Training and Optimization of Product Unit Neural Networks

by

Adiel Ismail

Submitted in partial fulfillment of the requirements

for the degree

Master of Science

in the Faculty of Natural & Agricultural Science

at

The University of Pretoria

Pretoria

14 November 2001

Declaration

I the undersigned hereby declare that the work contained in this thesis is my own original work and has not previously in its entirety or in part been submitted at any university for a degree.

Adiel Ismail

14 November 2001

Abstract

Product units in the hidden layer of multilayer neural networks provide a powerful mechanism for neural networks to efficiently learn higher-order combinations of inputs. Training product unit neural networks using local optimization algorithms is difficult due to an increased number of local minima and increased chances of network paralysis. This thesis discusses the problems with using local optimization, especially gradient descent, to train product unit neural networks, and shows that particle swarm optimization, genetic algorithms and leapfrog are efficient alternatives to successfully train product unit neural networks. Architecture selection, i.e. pruning, of product unit neural networks is also studied and applied to determine near optimal neural network architectures that are used in the comparative studies.

Opsomming

Produk-eenhede in die versteekte laag van multi-vlak neurale netwerke verskaf 'n kragtige meganisme aan neurale netwerke om hoë-orde kombinasies van invoer doeltreffend aan te leer. Die leer van neurale netwerke met produk-eenhede word bemoeilik weens die verhoogde aantal lokale minima teenwoordig, asook die verhoogde kans om netwerk paralise te ondervind. Hierdie tesis spreek die probleme aan wanneer lokale optimeringsmetodes gebruik word, veral in die geval van gradientdaling om produk-eenheid neurale netwerke te leer en dui aan dat partikel swerm optimering, genetiese algoritmes en *'leapfrog'* optimering baie doeltreffende alternatiewe is om produk-eenheid neurale netwerke te leer. Argitektuurseleksie, of te wel besnoeiing, van produk-eenheid neurale netwerke word ook bestudeer en toegepas om optimale neurale netwerk argitekture te bepaal, wat gevolglik in die vergelykende studies gebruik word.

Acknowledgements

I wish to thank the following people,

- my supervisor, Prof AP Engelbrecht, for all the positive criticism and useful suggestions.
- my wife, Shirene for her patience and moral support.
- my friend and colleague, Michael Norman, for his advice and encouragement.
- my colleague, Anwar Vahed, for the assistance provided during this project.
- my mother who single-handedly, as a widow from 1980, reared me, my three brothers and sister.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The recent resurgence of interest in neural networks can be ascribed to the recognition that the brain does not perform calculations in the same way as conventional (Von Neumann) computers. Despite the fact that computers execute instructions at extremely fast speeds, human beings whose brains operate at much slower speeds still outperform computers at tasks such as speech recognition, face recognition, etc. The human brain consists of an extremely large number of interconnected nerve cells, or neurons, which operate in parallel to process information. An artificial neural network (ANN) is an information processing system that mimics the structure and operating principles found in the information processing systems of human beings. The study of neural networks is one of the most rapidly expanding fields attracting researchers from a wide variety of disciplines such as biology, engineering, linguistics, mathematics, medicine, neuroscience, physics, psychology and statistics. ANNs have been applied successfully in many applications such as speech recognition [Cohen *et al* 1993], handwritten character recognition [Guyon 1990], steering of an autonomous vehicle [Pomerlau 1989], medical diagnosis of heart attacks [Harrison *et al* 1991], radar target detection and classification [Haykin *et al* 1992], and many more.

1.1 Why Product Unit Neural Networks?

Standard neural networks use summation units (SUs), where the net input signal to a unit is the weighted sum of the inputs connected to that unit. Research has shown that these summation unit neural networks (SUNNs) can approximate any continuous function to an arbitrary degree of accuracy, provided that the hidden layers contain a sufficient number of hidden units [Funahashi 1989, Hornik *et al* 1989a]. However, these networks require a large number of summation units (SUs) when approximating complex functions that involve higher order combinations of its inputs [Leerink *et al* 1995]. When approximating polynomials, higher-order combinations of inputs, such as x^3y^7, are often required. Networks that utilize higher-order combinations of its inputs will greatly reduce the number of processing units required to represent these complex functions [Janson *et al* 1993].

Several neural network models have been developed to gain an advantage in using higher-order terms [Gurney 1992, Leerink *et al* 1995, Redding *et al* 1993]. Examples of these higher-order neural networks are: pi-sigma network (PSN) [Ghosh *et al* 1992], sigma-pi networks [Lee Giles 1987], second-order neural networks [Milenković *et al* 1996] and functional link neural networks [Ghosh *et al* 1992, Hussain *et al* 1997, Zurada 1992]. An alternative network that also employs higher-order terms, is a product unit neural network (PUNN), where the net input is now a product of terms; each term consisting of an input raised to a weight [Durbin *et al* 1989]. Advantages of PUNNs are increased information capacity and the ability to form higher-order combination of inputs. Durbin and Rumelhart determined empirically that the information capacity of product units PUs (as measured by their capacity for learning random boolean patterns) is approximately $3N$, compared to $2N$ of a SU network for a single threshold logic function, where N denotes the number of inputs to

the network [Durbin *et al* 1989].

The next section briefly describes the problems associated with the training of PUNNs using back-propagation by gradient descent.

1.2 Problems with Training Product Unit Neural Networks using Gradient Descent

The back-propagation algorithm, independently developed by Werbos [Werbos 1974] and Bryson [Bryson *et al* 1969], provides a computationally efficient method for the training of multilayer neural networks. Its greatest strength is in finding non-linear solutions to ill-defined problems [Haykin 1994]. Unfortunately, the search space for PUNNs can be extremely convoluted, with numerous local minima that trap gradient descent [Durbin *et al* 1989, Leerink *et al* 1995]. While it is possible for local minima to occur in SU networks, they are particularly prevalent in networks containing PUs, due to the effect of the exponential terms in the learning equations. Thus, while PUs increase a neural network's capability, they also add complications in the training process. Although gradient descent has shown to be successful in training SUNNs, gradient descent fails to train PUNNs in general. The reason for its failure is discussed in more detail in section 3.8 on page 74. Gradient descent requires auxiliary information such as function derivatives, in order to calculate the minimum. The search space of PUNNs have an increased number of local minima, deep ravines and valleys, often surrounded by steep gradients that lead to huge adjustments of the weights when gradient descent is used, and consequent saturation.

1.3 Global Optimization Algorithms to Train PUNNs

What is needed, is a global optimization method instead of gradient descent, which is a local optimizer, to allow searching for larger parts of the search space, and which has the ability to get out of local minima. Genetic algorithms (GA) and particle swarm optimization (PSO) are global optimization methods that do not require auxiliary information, such as function derivatives, about the function being approximated in order to calculate the minimum. Leapfrog optimization (LFOP), a derivative based global optimizer, will also be used in training PUNNs. Each optimization algorithm has its own set of parameters. Optimal parameters are determined for each of these optimization algorithms. This thesis is dedicated to the training of PUNNs using PSO, GA and LFOP. Architecture selection, i.e. pruning, of PUNNs is also studied and applied to determine near optimal neural network architectures. The variance nullity pruning algorithm of Engelbrecht is applied to PUNN to determine near optimal network architectures [Engelbrecht *et al* 1999c, Engelbrecht 2001].

1.4 Objectives

The main objective of this thesis is to illustrate that gradient descent fails to train PUNNs and to show that global optimization algorithms, such as particle swarm optimization, genetic algorithms and leapfrog optimization, are more successful at training PUNNs.

The second objective is to determine which global optimization algorithm is more efficient and robust in training PUNNs. This thesis assumes a PUNN architecture with a

bias (for an explanation of a bias, refer to section 2.7 on page 15) to the output units and no bias to the hidden units. Instead, an extra unit, referred to as a 'distortion unit', is included in the hidden layer (refer to section 3.7.3 on page 72 for an explanation of the distortion unit). In this case, product units compute the net input signal as:

$$net_{y_j} = \prod_{i=1}^{I+1} z_i^{v_{ji}} \tag{1.1}$$

instead of

$$net_{y_j} = \prod_{i=1}^{I} z_i^{v_{ji}} + z_{I+1} \cdot v_{j,I+1} \tag{1.2}$$

where net_{y_j} is the net input to hidden unit Y_j, I is the total number of input units, Z_i is an input unit, z_i is an input signal to unit Z_i, v_{ji} is the weight between input unit Z_i and hidden unit Y_j. In equation (1.2), the threshold (or bias) is denoted by $v_{j,I+1}$, z_{I+1} is the input to the bias unit and has a value of -1. In equation (1.1), z_{I+1} cannot be viewed as the input to the bias, since it does not perform the function of a bias, i.e. it does not act as an offset to the other hidden units, but rather distorts the activation function to more accurately fit the data. In this case, z_{I+1} is now referred to as the input to the 'distortion' unit, Z_{I+1} with value -1, clearly distinguishing it from a bias unit. Note, equation (1.1) does not contain a bias to the hidden units. The linear activation function is assumed for the hidden and output units of the PUNNs, while the sigmoidal activation function is assumed for the hidden and output units of the SUNNs.

The third objective is to find the smallest architecture in training PUNNs for a particular function using the variance nullity pruning algorithm of Engelbrecht [Engelbrecht *et al* 1999c, Engelbrecht 2001]. The thesis also compares the pruned architectures of PUNNs and SUNNs to determine whether there is any gain in architecture complexity and performance using PUs.

1.5 Outline of the Thesis

The thesis is organized as follows. Chapter 2 presents an overview of ANNs. Topics covered include: description of a typical ANN, advantages and limitations of ANNs, training of ANNs, types of network architectures, activation functions, classification of learning rules, different learning paradigms, performance measures for ANNs, approximation capabilities of feed-forward neural networks, architecture selection and back-propagation by gradient descent.

An overview of the different higher-order neural networks such as pi-sigma, sigma-pi, functional link and second-order neural networks is given in chapter 3. A product unit neural network is described and the training rule for PUNNs presented. The problems associated with training of PUNNs using gradient descent are also investigated.

Chapter 4 is dedicated to a detailed discussion of genetic algorithms, particle swarm optimization and leapfrog optimization. Optimal parameters for each optimization algorithm are determined for each of the test functions. Results of these global optimization algorithms applied to 8 test functions are also discussed.

Chapter 5 discusses pruning of artificial neural networks. The variance nullity pruning algorithm of Engelbrecht is then adapted to prune PUNNs [Engelbrecht *et al* 1999c, Engelbrecht 2001]. Results of pruning are tabulated for PUNNs.

Chapter 6 summarizes the main conclusions and the goals, shortcomings and possible

improvements are suggested.

Chapter 2

Background

An important aspect of this thesis is to compare the performance of summation feed-forward neural networks with product unit neural networks using global optimization algorithms. The objective is to test the hypothesis that global optimization algorithms are more successful in training product unit neural networks (PUNNs) than local optimization algorithms. In this chapter an overview of ANNs is given. Issues regarding training of neural networks (NNs), learning algorithms and neural network architectures are addressed. Another important aspect of this thesis is the approximation of functions using feed-forward neural networks. It is therefore important to investigate the approximation capabilities of feed-forward neural networks for continuous functions and determine an appropriate architecture for such approximations.

2.1 A Brief History of ANNs

Attempts to mimic the human brain date back to work in the 1930's, 1940's and 1950's by Alan Turing, Warren McCullough, Walter Pitts, Donald Hebb and James von Neumann. Neural network simulations appear to be a recent development. However, this field was established before the advent of computers. The first artificial neuron was

produced in 1943 by the neurophysiologist Warren McCullogh and the logician Walter Pitts [Pitts *et al* 1943]. These neurons were presented as conceptual components for circuits that could perform computational tasks. In 1957 Rosenblatt at Cornell University developed 'Perceptron', a hardware neural network for character recognition. Much of Rosenblatt's work is described in his book 'Principles of Neurodynamics' [Rosenblatt 1962]. One of the most significant results presented in this book, was the proof that a simple training procedure, i.e. the perceptron training rule, would converge if a solution to the problem existed. In 1959 Widrow and Hoff at Stanford University developed Adaline for adaptive control of noise on telephone lines. The 1960's and 1970's period was hindered by inflated claims and criticism of early work. When Minsky and Papert published their book Perceptrons in 1969 [Minsky *et al* 1969] in which they pointed out the deficiencies of perceptron models, most neural network funding was redirected and researchers left the field. Minsky and Papert showed that there is an interesting class of problems that single layer perceptrons cannot solve, and they also held out little hope for the training of multilayer systems that might deal successfully with some of these deficiencies. Only a few researchers continued their efforts, most notably Teuvo Kohonen, who was investigating nets that used topological features [Kohonen 1988b], Stephen Grossberg was laying the foundations for his Adaptive Resonance Theory (ART) [Grossberg 1987], and Kunihiko Fukushima was developing the cognitron [Fukushima 1975].

In 1982 Hopfield, a Caltech physicist, tied together many of the ideas from previous research and showed that a highly interconnected network of threshold logic units could be analyzed by considering it to be a physical dynamic system possessing an 'energy' [Hopfield 1982]. A similar breakthrough occurred in connection with feed-forward networks, when it was shown that the 'credit assignment problem' (i.e. the contribution that each unit makes to the error the network has made in

processing the current training vector) had an exact solution. The interest in neural networks re-emerged only after some important theoretical results were attained in the early eighties (most notably the discovery of the error back-propagation [Parker 1985, Rumelhart *et al* 1986b, Werbos 1974]) and new hardware developments increased the processing capacities. This renewed interest is reflected in the number of scientists, the amounts of funding, the number of large conferences and the number of journals associated with neural networks.

The next section defines the term ANN.

2.2 What is An Artificial Neural Network?

There is no universally accepted definition for an artificial neural network. There are several definitions of an ANN. Zurada defines ANNs as 'physical systems which can acquire, store and utilize experiential knowledge' [Zurada 1992]. Aleksander defines neural computing as ' the study of adaptable nodes which, through a process of learning from task examples, store experiental knowledge and make it available for use' [Aleksander *et al* 1990]. Haykin defines ANN as 'a massively parallel distributed processor that has a natural propensity for storing experiential knowledge and making it available for use' [Haykin *et al* 1992]. Fausett defines an ANN as 'an information processing system that has certain performance characteristics, such as adaptive learning, and parallel processing of information, in common with biological neural networks' [Fausett 1994]. Nigrin defines an ANN 'as a circuit composed of a very large number of simple processing elements that are neurally based. Each element operates only on local information. Furthermore, each element operates asynchronously, thus there is no overall system clock' [Nigrin 1993].

From these definitions we can conclude that an ANN

- consists of several simple processing elements called units;
- is well suited for parallel computations, since each unit operates independently of the other units;
- contains a high degree of interconnections between units;
- contains links between units, each with a weight (scalar value) associated with it;
- has adaptable weights that can be modified during training.

2.3 Advantages of Neural Networks

ANNs offer several advantages, including:

- **Adaptive learning**: A neural network is a dynamic system which has a built-in capability to adapt its weights to changing environments.
- **Self-organization**: An artificial neural network can create its own organization or representation of the information it receives during learning. There is little need for extensive characterization of the problem other than through training.
- **Generalization**: Neural networks are able to extrapolate to a certain extent from the training to previously unseen data.
- **Graceful degradation**: Partial destruction of a network leads to a corresponding degradation of performance. However, network capabilities such as generalization may be retained even with major network damage. Neural networks have a gradual rather than sharp drop-off in performance as conditions worsen [Kohonen 1988a].

2.4 Limitations of Neural Networks

Neural networks have some important limitations, namely:

- ANNs have *poor explanation facilities.* There are no facilities for *justifying answers* and responding to what or how questions.
- ANNs are not very good at performing *symbolic computations.* They cannot be used effectively for rule based reasoning and arithmetic operations.
- The accuracy of an ANN's *performance is dependent upon the quality of the training examples.* It is difficult to find a complete and accurate set of training examples in real world problems.

The next section justifies the use of ANNs.

2.5 Why Artificial Neural Networks?

Neural networks take a different approach to problem solving than that of conventional computers. Conventional computers use an algorithmic approach, i.e. the computer follows a set of instructions to solve a problem. The computer can solve a problem only if the specific steps that the computer needs to follow are known. The problem solving of conventional computers is therefore restricted to problems that we already understand and know how to solve. Neural networks, on the other hand, with their remarkable ability to derive meaning from complicated or imprecise data, can be used to extract patterns and detect trends that are too complex to be noticed by either humans or other computer techniques. The ability of neural networks to learn by example, make them suitable for tasks that cannot be solved algorithmically. One of the distinct strengths of neural networks is their ability to generalize. The network is said to generalize well when it sensibly interpolates input patterns that are new to the

network. Neural networks provide, in many cases, input-output mappings with good generalization capability. It can be said that neural networks behave as trainable, adaptive and even self-organizing information systems [Schalkoff 1997].

The following section describes the main classes of ANN applications.

2.6 Classes of ANN Applications

The following classes of neural network applications can be found.

1. **Pattern Classification**

   Pattern classification concerns the classification of patterns into a fixed number of categories. The network is first trained on a set of patterns along with the categories to which each pattern belongs. Once the network is trained, a new pattern is presented to the network to be categorized. An example of a neural network classifier is the EEG (electroencephalogram) spike detector developed by Eberhart and Dobbins [Eberhart *et al* 1990]. The EEG spike detector successfully identifies an EEG spike which indicates an imminent epileptic seizure in patients. Despite the few false alarms recorded, the performance of the network has been found to be significantly better than that required for practical application in hospitals [Eberhart *et al* 1990].

2. **Association or Pattern Completion**

   In association each training pattern is associated with an image stored in the network. Association can be subdivided into autoassociation and heteroassociation. In autoassociation a neural network is repeatedly presented with a set of patterns to be stored by the network. After training, a partial description of the original pattern is presented to the network, the task is then to retrieve the original pat-

tern. In heteroassociation an arbitrary set of patterns are paired with another arbitrary set of patterns. After training, when a partial description of the original pattern of the first set is presented to the network, the task is to retrieve the pattern paired off with the original pattern. Applications include the 'Human Face Detection Network' of Rowley *et al* [Rowley *et al* 1996] and the NETtalk neural network of Sejnowski and Rosenberg that produced phonetic strings which specified pronunciation for English text [Sejnowski *et al* 1987].

3. **Approximation**

   Approximation requires a neural network to approximate a non-linear function or time-series given a set of patterns in the form of input and desired (target) output pairs. Once the network is trained, the neural network is then used to calculate an output for patterns not used in training (i.e. the neural network interpolates). An application of approximation is weather forecasting [Hsieh *et al* 1998] and forecasting the behaviour of multivariate time series [Chakraborty *et al* 1992].

4. **Clustering**

   The objective of clustering networks is to group similar patterns into groups, or clusters. Similarity is usually measured as the Euclidean distance between patterns [Kohonen 1988a]. Clustering was achieved by the Kohonen network that simply inspects the data for regularities, and organizes itself in such a way as to form an ordered description of the data [Bilbro *et al* 1989, Kawato 1990]. Feature detection aims at detecting a subset of input data or features which is relevant for a given problem. Feature detection is usually related to the dimensionality reduction of data [Saund 1989]. More sophisticated processing methods can then be applied to the smaller dimensional spaces. Applications of feature selection clustering has been applied to document classification to enhance information retrieval [MacLeod 1990].

5. **Control**

   There have been a number of successful applications to control systems. Application fields range from process control, robotics, industrial manufacturing, aerospace applications and vehicle and automobile control [Pomerlau 1989]. The basic objective of control is to provide the appropriate input signal to a given physical process to yield its desired response. Neural networks for control were developed by Werbos [Werbos 1989] and Jordan *et al* [Jordan *et al* 1990]. The term neuro-control has been coined by Werbos to refer to the class of controllers that involve the use of neural networks [Werbos 1974].

6. **Optimization**

   The objective of neural networks in optimization application is to optimize certain cost functions. Neural networks have successfully been applied to optimization problems such as job-shop scheduling [Foo *et al* 1988]. Problems that are simpler but which belong to the same group of optimization tasks include scheduling classrooms to classes, hospital patients to beds, etc. [Zurada 1992].

2.7 A Typical Artificial Neural Network

An artificial neural network (ANN) consists of interconnected artificial neurons, organized in a layered structure. Usually, all the neurons of a current layer are connected to neurons that occur at the next immediate layer. An artificial neuron receives a number of inputs (either from the given input pattern, or from the output of other neurons in a previous layer of the neural network). Each input comes via a connection which has a strength (or weight) associated with it. Each neuron also has a single threshold value (also referred to as a bias). The input to a neuron can be excitatory if they cause the firing of a neuron, or inhibitory if they hinder the firing

of a response. A more precise condition for firing is that the excitation should exceed the inhibition by the threshold. In mathematical terms the net input of neuron j is usually $net_j = \sum_i^I z_i w_{ji} - \theta_j$ where $z_1, z_2, ...z_I$ are the input signals, $w_{j1}, w_{j2}, ...w_{jI}$ are the synaptic weights leading to neuron j, net_j is the neuron's net input and θ_j is the threshold.

An activation function is used to determine the output signal based on a net input signal. In summation unit neural networks (SUNNs) the threshold can be treated as any other weight, by adding an extra unit, Z_{I+1}, whose input z_{I+1} is -1 and whose weight, $w_{j,I+1}$ is θ_j. The net input signal for this augmented network is computed as $net_j = \sum_i^{I+1} z_i w_{ji}$. The activation signal, or net input, is passed through an activation function (also known as a transfer function) to produce the output signal of the neuron. The activation function, also called the squashing function, often squashes or limits the permissible amplitude range of the output signal to some finite value; except in the case of linear functions where the output is unlimited. A neural network is trained by adjusting the weights of the neural network and thresholds so as to minimize the error in its output on the training data. If the network is properly trained, it has then learned to model the unknown function which relates the input variables to the target variables, and can subsequently be used in predictions where the target is not known.

2.8 Learning

Neural networks, like human beings learn from examples. This feature distinguishes neural networks from conventional programming paradigms. In conventional computer programming the relationship between the output and the input must be well defined. In the case of neural networks, this requirement is not needed. In fact, the strength of neural networks lies in their ability to learn the relationship between the input

and the output, given a set of representative examples. One of the most significant attributes of a neural network is its ability to learn by interacting with its environment or with an information source. Learning or training of a neural network is normally accomplished through a learning rule or algorithm, whereby the weights of the network are incrementally adjusted so as to improve a predefined performance measure over time. Essentially, learning of a neural network entails presenting a training pattern at the input units, resulting in an actual output to be produced by the network. The error between the desired output and actual output is then determined. The synaptic weights are then subsequently adjusted so as to reduce the error between the desired and actual output. The entire set of training patterns is usually used in adjusting the weights during the training process. The training terminates when an acceptable training error is reached. On a definition for 'learning', Minsky noted that there are too many notions associated with *learning* to justify the term in a precise manner [Minsky 1961]. As stated in section 2.2, Aleksander [Aleksander *et al* 1990] defines neural computing as "The study of networks of adaptable nodes which, through a process of learning from task examples, store experiential knowledge and make it available for use."

The next section introduces a typical artificial neuron that is the basic building block for neural networks.

2.9 Model of An Artificial Neuron

A neuron is an information processing element that is fundamental to the operation of a neural network. Figure 2.1 represents a model of a neuron [Haykin 1994]. A neuron consists of three basic elements:

1. A set of synapses or connecting links, each with its own strength or weight.

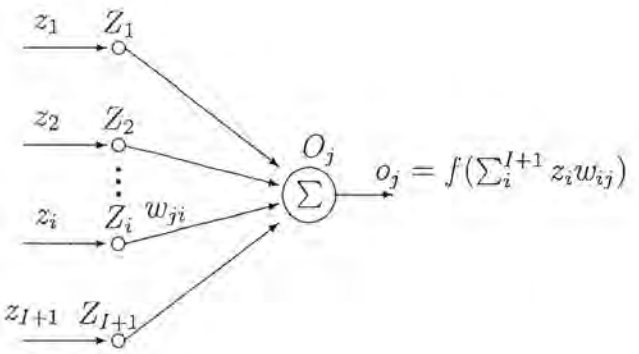


Figure 2.1: Model neuron using a Summation Unit

2. An adder, $\sum$, that computes the weighted sum of the signals in the case of summation unit networks or a multiplier, $\prod$, in the case of product unit neural networks that performs weighted multiplication of the signals.

3. An activation function that squashes the amplitude of the neuron to a finite range if a bounded activation function is used or an infinite range when an unbounded activation function, such as the identity function, is used.

In figure 2.1, the input signals are denoted by $z_1, z_2, z_3, ..., z_I$ and $w_{j1}, w_{j2}, w_{j3}, ..., w_{jI}$ are the synaptic weights of neuron O_j, o_j is the output of neuron O_j, $Z_1, Z_2, Z_3, ..., Z_I$ are the input units and $f(\cdot)$ is the activation function. In this model an additional input signal, z_{I+1}, fixed at -1 and synaptic weight, $w_{j,I+1}$, is added to the neuron to represent the threshold, or bias. This additional unit is referred to as the bias unit. A bias is added to the hidden and output units only. Neurons can be combined in different ways to construct neural networks such as feed-forward, recurrent neural networks, etc.

The next section discusses the different types of network architectures.

2.10 Network Architectures

There are three basic classes of network structures, namely, single-layer, multilayer and recurrent neural networks.

1. **Single-layer feed-forward neural networks**

   A single-layer feed-forward neural network as defined by Haykin [Haykin 1994] consists of an input layer of units that are connected to an output layer of nodes. The input layer of nodes is not counted as a layer since no computation is performed in this layer.

2. **Multilayer feed-forward neural networks**

   Refer to figure 2.3 on page 23 for an illustration of a multilayer feed-forward neural network with one hidden layer. A multilayer feed-forward neural network contains at least one or more hidden layers situated between the input and output layers. Neurons that occur in the hidden layer network are referred to as hidden units. In a feed-forward neural network, links are unidirectional, and there are no cycles. In a layered feed-forward neural network, each unit is usually linked to units in the next layer, although direct connections between the input and output layers are possible. There are no links between units in the same layer, and thus no computational dependencies between units in the same layer. This allows the outputs of these units to be computed in parallel. Also, no links point backwards to a previous layer. The units in the input layer receive signals from the environment and distribute the signals to the next layer in the network. The hidden layer(s) enable a network to extract higher-order statistics and thus provides the network with a global perspective, because of the extra set of synaptic connections and the extra dimension of neural interactions [Churchland *et al* 1992]. The output

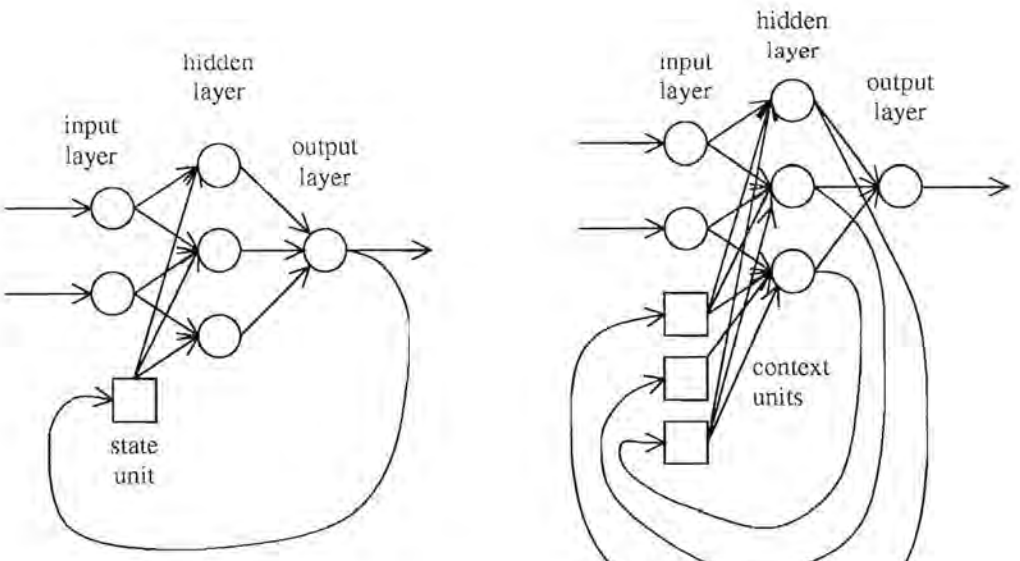
layer provides results of the network to the environment. Each neuron in the network provides an output which is a weighted sum, in the case of summation unit networks, or a weighted product of terms in the case of product unit neural networks. A feed-forward neural network has no memory and the output is solely determined by the inputs and synaptic weights.

3. **Recurrent Neural Networks (RNNs)**

A recurrent neural network contains at least one feedback loop, where the activations of the hidden units are fed as the network's inputs. The feedback loops in recurrent networks have a profound impact on the learning capability of the network and its performance when data exhibit temporal tendencies or characteristics. Temporal learning is concerned with capturing a sequence of patterns necessary to achieve some final outcome. In temporal learning, the current response of the network is dependent on previous inputs and responses. Through the feedback connections, RNNs can learn temporal characteristics of data presented for learning, thereby exhibiting properties very similar to short term memory in human beings. Recurrent networks are dynamic in the sense that their state is changing continuously until an equilibrium is reached. There are different types of RNNs, namely, Jordan and Elman RNNs, as shown in figure 2.2. In the Jordan RNN, the activation values of the output units are fed back into the input layer through a set of extra inputs referred to as the state units [Jordan 1986]. There are as many state units as there are output units in the Jordan network. The connections between the output and state units usually have a fixed weight of 1. Learning takes only place in the connections between input and hidden units as well as hidden and output units. In the Elman RNN a set of context units are introduced, which are extra input units representing activation values of the hidden units from the previous time step [Elman 1990].

Thus the Elman RNN is very similar to the Jordan network except that the hidden units, instead of the output units, are fed back.

Minsky and Papert also pointed out that every discrete-time recurrent network can be represented by a feed-forward network with identical behaviour [Minsky *et al* 1969]. The Elman and Jordan RNNs can also be combined to exploit the benefits of both RNNs.



(a) Jordan RNN (b) Elman RNN

Figure 2.2: Typical Recurrent Neural Networks

Neurons can be connected to neurons in the adjacent layers in various ways. In a fully connected network every node in each layer is connected to every node in the adjacent forward layer. A network is partially connected if some of its synaptic connections

are missing from the network. The neurons in a feed-forward neural network (FNN) can also be combined to form higher-order networks. Examples of such higher-order networks are pi-sigma [Ghosh *et al* 1992], sigma-pi [Lee Giles 1987] and functional link networks [Hussain *et al* 1997, Pao 1989].

In the preceding section, the term architecture referred to classification of neural networks, i.e. single-layer FNNs, multilayer FNNs and RNNs. The term topology on the other hand, refers to:

1. the network architecture,
2. the type of neurons, and
3. the connections between these neurons.

This thesis distinguishes between two types of units, i.e. summation and product units. A summation unit (SU) computes the net input signal to a unit as a weighted sum, i.e. $\sum_{i=1}^{I+1} z_i v_{ji}$. A product unit, however, calculates the net input signal as a product of 'terms', where each 'term' comprises an input exponentiated to a weight value, i.e. $\prod_{i=1}^{I} z_i^{v_{ji}}$. In a feed-forward network the flow of signals is in the direction of the outputs, with no feedback loops present. The architecture of a two-layer feed-forward network is illustrated in figure 2.3 where Z, Y and O are respectively the input, hidden and output layers.

The input signals to the network is denoted by z_i $(1 \leq i \leq I)$ where I denotes the total number of input units (excluding the bias unit) to the network. The activation or output of a hidden unit is denoted by y_j $(1 \leq j \leq J)$ where J denotes the number of hidden units (excluding the bias unit). The activation of an output unit is denoted by o_k $(1 \leq k \leq K)$, where K refers to the total number of output units in the network. The weight between input unit Z_i and hidden unit Y_j is denoted by v_{ji}, while w_{kj}

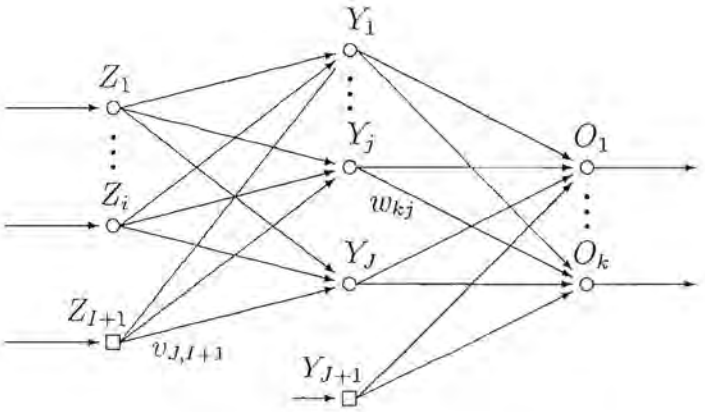


Figure 2.3: Multilayer feed-forward Neural Network

denotes the weight between hidden unit Y_j and output unit O_k. The biases for the hidden and output units are respectively denoted by $v_{j,I+1}$ and $w_{k,J+1}$. The biases in the input and hidden layers have a constant input of -1; the inputs to the bias units are denoted respectively by z_{I+1} and y_{J+1}. The biases are trained in exactly the same way as the other weights. The activation of hidden unit Y_j is calculated as $y_j = f(\sum_{i=1}^{I+1} z_i v_{ji})$, using summation units, or $y_j = f(\prod_{i=1}^{I} z_i^{v_{ji}} + z_{I+1} \cdot v_{j,I+1})$ using product units (if a bias is included for PUs).

The activation functions for the hidden units of summation unit networks should be non-linear, in order to derive any benefit from the multilayer architecture over a single layer network. Rumelhart *et al* have shown that everything that can be computed by a multilayer network, using linear activation functions, can also be computed by an equivalent single layer network [Rumelhart *et al* 1986a]. The standard logistic activation function is assumed for the summation unit neural networks. In this thesis the activation functions for product unit neural networks are assumed to be linear. The activation of unit O_k is calculated as $o_k = f(\sum_{j=1}^{J+1} y_j w_{kj})$ for both the summation and product units. The network in figure 2.3 has only one hidden layer, but networks

can be constructed with any number of hidden layers.

The next section highlights the activation functions that can be used by neural networks.

2.11 Activation Functions

This section introduces the different types of activation functions that can be used with neural networks. An activation function maps the net input signal of a neuron to an output signal. Usually, activation functions are used to limit the amplitude of the output of a neuron. Activation functions are also referred to as squashing functions because of the squashing or limiting effect of most activation functions. All the activations listed below, except the linear activation function are bounded. In this thesis the activation functions will be denoted by $f(\cdot)$. Types of activation functions that are commonly used are given below, where z is the net input of the unit.

- Threshold function

$$f(z) = \begin{cases} 1 & \text{for } z \geq 0 \\ 0 & \text{for } z < 0 \end{cases} \tag{2.1}$$

- Signum function

$$f(z) = \begin{cases} 1 & \text{for } z > 0 \\ 0 & \text{for } z = 0 \\ -1 & \text{for } z < 0 \end{cases} \tag{2.2}$$

- Ramp function

$$f(z) = \begin{cases} y & \text{for } z \geq y \\ z & \text{for } |z| < y \\ -y & \text{for } z \leq -y \end{cases} \tag{2.3}$$

- Sigmoid function

$$f(z) = \frac{1}{1 + e^{-az}} \tag{2.4}$$

where α is the slope parameter of the sigmoid function.

- Hyperbolic tangent function

$$f(z) = \tanh(\tfrac{z}{2}) = \tfrac{1-e^{-z}}{1+e^{-z}} \tag{2.5}$$

- Linear activation function (or identity function)

$$f(z) = z \tag{2.6}$$

The threshold and sigmoid functions produce outputs in the range $[0, 1]$. Engelbrecht *et al* showed that scaling data is not only time-consuming but can also introduce inaccuracies in modelling of the data [Engelbrecht *et al* 1995a]. Also, the maximum and minimum ranges must be known when scaling is performed. These values are difficult to obtain in incremental learning (refer to section 2.14 on page 34) systems, since all training pairs are not available before training. When it is desirable to have activation functions with output in the range -1 to $+1$, the activation function assumes an antisymmetric form with respect to the origin, i.e. $f(-a) = -f(a)$. Engelbrecht *et al* highlighted the benefits of scaling the output to [-1,1] [Engelbrecht *et al* 1995a]. The signum and hyperbolic tangent functions yield values in the range $[-1, 1]$.

Activation functions can further be classified as (a) discrete and (b) continuous activation functions. Examples of discrete activation functions are the threshold, signum and ramp functions. The sigmoid, hyperbolic tangent and the identity function are examples of continuous activation functions. Neural networks with no hidden layers and linear or discrete activation functions can only solve problems that are linearly separable [Aleksander *et al* 1990]. A set of input vectors, $Z = \{\vec{z}_p : p = 1, ..., P\}$ of dimension I is linearly separable if there exists a set of non-zero constants c_i, resulting in a hyperplane, specified below, that separates the set of input vectors into two disjoint

sets,

$$\sum_{i=1}^{l} c_i z_i = 0 \tag{2.7}$$

where z_i is the i^{th} coordinate of the hyperplane.

Linear separability limits the neural network to classification problems where the sets of points, corresponding to input values, can be separated geometrically. Non-linear units have a higher representational power than ordinary linear units. Research has shown that a network with a single hidden layer consisting of a sufficient number of non-linear units can approximate any continuous function [Hornik *et al* 1989a]. Neural networks could handle linearly inseparable functions with the discovery of the back-propagation algorithm [Werbos 1974]. Back-propagation, however, requires that the activation function must be continuous and differentiable to enable weight update calculations.

2.12 Learning Paradigms

This section presents a short overview of early learning paradigms. The four basic training or learning rules, namely error-correction, Hebbian, competitive and stochastic/probabilistic learning rules are discussed in the following sections.

2.12.1 Error-Correction Learning Rule

The aim of the error-correction rule is to minimize a cost function based on an error signal. When a pattern (a vector), $\vec{z}_p$ is presented to the network's input layer, a corresponding output, $o_{k,p}$ is produced by the network at the k^{th} output unit, O_k. Usually, the actual response $o_{k,p}$ of output unit O_k is different from the desired response, $t_{k,p}$. An error signal can now be defined, which is usually the difference between the

target and actual output. The error correction learning rule for a single layer network, assuming linear activation for the output units, is given by,

$$\Delta w_{ki} = \eta \cdot (t_{k,p} - o_{k,p}) \cdot z_{i,p} \tag{2.8}$$

where η is the learning rate and Δw_{ki} is the adjustment to weight, w_{ki}, between input unit Z_i and output unit O_k [Widrow *et al* 1960]. The synaptic weights are subsequently adjusted so that the resulting error signal is minimized. Once a cost function is selected, then error-correction learning can be viewed as an optimization problem. More precisely, the error-correction learning process can be viewed as a 'search' in a multidimensional parameter (weight) space, which gradually optimizes a pre-specified objective (criterion) function [Hassoun 1995]. A criterion commonly used for the cost function is the mean-squared-error criterion, defined as:

$$E = \frac{1}{2PK} \sum_{i=1}^{P} \sum_{k=1}^{K} (t_{p,k} - o_{p,k})^2 \tag{2.9}$$

where P is the total number of training patterns (or observations), K is the number of outputs, $t_{k,p}$ is the target output for the k^{th} output unit for a specific pattern p, and $o_{k,p}$ is the actual output generated by the k^{th} output unit for pattern p. The error is a sum of P errors computed for single patterns. The fraction, $\frac{1}{2}$, is a matter of convenience and simplifies the calculation of the derivative of the error with respect to a weight in back-propagation by gradient descent.

2.12.2 Hebbian Learning

Hebb's postulate of learning is the oldest and most famous of all learning rules, stated as [Rumelhart *et al* 1986a]:

> "When unit A and unit B are simultaneously excited, increase the strength of the connection between them."

An extension to this rule to cover the positive and negative activation values is,

> "Adjust the strength of the connection between units A and B in proportion to the product of their simultaneous activation."

According to Hebb's postulate, the adjustment applied to the synaptic weight w_{ki} that links unit Z_i with unit O_k at time t is expressed by the following function:

$$\Delta w_{ki}(t) = F(o_{k,p}(t), z_{i,p}(t)) \tag{2.10}$$

where F is a function of both the input and the output of unit k. The following is a special case of the above,

$$\Delta w_{ki}(t) = \eta \cdot o_{k,p}(t) \cdot z_{i,p}(t) \tag{2.11}$$

where η is a positive constant that determines the rate of learning. Equation (2.11) is the simplest rule for a change in the synaptic weight w_{ki}. It is sometimes referred to as the activity product rule [Haykin 1994]. The rule states that if the crossproduct of output and input is positive, then weight w_{ki} is increased, otherwise the weight is decreased. It can also be proved that if the set of input patterns used in training are mutually orthogonal, then association can be learned by a two-layer pattern network using Hebbian learning. However, if the set of input patterns are not mutually orthogonal, interference may occur and the network may not be able to learn associations.

The basic Hebbian learning rule in equation (2.11) is fundamentally unstable, since the weights reveal an unlimited growth during the learning process. Stabilization of the Hebbian rule is achieved by Oja's rule, assuming a single output neuron, in (2.12),

$$\Delta w_i(t) = \eta \cdot o(t) \cdot (\vec{z}(t) - o(t) \cdot \vec{w}(t)) \tag{2.12}$$

In Oja's rule the negative term brings in the required stabilization of the learning law [Hassoun 1995].

2.12.3 Competitive Learning

In competitive learning the output units of a neural network compete against each other for the 'right' to represent the input data on a winner takes all basis. In a winner take all circuit, the output unit receiving the largest input is assigned a full value (e.g. 1), whereas all other units are suppressed to a zero value. Therefore, in the case of competitive learning, using a single layer network, only a single output unit is active at any one time, compared to Hebbian learning, where several output units may be active simultaneously.

There are three basic elements to a competitive learning rule [Rumelhart *et al* 1985].

1. A set of units that are the same except for some randomly distributed synaptic weights, which makes each of the units respond differently to a given set of input patterns.

2. A limit is imposed on the strength of each unit.

3. A mechanism that allows the units to compete for the right to respond to a given subset of inputs, such that only *one* output unit is active at a time. All the units that lose the competition are regarded as being inactive.

In competitive learning, individual units learn to specialize on sets of similar patterns and thereby become feature detectors. In order to ensure a fair competition, the sum of all the weights linked to all the output nodes should be normalized. If w_{ki} denotes the synaptic weight connecting input node Z_i to output node O_k, then

$$\sum_{i=1}^{I} w_{ki} = 1, \quad \text{for all } k \tag{2.13}$$

A neuron learns by shifting synaptic weights from its inactive to active input nodes or neurons. If a neuron does not respond to a particular input pattern, no learning takes

place in that neuron. If a particular neuron wins the competition, then each input node of that neuron gives up some proportion of its synaptic weight, and that weight is then distributed equally among the active input nodes. According to the standard competitive learning rule, the change Δw_{ki} applied to synaptic weight w_{ki} is defined by

$$\Delta w_{ki} = \begin{cases} \eta \cdot (z_i - w_{ki}) & \text{if neuron } k \text{ wins the competition} \\ 0 & \text{if neuron } k \text{ loses the competition} \end{cases} \tag{2.14}$$

where η is the learning rate parameter [Zurada 1992]. The effect of the rule is to move the weight vector w_{ki} of winning neuron k towards the input pattern z_i. The number of classes that a competitive network is capable of representing is limited to the number of nodes in the output layer.

2.12.4 Stochastic Learning

Stochastic learning is characterized by an energy function E. In the case of simulated annealing, the energy function is defined as $E(t) = \Sigma_{k=1}^{K} E_k = \Sigma_{p=1}^{P}(t_{p,k}(t) - o_{p,k}(t))^2$. The stochastic learning procedure consists of the following steps [Rojas 1996]:

1. The output value of a hidden layer neuron is changed randomly.

2. The change in energy is evaluated, i.e $\Delta E(t+1) = E(t+1) - E(t)$. If the energy is lower than the energy of the previous state, then the change is accepted, meaning that the current configuration is accepted, otherwise the change is accepted according to a predefined probability distribution. In the case of simulated annealing the change is accepted with a certain probability, given by P,

$$P = e^{(\frac{-(E_{t+1} - E_t)}{KT})} \tag{2.15}$$

where E_t is the energy at time t, T denotes the temperature and K is the Boltzmann constant.

3. Applying the above will eventually result in the network becoming stable, i.e. the network will converge.

4. Steps 1 to 3 are repeated for each input-target pair in the data set. The output is used to statistically adjust the weights.

5. Steps 1 to 4 are repeated until the network performance is adequate as defined by an acceptance criterion. Simulated annealing is terminated when the acceptance ratio drops below a certain preset value, or when the temperature reaches zero. Also, for simulated annealing a cooling schedule defines the rate at which T is reduced, and hence the probability of accepting a new weight vector with a higher energy than current. The most common cooling law uses a geometric decrement function first proposed by Kirkpatrick *et al* [Kirkpatrick *et al* 1983]:

$$T_k = \alpha \cdot T_{k-1} \tag{2.16}$$

where α is a constant usually chosen in the range (0.8, 1.0).

The ability of these networks to probabilistically accept higher energy states, despite poorer performance as reflected by the increase in energy, allows these networks to escape local energy minima in favour of a deeper energy minimum. The Boltzmann machine was the first neural network to employ stochastic learning [Ackley *et al* 1985]. This technique was also applied in simulated annealing where a temperature parameter slowly decreases the number of probabilistically accepted higher energy states [Kirkpatrick *et al* 1983].

The following section discusses the three classes of learning paradigms.

2.13 Learning Paradigms

There are basically three learning paradigms, namely supervised, unsupervised and reinforcement learning. These paradigms are discussed in this section.

2.13.1 Supervised Learning

In supervised learning, a supervisor (or teacher) provides the network with an input pattern and the associated target, or desired response. The difference between the actual output of the network and the target output serves as an error measure and is used in correcting synaptic weights. The weights are adjusted gradually, by updating them at each step of the learning process so that the error between the network's output and corresponding desired output is reduced. This adjustment is carried out iteratively in a step-by-step fashion with the aim of eventually making the neural network emulate the teacher. Since adjustable weights are assumed, the teacher may implement a reward-and-punishment scheme to adapt the network's weights [Zurada 1992]. This type of learning is also known as reinforcement learning. Supervised learning rewards accurate classifications or associations and punishes those that yield inaccurate responses. The reward or punishment is based on the teacher's estimate of the negative error gradient direction. An example of supervised learning is error-correction learning, of which gradient-descent by back-propagation is an example.

2.13.2 Unsupervised Learning

Unsupervised learning, also referred to as self-organization, requires no target or desired outputs and relies only upon local information during the entire learning process. Error information cannot be used to improve network behaviour, since the desired response is not known. With no information being available as to the correctness or incorrectness of responses, learning must somehow be accomplished based on observations of responses

to inputs that the neural network has little or no knowledge about. The task of unsupervised learning is to learn to group together patterns that are similar of a given training set. In this mode of learning, the network must discover for itself any possibly existing patterns, regularities, separating properties, etc. [Kohonen 1988b]. While discovering these, the network undergoes change of its parameters, which is referred to as self-organization. Examples of unsupervised learning are Kohonen's self-organizing feature maps and Hebbian learning [Kohonen 1988b].

2.13.3 Reinforcement Learning

Reinforcement learning is similar to error correction learning in that weights are reinforced for properly performed actions and punished for poorly performed actions. The difference between the two types of learning is that error correction learning utilizes more specific error information by using the error values at each output unit, while reinforcement learning uses non specific error information to determine the performance of the network. In error correction learning an entire vector of values is used for error correction, whereas only one value is used to describe the output layer's performance during reinforcement learning. This form of learning is ideal in areas such as prediction and control where specific error information is not available, but overall performance is [Barto 1992].

This thesis concentrates on supervised learning.

2.14 Modes of Learning

Mode of learning refers to the type of weight adjustment implemented during training. Weights can be updated in two ways, namely batch and on-line modes.

- **Batch learning**

  In batch, or off-line learning, weight changes are done only after the entire training set has been presented to the network. Weight changes for each presented pattern are therefor accumulated and updated after each epoch. An epoch is one complete presentation of the entire training set during the training process. In off-line learning, once the network has been trained and enters recall mode (i.e. when the network is in operation) the weights are fixed and not modified at all. All the patterns must be resident for training in off-line training systems with the result that new patterns cannot automatically be incorporated into the system as they occur. To include new training patterns, it must be added to the entire training set and the network must be re-trained. Off-line training provides a more accurate estimate of the gradient vector even though it requires more storage space than on-line training [Haykin 1994].

- **On-line learning**

  In on-line, or incremental learning the weights are adjusted after each pattern is presented to the network. Once the network has been trained and enters recall mode, the weights are fixed and not modified at all. The advantage of on-line learning is that it requires less storage space than batch training.

This thesis assumes on-line learning.

The next section discusses the different performance measures used in training neural networks.

2.15 Performance Measures

This section discusses the various training errors that are used in training neural networks. All supervised training algorithms involve the reduction of an error value. When weights are adjusted in a single training step, the error to be reduced is usually computed for a single pattern presented at the input layer. However, the prediction error of the neural network must be computed using the entire set of training patterns in order to assess the quality and success of the training process [Zurada 1992].

2.15.1 True Error versus Empirical Error

During training of a NN a finite set of input-target pairs $D = \{d_p = (\vec{z}_p, \vec{t}_p) \mid p = 1, ..., P\}$, sampled from a stationary density $\Omega(D)$, is used where $z_{i,p}$ is the value of input unit Z_i and $t_{k,p}$ is the target value of output unit O_k for pattern p. The target can be expressed as a function of the input vector, i.e.

$$\vec{t}_p = \mu(\vec{z}_p) + \vec{\zeta}_p \tag{2.17}$$

where $\mu(\vec{z})$ is the unknown function approximated by the network. The objective of learning is then to approximate the unknown function using the information contained in the finite data set D. Since prior knowledge about $\Omega(D)$ is usually not known, a non-parametric regression approach is used by the NN learner to search through its hypothesis space $\mathcal{H}$ for a function $\mathcal{F}_{NN}(D, W)$ which gives a good estimation of the unknown function $\mu(\vec{z})$, where $\mathcal{F}_{NN}(D, W) \in \mathcal{H}$. In the case of multilayer NNs, the hypothesis space consists of all functions realizable from the given network architecture as described by the weight vector W.

The function $\mathcal{F}_{NN} : R^I \rightarrow R^K$ is found which minimizes the empirical error,

$$\mathcal{E}_T(D, W) = \frac{1}{P_T} \sum_{p=1}^{P_T} = ((\mathcal{F}_{NN}(\vec{z}_p, W) - \vec{t}\,)^2 \tag{2.18}$$

where P_T is the total number of training patterns. Hopefully, a small empirical error will also yield a small true error, defined as

$$\mathcal{E}_G(\Omega, W) = \int (\mathcal{F}_{NN}(\vec{z}, W) - \vec{t}\,)^2 \, \Omega(\vec{z}, \vec{t}\,) \tag{2.19}$$

The empirical error in equation (2.18) is usually referred to as the objective function. Prediction errors are mainly defined as:

- **Sum-squared-error (SSE)**

  The sum-squared-error is computed over the entire training cycle and is expressed as a quadratic error,

  $$\mathcal{E}_{SSE} = \frac{1}{2} \sum_{p=1}^{P} \sum_{k=1}^{K} (t_{k,p} - o_{k,p})^2 \tag{2.20}$$

  where P is the total number of training patterns (or observations), K is the number of outputs, $t_{k,p}$ is the target output for the k^{th} output unit for a specific pattern p, and $o_{k,p}$ is the actual output generated by the k^{th} output unit for pattern p. The error above reflects the accuracy of the neural network mapping after a number of training cycles have been completed. The SSE is not very useful when comparing networks with different numbers of training patterns and having a different number of output units. If a large training set is used to train different networks that contain the same number of output units, a large SSE will be produced due to the large number of terms in the summation, while a smaller training set will produce a smaller SSE. Similarly, networks with a large number of output units trained using the same training set would usually also produce large SSE errors.

- **Root-mean-squared error (RMS)**

  $$\mathcal{E}_{RMS} = \frac{1}{PK} \sqrt{\sum_{p=1}^{P} \sum_{k=1}^{K} (t_{k,p} - o_{k,p})^2} \tag{2.21}$$

The value has the sense of a root-mean squared normalized error, and is more descriptive than, $\mathcal{E}_{SSE}$, when comparing the outcome of the training of different neural networks among each other [Zurada 1992].

- **Mean-squared-error (MSE)**

$$\mathcal{E}_{MSE} = \frac{1}{2PK} \sum_{p=1}^{P} \sum_{k=1}^{K} (t_{k,p} - o_{k,p})^2 \tag{2.22}$$

A more adequate error measure is given by the root-mean-squared error and the mean-squared-error, since they have no bias towards networks with fewer output units, or networks trained on fewer patterns.

The following section discusses the minimum number of hidden layers that are required to approximate continuous functions using feed-forward neural networks.

2.16 Approximation Capabilities of Feed-Forward Neural Networks

Cybenko proved that a feed-forward neural network with 1 hidden layer and a sufficient number of hidden units, of the sigmoidal activation type, and a single linear output unit is capable of approximating any continuous function, $\{f : \mathcal{R}^n \rightarrow \mathcal{R}\}$ to any desired accuracy [Cybenko 1969]. Rigorous mathematical proofs for the universality of feed-forward layered neural networks employing continuous sigmoid activation functions as well as other *more general activation* functions were also given independently by Funahashi [Funahashi 1989] and Hornik *et al* [Hornik *et al* 1989b]. The universality of single-hidden-layer nets with units having non-sigmoidal activation functions was formally proved by Stinchcombe and White [Stinchcombe *et al* 1989]. Baldi showed that a large class of continuous multivariate functions can be approximated by a weighted sum

of bell-shaped functions, referred to as multivariate Bernstein polynomials [Baldi 1991]. Baldi also proved that a single-hidden-layer network with bell shaped activation functions in the hidden layer and a single linear output unit is a possible approximator of functions $f : \mathcal{R}^n \rightarrow \mathcal{R}$. Similarly, Hornik proved that a sufficient condition for universal approximation can be obtained by using *continuous, bounded, and non-constant* hidden unit activation functions [Hornik *et al* 1989b]. Li *et al* proved that higher-order neural networks can approximate any continuous function on a compact set with an arbitrary degree of accuracy, provided that the activation function belongs to the complex domain [Li *et al* 1996]. A single-hidden-layer neural network would thus be adequate to approximate the continuous functions in this thesis, provided that a sufficient number of hidden units are included.

2.16.1 Generalization

The objective of back-propagation is to train a network, using as many patterns as possible, that will subsequently produce correct (or nearly correct) output for input patterns that were not presented to the network during training. For a given input-target pair, $(\vec{z}_p, \vec{t}_p)$, the output $\vec{o}_p$, produced by the trained network when presented with $\vec{z}_p$ as input, is correct if $\|\vec{t}_p - \vec{o}_p\| = 0$, or nearly correct if $\|\vec{t}_p - \vec{o}_p\| \leq \epsilon$, where $\epsilon > 0$ is an arbitrary small number. A network that does achieve the preceding objective, is said to generalize well. Although enough information is crucial to effective learning, too large training set sizes may also be of disadvantage to generalization performance and training time [Engelbrecht *et al* 1999d, Lange *et al* 1996, Zhang 1994]. The learning process may be visualized as a "curvefitting" problem, where the network itself may be considered as a nonlinear input-desired-output mapping [Haykin 1994]. This viewpoint allows generalization of neural networks to be looked at as the effect of a good nonlinear interpolation of the input data [Wieland *et al* 1987]. The network

performs useful interpolation simply because multilayer perceptrons with continuous activation functions lead to output functions that are also continuous [Haykin 1994]. A neural network that generalizes well will produce a correct input-output mapping even in cases where the input is slightly different from the patterns of the training set. A network is said to be overtrained if too many weights were used in training the network, resulting in the network to accurately memorize the training data, but not generalizing well on similar input-output patterns. Generalization is influenced by three factors:

1. The size and relevance of the training set.
2. The architecture of the network.
3. The complexity of the problem to be solved.

Hush and Horne viewed the problem of generalization from two different perspectives regarding the first two factors [Hush *et al* 1993],

- by fixing the architecture of the network and then to determine the size of the training set needed for good generalization, or
- by fixing the size of the training set and then determine the best architecture that results in good generalization.

2.17 Architecture Selection

One of the most important problems encountered in the practical application of neural networks is to find a suitable, or ideally minimal, neural network topology that accurately maps the true function described by the training data. An unsuitable topology increases the training time or even causes non-convergence, and is likely to decrease the generalization capability of a network [Ghosh *et al* 1994]. An oversized network (too many training units) can lead to overfitting, while a network with too few

training units can lead to underfitting [Baum *et al* 1989, Le Cun 1989]. Overfitting occurs when the network 'memorizes' the training patterns, including all of their peculiarities resulting in a network that does not generalize well. In both over and underfitting the network fails to approximate the true mapping between the inputs and desired outputs. Architecture selection has to reduce network complexity while maintaining good generalization. The objective of training is that the network should only learn the general properties of the examples.

Architecture selection approaches are grouped into the following four classes.

1. **Brute Force Pruning**

   Successively smaller networks are trained until the smallest network with the best generalization is found. This approach is time-consuming and prohibitive for large networks, since the search space explodes as the weights are increased [Moody *et al* 1996].

2. **Network Growing**

   With network growing, a small network configuration is used initially, and new neurons are added only when the performance is unsatisfactory. Network growing algorithms start training with a small network and incrementally add hidden units during training when the network is trapped in a local minimum [Hirose *et al* 1991, Kwok *et al* 1995, Zhang *et al* 1997]. This process of adding units is stopped when a satisfactory performance of the network is attained. Examples of the network growing approach are the cascade-correlation learning architecture developed by Fahlman and Lebiere [Fahlman *et al* 1990], the upstart algorithm of Frean [Frean 1990] and the pocket algorithm developed by Gallant [Gallant 1986].

3. Network Pruning

With network pruning, training commences with an oversized network that yields an adequate performance for the problem under consideration, but possibly overfits the training data. The network is pruned by removing redundant or excess parameters, i.e. weights, hidden and input units, in a selective and orderly process to produce smaller networks [Le Cun *et al* 1990, Sietsma *et al* 1988]. Small networks are usually faster and generalize better than large networks [Reed 1994]. The aim of pruning is therefor to solve the problem of overfitting and to reduce the computational cost of training and using the network [Le Cun *et al* 1990]. The various pruning algorithms use different criteria to identify irrelevant parameters that must be removed. The decision to prune a network parameter is based on some measure of parameter relevance or significance. A relevance is computed for each parameter and a pruning heuristic is used to decide when a parameter is pruned or not.

Optimal Brain Damage (OBD), developed by Le Cun *et al* [Le Cun *et al* 1990], uses the criterion of minimal increase in training error for weight elimination. OBD can only prune network weights. The goal of OBD is to find a set of weights that, when deleted, would cause the least increase in the training error. Le Cun *et al* defined the saliency of a parameter as the change in the error caused by deleting that set of weights. A strategy was employed to delete weights with low saliency [Le Cun *et al* 1990]. The saliency when the weight vector, W, is perturbed is computed as follows,

$$\delta E = \Sigma_i g_i \delta w_i + \frac{1}{2}\Sigma_i h_{ii} \delta w_i^2 + \frac{1}{2}\Sigma h_{i,j} \delta w_i \delta w_j + O(\|\delta W\|^2) \qquad (2.23)$$

where the δw_i's are the components of δW, g_i are the components of the gradient of E with respect to W, i.e. $g_i = \frac{\partial E}{\partial w_i}$ and the h_{ij} are the elements of the Hessian

matrix (H). Second order derivatives of the error with respect to the weights, which are computationally complex due to the size of the Hessian matrix (H), where each $h_{ij} = \frac{\partial^2 E}{\partial w_i \partial w_j}$, are required for the computation of the saliency. In OBD, pruning is done on a well trained network, hence the first term in equation (2.23) will be approximately zero, since E is at a minimum. Also, for small perturbations of the weights the last term will be negligible. For computational simplicity, OBD assumes that the off-diagonal elements of the large Hessian matrix are zero; thus the third term evaluates to zero. Equation (2.23) then simplifies to,

$$\delta E \approx \frac{1}{2}\Sigma_i h_{ii} \delta w_i^2 \tag{2.24}$$

An efficient way of evaluating the diagonal second order derivatives h_{ii} was derived using a fast back-propagation method. The saliency of weight w_i is then,

$$s_i = \frac{1}{2} h_{ii} w_i^2 \tag{2.25}$$

A drawback of OBD is that it does only prune network weights and not units. However, if all the weights leading to, or emanating from a unit are pruned, that unit can be pruned also.

Optimal Cell Damage (OCD) was developed to extend OBD to allow pruning of input and hidden units [Cibas *et al* 1996].

Hassibi and Stork have discovered that Hessian matrices, for the problems that they considered, were all strongly non-diagonal, resulting in OBD to eliminate the wrong weights [Hassibi *et al* 1994]. Optical Brain Surgeon (OBS) was developed by Hassibi and Stork as an extension of OBD to remove the restrictive assumption about the (diagonal) form of the Hessian. The typical slow retraining

by back-propagation of the network after pruning required by OBD was also not required in OBS, since OBS not only removed the irrelevant weights, but also adjusted the remaining weights automatically to minimize the error. OBS, like OBD prune network weights, but the same technique can be applied to prune network units. Disadvantages of OBS are, (a) OBS is computational intensive due to the calculation of the large Hessian matrix and (b) it also requires large storage space for intermediate results.

Skeletonization, developed by Mozer and Smolensky, defined a measure of the relevance of a unit as the error when the unit is removed from the network minus the error when the unit is left in the network [Mozer *et al* 1989]. The least relevant units can then be removed to construct a skeleton version of the network. The usual sum of squared errors was used for training, however, since the quadratic error provided a poor estimate of relevance if the output pattern is close to the target, a linear error function, i.e. $E = \Sigma|t_{k,p} - o_{k,p}|$, was used to measure relevance. Skeletonization pruned network units only, but it can also be applied to prune network synaptic weights [Mozer *et al* 1989].

Zurada *et al* developed a sensitivity analysis tool which can be applied to a trained neural network in order to automatically identify all input parameters which have a significant influence on any one of the possible outcomes [Zurada *et al* 1997]. Sensitivity analysis thus provides a tool to automatically identify all relevant input parameters from a set of potential parameters. The irrelevant parameters can then be pruned using the significance measures obtained from the sensitivity analysis tool.

Engelbrecht *et al* developed a pruning algorithm where the sensitivity of the output of the network to small changes to the parameters is used to identify irrelevant parameters [Engelbrecht *et al* 1999b, Engelbrecht 2001], compared to OBD where the sensitivity of the objective function is used. Engelbrecht's algorithm prunes both input and hidden units, and can be adapted to prune weights also. Engelbrecht also developed a computationally efficient pruning heuristic based on variance analysis of sensitivity information [Engelbrecht *et al* 1999c, Engelbrecht 2001]. This algorithm utilizes first-order derivatives, which are already calculated during training. Thus Engelbrecht's algorithm is not as computational intensive as OBD and OBS. The only assumptions are that the network must be well trained and that the activation function must at least be once differentiable.

4. **Complexity Regularization**

In regularization a penalty term is added to the objective function to penalize all the weights. This augmented function then serves as the objective function to be minimized [Poggio *et al* 1990, Weigend *et al* 1991]. The objective function is expressed as

$$\xi = \xi_T + \lambda \xi_C \tag{2.26}$$

where ξ_T is the standard performance measure and ξ_C is the complexity term [Girosi *et al* 1995, Shittenkopf *et al* 1997, Weigend *et al* 1991]. The regularization parameter λ controls the influence of the penalty term. If λ is zero, then the penalty term will have no effect. A too large λ will drive all weights to zero. Regularization requires a delicate balance between the normal error term and the complexity term. In complexity regularization the redundant synaptic weights are forced to take on values close to zero, while permitting other weights to retain

their relatively large values. This improves generalization of the resulting network. Examples of regularization are weight-decay [Hinton 1987] and the weight-elimination procedures [Weigend *et al* 1991]. A disadvantage of regularization is that the complexity terms tend to create additional local minima, thus increasing the possibility of converging to bad local minima [Hanson *et al* 1989]. Training time is also increased due to the extra calculations required during updating of the weights.

2.18 Back-propagation

Back-propagation, also referred to as backprop, is probably the most widely applied neural network learning algorithm. Backprop's popularity is related to its ability to deal with complex multi-dimensional mappings. The feed-forward, back-propagation architecture was discovered independently in the early 1970's by Werbos and Bryson [Bryson *et al* 1969, Werbos 1974]. It was re-discovered and popularized by Rumelhart in the 1980's [Rumelhart *et al* 1986b]. A generalization of the back-propagation algorithm was derived by Parker in 1985 [Parker 1985]. Its greatest strength is in finding non-linear solutions to ill-defined problems [Haykin 1994]. Although the back-propagation algorithm did not provide a solution for all solvable problems it has put to rest the pessimism about learning in multilayer networks that may have been inferred from the book by Minsky and Papert [Minsky *et al* 1969]. Back-propagation provides a computationally efficient method for changing the weights in a feed-forward network, with differentiable activation function units, to learn a training set of input and desired-output examples. Back-propagation multilayer neural nets have been applied successfully to solve some difficult and diverse problems such as speech recognition [Cohen *et al* 1993], handwritten character recognition [Guyon 1990], steering of an autonomous vehicle [Pomerlau 1989], medical diagnosis of heart attacks

[Harrison *et al* 1991], radar target detection and classification [Haykin *et al* 1992], and many more.

The next section discusses the back-propagation algorithm.

2.18.1 Overview of Back-propagation

The discussion of back-propagation assumes that the multilayer network in figure 2.3 on page 23, consisting of an input, a hidden and an output layer, is fully connected, which means that a neuron in the second or third layer of the network is connected to all neurons in the previous layer. Back-propagation uses gradient descent as optimization algorithm. The process of back-propagation consists of two distinct phases, namely, (a) the forward phase and (b) the backward propagation phase.

- Phase 1: Forward Phase

  During the forward phase, a pattern, p, presented at the input layer of the network results in signals to be propagated through to the hidden units. An activation signal is computed for each hidden unit and then propagated through to the next layer, which is either another hidden layer or the output layer. Eventually, the activation of the output units are calculated. The output layer provides the response of the network for a given input pattern, p. The actual output for pattern p at output unit O_k is denoted by $o_{k,p}$ and the desired output of O_k is denoted by $t_{k,p}$. The error signal for each output unit, O_k, for a given pattern, p, is computed as the difference between the desired and the actual output, i.e. $t_{k,p} - o_{k,p}$.

- Phase 2: Backward Propagation Phase

  In the backward pass, which starts at the output layer, the error computed in the forward pass is propagated backwards through the network, layer by layer, and the δ, i.e. the local error or gradient, for each neuron is computed recursively. For

a neuron in the output layer, the local error is simply equal to the error signal of this neuron, $t_{k,p} - o_{k,p}$, multiplied by the first derivative of the output of this neuron with respect to the neuron's net input. For a neuron in the hidden layer, the local error equals the product of the associated derivative $f'(net_{y_j})$ and the weighted sum of the error signals (i.e. δ's) computed for the neurons in the output layer that are connected to neuron Y_j. The objective of the learning process is to adjust the weights of the network so as to minimize the error, $E = \sum_{p=1}^{P} E_p$ and $E_p = \frac{1}{2}\sum_{k=1}^{K}(t_{k,p} - o_{k,p})^2$, where p refers to a specific pattern, and k refers to the k^{th} component of the output vector. For notational convenience, the subscript p is dropped from subsequent equations. The adjustment of weights are computed as follows,

$$\Delta v_{ji} = -\eta \frac{\partial E}{\partial v_{ji}} \tag{2.27}$$

$$\Delta w_{kj} = -\eta \frac{\partial E}{\partial w_{kj}} \tag{2.28}$$

where η is a constant that determines the rate of learning, v_{ji} is the weight between input unit Z_i and hidden unit Y_j and w_{kj} is the weight between hidden unit Y_j and output unit O_k.

The learning rate has a profound impact on the convergence of the back-propagation algorithm, as is discussed in the following section.

The weights can be updated using on-line or off-line modes of learning. In the on-line or incremental update mode the weights are updated after the presentation of a single pattern to the network. In the off-line or batch mode, weight updating is performed after all the training examples that constitute an epoch have been

presented to the network. Finnoff showed that for "very small" learning rates, on-line back-propagation approaches batch back-propagation, producing essentially the same results [Finnoff 1993a].

The on-line mode is preferred over the batch mode for the following two reasons:

1. On-line training requires less storage, and
2. With on-line training, the patterns are presented in a random manner, thus making the search in weight space stochastic in nature, which in turn makes it less likely for back-propagation to be trapped in a local minimum.

2.18.2 The Effect of the Learning Rate

The effectiveness and convergence of back-propagation training depend significantly on the learning rate. A good initial learning rate can speed up the training of a neural network. A small learning rate will result in slow convergence due to the large number of update steps required to reach a local minimum. Thus, the smaller the learning rate, the smaller will the changes to the synaptic weights in the network be from one iteration to the next. If the learning rate parameter is too large, the resulting large changes in the weights cause the network to produce oscillations between relatively poor solutions, or it may jump over the global minimum and end in a weaker local minimum. It is desirable to have large steps when the search point is far from a minimum, which are decreased as the search approaches a minimum. For small constant learning rates there is a nonneglible stochastic element in the training process that allows the search to escape local minima with shallow basins of attraction [Hassoun 1995]. The danger of a learning rate that is too small may still cause the search to be trapped in local minima.

Many heuristics have been proposed so as to adapt the learning rate automatically. Sutton presented a method that increases or decreases the learning rate for each weight w_i according to the number of sign changes observed in the associated partial derivative $\frac{\partial E}{\partial w_i}$ [Sutton 1986]. Franzini investigated a technique that heuristically adjusts the learning rate, increasing it whenever $\nabla E(t)$ is close to $\nabla E(t-1)$ and decreasing it otherwise [Franzini 1987]. Chan and Fallside proposed an adaptation rule for the learning rate that is based on the cosine of the angle between the gradient vectors $\nabla E(t)$ - $\nabla E(t-1)$ [Chan *et al* 1987]. Silva and Almeida used a method where the learning rate parameter for a given weight w_i is multiplied by factor a, where $a > 1$, if $\frac{\partial E(t)}{\partial w_i}$ and $\frac{\partial E(t-1)}{\partial w_i}$ have the same sign; if the partial derivatives have different signs, then the learning rate parameter is multiplied by b, where $0 < b < 1$ [Silva *et al* 1990]. The disadvantage of Silva and Almeida's method is that it introduced two extra parameters. Moreira also employed adaptive learning rates and showed that the adaptive learning rates can compensate for a bad initial value [Moreira *et al* 1995]. Haffner *et al* propose a learning rate $\eta = e^{-4 \cdot log(s)+c}$ for a sigmoid activation function of the form $f(net_i) = \frac{s}{1+e^{-net_i}}$. Unfortunately, they do not compare their approach to others, neither give details (the constant c is not precisely given) [Haffner *et al* 1988].

The local minima problem can be eased by adding noise to the weights [Von Lehman *et al* 1988] or by adding noise to the input patterns [Sietsma *et al* 1988]. Convergence in back-propagation can also be increased by using a momentum term, which is discussed in the following section.

2.18.3 The Effect of Momentum on Back-propagation

A momentum term is used to stabilize the weight change by making nonradical revisions using a combination of the gradient decreasing term with a fraction of the

previous weight change. A momentum term was first introduced by Rumelhart *et al* [Rumelhart *et al* 1986b], where weight changes are calculated as

$$\Delta v_{ji}(t) = \eta \cdot \delta_{y_j}(t) \cdot z_i(t) + \alpha \cdot \Delta v_{ji}(t-1) \tag{2.29}$$

where the momentum constant, α, is restricted to the range $0 \leq \alpha < 1$. The effect of α on $\Delta v_{ji}(t)$ in equation (2.29) is described below:

- When α is zero, the back-propagation algorithm operates without momentum.
- When $\frac{\partial E}{\partial v_{ji}}$ has the same algebraic sign on consecutive iterations, then the adjustment Δv_{ji} grows in magnitude, and the weight is adjusted by a large amount. Thus, the inclusion of the momentum term tends to accelerate descent in steady downhill directions, instead of fluctuating with every change in the sign of the associated partial derivative, $\frac{\partial E}{\partial v_{ji}}$.
- When $\frac{\partial E}{\partial v_{ji}}$ has opposite algebraic signs on consecutive iterations, then the adjustment Δv_{ji} shrinks in magnitude, resulting in the weight being adjusted by a small amount. Thus, the effect of the momentum term has a stabilizing effect in directions that oscillate in sign.

Adaptive momentum rates may also be employed. Fahlman proposed, and extensively simulated, a heuristic variation of backprop, called quickprop, that employs a dynamic momentum rate given by [Fahlman 1989]:

$$\alpha(t) = \frac{\frac{\partial E}{\partial w_i(t)}}{\frac{\partial E}{\partial w_i(t-1)} + \frac{\partial E}{\partial w_i(t)}} \tag{2.30}$$

With this adaptive $\alpha(t)$ substituted in equation (2.29), if the current slope is persistently smaller than the previous one but has the same sign, then $\alpha(t)$ is positive, and the weight change will accelerate. Thus the acceleration rate is determined by magnitude of successive differences between slope values. If the current slope is in the

opposite direction from the previous one, it signals that the weights are crossing over a minimum. In this case $\alpha(t)$ has a negative sign, and the weight change starts to decelerate.

The momentum term may also have the benefit of preventing the learning process from terminating in a shallow local minimum on the error surface [Haykin 1994]. The net effect of momentum is that of traversing flat error surfaces quickly, while moving slower when the surface becomes irregular.

The next section discusses the on-line implementation of back-propagation applied to SUNN.

2.18.4 On-line Implementation of Back-propagation

1. **Initialization**

   Choose a reasonable network configuration and set all weights including biases to small random numbers.

   For each pattern in the set, perform processes listed in 2 and 3 below.

2. **Forward Phase**

   The input vector $\vec{z}$, is presented to the input layer of the network, and the target vector, $\vec{t}$, to the output layer of the network. The activation values are then computed for the hidden and output units, respectively. The activation value for a summation hidden neuron is calculated as,

$$y_j = f(\sum_{i=1}^{I+1} v_{ji} z_i) \tag{2.31}$$

while the activation value for the k^{th} neuron of the output layer is computed as,

$$o_k = f(\sum_{j=1}^{J+1} w_{kj} y_j) \tag{2.32}$$

The error signal, i.e. the difference between the desired response t_k and the networks output o_k, is subsequently computed:

$$e_k = t_k - o_k \tag{2.33}$$

3. **Backward propagation phase**

The local gradients (or errors) of the network, i.e. δ's, are computed by proceeding backward layer-by-layer. For a neuron in the outer layer, δ_{o_k} is computed using,

$$\delta_{o_k} = e_k \cdot f'(net_{o_k}) \tag{2.34}$$

For a neuron in the hidden layer, δ_{y_j} is computed using,

$$\delta_{y_j} = f'(net_{y_j}) \cdot \sum_{k=1}^{K} \delta_{o_k} \cdot w_{kj} \tag{2.35}$$

Subsequently, the weights in the output layer are adjusted with,

$$\Delta w_{kj}(t) = \alpha \Delta w_{kj}(t-1) + \eta \cdot \delta_{o_k}(t) \cdot y_j(t) \tag{2.36}$$

and the weights in the hidden layer are adjusted with,

$$\Delta v_{ji}(t) = \alpha \Delta v_{ji}(t-1) + \eta \cdot \delta_{y_j}(t) \cdot z_i(t) \tag{2.37}$$

4. **Iteration**

Repeat the process listed in 2 to 4 by presenting all the patterns in the training set repetitively until the weights of the network stabilize their values and the average error computed over the entire training set is acceptable.

The next section discusses the stopping criteria for the back-propagation algorithm.

2.18.5 Terminating criteria

In general, it cannot be shown that the back-propagation algorithm converges, nor are there well defined criteria for stopping its operation. However, reasonable criteria do exist, each with its own practical merit, which may be used to teminate the back-propagation algorithm [Haykin 1994].

The back-propagation algorithm is considered to have converged, when any of the following becomes true:

1. When the Euclidean norm of the gradient vector reaches a sufficiently small gradient threshold [Kramer *et al* 1989].

2. When the absolute rate of change in the average squared error per epoch is sufficiently small.

3. If the maximum value of the average squared error on the test set is equal to or less than a sufficiently small threshold.

4. When the generalization performance, tested after each learning iteration, is adequate, or when it is clear that the generalization performance has peaked.

5. When the network starts to overfit, i.e. when

$$\xi_V > \overline{\xi}_V + \delta_{\xi_V} \tag{2.38}$$

where ξ_V is the current error on the validation set and $\overline{\xi}_V$ is the average error on the validation set over the previous iterations and δ_{ξ_V} is the standard deviation in validation error.

2.18.6 Initialization

The first step in the back-propagation algorithm is the initialization of the synaptic weights. Owing to its gradient-descent nature, back-propagation is very sensitive

to initial conditions. If the choice of the initial weight vector is located within the attraction basin of a strong local minima attractor, convergence of back-propagation will be fast. On the other hand, back-propagation converges very slowly if the initial weights start the search in a relatively flat region of the error surface.

A good choice for the initial weights can be of a tremendous help in a successful network design. The random weight initialization method is often preferred for its simplicity and ability to produce multiple solutions, as the weights may, due to their initial randomness, converge to various attractors [Kolen *et al* 1990]. In practice all the weights are set to random numbers that are uniformly distributed inside a small range of values [Rumelhart *et al* 1986b]. Rumelhart, Hinton and Williams discovered that if all weights start out with *equal values,* where the solution requires that unequal weights be developed, the network does not learn [Rumelhart *et al* 1986b].

Premature saturation occurs when the error value remains almost constant for some period of time during the learning process. This point in the error surface cannot be considered as a local minimum, because the squared error continues to decrease on subsequent iterations. Premature saturation corresponds to a saddle point in the error surface. Large weights tend to prematurely saturate units in a network and render them insensitive to the learning process [Hush *et al* 1991, Lee *et al* 1991]. Wessels and Barnard describe two initialization methods [Wessels *et al* 1992]. The first method sets the initial weight range to a value which assumes that the output of the network and the target patterns have the same variance. The second method puts equally distributed decision boundaries in the input space which produces initial weights for the first layer of connections. The weights of the second layer are set to 1.0. A comparison of generalization on both methods was done, on three sets of data. Wessels and Barnard found that the second method outperformed the first in terms

of generalization. However, convergence speeds were not compared [Wessels *et al* 1992].

2.19 Conclusion

This chapter provided an overview of summation unit neural networks and gradient descent applied to SUNNs (the so-called back-propagation networks). Various network architectures, learning paradigms, learning rules and modes of learning were discussed. The back-propagation neural network which uses gradient descent was introduced and explained. The effect of weight initialization, momentum and the learning rate on convergence of back-propagation was addressed in this chapter. The next chapter discusses higher-order neural networks, where the training of product unit neural networks is discussed in detail.

Chapter 3

Higher-Order Neural Networks

Higher-order neural networks are networks that utilize higher combinations of its inputs. A goal of this thesis is to train PUNNs, which are examples of higher-order neural networks. In this context, this section provides an overview of higher-order neural networks. This chapter discusses four types of higher-order neural networks and the problems associated with the training of product unit neural networks specifically.

3.1 Sigma-Pi Networks

Hidden units of a sigma-pi neural network calculate a product (or conjunct) of the inputs [Lee Giles 1987, Maxwell *et al* 1986]. In sigma-pi neural networks a weight is applied, not only to each input, but also to the second and possibly higher-order products or conjuncts of the inputs. The connections in sigma-pi neural networks allow one unit to gate another: Thus, if one unit of a multiplicative pair of units is zero, then the other member can have no effect on the output. On the other hand, if one unit of a pair has a value 1, the output of the other unit is passed unchanged to the receiving unit. In this way a polynomial function of the inputs is presented as input to the transfer function of the output layer, i.e. the value of the output unit O_k is

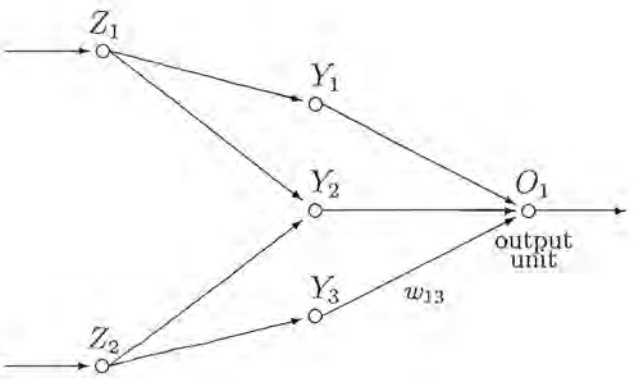


Figure 3.1: Sigma-pi network

$$o_k = f\left(\sum_{q \in conjunct} w_{qk} \prod_{k=1}^{N} z_{qk}\right)$$

where f is the activation function, w_{qk} a synaptic weight, $z_{q1}, z_{q2}, ..., z_{qN}$ are the N input signals combined to form the product or conjunct, and q indexes the conjuncts or products that are used in unit k; conjunct is the set of all conjuncts of subscripts for the inputs. The architecture derived from the above function presents a method of constructing higher-order networks.

Figure 3.1 illustrates a sigma-pi network with two inputs, where multiplication instead of summation is performed in the hidden layer, followed by a summation unit in the output layer. That is, for example, $y_2 = z_1 z_2$ and $y_1 = z_1$, where y_j is the output of hidden unit Y_j. The weight between hidden unit Y_j and output unit O_k is denoted by w_{kj}. In sigma-pi networks a polynomial function of the inputs is presented to the activation function of the output layer. For the example in figure 3.1,

$$\begin{aligned} o_1 &= f(w_{11}y_1 + w_{12}y_2 + w_{13}y_3) \\ &= f(w_{11}z_1 + w_{12}z_1z_2 + w_{13}z_3) \end{aligned} \tag{3.1}$$

Although the terms contain products of inputs there are no powers of each input greater than one, this gives rise to the name *multi-linear* for the terms in this kind of expression. Nodes with multi-linear terms are also called higher-order nodes, since their activation depends on terms whose multiplicative order is greater than one. The problem with sigma-pi units is that the number of terms, and therefore the weights, increase very rapidly with the number of inputs, thus becoming unacceptably large for use in many situations [Durbin *et al* 1989, Lee Giles 1987]. Thus a disadvantage of this type of architecture (refer to figure 3.1) is that a combinatorial explosion in the number of weights may result if conjuncts are not hand coded [Lee Giles 1987]. Researchers combat this problem by restricting the number of units, i.e. the number of terms, to a configuration sufficient to achieve the desired degree of accuracy using a priori knowledge about the given task. Normally, only one or a few of these terms are relevant in neural networks [Lee Giles 1987]. The most common approach to determine the best architecture is to let the network grow incrementally. In this approach an initial network consisting of a few terms is chosen and new terms are added to the network as soon as the error cannot be reduced using the existing architecture. This incremental growth process is repeated until the desired error level or accuracy is reached.

3.2 Pi-Sigma Networks

Ghosh and Shin introduced another higher-order network, the 'pi-sigma' network, which avoided the exponential increase in the number of weights and processing units normally associated with higher-order networks [Ghosh *et al* 1992]. A pi-sigma network (PSN) consists of an input layer, a single hidden layer of linear summation units and product units in the output layer. The term pi-sigma comes from the fact that these networks use *products of sums* of input components. PSNs have only one layer of adjustable weights, the weights of the output layer is normally fixed at 1,

resulting in PSNs to exhibit fast learning [Ghosh *et al* 1992].

The output of a pi-sigma network is computed as follows,

$$o_k = f\left(\prod_{j=1}^{J} y_{kj}\right) \tag{3.2}$$

where

$$y_{kj} = \sum_{i=1}^{I+1} w_{kji} z_i \tag{3.3}$$

where f is the activation function, $z_i, ..., z_I$ are the input signals, z_{I+1} an input to the bias unit, w_{kji} is the weight between input unit Z_i and hidden unit Y_{kj} for the k^{th} output unit O_k, $w_{kj,I+1}$ is the threshold (or bias), y_{kj} is the output of hidden unit Y_{kj} and o_k is the output of output unit O_k. Each hidden unit is connected to only one output unit, as indicated to by the subscript k in y_{kj}. Thus equation (3.2) also shows that for multiple output PSNs an independent summing unit is required for each output unit. PSNs show a combinatorial explosion of higher-order terms as the number of inputs to the network increases.

Figure 3.2 illustrates a typical pi-sigma network, where w_{kji} is the weight between input unit Z_i and hidden unit Y_{kj}, $f(\cdot)$ is the standard logistic function applied to the output units and all the weights leading to the output unit are fixed to 1. The hidden layer consists of summation units and the output layer of product units. Let y_{kj} be the output of the j^{th} summation unit of the k^{th} output unit, O_k. A linear activation is assumed for the hidden units. A PSN provides only constrained approximation of a power series, resulting in the PSN not to uniformly approximate all continuous multivariate functions that can be defined on a compact set. However, universal approximation can be attained by summing the outputs of several PSNs of different order. The resulting network of PSNs is called a Ridge Polynomial Network

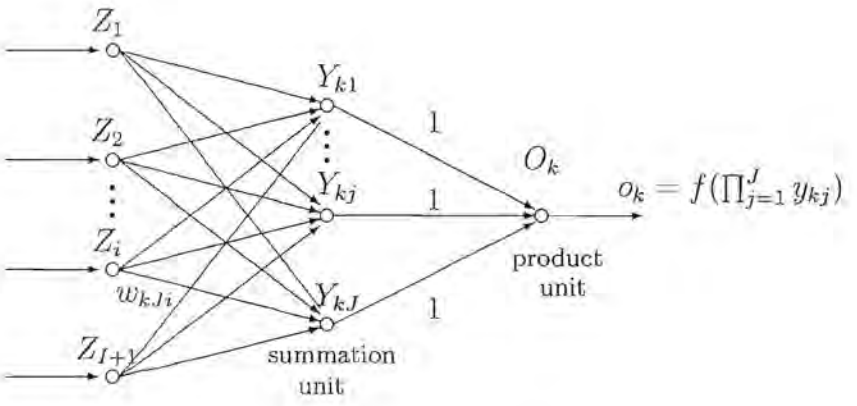


Figure 3.2: Pi-Sigma Network

(RPN) [Shin *et al* 1995]. A PSN can accept both analog and binary input/output by using suitable non-linear activation functions. The logistic function can be used as a non-linear activation function and the signum or thresholding function can be used for binary outputs.

The learning rule used by Ghosh *et al* for PSN is a randomized version of the gradient descent procedure [Ghosh *et al* 1992]. During each training cycle of a PSN, a summing unit is randomly selected and all the weights associated with this summing unit are updated using gradient descent. This modification of updating only a subset of weights, instead of all the weights, in each training cycle resulted in reduced training time of a PSN. Ghosh and Shin reported that pi-sigma networks using only three or four summing units could tackle fairly complex approximation and classification problems [Ghosh *et al* 1992].

3.3 Functional Link Networks

Functional link networks (FLNs) also generate higher-order functions of the input components [Pao 1989, Pao *et al* 1992]. FLNs are usually single-layer networks that are able to handle linearly non-separable classes by increasing the dimension of the input space by using non-linear combinations of the inputs. In FLNs, the input vector is augmented with a suitably enhanced representation of the input data, thereby artificially increasing the dimension of the input space [Ghosh *et al* 1992, Hussain *et al* 1997, Pao 1989, Pao *et al* 1992]. The extended input data are then used for training, as for standard feed-forward neural networks. Basically, the inputs are transformed in a well understood mathematical way so that the network does not have to learn basic math functions. Figure 3.3 depicts a typical functional network. In figure 3.3 the I inputs are denoted by $z_1, z_2, ..., z_I$, the bias to the hidden layer is denoted by z_{I+1} and the M extended inputs for functional links by $h_1, h_2, ..., h_M$.

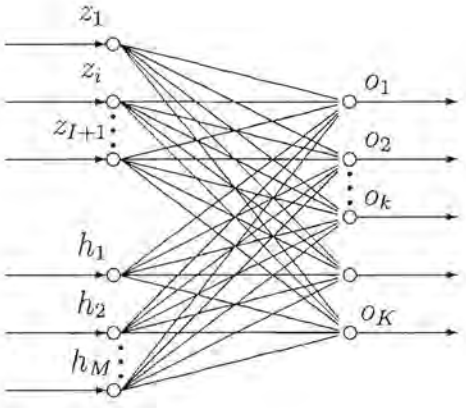


Figure 3.3: Functional Link Network

The dimensionality of the input space for FLNs can be increased in two ways [Pao 1989]:

- The *tensor* or *output product model*, where the cross-products of the input terms

are added to the model. For example, for a network with three inputs z_1, z_2, and z_3, the cross products are: z_1z_2, z_1z_3, z_2z_3, therefor adding second order terms to the network. Third order terms such as $z_1z_2z_3$ can also be added.

- *Functional expansion of base inputs*, where mathematical functions, such as sin, cos, log, etc. are used to transform the input data.

The number of terms generated using these methods grow rapidly with the increase of the dimension of the input vector. In FLNs no new information is added, but the representation of the input is merely enhanced. An advantage of FLNs is reduced training time due to the higher-order representation of the inputs, since the network does not have to learn these higher-order terms. Klassen *et al* found that functional links not only increases learning rates, but also has an effect of simplifying the learning algorithms [Klassen *et al* 1988]. Another advantage of FLNs is that it can outperform multilayer networks in certain cases due to its intrinsic mapping properties [Ghosh *et al* 1992].

3.4 Second-Order Neural Networks

Another type of higher-order neural network, the second order neural network, was developed by Milenković *et al* [Milenković *et al* 1996]. The research of Milenković *et al* was inspired by a greedy constructive neural network algorithm called the Hyperplane Determination from Examples (HDE) that suggested a discrete approach to neural network optimization suitable for parallel and distributed implementation [Fletcher *et al* 1995]. The objective of the neural network architecture developed by Milenković *et al* was to overcome the HDE local minima problem by allowing hidden units with higher representational power. The higher representational power was achieved by allowing neurons with input interactions of the following forms:

$$f(\vec{z}) = \sum_{i=1}^{I} w_i^{(1)} z_i \tag{3.4}$$

$$f(\vec{z}) = \sum_{i=1}^{I} w_i^{(1)} z_i + \sum_{i=1}^{I} w_i^{(2)} z_i z_i \tag{3.5}$$

$$f(\vec{z}) = \sum_{i=1}^{I} w_i^{(1)} z_i + \sum_{i=1}^{I} w_i^{(2)} z_i z_i + \sum_{i=1}^{I-1} \sum_{j=i+1}^{I} w_{ij}^{(3)} z_i z_j \tag{3.6}$$

where f is the activation function, $\vec{z}$ is the input vector to the network, $w_i^{(1)}$, $w_i^{(2)}$ are weight parameters associated with the i^{th} input value z_i, while $w_{ij}^{(3)}$ is a weight associated with the product of the i^{th} and j^{th} input values z_i and z_j. First-order neural networks contain neurons only constructed with interaction functions described by equation (3.4). Feed-forward neural networks that are constructed using neurons as described by all three interaction functions above, i.e. as described by equations (3.4), (3.5) and (3.6), are referred to as second-order neural networks.

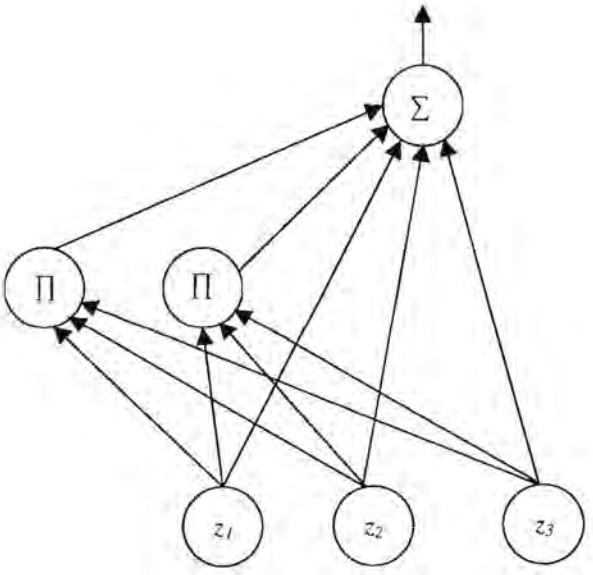
3.5 Product Unit Neural Networks

Product unit neural networks were introduced by Durbin and Rumelhart [Durbin *et al* 1989], and further explored by Janson and Frenzel [Janson *et al* 1993] and Leerink *et al* [Leerink *et al* 1995]. Durbin and Rumelhart suggested two types of networks incorporating PUs [Durbin *et al* 1989]. In the one network type (refer to figure 3.4(a)) each SU is directly connected to the input units, and also connected to a group of dedicated PUs. The other network (refer to figure 3.4(b)) consists of alternating layers of product and summation units, terminating the network with a SU.
Product units compute the net input signal as:

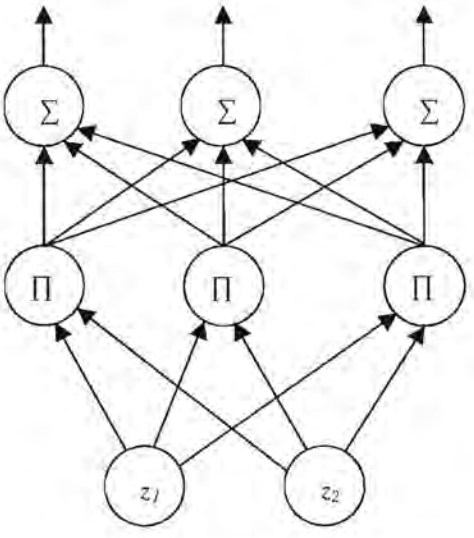
$$net_{y_j} = \prod_{i=1}^{I} z_i^{v_{ji}} + z_{I+1} \cdot v_{j,I+1}$$

instead of

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



(a) Summing units fed by inputs and dedicated product units



(b) Alternating layers of product and summing units

Figure 3.4: Two types of PUNNs

A product unit can automatically learn the higher-order term that is required by the network, unlike pi-sigma and sigma-pi units where the higher-order terms are hard-coded in the network. Product units can learn polynomials such as,

$$f(z) = a_0 + a_1 \cdot z^1 + a_2 \cdot z^2 + ... + a_n \cdot z^n \tag{3.7}$$

and any other function that can be represented by a polynomial. It can be shown that any function can be represented by a polynomial of degree n, which is a Fourier series expansion of z. The problem however, is to determine what the value of n should be when approximating a specific function using a Fourier series. Product units are much more general than sigma-pi units: While a sigma-pi unit is constrained to using just polynomial products, product units can use fractional and even negative products [Durbin *et al* 1989]. The net input to a product unit is computed as follows,

$$net_{y_j} = \prod_{i=1}^{I} z_i^{v_{ji}} - \theta_j$$

where net_{y_j} is the net input to hidden unit Y_j, Z_i is an input unit, v_{ji} is the weight between input unit Z_i and hidden unit Y_j, the threshold is denoted by θ_j and I is the total number of input units. Durbin and Rumelhart suggested two types of networks incorporating PUs [Durbin *et al* 1989]. This thesis assumes a network architecture which consists of an input layer, a hidden layer consisting of product units and an output layer consisting of summation units. Linear activations are assumed for all units. It is assumed that bias units occur in both the input and hidden layers, that respectively serve as bias to hidden units and bias to output units. However, it is shown in section 3.7.2 that the bias unit in the input layer is redundant and thus omitted from the input layer of PUNNs and replaced by a 'distortion unit' in this thesis, while retaining the bias in the hidden layer.

Neural networks are trained using learning or training rules. The next section derives

the product unit training rule for the PUNN architecture used in this thesis, assuming gradient descent as optimization algorithm.

3.6 Product Unit Training Rule

Using the architecture outlined above, the activation of a product unit for a specific pattern p is expressed in terms of logarithms and exponentials:

$$\begin{aligned} y_{j,p} &= net_{y_j,p} \\ &= \prod_{i=1}^{I} z_{i,p}^{v_{ji}} + z_{I+1,p} \cdot v_{j,I+1} \qquad (3.8) \\ &= e^{\ln(\prod_{i=1}^{I+1} z_{i,p}^{v_{ji}})} + z_{I+1,p} \cdot v_{j,I+1} \\ &= e^{\sum_{i=1}^{I} v_{ji} \ln|z_{i,p}|} + z_{I+1,p} \cdot v_{j,I+1} \\ y_{j,p} &= e^{\sum_{i=1}^{I} v_{ji} \ln|z_{i,p}|}(\cos(\pi \sum_{i=1}^{I} v_{ji}\mathcal{I}_i) + \imath \cdot \sin(\pi \sum_{i=1}^{I} v_{ji}\mathcal{I}_i)) + z_{I+1,p} \cdot v_{j,I+1} \qquad (3.9) \end{aligned}$$

where

$$\mathcal{I}_i = \begin{cases} 0 & \text{if } z_{i,p} \geq 0 \\ 1 & \text{if } z_{i,p} < 0 \end{cases} \qquad (3.10)$$

and $z_{i,p} \neq 0$. The complex part of equation (3.9) is omitted for training the PUNN, since Durbin and Rumelhart have discovered in their experiments that apart from the added complexity of working in the complex domain, i.e. doubling of equations and weight variables, no substantial improvements in results were gained [Durbin *et al* 1989].

Equation (3.9) then simplifies to (refer to appendix A),

$$y_j = e^{\rho} \cdot \cos(\pi\phi) + z_{I+1} \cdot v_{j,I+1} \qquad (3.11)$$

where

$$\rho = \sum_{i=1}^{I} v_{ji} \ln|z_i| \qquad (3.12)$$

and

$$\phi = \sum_{i=1}^{I} v_{ji}\mathcal{I}_i \qquad (3.13)$$

The objective of supervised training of NNs is to minimize the error between the approximation by the NN and the target function. The error in approximation is usually expressed as the mean squared error (MSE)

$$\mathcal{E}_{MSE} = \frac{1}{2PK} \sum_{p=1}^{P} \sum_{k=1}^{K} (t_{k,p} - o_{k,p})^2 \tag{3.14}$$

where P is the total number of training patterns (or observations), K is the number of outputs, $t_{k,p}$ is the desired (target) output for the k^{th} output unit, O_k, for a specific pattern p, and $o_{k,p}$ is the actual output of the NN. If gradient descent is used, the change in hidden-to-output weights are

$$\Delta w_{kj} = -\eta \cdot \frac{\partial E}{\partial w_{kj}} \tag{3.15}$$

and for input-to-hidden weights,

$$\Delta v_{ji} = -\eta \cdot \frac{\partial E}{\partial v_{ji}} \tag{3.16}$$

where η is the learning rate, w_{kj} is the weight between hidden unit Y_j and output unit O_k, and v_{ji} is the weight between input Z_i and hidden unit Y_j. For more detail on the derivations of these equations refer to appendix A. In the case of the hidden-to-output weights, the equations are as for standard feed-forward networks, i.e.

$$\frac{\partial E}{\partial w_{kj}} = -\delta_{o_{k,p}} \cdot f'(net_{o_k,p}) \cdot y_{j,p} \tag{3.17}$$

$$= -(t_{k,p} - o_{k,p}) \cdot y_{j,p} \tag{3.18}$$

where $f'(net_{o_k,p})$ is the derivative of the activation function used for output unit O_k (which is equal to one in the case of linear activation functions), $\delta_{o_k,p}$ is the output error, $\delta_{y_j,p}$ is the hidden layer error, $net_{o_k,p}$ and $net_{y_j,p}$ are the net input to the k^{th} output unit and j^{th} hidden unit respectively and $y_{j,p}$ is the activation of the j^{th} hidden unit. For the input-to-hidden weights,

$$\frac{\partial E}{\partial v_{ji}} = -\delta_{y_j,p} \cdot D_{ji,p} \tag{3.19}$$

where

$$\delta_{y_j,p} = \sum_{k=1}^{K} \delta_{o_k,p} \cdot w_{kj} \cdot f'(net_{y_j,p})$$

with $f'(net_{y_j,p})$ the derivative of the activation function used for hidden unit Y_j (which equals one in this case). In equation (3.19), $D_{ji,p}$ is computed as (refer to appendix A for the derivations)

$$D_{ji,p} = e^{\rho} \cdot [\ln|z_{i,p}| \cdot \cos(\pi\phi) - \pi\mathcal{I}_i \cdot \sin(\pi\phi)]$$

with

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

and

$$\phi = \sum_{i=1}^{I} v_{ji}\mathcal{I}_i$$

3.7 The Bias Unit

For the equations derived up to now, it was assumed that both the hidden units and output units receive a bias. This section shows that it is sufficient to use a bias only for the output units. This section refers to networks with biases for both hidden and output units as case 1, and networks with biases for only the output layer as case 2. The aim is to show that the learning rules for these two cases for PUNNs are equivalent, which justifies the removal of the hidden unit biases for the remainder of this thesis. For both cases 1 and 2, consider a PUNN consisting of 1 input unit, 2 hidden units and a single output unit. Figure 3.5 illustrates a network of case 1, while a network of case 2 is illustrated in figure 3.6. In figure 3.6 a bias occurs only in the output layer. An extension of PUNNs of case 2 is considered in section 3.7.3, where a 'distortion unit' is included in the product term of the product units as illustrated in figure 3.7.

3.7.1 Case 1 PUNNs

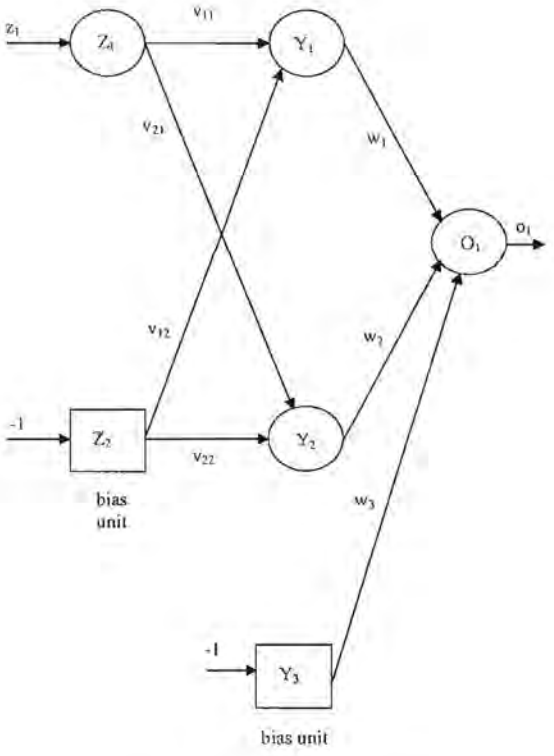


Figure 3.5: Case 1 PUNN

The output of the PUNN in figure 3.5 is,

$$o_1 = \sum_{i=1}^{I+1} y_i \cdot w_i \tag{3.20}$$

$$= y_1 \times w_1 + y_2 \times w_2 + y_3 \times w_3 \tag{3.21}$$

where w_3 is the bias and y_3 is the net input of the bias unit (always -1). Equation (3.21) simplifies to,

$$o_1 = y_1 \times w_1 + y_2 \times w_2 + c_0 \tag{3.22}$$

where $c_0 = -w_3$. We now proceed to compute the activation values y_1 and y_2 for the hidden units Y_1 and Y_2 respectively:

$$y_1 = z_1^{v_{11}} + z_2 \cdot v_{12}$$

$$= z_1^{v_{11}} - v_{12} \tag{3.23}$$

$$y_2 = z_1^{v_{21}} + z_2 \cdot v_{22}$$

$$= z_1^{v_{21}} - v_{22} \tag{3.24}$$

Substitutition of (3.23) and (3.24) in (3.22) yields,

$$\begin{aligned} o_1 &= (z_1^{v_{11}} - v_{12}) \times w_1 + (z_1^{v_{21}} - v_{22}) \times w_2 + c_0 \\ &= (z_1^{v_{11}} + c_1) \times w_1 + (z_1^{v_{21}} + c_2) \times w_2 + c_0 \\ &= z_1^{v_{11}} \cdot w_1 + c_1 \cdot w_1 + z_1^{v_{21}} \cdot w_2 + c_2 \cdot w_2 + c_0 \\ &= z_1^{v_{11}} \cdot w_1 + z_1^{v_{21}} \cdot w_2 + c_3 \end{aligned} \tag{3.25}$$

where $c_3 = c_0 + c_1 \cdot w_1 + c_2 \cdot w_2$. All the c_i's (i.e. c_0, c_1 and c_2) are basically weights obtained through training and can thus be replaced and trained as a single weight, c_3.

3.7.2 Case 2 PUNNs

Now consider the second case where the bias is removed from the hidden layer. The PUNN in figure 3.6 represents a 1:2:1 network configuration.

The output for the PUNN in figure 3.6 is,

$$o_1 = \sum_{i=1}^{I+1} y_i \cdot w_i \tag{3.26}$$

$$= y_1 \times w_1 + y_2 \times w_2 + y_3 \times w_3 \tag{3.27}$$

Once again, the net input of the bias unit, Y_3, is assumed as -1. Equation (3.27) then simplifies to,

$$o_1 = y_1 \times w_1 + y_2 \times w_2 + c_4 \tag{3.28}$$

where $c_4 = -w_3$. The values for y_1 and y_2 are computed as,

$$y_1 = z_1^{v_{11}} \tag{3.29}$$

$$y_2 = z_1^{v_{21}} \tag{3.30}$$

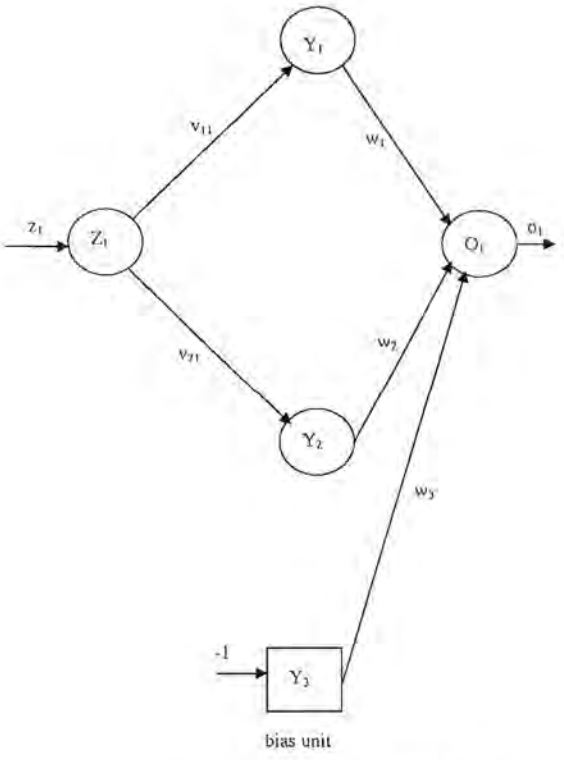


Figure 3.6: Case 2 PUNN

Substitution of (3.29) and (3.30) in (3.28) yields,

$$o_1 = z_1^{v_{11}} \cdot w_1 + z_1^{v_{21}} \cdot w_2 + c_4 \tag{3.31}$$

A comparison of equations (3.25) and (3.31) indicates that these two equations are equivalent if and only if c_3 is equal to c_4. In equation (3.25), c_3 is a function of biases and weights that are not dependent on inputs, clearly indicating that c_3 is basically a constant. Thus, instead of learning three different constants (i.e. c_0, c_1 and c_2), only one constant, namely c_4, can be learnt, which will result in equations (3.25) and (3.31) to be equivalent. Note, however, that 'case 1' has more weights and thus more degrees of freedom than 'case 2'. The higher the degrees of freedom, the higher the probability of getting poor results, since an increase in the number of weights causes a corresponding increase in dimensionality of the search space, that will inevitably contain more local minima and plateaus. Generally, case 2 should therefor produce

better results than 'case 1'.

3.7.3 The Distortion Unit

In addition to having biases only for the output units, the PUNNs studied in this thesis were further extended by including a 'distortion' factor in the product term of the product units. The PUNN in figure 3.7 represents a 1:2:1 network configuration with a distortion unit replacing the bias unit in the input layer.

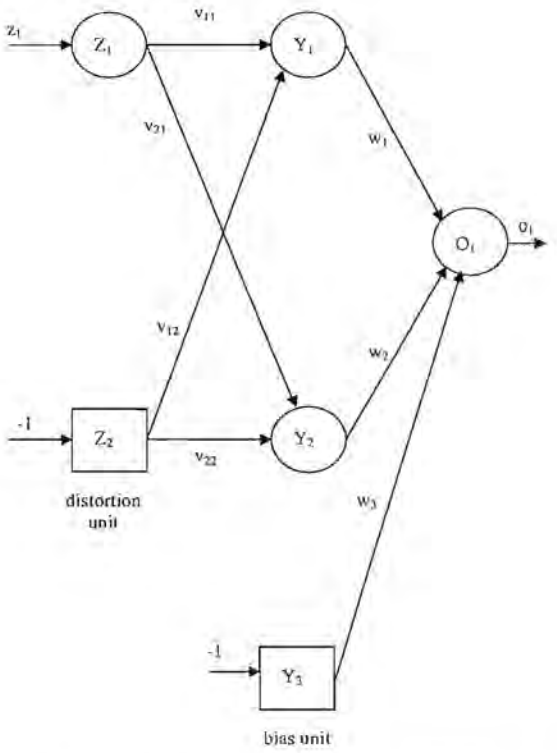


Figure 3.7: PUNNs with a distortion unit

For the product units, the net input signal is therefor calculated as

$$net_{y_j} = \prod_{i=1}^{I+1} z_i^{v_{ji}} \tag{3.32}$$

where $z_{I+1} = -1$ and $v_{j,I+1}$ is the distortion factor. The distortion unit has a constant input of -1. Thus, $z_{I+1}^{v_{j,I+1}}$ simplifies to $(-1)^{v_{j,I+1}}$. This expression, $(-1)^{v_{j,I+1}}$, is not defined for all values of $v_{j,I+1}$ in the real domain. However, in the derivation of the learning equations for PUNNs the calculation of $(-1)^{v_{j,I+1}}$ is performed in the complex domain (refer to equations A.20 to A.22 on page 196). Since, $z_{I+1} = -1$, $ln|z_{I+1}|$ reduces to $ln|-1|$ which equals 0. Thus, the distortion unit does not make any contribution to term ρ, however the value of the weight $v_{j,I+1}$ is added in calculating term ϕ on page 196 when $z_{I+1} = -1$. This shows that $(-1)^{v_{j,I+1}}$ is defined for all negative values of $v_{j,I+1}$. The distortion unit acts to assist in shaping the activation function to more accurately fit the true function as represented by the training data.

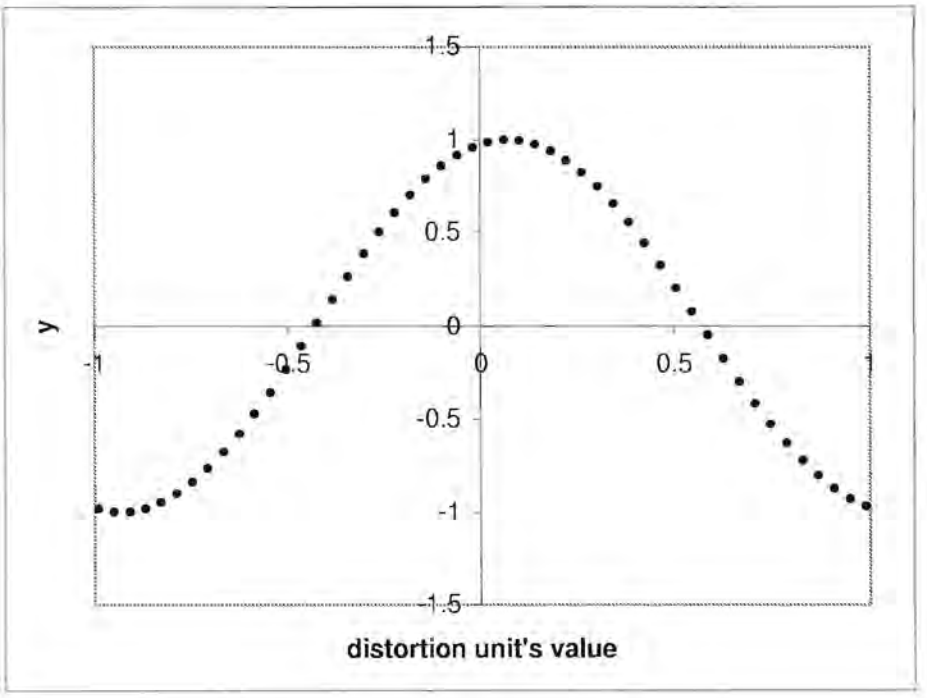


Figure 3.8: Effect of the distortion unit in approximating $f(z) = z^2$

This unit cannot be seen as a bias, since it is not added to the learning rule and plays no role in offsetting the origin of the function, but is rather included in the product. Upon inspection of $net_{y_j} = e^{\rho} \cdot cos(\pi \cdot \phi)$ (from equation (A.45) on page 200), it is observed that the distortion factor's contribution to term ρ, is 0, since in $e^{weight\_of\_distortion\_unit \times ln(|-1|)}$, $ln|-1| = 0$ thus reducing $e^{weight\_of\_distortion\_unit \times ln(|-1|)}$ to 1. The distortion unit thus only contributes to term ϕ. The net effect of the distortion factor on net_{y_j} is thus limited to the contribution of $cos(\pi \cdot \phi)$ to the net input signal. Thus,

$$net_{y_j} = e^{\rho} \cdot \cos(\pi\phi) \tag{3.33}$$

where

$$\rho = \sum_{i=1}^{I} v_{ji} \ln |z_i| \tag{3.34}$$

and

$$\phi = \sum_{i=1}^{I+1} v_{ji} \mathcal{I}_i \tag{3.35}$$

To explain the purpose of the distortion unit, consider approximation of function $f(z) = z^2$ as illustrated in figure 3.9. Further inspection of the distortion term, in the case of function $f(z) = z^2$, $for\ -1 < z < 1$, revealed that the unit mapped a function of the form $cos(3z)$ over the data, re-affirming the fact that this distortion unit effectively assists in shaping the function to better fit the set of training data. The remainder of this thesis assumes PUNNs with a distortion unit.

3.8 Problems with Training of PUNN using Gradient Descent

Gradient descent (GD) is one of the most popular optimization algorithms used to train multilayer neural networks that employ summation units, resulting in the so-called

back-propagation neural network [Werbos 1974]. Gradient descent works best when the search space is relatively smooth, with few local minima or plateaus. In such cases the minima are not too deep and any randomness added to the training process will prevent the network from getting stuck in local minima [Zurada 1992]. This section shows that GD has difficulties in training networks that use product units. These difficulties arise from the increased number of local minima and more convoluted search space due to PUs [Durbin *et al* 1989, Leerink *et al* 1995].

Durbin and Rumelhart have constructed a neural network consisting of 1 hidden product unit and a standard summing output unit to solve the 6-parity problem where the weights were calculated from first principles [Durbin *et al* 1989]. The parity function, when implemented using summation unit neural networks, require as many hidden units as inputs. Leerink *et al* [Leerink *et al* 1995], however, have found that the *back-propagation algorithm could not train* a product unit neural network on the 6-parity problem, due to the following reasons:

- **Incorrect weight initialization:**

  The initial weights of a network is usually computed as small random values in order to use the dynamic range of the sigmoid function and its derivative. Leerink *et al* argued that this is the worst possible choice of initial weights for PUNNs, and suggested that larger initial weights be used instead [Leerink *et al* 1995]. From own experience, back-propagation only manages to train product unit neural networks when the weights are initialized in close proximity of the optimal weight values. The optimal weight values are, however, usually not available. Gradient descent procedures are usually not able to compensate for bad initial values of weights and biases, getting stuck in local minima. To combat the problem of bad initial weights, global optimization algorithms can be used to find

initial weights and GD subsequently applied to train the network, as suggested in [Ismail *et al* 2000].

- **Increased number of local minima:**

  A major drawback of product units is an increased number of local minima, deep ravines and valleys on its error surface. The search space for product units is usually extremely convoluted [Janson *et al* 1993]. This is because the exponent component, v_{ji}, in equation (3.8) can cause large changes in the computation of the total error. Back-propagation by gradient descent therefore frequently gets *trapped in local minima* that it cannot escape from, or becomes *paralyzed* if a local minimum is reached, thus resulting in no adjustment of the weights due to the fact that the error with respect to the current weight is close to zero; the weight vector thus remains the same for the remainder of the training session.

As an example to illustrate the complexity of the search space for product units, and the problems mentioned above, consider the approximation of the function $f(z) = z^3$, with $z \in [-1, 1]$. To approximate this function using PUNNs, one PU is sufficient, resulting in a minimal 1-1-1 architecture. In this case the optimal weight values are $v = 3$ (the input-to-hidden weight) and $w = 1$ (the hidden-to-output weight), where the bias and the distortion are both equal to zero. Figure 3.9 visualizes the search space for $v \in [-1, 4]$ and $w \in [-1, 1.5]$. The error is computed as the mean squared error over 500 randomly generated patterns. Figure 3.9 clearly illustrates 2 local minima, one located at $v = -0.55$ and the other at $v = 1.25$. The global minimum is at $v = 3$. Initial small random weights will cause the network to be trapped in one of the local minima (having very large MSE). Large initial weights may also be a bad choice. Assume an initial weight $v \geq 4$ (or $v \leq -1$). The derivative of the error with respect to v is extremely large due to the steep gradient of the error surface. Consequently, a large

weight update will be made which may cause jumping over the global minimum. The neural network either becomes trapped in a local minimum, or oscillates between the extreme points of the error surface.

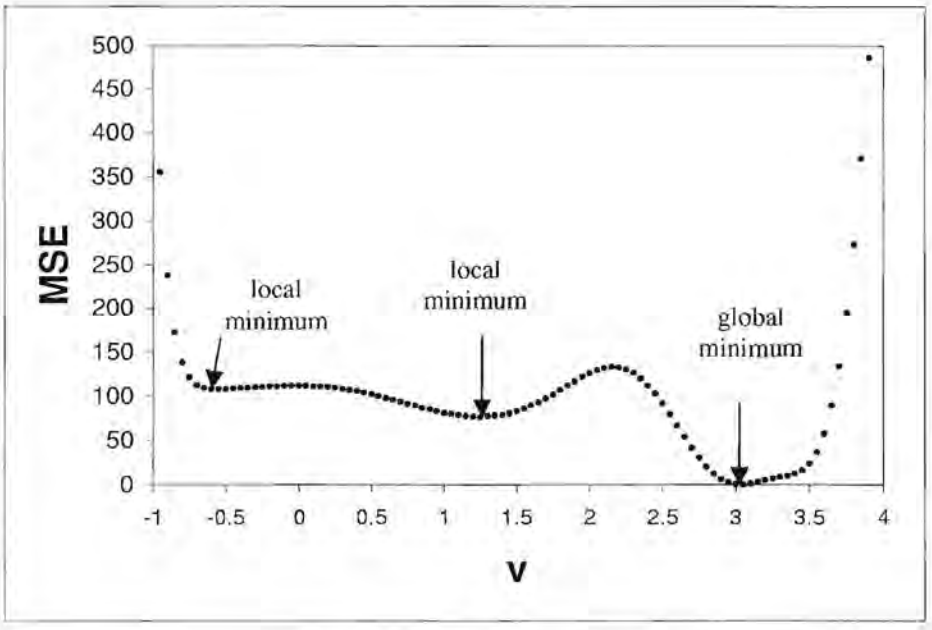


Figure 3.9: MSE values with weight w fixed at 1 for $f(z) = z^2$

Another example to illustrate the numerous local and global minima that occur in the search space of PUNNs is illustrated in figure 3.10, for approximation of the function $f(z_1, z_2) = z_1^2 + z_2^2$.

The function $f(z_1, z_2) = z_1^2 + z_2^2$, with $z_1, z_2 \in [-1, 1]$, can be approximated with a PUNN that contains a minimum of 2 hidden PUs which amounts to 6 weights. A PUNN to approximate $f(z_1, z_2) = z_1^2 + z_2^2$, comprising a 2:2:1 configuration, is represented in figure 3.11. The search space for this PUNN is thus 6-dimensional, making visualization of the error surface very difficult. However, slices of the error surface can be viewed by

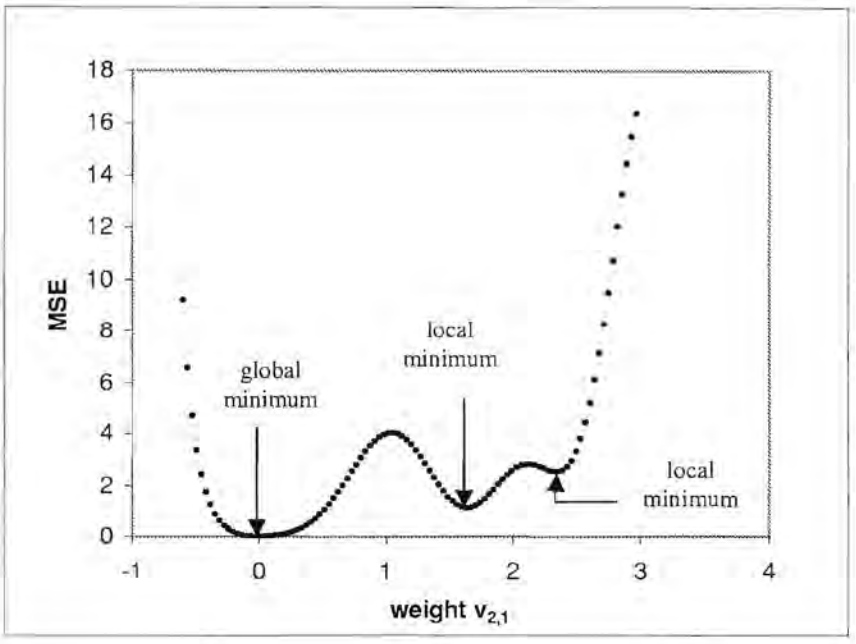


Figure 3.10: Error surface for the straight line between 3 minima, $f(z_1, z_2) = z_1^2 + z_2^2$

fixing most of the weights and varying only one or two of the weights. Figure 3.10 visualizes the search space for all weights fixed, except $v_{21} \in [-1, 4]$. Three minima are illustrated with the global minimum at $v_{21} = 0$. Initial small random weights will cause the network to converge to the global minimum. Large initial weights, however, will cause the network to be trapped in one of the local minima resulting in a large MSE. Initial weights $v_{21} > 3$ or $v_{21} < -1$ will also be a bad choice, since the derivative of the error with respect to v_{21} is extremely large due to the steep gradient of the error surface. A large weight update will be made which may result in overshooting the global minimum. Thus, the neural network becomes trapped in a local minimum, or oscillates between extreme points of the error surface.

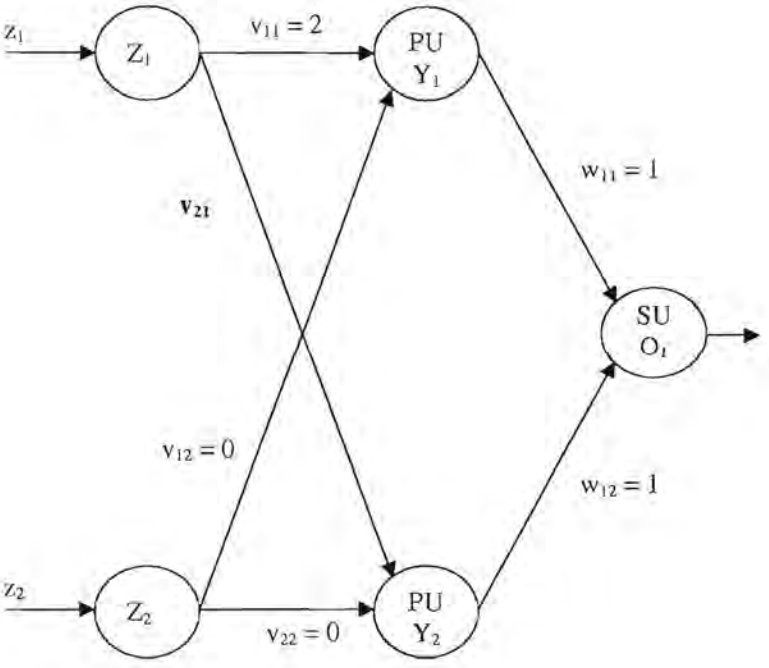


Figure 3.11: PUNN to approximate $f(z_1, z_2) = z_1^2 + z_2^2$

3.9 Conclusion

This chapter discussed the problems encountered when GD is used to train PUNNs. Gradient descent is frequently trapped by local minima that occur in the search space for PUNNs. Local minima are particularly prevalent in networks containing PUs, due to the effect of the exponential terms in the learning equations. These exponential terms cause large weight adjustments that result in the network to be trapped or oscillate between the extreme points. To alleviate these problems, global optimization algorithms should be used to train PUNNs.

The next chapter discusses various global optimization algorithms to train PUNNs.

Chapter 4

Global Optimization Algorithms

Gradient descent (GD) is the most popular local optimization algorithm to train multilayer NNs. While GD has shown to be successful in training SUNNs, GD fails to train PUNNs under general assumptions of weight initialization, as shown in the previous chapter. This chapter presents an overview of the following global optimization algorithms: Particle Swarm Optimization (PSO), Genetic Algorithms (GAs) and Leapfrog Optimization (LFOP). These algorithms are subsequently applied to approximate a set of functions, using PUNNs. The results are compared with that of SUNNs, using gradient descent optimization.

4.1 Particle Swarm Optimization

Particle swarm optimization (PSO) is a global optimization approach, modeled after the social behaviour of flocks of birds [Eberhart *et al* 1996, Heppner *et al* 1990, Reynolds 1987] and schools of fish [Wilson 1975] . Heppner was interested in discovering the underlying rules that enabled large numbers of birds to flock synchronously, often changing direction suddenly, scattering and regrouping [Kennedy *et al* 1995b].

These scientists had the insight that local processes, such as those modelled by cellular automata, might underlie the unpredictable group dynamics of bird social behaviour [Kennedy *et al* 1995b]. The models proposed by these scientists relied heavily on manipulation of inter-individual distances; that is, the synchrony of flocking behaviour was thought to be a function of birds' efforts to maintain an optimum distance between themselves and their neighbours.

Particle swarm optimization was originally developed by Eberhart and Kennedy [Eberhart *et al* 1995, Eberhart *et al* 1996, Kennedy 1995a, Kennedy *et al* 1995b]. PSO is a population based search procedure where the individuals, referred to as particles, are grouped into a swarm. Each particle in the swarm represents a possible solution to the optimization problem under consideration. In a PSO system, each particle is 'flown' through the multidimensional search space, adjusting its position in search space according to own experience and that of neighbouring particles. Each particle is treated as a point in a D-dimensional space. The p^{th} particle is represented as $\vec{x}_p = (x_{p,1}, x_{p,2}, ..., x_{p,D})$. The best previous position (i.e. the position that produces the best fitness value) of the p^{th} particle is recorded and represented as $\overrightarrow{BESTx}_p = (BESTx_{p,1}, BESTx_{p,2}, ..., BESTx_{p,D})$, and the index of the best particle among all the particles in the population is represented by, GBEST. Let the rate of change in position (i.e. velocity) for particle p be represented as $\vec{v}_p = (v_{p,1}, v_{p,2}, ..., v_{p,D})$. The p^{th} particle is adjusted according to the following equation,

$$\vec{v}_p(t) = w \times \vec{v}_p(t-1) + c_1 \times rand1() \times (\overrightarrow{BESTx}_p - \vec{x}_p(t)) + c_2 \times rand2() \times (\overrightarrow{BESTx}_{GBEST} - \vec{x}_p(t)) \quad (4.1)$$

$$\vec{x}_p(t+1) = \vec{x}_p(t) + \vec{v}_p(t) \quad (4.2)$$

where c_1 and c_2 are positive constants, referred to as the acceleration constants, and $c_1 + c_2 < 4$ to ensure convergence [Van den Bergh 2001a], $rand1()$ and $rand2()$ are two

random functions with output in the range $[0, 1]$, w is the inertia weight and $\vec{v}_p$ the velocity of particle p before the adjustment [Shi *et al* 1998].

Equation (4.1) is used to calculate the particle's new velocity using its previous velocity and the distances of its current position from its own best experience (position) and the group's best experience, which is defined in terms of the type of social interaction that is being modeled [Shi *et al* 1998]. Two approaches of PSO have been developed by Eberhart and Kennedy, one globally oriented, referred to as GBEST, and one locally oriented referred to as LBEST [Eberhart *et al* 1995]. In both approaches, each particle of the swarm keeps track of its coordinates in search space which are associated with the best solution the particle has achieved so far. This position is referred to as $\overrightarrow{BESTx}$. In the local version of PSO, each particle keeps track of the best solution called 'LBEST', attained within a local topological neighbourhood of particles. In the GBEST model the group's best experience is indicated to by index 'GBEST'. The particle therefore makes use of the best position encountered by itself and the overall best position of either,

- all particles, as indicated to by 'GBEST' (GBEST model) or
- a neighbourhood of particles, as indicated to by 'LBEST' (LBEST model)

to position itself towards the global minimum. The effect is that particles 'fly' towards the global minimum, while still searching a wide area around the best solution.

The performance of each particle (i.e. the 'closeness' of a particle to the global minimum) is measured according to a predefined fitness function which is related to the problem being solved. Research has shown that the GBEST version of PSO performs best in terms of a median number of iterations to converge compared to the LBEST model [Eberhart *et al* 1996].

The PSO algorithm is summarized below to illustrate its simplicity:

4.1.1 PSO Algorithm

1. Initialize a swarm of S D-dimensional particles, with positions and velocities, where D is the number of weights and biases.
2. Evaluate the fitness f_p of each particle p as the MSE over a given data set.
3. If $f_p < BEST_p$ then $BEST_p = f_p$ and $\overrightarrow{BEST}x_p = \vec{x}_p$, where $BEST_p$ is the current best fitness achieved by particle p, $\vec{x}_p$ is the current position of particle p in D-dimensional weight space, and $\overrightarrow{BEST}x_p$ is the position corresponding to particle p's best fitness so far.
4. If $f_p < BEST_{GBEST}$ then $GBEST = p$, where $GBEST$ is the particle having the overall best fitness over all particles in the swarm.
5. Change the velocity $\vec{v}_p$ of each particle p using equation (4.1).
6. Fly each particle p to $\vec{x}_p + \vec{v}_p$
7. Loop to step 2 until convergence

In step 5, the coordinates $\overrightarrow{BEST}x_p$ and $\overrightarrow{BEST}x_{GBEST}$ are used to pull the particles towards the global minimum, and the acceleration constants, c_1 and c_2, control how far particles fly from one another.

Initially, all particles are assigned random positions, selected from a range that covers the entire search space, and random velocities that do not exceed a maximum velocity as specified for the problem.

The next section discusses the parameters; inertia weight, maximum velocity and acceleration constant of PSO.

4.1.2 Inertia Weight

The purpose of the inertia weight is to control the impact of the previous history of velocities on the current velocity. A larger inertia weight favours global exploration, while a smaller inertia weight tends to facilitate local exploration of the search area [Kennedy *et al* 1995b]. Suitable selection of the inertia weight w can provide a balance between local and global exploration abilities of PSO, and thereby reducing the number of iterations required in reaching an optimum. The value for the inertia weight is problem dependent, but usually values between 0 and 1.0 are used. PSO with decreasing inertia weight has also been implemented, where the PSO usually starts off with a large inertia weight, say 1.0, and gradually reduces it with time.

4.1.3 Maximum Velocity

The maximum velocity is used to prevent large velocity updates, thereby preventing particles from leaving the search space. Thus, in PSO, the value of the maximum velocity is *limited* to prevent particles from flying out of the search space. Shi and Eberhart pointed out that the maximum velocity acts as a constraint that controls the maximum global exploration ability of PSO [Shi *et al* 1998]. The maximum global exploration ability of PSO is limited if the maximum velocity is too small. If the maximum velocity is too small, then particles may not explore sufficiently beyond good regions. Further, they may become trapped in local minima, unable to jump far enough to reach a better position in the search space [Eberhart *et al* 1996]. A larger maximum velocity, increases PSO's maximum global exploration ability. A too high

maximum velocity, however, will result in particles flying past good solutions. The maximum velocity determines also the fineness with which regions between the present position and the target position will be searched. The value of the maximum velocity should thus be selected carefully. The maximum velocity is limited by the maximum value of the parameters for the problem at hand, i.e. the maximum value of the inputs. Usually values in the range 0 to 5 are chosen for the maximum velocity.

4.1.4 Acceleration Constants

The acceleration constants, c_1 and c_2, represent the weighting of the stochastic acceleration terms that pull each particle toward positions $\overrightarrow{BEST x_p}$ and $\overrightarrow{BEST x_{GBEST}}$. Thus, adjustment of this factor changes the amount of 'tension' in the system. Low values allow particles from far target regions to explore the search space before being tugged back, while high values result in abrupt movement toward the target regions [Eberhart *et al* 1995]. The selection of values for the acceleration constant is problem dependent, however values normally range between 0 and 5. Also, if the constraint, $c_1 + c_2 < 4$, is not satisfied then PSO does not usually converge [Eberhart *et al* 2000, Van den Bergh 2001a].

4.1.5 Applications of PSO

Particle swarm has been used successfully to train SUNNs [Kennedy *et al* 1995b, Van den Bergh 1999, Van den Bergh *et al* 2000] and PUNNs [Engelbrecht *et al* 1999a, Ismail *et al* 1999, Ismail *et al* 2000, Van den Bergh *et al* 2001b], for function optimization [Eberhart *et al* 1995, Shi *et al* 1999, Van den Bergh *et al* 2001d] and for human tremor analysis [Eberhart *et al* 1999]. Van den Bergh found that training of

multilayer feed-forward networks using various gradient descent based algorithms can be improved significantly by using particle swarm optimization in selecting initial weights. Van den Bergh showed that the initial weights produced by PSO increased the speed and accuracy with which gradient descent algorithms find the minimum [Van den Bergh *et al* 2000]. Another researcher, Salerno, also applied particle swarm optimization successfully to train a recurrent neural network in parsing natural language phrases [Salerno 1997].

PSO has also been demonstrated to perform well in optimizing genetic algorithm test functions, such as the extremely nonlinear Schaffer $f6$ function. The $f6$ function is very difficult to optimize, as the highly discontinuous data surface features many local minima. PSO found the global optimum each run and appears to approximate the results reported by Davis for basic genetic algorithms in terms of the number of evaluations to reach certain performance levels [Kennedy *et al* 1995b]. PSO can be used to solve many of the same kind of problems as solved by genetic algorithms [Kennedy *et al* 1995b]. Eberhart *et al* used PSO successfully to extract rules from fuzzy neural networks [Eberhart *et al* 1998]. BK Birge, a former student of Eberhart, one of the developers of PSO, and Y Shi is currently applying PSO to 'intelligent control' for NASA's next generation 'Robotic Mars Landers'. Current research in PSO use constriction coefficients which have lead to improved performance of PSO [Eberhart *et al* 2000].

4.1.6 Advantages of PSO

The PSO offers several advantages, which makes it an excellent choice to solve optimization problems with a continuous search space. These advantages include:

- PSO is conceptually simple and can be implemented in a few lines of code, requiring only basic mathematical operations.
- In PSO, neural network weights and structures are evolved in such a way as to make preprocessing of neural network data unnecessary.
- PSO is computationally inexpensive in terms of both memory requirements and speed [Kennedy *et al* 1995b].
- PSO is a stochastic global optimization algorithm.

Another advantage is that PSO does not suffer from some of the difficulties encountered with genetic algorithms (e.g. running the risk of finding suboptimal solutions); interaction in the group enhances rather than detracts from progress toward a solution [Eberhart *et al* 1996]. PSO also has memory, which a genetic algorithm generally does not have. Change in genetic populations results in destruction of previous knowledge of the problem except when elitism (i.e. when individuals with highest fitness of the current generation is copied into the next generation) is employed, in which case a small number of individuals retain their identities. This serves as limited memory.

PSO is a global optimization algorithm and training of a NN is an optimization problem. Hence, PSO can be used for training a NN, in which case each particle will represent a weight of the NN (including biases). The dimension of the search space is therefore the total number of weights and biases. The fitness function is the mean squared error (MSE) over the training set, or the test set (as measure of generalization). This thesis implements the GBEST version of PSO.

This concludes the presentation of PSO. The next section presents genetic algorithms (GAs) as an optimization algorithm.

4.2 Genetic Algorithms

Evolutionary computing has been used successfully to solve optimization problems. Of these, genetic algorithms (GAs) are the most popular. GAs are based on the principle of natural evolution where principles such as survival of the fittest, natural selection, reproduction and mutation are used to produce a 'best' individual. The idea of a genetic algorithm as a global optimization tool was first introduced by John Holland in the 1970's [Goldberg 1989]. A genetic algorithm is a *global search* technique compared to gradient descent that is a *local* search method. A GA represents an intelligent exploitation of a random search used to solve optimization problems.

Genetic algorithm paradigms work on populations of individuals, rather than on single data points or vectors. In a GA, a population of individuals compete to survive. Each individual represents a point in search space, which represents one possible solution to the optimization problem. In this thesis, an individual represents a weight vector (including biases and distortion units) of a NN. Each individual is represented as a character string that is analogous to the chromosome that occurs in DNA. The survival strength, or fitness, of an individual is measured using a fitness function, the MSE when a GA is used to train a NN. The fitness value represents the abilities of an individual to survive.

Most optimization paradigms move around in the search space using some heuristic. One of the drawbacks of this approach is the likelihood of getting stuck at a local minimum. GAs on the other hand start off with a diverse set of points called a population. From one population to the next the same number of individuals is maintained, thus allowing many maxima to be explored efficiently and thereby lowering the probability of getting stuck in a local minimum. GAs use 'selective breeding' of

the solutions to produce 'offspring' that exhibit better fitness than the parents by combining 'genes' of the parents. GAs do not require any auxiliary information, such as derivatives in determining the maximum (or minimum). GAs are generally more robust than conventional artificial intelligence systems, in that they will still produce reasonable results in the presence of noise or if the inputs change. A GA may offer significant benefits over more typical search optimization techniques, such as linear programming, heuristics, depth-first and breadth-first [Mitchel 1996].

Optimization in a GA proceeds through the generation of new individuals by probabilistically applying crossover and mutation operators. Parents are selected for reproduction based on their fitness. Individuals with high fitness are given more opportunities to reproduce, than individuals with low fitness. Thus the larger the fitness of an individual, the more likely it is that it will be used during crossover to exchange 'genetic material' with another individual to produce better individuals. Thus 'genes' from good individuals produce offspring that are often 'better' than the parents. Mutation occurs by randomly changing a 'gene' of an individual. New offspring replace other individuals with lower fitness. It is hoped that after successive generations better solutions will replace weaker ones.

4.2.1 Applications of GAs

GAs have been used successfully for many applications, which include the training of NNs. Schiffman *et al* used GAs to train SUNNs [Schiffman *et al* 1992], while Frenzel has applied GAs to train PUNNs [Janson *et al* 1993]. In a study conducted by Dagli and James to search for optimal parameters (such as the number of nodes in each layer and the number of layers) for a neural network, the parameters rather than the weights were encoded in the GA chromosome where the neural network's performance with

these parameters was used as the fitness function [Dagli *et al* 1995]. Other applications of GAs include,

- pattern classification [Chang *et al* 1991],
- feature selection for neural networks [Guo *et al* 1992],
- the initialization of Radial Basis Networks [Billings *et al* 1995, Burdsall *et al* 1997],
- the training of cellular neural networks [Zamparelli 1997],
- to explain the behaviour of neural networks by defining a function linking the network inputs and outputs [Opitz *et al* 1994], and
- to configure radial basis function (RBF) neural networks [Billings *et al* 1995, Kuo *et al* 1994, Whitehead *et al* 1996]. Specifically, they have been applied to find the optimal (Gaussian) parameters used (centres, widths), as well as the structure (number of hidden nodes) of the RBF network.

A general genetic algorithm for training NNs is presented below.

4.2.2 Genetic Algorithm

1. Initialize a population, $G(t)$, of individuals (weight vectors).
2. Calculate the fitness of each individual of the population as the MSE over the training set.
3. Select parents for reproduction from the current population, $G(t)$. Two individuals are selected from the population using *ranking* as the selection operator (refer to section 4.2.8).
4. Perform crossover to produce new individuals for population, $G(t+1)$. A two-point crossover operator is used (refer to section 4.2.8).

5. Perform mutation of population, $G(t+1)$.

6. Loop to step 2 until best individual is acceptable.

4.2.3 Initialization and Size of Population

Initialization of the population is usually done stochastically. It is sometimes appropriate to start with one or more individuals that are selected heuristically, to aim the GA in a promising direction. Generally the population should represent a wide assortment of individuals. Researchers have shown that the urge to skew the population significantly should generally be avoided. Choosing the size of the population is more an art than science. Following De Jong's guidelines, a moderately sized population should be used initially [De Jong 1975].

4.2.4 Representation

The objective of applying GAs to neural network training is to find a suitable set of weights that results in the smallest MSE on the training set and that generalizes well on the test set. To achieve this objective the GA has to be populated with sets of weights where each set of weights is a possible solution in training the network. Thus, for NN training, each individual of the GA contains the same number of genes as the number of weights (including biases) that occur in the neural network. Each weight value has to be converted to a binary representation, since this thesis assumes that the GA paradigm uses a binary alphabet. The accuracy of the final weight values are determined by the number of bits used in the binary representation and the range of values that is mapped onto this binary representation, e.g. to map real numbers in the range [-3.0,3.0] onto a 30 bit binary representation, implies a mapping onto 2^{30},(i.e. 1073741824), distinct values. Thus -3.0 is mapped onto say 000000000000000000000000000000 and 3.0 mapped onto

11111111111111111111111111111111 (30 ones); a finite number of real values between -3.0 and 3.0 are then mapped onto the remaining binary representations that exist between these two binary numbers. In this representation two successive real numbers will therefor differ by magnitude $(3.0-(-3.0))/(2^{30}-1) = 6/1073741823 = 5.5879E-09$. In this representation any two weights that differ by less than this magnitude are thus indistinguishable or regarded as representing the same number. For the implementation in this thesis, each weight is mapped onto a 30 bit binary number. Weight values are restricted to the range $[w_{min}, w_{max}]$, in other word the evolutionary process cannot evolve weights beyond these boundaries. The following mapping function is used to convert floating-point weight values to binary representation:

$$(2^{30}-1)\frac{w-w_{min}}{w_{max}-w_{min}} \tag{4.3}$$

4.2.5 Fitness

One method of fitness calculation is to 'equally space' the fitness values in some manner, say from 0 to 1. The most fit individual has a maximum fitness of 1. Another method of fitness calculation is 'scaling' that takes into account the recent history of the population. If the objective of a GA is to maximize some function, then scaling involves keeping a record of the minimum fitness value obtained in the last s generations, where s is the size of the scaling window. If, for example , $s = 10$, then the minimum fitness value in the last 10 generations is kept and used instead of 0 as the 'floor' of fitness values. Fitness values are then assigned a value based on their actual distance from the floor value. The fitness function used in this thesis is defined as $f(w) = \frac{1}{1+MSE(w)}$. Hence, the smaller the MSE, the larger the fitness value.

4.2.6 Crossover

Crossover is inspired by natural evolution processes. Crossover is a reproduction operator which forms a new individual (chromosome) by combining parts of each of two 'parent' chromosomes with an objective of increasing the fitness value of the new individual. The most 'basic' crossover type is *one-point crossover*, as describe by Holland [Holland 1992] and others, e.g. Goldberg [Goldberg 1989] and Davis [Davis 1991]. One-point crossover involves selecting a single crossover point at random and exchanging the portions of the individual strings of the parents to the right of the crossover point. In *two-point crossover*, on the other hand, two parents are randomly selected from the population and a stochastic decision is made whether or not to perform crossover. Subsequently, if crossover has to be performed, a two-point crossover site along the character string is randomly chosen. The corresponding values occurring between these two points in each parent are then exchanged. An alternative is *uniform crossover*, where two parents are chosen at random and a stochastic decision is made whether or not to perform crossover [Syswerda 1991]. If crossover has to be performed then a random decision is made at each bit position in the string as to whether or not to exchange corresponding bits between the two parent strings. De Jong suggested a high crossover rate of between 0.5 and 0.9 [De Jong 1975]. The values for crossover is, however, problem dependent. In this thesis two-point crossover is used with crossover rates varying between 0.5 and 0.9.

4.2.7 Mutation

Mutation is a way of varying the 'gene pool' that provides some protection against 'in breeding' in a population. Mutation is achieved by stochastically flipping the bits of the individuals during each generation at a certain probability. Mutation is usually performed with a low probability, but higher probabilities are not unusual. A good

strategy is to start off with a fairly high probability for mutation that is decreased with time. This allows the GA, initially, greater exploration abilities. In this thesis fixed mutation rates between 0.001 and 0.5 are used.

4.2.8 Reproduction or Selection

Reproduction is a process in which individual strings are selected for mating according to their fitness values. Thus strings with high fitness values have a high probability of contributing to one or more offspring in the next generation [Goldberg 1989]. Operators for implementing reproduction are random selection, biased roulette wheel or tournament selection. In *random selection* individuals for the next generation are randomly selected from the current population. In the *biased roulette wheel* approach each individual is assigned a roulette wheel slot sized in proportion to its fitness. Individuals with high fitness will thus have a bigger size slot than individuals with low fitness values. The roulette wheel is then spun n times to generate a population of size n. The individual that corresponds to the slot that the dice ends up in after each spin is added to the next generation. Thus the bigger the slot size, the greater the probability that the dice will land in it and thus the greater the probability of that individual being added to the next generation. One variation on the roulette wheel was developed by Baker in which the portion of the roulette wheel is assigned, based on each unique string's relative fitness [Baker 1987]. One spin of the roulette wheel then determines the number of times each string will appear in the next generation.

In the most common variation of *tournament selection* two individuals are selected at random and the member with a higher fitness value is selected for the next generation. This process is repeated n times for a population of size n. Other variations include using more than two members selected at a time, and selecting the highest fitness

valued member with a certain probability. In the reproduction operators discussed so far all individuals are replaced each generation.

Another approach for selection is referred to as ranking where the individuals are sorted or ranked in ascending fitness values. The pool of individuals to take part in the reproduction is constructed as follows: The top 20% individuals are placed in the mating pool and duplicated. The bottom 20% are culled and do not take part in reproduction. The remainder of the mating pool comprises all the individuals that appear between the top 20% and the bottom 20% on the sorted list of fitness values.

The fitness function defined in section 4.2.5 reveals that the higher the fitness, the lower the corresponding MSE for an individual. In this thesis the 'Simple Genetic Algorithm' (SGA) of Goldberg is implemented [Goldberg 1989]. In the SGA the bottom 20% of each population with respect to fitness is culled. A two-point crossover is used with ranking as the selection operator, where the top 20% individuals with respect to fitness were added twice to the mating pool. The other 60% comprise all individuals between the top 20% and the bottom 20% of the sorted list. For crossover two individuals were randomly selected from the mating pool. Random mutation is used on the offspring. In this thesis, De Jong's guidelines were followed by using a relatively high crossover rate, a relatively low mutation rate and a moderately sized population [De Jong 1975]. On subsequent simulations the crossover rate is decreased, while the mutation and size of population are increased in order to find optimal values for these parameters.

4.2.9 Advantages of GAs

Genetic algorithms have several advantages, for example,

- they require no knowledge or gradient information about the error surface,
- they are generally not trapped by local optima,
- discontinuities on the error surface have little effect on overall optimization performance,
- GAs perform very well for large-scale optimization problems, and
- GAs can be used in a wide variety of optimization problems,

4.2.10 Disadvantages of GAs

While GAs do offer several advantages as mentioned above, they do have drawbacks, such as:

- they generally require *more fitness evaluations* compared to hill-climbing techniques,
- have trouble finding the *exact minimum*. GAs are best at reaching the global region but sometimes have difficulty reaching the exact optimum location. This problem can be overcome by a hybrid approach that uses a genetic algorithm to find the general area of a minimum followed by using gradient descent to find the corresponding minimum.
- finding a *suitable configuration* for a GA, using the various parameters and operators for GAs, is not straightforward.

The next section presents an overview of the Leapfrog Optimization Algorithm (LFOP) developed by Snyman [Snyman 1982b].

4.3 Leapfrog

Leapfrog is a gradient-based optimization approach based on the physical problem of the motion of a particle of unit mass in an n-dimensional conservative force field [Snyman 1982a, Snyman 1982b]. The potential energy of the particle in the force field is represented by the function to be minimized - in the case of NNs, the potential energy is the MSE. The objective is to conserve the total energy of the particle within the force field, where the total energy consists of the particle's potential and kinetic energy. The optimization method simulates the motion of the particle, and by monitoring the kinetic energy, an interfering strategy is adopted to appropriately reduce the potential energy. This algorithm employs an improved time step selection routine in which the time step is automatically reduced or increased to ensure an efficient utilization of the basic dynamic algorithm developed by Snyman [Snyman 1982b]. Snyman recorded results for leapfrog optimization that performed well compared to the conjugate gradient algorithm on the three test functions; Rosenbrock, the homogeneous quadratic and Oren's extended functions [Snyman 1982b]. Other researchers, Holm and Botha have shown that the leapfrog optimization algorithm is a robust algorithm for training summation unit neural networks [Holm *et al* 1999].

The algorithm is summarized below:

4.3.1 Leapfrog Algorithm

1. Compute an initial weight vector $\vec{w}_0$, with random components. Let $\Delta t = 0.5$, $\delta = 1$, $m = 3$, $\delta_1 = 0.001$ and $\epsilon = 10^{-5}$. Initialize $i = 0$, $j = 2$, $s = 0$, $p = 1$ and $k = -1$, where δ denotes the maximum allowable stepsize.

2. Compute the initial acceleration $\vec{a}_0 = -\nabla E(\vec{w}_0)$ and velocity $\vec{v}_0 = \frac{1}{2}\vec{a}_0\,\Delta t$, where $E(\vec{w}_0)$ is the MSE for weight vector $\vec{w}_0$.

3. Set $k = k + 1$ and compute $\|\Delta\vec{w}_k\| = \|\vec{v}_k\|\Delta t$.

4. If $\|\Delta\vec{w}_k\| < \delta$ goto step 5, otherwise set $\vec{v}_k = \frac{\delta\vec{v}_k}{\Delta t\|\vec{v}_k\|}$ and goto step 6.

5. Set $p = p + \delta_1$ and $\Delta t = p\Delta t$.

6. If $s < m$, goto step 7, otherwise set $\Delta t = \frac{\Delta t}{2}$ and $\vec{w}_k = \frac{\vec{w}_k + \vec{w}_{k-1}}{2}$, $\vec{v}_k = \frac{\vec{v}_k + \vec{v}_{k-1}}{4}$, $s = 0$ and goto 7.

7. Set $\vec{w}_{k+1} = \vec{w}_k + \vec{v}_k\Delta t$.

8. Compute $\vec{a}_{k+1} = -\nabla F(\vec{w}_{k+1})$ and $\vec{v}_{k+1} = \vec{v}_k + \vec{a}_{k+1}\Delta t$.

9. If $\vec{a}_{k+1}^T \cdot \vec{a}_k > 0$, then $s = 0$ and goto 10, otherwise $s = s + 1$, $p = 1$ and goto 10.

10. If $\|\vec{a}_{k+1}\| \leq \epsilon$ then stop, otherwise goto 11.

11. If $\|\vec{v}_{k+1}\| > \|\vec{v}_k\|$ then $i = 0$ and goto 3, otherwise $\vec{w}_{k+2} = \frac{\vec{w}_{k+1} + \vec{w}_k}{2}$, $i = i + 1$ and goto 12.

12. Perform a restart: If $i \leq j$, then $\vec{v}_{k+1} = \frac{\vec{v}_{k+1} + \vec{v}_k}{4}$ and $k = k + 1$, goto 8, otherwise $\vec{v}_{k+1} = 0$, $j = 1$, $k = k + 1$ and goto 8.

Whereas PSO and GA perform stochastic parallel searches, leapfrog uses gradient information to guide the search from one search point to the next.

4.4 Experimental Results

This section applies the optimization algorithms discussed in the previous section to the training of product unit neural networks. Results are also compared to that obtained from applying these optimization algorithms to SUNNs. The comparison also includes training PUNNs and SUNNs using gradient descent. The objective of these experiments is to determine if any gain in performance with respect to generalization

can be achieved by using PUs. First, the functions used in the experiments are summarized, followed by a description of the experimental procedure.

4.4.1 Test Functions

Eight functions, varying in complexity, were used. These functions are summarized below.

1. The quadratic function $f(z) = z^2$, with $z \sim U(-1,1)$. The training, test and validation sets consisted of 50 distinct randomly selected patterns.

2. The cubic function $f(z) = z^3 - 0.04z$, with $z \sim U(-1,1)$. The training, test and validation sets consisted of 50 distinct randomly selected patterns.

3. The henon time series $z_t = 1 + 0.3z_{t-2} - 1.4z_{t-1}^2$, with $z_1, z_2 \sim U(-1,1)$. The training, test and validation sets consisted of 200 distinct randomly selected patterns.

4. The surface $f(x,y) = y^7x^3 - 0.5x^6$, with $x, y \sim U(-1,1)$. The training, test and validation sets consisted of 300 distinct randomly selected patterns.

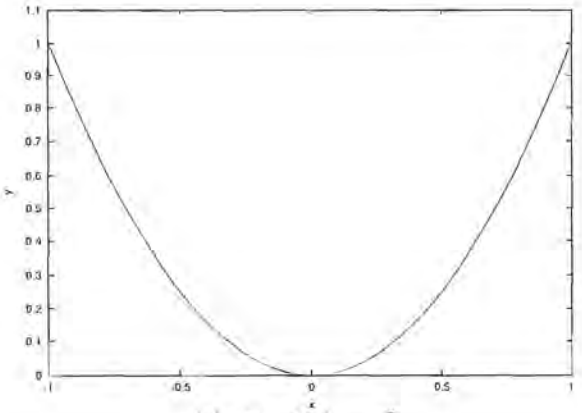
5. The paraboloid $f(x,y) = x^2 + y^2$, with $x, y \sim U(-2,2)$. The training, test and validation sets consisted of 300 distinct randomly selected patterns.

6. The function $f(x,y) = sin(x^2) + sin(y^2)$, with $x, y \sim U(-2,2)$. The training, test and validation sets consisted of 300 distinct randomly selected patterns.

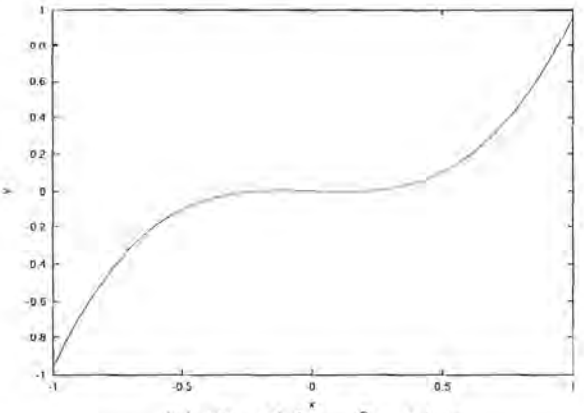
7. The camel function $f(x,y) = 4 - 2.1x^2 + \frac{x^3}{3}x^2 + x \cdot y + (4y^2 - 4)y^2$, with $x \sim U(0,10)$ and $y \sim U(0,10)$. The training, test and validation sets consisted of 500 distinct randomly selected patterns.

8. The function $f(x,y) = sin(x) \cdot sin(y) \cdot \sqrt{x \cdot y}$, with $x, y \sim U(0, 10)$. This function is also referred to as the 'graph' in this thesis. The training, test and validation sets consisted of 500 distinct randomly selected patterns.

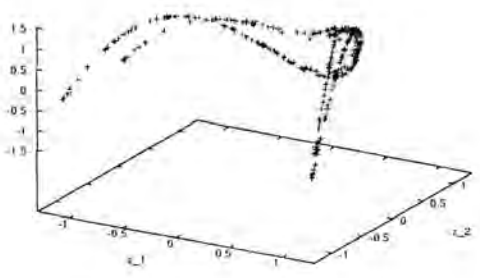
The graphs for the 8 test functions above are displayed in figures 4.1 and 4.2 on pages 100 and 101, respectively.



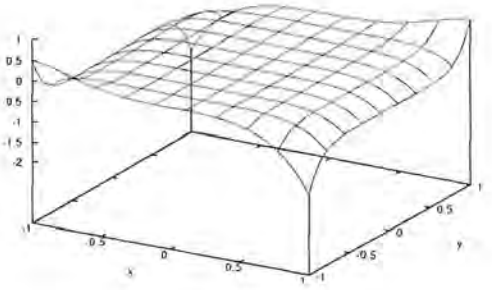
(a) F1: $f(x) = x^2$



(b) F2: $f(x) = x^3 - 0.04x$



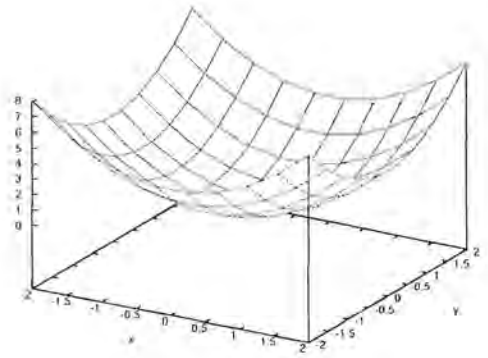
(c) F3: $z_t = 1 + 0.3z_{t-2} - 1.4z_{t-1}^2$



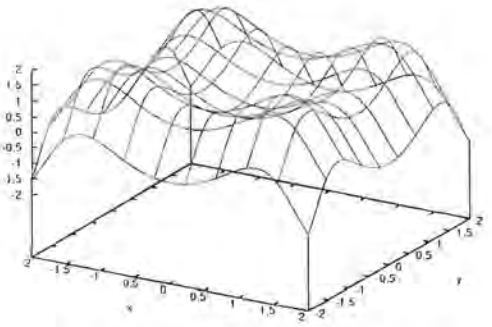
(d) F4: $f(x,y) = y^7x^3 - 0.5x^6$

Figure 4.1: Functions to be approximated

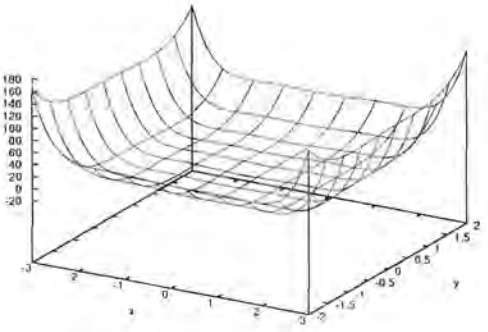
The next section discusses the performance criteria used for optimization of the algorithms.



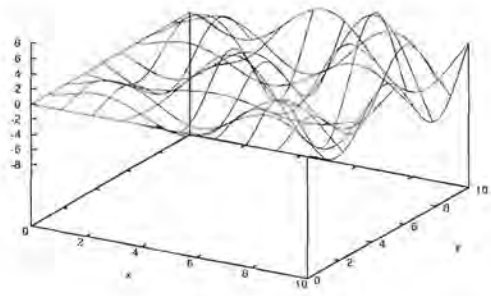
(e) F5: $f(x,y) = x^2 + y^2$



(f) F6: $f(x,y) = sin(x^2) + sin(y^2)$



(g) F7: $f(x,y) = (4 - 2.1x^2 + (\frac{x^3}{3}))x^2 + xy + (4y^2 - 4)y^2$



(h) F8: $f(x,y) = sin(x) \cdot sin(y) \cdot \sqrt{x \cdot y}$

Figure 4.2: Functions to be approximated

4.4.2 Performance Criteria

Each optimization algorithm contains a set of parameters that must be fine-tuned to improve convergence. Thus, the optimal parameters for each global optimization algorithm have to be determined before comparing the performance of the various optimization algorithms.

Three performance criteria were considered to determine the optimal PSO, GA, LFOP and BP for each function, namely,

1. the average of 30 simulations of the mean squared error (MSE) on the training and test sets after 500 training epochs for each of the eight functions;

2. the number of epochs required within a maximum of 1000 epochs to reach various generalization levels, i.e. 0.5, 0.1, 0.05, 0.01, 0.001, 0.0001 and 0.00001;

3. the number of simulations that converged for each of the generalization levels mentioned above.

The next section describes the experimental procedure applied to determine the optimal parameters for each optimization algorithm.

4.5 Experimental procedure

In determining the optimal parameters for each of the optimization algorithms, the following procedure was adhered to. All but one parameter were fixed, while the parameter that was not fixed (i.e. the one for which the optimal value has to be determined) assumed values from a range of possible values for that parameter. For each training session one value was selected from the range of values for the parameter under consideration. Training proceeded until all the values from the range were exhausted. A training session consisted of 30 simulations, where each simulation was trained for 500 epochs (for GD and LFOP), iterations (for PSO) or generations (for GA). In each simulation a different training set was used. The average MSE on the test set over the 30 simulations and the number of simulations that converged to a predefined generalization level were recorded. The parameter value that resulted in the lowest average MSE on the training and test sets and that had a high number of simulations that converged to a predefined generalization level was then selected as the best value for the parameter under consideration. This optimal value is then subsequently used in training to determine the optimal parameter values for the re-

maining parameters. The other remaining parameters are optimized in a similar fashion.

The next sections determines the optimal values of parameters for each of PSO, GA, LFOP and BP for each function.

4.5.1 Parameters for the Optimization Methods

Performance of each of the optimization algorithms used in this thesis is influenced by a number of parameters which should be optimized for each new problem. This section lists for each algorithm the set of parameters for which optimal values were found.

1. **PSO**

   Parameters that influence the performance of PSO include,

   (a) the inertia weight, which controls the balance between the global and local exploration abilities.

   (b) the maximum velocity, which limits the maximum jump that a particle can make in one step.

   (c) the acceleration constants, which control how far particles fly from one another.

   (d) the size of the population (i.e. number of particles in the swarm) affects the run-time of PSO; the larger the swarm, the longer the PSO will take to find a solution.

2. **Back-propagation (BP)**

   A number of factors influence the performance of back-propagation by gradient descent. These include,

   (a) the interval for initial weights, which influences the speed and accuracy with which BP will find the minimum.

(b) the learning rate, which controls the step sizes in the direction of the negative gradient of the error surface.

(c) the momentum, which smoothes out the oscillatory behaviour caused by the stochastic selection of training patterns for on-line learning.

3. **GA**

Convergence of genetic algorithms is influenced by the following three factors,

(a) probability for crossover, which determines how much genetic material will be exchanged between individuals. The higher the crossover rate, the greater the chance of convergence.

(b) probability for mutation, which determines the rate at which bits are mutated or flipped in a bit string. Convergence of a GA, generally requires a small rate for mutation.

(c) the size of the population; the larger the population, the greater the chance of convergence, but the longer it takes to find the solution.

4. **LFOP**

Parameters that influence the convergence of LFOP are,

(a) δ (the maximum allowable stepsize)

(b) δ_1

(c) Δt (the time step) (refer to section 4.3.1 on page 97).

(d) m specifies the number of steps before re-start. A value of 3 for m, worked well in practice.

4.6 Optimizing the Parameters

This section determines the optimal values for parameters for BP and each of the global optimization algorithms. Experimental results are presented to support decisions on which values to use.

4.6.1 Optimal Parameters for PSO

A range of values for the inertia weight, maximum velocity, acceleration constant and number of particles have been tested to find the best combination of parameter values for each experiment (these values are listed in tables 4.1 to 4.3). The acceleration constant in this thesis was implemented using one value to represent both acceleration constants c_1 and c_2 as suggested by Eberhart [Eberhart *et al* 1996]. The PUNNs were trained for each of the functions for a fixed number of epochs (500), where an epoch is one training pass through the training set. Training started with a swarm of 50 particles and parameters maximum velocity and acceleration constant, both initialized to 1.0. In all 8 functions, the weights (or particles) were initialized to random values in the range [-1, 1], also ensuring that approximately 50% of the particles had negative values.

The values for the parameters inertia weight, maximum velocity and acceleration constant appear below in tables 4.1, 4.2 and 4.3, respectively.

Parameter	Values								
Inertia weight	0.01	0.25	0.5	0.75	0.875	0.9	0.925	0.95	1.0

Table 4.1: Range of values for inertia weight for PSO

Parameter	Values							
Maximum velocity	0.0	0.5	1.0	1.5	2.0	2.5	5.0	10.0

Table 4.2: Range of values for maximum velocity for PSO

Parameter	Values							
Acceleration constant	0.0	0.25	0.5	0.75	1.0	1.5	1.75	2.0

Table 4.3: Range of values for acceleration constant for PSO

The optimal values for each of the parameters for the eight functions using PUs and SUs appear in tables 4.4 and 4.5, respectively.

Function	PSO Parameters (PU)			
	Inertia weight	Maximum velocity	Acceleration Constant	No of particles
$f(x) = x^2$	1.0	1.0	2.0	30
$f(x) = x^3 - 0.04x$	0.95	1.5	0.75	30
Henon	0.8	5.0	0.75	50
$f(x,y) = y^7x^3 - 0.5x^6$	0.75	10.0	1.5	50
$f(x,y) = x^2 + y^2$	0.9	2.0	1.0	50
$f(x,y) = sin(x^2) + sin(y^2)$	0.75	2.0	1.5	50
Camel	0.75	10.0	1.0	100
$f(x,y) = sin(x) \cdot sin(y) \cdot \sqrt{x \cdot y}$	0.5	1.0	1.5	100

Table 4.4: Best parameters for PSO using PUs

The following section determines the optimal parameters for BP for PUNNs and SUNNs.

Function	PSO Parameters (SU)			
	Inertia weight	Maximum velocity	Acceleration Constant	No of particles
$f(x) = x^2$	0.875	5.0	1.0	30
$f(x) = x^3 - 0.04x$	0.875	1.75	0.75	30
$f(x,y) = y^7x^3 - 0.5x^6$	0.75	10.0	1.0	50
Henon	1.0	1.5	0.75	50
$f(x,y) = x^2 + y^2$	0.875	1 5	1.0	50
$f(x,y) = sin(x^2) + sin(y^2)$	0.75	2.5	1.0	50
Camel	0.75	1.0	1.0	100
$f(x,y) = sin(x) \cdot sin(y) \cdot \sqrt{x \cdot y}$	0.75	1.5	1.75	100

Table 4.5: Best parameters for PSO using SUs

4.6.2 Optimal Parameters for BP

The same procedure for determining the optimal parameters in PSO, was also applied to BP, using various initial weight initialization, learning rate and momentum values. The range of values tested for weight initialization appear in table 4.6 on page 108. Various intervals were considered ranging from values close to zero to intervals that contain larger initial values such as [2.0, 4.0], since research by Leerink *et al* suggested larger values for weight initialization [Leerink *et al* 1995]. During weight selection for SUNN and PUNN, it was ensured that approximately 50% of the weights were negative. Tables 4.7 and 4.8 contain the range of values tested for the learning rate and momentum, respectively. The number of simulations of BP applied to PUNNs were increased to 70 to compensate for the high number of simulations that resulted in overflows, due to exponentiation in the learning rule of gradient descent, while the number of simulations for SUNNs remained at 30. In the case of PUNN using gradient descent, optimal parameter values for functions F3, F4, F5, F6, F7 and F8 could not be established, since none of the simulations returned a result, other than overflows.

Parameter	Intervals					
Weight interval	$[-1.0, 1.0]$	$[-1.5, 1.5]$	$[-2.0, 2.0]$	$[-3.0, 3.0]$	$[-4.0, 4.0]$	$[-1.0, -0.5]$ & $[0.5, 1.0]$
	$[-1.5, -0.5]$ & $[0.5, 1, 5]$	$[-2.0, -0.5]$ & $[0.5, 2.0]$	$[-2.0, -1.0]$ & $[1.0, 2.0]$	$[-2.5, -1.0]$ & $[1.0, 2.5]$	$[-3.0, -1.0]$ & $[1.0, 3.0]$	$[-2.0, -1.5]$ & $[1.5, 2.0]$
	$[-3.0, -1.5]$ & $[1.5, 3.0]$	$[-3.5, -1.5]$ & $[1.5, 3.5]$	$[-3.0, -2.0]$ & $[2.0, 3.0]$	$[-4.0, -2.0]$ & $[2.0, 4.0]$	$[-4.0, -3.0]$ & $[3.0, 4.0]$	$[-5.0, -3.0]$ & $[3.0, 5.0]$

Table 4.6: Intervals for initial weights for BP

Parameter	Values											
Learning rate	0.001	0.01	0.025	0.0275	0.05	0.075	0.1	0.15	0.2	0.25	0.5	0.75

Table 4.7: Range of values for learning rate for BP

A possible explanation for this behaviour is that the weights selected from the initial interval are too far from the optimal weights, causing too large jumps in weight space. This already illustrates the failure of GD to train PUs. Tables 4.9 and 4.10 on pages 109 and 109 contain the optimal values for the parameters for BP applied to product and summation unit networks, respectively. In tables 4.9 and 4.10, the notation $\pm[1.5, 3.5]$ is an abbreviation for the intervals $[-3.5, -1.5]$ and $[1.5, 3.5]$.

The next section determines the optimal parameters for GAs applied to SUNNs and PUNNs.

Parameter	Values											
Momentum	0.0	0.25	0.5	0.6	0.7	0.8	0.9	1.0	1.25	1.5	2.0	5.0

Table 4.8: Range of values for momentum for BP

Backpropagation (SU)			
Function	**weight interval**	**learning rate**	**momentum**
$f(x) = x^2$	$\pm[1.5, 3.5]$	0.01	0.7
$f(x) = x^3 - 0.04x$	$\pm[1.5, 3.0]$	0.0275	0.5

Table 4.9: Best parameters for BP using PUs

Backpropagation (SU)			
Function	**weight interval**	**learning rate**	**momentum**
$f(x) = x^2$	$\pm[0.5, 1.5]$	0.25	2.0
$f(x) = x^3 - 0.04x$	$\pm[0, 3.0]$	0.5	2.0
Henon	$\pm[0, 1.5]$	0.5	1.25
$f(x, y) = y^7x^3 - 0.5x^6$	$\pm[1.5, 3.5]$	0.5	2.0
$f(x, y) = x^2 + y^2$	$\pm[0.5, 1.0]$	0.25	0.7
$f(x, y) = sin(x^2) + sin(y^2)$	$\pm[0.5, 2.0]$	0.15	1.0
Camel	$\pm[0.5, 2.0]$	0.25	1.5
$f(x, y) = sin(x) \cdot sin(y) \cdot \sqrt{xy}$	$\pm[0.5, 1.0]$	0.5	0.9

Table 4.10: Best parameters for BP using SUs

Probability of crossover	0.5	0.6	0.7	0.8	0.9

Table 4.11: Range of values for crossover

Probability mutation	0.001	0.005	0.01	0.05	0.1	0.25	0.5

Table 4.12: Range of values for mutation

4.6.3 Optimal parameters for GA

In determining the optimal parameter values for the GA, 30 simulations were used. Various initial values were used for the probability of crossover and mutation in the GA algorithm. The number of individuals was also increased to determine the optimal size of the population. The range of values for mutation and crossover appear in tables 4.11 and 4.12 follow the guidelines suggested by De Jong [De Jong 1975]. In determining the optimal population size, the number of individuals were gradually increased from 50 to 200 during training.

The optimal values for GAs using SUNNs and PUNNs appear in tables 4.13 and 4.14 on page 111.

The optimal parameters for leapfrog optimization are determined in the next section.

4.6.4 Optimal parameters for LFOP

In determining the best parameters, the average MSE of 30 runs was used during training. During each training session the parameters were fixed to values that appear

Genetic Algorithm - PU			
Function	Probability of crossover	Probability of mutation	Size of population
$f(x) = x^2$	0.6	0.01	50
$f(x) = x^3 - 0.04x$	0.6	0.01	50
Henon	0.7	0.005	100
$f(x,y) = y^7x^3 - 0.5x^6$	0.6	0.01	100
$f(x,y) = x^2 + y^2$	0.7	0.005	120
$f(x,y) = sin(x^2) + sin(y^2)$	0.6	0.005	120
camel	0.8	0.005	200
$f(x,y) = sin(x) \cdot sin(y) \cdot \sqrt{x \cdot y}$	0.7	0.005	200

Table 4.13: Best parameters for GA using PUs

Genetic Algorithm - SU			
Function	Probability of crossover	Probability of mutation	Size of population
$f(x) = x^2$	0.5	0.005	50
$f)(x) = x^3 - 0.04x$	0.8	0.005	50
Henon	0.7	0.005	100
$f(x,y) = y^7x^3 - 0.5x^6$	0.8	0.005	100
$f(x,y) = x^2 + y^2$	0.5	0.005	120
$f(x,y) = sin(x^2) + sin(y^2)$	0.8	0.001	120
camel	0.8	0.001	200
$f(x,y) = sin(x) \cdot sin(y) \cdot \sqrt{x \cdot y}$	0.5	0.001	200

Table 4.14: Best parameters for GA using SUs

Parameter	Values		
δ	10	100	1000

Table 4.15: Range of values for δ

Parameter	Values		
δ_1	0.001	0.01	0.1

Table 4.16: Range of values for δ_1

in tables 4.15 to 4.17.

The optimal values for LFOP applied to PUNNs and SUNNs appear in tables 4.18 and 4.19 respectively. An entry of '$-$' in table 4.18 indicates that all simulations produced overflows and optimal values could not be determined in these cases. The overflows can again be ascribed to the fact that gradient methods suffer from an explosion in the growth of weight values due to large derivatives.

The next section summarizes the initial NN architectures used in training each function using SUNNs and PUNNs.

4.7 Initial Neural Network Architectures

Tables 4.20 and 4.21 contain the initial neural network architectures that were used in optimizing the parameters for each of PSO, GA, LFOP and BP using PUNNs and SUNNs. The oversized networks of tables 4.20 and 4.21 are used to determine the

Parameter	Values			
Δt	0.01	0.05	0.075	0.1

Table 4.17: Range of values for Δt

Leapfrog Algorithm - PU			
Function	δ	δ_1	Δt
$f(x) = x^2$	10.0	0.01	0.05
$f(x) = x^3 - 0.04x$	100.0	0.01	0.01
Henon	10.0	0.01	0.05
$f(x,y) = y^7x^3 - 0.5x^6$	10.0	0.001	0.01
$f(x,y) = x^2 + y^2$	10.0	0.01	0.01
$f(x,y) = sin(x^2) + sin(y^2)$	–	–	–
camel	–	–	–
$f(x,y) = sin(x) \cdot sin(y) \cdot \sqrt{x \cdot y}$	–	–	–

Table 4.18: Best parameters for LFOP using PUs

Leapfrog Algorithm - SU			
Function	δ	δ_1	Δt
$f(x) = x^2$	100.0	0.001	0.075
$f(x) = x^3 - 0.04x$	10.0	0.001	0.01
Henon	100.0	0.001	0.05
$f(x,y) = y^7x^3 - 0.5x^6$	10.0	0.001	0.01
$f(x,y) = x^2 + y^2$	10.0	0.001	0.075
$f(x,y) = sin(x^2) + sin(y^2)$	100.0	0.001	0.01
camel	100.0	0.001	0.1
$f(x,y) = sin(x) \cdot sin(y) \cdot \sqrt{x \cdot y}$	100.0	0.001	0.075

Table 4.19: Best parameters for LFOP using SUs

Configuration for PUNNs	
Function	**Initial Configuration**
$f(x) = x^2$	1 : 4 : 1
$f(x) = x^3 - 0.04x$	1 : 5 : 1
Henon	2 : 6 : 1
$f(x,y) = y^7x^3 - 0.5x^6$	2 : 5 : 1
$f(x,y) = x^2 + y^2$	2 : 4 : 1
$f(x,y) = sin(x^2) + sin(y^2)$	2 : 10 : 1
Camel	2 : 10 : 1
$f(x,y) = sin(x) \cdot sin(y) \cdot \sqrt{x \cdot y}$	2 : 12 : 1

Table 4.20: Initial configuration for PUNNs

optimal parameters for the different global optimization algorithms.

The next section discusses the procedure to determine the optimal number of hidden units for SUNNs and PUNNs.

4.8 Best Configuration for SUNNs and PUNNs

Once the optimal parameters for each of the global optimization algorithms and BP have been determined, pruning by 'brute force' is then applied to the optimization algorithms using the optimal parameters of section 4.6 to find near optimal configurations for PUNNs and SUNNs. The optimal configurations for PUNNs and SUNNs were determined by comparing results of experiments where the number of hidden units varied for each training session. The architecture that produced the smallest average MSE on the test set over 30 simulations and the highest number of simulations that converged to a specified generalization level of 0.001 was accepted as the best configuration. Table 4.22 contains the results for PUNNs, and table 4.23 for SUNNs, where training started with the initial configuration tabulated. After each training

Configuration for SUNNs	
Function	**Initial Configuration**
$f(x) = x^2$	1 : 4 : 1
$f(x) = x^3 - 0.04x$	1 : 5 : 1
Henon	2 : 8 : 1
$f(x,y) = y^7x^3 - 0.5x^6$	2 : 10 : 1
$f(x,y) = x^2 + y^2$	2 : 6 : 1
$f(x,y) = sin(x^2) + sin(y^2)$	2 : 10 : 1
Camel	2 : 12 : 1
$f(x,y) = sin(x) \cdot sin(y) \cdot \sqrt{x \cdot y}$	2 : 15 : 1

Table 4.21: Initial configuration for SUNNs

session, consisting of 30 simulations, the number of hidden units was decreased by 1 and training was re-started on this smaller network until the performance on the test set deteriorated by 20% or more compared to the initial oversized network. The number of hidden units that occurred in optimal PUNNs expressed as a percentage of the number of hidden units that occurred in the equivalent optimal SUNNs, are for functions F1 50%, F2 33.3%, F3 80%, F4 33.3%, F5 50%, F6 66.7%, F7 75% and F8 77.8%.

The optimal configurations or architectures for each function obtained in this section is subsequently used in the remainder of this thesis in experiments that compare the various global optimization algorithms and BP. In chapter 5 the variance nullity pruning algorithm is applied to the oversized PUNNs to determine optimal architectures. The results obtained in chapter 5 will then be compared to the results obtained by brute force pruning.

Configuration for PUNNs		
Function	**Initial Configuration**	**Best Configuration**
$f(x) = x^2$	1 : 4 : 1	1 : 1 : 1
$f(x) = x^3 - 0.04x$	1 : 5 : 1	1 : 1 : 1
Henon	2 : 6 : 1	1 : 4 : 1
$f(x, y) = y^7x^3 - 0.5x^6$	2 : 5 : 1	1 : 2 : 1
$f(x, y) = x^2 + y^2$	2 : 4 : 1	2 : 2 : 1
$f(x, y) = sin(x^2) + sin(y^2)$	2 : 10 : 1	2 : 4 : 1
Camel	2 : 10 : 1	2 : 6 : 1
$f(x, y) = sin(x) \cdot sin(y) \cdot \sqrt{x \cdot y}$	2 : 12 : 1	2 : 7 : 1

Table 4.22: Configuration for PUNNs

Configuration for SUNNs		
Function	**Initial Configuration**	**Best Configuration**
$f(x) = x^2$	1 : 4 : 1	1 : 2 : 1
$f(x) = x^3 - 0.04x$	1 : 5 : 1	1 : 3 : 1
Henon	2 : 8 : 1	1 : 5 : 1
$f(x, y) = y^7x^3 - 0.5x^6$	2 : 10 : 1	1 : 6 : 1
$f(x, y) = x^2 + y^2$	2 : 6 : 1	2 : 4 : 1
$f(x, y) = sin(x^2) + sin(y^2)$	2 : 10 : 1	2 : 6 : 1
Camel	2 : 12 : 1	2 : 8 : 1
$f(x, y) = sin(x) \cdot sin(y) \cdot \sqrt{x \cdot y}$	2 : 15 : 1	2 : 9 : 1

Table 4.23: Configuration for SUNNs

4.9 Comparison Between PUNNs Containing Bias and Distortion Units

The MSEs of two PUNNs, one containing a bias and the other a distortion unit in the hidden layer, are compared in this section. Both architectures contained a bias in the output layer. The objective is to determine whether there is any gain in performance when using a distortion unit compared to a bias unit in the hidden layer of a PUNN. PSO was used to compare the resulting MSEs of these two architectures of PUNNs. First, optimal parameters were determined for both type of architectures (similarly to section 4.6.1 on page 105) for the following three functions: $f(x) = x^3 - 0.04x$, $f(x,y) = y^7x^3 - 0.5x^6$ and $f(x,y) = sin(x) \cdot sin(y) \cdot \sqrt{x \cdot y}$. Subsequently, PSO with optimal parameter values for each type of PUNN architecture (i.e. bias or distortion unit PUNN) was trained to approximate each of the three functions for a maximum of 500 epochs. The average MSEs on the training and test sets over 30 simulations together with a 95% confidence interval for the three functions are displayed in Table 4.24.

The results of the tests show that PUNNs with a distortion unit produced smaller MSEs on the training set and generalized much better than PUNNs that contain a bias in the hidden layer. The PUNN with distortion unit is therefor chosen as the PUNN architecture for implementation in the remainder of this thesis.

Function	PUNN using a bias unit		PUNN using a distortion unit	
	MSE on Training set	MSE on Test set	MSE on Training set	MSE on Test set
$f(x) = x^3 - 0.04x$	0.002970 ±0.001128	0.002785 ±0.000967	0.000018 ±0.0000043	0.000016 ±0.0000042
$f(x,y) = y^7x^3 - 0.5x^6$	0.002574 ±0.000777	0.002840 ±0.00079	0.000919 ±0.000476	0.001213 ±0.000625
$f(x,y) = sin(x) \cdot sin(y) \cdot \sqrt{x \cdot y}$	0.002383 ±0.001665	0.002614 ±0.001841	0.0005684 ±0.0004004	0.0007953 ±0.0005696

Table 4.24: Comparison of MSEs on PUNN containing a bias and a PUNN containing a distortion unit

4.10 Comparison of Global Optimization Algorithms

This section uses the optimized parameters from section 4.6.1 to compare the performance of PSO, GA, LFOP and BP on both PUNNs and SUNNs.

Tables 4.25 and 4.26 summarize the average mean squared error (MSE) for the training and test sets for PUNNs and SUNNs, respectively, after 500 training passes for each of the algorithms, PSO, LFOP, GA and BP, together with 95% confidence intervals. A '$\star$' in tables 4.25 and 4.26 indicates the algorithm that performed the best in each case of PUNNs and SUNNs, respectively. A '†' in tables 4.25 and 4.26 indicates the algorithm that performed the best in both, PUNNs and SUNNs, cases.

Figures 4.3, 4.4 and 4.5 illustrate the learning profiles for each optimization method for the training and test sets (as a measure of generalization). Tables 4.27 to 4.29 list the average number of epochs to reach specified MSE levels on the training set. The entries in the first column for the tables 4.27 to 4.29 refer to (a) the type of algorithm and (b) the type of network used, i.e. BP:SU refers to back-propagation by gradient descent applied to a summation unit neural network. Similarly, LFOP:PU refers to leapfrog optimization applied to a product unit neural network. Tables 4.30 to 4.33 summarize, for different generalization levels (i.e. the MSE on the test set), the percentage of simulations that did converge to these generalization levels. A '$-$' entry in table 4.25 implies that not a single simulation out of the 30 simulations produced any result other than overflows. In tables 4.27 to 4.29, a '$-$' entry means that not a single simulation reached convergence within the maximum of 1000 epochs allowed. In tables 4.32 and 4.33, a '$-$' indicates that all simulations produced overflows.

The fact that only the quadratic and cubic functions produced results for BP using product units, is an indication of the difficulty that is associated with backpropagation by gradient descent when applied to PUs. LFOP for training PUs, a derivative based algorithm, also produced overflows for the functions $f(x,y) = sin(x^2) + sin(y^2)$, $f(x,y) = sin(x) \cdot sin(y) \cdot \sqrt{x \cdot y}$ and the camel function as reflected in table 4.32.

4.11 Analysis of Results

The results in this section were produced by experiments that used optimal architectures as determined in section 4.8 for both PUNNs and SUNNs. Table 4.25 indicates that PSO produced better MSEs on both the training and test sets in training PUNNs compared to the other global optimization algorithms. GAs also performed fairly well, but not as good as PSO. In table 4.26 LFOP using SUs produced the smallest MSEs on both the training and test sets, except for $f(x) = x^2$ and $f(x,y) = x^2 + y^2$, where BP using SUs produced much better results. Thus, in training SUNNs, LFOP would be the recommended optimization algorithm.

Next, the results for each function are discussed separately.

$f(x) = x^2$

The graphs in figure 4.3 show that PSO using PUs and GA using PUs started off with fairly low MSEs on both training and test sets. All the global optimization algorithms, (i.e. PSO, LFOP and GA) produced substantially better training errors and generalization than the equivalent SUs for the quadratic function as shown in tables 4.25 and 4.26. BP:SUs (read as back-propagation using summation units) performed slightly better than BP:PUs (read as back-propagation using PUs). However, the

Function	Algorithm	Average Mean Squared Error	
		Training set	Test set
$f(x) = x^2$	BP	0.006823 ± 0.004255	0.005949 ± 0.003679
	PSO	0.000344 ± 0.000112	0.000334 ± 0.000112 ⋆ †
	GA	0.000518 ± 0.000518	0.001340 ± 0.002131
	LFOP	0.001583 ± 0.000725	0.001930 ± 0.000893
$f(x) = x^3 - 0.04x$	BP	0.001210 ± 0.000696	0.000977 ± 0.000510
	PSO	0.000018 ± 0.0000043	0.000016 ± 0.0000042 ⋆ †
	GA	0.000072 ± 0.0000528	0.000082 ± 0.0000585
	LFOP	0.000122 ± 0.000141	0.000156 ± 0.0001863
Henon	BP	–	–
	PSO	0.003173 ± 0.001754	0.007881 ± 0.004455
	GA	0.004365 ± 0.002613	0.006101 ± 0.004645
	LFOP	0.000651 ± 0.000698	0.000608 ± 0.000651 ⋆
$f(x,y) = y^7x^3 - 0.5x^6$	BP	–	–
	PSO	0.000919 ± 0.000476	0.001213 ± 0.000625 ⋆
	GA	0.0021079 ± 0.0007105	0.0022844 ± 0.0007620
	LFOP	0.0043720 ± 0.0005151	0.0049928 ± 0.0004130
$f(x,y) = x^2 + y^2$	BP	–	–
	PSO	0.0088368 ± 0.0028955	0.0083125 ± 0.0027094 ⋆
	GA	0.0094581 ± 0.0025222	0.0089599 ± 0.0023852
	LFOP	0.0200444 ± 0.0054125	0.0195222 ± 0.0050495
$f(x,y) = sin(x^2) + sin(y^2)$	BP	–	–
	PSO	0.0021190 ± 0.0009562	0.0041067 ± 0.0031883 ⋆ †
	GA	0.005998 ± 0.002610	0.005807 ± 0.002668
	LFOP	–	–
camel	BP	–	–
	PSO	0.0316965 ± 0.0026735	0.0398755 ± 0.0032916 ⋆
	GA	0.0509293 ± 0.0037854	0.0632008 ± 0.0059496
	LFOP	–	–
$f(x,y) = sin(x) \cdot sin(y) \cdot \sqrt{x \cdot y}$	BP	–	–
	PSO	0.0005684 ± 0.0004004	0.0007953 ± 0.0005696 ⋆ †
	GA	0.0007359 ± 0.0002295	0.0009474 ± 0.0002870
	LFOP	–	–

Table 4.25: Mean squared error results for PUs

Function	Algorithm	Average Mean Squared Error	
		Training set	Test set
$f(x) = x^2$	BP	0.001477 ± 0.001323	0.001434 ± 0.001023 ⋆
	PSO	0.001945 ± 0.00172	0.001873 ± 0.001355
	GA	0.005696 ± 0.002512	0.006965 ± 0.003288
	LFOP	0.006117 ± 0.003559	0.009518 ± 0.005372
$f(x) = x^3 - 0.04x$	BP	0.001695 ± 0.000498	0.001957 ± 0.000501
	PSO	0.000095 ± 0.0000254	0.000165 ± 0.0000496
	GA	0.000410 ± 0.0002135	0.000525 ± 0.0002425
	LFOP	0.000053 ± 0.0000177	0.000065 ± 0.0000197 ⋆
Henon	BP	0.000800 ± 0.000773	0.000926 ± 0.000810
	PSO	0.0004200 ± 0.0000490	0.0004050 ± 0.0000450
	GA	0.0016431 ± 0.0003998	0.0017022 ± 0.0003907
	LFOP	0.0002004 ± 0.0000232	0.0001943 ± 0.0000231 ⋆ †
$f(x, y) = y^7x^3 - 0.5x^6$	BP	0.000493 ± 0.000061	0.002172 ± 0.000637
	PSO	0.001086 ± 0.000089	0.003604 ± 0.000953
	GA	0.0014528 ± 0.0001069	0.0030766 ± 0.0005732
	LFOP	0.0003959 ± 0.0000482	0.0007645 ± 0.0000765 ⋆ †
$f(x, y) = x^2 + y^2$	BP	0.000413 ± 0.000013	0.000529 ± 0.000019 ⋆ †
	PSO	0.001860 ± 0.000776	0.002164 ± 0.000753
	GA	0.0096575 ± 0.0020188	0.0143728 ± 0.0029755
	LFOP	0.0010507 ± 0.0010435	0.0012780 ± 0.0012015
$f(x, y) = sin(x^2) + sin(y^2)$	BP	0.005595 ± 0.000883	0.006722 ± 0.001068
	PSO	0.008542 ± 0.000572	0.011924 ± 0.000804
	GA	0.012365 ± 0.001019	0.014279 ± 0.001087
	LFOP	0.004477 ± 0.001194	0.005483 ± 0.001184 ⋆
camel	BP	0.000560 ± 0.000090	0.002256 ± 0.000702
	PSO	0.001228 ± 0.000217	0.002044 ± 0.000344
	GA	0.005017 ± 0.000917	0.006842 ± 0.000809
	LFOP	0.0000535 ± 0.0000022	0.0000963 ± 0.0000058 ⋆ †
$f(x, y) = sin(x) \cdot sin(y) \cdot \sqrt{x \cdot y}$	BP	0.008764 ± 0.000144942	0.010274 ± 0.000189
	PSO	0.010991 ± 0.000368	0.012457 ± 0.000309
	GA	0.0111821 ± 0.0003309	0.0125407 ± 0.0002641
	LFOP	0.004335 ± 0.000975	0.005601 ± 0.001115 ⋆

Table 4.26: Mean squared error results for SUs

PUNN used a smaller architecture; the SUNN consisted of 2 hidden units, whereas the PUNN contained only 1 hidden unit. Table 4.27 shows that PSO:PUs, GA:PUs and LFOP:PUs required substantially fewer epochs than its SU equivalents to reach each of the generalization levels, 0.01 and 0.001, with LFOP:PUs having the least number of epochs for each of the generalization levels. Not one of the global optimization algorithms using SUs could generalize up to a level of 0.0001; these algorithms could only manage to generalize up to a level of 0.001, with convergence observed in 46.7% of the simulations of PSO:SUs. All algorithms using PUs were able to reach an MSE of 0.0001 on the training sets as shown in tables 4.30 to 4.33. For the quadratic function, GA:PUs and LFOP:PUs managed to generalize up to a low level of 0.00001 with LFOP:PUs having the highest percentage of simulations (47.1%) that converged to this low level followed by GA where 40% of the simulations converged. The average number of simulations required by LFOP:PUs and GA:PUs to reach this level of generalization are 90.1 and 301.5, respectively. Tables 4.25 and 4.26 show that GA using PUs and LFOP using SUs, overfitted the data, as indicated by the MSE on the test set compared to the MSE on the training set. In both cases the MSE on the test set exceeded the MSE on the training set, indicating overfitting of the data. Thus, PSO:PUs is the recommended algorithm for training the quadratic function.

$f(x) = x^3 - 0.04x$

The graphs in figure 4.3 show that GA:PUs started off with very small MSEs on both training and test sets. Once, again all the global optimization algorithms produced a smaller MSE on the training set than its SU equivalent, except for LFOP. The optimal architecture of PUNNs for this function contained 1 hidden unit compared to the 3 hidden units in the case of SUNNs. LFOP using SUs with the larger network than its PUs equivalent had a much lower MSE on the training and test sets than LFOP:PUs as shown in tables 4.25 and 4.26. Table 4.26 indicates that all the algorithms that used

SUs overfitted the data. Tables 4.25 and 4.26, further show that BP:PUs produced results similar to BP:SUs, with BP:PUs being slightly better than BP:SUs. BP:SUs exhibits a generalization that is much lower than the MSE on the training set. It should however also be borne in mind that the PUNN contained 1 hidden unit and is much smaller than the SUNN with 3 hidden units. Table 4.27 shows that PSO:PUs, GA:PUs and LFOP:PUs required fewer epochs than its SU equivalents to reach the generalization level of 0.001, 0.0001 and 0.00001, with LFOP:PUs having the least number of epochs (314.7 ± 64.03) to reach 0.00001. Not one of the global optimization algorithms using SUs could generalize up to a level of 0.00001. Global optimization algorithms using SUNNs could only manage to generalize up to 0.0001, with 33.3% of the simulations of PSO converging to this low generalization level. The results of the cubic function using GA with PUs are significantly better than GA using SUs as reflected in Tables 4.25 and 4.26. PSO:PUs produced the best training and test error compared to all the other algorithms including SUNNs. It is interesting to note that although LFOP:PUs has a larger number of simulations that converged to a level of 0.00001 as indicated in table 4.32, it has an average MSE much greater than PSO:PUs. PSO:PUs is thus recommended for training the function $f(x) = x^3 - 0.04x$.

$f(z) = 1 + 0.3z_{t-2} - 1.4z_{t-1}^2$ (henon)

The graphs in 4.3 reflect fairly low MSEs for LFOP using PUs and BP using SUs on both training and test sets early in training. The global optimization algorithms applied to the SUs outperformed the PUs as indicated in tables 4.25 and 4.26, with the results of the optimization algorithms of SUs having a much smaller variance than its PU equivalents. LFOP:SUs had the lowest training error and generalization than the optimization algorithms applied to PUs. Note that LFOP:SUs had a larger network (2:5:1) with a training error of 0.0002004 ± 0.0000232, compared to LFOP:PUs, 2:4:1 network that produced a training error of 0.000651 ± 0.000698. Interestingly, In the case

of the SUs the global optimization algorithms PSO:SUs and LFOP:SUs gave better training and generalization than BP:SUs (which is a local optimizer). BP:PUs did not produce a single result, except for overflows (as indicated to by '-' in table 4.31), in all the training sessions. LFOP using PUs required the least number of epochs of all the optimization algorithms to reach the various generalization levels for the henon time series. The LFOP:PUs as shown in table 4.27 used much less epochs than LFOP:SUs to reach generalization levels 0.01, 0.001, and 0.0001. Only LFOP:PUs and PSO:PUs managed to reach low generalization levels of 0.00001, with convergence reached in only 1.4% of the simulations in LFOP:PUs. In PSO:PUs slightly more simulations (3.3%) converged than LFOP:PUs at this low level of generalization. However, LFOP:PUs only needed 63.3 epochs compared to 465.3 epochs required by PSO:PUs to reach this low level. For a generalization of 0.0001, 35.7% of simulations for LFOP:PUs converged compared to only 10% convergence in the case of PSO:PUs. In the case of SUNNs only the LFOP managed to reach a level of 0.0001, with convergence in 13.3% of the simulations. PSO and GA using PUs overfitted the data as reflected in table 4.25. LFOP:SUs, with a low training and test error (see table 4.25) is recommended as training algorithm for the henon time series.

$f(x, y) = y^7 x^3 - 0.5x^6$

PSO using PUs produced small MSEs on the training set but overfitted the data as shown by MSEs on the test set in table 4.25. Similarly, BP:SUs produced low training errors but did not generalize equally well. GAs:SUs performance was similar to GA:PUs. LFOP:SUs produced the best MSEs on the training and test sets of all the algorithms. LFOP:SUs, however, could not reach a generalization level of 0.0001 (all 30 simulations ended in MSEs on the test sets between 0.0002 and 0.0007), whereas 13.3% simulations of PSO:PUs achieved 0.00001 (refer to table 4.30). PSO using PUs is the only algorithm that managed to reach a generalization level of 0.00001 as

shown in table 4.30. The lowest generalization level achieved by SUNNs is 0.001; with convergence of 63.3% and 6.7% of simulations for LFOP and BP, respectively. Also, did PSO:PUs and GA:PUs require fewer epochs than its SU equivalent to reach a generalization level of 0.001. BP:PUs did not produce any results, other than overflows in all training sessions. LFOP:SUs with its much lower MSEs on training and test sets, as tabulated in 4.26 is recommended as the optimization algorithm for the function $f(x, y) = y^7x^3 - 0.5x^6$.

$f(x, y) = x^2 + y^2$

PSO:PUs reached lower generalization levels than PSO:SUs as indicated in table 4.28. Table 4.26 shows that BP:SUs produced the smallest MSEs on the training and test sets, but could only generalize up to a level of 0.001, despite the fact that the generalization in table 4.26 is 0.000529 ± 0.000019; the average of all simulations is 0.000529 with not a single simulation reaching an MSE on the test set lower than 0.0001 (all simulations ended with values greater than 0.0001 but smaller than 0.0009). However, both PSO:PUs and LFOP:PUs managed to reach lower generalization levels than BP:SUs as shown in table 4.28. The lowest generalization level achieved by SUNNs is 0.001, with 16.7% of the simulations of BP:SUs and 13.3% of the simulations of PSO:SUs converging at this low level. However, all the global optimization algorithms using PUs managed to reach generalization levels of 0.001 and 0.0001, with 33.3% of PSO:PUs and 15.7% of LFOP:PUs generalizing up to a level of 0.00001. PSO:PUs had twice as many simulations than LFOP:PUs that converged to this low level of generalization as shown in tables 4.30 and 4.32. Of all the global optimization algorithms, LFOP:SUs, not forgetting the bigger network (i.e. 2:4:1) than the equivalent PUNN (2:2:1), produced the lowest MSE on the training and test sets within the allowed 500 epochs as reflected in table 4.25. LFOP:SUs, however, could not manage to reach a generalization level lower than 0.001 within the 1000 epochs allowed. Neither

BP:PUs nor LFOP:PUs produced any results. BP:PUs, with the smallest MSE average as reflected in table 4.26, is recommended for training the function $f(x,y) = x^2+y^2$.

$f(x,y) = sin(x^2) + sin(y^2)$

The graphs in figure 4.4 show that PSO using PUs produced large MSEs early in training, but eventually had the lowest MSEs of all the algorithms when training terminated. In this case PSO:PUs produced the lowest training error as indicated in table 4.25. GA:PUs and LFOP:SUs produced similar training errors. Table 4.28 shows that only PSO:PUs and LFOP:SUs managed to reach generalization levels of 0.0001, with PSO:PUs taking fewer epochs to achieve this generalization level. In both cases only 3.3% simulations converged. BP:PUs and LFOP:PUs did not produce any results. PSO:PUs with its good generalization ability is recommended as global optimization algorithm for the function $f(x,y) = sin(x^2) + sin(y^2)$.

$f(x,y) = 4 - 2.1x^2 + (\frac{x^3}{3})x^2 + xy + (4y^2 - 4)y^2$(camel)

The graphs in figure 4.5 show that all algorithms using SUs had small MSEs early in training and that not one of the algorithms using PUs managed to reach MSE levels lower than 0.03. The SUs performed much better than the PUs in this case. The LFOP:SUs produced the smallest MSEs on the training and test sets. LFOP:SUs was able to achieve a generalization level of 0.0001 (refer to table 4.29). BP:SUs also produced very good MSEs but indicated overfitting of the data in table 4.25. BP:PUs and LFOP:PUs did not produce any results, other than overflows in all the training sessions. LFOP:SUs is recommended for training of the camel function.

$f(x,y) = sin(x)sin(y)\sqrt{x \cdot y}$

The graphs in figure 4.5 clearly show that PSO using PUs had the lowest MSEs early in training and ended the training session with the lowest MSEs on both the training

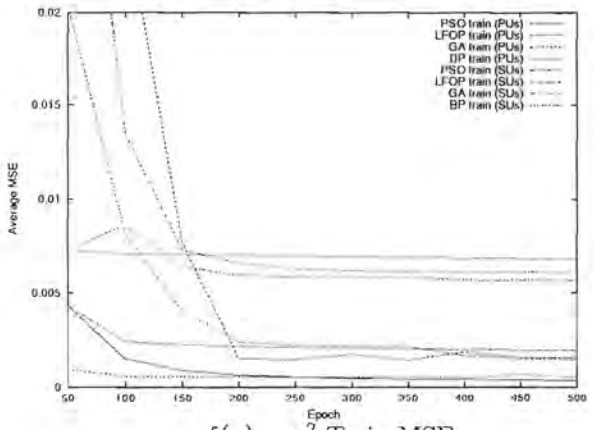
and test sets. PSO:PUs and GA:PUs outperformed all the other algorithms. PSO:PUs is the only algorithm that managed to generalize up to a level of 0.00001, GA:PUs could only generalize up to 0.001 (with MSEs on the test set ranging between 0.0002 and 0.0006) simulations. The lowest generalization level achieved by SUNNs is 0.01, with 15.7% of simulations converging at this level. In the case of SUNNs, LFOP:SUs BP:PUs and LFOP:PUs did not produce any results, other than overflows. Only PSO:PUs managed to generalize up to level 0.00001 (refer to table 4.29), with only 3.3% simulations converging as reflected in table 4.30.

Table 4.27 shows that LFOP:PUs had the most simulations that converged to the different generalization levels, especially for the low generalization level of 0.00001. BP:SUs did manage to have simulations that converged to a generalization level up to 0.001 but not a single simulation converged to a MSE on the test set lower than 0.0001. Figures 4.3 to 4.5 illustrate that PSO:PUs and GA:PUs have larger reductions in error early in training reaching low errors using substantially less training epochs. LFOP:PUs has shown to use much less epochs than do PSO and GA for low generalization levels of 0.001, 0.0001 and 0.00001.

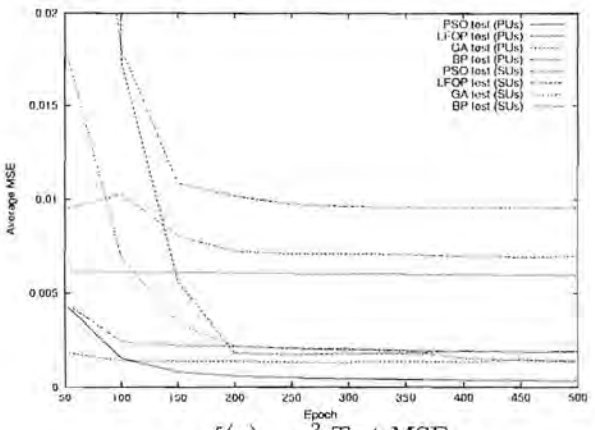
4.12 Conclusion

PUNNs compared favourably with SUNNS with respect to functions F1, F2, F6 and F8. However, SUNNs performed much better on functions F3, F4, F5 and F7. Generally, LFOP using SUs, produced a much smaller training error than BP:SUs. LFOP:SUs also generalized far better than BP:SUs. LFOP:SUs produced much smaller MSEs than BP:SUs in training the test functions, except for functions $f(x) = x^2$ and $f(x, y) = x^2 + y^2$, where BP:SUs outperformed PSO:SUs and LFOP:SUs. Although

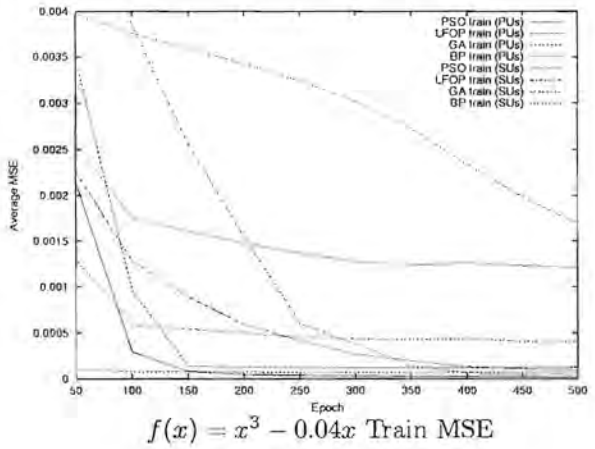
the global optimization algorithms did not perform better than BP:SUs in all cases, it did however manage to achieve lower generalization levels using much fewer epochs than BP:SUs with a corresponding higher convergence than BP:SUs. Thus, global optimization algorithms tend to find the best minimum on the error surface faster than BP:SUs. PSO:PUs is the only algorithm that managed to reach a low generalization level of 0.0001 in all functions except for the camel function. Although, BP:SUs applied to function F5 had a smaller training error than PSO:PUs, it did not manage to reach the low generalization level of 0.0001, that was achieved by PSO:PUs. The results also show that global optimization algorithms can reach lower generalization levels than BP when applied to SUNNs. The tests have also indicated that PUNNs are not always an improvement over SUNNs, even though PUs may produce smaller networks. These smaller networks do not always produce good training errors and generalization compared to the slightly bigger SUNNs. However, global optimization using SUs showed an improvement in performance compared to back-propagation. In certain cases the PUNNs ($f(x,y) = x^2$, $f(x,y) = x^3 - 0.04x$, $f(x,y) = sin(x^2) + sin(y^2)$ and $f(x,y) = sin(x) \cdot sin(y) \cdot \sqrt{(x \cdot y)}$) outperformed the SUNNs. PSO appears to be more robust with respect to functions F1, F2 and F5 since they have a larger percentage of simulations that converged to a generalization level of 0.00001 than LFOP:PUs. In general, PUNNs did not show a remarkable gain in performance, other than reaching lower generalization levels faster than back-propagation.



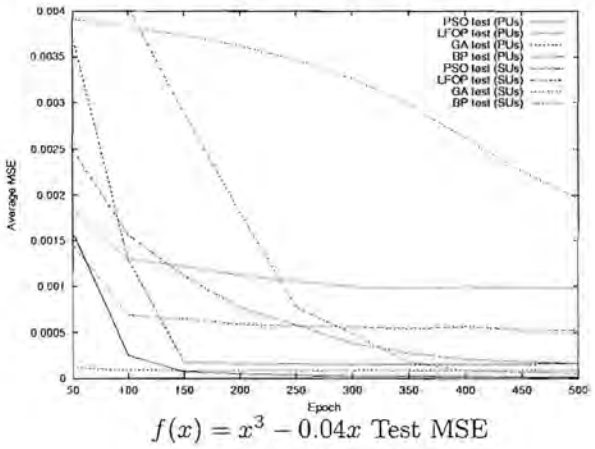
$f(x) = x^2$ Train MSE



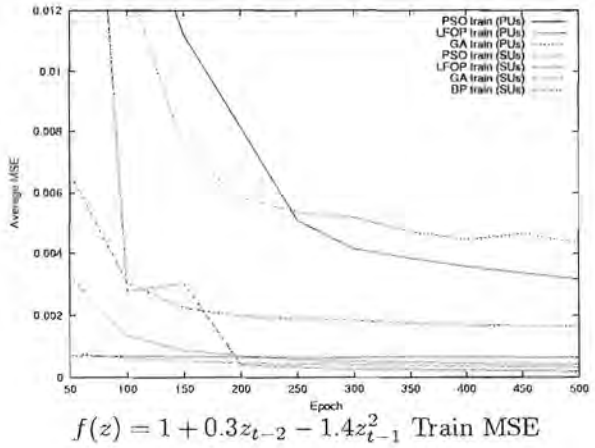
$f(x) = x^2$ Test MSE



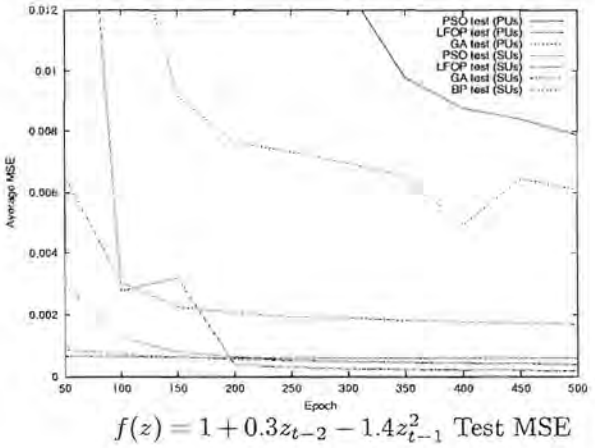
$f(x) = x^3 - 0.04x$ Train MSE



$f(x) = x^3 - 0.04x$ Test MSE

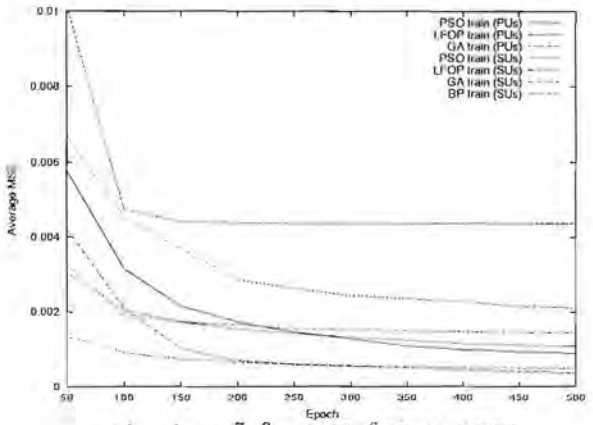


$f(z) = 1 + 0.3z_{t-2} - 1.4z_{t-1}^2$ Train MSE

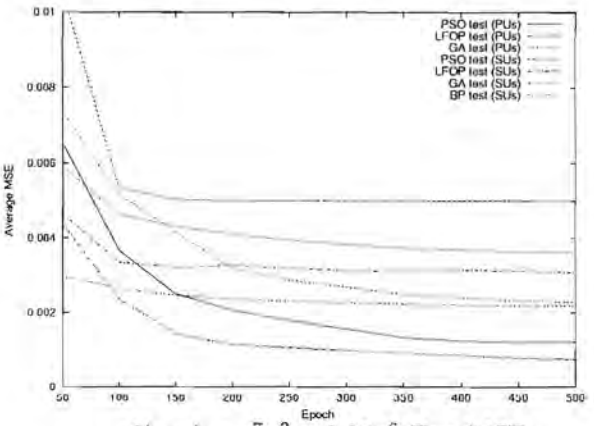


$f(z) = 1 + 0.3z_{t-2} - 1.4z_{t-1}^2$ Test MSE

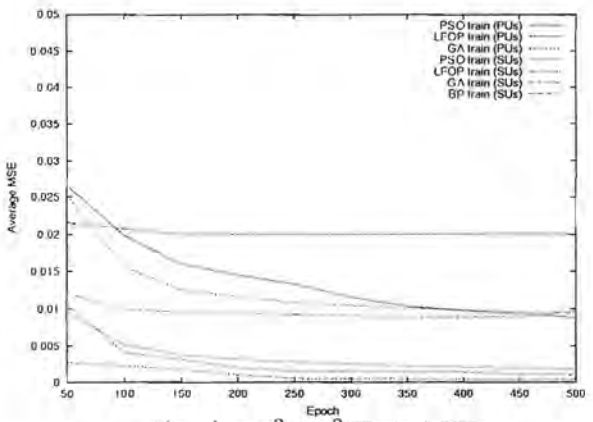
Figure 4.3: Learning profiles for functions F1, F2 and F3



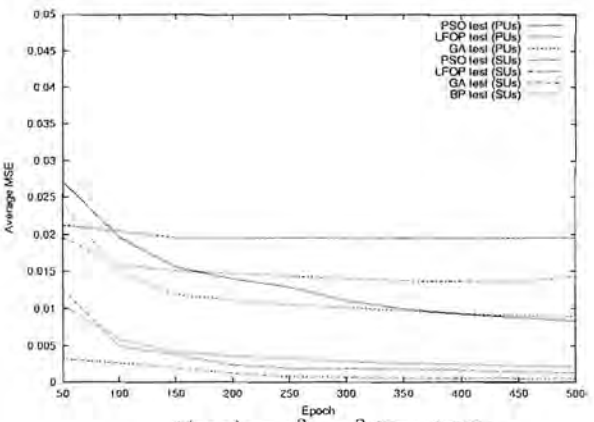
$f(x,y) = y^7x^3 - 0.05x^6$ Train MSE



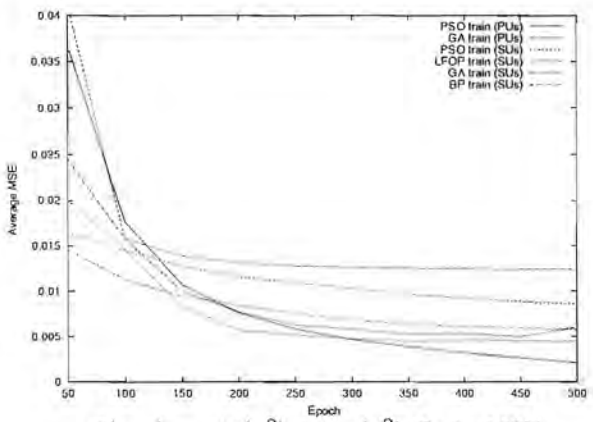
$f(x,y) = y^7x^3 - 0.05x^6$ Test MSE



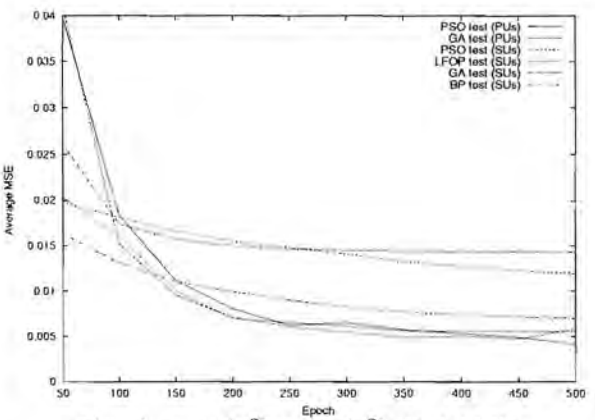
$f(x,y) = x^2 + y^2$ Train MSE



$f(x,y) = x^2 + y^2$ Test MSE

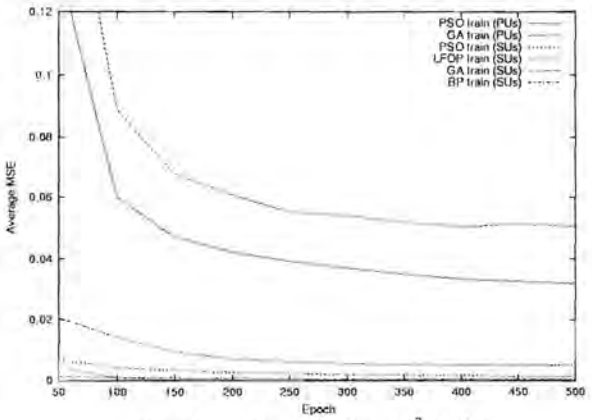


$f(x,y) = sin(x^2) + sin(y^2)$ Train MSE

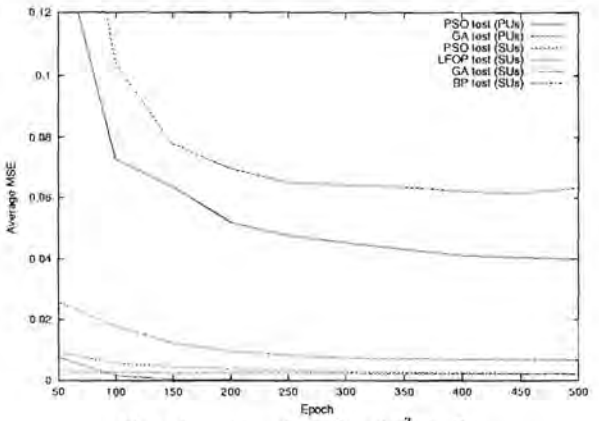


$f(x,y) = sin(x^2) + sin(y^2)$ Test MSE

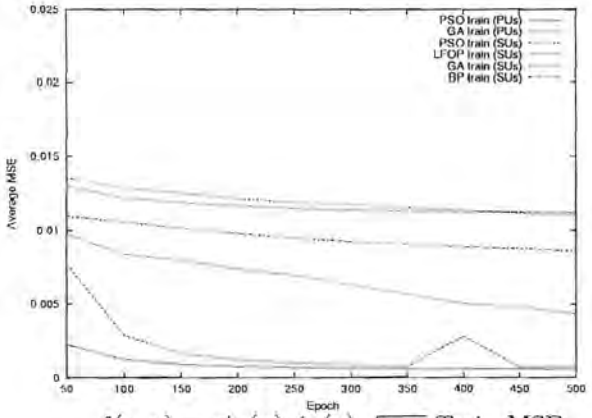
Figure 4.4: Learning profiles for functions F4, F5 and F6



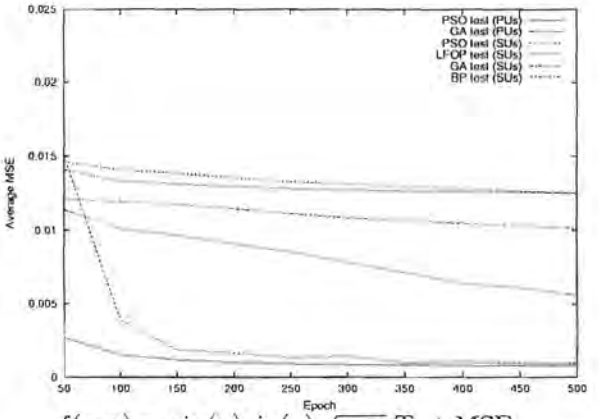
$f(x,y) = (4 - 2.1x^2 + (\frac{x^3}{3}))x^2 +$
$xy + (4y^2 - 4)y^2$ Train MSE



$f(x,y) = (4 - 2.1x^2 + (\frac{x^3}{3}))x^2 +$
$xy + (4y^2 - 4)y^2$ Test MSE



$f(x,y) = sin(x)sin(y)\sqrt{x \cdot y}$ Train MSE



$f(x,y) = sin(x)sin(y)\sqrt{x \cdot y}$ Test MSE

Figure 4.5: Learning profiles for functions F7 and F8

$f(x) = x^2$							
MSE	0.5	0.1	0.05	0.01	0.001	0.0001	0.00001
BP:SU	1.0 ± 0	1.0 ± 0.07	6.8 ± 4.00	108.5 ± 32.61	227.6 ± 53.72	–	–
BP:PU	2.2 ± 0.10	3.9 ± 0.48	17.8 ± 23.06	101.7 ± 52.95	216.2 ± 76.33	216.2 ± 76.33	395.4 ± 58.01
PSO:SU	1.0 ± 0	1.0 ± 0	1.4 ± 0.36	101.7 ± 58.73	228.5 ± 62.69	–	–
PSO:PU	1.0 ± 0	1.0 ± 0	1.3 ± 0.25	38 ± 19.84	154.5 ± 44.09	485.6 ± 28.16	–
GA:SU	1.0 ± 0	1.0 ± 0	1.2 ± 0.27	40.0 ± 31.85	494.9 ± 9.93	–	–
GA:PU	1.1 ± 0.09	1.2 ± 0.17	2 ± 0.57	8.8 ± 2.15	95.4 ± 58.81	211.3 ± 77.02	301.5 ± 74.58
LFOP:SU	2.4 ± 0.41	7.4 ± 1.64	9.6 ± 5.60	192.6 ± 99.35	273 ± 53.80	–	–
LFOP:PU	5.0 ± 0.69	10.3 ± 1.45	13.7 ± 1.69	78.4 ± 46.30	143.3 ± 62.80	173.9 ± 59.92	90.1 ± 40.11

$f(x) = x^3 - 0.04x$							
MSE	0.5	0.1	0.05	0.01	0.001	0.0001	0.00001
BP:SU	1.0 ± 0	1.0 ± 0.00	1.0 ± 0.00	6.3 ± 0.52	448.6 ± 22.51	–	–
BP:PU	1.9 ± 0.09	2.8 ± 0.31	48.8 ± 49.45	36.8 ± 29.00	224.5 ± 56.78	377.5 ± 50.54	408.5 ± 50.46
PSO:SU	1.0 ± 0	1.0 ± 0	1.1 ± 0.13	7.0 ± 0.55	144.7 ± 38.43	330.4 ± 50.85	–
PSO:PU	1.0 ± 0	1.0 ± 0	1.7 ± 0.43	16 ± 3.32	63.1 ± 12.01	138.4 ± 30.03	422.5 ± 40.42
GA:SU	1.0 ± 0	1.0 ± 0	1.0 ± 0	5.1 ± 1.50	91.1 ± 49.06	397.0 ± 64.84	–
GA:PU	1 ± 0	1.4 ± 0.26	2.8 ± 0.48	11.1 ± 2.81	105.8 ± 92.27	91.3 ± 50.08	485.5 ± 28.48
LFOP:SU	5.4 ± 1.12	16.7 ± 7.31	18.0 ± 3.94	43.3 ± 7.94	185 ± 11.17	310.7 ± 25.15	–
LFOP:PU	15.3 ± 2.19	22.4 ± 1.81	24.1 ± 2.15	30.8 ± 2.36	166.6 ± 68.53	159.6 ± 58.87	314.7 ± 64.03

$z_t = 1 + 0.3z_{t-2} - 1.4z_{t-1}^2$ (henon)							
MSE	0.5	0.1	0.05	0.01	0.001	0.0001	0.00001
BP:SU	30.5 ± 3.26	53.2 ± 10.46	63.0 ± 6.68	85.0 ± 12.56	123.2 ± 58.01	152.5 ± 68.35	–
BP:PU	–	–	–	–	–	–	–
PSO:SU	1.0 ± 0	1.0 ± 0	1.0 ± 0	26.6 ± 4.84	147.9 ± 37.05	–	–
PSO:PU	2.4 ± 1.26	31.9 ± 4.27	83.9 ± 64.31	116 ± 49.06	233.6 ± 58.26	371.3 ± 53.32	465.3 ± 31.72
GA:SU	1.0 ± 0	1.0 ± 0	1.2 ± 0.18	31.8 ± 3.76	436.2 ± 47.46	–	–
GA:PU	5.1 ± 0.77	21.9 ± 3.66	45.4 ± 13.22	109.2 ± 32.05	357.1 ± 64.38	–	–
LFOP:SU	2.9 ± 0.51	15.3 ± 3.74	21.6 ± 5.28	73.0 ± 3.64	112 ± 8.70	480.4 ± 23.54	–
LFOP:PU	3.8 ± 0.53	15.2 ± 4.43	15.3 ± 2.35	38.4 ± 29.29	32.2 ± 2.38	69.8 ± 41.89	63.3 ± 28.65

Table 4.27: Epochs needed to reach MSE levels

$f(x,y) = y^7x^3 - 0.5x^6$							
MSE	**0.5**	**0.1**	**0.05**	**0.01**	**0.001**	**0.0001**	**0.00001**
BP:SU	1.0 ± 0	1.1 ± 0.13	1.1 ± 0.11	7.6 ± 10.92	–	–	–
BP:PU	1.0 ± 0	1.0 ± 0	1.0 ± 0	1.2 ± 0.33	–	–	–
PSO:SU	1.0 ± 0	1.0 ± 0	1.0 ± 0	1.2 ± 0.33	580.1 ± 102.99	–	–
PSO:PU	1.0 ± 0	1.5 ± 0.58	3.7 ± 1.59	25.2 ± 3.64	394 ± 121.32	783.3 ± 90.39	904.5 ± 81.34
GA:SU	1.1 ± 0.2	2.2 ± 0.6	$63. \pm 1.8$	54.8 ± 13.9	924.5 ± 78.60	–	–
GA:PU	3.5 ± 0.86	6.8 ± 1.01	7.8 ± 1.27	20.3 ± 2.11	279.1 ± 100.29	994.8 ± 8.88	–
LFOP:SU	1.8 ± 0.33	4.0 ± 0.43	4.9 ± 0.33	10.1 ± 1.27	473.2 ± 32.71	–	–
LFOP:PU	9.2 ± 1.84	18.7 ± 1.62	21.8 ± 0.95	37.5 ± 4.02	–	–	–

$f(x,y) = x^2 + y^2$							
MSE	**0.5**	**0.1**	**0.05**	**0.01**	**0.001**	**0.0001**	**0.00001**
BP:SU	1.0 ± 0	1.0 ± 0.00	1.0 ± 0.00	$16.23 \pm 2,68$	98.9 ± 40.68	–	–
BP:PU	–	–	–	–	–	–	–
PSO:SU	1.0 ± 0	1.0 ± 0	1.0 ± 0	56.1 ± 20.72	766.0 ± 154.22	–	–
PSO:PU	1.2 ± 0.24	3.0 ± 0.51	14.3 ± 2.28	543 ± 153.78	786.2 ± 110.16	821.0 ± 95.12	864.4 ± 86.07
GA:SU	1.0 ± 0	1.0 ± 0	1.1 ± 0.20	182.3 ± 67.47	–	–	–
GA:PU	4.1 ± 0.75	12.4 ± 1.19	19.2 ± 2.02	277.6 ± 77.15	480.8 ± 26.14	896.3 ± 59.61	–
LFOP:SU	30.9 ± 7.23	0.5 ± 0.35	1.6 ± 0.53	153.7 ± 55.90	-	–	–
LFOP:PU	1.7 ± 0.14	2.7 ± 0.53	95.3 ± 98.01	511.9 ± 124.63	850.6 ± 90.63	822.6 ± 95.57	818.1 ± 97.62

$f(x,y) = sin(x^2) + sin(y^2)$							
MSE	**0.5**	**0.1**	**0.05**	**0.01**	**0.001**	**0.0001**	**0.00001**
BP:SU	1.0 ± 0	1.0 ± 0.00	1.0 ± 0.00	195.8 ± 49.32	–		–
BP:PU	–	–	–	–	–	–	–
PSO:SU	1.0 ± 0	1.0 ± 0	1.0 ± 0	638.3 ± 177.60	–	–	–
PSO:PU	2.7 ± 1.18	17.4 ± 2.38	33.9 ± 6.61	171 ± 28.58	720.5 ± 118.47	927.2 ± 56.35	–
GA:SU	1.0 ± 0	1.0 ± 0	12.6 ± 3.52	870.6 ± 107.36	–	–	–
GA:PU	14.1 ± 1.03	28.2 ± 1.98	39 ± 5.01	180 ± 48.28	452 ± 33.52	–	–
LFOP:SU	1.0 ± 0	11.9 ± 6.09	24.8 ± 6.72	121.5 ± 4.90	981 ± 19.97	970.8 ± 55.30	–
LFOP:PU	–	–	–	–	–	–	–

Table 4.28: Epochs needed to reach MSE levels

$f(x,y) = (4 - 2.1x^2 + (\frac{x^3}{3}))x^2 + xy + (4y^2 - 4)y^2$							
MSE	**0.5**	**0.1**	**0.05**	**0.01**	**0.001**	**0.0001**	**0.00001**
BP:SU	1.0 ± 0	1.0 ± 0.07	1.1 ± 0.09	6.5 ± 1.52	75.5 ± 7.27	–	–
BP:PU	–	–	–	–	–	–	–
PSO:SU	1.0 ± 0	1.0 ± 0	1.0 ± 0	31.5 ± 5.37	–	–	–
PSO:PU	26.2 ± 1.68	66.9 ± 10.15	154.6 ± 61.23	–	–	–	–
GA:SU	1.0 ± 0	1.0 ± 0	1.0 ± 0.07	214.1 ± 97.56	–	–	–
GA:PU	19.6 ± 1.92	77.7 ± 12.82	302.6 ± 96.03	–	–	–	–
LFOP:SU	1.0 ± 0	0.7 ± 0.47	0.7 ± 0.31	30.7 ± 3.18	89 ± 5.86	247.8 ± 28.7	–
LFOP:PU	–	–	–	–	–	–	–

$f(x,y) = sin(x) \cdot sin(y)\sqrt{x \cdot y}$							
MSE	**0.5**	**0,1**	**0.05**	**0.01**	**0.001**	**0.0001**	**0.00001**
BP:SU	1.0 ± 0	1.0 ± 0.00	1.0 ± 0.00	161.4 ± 27.37	–	–	–
BP:PU	–	–	–	–	–	–	–
PSO:SU	1.0 ± 0	1.0 ± 0	1.0 ± 0	–	–	–	–
PSO:PU	1.4 ± 0.26	4.0 ± 0.82	5.9 ± 0.79	16 ± 1.63	199.8 ± 88.82	761.3 ± 123.76	991.2 ± 17.18
GA:SU	1.0 ± 0	1.0 ± 0.07	1.0 ± 0	–	–	–	–
GA:PU	10 ± 1.15	16.7 ± 1.64	19 ± 1.56	35.1 ± 2.07	277.3 ± 103.85	–	–
LFOP:SU	1.0 ± 0	0.6 ± 0.38	3.2 ± 1.68	–	–	–	–
LFOP:PU	–	–	–	–	–	–	–

Table 4.29: Epochs needed to reach MSE levels

PSO								
Function	Unit Type	Generalization levels (MSE)						
		0.5	0.1	0.05	0.01	0.001	0.0001	0.00001
x^2	SU	100.0%	100.0%	100.0%	63.3%	46.7%	0.0%	0.0%
	PU	100.0%	100.0%	100.0%	96.7%	86.7%	13.3%	0.0%
$x^3 - 0.04x$	SU	100.0%	100.0%	100.0%	76.7%	53.3%	33.3%	0.0%
	PU	100.0%	96.7%	93.3%	90.0%	90.0%	90.0%	53.3%
Henon	SU	100.0%	100.0%	100.0%	93.3%	86.7%	0.0 %	0.0%
	PU	83.3%	80.0%	60.0%	53.3%	30.0 %	10.0%	3.3%
$y^7x^3 - 0.5x^6$	SU	100.0%	100.0%	100.0%	66.7 %	0.0%	0.0%	0.0%
	PU	100.0%	93.3%	93.3%	56.7%	20.0%	20.0 %	13.3%
$x^2 + y^2$	SU	100.0%	100.0%	100.0 %	33.3%	13.3%	0.0%	0.0%
	PU	100.0%	76.7%	70.0%	40.0%	30.0%	20.0%	33.3%
$sin(x^2) + sin(y^2)$	SU	100.0%	100.0%	100.0%	16.7%	0.0%	0.0%	0.0 %
	PU	96.7%	83.3%	66.7%	53.3%	16.7%	3.3%	0.0%
camel	SU	100.0%	100.0%	100.0%	16.7%	0.0%	0.0%	0.0%
	PU	70.0%	46.7%	10.0%	0.0%	0.0 %	0.0%	0.0%
graph	SU	100.0%	100.0 %	100.0%	0.0%	0.0%	0.0%	0.0%
	PU	96.7%	96.7%	90.0%	50.0%	6.7%	3.3%	3.3 %

Table 4.30: Percentage simulations that converged to MSE levels

GA								
Function	Unit Type	Generalization levels (MSE)						
		0.5	0.1	0.05	0.01	0.001	0.0001	0.00001
x^2	SU	100.0%	100.0%	86.7%	63.3%	0.0%	0.0%	0.0%
	PU	100.00%	96.7%	93.3%	90.0%	66.7%	50.0%	40.0%
$x^3 - 0.04x$	SU	100.00%	100.0%	100.0%	80.0%	66.7%	13.3%	0.0
	PU	100.0%	96.7%	93.3%	83.3%	76.7%	70.0%	6.7%
Henon	SU	100.0%	100.0%	100.0%	93.3%	13.3%	0.0%	0.0%
	PU	93.3%	90.0%	83.3%	70.0 %	30.0%	0.0%	0.0%
$y^7x^3 - 0.5x^6$	SU	100.0%	100.0%	100.0%	100.0%	0.0 %	0.0%	0.0%
	PU	96.7%	80.0%	76.7%	50.0 %	43.3%	3.3%	0.0%
$x^2 + y^2$	SU	100.0%	100.0%	86.7%	20.0%	0.0%	0.0 %	0.0%
	PU	90.0%	83.3%	80.0%	60.0%	13.3%	6.7%	0.0%
$sin(x^2) + sin(y^2)$	SU	100.0%	96.7%	60.0%	3.3%	0.0%	0.0%	0.0%
	PU	90.0%	83.3%	80.0%	76.7%	23.3%	0.0 %	0.0%
camel	SU	100.0%	100.0%	100.0%	3.3%	0.0%	0.0%	0.0%
	PU	70.0%	36.7%	10.0%	3.3%	0.0%	0.0%	0.0%
graph	SU	100.0%	100.0%	100.0%	0.0%	0.0%	0.0%	0.0%
	PU	63.3%	60.0%	56.7%	50.0%	50.0%	0.0%	0.0%

Table 4.31: Percentage simulations that converged to MSE levels

LFOP								
Function	Unit Type	Generalization levels (MSE)						
		0.5	0.1	0.05	0.01	0.001	0.0001	0.00001
x^2	SU	100.0%	100.0%	90.0%	56.7%	33.3 %	0.0%	0.0%
	PU	62.9%	58.6%	48.6%	42.9%	41.4%	42.9%	47.1%
$x^3 - 0.04x$	SU	100.0%	100.0%	96.7%	93.3%	46.7%	26.7%	0.0%
	PU	94.3%	94.3%	93.3%	90.0%	84.3%	72.9 %	72.9%
Henon	SU	100.0%	100.0%	100.0%	76.7%	53.3%	13.3%	0.0%
	PU	75.7%	48.6%	44.3%	44.3%	37.1%	35.7%	1.4%
$y^7x^3 - 0.5x^6$	SU	100.0%	100.0%	100.0%	93.3%	63.3%	0.0 %	0.0%
	PU	91.4%	90.0%	78.6%	57.1%	0.0%	0.0%	0.0%
$x^2 + y^2$	SU	100.0%	90.0%	86.7%	6.7%	0.0%	0.0%	0.0%
	PU	98.6%	98.6%	87.1%	35.7%	12.9%	15.7 %	15.7%
$sin(x^2) + sin(y^2)$	SU	100.0%	90.0%	80.0%	13.3%	3.3%%	3.3%	0.0%
	PU	–	–	–	–	–	–	–
camel	SU	100.0%	96.7%	86.7%	16.7 %	10.0%	6.7%	0.0%
	PU	–	–	–	–	–	–	–
graph	SU	100.0%	90.0%	86.7%	15.7%	0.0%	0.0%	0.0%
	PU	–	–	–	–	–	–	–

Table 4.32: Percentage simulations that converged to MSE levels

BP								
Function	Unit Type	Generalization levels (MSE)						
		0.5	0.1	0.05	0.01	0.001	0.0001	0.00001
x^2	SU	93.3%	86.7%	33.3%	10.0%	6.7%	0.0%	0.0%
	PU	100.0%	97.6%	97.6%	78.7 %	27.8%	13.8%	0.0%
$x^3 - 0.04x$	SU	90.0%	93.3%	83.3%	13.3%	10.0%	0.0%	0.0%
	PU	100.0%	100.0%	87.9%	67.2%	41.3%	17.0%	11.4%
Henon	SU	96.7%	96.7 %	93.3%	93.3%	63.3%	0.0%	0.0%
	PU	–	–	–	–	–	–	–
$y^7x^3 - 0.5x^6$	SU	100.0%	100.0%	100.0%	96.7%	6.7%	0.0%	0.0%
	PU	–	–	–	–	–	–	–
$x^2 + y^2$	SU	100.0%	100.0%	100.0%	83.3%	16.7%	0.0%	0.0%
	PU	–	–	–	–	–	–	–
$sin(x^2) + sin(y^2)$	SU	100.0%	100.0%	100.0%	0.0%	0.0%	0.0%	0.0%
	PU	–	–	–	–	–	–	–
camel	SU	100.0%	100.0%	100.0%	33.3%	10.0%	0.0%	0.0%
	PU	–	–	–	–	–	–	–
graph	SU	100.0%	100.0%	100.0 %	0.0%	0.0%	0.0%	0.0%
	PU	–	–	–	–	–	–	–

Table 4.33: Percentage simulations that converged to MSE levels

Chapter 5

Architecture selection

Architecture selection is critical to NN modeling where the objective is to find the smallest network that accurately maps the true function described by the training data. Architecture selection has to reduce network complexity while maintaining good generalization. A network that is too large may lead to overfitting of the training data resulting in poor generalization when presented with similar but slightly different data. If the network is too small, underfitting may occur that results in poor approximation of the function [Baum *et al* 1989, Le Cun 1989].

The objectives of pruning are usually motivated by two aims: to obtain networks of a *small size* and with a *good generalization performance*. The objective is to find a minimal network topology. It is usually not obvious what the smallest network with the best generalization is for a particular task. Different approaches have been devised to solve this problem.

Architecture selection approaches can be grouped in four categories, i.e. brute-force approaches, regularization, network growing (network construction) and pruning. While these topics have been introduced in section 2.17, this chapter focuses on pruning.

Pruning starts training with a neural network which is expected to be big enough to ensure successful training. At convergence of the oversized network, weights and/or units that are irrelevant or redundant are removed, upon which the pruned network is retrained [Thimm *et al* 1995]. If the retraining converges then the removal-retraining cycle is resumed. If the retraining fails, the smallest network that satisfied the convergence criterion is assumed to have the most suitable topology for the given data set. The decision to prune a network is based on some measure of parameter (i.e. a unit or weight) relevance. A relevance is computed for each parameter and a pruning heuristic is applied to determine whether a parameter is irrelevant or not. Numerous pruning algorithms have been proposed, including the following,

- The Smallest Variance, $(min(\sigma))$, method of Sietsma and Dow that removed connections with smallest contribution variance on the training set, where the contribution of a connection is the value available to the connection from the lower layer, multiplied by its weight. The mean output of the removed connection is then added to the corresponding bias [Sietsma *et al* 1991].

- Skeletonization, which is a weight removal method, defines a measure of the relevance of a unit as the error when the unit is removed from the network, minus the error when the unit is left in the network [Mozer *et al* 1989]. This is accomplished by multiplying the output of a unit j by a coefficient, α_j, that represents the attentional strength of the unit [Mozer *et al* 1989]. In the case of hidden units,

$$o_k = f(\sum_{j}^{J+1} w_{kj}\alpha_j y_j) \tag{5.1}$$

where $f(\cdot)$ is the activation function, o_k is the activation of output unit O_k, w_{kj} is the weight between hidden unit Y_i and output unit O_k and y_j is the activation of hidden unit Y_j. If $\alpha_j = 0$, unit Y_j has no influence on the rest of the network. If $\alpha_j = 1$, unit Y_j is a conventional unit. The units are then removed for which

the derivative of the error function to these attentional strengths, α_j's, are small [Thimm *et al* 1995]. Skeletonization can also be applied to prune input units.

- Karnin's method that estimates the sensitivity s of a weight by :

$$s = \sum_{n=1}^{N} (\Delta w(n)^2 \cdot \frac{w(n)}{\eta \cdot (w(n) - w(0))}) \tag{5.2}$$

where $w(n)$ is the weight in the current training epoch n, $w(0)$ the initial weight, and $\Delta w(n)$ the weight change in the n^{th} epoch [Karnin 1990]. The denominator in this formula can become zero, and experiments have indeed shown this to happen. This problem is not dealt with in Karnin's publication. It can, however, easily be solved by setting the whole fraction to zero. The calculation for s then becomes,

$$s = \begin{cases} \sum_{n=1}^{N} (\Delta w(n)^2 \cdot \frac{w(n)}{\eta \cdot (w(n) - w(0))}) & \text{if } w(n) \neq w(0) \\ 0 & \text{if } w(n) = w(0)) \end{cases} \tag{5.3}$$

- Autoprune developed by Finnof *et al*, where a test statistic is defined based on the probability that a weight becomes zero [Finnoff 1993b]. A weight is then removed if the probability that it will become zero is high. Prechelt extended Autoprune to $\lambda - prune$ to calculate the number of units to be pruned at each pruning step [Prechelt 1994].

- Genetic algorithms (GAs) also provide a biological plausible approach to pruning of NNs [Whitley *et al* 1990]. The GA is populated with several pruned versions of the original network. Each of these networks must be trained separately. In this type of pruning genetic operators such as mutation, reproduction and cross-over are applied to create differently pruned networks. These pruned networks 'compete' for survival, being awarded for using fewer parameters and for improving generalization. A drawback of the GA approach to pruning of neural networks is that it is time consuming.

- The Variance Nullity method developed by Engelbrecht is a computationally efficient pruning heuristic based on variance analysis of sensitivity information [Engelbrecht *et al* 1999c, Engelbrecht 2001]. This algorithm utilizes first-order derivatives of the NN output with respect to parameter perturbations, which are already calculated when gradient descent is used for neural network training.

- Optimal Brain Damage (OBD), Optimal Brain Surgeon (OBS) and Optimal Cell Damage (OCD) are all based on second-order derivatives of the 'objective function' with respect to parameter perturbations. In OBD and OCD complexity is reduced by assuming that (a) the function is well approximated by a second-order expansion around its minimum, (b) the off-diagonal elements of the Hessian matrix are zero and (c) all errors between the target and output values are zero. In OBS assumption (b) mentioned above is removed and also is retraining after pruning avoided by automatically adjusting the remaining weights. OBD, OBS and OCD all require differentiable activation functions. Criticism concerning assumption (c) is that outliers in the data nullify the assumption. The calculation of the Hessian in OBD, OBS and OCD increases the complexity of these pruning methods. In OBS the complexity is further increased since the inverse of the Hessian must also be calculated.

The only assumptions for variance nullity pruning method are,

1. that the network must be well trained and

2. that the activation functions must at least be once differentiable.

Also, this algorithm is not as computational intensive as other pruning algorithms. For this reason the 'Variance Nullity pruning Method' is the algorithm of choice to be implemented in this thesis. The next section provides an overview of sensitivity analysis.

5.1 Overview of Sensitivity Analysis

Research has shown that any continuous function can be approximated by a multilayer NN using a monotonically increasing differentiable activation function [Funahashi 1989, Hornik *et al* 1989a]. Gallant *et al* further showed that when a NN converges towards the underlying (target) function, then all the NN derivatives also converge towards the derivatives of the underlying function [Gallant 1992]. This property of NNs derivatives allows efficient use of the NN derivatives to compute sensitivity information. Sensitivity analysis of a system is the study of how the derivatives of a performance function can be used to quantify the response of the system to parameter perturbations [Holtzman 1992]. Thus, sensitivity analysis techniques quantify the relevance of a network parameter (i.e. an input unit, hidden unit or weight) as the influence that small parameter perturbations have on a performance function [Engelbrecht 2001]. Sensitivity analysis also provides a neural network tool to automatically identify all relevant parameters using the significance measures obtained from a sensitivity analysis tool.

There are two main approaches to sensitivity analysis for feed-forward neural networks (FFNNs). These approaches differ in the performance function used. In the one approach the *objective function* to be minimized serves as the performance function, in the second approach it is the neural network *output function*. Objective function sensitivity analysis has been used widely in pruning of NN parameters [Hassibi *et al* 1994, Le Cun *et al* 1990], to develop more sophisticated optimization techniques [Battiti 1992], and to study the robustness and stability of NNs [Oh *et al* 1995]. NN output sensitivity analysis on the other hand has been used to study the generalization abilities of FFNNs [Fu *et al* 1993], to assess the significance of input parameters [Engelbrecht *et al* 1995b], for selective learning and incremental

learning [Engelbrecht *et al* 1999d] and for pruning irrelevant network parameters [Engelbrecht 2001, Zurada *et al* 1997]. In OBD, OBS and OCD sensitivity analysis is performed with regards to the training error. Assuming gradient descent optimization and the sum-squared objective function, Engelbrecht has shown that output sensitivity analysis and OBD are conceptually the same under the assumptions of OBD [Engelbrecht 2001]. Output sensitivity analysis has the advantages that it is not based on assumptions to simplify complexity, as is the case with OBD, OBS and OCD. Also, output sensitivity analysis is less complex than objective function sensitivity analysis. Furthermore, objective function sensitivity analysis is dependent on the objective function and the optimization algorithm used to update the weights. while output sensitivity does not depend on the objective function or the optimization algorithm.

The next section describes the variance nullity pruning algorithm of Engelbrecht. The pruning algorithm is subsequently applied to prune oversized PUNNs used to learn the test functions of section 4.4.1 on page 99.

5.2 The Variance Nullity Pruning Approach

The variance nullity pruning algorithm of Engelbrecht is based on NN output sensitivity where the relevance of parameters is based on parameter sensitivity information [Engelbrecht 2001]. In this algorithm a variance nullity measure is computed for each parameter. The statistical nullity in parameter sensitivity variance is defined in equation (5.4). Thus, the variance nullity measure provides a statistically sound mechanism to decide whether or not a unit or weight is pruned. The objective of the variance nullity measure is to test whether the variance in parameter sensitivity for the different patterns is significantly different from zero [Engelbrecht *et al* 1999c]. If the latter is not the case, then it indicates that the corresponding parameter has little

or no effect on the output of the NN over the entire set of patterns presented to the network. A hypothesis testing step developed by Engelbrecht *et al* uses these variance nullity measures to statistically test if a parameter should be pruned, using the χ^2 distribution [Engelbrecht *et al* 1999c, Engelbrecht 2001].

Engelbrecht *et al* defines statistical nullity in parameter sensitivity variance, Υ_{θ_i}, of a NN parameter θ_i over patterns $p = 1, ..., P$ as follows:

$$\Upsilon_{\theta_i} = \frac{(P-1)\sigma^2_{\theta_i}}{\sigma^2_0} \tag{5.4}$$

where $\sigma^2_{\theta_i}$ is the variance of the sensitivity of the network to perturbations in parameter θ_i, σ^2_0 is a value close to zero and P the number of patterns in the pruning set.
The variance in parameter sensitivity, $\sigma^2_{\theta_i}$, is computed as

$$\sigma^2_{\theta_i} = \frac{\sum_{p=1}^{P}(\aleph^{(p)}_{\theta_i} - \overline{\aleph}_{\theta_i})^2}{P-1} \tag{5.5}$$

where

$$\aleph^{(p)}_{\theta_i} = \frac{\sum_{k=1}^{K} S^{(p)}_{O\theta,ki}}{K} \tag{5.6}$$

and $\overline{\aleph}_{\theta_i}$ is the average parameter sensitivity over all patterns $p = 1, ..., P$, i.e.

$$\overline{\aleph}_{\theta_i} = \frac{\sum_{p=1}^{P} \aleph^{(p)}_{\theta_i}}{P} \tag{5.7}$$

$S_{O\theta}$ refers to the sensitivity matrix of the output vector $\vec{o}$ with respect to the parameter vector $\vec{\theta}$, and individual elements $S_{O\theta,ki}$ refers to the sensitivity of output o_k to perturbations in parameter θ_i over all patterns; $S^{(p)}_{O\theta,ki}$ refers to the sensitivity of output o_k to changes in parameter θ_i for a single pattern p, defined as (assuming differentiable activation functions)

$$S^{(p)}_{O\theta,ki} = \frac{\partial o_k}{\partial \theta^{(p)}_i} \tag{5.8}$$

where $\theta_i^{(p)}$ is the activation value of unit θ_i for pattern p. Section 5.3 derives the sensitivity equations with respect to input and hidden units. In equation (5.6), $\aleph_{\theta_i}^{(p)}$ is the average sensitivity of the NN output to perturbations in parameter θ_i for pattern p over the K output units.

In 'autoprune', developed by Finnoff *et al*, the final weight test variables are based on significance tests for deviations from zero in the weight update process [Finnoff 1993b]. Weights are updated using,

$$\Delta w_h^p(w) = -\eta \cdot \left(\frac{\partial E_p(w)}{\partial w_h} \right) \tag{5.9}$$

where the above denotes the local gradient of the error with respect to pattern p and weight w_h. The results of further training were estimated using an average over the variables ξ_h^p, where

$$\xi_h^p = w_h + \Delta w_h^i(w) \tag{5.10}$$

for $(z_p, t_p) \in \mathcal{D}_t$. For the null hypothesis that the expected value of variable ξ_h^p is equal to zero; the significance of the deviation from zero was tested using the test variable,

$$T_h = \frac{\left| \sum_{p,(z_p,t_p)\in(D)_t} \xi_h^p \right|}{\sqrt{\sum_{p,(z_p,t_p)\in\mathcal{D}_t} (\xi_h^p - \overline{\xi}_h)^2}} \tag{5.11}$$

where $\overline{\xi}_h$ denotes the average over the set ξ_h^p and $(z_p, t_p) \in \mathcal{D}_t$. A large value for T_h indicates high importance of the connection with weight h_p. Connections with small weights can be pruned. In the analysis of means, as is done by Finnoff *et al* a problem may arise where large negative and positive values may cancel each other or produce a sum close to zero, thus incorrectly indicating that the parameter is insignificant. In variance analysis pruning, Engelbrecht *et al* adopted an analysis of variance instead of an analysis of means, as is done by Finnof *et al*, to address this problem.

Basically the statistical pruning heuristic of Engelbrecht is based on proving or disproving the null hypothesis that the variance in parameter sensitivity is approximately zero.

The null hypothesis is then defined as

$$\mathcal{H}_0 : \sigma^2_{\theta_i} = \sigma^2_0 \tag{5.12}$$

This hypothesis can however not be used, since equation (5.4) does not allow $\sigma^2_0 = 0$, and therefore it cannot be hypothesized that the variance in parameter sensitivity over all patterns is exactly zero. To alleviate this problem a small value close to zero is chosen for σ^2_0, and the alternative hypothesis,

$$\mathcal{H}_1 : \sigma^2_{\theta_i} < \sigma^2_0 \tag{5.13}$$

is tested. The variance nullity measure defined in equation (5.14) has a $\chi^2(P-1)$ distribution in the case of P patterns. The critical value, Υ_c, can therefor be obtained from χ^2 distribution tables, i.e.

$$\Upsilon_c = \chi^2_{v;1-\alpha} \tag{5.14}$$

where $v = P - 1$ is the number of degrees of freedom and α is the level of significance. A significance level $\alpha = 0.01$, for example, means that we are satisfied with incorrectly rejecting the hypothesis once out of 100 times. Using the critical value defined in equation (5.14), if $\Upsilon_{\theta_i} \leq \Upsilon_c$, the alternative hypothesis $\mathcal{H}$ is accepted and parameter θ_i is pruned. Engelbrecht *et al* pointed out that the success of this pruning heuristic depended on the value of σ^2_0. A too small value for σ^2_0 will result in no parameters to be pruned. If σ^2_0 is too large, then important parameters may be pruned. It was recommended that the algorithm should start off with a small value for σ^2_0 that is gradually increased if no parameter is pruned. The performance of the network is first tested after each step of pruning. If the performance of the network has not degraded too much, the pruned network is accepted, otherwise the original network is restored and pruning is stopped. Engelbrecht pointed out that the testing of the performance of the pruned network makes the validity of the algorithm insensitive to the value with

which σ_0^2 is increased: if relevant parameters are pruned due to the repetitive increase in σ_0^2, the performance of the network will degrade unacceptably, and the previous architecture will thus be restored.

Computational time during the hypothesis testing phase can be reduced by arranging the variance nullity measures Υ_{θ_i} in increasing order. Hypothesis tests start on the smallest Υ_{θ_i} and continue until no more parameters can be identified for pruning.

The statistical pruning heuristic based on variance nullity is summarized below:

1. Initialize the NN architecture and learning parameters
2. Repeat
   (a) train the NN until overfitting is observed
   (b) let $\sigma_0^2 = 0.0001$
   (c) for each θ_i
      i. for each $p = 1, ..., P$, calculate $\aleph_{\theta_i}^{(p)}$ using equation (5.6)
      ii. calculate the average $\overline{\aleph}_{\theta_i}$ using equation (5.7)
      iii. calculate the variance in parameter sensitivity using $\sigma_{\theta_i}^2$ from equation (5.5)
      iv. calculate test variable Υ_{θ_i} using equation (5.4)
   (d) apply the pruning heuristic
      i. arrange Υ_{θ_i} in increasing order
      ii. find Υ_c using equation (5.14)
      iii. for each θ_i, if $\Upsilon_{\theta_i} \leq \Upsilon_c$, then prune θ_i
      iv. if $\Upsilon_{\theta_i} > \Upsilon_c$ for all θ_i, let $\sigma_0^2 = \sigma_0^2 \times 10$

   until no θ_i is pruned, or the reduced network is not accepted due to an unacceptable deterioration in generalization performance

3. Train the final pruned NN architecture

The variance nullity algorithm starts pruning the hidden layer first, followed by the input layer. Weights can also be pruned once the irrelevant units have been removed. The calculation of the nullity measures can be done on any one of the training, test or validation sets. In this thesis a separate set consisting of 100 randomly generated values was used to calculate variance nullity measures. Pruning is initiated when overfitting is detected on the validation set, i.e. when $\xi_V > \overline{\xi}_V + \delta_{\xi_V}$ where ξ_V is the current error on the validation set, $\overline{\xi}_V$ is the average error on the validation set over the previous iterations and δ_{ξ_V} is the standard deviation in test error. After each pruning step, retraining starts on the reduced network on new initial random weights. The pruning process stops when no more parameters can be identified for pruning, or if the reduced network's performance has degraded too much.

The next section derives the sensitivity equations that are used to calculate the variance nullity measures.

5.3 Sensitivity equations

This section defines equations for the sensitivity analysis of output units with respect to hidden units and input units. It is assumed that the network consists of an input layer, a single hidden layer of product units and an output layer of summation units. Linear activation functions are assumed in both hidden and output layers.

5.3.1 Output-Hidden Layer Analysis

For the sake of notational convenience the superscript p, that refers to a specific pattern, is removed. Let $S_{OY,kj} = \frac{\partial o_k}{\partial y_j}$ be the sensitivity of output unit o_k to small perturbations in hidden hidden unit y_j for a single pattern (The first part of the subscript indicates the layer involved and the second part indicates the respective unit of each layer). Then,

$$\begin{aligned} S_{OY,kj} &= \frac{\partial o_k}{\partial y_j} \\ &= \frac{\partial o_k}{\partial net_{o_k}} \frac{\partial net_{o_k}}{\partial y_j} \\ &= f'(net_{o_k}) \cdot w_{kj} \\ &= w_{kj} \end{aligned} \tag{5.15}$$

where $f'(net_{o_k}) = 1$ for linear activation.

5.3.2 Output-Input Layer Analysis

The sensitivity of output unit O_k with respect to input unit Z_i is calculated as,

$$\begin{aligned} S_{OZ,ki} &= \frac{\partial o_k}{\partial z_i} \\ &= \frac{\partial o_k}{\partial net_{o_k}} \frac{\partial net_{o_k}}{\partial z_i} \end{aligned} \tag{5.16}$$

where, for linear activation,

$$\begin{aligned} o_k &= f(net_{o_k}) \\ &= net_{o_k} \end{aligned} \tag{5.17}$$

Then

$$\begin{aligned} \frac{\partial o_k}{\partial z_i} &= \frac{\partial o_k}{\partial net_{o_k}} \frac{\partial net_{o_k}}{\partial z_i} \\ &= f'(net_{o_k}) \cdot \frac{\partial net_{o_k}}{\partial z_i} \end{aligned}$$

$$
\begin{aligned}
&= 1 \cdot \frac{\partial net_{o_k}}{\partial z_i} \\
&\quad \sum_{j=1}^{J} \frac{\partial net_{o_k}}{\partial y_j} \cdot \frac{\partial y_j}{\partial z_i} \\
&= \sum_{j=1}^{J} w_{kj} \cdot \frac{\partial y_j}{\partial z_i} \qquad (5.18)
\end{aligned}
$$

For a PUNN with a distortion unit, using equation (A.45) on page 200, we have

$$y_j = e^{\rho} \cdot \cos(\pi\phi) \qquad (5.19)$$

where

$$\rho = \sum_{i=1}^{I+1} v_{ji} \ln |z_i|$$

$$\phi = \sum_{i=1}^{I+1} v_{ji} \mathcal{I}_i$$

Thus,

$$
\begin{aligned}
\frac{\partial y_j}{\partial z_i} &= \frac{\partial}{\partial z_i}(e^{\rho} \cdot \cos(\pi\phi)) \\
&= e^{\rho} \frac{\partial \rho}{\partial z_i} \cdot \cos(\pi\phi) + \frac{\partial \cos(\pi\phi)}{\partial z_i} \\
&= e^{\rho} \cdot \frac{\partial (\sum_{i=1}^{I+1} v_{ji} \ln |z_i|)}{\partial z_i} \cdot \cos(\pi\phi) + \frac{\partial \cos(\pi \sum_{i=1}^{I+1} v_{ji} \mathcal{I}_i)}{\partial z_i} \\
&= e^{\rho} \cdot \frac{v_{ji}}{|z_i|} \cdot \cos(\pi\phi) \\
&= \frac{v_{ji}}{|z_i|} \cdot (e^{\rho} \cdot \cos(\pi\phi)) \qquad (5.20)
\end{aligned}
$$

Substitution of (5.20) in (5.18) gives

$$\frac{\partial o_k}{\partial z_i} = \sum_{j=1}^{J} w_{kj} \frac{v_{ji}}{|z_i|} \cdot (e^{\rho} \cdot \cos(\pi\phi)) \qquad (5.21)$$

This concludes the derivation of the sensitivity equations for a PUNN with a distortion unit.

The next section applies the variance nullity pruning algorithm to PUNNs for selected function approximation problems.

5.4 Application of the Variance Nullity Pruning Algorithm to PUNNs

The variance nullity pruning algorithm was applied to the eight test functions described on page 99. The training and test sets of section 4.4.1 were used in training the network. The PUNNs were trained using particle swarm optimization. The optimal parameters for each of the eight test functions, as determined in chapter 4, were used for oversized initial networks. In these oversized networks, the number of hidden units were deliberately increased from the optimal architectures as determined in chapter 4. After each pruning step the weights were randomly re-initialized, as stipulated by the variance nullity pruning algorithm. In the case of PSO this implies re-initializing the positions and velocities for the particles before the next pruning step commenced. This random re-initialization of weights sometimes resulted in poorer performance of the re-initialized network by often producing larger MSEs on the training set and poorer generalization on the test set.

In this section a network's performance was measured by its MSE on the test set, in other words, its generalization. Tables 5.1 to 5.8 contain for both the oversized and pruned network the number of hidden units, MSE on training and test sets for 30 simulations, the average MSE on training and tests sets and the average number of hidden units. Unacceptable performance was defined as a reduction of 20% or more in the MSE on the test set on the subsequent pruning step, in which case the pruning process was stopped. This explains the entries in tables 5.1 to 5.8 where pruning ended with the same number of hidden weights as the initial oversized network. In these instances no parameters were identified for pruning and the pruning process was repeated with a smaller value for θ_0^2, with re-initialized particles that often resulted in larger MSE values. For each function, 30 pruning simulations were conducted as

reflected in tables 5.1 to 5.11. Tables 5.1 to 5.8 contain the results for the pruning of the hidden layer for all eight functions. Further, to show that the variance nullity can also be applied to prune the input units of PUNNs, pruning was applied to three of the eight test functions. In these cases extra input units were added to the architecture, with inputs for these units randomly generated. Tables 5.9 to 5.11 contain the results for pruning of the input layer for PUNNs. The tables also contain the averages for the number of hidden and input units calculated over the 30 simulations and the average MSEs calculated on the training and test sets together with a 95% confidence interval. Tables 5.1 to 5.8 show that the average number of hidden units for the various functions are close to the optimal number of hidden units as determined in chapter 4, bearing in mind that the average also includes the number of simulations where the initial oversized networks showed a degradation in performance due to re-initialization of weights.

Each training session started with random weights. Due to the stochastic search employed by PSO, the particle swarm optimizer is not always guaranteed to converge to a global optimum . PSO did therefor not always succeed in pruning all the irrelevant hidden units [Van den Bergh *et al* 2001c]. This explains why the average number of hidden units is slightly higher than the values obtained in chapter 4. Table 5.2 on page 157 shows that an initial network comprising 8 hidden units, for the cubic function, were pruned to 1 unit in 27 out of 30 simulations. The average number of hidden units for the pruned network as reflected in table 5.2 over 30 simulations is 2 (i.e. 1.5 rounded). This shows that the function $f(x) = x^3 - 0.04x$ can be represented by a PUNN containing two hidden units compared to an optimal SUNN that requires 3 hidden units. The tables also reflect a performance similar to the results contained in table 4.25 on page 121.

The variance nullity method can only remove irrelevant units; it cannot remove redun-

dant units from a network. This may explain why certain simulations ended in a higher number of units than the optimal number of units for the specific function. Similarly, the average number of input units are comparable to the optimal number of input units as determined in chapter 4. The performance of the oversized network, in all cases, is much poorer than the performance of the pruned network. Thus, a larger PUNN architecture does not necessarily translate to an increase in network performance, since the larger NNs overfitted the data.

5.5 Conclusion

In this chapter the variance nullity pruning algorithm developed by Engelbrecht was discussed and applied to oversized PUNN architectures of the eight test functions (as defined in chapter 4). The variance nullity pruning approach successfully pruned irrelevant hidden and input units of PUNNs. The variance nullity pruning algorithm produced averages for the number of hidden and input units that were comparable to the optimal number of units as determined by brute force in chapter 4. The results also indicate that the initial oversized PUNNs did not produce smaller MSEs than the pruned networks. This implies that in the case of PUNNs, a larger network does not necessarily translate into better performance. Re-initialization of oversized networks when no parameters were identified for pruning often resulted in a poorer performance that may lead to early termination of the pruning process. This could have been avoided, if all the unpruned weights were retained for the next pruning step, where a smaller value for θ_0^2 will subsequently be used by the pruning algorithm. Thus, an improvement for the variance nullity algorithm applied to PUNNs is to avoid re-initialization of weights in cases where no parameters were pruned by retaining the unpruned weights and to continue the pruning process by re-training only the bias.

$f(x) = x^2$						
	Oversized network				Pruned Network	
Simulation No	No of hidden units	MSE on Training set	MSE on Test set	No of hidden units	MSE on Training set	MSE on Test set
1	8	0.0031	0.0039	1	0.000377	0.000619
2	8	0.0134	0.0240	1	0.001376	0.001675
3	8	0.0062	0.0108	1	0.000059	0.000065
4	8	0.0292	0.0210	1	0.000219	0.000324
5	8	0.0290	0.0844	1	0.000448	0.000290
6	8	0.0041	0.0045	8	0.004073	0.004544
7	8	0.0102	0.0178	2	0.000643	0.003382
8	8	0.0157	0.0130	1	0.000205	0.000156
9	8	0.0077	0.0088	1	0.000172	0.000186
10	8	0.0018	0.0014	1	0.000051	0.000054
11	8	0.0244	0.0271	1	0.000100	0.000082
12	8	0.0193	0.0392	2	0.000563	0.000652
13	8	0.0171	0.0131	1	0.000238	0.000269
14	8	0.0032	0.0034	1	0.000061	0.000048
15	8	0.0265	0.0187	2	0.000777	0.000297
16	8	0.0042	0.0895	2	0.000434	0.000530
17	8	0.0246	0.0206	2	0.000195	0.000098
18	8	0.0252	0.5363	2	0.000906	0.000553
19	8	0.0203	0.0342	1	0.000270	0.000180
20	8	0.0083	0.0090	3	0.001427	0.001020
21	8	0.0056	0.0052	1	0.000126	0.000153
22	8	0.0369	0.0259	1	0.000101	0.000087
23	8	0.0177	0.0245	2	0.001014	0.000892
24	8	0.0172	0.0127	2	0.000189	0.000230
25	8	0.0121	0.0610	1	0.000140	0.000383
26	8	0.0124	0.0171	2	0.000359	0.000793
27	8	0.0051	0.0063	1	0.000131	0.000111
28	8	0.0046	0.0032	1	0.000648	0.000635
29	8	0.0224	0.0896	1	0.000632	0.000501
30	8	0.0075	0.0185	1	0.000253	0.000475
	Average no of hidden units	8		Average no of hidden units	1.6	
	Average	0.01450	0.04149	Average	0.00054	0.00064
	Confidence	0.00340	0.03512	Confidence	0.00028	0.00036

Table 5.1: Pruning of hidden units - function F1

$f(x) = x^3 - 0.04x$							
	Oversized network				Pruned Network		
Simulation No	No of hidden units	MSE on Training set	MSE on Test set		No of hidden units	MSE on Training set	MSE on Test set
1	8	0.0025	0.0075		1	0.000009	0.000009
2	8	0.0044	0.0106		1	0.000042	0.000035
3	8	0.0016	0.0008		8	0.001625	0.000844
4	8	0.0050	0.0043		1	0.000008	0.000006
5	8	0.0055	0.0037		2	0.000229	0.000132
6	8	0.0074	0.0142		1	0.000024	0.000026
7	8	0.0061	0.0072		1	0.000009	0.000008
8	8	0.0033	0.0034		1	0.000008	0.000007
9	8	0.0060	0.0065		1	0.000008	0.000009
10	8	0.0121	0.0165		1	0.000012	0.000007
11	8	0.0048	0.0042		1	0.000013	0.000013
12	8	0.0024	0.0039		1	0.000007	0.000008
13	8	0.0009	0.0013		1	0.000013	0.000010
14	8	0.0024	0.0027		1	0.000035	0.000035
15	8	0.0035	0.0022		1	0.000014	0.000010
16	8	0.0017	0.0016		1	0.000010	0.000008
17	8	0.0063	0.0052		1	0.000011	0.000009
18	8	0.0098	0.0063		1	0.000019	0.000016
19	8	0.0003	0.0003		1	0.000009	0.000007
20	8	0.0037	0.0051		1	0.000010	0.000009
21	8	0.0033	0.0038		1	0.000034	0.000035
22	8	0.0058	0.0076		1	0.000009	0.000010
23	8	0.0030	0.0034		1	0.000009	0.000008
24	8	0.0009	0.0008		1	0.000007	0.000006
25	8	0.0075	0.0114		1	0.000007	0.000016
26	8	0.0023	0.0040		8	0.002251	0.004021
27	8	0.0028	0.0039		1	0.000013	0.000008
28	8	0.0059	0.0049		1	0.000011	0.000008
29	8	0.0053	0.0061		1	0.000015	0.000011
30	8	0.0081	0.0092		1	0.000008	0.000008
	Average no of hidden units	8			Average no of hidden units	1.5	
	Average	0.00449	0.00542		Average	0.00015	0.00018
	Confidence	0.00099	0.00140		Confidence	0.00018	0.00027

Table 5.2: Pruning of hidden units - function F2

$z_t = 1 + 0.3z_{t-2} - 1.4z_{t-1}^2$						
	Oversized network				Pruned Network	
Simulation No	No of hidden units	MSE on Training set	MSE on Test set	No of hidden units	MSE on Training set	MSE on Test set
1	10	0.0004	0.0010	4	0.000102	0.006468
2	10	0.0022	0.0409	5	0.000039	0.000083
3	10	0.0003	0.0038	7	0.000220	0.000315
4	10	0.0010	0.0061	9	0.000036	0.000050
5	10	0.0002	0.0003	10	0.000175	0.000293
6	10	0.0007	0.0009	10	0.000658	0.000871
7	10	0.0117	0.0450	9	0.000069	0.000249
8	10	0.0008	0.0171	6	0.000017	0.000022
9	10	0.0003	0.0031	9	0.000015	0.000034
10	10	0.0006	0.0010	8	0.000109	0.000286
11	10	0.0025	0.0078	4	0.000011	0.000018
12	10	0.0092	0.0684	7	0.000189	0.000220
13	10	0.0002	0.0009	10	0.000233	0.000908
14	10	0.0003	0.0002	7	0.000062	0.000065
15	10	0.0011	0.1009	4	0.000041	0.000055
16	10	0.0001	0.0010	4	0.000088	0.000112
17	10	0.0623	0.0637	6	0.000007	0.000009
18	10	0.0009	0.0035	6	0.000082	0.000098
19	10	0.0001	0.0007	10	0.000092	0.000660
20	10	0.0002	0.0002	5	0.000011	0.000014
21	10	0.0032	0.0495	6	0.000081	0.000446
22	10	0.0094	0.0395	4	0.000129	0.001090
23	10	0.0010	0.0273	5	0.000027	0.000028
24	10	0.0004	0.0012	10	0.000432	0.001203
25	10	0.0010	0.0016	5	0.000064	0.000070
26	10	0.0002	0.0041	4	0.000001	0.000001
27	10	0.0032	0.0067	6	0.000005	0.000009
28	10	0.0010	0.0037	10	0.000196	0.000431
29	10	0.0008	0.0027	10	0.000829	0.002712
30	10	0.0020	0.0462	5	0.000090	0.000126
	Average no of hidden units	10		Average no of hidden units	6.8	
	Average	0.00391	0.03882	Average	0.00014	0.00056
	Confidence	0.00415	0.04519	Confidence	0.00007	0.00020

Table 5.3: Pruning of hidden units - function F3

$f(x,y) = y^7x^3 - 0.5x^6$						
	Oversized network				Pruned Network	
Simulation No	No of hidden units	MSE on Training set	MSE on Test set	No of hidden units	MSE on Training set	MSE on Test set
1	10	0.0015	0.0045	2	$2.15E-06$	$1.20E-03$
2	10	0.0037	0.0027	2	$1.02E-04$	$4.09E-05$
3	10	0.0007	0.0009	3	$1.22E-04$	$1.03E-04$
4	10	0.0009	0.0020	2	$1.52E-06$	$2.15E-03$
5	10	0.0013	0.0024	4	$7.04E-05$	$2.27E-04$
6	10	0.0017	0.0035	2	$4.50E-08$	$6.77E-03$
7	10	0.0002	0.0012	6	$1.17E-04$	$6.84E-04$
8	10	0.0008	0.0027	3	$4.57E-05$	$9.08E-05$
9	10	0.0023	0.0028	2	$1.32E-04$	$1.98E-03$
10	10	0.0011	0.0018	2	$1.76E-06$	$2.10E-04$
11	10	0.0028	0.0035	2	$3.43E-05$	$4.85E-04$
12	10	0.0009	0.0012	3	$3.49E-04$	$5.79E-04$
13	10	0.0007	0.0009	2	$7.24E-04$	$1.30E-04$
14	10	0.0002	0.0003	10	$2.22E-04$	$1.20E-04$
15	10	0.0025	0.0032	2	$7.34E-05$	$5.48E-05$
16	10	0.0011	0.0019	2	$4.00E-04$	$3.36E-03$
17	10	0.0002	0.0003	8	$1.85E-04$	$2.28E-03$
18	10	0.0003	0.0003	10	$2.80E-04$	$7.76E-04$
19	10	0.0013	0.0014	2	$1.94E-03$	$1.21E-03$
20	10	0.0064	0.0068	3	$1.24E-04$	$2.58E-04$
21	10	0.0028	0.0042	2	$2.07E-04$	$3.11E-05$
22	10	0.0027	0.0030	3	$1.77E-04$	$6.22E-05$
23	10	0.0014	0.0021	2	$6.76E-04$	$2.76E-03$
24	10	0.0001	0.0001	10	$8.91E-05$	$1.54E-04$
25	10	0.0003	0.0004	3	$3.21E-04$	$9.21E-04$
26	10	0.0005	0.0005	10	$5.17E-04$	$1.85E-04$
27	10	0.0045	0.0059	4	$4.13E-05$	$5.70E-03$
28	10	0.0045	0.0057	3	$7.12E-04$	$2.46E-04$
29	10	0.0012	0.0014	2	$4.32E-04$	$6.67E-03$
30	10	0.0012	0.0020	2	$1.57E-04$	$2.40E-04$
	Average no of hidden units	10		Average no of hidden units	3.8	
	Average	0.00166	0.00232	Average	0.00028	0.00132
	Confidence	0.00055	0.00062	Confidence	0.00014	0.00071

Table 5.4: Pruning of hidden units - function F4

$f(x,y) = x^2 + y^2$							
	Oversized network				Pruned Network		
Simulation No	No of hidden units	MSE on Training set	MSE on Test set		No of hidden units	MSE on Training set	MSE on Test set
1	10	0.0470	0.0491		2	0.00727	0.00777
2	10	0.0301	0.0357		2	0.01654	0.01864
3	10	0.0228	0.0262		2	0.00781	0.00875
4	10	0.0171	0.0174		2	0.01693	0.01674
5	10	0.0407	0.0774		4	0.02883	0.03137
6	10	0.0175	0.0186		10	0.01750	0.01859
7	10	0.0490	0.0433		2	0.00598	0.00596
8	10	0.1183	0.1338		2	0.00214	0.00212
9	10	0.0158	0.0152		10	0.01576	0.01517
10	10	0.0354	0.0512		3	0.00688	0.00699
11	10	0.0309	0.0341		3	0.00714	0.00785
12	10	0.0215	0.0248		2	0.00402	0.00393
13	10	0.0304	0.0345		2	0.01887	0.02020
14	10	0.0679	0.0774		2	0.00567	0.00508
15	10	0.0325	0.0434		4	0.00037	0.00043
16	10	0.0096	0.0113		10	0.00959	0.01129
17	10	0.0320	0.0328		3	0.02237	0.02253
18	10	0.0300	0.0297		3	0.00002	0.00002
19	10	0.0260	0.0359		2	0.01418	0.01467
20	10	0.0358	0.0368		2	0.01639	0.01590
21	10	0.0303	0.0457		6	0.01378	0.01374
22	10	0.0273	0.0328		3	0.00067	0.00069
23	10	0.0110	0.0162		3	0.00605	0.00633
24	10	0.0185	0.0233		2	0.00588	0.00591
25	10	0.0220	0.0291		3	0.01463	0.01780
26	10	0.0346	0.0309		2	0.01591	0.01385
27	10	0.0129	0.0169		2	0.00112	0.00133
28	10	0.0512	0.0586		2	0.02316	0.02118
29	10	0.0265	0.0359		2	0.01582	0.01483
30	10	0.0362	0.0537		3	0.00693	0.00687
	Average no of hidden units	10			Average no of hidden units	3.3	
	Average	0.03269	0.03501		Average	0.01024	0.01122
	Confidence	0.00744	0.00653		Confidence	0.00299	0.00281

Table 5.5: Pruning of hidden units - function F5

$f(x,y) = sin(x^2) + sin(y^2)$							
	Oversized network				Pruned Network		
Simulation No	No of hidden units	MSE on Training set	MSE on Test set		No of hidden units	MSE on Training set	MSE on Test set
1	10	0.0068	0.0210		6	0.00019	0.00021
2	10	0.0015	0.0020		8	0.00020	0.00048
3	10	0.0016	0.0025		4	0.00020	0.00021
4	10	0.0163	0.0440		2	0.00224	0.00343
5	10	0.0017	0.0142		2	0.00007	0.00007
6	10	0.0071	0.0112		2	0.00007	0.00008
7	10	0.0013	0.0035		7	0.00029	0.00043
8	10	0.0014	0.0038		10	0.00136	0.00377
9	10	0.0002	0.0017		6	0.00020	0.00038
10	10	0.0018	0.0021		7	0.00042	0.00069
11	10	0.0063	0.0091		4	0.00012	0.00021
12	10	0.0084	0.0141		6	0.00073	0.00134
13	10	0.0009	0.0010		10	0.00092	0.00105
14	10	0.0039	0.0043		3	0.00019	0.00023
15	10	0.0046	0.0069		4	0.00017	0.00025
16	10	0.0061	0.0111		7	0.00013	0.00017
17	10	0.0039	0.0047		5	0.00064	0.00119
18	10	0.0006	0.0007		10	0.00061	0.00071
19	10	0.0007	0.0026		4	0.00018	0.00020
20	10	0.0036	0.0046		5	0.00030	0.00045
21	10	0.0082	0.0437		2	0.00010	0.00013
22	10	0.0025	0.0102		3	0.00008	0.00011
23	10	0.0037	0.0079		5	0.00009	0.00013
24	10	0.0011	0.0014		4	0.00027	0.00035
25	10	0.0023	0.0037		2	0.00006	0.00008
26	10	0.0008	0.0015		6	0.00041	0.00106
27	10	0.0053	0.1544		5	0.00020	0.00028
28	10	0.0019	0.0121		6	0.00007	0.00036
29	10	0.0097	0.0163		7	0.00108	0.00182
30	10	0.0046	0.0130		2	0.00250	0.00280
	Average no of hidden units	10			Average no of hidden units	5.1	
	Average	0.00396	0.01431		Average	0.00047	0.00076
	Confidence	0.00127	0.01039		Confidence	0.00022	0.00036

Table 5.6: Pruning of hidden units - function F6

$f(x,y) = (4 - 2.1x^2 + (\frac{x^3}{3}))x^2 + xy + (4y^2 - 4)y^2$						
	Oversized network				Pruned Network	
Simulation No	No of hidden units	MSE on Training set	MSE on Test set	No of hidden units	MSE on Training set	MSE on Test set
1	15	0.0381	0.0459	12	0.03840	0.04219
2	15	0.0311	0.0471	5	0.02066	0.02591
3	15	0.0244	0.0317	10	0.03526	0.04204
4	15	0.0329	0.0421	2	0.06240	0.06875
5	15	0.0343	0.0436	15	0.03427	0.04364
6	15	0.0328	0.0362	6	0.03735	0.03649
7	15	0.0240	0.0322	4	0.02728	0.03045
8	15	0.0356	0.0570	4	0.03397	0.04387
9	15	0.0327	0.0407	3	0.02985	0.03440
10	15	0.0253	0.0411	5	0.02566	0.03724
11	15	0.0290	0.0512	3	0.04249	0.05062
12	15	0.0273	0.0399	5	0.01284	0.01518
13	15	0.0293	0.0351	4	0.02905	0.03501
14	15	0.0462	0.0145	7	0.02749	0.03669
15	15	0.0268	0.0772	4	0.02502	0.03588
16	15	0.0245	0.0394	4	0.02544	0.03403
17	15	0.0250	0.0302	15	0.02503	0.03022
18	15	0.0333	0.0383	5	0.02383	0.02318
19	15	0.0292	0.0351	3	0.03048	0.03614
20	15	0.0308	0.0394	14	0.02389	0.02961
21	15	0.0210	0.0276	4	0.02909	0.02626
22	15	0.0324	0.0783	4	0.02561	0.02977
23	15	0.0284	0.0403	4	0.03356	0.03624
24	15	0.0292	0.0388	4	0.03355	0.03580
25	15	0.0226	0.0384	3	0.07735	0.08508
26	15	0.0298	0.0546	3	0.04043	0.04510
27	15	0.0271	0.0301	14	0.02730	0.02504
28	15	0.0372	0.0611	3	0.02646	0.05175
29	15	0.0451	0.0110	5	0.05121	0.06213
30	15	0.0319	0.0459	3	0.02680	0.03542
	Average no of hidden units	15		Average no of hidden units	5.9	
	Average	0.03058	0.04252	Average	0.03273	0.03880
	Confidence	0.00208	0.00498	Confidence	0.00457	0.00512

Table 5.7: Pruning of hidden units - function F7

$f(x,y) = sin(x) \cdot sin(y) \cdot \sqrt{xy}$						
	Oversized network				Pruned Network	
Simulation No	No of hidden units	MSE on Training set	MSE on Test set	No of hidden units	MSE on Training set	MSE on Test set
1	12	0.0001	0.0001	8	0.000048	0.000056
2	12	0.0006	0.0010	7	0.000010	0.000011
3	12	0.0021	0.0038	8	0.000116	0.001136
4	12	0.0001	0.0002	6	0.000107	0.000102
5	12	0.0001	0.0001	12	0.000129	0.000131
6	12	0.0013	0.0064	4	0.000005	0.000005
7	12	0.0001	0.0001	7	0.000060	0.000065
8	12	0.0003	0.0005	5	0.000016	0.000013
9	12	0.0002	0.0006	6	0.000084	0.000137
10	12	0.0001	0.0017	12	0.000096	0.001705
11	12	0.0007	0.0159	8	0.000168	0.000421
12	12	0.0011	0.0021	6	0.000041	0.000055
13	12	0.0038	0.0198	8	0.000089	0.000071
14	12	0.0005	0.0009	6	0.000025	0.000069
15	12	0.0001	0.0003	12	0.000108	0.000348
16	12	0.0002	0.0331	4	0.000060	0.000065
17	12	0.0003	0.0168	4	0.000043	0.000167
18	12	0.0002	0.0002	5	0.000014	0.000019
19	12	0.0003	0.0006	5	0.000010	0.000012
20	12	0.0007	0.0009	6	0.000016	0.000017
21	12	0.0001	0.0001	7	0.000050	0.000061
22	12	0.0002	0.0003	12	0.000178	0.000298
23	12	0.0001	0.0002	12	0.000102	0.000192
24	12	0.0004	0.0005	4	0.000010	0.000008
25	12	0.0004	0.0008	5	0.000039	0.000039
26	12	0.0003	0.0007	4	0.000076	0.000220
27	12	0.0001	0.0014	8	0.000333	0.001303
28	12	0.0002	0.0286	12	0.000608	0.003324
29	12	0.0006	0.0138	4	0.000047	0.000062
30	12	0.0003	0.0003	5	0.000042	0.000035
	Average no of hidden units	12		Average no of hidden units	7.1	
	Average	0.00052	0.00506	Average	0.00009	0.00068
	Confidence	0.00027	0.00325	Confidence	0.00004	0.00077

Table 5.8: Pruning of hidden units - function F8

$f(x) = x^2$						
	Oversized network			Pruned Network		
Simulation No	No of input units	MSE on Training set	MSE on Test set	No of input units	MSE on Training set	MSE on Test set
1	4	0.0239	0.0365	1	0.00063	0.00054
2	4	0.0231	0.0224	1	0.00019	0.00016
3	4	0.0354	0.0313	4	0.04229	0.03902
4	4	0.0118	0.0229	1	0.00020	0.00028
5	4	0.0431	0.0711	1	0.00060	0.00087
6	4	0.0103	0.0100	1	0.00019	0.00020
7	4	0.0167	0.0166	2	0.00157	0.00246
8	4	0.0177	0.0318	2	0.00135	0.00129
9	4	0.0225	0.0284	1	0.00046	0.00055
10	4	0.0093	0.0359	1	0.00029	0.00031
11	4	0.0227	0.0242	1	0.00028	0.00017
12	4	0.0151	0.0122	1	0.00026	0.00021
13	4	0.0176	0.0176	2	0.00021	0.00019
14	4	0.0219	0.0142	1	0.00054	0.00023
15	4	0.0161	0.0170	2	0.00135	0.00139
16	4	0.0187	0.0214	1	0.00021	0.00016
17	4	0.0191	0.0078	1	0.00053	0.00050
18	4	0.0333	0.0275	1	0.00045	0.00061
19	4	0.0185	0.0160	1	0.00030	0.00033
20	4	0.0132	0.0113	2	0.00107	0.00095
21	4	0.0164	0.0179	1	0.00025	0.00022
22	4	0.0404	0.0363	1	0.00053	0.00039
23	4	0.0219	0.0257	1	0.00009	0.00009
24	4	0.0243	0.0385	1	0.00049	0.00045
25	4	0.0082	0.0101	1	0.00007	0.00009
26	4	0.0275	0.0248	1	0.00011	0.00009
27	4	0.0209	0.0096	2	0.00034	0.00025
28	4	0.0139	0.0091	1	0.00051	0.00032
29	4	0.0036	0.0026	1	0.00002	0.00007
30	4	0.0189	0.0161	1	0.00048	0.00031
	Average no of hidden units	4		Average no of hidden units	1.3	
	Average	0.01970	0.02223	Average	0.00186	0.00171
	Confidence	0.00327	0.00477	Confidence	0.00278	0.00257

Table 5.9: Pruning of input units - function F1

$f(x) = x^3 - 0.04x$							
	Oversized network				Pruned Network		
Simulation No	No of input units	MSE on Training set	MSE on Test set		No of input units	MSE on Training set	MSE on Test set
1	4	0.00201	0.00370		1	0.000011	0.000038
2	4	0.00624	0.00605		1	0.000012	0.000010
3	4	0.00011	0.00009		1	0.000040	0.000036
4	4	0.00038	0.00065		1	0.000008	0.000007
5	4	0.00033	0.00054		1	0.000011	0.000009
6	4	0.00002	0.00001		4	0.000015	0.000014
7	4	0.00228	0.00522		1	0.000008	0.000007
8	4	0.00024	0.00085		1	0.000009	0.000007
9	4	0.00003	0.00003		4	0.000026	0.000026
10	4	0.00004	0.00003		1	0.000007	0.000007
11	4	0.01621	0.02198		1	0.000007	0.000006
12	4	0.00015	0.00008		1	0.000041	0.000048
13	4	0.00007	0.00004		4	0.000072	0.000044
14	4	0.00010	0.00015		4	0.000059	0.000047
15	4	0.00622	0.00465		1	0.000008	0.000006
16	4	0.00041	0.00020		1	0.000016	0.000025
17	4	0.00034	0.00047		1	0.000036	0.000068
18	4	0.00007	0.00005		4	0.000155	0.000152
19	4	0.00238	0.00376		1	0.000007	0.000006
20	4	0.00572	0.00484		1	0.000009	0.000008
21	4	0.01302	0.01765		1	0.000011	0.000015
22	4	0.00243	0.00337		1	0.000007	0.000006
23	4	0.00022	0.00045		4	0.000216	0.000450
24	4	0.00001	0.00001		1	0.000013	0.000009
25	4	0.00025	0.00021		1	0.000038	0.000040
26	4	0.00011	0.00006		1	0.000007	0.000007
27	4	0.00041	0.00052		1	0.000008	0.000006
28	4	0.00005	0.00003		1	0.000008	0.000006
29	4	0.00013	0.00013		4	0.000130	0.000132
30	4	0.00074	0.00108		1	0.000009	0.000007
	Average no of hidden units	4			Average no of hidden units	1.7	
	Average	0.00199	0.00256		Average	0.00003	0.00004
	Confidence	0.00142	0.00185		Confidence	0.00002	0.00003

Table 5.10: Pruning of input units - function F2

$f(x,y) = y^7x^3 - x^6$						
	Oversized network				Pruned Network	
Simulation No	No of hidden units	MSE on Training set	MSE on Test set	No of hidden units	MSE on Training set	MSE on Test set
1	4	0.0046	0.0067	2	0.00038	0.00063
2	4	0.0087	0.0109	2	0.00003	0.00006
3	4	0.0031	0.0066	2	0.00033	0.00058
4	4	0.0059	0.0072	2	0.00056	0.00071
5	4	0.0040	0.0077	2	0.00041	0.00073
6	4	0.0089	0.0160	4	0.00887	0.01596
7	4	0.0077	0.0094	2	0.00038	0.00063
8	4	0.0100	0.0125	2	0.00055	0.00095
9	4	0.0071	0.0083	2	0.00005	0.00006
10	4	0.0057	0.0096	2	0.00032	0.00062
11	4	0.0045	0.0074	2	0.00046	0.00067
12	4	0.0030	0.0060	2	0.00030	0.00059
13	4	0.0092	0.0119	2	0.00036	0.00060
14	4	0.0046	0.0062	2	0.00042	0.00060
15	4	0.0092	0.0187	4	0.00921	0.01865
16	4	0.0100	0.0118	2	0.00041	0.00065
17	4	0.0093	0.0109	3	0.00108	0.00158
18	4	0.0033	0.0060	2	0.00036	0.00062
19	4	0.0083	0.0113	2	0.00036	0.00062
20	4	0.0077	0.0086	2	0.00049	0.00064
21	4	0.0078	0.0116	2	0.00370	0.00600
22	4	0.0067	0.0093	4	0.00665	0.00932
23	4	0.0059	0.0062	2	0.00064	0.00071
24	4	0.0087	0.0105	2	0.00053	0.00059
25	4	0.0096	0.0119	3	0.00044	0.00058
26	4	0.0067	0.0079	2	0.00477	0.00622
27	4	0.0050	0.0074	2	0.00042	0.00062
28	4	0.0100	0.0107	2	0.00005	0.00006
29	4	0.0029	0.0061	2	0.00030	0.00060
30	4	0.0060	0.0097	4	0.00599	0.00973
	Average no of hidden units	4		Average no of hidden units	2.3	
	Average	0.00680	0.00950	Average	0.00163	0.00269
	Confidence	0.00084	0.00108	Confidence	0.00097	0.00172

Table 5.11: Pruning of input units - function F4

Chapter 6

Conclusions

One of the objectives of this thesis was to show that the back-propagation algorithm, that provides a computationally efficient method for the training of multilayer summation neural networks, fails to train PUNNs. The reason for its failure can be ascribed to the search space for PUNNs that is usually extremely convoluted [Durbin *et al* 1989, Leerink *et al* 1995]. The main reasons for the failure of gradient descent in the convoluted search space of PUNNs are (a) incorrect weight initialization and (b) the presence of an increased number of local minima. Generally, gradient descent only manages to train PUNNs when the weights are initialized in close proximity of the optimal weight values. Usually, the optimal weight values are often not available resulting in bad choices for weight initialization, which in turn causes gradient descent to get stuck in one of the numerous local minima that occur on the error surface or become paralyzed (which occurs when the gradient of the error with respect to the current weight is close to zero). In chapter 3 an inspection of the error surfaces of $f(z) = z^3$, with $z \in [-1, 1]$, and $f(z_1, z_2) = z_1^2 + z_2^2$, with $z_1, z_2 \in [-1, 1]$, indicated that weight initialization greatly influenced the convergence of gradient descent when applied to PUNNs. It illustrated that initial weights chosen such that the direction of the negative of its gradient points to a local rather than a global minimum,

often resulted in gradient descent to converge to and become trapped by this bad local minimum. Also, it was shown that if initial weights are chosen along a steep incline of the error surface, where the derivative of the error surface with respect to the weight is extremely large, then weight updates will be large which may cause jumping over the global minimum. The neural network then oscillates between extreme points of the error surface overshooting the global minimum each time. The results in chapter 4 with respect to functions F1 and F2 indicate that gradient descent using PUNNs were trapped in local minima resulting in much larger MSEs than achieved by particle swarm optimization and genetic algorithms.

Another objective was to show that global optimization algorithms such as genetic algorithms, particle swarm optimization and leapfrog optimization could be used to avoid the numerous local minima that occur on the error surface of PUNNs in training PUNNs successfully. The results in chapter 4 in table 4.25 on page 121 show that the various optimization algorithms applied to PUNNs produced much lower MSEs on the training and test sets for each function than gradient descent applied to PUNNs, indicating that the PSO, GA and LFOP are more successful in training PUNNs than gradient descent. In functions F3, F4, F5, F6, F7 and F8 gradient descent was unsuccessful in training the corresponding PUNNs. In a comparison of the global optimization algorithms applied to SUNNs, it is evident that LFOP:SUs managed to produce smaller training errors and generalized much better than BP:SUs, except for functions $f(x) = x^2$ and $f(x, y) = x^2 + y^2$, where BP:SUs outperformed PSO:SUs and LFOP:SUs. The global optimization algorithms applied to SUNNs in all eight functions did not perform better than BP:SUs, however, they did manage to reach lower generalization levels using much fewer iterations than BP:SUs and a corresponding higher percentage of convergence for the various generalization levels. In the case of function F5, although BP:SUs achieved a smaller training error than

PSO:PUs, it did not manage to reach the low generalization level of 0.0001, that was achieved by PSO:PUs. It can be concluded that the global optimization algorithms appeared to find the global minimum on the error surface faster than BP:SUs.

Another objective was to determine the global optimization algorithm which is more efficient and robust in training PUNNs. Results in chapter 4 indicated that PUNNs performed the best with respect to functions F1, F2, F6 and F8, while SUNNs outperformed the PUNNs in functions F3, F4, F5 and F7 achieving lower MSEs on the training sets and improved generalization. In the case of the global algorithms applied to PUNNs, PSO was the only algorithm that managed to reach a low generalization level of 0.0001 for all functions, except for function F7. LFOP applied to PUNNs also managed to reach low generalization levels of 0.00001 in functions F1, F2, F3 and F5. GAs only managed to achieve a low generalization level in functions F1 and F2. In choosing a global algorithm applied to PUNNs that is the most robust, it appears that PSO is more robust than LFOP with respect to functions F1 and F2 from tables 4.30 and 4.32, also taking fewer iterations than PSO to reach convergence, whereas PSO appears to be more robust with respect to functions F4 and F5 since they have a larger percentage of simulations that converged to a generalization level of 0.00001. PSO tends to be more robust than LFOP if one takes into account the instances of functions F6, F7 and F8 where not a single simulation of LFOP could train the PUNNs successfully as reflected in tables 4.25 and 4.32; only overflows were produced in these cases due to the large weight adjustments caused by gradient descent.

The optimal architectures for PUNNs were initially determined using brute force pruning in chapter 4 which resulted in much smaller architectures. The number of hidden units that occurred in optimal PUNNs expressed as a percentage of the number of hidden units that occurred in the equivalent optimal SUNNs, are for

functions F1 50%, F2 33.3%, F3 80%, F4 33.3%, F5 50%, F6 66.7%, F7 75% and F8 77.8%. This shows that the optimal PUNNs were smaller than the equivalent SUNNs for the eight test functions. The variance nullity pruning algorithm applied in chapter 5 produced similar PUNN architectures as the brute force pruning approach of section 4.8 on page 114. The results of chapter 4 show that PUNNs, with their much smaller optimal architectures compared to the corresponding larger optimal SUNNs, did not always result in an improvement with respect to performance of the neural networks. These smaller PUNNs networks did not always produce good training errors and generalization compared to the larger architectures of SUNNs. However, global optimization using SUs showed an improvement in performance compared to gradient descent using SUs. In certain instances the PUNNs outperformed the SUNNs.

In general, PUNNs did not show a remarkable gain in performance, other than reaching lower generalization levels faster than gradient descent applied to SUNNs. One has to consider the trade-off between (a) added complexity when using PUNNs due to exponentiation and (b) the larger architecture required by SUNNs, before deciding on implementing a neural network using either PUs or SUs.

6.1 Possible Improvements and Future Research

The following aspects are suggested for future research:

1. The learning profiles in chapter 4 reflected that PSO and GA applied to PUNNs had larger reductions in error early in training, reaching low errors using substantially less training epochs than gradient descent applied to SUNNs. This suggests using the global optimization algorithms for initial training to produce weights close to the global minima on the error surface. Once the area where the global

minimum occurs is reached, gradient descent can then be applied to further train the PUNN to completion, an approach which has been used successfully in the past using different optimization algorithms.

2. PSO can be enhanced by incorporating constriction coefficients in the algorithm, which have lead to improved performance as reported by Eberhart *et al* [Eberhart *et al* 2000].

3. The variance nullity algorithm applied to PUNNs can be improved by avoiding re-initialization of weights in cases where no parameters were pruned by retaining the unpruned weights and to continue the pruning process by re-training only the bias.

4. An investigation into the overfitting tendencies of PSO, GA and LFOP to identify the algorithm that exhibits the smallest degree of overfitting and consequently the best generalization. These results can then be compared to results obtained by Lawrence and Giles that showed that the Scaled Conjugate Gradient algorithm tended to overfit more than gradient descent [Lawrence *et al* 2000].

Bibliography

[Ackley *et al* 1985] DG Ackley, G Hinton and T Sejnowski, *A Learning Algorithm for Boltzmann Machines*, Cognitive Science, 9, pp 147-169, 1985.

[Aleksander *et al* 1990] I Aleksander and H Morton, *An Introduction to Neural Computing*, Chapman and Hall, 1990.

[Baker 1987] JA Baker, *Reducing Bias and Inefficiency in the Selection Algorithm*, Proceedings of the Second International Conference on Genetic Algorithms: Genetic Algorithms and Their Applications, Lawrence ErlBaum Associates, 1987.

[Baldi 1991] P Baldi, *Computing with Arrays of Bell-Shaped and Sigmoid Functions*, Neural Information Processing Systems, 3, (RP Lippmann, JE Moody and DS Touretzky (eds.)), Morgan Kaufmann, pp 735-742, 1991.

[Barto 1992] AG Barto, *Reinforcement Learning and Adaptive Critic Methods*, in Handbook of Intelligent Control, (DA White and DA Sofge (eds.)), Van Nostrand Reinhold, pp 469-491, 1992.

[Battiti 1992] R Battiti, *First- and Second-OrderMethods for Learning: Between Steepest Descent and Newton's Method*, Neural Computation, 4, pp 141-166, 1992.

[Baum *et al* 1989] EB Baum and D Haussler, *What Size Net Gives Valid Generalizations?*, Neural Computation, 1, pp 151-160, 1989.

[Billings *et al* 1995] SA Billings and GL Zheng, *Radial Basis Function Network Configuration Using Genetic Algorithms,* Neural Networks, 8(6), pp 877-890, 1995.

[Bilbro *et al* 1989] GL Bilbro and WE Snyder, *Range Image Restoration using Mean Field Annealing,* (DS Touretzky, ed.), Advances in Neural Information Processing Systems, 1, Morgan Kaufmann, pp 594-601, 1989.

[Bryson *et al* 1969] AE Bryson and Y-C Yo, *Applied Optical Control,* Hemisphere Publishing, 1969.

[Burdsall *et al* 1997] B Burdsall and C Giraud-Carrier, *GA-RBF: A Self-Optimizing RBF Network,* In Proceedings of the Third International Conference on Artificial Neural Networks and Genetic Algorithms, Springer-Verlag, pp 348-351, 1997.

[Chakraborty *et al* 1992] K Chakraborty, K Mehtotra, CK Mohan and S Ranka, *Forecasting the Behavior of Multivariate Time Series using Neural Networks,* Neural Networks, 5, pp 961-970, 1992.

[Chan *et al* 1987] LW Chan and F Fallside, *An Adaptive Training Algorithm for Backpropagation Networks,* Nature, 264, pp 705-712, 1987.

[Chang *et al* 1991] EI Chang and RP Lippmann, *Using Genetic Algorithms to Improve Pattern Classification Performance,* In Advances in Neural Information processing Systems, 3, Morgan Kaufmann, pp 797-803, 1991.

[Churchland *et al* 1992] PS Churchland and TJ Sejnowski, *The Computational Brain,* MIT Press, 1992.

[Cibas *et al* 1996] T Cibas, F Fogelman, P Soulié and S Raudys, *Variable selection with Neural Networks,* Neurocomputing, 12, pp 223-248, 1996.

[Cohen *et al* 1993] M Cohen, H Franco, N Morgan, D Rumelhart and V Abrash, *Context Dependent Multiple Distribution Phonetic Modeling with MPLs,* In Advances

in Neural Information Processing Systems, 5, (SJ Hanson, JD Cowan and C Lee Giles (eds.)), Morgan Kaufmann, pp 649-657, 1993.

[Cybenko 1969] G Cybenko, *Approximation by Superpositions of a Sigmoidal Function,* Mathematical Control Signals Systems, 2, pp 203-204, 1969.

[Dagli *et al* 1995] CH Dagli and J James, *Use of Genetic Algorithms for Encoding Efficient Neural Network Architectures: Neuro-Computer Implementation,* In SPIE Proceedings of Applications and Science of Artificial Neural Networks, 2492, pp 323-330, 1995.

[Davis 1991] L Davis, *Handbook of Genetic Algorithms,* Von Nostrand Reinhold, 1991.

[De Jong 1975] KA De Jong, *An Analysis of the Behavior of a Class of Genetic Adaptive Systems,* PhD Thesis, University of Michigan, 1975.

[Durbin *et al* 1989] R Durbin and DE Rumelhart, *Product units: A Computationally Powerful and Biologically Plausible Extension to Backpropagation Networks,* Neural Computation, 1, pp 133-142, 1989.

[Eberhart *et al* 1990] RC Eberhart and RW Dobbins, *Case Study I: Detection of Electrocephalogram Spikes,* in Neural Networks PC Tools, (RC Eberhart, RW Dobbins (eds.)), Academic Press, 1990.

[Eberhart *et al* 1995] RC Eberhart and J Kennedy, *A New Optimizer Using Particle Swarm Theory,* Proceedings of the Sixth International Symposium on Micro Machine and Human Science, IEEE Service Center, Piscataway, NJ, pp 39-43, 1995.

[Eberhart *et al* 1996] RC Eberhart, P Simpson and R.W Dobbins, *Computational Intelligence PC Tools,* Academic Press Limited, 1996.

[Eberhart *et al* 1998] RC Eberhart, Z He, C Wei, L Yang, X Gao, S Yao and Y Shi, *Extracting Rules from Fuzzy Neural Networks by Particle Swarm Optimization,* IEEE International Conference on Evolutionary Computation, Alaska, 1998.

[Eberhart *et al* 1999] RC Eberhart and X Hu, *Human Tremor Analysis using Particle Swarm Optimization,* In Proceedings of Congress on Evolutionary Computation 1999, Piscataway, pp 1951-1957, 1999.

[Eberhart *et al* 2000] RC Eberhart and Y Shi, *Comparing Inertia Weights and Constriction Factors in Particle Swarm Optimization,* In Proceedings of International Congress on Evolutionary Computation 2000, San Diego, CA, pp 84-88, 2000.

[Elman 1990] JL Elman, *Finding Structure in Time*, Cognitive Science, 14, pp 179-211, 1990.

[Engelbrecht *et al* 1995a] AP Engelbrecht, I Cloete, J Geldenhuys, JM Zurada, *Automatic Scaling using Gamma Learning in Feedforward Neural Networks*, In International Workshop on Artificial Neural Networks, (J Mira and F Sandoval (eds.)), 'From Natural Science to Artificial Neural Computing' in the series 'Lecture notes in Computer Science', 930, pp 374-381, 1995.

[Engelbrecht *et al* 1995b] AP Engelbrecht, I Cloete and JM Zurada, *Determining the Significance of Input Parameters using Sensitivity Analysis,* International Workshop on Artificial Neural Networks, In 'From Natural Science to Artificial Neural Computing' (J Mira and F Sandoval (eds.)), In the Series 'Lecture Notes in Computer Science', 930, pp 382-388, 1995.

[Engelbrecht *et al* 1999a] AP Engelbrecht and A Ismail, *Training Product Unit Neural Networks,* Stability and Control: Theory and Applications, 2 (1/2), pp 59-74, 1999.

[Engelbrecht *et al* 1999b] AP Engelbrecht and I Cloete, *A Sensitivity Analysis Algorithm for Pruning Feedforward Neural Networks,* IEEE International Conference on Neural Networks, 2, Washington DC, USA, pp 1274-1277, 1996.

[Engelbrecht *et al* 1999c] AP Engelbrecht, L Fletcher, I Cloete, *Variance Analysis of Sensitivity Information for Pruning Multilayer Feedforward Neural Networks,* IEEE International Joint Conference on Neural Networks, Washington DC, USA, paper 379, 1999.

[Engelbrecht *et al* 1999d] AP Engelbrecht and I Cloete, *Incremental Learning using Sensitivity Analysis,* IEEE Joint Conference on Neural Networks, Washington, paper 380, 1999.

[Engelbrecht 2001] AP Engelbrecht, *A New Pruning Heuristic Based on Variance Analysis of Sensitivity Information,* IEEE Transactions on Neural Networks, 12(6), 2001.

[Fahlman 1989] SE Fahlman, *Fast Learning Variations on Back-Propagation: An Empirical Study,* In Proceedings of the 1988 Connectionist Models Summer School (Pittsburgh, 1988), (D Touretzky, G Hinton and T Sejnowski (eds.)), Morgan Kaufmann, pp 38-51, 1989.

[Fahlman *et al* 1990] SE Fahlman and C Lebiere, *The Cascade-Correlation Learning Architecture,* In Advances in Neural Information Processing Systems 2 (DS Touretzky, ed.), Morgan Kaufmann, pp 524-532, 1990.

[Fausett 1994] L Fausett, *Fundamentals of Neural Networks: Architectures, Algorithms and Applications,* Prentice Hall, 1994.

[Finnoff 1993a] W Finnoff, *Diffusion Approximations for the Constant Learning Rate of Backpropagation Algorithm and Resistance to Local Minima,* Advances in Neural

Information Processing Systems, 5, (SJ Hanson, JD Cowan and C Lee Giles (eds.)), Morgan Kaufmann, pp 459-466, 1993.

[Finnoff 1993b] W Finnoff, F Hergert and HG Zimmermann, *Improving Model Selection by Nonconvergent Methods,* Neural Networks, 6, pp 771-783, 1993.

[Fletcher *et al* 1995] J Fletcher and Z Obradovic, *A Discrete Approach to Constructive Neural Network Learning,* Neural, Parallel and Scientific Computations, 3(3), pp 307-320, 1995.

[Foo *et al* 1988] YPS Foo and Y Takefuji, *Integer Linear Programming Neural Networks for Job-Shop Scheduling,* Proceedings of the IEEE International Conference on Neural Networks, San Diego, California, 1988.

[Franzini 1987] MA Franzini, *Speech Recognition with Back Propagation,* Proceedings of the Nineth Annual Conference of the IEEE Engineering in Medicine and Biology Society, New York, pp 1702-1703, 1987.

[Frean 1990] M Frean, *The Upstart Algorithm: A Method for Constructing and Training Feedforward Neural Networks,* Neural Computation, 2, pp 198-209, 1990.

[Fu *et al* 1993] L Fu and T Chen, *Sensitivity Analysis for Input Vector in Multilayer Feedforward Neural Networks,* IEEE International Conference on Neural Networks, 1, 1993, pp 215-218, 1993.

[Fukushima 1975] K Fukushima *Cognitron: A Self-Organizing Multilayered Neural Network,* Biological Cybernetics, 20, pp 121-136, 1975.

[Funahashi 1989] KI Funahashi, *On the Approximate Realization of Continuous Mappings by Neural Networks*, Neural Networks, 2, pp 183-192, 1989.

[Gallant 1992] AR Gallant and H White, *On Learning the Derivatives of an Unknown Mapping with Multilayer Feedforward Networks*, Neural Networks, 5, pp 129-138, 1991.

[Gallant 1986] SI Gallant, *Three Constructive Algorithms for Network Learning,* In Proceedings of the Eighth Annual Conference of the Cognitive Science Society, pp 652-660, 1986.

[Ghosh *et al* 1992] J Ghosh and Y Shin, *Efficient High-Order Neural Networks for Classification and Function Approximation,* International Journal of Neural Systems, 3(4), pp 323-325, 1992.

[Ghosh *et al* 1994] J Ghosh and K Turner, *Structural Adaptation and Generalization in Supervised Feed-Forward Networks,* Journal of Artificial Neural Networks, 1(4), pp 431-458, 1994.

[Girosi *et al* 1995] F Girosi and T Poggio, *Regularization Theory and Neural Networks Architectures*, Neural Computation, 7, pp 219-269, 1995.

[Goldberg 1989] DE Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, 1989.

[Grossberg 1987] S Grossberg, *Competitive Learning: from Interactive Activation to Adaptive Resonance,* Cognitive Science, 11, pp 23-63, 1987.

[Guo *et al* 1992] Z Guo and RO Uhrig, *Use of Genetic Algorithms to Select Inputs for Neural Networks,* In Proceedings on Combination of Genetic Algorithms and Neural Networks, pp 223-234, 1992.

[Gurney 1992] KN Gurney, *Training Nets of Hardware Realizable Sigma-Pi Units*, Neural Networks, 5, pp 289-303, 1992.

[Guyon 1990] I Guyon, *Neural Networks and Applications*, Computer Physics Reports, Amsterdam: Elsevier, 1990.

[Haffner *et al* 1988] P Haffner, A Waibel, H Sawai and K. Shikano, *Fast Back-Propagation Learning Methods for Neural Networks in Speech,* Technical Report TR-1-0058, ATR Interpreting Telephony Research Laboratories, Osaka, Japan, 1988.

[Harrison *et al* 1991] R Harrison, S Marshall and R Kennedy, *The Early Diagnosis of Heart Attacks: A Neurocomputational Approach,* IEEE International Joint Conference on Neural Networks, 1, Seattle, pp 1-5, 1991.

[Hanson *et al* 1989] SJ Hanson and LY Pratt, *Comparing Biases for Minimal Network Construction with Back-Propagation*, Advances in Neural Information Processing Systems, 1, (DS Touretzky ed.), pp 177-185, 1989.

[Hassibi *et al* 1994] B Hassibi and DG Stork, *Second Order Derivatives for Network Pruning: Optimal Brain Surgeon,* Advances in Neural Information Processing Systems, 5, (SJ Hanson, JD Cowan, C Lee Giles (eds.)), Morgan Kaufmann, 1994.

[Hassoun 1995] MH Hassoun, *Fundamental Artificial Neural Networks,* MIT Press, 1995.

[Haykin *et al* 1992] S Haykin and TK Bhattacharya, *Adaptive Radar Detection Using Supervised Learning Networks,* Computational Neuroscience Symposium, Indiana University - Purdue University at Indianapolis, pp 35-51, 1992.

[Haykin 1994] S Haykin, *Neural Networks: A Comprehensive Foundation*, MacMillan Publishing Company, 1994.

[Heppner *et al* 1990] F Heppner and U Grenander, *A Stochastic Nonlinear Model for Coordinated Bird Flocks*, in The Ubiquity of Chaos (S Kraner, ed.), AAAS Publications, 1990.

[Hinton 1987] GE Hinton, *Connectionist Learning Procedures,* Reproduced in Machine Learning: Paradigms and Methods (J Carbonell ed.), MIT Press, pp 185-234, 1990.

[Hirose *et al* 1991] Y Hirose, K Yamashita and S Hijiya, *Back-propagation Algorithm which Varies the Number of Hidden Units*, Neural Networks, 4, pp 61-66, 1991.

[Holland 1992] JH Holland, *Adaptation in Natural and Artificial Systems,* MIT Press, 1992.

[Holm *et al* 1999] JEW Holm and EC Botha, *Leap-frog is a Robust Algorithm for Training Neural Networks,* Network Computation in Neural Systems, 10, pp 1-13, 1999.

[Holtzman 1992] JM Holtzman, *On Using Perturbation Analysis to do Sensitivity Analysis: Derivatives versus Difference,* IEEE Transactions on Automatic Control, 37(2), pp 243-247, 1992.

[Hopfield 1982] JJ Hopfield, *Neural Networks and Physical Systems with Emergent Collective Computational Properties,* Proceedings of the National Academy of Sciences of the USA, 79, pp 2554-2588, 1982.

[Hornik *et al* 1989a] K Hornik, M Stinchcombe and H White, *Universal Approximation of an Unknoum Mapping and its Derivatives Using Multilayer Feedforward Networks*, Neural Networks, 3, pp 551-560, 1989.

[Hornik *et al* 1989b] K Hornik, M Stinchcombe and H White, *Multilayer Feedforward Networks are Universal Approximators,* Neural Networks, 2(5), pp 359-366.

[Hsieh *et al* 1998] WW Hsieh and B Tang, *Applying Neural Network Models to Prediction and Data Analysis in Meteorology and Oceanography,* Bulletin of the American Meteorological Society, 79, pp 1855-1870, 1998.

[Hush *et al* 1991] DR Hush, JM Salas and B Horne, *Error Surfaces for Multilayer Perceptrons,* IEEE International Joint Conference on Neural Networks, Seattle, I, pp 759-764, 1991.

[Hush *et al* 1993] DR Hush and GG Horne, *Progress in Supervised Neural Networks: What's New Since Lippmann?,* IEEE Signal Processing Magazine, 10, pp 8-39, 1993.

[Hussain *et al* 1997] A Hussain, JJ Soraghan and TS Durbani, *A New Neural Network for Nonlinear Time-Series Modelling,* Neurovest Journal, pp 16-26, 1997.

[Ismail *et al* 1999] A Ismail and AP Engelbrecht, *Training Product Units in Feedforward Neural Networks using Particle Swarm Optimization,* In Development and Practice of Artificial Intelligence Techniques, (VB Bajic and D Sha (eds.)), Proceedings of the International Conference on Artificial Intelligence, pp 36-40, 1999.

[Ismail *et al* 2000] A Ismail and AP Engelbrecht, *Global Optimization Algorithms for Training Product Unit Neural Networks,* IEEE International Joint Conference on Neural Networks, paper 032, 2000.

[Janson *et al* 1993] DJ Janson and JF Frenzel, *Training Product Unit Neural Networks with Genetic Algorithms,* IEEE Expert Magazine, pp 26-33, 1993.

[Jordan 1986] *Attractor Dynamics and Parallelism in a Connectionist Sequential Machine,* In Proceedings of the Eighth Annual Conference of the Cognitive Science Society, Erlbaum, pp 531-546, 1986.

[Jordan *et al* 1990] MI Jordan and RA Jacobs, *Learning to Control an Unstable System with Forward Modeling,* In Advances in Neural Information Processing Systems, 2 (DS Touretzky, ed.), Morgan Kaufmann, pp 324-331, 1990.

[Karnin 1990] ED Karnin, *A Simple Procedure for Pruning Back-Propagation Trained Neural Networks,* IEEE Transactions on Neural Networks, 1(2), pp 239-242, 1990.

[Kawato 1990] M Kawato, *Computational Schemes and Neural Network Models for Formation and Control of Multijoint Arm Trajectory,* In Neural Networks for Robotics and Control, (T Miller, R Sutton and P Werbos (eds.)), MIT Press, 1990.

[Kennedy 1995a] J Kennedy, *The Particle Swarm: Social Adaptation of Knowledge,* Proceedings of the IEEE International Conference on Evolutionary Computation, Indianapolis, Indiana, IEEE Service Center, Piscataway, NJ, pp 303-308, 1995.

[Kennedy *et al* 1995b] J Kennedy and RC Eberhart, *Particle Swarm Optimization,* Proceedings of the IEEE International Conference on Neural Networks (Perth, Australia), IEEE Service Center, Piscataway, NJ, IV, pp 1942-1948, 1995.

[Kirkpatrick *et al* 1983] S Kirkpatrick, C Gelatt and M Vecchi, *Optimization by Simulated Annealing*, Science, 220, pp 671-680, 1983.

[Klassen *et al* 1988] MS Klassen and YH Pao, *Characteristics of the Functional-link Net: A Higher Order Delta Rule Net,* IEEE Proceedings of Second Annual International Conference on Neural Networks, 1988.

[Kohonen 1988a] T Kohonen, *The Neural Phonetic Typewriter,* IEEE Computer, 27(3), pp 11-22, 1988.

[Kohonen 1988b] T Kohonen, *Self-organization and Associative Memory,* Springer-Verlag, 1988.

[Kolen *et al* 1990] JF Kolen and JB Pollack, *Back Propagation is Sensitive to Initial Conditions,* Technical Report TR 90-JK-BPSIC, Laboratory for Artificial Intelligence Research, Computer and Information Science Department, The Ohio State University, Columbus, 1990.

[Kramer *et al* 1989] AH Kramer and A Sangiovanni-Vincentelli, *Efficient Parallel Learning Algorithms for Neural Networks,* Advances in Neural Information Processing Systems, 1, (DS Touretzky, ed.), Morgan Kaufmann, pp 40-48, 1989.

[Kuo *et al* 1994] IE Kuo and SS Melsheimer, *Using Genetic Algorithms to Estimate the Optimum Width Parameter in Radial Basis Function Networks,* In Proceedings of the 1994 American Control Conference, 1994.

[Kwok *et al* 1995] T-Y Kwok and D-Y Yeung, *Constructive Feedforward Neural Networks for Regression Problems: A Survey,* Technical Report HKUST-CS95-43, Department of Computer Science, The Hong Kong University of Science and Technology, 1995.

[Lange *et al* 1996] S Lange and T Zeugmann, *Incremental Learning from Positive Data,* Journal of Computer and System Sciences, 53, pp 88-103, 1996.

[Lawrence *et al* 2000] S Lawrence and C Lee Giles, *Overfitting and Neural Networks: Conjugate Gradient and Backpropagation,* In Proceedings of the IEEE International Joint Conference on Neural Networks, Como, Italy, July 2000.

[Le Cun 1989] Y Le Cun, *Generalization and Network Design Strategies,* In Connectionism in Perspective, (R Pfeifer, Z Schreter, F Fogelman-Soulié and L. Steels (eds.)), Amsterdam: North-Holland, pp 143-155, 1989.

[Le Cun *et al* 1990] Y Le Cun, JS Denker and SA Solla, *Optimal Brain Damage,* Advances in Neural Information Processing Systems, 2, (DS Touretzky, ed.), Morgan Kaufmann, pp 598-605, 1990.

[Lee *et al* 1991] Y Lee, S Oh and M Kim, *The Effect of Initial Weights on Premature Saturation in Back-propagation Learning*, International Joint Conference on Neural Networks, 1, Seattle, pp 765-770, 1991.

[Lee Giles 1987] C Lee Giles, *Learning, Invariance and Generalization in Higher-Order Neural Networks*, Applied Optics, 26(23), pp 4972-4978, 1987.

[Leerink *et al* 1995] LR Leerink, C Lee Giles, BG Horne and MA Jabri, *Learning with Product Units*, Advances in Neural Information Processing Systems, (G Tesauro, D Touretzky and T Leen (eds.)), 7, pp 537-544, 1995.

[Li *et al* 1996] JY Li and TWS Chow, *Functional Approximation of Higher-Order Neural Networks,* Journal of Intelligent Systems, (R Paul and R Macredie (eds.)), 6, 1996.

[MacLeod 1990] K MacLeod, *An Application Specific Neural Model for Document Clustering,* In Proceedings of the Fourth Annual Parallel Processing Symposium, 1, pp 5-16, 1990.

[Maxwell *et al* 1986] T Maxwell, C Lee Giles, YC Lee and HH Chen, *Nonlinear Dynamics of Artificial Neural Systems*, In Neural Networks For Computing, (J Denker (ed.)), New York: American Institute of Physics, p 299, 1986.

[Milenković *et al* 1996] S Milenković, Z Obradović and V Litovski, *Annealing Based Dynamic Learning in Second-Order Neural Networks*, Technical Report, Department of Electronic Engineering, University of Nis, Yugoslavia, 1996.

[Minsky 1961] ML Minsky, *Steps Towards Artificial Intelligence,* In Proceedings of the Institute of Radio Engineers, 49, pp 8-30, 1961.

[Minsky *et al* 1969] ML Minsky and SA Papert, *Perceptrons: An Introduction to Computational Geometry,* MIT Press, 1988.

[Mitchel 1996] M Mitchel, *An Introduction to Genetic Algorithms,* MIT-Press, 1996.

[Moody *et al* 1996] J Moody and PJ Antsaklis, *The Dependence Identification Neural Network Construction Algorithm,* IEEE Transactions on Neural Networks, 7(1), pp 3-15, 1996.

[Moreira *et al* 1995] M Moreira and E Fiesler, *Neural Networks with Adaptive Learning Rate and Momentum Terms,* In Technical Report 95-04 Institut Dalle Molle d'Intelligence Artificielle Perceptive, Martigny, Switzerland, 1995.

[Mozer *et al* 1989] MC Mozer and P Smolensky, *Skeletonization: A Technique for Trimming the Fat from a Network Via Relevance Assessment,* Advances in Neural Information Processing, 1, (DS Touretzky ed.), Morgan Kaufmann, pp 107-115, 1989.

[Nigrin 1993] A Nigrin, *Neural Networks for Pattern Recognition,* MIT Press, p 11, 1993.

[Oh *et al* 1995] S-H Oh and Y Lee, *Sensitivity Analysis of Single Hidden-Layer Neural Networks with Threshold Functions,* IEEE Transactions on Neural Networks, 6(4), pp 1005-1007, 1995.

[Opitz *et al* 1994] DW Opitz and JW Shavlik, *Genetically Refining Topologies of Knowledge-based Neural Networks,* In International Symposium on Integrating Knowledge and Neural Heuristics, pp 57-66, 1994.

[Pao 1989] YH Pao, *Adaptive Pattern Recognition and Neural Networks,* Addison Wesley Publishing, 1989.

[Pao *et al* 1992] YH Pao and Y Takefuji, *Functional-Link Net Computing: Theory, System Architecture and Functionalities*, IEEE Computer, 25(5), pp 76-79, 1992.

[Parker 1985] DB Parker, *Learning-logic: Casting the Cortex of the Human Brain in Silicon*, Technical Report TR-47, Center for Computational Research in Economics and Management Science, MIT, Cambridge, 1985.

[Pitts *et al* 1943] W Pitts and WS McCulloch, *A Logical Calculus of the Ideas Imminent in Nervous Activity*, Bulletin of Mathematical Biophysics, 5, pp 115-133, 1943.

[Poggio *et al* 1990] T Poggio and F Girosi, *Networks for Approximation and Learning*, In Proceedings of the IEEE, 78(9), pp 1481-1497, 1990.

[Pomerlau 1989] DA Pomerlau, *ALVINN: An Autonomous Land Vehicle in a Neural Network*, Advances in Neural Information Processing, 1, (D Touretzky ed.), Morgan-Kaufmann, 1989.

[Prechelt 1994] L Prechelt, *PROBEN1- A Set of Benchmarks and Benchmarking Rules for Neural Network Training Algorithms*, Technical Report 21/94, Fakultät für Informatik, Universität Karlsruhe, D-76128 Karlsruhe, Germany, Anonymous FTP:/pub/papers/techreports/1994/1994-21.ps.Z on ftp.ira.uka.de, 1994.

[Redding *et al* 1993] NJ Redding, A Kowalczyk and T Downs, *Constructive Higher-Order Network Algorithm that is Polynomial in Time*, Neural Networks, 6, pp 997-1010, 1993.

[Reed 1994] R Reed, *Pruning Algorithms - A Survey*, IEEE Transactions on Neural Networks, 4(5), pp 740-747, 1993.

[Reynolds 1987] CW Reynolds, *Flocks, Herds and Schools: A Distributed Behavioral Model*, Computer Graphics, 21(4), pp 25-34, 1987.

[Rojas 1996] R Rojas, *Neural Networks: A Systematic Introduction,* Springer-Verlag, 1996.

[Rosenblatt 1962] F Rosenblatt, *Principles of Neurodynamics,* Spartan Books, 1962.

[Rowley *et al* 1996] H Rowley, S Baluja, and T Kanade, *Human Face Detection in Visual Scenes,* Advances in Neural Information Processing Systems, 8, pp 875-881, 1996.

[Rumelhart *et al* 1985] DE Rumelhart and Zipser, *Feature Discovery by Competitive Learning,* Cognitive Science, 9, pp 75-112, 1985.

[Rumelhart *et al* 1986a] DE Rumelhart and JL McClelland, *Parallel Distributed Processing: Exploration in the Microstructure of Cognition,* 1, MIT Press, 1986.

[Rumelhart *et al* 1986b] DE Rumelhart, GE Hinton and RJ Williams. *Learning Internal Representations by Back-propagation Errors,* Nature, 323, pp 533-536, 1986.

[Saund 1989] E Saund, *Dimensionality Reduction Using Connectionist Networks,* IEEE Transactions on Pattern Analysis and Machine Intelligence, 11, pp 304-314, 1989.

[Salerno 1997] J Salerno, *Using the Particle Swarm Optimization Technique to Train a Recurrent Neural Model,* Proceedings of the Nineth IEEE International Conference on Tools with Artificial Intelligence, 1997.

[Schalkoff 1997] RJ Schalkoff, *Artificial Neural Networks,* McGraw-Hill Series in Computer Science, 1997.

[Schiffman *et al* 1992] W Schiffmann, M Joost and R Werner, *Synthesis and Performance Analysis of Multilayer Neural Network Architectures,* Technical Report, Institut für Physik, Universität Koblenz, 16/1992, 1992.

[Sejnowski *et al* 1987] TJ Sejnowski and CR Rosenberg, *Parallel Networks that Learn to Pronounce English Text,* Complex Systems, 1, pp 145-168, 1987.

[Shi *et al* 1998] Y Shi and RC Eberhart, *Parameter Selection in Particle Swarm Optimization,* The Seventh Annual Conference on Evolutionary Programming, pp 591-600, 1998.

[Shi *et al* 1999] Y Shi and RC Eberhart, *Empirical Study of Particle Swarm Optimization,* In Proceedings of Congress on Evolutionary Computation 1999, pp 1945-1950, 1999.

[Shin *et al* 1995] Y Shin and J Ghosh, *Ridge Polynomial Networks,* IEEE Transactions on Neural Networks, 6(2), pp 610-622, 1995.

[Shittenkopf *et al* 1997] C Schittenkopf, G Deco and W Brauer, *Two Strategies to Avoid Overfitting in Feedforward Neural Networks*, Neural Networks, 10(30), pp 505-516, 1997.

[Sietsma *et al* 1988] J Sietsma and RJF Dow, *Neural Net Pruning - Why and How,* IEEE International Conference on Neural Networks, I, pp 325-333, 1988.

[Sietsma *et al* 1991] J Sietsma and RJF Dow, *Creating Artificial Neural Networks that Generalize,* Neural Networks, 4(1), pp. 67-69, 1991.

[Silva *et al* 1990] FM Silva and LB Almeida, *Acceleration Techniques for the Backpropagation Algorithm,* Neural Networks, Europe Lecture Notes in Computer Science (LB Almeida and Wellekens (eds.)), Springer-Verlag, pp 110-119, 1990.

[Snyman 1982a] JA Snyman, *A New and Dynamic Method for Unconstrained Minimization,* Applied Mathematical Modelling, 6, pp 449-462, 1982.

[Snyman 1982b] JA Snyman, *An Improved Version of the Original LeapFrog Dynamic Method for Unconstrained Minimization: LFOP1(b),* Applied Mathematical Modelling, 7, pp 216-218, 1983.

[Stinchcombe *et al* 1989] M Stinchcombe and H White, *Universal Approximations using Feedforward Networks with Non-Sigmoid Hidden Layer Activation Functions,* Proceedings of the International Joint Conference on Neural Networks, I, pp 613-617, 1989.

[Sutton 1986] R Sutton, *Two Problems with Backpropagation and Other Steepest-Descent Learning Procedures for Networks,* In Proceedings of the 8th Annual Conference on the Cognitive Science Society, Amherst, pp 823-831, 1986.

[Syswerda 1991] G Syswerda *Schedule Optimization using Genetic Algorithms,* In Handbook of Genetic Algorithms, (L Davis (ed.)), Van Nostrand Reinhold, 1991.

[Thimm *et al* 1995] G Thimm and E Fiesler, *Evaluating Pruning Methods,* Proceedings of the International Symposium on Artificial Neural Networks, pp 20-25, 1995.

[Van den Bergh 1999] F van den Bergh, *Particle Swarm Weight Initialization in Multi-layer Perceptron Artificial Neural Networks,* In Development and Practice of Artificial Intelligence Techniques, Proceedings of International Conference on Artificial Intelligence, pp 41-45, 1999.

[Van den Bergh *et al* 2000] F van den Bergh and AP Engelbrecht, *Cooperative Learning in Neural Networks using Particle Swarm Optimizers,* South African Computer Journal, 26, pp 84-90, 2000.

[Van den Bergh 2001a] F van den Bergh, *Analysis of Particle Swarm Optimizers,* PhD Thesis, Department of Computer Science, University of Pretoria, submitted 2001.

[Van den Bergh *et al* 2001b] F van den Bergh and AP Engelbrecht, *Using Cooperative Particle Swarm Optimization to Train Product Unit Neural Networks,* in Proceedings of International Joint Conference on Neural Networks, Washington, 2001.

[Van den Bergh *et al* 2001c] F van den Bergh, AP Engelbrecht and DG Kourie, *A Convergence Proof for Particle Swarm Optimizers,* submitted to IEEE Transactions on Evolutionary Computing, 2001.

[Van den Bergh *et al* 2001d] F van den Bergh and AP Engelbrecht,*Effects of Swarm Size on Cooperative Particle Swarm Optimizers,* In Proceedings of Genetic and Evolutionary Computation Conference 2001, San Francisco, 2001.

[Von Lehman *et al* 1988] A Von Lehman, EG Liao, PF Marrakchi and JS Patel, *Factors Influencing Learning by Back-propagation,* IEEE International Conference on Neural Networks, I, pp 765-770, 1988.

[Weigend *et al* 1991] AS Weigend, DE Rumelhart and BA Huberman, *Generalization by Weight Elimination with Application to Forecasting,* Advances in Neural Information Processing Systems, 3, (R Lippman and J Moody and DS Touretzky (eds.)), pp 872-882, 1991.

[Werbos 1974] PJ Werbos, *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences,* Ph.D Thesis, Harvard University, Cambridge, 1974.

[Werbos 1989] PJ Werbos, *Backpropagation and Neurocontrol: A Review and Prospectus,* International Joint Conference on Neural Networks, 1, pp 209-216, 1989.

[Wessels *et al* 1992] LFA Wessels and E Barnard, *Avoiding False Local Minima by Proper Initialization of Connections,* IEEE Transactions on Neural Networks, 3(6), pp 899-905, 1992.

[Whitehead *et al* 1996] BA Whitehead and TD Choate, *Cooperative-Competitive Genetic Evolution of Radial Basis Function Centers and Widths for Time Series Prediction,* IEEE Transactions on Neural Networks, 7(4), pp 869-881, 1996. D.

[Whitley *et al* 1990] D Whitley and C Bogart, *The Evolution of Connectivity: Pruning Neural Networks using Genetic Algorithms*, International Joint Conference on Neural Networks, 1, pp 134-137, 1990.

[Widrow *et al* 1960] B Widrow and ME Hoff, *Adaptive Switching Circuits,* IRE WESCON Convention Record, pp 96-104, 1960.

[Wieland *et al* 1987] A Wieland and R Leighton, *Geometric Analysis of Neural Network Capabilities*, First IEEE International Conference on Neural Networks, 3, pp 385-392, 1987.

[Wilson 1975] EO Wilson, *Sociology: The New Synthesis*, Belknap Press, 1975.

[Zamparelli 1997] M Zamparelli, *Genetically Trained Cellular Networks,* Neural Networks, 10(6), pp 1143-1151, 1997.

[Zhang 1994] B-T Zhang, *Accelerated Learning by Active Example Selection,* International Journal of Neural Systems, 5(1), pp 67-75, 1994.

[Zhang *et al* 1997] J Zhang and A Morris, *A Sequential Learning Approach for Single Hidden Layer Neural Networks*, Neural Networks, 11, pp 65-80, 1997.

[Zurada 1992] JM Zurada, *Introduction to Artificial Neural Systems,* West Publishing Company, 1992.

[Zurada *et al* 1997] JM Zurada, A Malinowski and S Usui, *Perturbation Method for Deleting Redundant Inputs of Perceptron Networks,* Neurocomputing, 4, pp 177-193, 1997.

Appendix A

Derivation of learning rules for PUNNs

The product learning equations for the feed-forward neural network type used in this thesis are derived in this appendix. This thesis assumes a network architecture which consists of an input layer, a hidden layer consisting of product units and an output layer consisting of summation units. Linear activations are assumed for all units. This thesis assumes a PUNN architecture with a bias to the output units and no bias to the hidden units. Instead, an extra unit referred to as a 'distortion unit', is included to the hidden units. For a discussion on the 'distortion unit', refer to section 3.7.3 on page 72. Section A.1 derives the learning equations for a PUNN architecture with biases to both the hidden and output units. In section A.2 the equations of section A.1 are then adapted for a PUNN where a 'distortion unit' replaces the bias in the hidden layer. The derivations assume gradient descent as optimization algorithm and on-line learning.

The mean squared error (MSE) function is assumed as the objective function, with linear activation functions in both, the hidden and output layers of the product unit neural network (PUNN).

The objective function is expressed as,

$$E = \frac{\sum_{p=1}^{P} E_p}{PK} \tag{A.1}$$

where P is the total number of patterns in the training set, K is the number of output units, and E_p is the error of pattern p, defined as

$$E_p = \frac{1}{2} \cdot \sum_{k=1}^{K} (t_{k,p} - o_{k,p})^2 \tag{A.2}$$

where $t_{k,p}$ and $o_{k,p}$ are respectively the target and actual output values of the k^{th} output unit, O_k, when pattern p is presented to the neural network.

The derivations in this appendix refer to individual patterns. For the sake of notational convenience the superscript p, that refers to a specific pattern, is removed. Throughout this appendix I, J and K refer, respectively, to the number of input, hidden and output units excluding biases.

The output of the k^{th} output unit is (under the assumption of linear activated outputs)

$$\begin{aligned} o_k &= f(net_{o_k}) \\ &= net_{o_k} \end{aligned} \tag{A.3}$$

and

$$f'(net_{o_k}) = 1 \tag{A.4}$$

The net input signal is calculated as

$$net_{o_k} = \sum_{j=1}^{J+1} w_{kj} y_j \tag{A.5}$$

The $(J+1)^{th}$ unit represents the bias to each output unit; w_{kj} is the weight between the j^{th} hidden and k^{th} output units; y_j is the output of the j^{th} hidden unit, defined as

(assuming linear activation)

$$y_j = f(net_{y_j}) \tag{A.6}$$

$$= net_{y_j} \tag{A.7}$$

and

$$f'(net_{y_j}) = 1 \tag{A.8}$$

A.1 Learning rules for a PUNN using a bias unit

This section derives the learning equations for a PUNN where it is assumed that bias units occur in both the input and hidden layers, that respectively serve as bias to hidden units and bias to output units.

The net input of the hidden units of a PUNN that contains a bias unit in the hidden layer is given by,

$$net_{y_j} = \prod_{i=1}^{I} z_i^{v_{ji}} + v_{j,I+1} \cdot z_{I+1} \tag{A.9}$$

The $(I+1)^{th}$ unit represents the bias unit to each hidden unit; v_{ji} is the weight between the i^{th} input and j^{th} hidden units; z_i is the value of the i^{th} input unit.

Weights are updated according to the following equations:

$$w_{kj}(t) = \Delta w_{kj}(t) + \alpha \cdot w_{kj}(t-1) \tag{A.10}$$

$$v_{ji}(t) = \Delta v_{ji}(t) + \alpha \cdot v_{ji}(t-1) \tag{A.11}$$

where α is the momentum, w_{kj} is the weight between the j^{th} hidden unit, Y_j, and k^{th} output unit, O_k and v_{ji} is the weight between the i^{th} input unit, Z_i, and j^{th} hidden unit, Y_j.

In the remainder of this appendix the equations for calculating $\Delta w_{kj}(t)$ and $\Delta v_{ji}(t)$ are derived. For notational convenience, the reference to time, t is omitted.

The error with respect to weight v_{ji} is calculated, applying the chain rule of differentiation,

$$\begin{aligned}\frac{\partial E}{\partial v_{ji}} &= \frac{\partial E}{\partial net_{y_j}} \cdot \frac{\partial net_{y_j}}{\partial v_{ji}} \\ &= (\sum_{k=1}^{K} \frac{\partial E}{\partial net_{o_k}} \cdot \frac{\partial net_{o_k}}{\partial net_{y_j}}) \cdot \frac{\partial net_{y_j}}{\partial v_{ji}} \\ &= \sum_{k=1}^{K} \frac{\partial E}{\partial net_{o_k}} \cdot \frac{\partial net_{o_k}}{\partial y_j} \cdot \frac{\partial y_j}{\partial net_{y_j}} \cdot \frac{\partial net_{y_j}}{\partial v_{ji}} \end{aligned} \tag{A.12}$$

Now define,

$$\delta_{o_k} = -\frac{\partial E}{\partial net_{o_k}} \tag{A.13}$$

Substitution of (A.13), (A.5) and (A.7) in (A.12) yields,

$$\begin{aligned}\frac{\partial E}{\partial v_{ji}} &= \sum_{k=1}^{K} -\delta_{o_k} \cdot \frac{\partial(\sum_{j=1}^{J+1} w_{kj} y_j)}{\partial y_j} \cdot \frac{\partial(f(net_{y_j}))}{\partial net_{y_j}} \cdot \frac{\partial y_j}{\partial v_{ji}} \\ &= -\sum_{k=1}^{K} \delta_{o_k} \cdot w_{kj} \cdot f'(net_{y_j}) \cdot \frac{\partial y_j}{\partial v_{ji}} \\ &= -\sum_{k=1}^{K} \delta_{o_k} \cdot w_{kj} \cdot \frac{\partial y_j}{\partial v_{ji}} \end{aligned} \tag{A.14}$$

The output of hidden unit Y_j, is calculated next, where $v_{j,I+1}$ is the bias to Y_j and z_{I+1} refers to the bias unit with a constant value of -1.

Substitution of (A.9) in (A.7), results in,

$$\begin{aligned} y_j &= \prod_{i=1}^{I} z_i^{v_{ji}} + z_{I+1} \cdot v_{j,I+1} \\ &= e^{\ln(\prod_{i=1}^{I} z_i^{v_{ji}})} + z_{I+1} \cdot v_{j,I+1} \\ &= e^{\sum_{i=1}^{I} v_{ji} \ln z_i} + z_{I+1} \cdot v_{j,I+1} \end{aligned} \tag{A.15}$$

If $z_i < 0$, then z_i can be written as the complex number $z_i = i^2 |z_i|$ which, substituted in equation (A.15), yields

$$y_j = e^{\sum_{i=1}^{I} v_{ji} \ln|z_i|} \cdot e^{\sum_{i=1}^{I} v_{ji} \ln i^2} + z_{I+1} \cdot v_{j,I+1} \tag{A.16}$$

Let $c = 0 + \imath = a + b\imath$ be a complex number representing $\imath$. Then,

$$\ln c = \ln re^{\imath\theta} = \ln r + \imath\theta + 2\pi k\imath \tag{A.17}$$

where $r = \sqrt{a^2 + b^2} = 1$.

Considering only the main argument, $arg(c)$, $k = 0$, which implies that $2\pi k\imath = 0$. Also, is $\ln r = 0$, if $r = 1$. Furthermore $\theta = \frac{\pi}{2}$ for $\imath = (0, 1)$. Therefore, $\imath\theta = \imath\frac{\pi}{2}$, which simplifies equation (A.17) to $\ln c = \imath\frac{\pi}{2}$, and consequently,

$$\ln \imath^2 = \imath\pi \tag{A.18}$$

Substitution of (A.18) in (A.16) yields

$$\begin{aligned} y_j &= e^{\sum_{i=1}^{I} v_{ji} \ln|z_i|} \cdot e^{\sum_{i=1}^{I} v_{ji}\pi\imath} + z_{I+1} \cdot v_{j,I+1} \\ &= e^{\sum_{i=1}^{I} v_{ji} \ln|z_i|}(\cos(\pi \sum_{i=1}^{I} v_{ji}) + \imath \cdot \sin(\pi \sum_{i=1}^{I} v_{ji})) + z_{I+1} \cdot v_{j,I+1} \end{aligned} \tag{A.19}$$

Omitting the imaginary part, which is allowed since its inclusion did not result in any substantial improvement as reported by Durbin *et al* [Durbin *et al* 1989], reduces (A.19) to

$$y_j = e^{\sum_{i=1}^{I} v_{ji} \ln|z_i|} \cdot \cos(\pi \sum_{i=1}^{I} v_{ji}) + z_{I+1} \cdot v_{j,I+1} \tag{A.20}$$

Let

$$\rho = \sum_{i=1}^{I} v_{ji} \ln|z_i| \tag{A.21}$$

and

$$\phi = \sum_{i=1}^{I} v_{ji} \mathcal{I}_i \tag{A.22}$$

where

$$\mathcal{I}_i = \begin{cases} 0 & \text{if } z_i \geq 0 \\ 1 & \text{if } z_i < 0 \end{cases} \tag{A.23}$$

Then equation (A.20) becomes,

$$y_j = e^{\rho} \cdot \cos(\pi\phi) + z_{I+1} \cdot v_{j,I+1} \tag{A.24}$$

Now, applying differentiation w.r.t v_{ji} in equation (A.24),

$$\frac{\partial y_j}{\partial v_{ji}} = e^{\rho} \frac{\partial \rho}{\partial v_{ji}} \cdot \cos(\pi\phi) + \frac{\partial \cos(\pi\phi)}{\partial v_{ji}} \cdot e^{\rho} + \frac{\partial z_{I+1} \cdot v_{j,I+1}}{\partial v_{ji}} \tag{A.25}$$

$$\frac{\partial y_j}{\partial v_{ji}} = \begin{cases} e^{\rho} \cdot \ln|z_i| \cdot \cos(\pi\phi) - \pi \mathcal{I}_i \cdot \sin(\pi\phi) \cdot e^{\rho} & \text{if } i < I+1 \\ z_{I+1} & \text{if } i = I+1 \end{cases} \tag{A.26}$$

Substitution of (A.26) in (A.14), results in,

$$\frac{\partial E}{\partial v_{ji}} = \begin{cases} -\sum_{k=1}^{K} \delta_{o_k} \cdot w_{kj} \cdot e^{\rho}(\ln|z_i| \cdot \cos(\pi\phi) - \pi \mathcal{I}_i \cdot \sin(\pi\phi)) & \text{if } i < I+1 \\ -\sum_{k=1}^{K} \delta_{o_k} \cdot w_{kj} \cdot z_{I+1} & \text{if } i = I+1 \end{cases} \tag{A.27}$$

The changes to input-to-hidden weights are calculated as,

$$\Delta v_{ji} = -\eta \cdot \frac{\partial E}{\partial v_{ji}} \tag{A.28}$$

Substitution of (A.27) in (A.28) yields,

$$\Delta v_{ji} = \begin{cases} \eta \cdot \sum_{k=1}^{K} \delta_{o_k} \cdot w_{kj} \cdot e^{\rho} \cdot (\ln|z_i| \cdot \cos(\pi\phi) - \pi \mathcal{I}_i \cdot \sin(\pi\phi)) & \text{if } i < I+1 \\ \eta \cdot \sum_{k=1}^{K} \delta_{o_k} \cdot w_{kj} \cdot z_{I+1} & \text{if } i = I+1 \end{cases} \tag{A.29}$$

The error at the hidden layer, δ_{y_j} is now defined as ,

$$\delta_{y_j} = -\frac{\partial E}{\partial net_{y_j}} \tag{A.30}$$

$$= -\frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial net_{y_j}} \tag{A.31}$$

$$= -\frac{\partial E}{\partial y_j} \cdot f'(net_{y_j})$$

$$= -\frac{\partial E}{\partial y_j} \tag{A.32}$$

Next, $\frac{\partial E}{\partial y_j}$, is calculated applying the chain rule for differentiation,

$$\begin{aligned}
\frac{\partial E}{\partial y_j} &= \sum_{k=1}^{K}\left(\frac{\partial E}{\partial o_k}\cdot\frac{\partial o_k}{\partial y_j}\right) \\
&= \sum_{k=1}^{K}\left(\frac{\partial E}{\partial o_k}\cdot\frac{\partial o_k}{\partial net_{o_k}}\cdot\frac{\partial net_{o_k}}{\partial y_j}\right) \qquad \text{(A.33)} \\
&= \sum_{k=1}^{K}\left(\frac{\partial(\frac{1}{2}\sum_{k=1}^{K}(t_k-o_k)^2)}{\partial o_k}\cdot\frac{\partial o_k}{\partial net_{o_k}}\cdot\frac{\partial(\sum_{j=1}^{J+1}w_{kj}y_j)}{\partial y_j}\right) \\
&= \sum_{k=1}^{K}(-(t_k-o_k)\cdot f'(net_{o_k})\cdot w_{kj}) \\
&= -\sum_{k=1}^{K}(t_k-o_k)\cdot w_{kj} \qquad \text{(A.34)}
\end{aligned}$$

where (A.2) and (A.5) have been substituted in equation (A.33).

Substitution of (A.34) in (A.32) results in,

$$\delta_{y_j} = \sum_{k=1}^{K}(t_k-o_k)\cdot w_{kj} \qquad \text{(A.35)}$$

The equation above reduces (A.35) to,

$$\delta_{y_j} = \sum_{k=1}^{K}\delta_{o_k}\cdot w_{kj} \qquad \text{(A.36)}$$

Substitution of (A.36) in (A.29) results in,

$$\Delta v_{ji} = \eta\cdot\delta_{y_j}\cdot D_{ji} \qquad \text{(A.37)}$$

where D_{ji} is defined as,

$$D_{ji} = \begin{cases} e^{\rho}\cdot(\ln|z_i|\cdot\cos(\pi\phi)-\pi\mathcal{I}_i\cdot\sin(\pi\phi)) & \text{if } i < I+1 \\ z_{I+1} & \text{if } i = I+1 \end{cases} \qquad \text{(A.38)}$$

The error with respect to weight w_{kj} is calculated in the same way as for summation multilayer networks using gradient descent, i.e.

$$\Delta w_{kj} = -\eta\cdot\frac{\partial E}{\partial w_{kj}}$$

$$
\begin{aligned}
&= -\eta \cdot \frac{\partial E}{\partial o_k} \cdot \frac{\partial o_k}{\partial w_{kj}} \\
&= -\eta \cdot \frac{\partial}{\partial o_k}(\frac{1}{2}\sum_{k=1}^{K}(t_k - o_k)^2) \cdot \frac{\partial o_k}{\partial w_{kj}} \\
&= -\eta \cdot (-(t_k - o_k) \cdot \frac{\partial}{\partial w_{kj}}(\sum_{j=1}^{J+1} w_{kj} y_j)) \\
&= \eta \cdot (t_k - o_k) \cdot y_j
\end{aligned} \tag{A.39}
$$

Define the error that needs to be back-propagated as $\delta_{o_k} = -\frac{\partial E}{\partial net_{o_k}}$.

Then,

$$
\begin{aligned}
\delta_{o_k} &= -\frac{\partial_E}{\partial net_{o_k}} \\
&= -\frac{\partial E}{\partial o_k} \cdot \frac{\partial o_k}{net_{o_k}} \\
&= -\frac{\partial}{\partial o_k}(\frac{1}{2}\sum_{k=1}^{K}(t_k - o_k)^2) \cdot \frac{\partial o_k}{\partial net_{o_k}} \\
&= -(-(t_k - o_k) \cdot f'(net_{o_k})) \\
&= (t_k - o_k)
\end{aligned} \tag{A.40}
$$

since for linear activation, $f'(net_{o_k}) = 1$. Substitution of (A.40) in (A.39) yields,

$$\Delta w_{kj} = \eta \cdot \delta_{o_k} \cdot y_j \tag{A.41}$$

A.2 Learning rules for PUNN using a distortion unit

In this section the learning equations for a PUNN using a distortion unit are derived. In the case where the bias unit is replaced by a distortion unit in the hidden layer, only the equations influencing (A.29) need to be modified. Thus, equation (A.9) becomes,

$$net_{y_j} = \prod_{i=1}^{I+1} v_{ji} z_i \tag{A.42}$$

The $(I+1)^{th}$ input now represents the distortion to each hidden unit (refer to section 3.7.3 on page 72 for a discussion on the distortion unit). The input to the distortion unit is -1, i.e. $z_{I+1} = -1$.

Equation (A.15) now becomes,

$$y_j = e^{\sum_{i=1}^{I+1} v_{ji} \ln |z_i|} \tag{A.43}$$

To include the distortion unit in the product, equations (A.20), (A.21) and (A.22) become,

$$y_j = e^{\sum_{i=1}^{I+1} v_{ji} \ln |z_i|} \cdot \cos(\pi \sum_{i=1}^{I+1} v_{ji}) \tag{A.44}$$

which can be written as,

$$y_j = e^{\rho} \cdot \cos(\pi\phi) \tag{A.45}$$

where

$$\rho = \sum_{i=1}^{I+1} v_{ji} \ln |z_i| \tag{A.46}$$

$$\phi = \sum_{i=1}^{I+1} v_{ji} \mathcal{I}_i \tag{A.47}$$

where

$$\mathcal{I}_i = \begin{cases} 0 & \text{if } z_i \geq 0 \\ 1 & \text{if } z_i < 0 \end{cases} \tag{A.48}$$

Equations (A.29) and (A.38) become,

$$\Delta v_{ji} = \eta \cdot \sum_{k=1}^{K} \delta_{o_k} \cdot w_{kj} \cdot e^{\rho} \cdot (\ln |z_i| \cdot \cos(\pi\phi) - \pi \mathcal{I}_i \cdot \sin(\pi\phi)) \tag{A.49}$$

$$D_{ji} = e^{\rho} \cdot (\ln |z_i| \cdot \cos(\pi\phi) - \pi \mathcal{I}_i \cdot \sin(\pi\phi)) \tag{A.50}$$

where $i \leq I+1$ in equations (A.49) and (A.50).

The weight adjustment for weights between the hidden and output layer remains the same as the adjustment for PUNNs with a bias unit, i.e.

$$\Delta w_{kj} = \eta \cdot \delta_{o_k} \cdot y_j \tag{A.51}$$

This concludes the discussion on learning rules for PUNNs that contain either a bias or a distortion unit.

Appendix B

Publications from this thesis

A Ismail, AP Engelbrecht, *Training Product Units in Feedforward Neural Networks using Particle Swarm Optimization,* In: Development and Practice of Artificial Intelligence Techniques, V Bajic, D Sha (eds), Proceedings of the International Conference on Artificial Intelligence, Durban, South Africa, pp 36-40, 1999.

AP Engelbrecht, A Ismail, *Training Product Unit Neural Networks,* Stability and Control: Theory and Applications, Vol 2, No 1/2, pp 59-74, 1999.

A Ismail, AP Engelbrecht, *Global Optimization Algorithms for Training Product Unit Neural Networks,* IEEE International Joint Conference on Neural Networks, 24-27 July 2000, Como Italy, paper 032.

A Ismail, AP Engelbrecht, *Pruning Product Unit Neural Networks*, submitted to IEEE World Congress on Computational Intelligence, 2002.

A Ismail, AP Engelbrecht, *Improved Product Neural Networks,* submitted to IEEE World Congress on Computational Intelligence, 2002.

Appendix C

Symbols and notation

Symbols	Meaning
ANN	artificial neural network
BP	back-propagation by gradient descent
FLN	functional link network
FNN	feed-forward neural network
GA	genetic algorithm
LFOP	leapfrog optimization algorithm
NN	neural network
PSN	pi-sigma network
PSO	particle swarm optimization
PSO:PUs	product unit using product units
PSO:SUs	product unit using summation units
PU	product unit
PUNN	product unit neural network
RNN	recurrent neural network
SU	summation unit
SUNN	summation unit neural network

Symbols	Meaning
$\vec{v}_p$	the currents velocity of particle p
$\vec{x}_p = (x_{p,1}, x_{p,2}, ..., x_{p,D})$	the current position of particle p
$BEST_p$	the current best fitness achieved by particle p
$\overrightarrow{BEST x}_p$	the position that produced the best fitness value of the p^{th} particle
$GBEST$	the index of the best particle among all the particles in the population
δ_{y_j}	the error at the hidden layer
δ_{o_k}	the error at the output layer
z_i	i^{th} input value
Z_i	i^{th} input unit
y_j	activation of j^{th} hidden unit
Y_j	j^{th} hidden unit
o_k	activation of k^{th} output unit
O_k	k^{th} output unit
v_{ji}	weight between i^{th} input unit and j^{th} hidden unit
w_{kj}	weight between j^{th} hidden unit and k^{th} output unit
$f(net_{o_k})$	the activation for the k^{th} output unit
$f(net_{y_j})$	the activation for the j^{th} hidden unit
net_{o_k}	the net input for the k^{th} output unit
net_{y_j}	the net input for the j^{th} hidden unit
$\pm[2.0, 5.0]$	interval $[-5.0, -2.0]$ and interval $[2.0, 5.0]$