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.

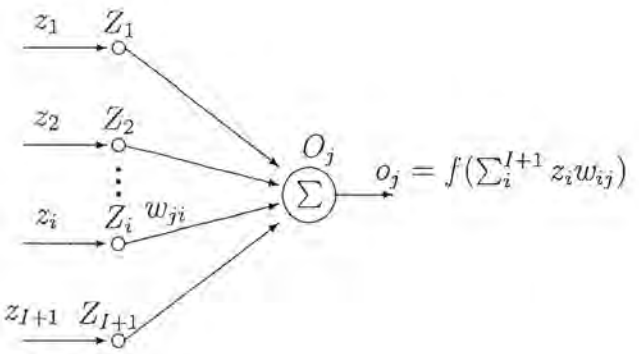


Figure 2.1: Model neuron using a Summation Unit

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

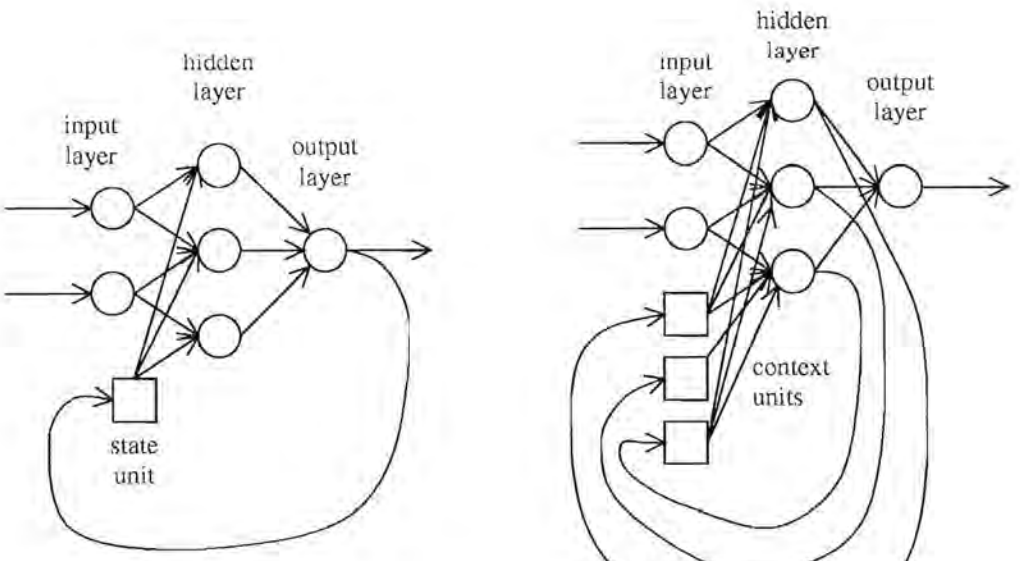
layer provides results of the network to the environment. Each neuron in the network provides an output which is a weighted sum, in the case of summation unit networks, or a weighted product of terms in the case of product unit neural networks. A feed-forward neural network has no memory and the output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

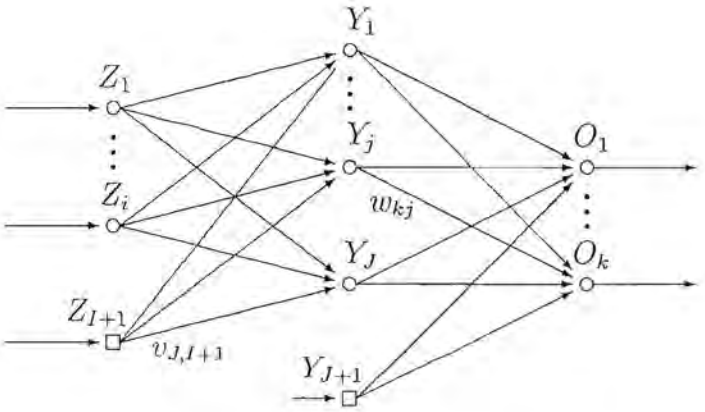


Figure 2.3: Multilayer feed-forward Neural Network

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.