when the process is in a queue waiting for the CPU to be available. This number may provide a good approximation to CPU time only when there are no more processes actively running.

To save the tree press "4" and the program will prompt you for a file name. The program will add the extension ".tre" to the file name given by the user. So if for instance you typed "spiral" the file name would be "spiral.tre"

A.2.2. Testing the Classification Tree

In order to test the classification tree a program, treetest, is provided. Make sure that the following files are in your working directory:

- treetest.c (C code for the program)
- makefile (instructions for compiling and linking all files)

If you do have all the files in the working directory type "make treetest" followed by CR (CR stands for Carriage Return or Enter) at the prompt and the make utility of UNIX will create the executable file for the simulator. The executable file is called treetest.

This program is not interactive. To use the program just type the program's name followed by the test set file and the tree file. Considering that you have the patterns in file *spiral.pp* and the tree in the file *spiralt.tre* just type

"treetest spiral spiral"

Note that the extensions for both the test set file and the tree file were left out. The program automatically adds the needed extensions to the file names introduced. If you just introduce the name of the program without any further arguments the program will output a message saying which parameters are needed.

The output of the program consists of two numbers:

- the number of non-terminal nodes in the tree;
- the number of patterns misclassified in the given test set by the tree.

If you wish to test a very large number of patterns the program can be run in background using for instance the following instruction

"nohup treetest spiral spiraltre > res &"

where *res* is the file which will have the results once the program is finished.

A.3. File Formats

In this section the description of the file's structure for holding patterns and weight is described. Whereas the pattern file format is common to all methods, the weight file format applies only to fixed Multilayer Feedforward architectures.

Following the descriptions below new pattern files can be created to experiment new problems. Also by creating a new weight file particular weight states can be considered as initial weight states for training using the fixed Multilayer Feedforward architectures

A.3.1. Pattern File Formats

The pattern files for all programs are text files with the following format : the first line indicates the number of patterns present in the file. Afterwards come the description of the sequence of patterns. Each pattern is described in the file as follows:

in_1
in_2
...
in_n
out_1
out_2
....
out_p

where in_i indicates the i^{th} input value from the input pattern and out_j is the j^{th} output value from the output pattern. Remember to press CR after each number.

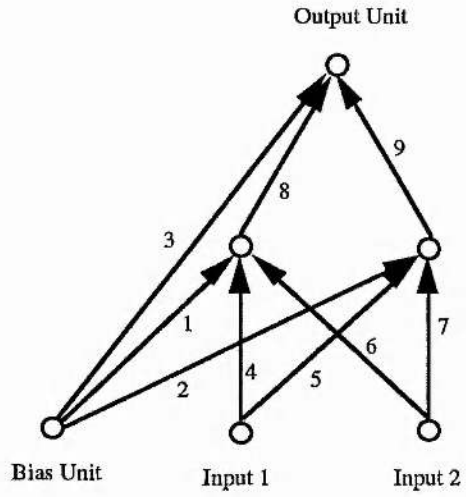
For example the XOR pattern file is as follows:

4
0
0
0.05
0
1
0.95
1
0
0.95
1
1
0.05

A.3.2. Weight File Formats

This section describes the format of the files used to keep weight states. Knowing the format allows you to build your own weight file that can serve as a specific initial weight state you want to test. Once a file is created it can be loaded using option "I" from the main menu, see §A.1.1.3.

An example is now provided to show the format. The network presented is the one used to solve the XOR problem. The numbers close to the weights indicate in which position they appear in the weight file.



From the figure one can see that first one has to indicate the weights from the bias unit to all other units in the net. Afterwards for each unit in each layer, starting in layer 0, the weights connecting that unit to the units in the proceeding layer are presented, and so on until the last unit on the last hidden layer.

The file format is a weight value per line so for instance the following text could refer to a weight file for the XOR problem:

```
-0.03
0.0212
0.054
0.084
0.014
-0.0644
-0.0081
-0.615
0.058
```

A.4. Automatic Random Pattern Set Generation

For the circle and line problems several training sets where needed. In order to build these training sets automatically two programs were created, one for each problem.

The files needed to create the executables are:

- circle.c for the circle problem;
- line.c for the line problem.

Also make sure that the *makefile* is in your working directory.

Both these programs behave in the same way, so only the circle program will be explained in here.

To create the executable type "make circle" at the prompt. The executable takes two parameters from the command line. The first parameter is the number of patterns for each training set. The second value is the number of training sets wanted. For example if one wants 10 different training sets with 100 patterns each the command line would be:

```
circle 100 10
```

Ten files would be created by the program, each one with a training set with 100 patterns. The structure of the file names is

circle*x*_*y*.pp

where x is the number of patterns is the training set, and y is a number between 0 and the number of training sets specified in the command line. Each file has a different value for y.

Bibliography

Anderson, James. A and Edward Rosenfeld eds., 1988, *Neurocomputing: Foundations of Research*, Cambridge, Massachusetts, MIT Press.

Atlas, L., Ronald Cole, Jerome Connor, Mohamed El-Sharkawi, Robert Marks II, Yeshwant Muthusamy and Etienne Barnard, 1989, "Performance Comparisons Between Backpropagation and Classification Trees on Three Real World Applications", in Proceedings of *Neural Information Processing Systems II*, Denver, Colorado.

Baum, E.B. and Kevin J. Lang, 1990, "Constructing Hidden Units Using Examples and Queries", Proceedings of *Neural Information Processing Systems*, Vol. III, Denver, Colorado, pp 904-910.

Bishop, C.M., 1993, "Neural Networks Validation: an Illustration from the Monitoring of Multi-phase Flows", in Proceedings of *III International Conference on Artificial Neural Networks*, pp41-50.

Bishop, C.M., 1995, *Neural Networks for Pattern Recognition*, Clarendon Press, Oxford.

Breiman, L, J.H Friedman, R.A. Olshen, C.J. Stone, *Classification and Regression Trees*, Wadsworth & Brooks, 1984.

Caudill and Butler, 1989, *Naturally Intelligent Systems*, Cambridge, Massachusetts, MIT Press.

Courrieu, Pierre, 1994, Three Algorithms for Estimating the Domain of Validity of Feedforward Neural Networks, Neural Networks, Vol. 7, No.1 pp169-174.

Denker, John, Daniel Schwartz, Ben Wittner, Sara Solla, Richard Howard, Lawrence Jackel and John Hopfield, 1987, "Large Automatic Learning, Rule Extraction and Generalization", *Complex Systems* I, pp 877-922.

Dietterich, Thomas G, Hermann Hild and Ghulum Bakiri, 1990, "A Comparison of ID3 and Backpropagation for English Text-to-Speech Mapping", in *Machine Learning Conference*.

Draghici, Sorin, 1995, *Using Constraints to Improve Generalisation and Training: Constraint Based Decomposition and Complex Backpropagation*, PhD. Thesis from the University of St. Andrews, Scotland.

Fahlman, Scott E., 1988, "An Empirical Study of Learning Speed in Back-Propagation Networks", Technical Report CMU-CS-88-162, Carnegie Mellon University, June 1988.

Fahlman, Scott E. and C. Lebiere, 1990, "The Cascade Correlation Learning Architecture" in Proceedings of *Neural Information Processing Systems II*, Denver, ed. D.S.Touretzky, pp 524-532, Morgan Kaufmann.

Fisher, R. A., 1936, "The use of Multiple Measurements in Taxonomic Problems" in *Annals of Eugenics* Vol. 7, pp 179-188.

Frean, M., 1990, "The Upstart Algorithm: A Method for Constructing and Training Feedforward Neural Networks", *Neural Computation* 2, pp 198-209.

Gallant, Stephen I., 1986, "Optimal Linear Discriminants", in Proceedings of the *IEEE Eight International Conference on Pattern Recognition*, Paris 1986, pp 849-852.

Gallant, Stephen I., 1993, *Neural Network Learning an Expert Systems*, Cambridge, Massachusetts,MIT Press.

Gibson, G.J., 1994, "Some Results on the Exact Realisation of Decision Regions Using Feed-Forward Networks with a Single Hidden Layer", Proceedings of the *IEEE International Conference on Neural Networks*, Orlando, USA, V2, pp. 912-917.

Golub, G.H. and C.F. Van Loan,1983, *Matrix Computations*, Baltimore, John Hopkins University Press.

Hebb, Donald O., 1949, *The Organisation of Behaviour*, New York: Wiley. Partially reprinted in Anderson and Rosenfeld, 1988.

Hecht-Nielsen, Robert, 1989, *Neurocomputing*, Addison-Wesley.

Hertz, J., Anders Krogh and Richard G. Palmer, 1991, *Introduction to the Theory of Neural Computation*, Lecture Notes Vol. I of the Santa Fe Institute, Addison-Wesley.

Hinton, Geoffrey E., 1986, "Learning Distributed Representation of Concepts", Proceedings of the *Eight Annual Conference of the Cognitive Science Society*, Amherst, pp 1-12, Hillsdale: Erlbaum.

Hinton, Geoffrey E., 1989, "Connectionist Learning Procedures", *Artificial Intelligence*, 40, pp 185-234.

Hornik, K., M. Stinchcombe and H. White, 1988, "Multilayer Feedforward Networks are Universal Approximators", Manuscript, Dept. of Economics, University of California at San Diego, June.

Irie, B and S. Miyake, 1988, "Capabilities of Three-layered Networks", Proceedings of *International Conference on Neural Networks*, Vol. I, pp 641-648, IEEE Press, New York, July.

Johansson, E.M., F.U. Dowla and D.M. Goodman, 1992, "Backpropagation Learning for Multilayer Feed-Forward Neural Networks Using the Conjugate Gradient Method", *International Journal of Neural Systems*, Vol. 2 no. 4, pp 291-301.

Jollife, I. T., 1986, *Principal Component Analysis*, Springer Verlag, New York.

Klimasaukas, Casimir C., 1991, "Neural Nets Tell Why", *Dr. Dobb's Journal*, April 1991.

Lang, Kevin and Geoffrey Hinton, 1990, "Dimension Reduction and Prior Knowledge in E-Set Recognition" in Proceedings of *Neural Information Processing Systems II*, Denver, ed. D.S.Touretzky, pp 178-185, Morgan Kaufmann

Lang, Kevin and Michael J. Witbrock, 1988, "Learning to Tell Two Spirals Apart", Proceedings of the *Connectionist Models Summer School*, Morgan Kaufmann.

Lapedes, A. and R. Farber, 1988, "How Neural Nets Work", Proceedings of *Neural Information Processing Systems*, Denver, Colorado, Anderson, D.Z. eds. pp 442-456, Am. Inst. of Physics, New York.

Le Cun, Yan, 1987, "Modeles Connexionnistes de l'Apprentisssage", Doctoral Dissertation, University of Pierre and Marie Curie, Paris, France.

Lippmann, R.P., 1987, "An Introduction to Computing with Neural Nets", *IEEE ASSP Magazine*, April 1987, pp 4-22.

McCulloch, Warren S. and Walter Pitts, 1943, "A Logical Calculus of the Ideas Immanent in Nervous Activity", *Bulletin of Mathematical Biophysics* 5: 115-133. Reprinted in Anderson and Rosenfeld, 1988.

Meggido, Nimrod, 1983, "Linear-Time Algorithms for Linear Programming in R^3 and Related Problems", *SIAM J. Comput.*, Vol 12 n° 4.

Mezard, M. and J. Nadal, 1989, "Learning in Feedforward Layered Networks: The Tiling Algorithm", *Journal of Physics* A 22, pp 2191- 2204.

Minsky, Marvin L. and Seymour A. Papert, 1969, *Perceptrons*, Cambridge, MIT Press. Partially reprinted in Anderson and Rosenfeld,1988.

Minsky, Marvin L. and Seymour A. Papert, 1988, *Perceptrons: Expanded Edition*, Cambridge, MIT Press.

Møeller, Martin Fodslette, 1993, "A Scaled Conjugate Gradient Algorithm for Fast Supervised Learning", *Neural Networks Journal*, Vol. 6, pp 525-533.

Nilsson, Nils J., 1965, *Learning Machines*, McGraw-Hill Book Company.

Parker, David B., 1985, "Learning Logic", Technical Report TR-47, Center for Computational Research in Economics and Management Science, Massachusetts Institute of Technology, Cambridge, MA.

Parker, David B., 1987, "Optimal Algorithm for Adaptive Networks: Second Order Direct Back Propagation and Second Order Hebbian Learning", Proceedings *IEEE Int. Conf. on Neural Networks*, Vol II, pp 593-600, San Diego, California.

Polak, E., 1971, *Computational Models in Optimisation*, Academic Press, New York.

Polhill, G., 1996, *Guaranteeing Generalisation in Neural Networks*, PhD. Thesis from the University of St. Andrews, Scotland..

Press, W.H., B.P Flannery, S.A. Teukolsky and W.T Vetterling, 1992, *Numerical Recipes in C*, Cambridge, Cambridge University Press.

Quinlan, J.R., 1983, "Learning Efficient Classification Procedures and Their Application to Chess Endgames", in *Machine Learning: An Artificial Intelligence Approach*, Vol I, pp 463-482, Palo Alto: Tioga Press.

Ripley, Brian, 1993, "Statistical Aspects of Neural Networks" in Networks and Chaos - Statistical and Probabilistic Aspects, Ed. Barndorff-Nielsen,Jensen and Kendal, Chapman & Hall.

Romaniuk , S.T. and Hall, L.O., 1993, "Divide and Conquer Neural Networks", *Neural Networks*, Vol. 6, pp 1105-1116, 1993.

Rosenblatt, Franck, 1958, "The Perceptron: A Probabilistic Model for Information Storage and Organisation in the Brain", *Psychological Review* 65: 386-408. Reprinted in Anderson and Rosenfeld, 1988.

Rosenblatt, Franck, 1961, *Principles of Neurodynamics*, Spartan Books, Washington DC.

Rumelhart, David, James L. McClelland and the PDP Research Group, 1986, *Parallel Distributed Processing: Explorations of the Microstructure of Cognition, Volume 1 : Foundations*, Cambridge MIT Press.

Stephen, Scott, 1996, "BICON: A Bidirectional Neural Network System", Computer Science Report, St. Andrews University.

Sejnowski, Terrence and Charles Rosenberg, 1986, "NETtalk: a Parallel Network That Learns to Read Aloud", The Johns Hopkins University Electrical Engineering and Computer Science Technical Report JHU/EECS-86/01.

Siestma, J and R.J.F. Dow, 1988, "Neural Nets Pruning - Why and How", Proceedings of International Conference on Neural Nets, Vol I, pp 325-333, San Diego, California, USA.

Silva, Fernando and Luis Almeida, 1990, "Acceleration Techniques for the Backpropagation Algorithm", Lecture Notes in Computer Science, Vol 412, pp 110-119, Springer-Verlag.

Sirat, J. and Nadal J.P., 1990, "Neural Trees: New Tools for Classification", Networks 1, pp. 423-438.

Smagt, Patrick Van Der, 1994, "Minimisation Methods for Training Feedforward Neural Networks", *Neural Networks Journal*, Vol. 7, pp 1-11.

Telfer, Brian A. and Harold H. Szu, 1994, "Energy Functions for Minimising Misclassification Error With Minimum-Complexity Networks", *Neural Networks Journal*, Vol 7. no. 5, pp 808-818

Turban, Efraim, 1992, *Expert Systems and Applied Artificial Intelligence*, Macmillan Publishing Company, New York.

Webb, A.R., David Lowe, M.D. Bedworth, 1988, "A Comparison of Non-Linear Optimisation Strategies for Feed-Forward Adaptive Layered Networks", RSRE Memorandum 4175, Royal Signals and Radar Establishment, St. Andrew's Road, Malvern, UK.

Weir, K. Michael and António Fernandes, 1994, "Tangent Hyperplanes and Subgoals as a Means of Controlling Direction in Goal Finding", Proceedings of the *World Conference on Neural Networks*, Vol III, pp 438-443, San Diego, California USA.

Weir, K. Michael, C. Lansley and A. Clark, 1993, "The Minimum Topology Finder", Project Report, St. Andrews University.

Weir, K. Michael and Li H. Chen, 1990, "Training and Generalisation Using Continuos Back-Propagation", *Technical Report CS/90/12*, St. Andrews University, Computational Science Division, Scotland.

Werbos, Paul, 1974, *Beyond Regression: New Tools for Prediction and Analysis in the Behavioural Sciences*, PhD Thesis, Harvard University.

Widrow, Bernard and Martian E Hoff, 1960, "Adaptive Switching Circuits", *IRE WESCON Convention Record*, New York: IRE, pp 96-104. Reprinted in Anderson and Rosenfeld, 1988.

Widrow, Bernard, 1987, ADALINE and MADALINE, Proceedings IEEE 1st International Conference on Neural Networks, Vol I, pp 143-158, San Diego, California.

Yao, F.F., 1992, "Computational Geometry", Handbook of Theoretical Science, Vol A: Algorithms and Complexity, Ed. Leeuwen, J. V., Cambridge, Massachusetts, MIT Press.