APPLICATION OF BACKPROPAGATION-LIKE GENERATIVE ALGORITHMS TO VARIOUS PROBLEMS

by Alan Roy Powell

Submitted in partial fulfilment of the requirements for
the degree of Master of Science, in the Department of
Computer Science, University of Natal, Durban, 1992

PREFACE

The work described in this thesis was carried out in the Department of Computer Science, University of Natal, Durban, from January 1991 to June 1992, under the supervision of Professor A.G. Sartori-Angus.

These studies represent original work by the author and have not been submitted in any form to another University. Where use was made of the work of others it has been duly noted in the text.

ACKNOWLEDGEMENTS

I would like to acknowledge the help or support (or both) provided by the following people:

Most importantly my parents: for putting up with endless demands and helping (financially and otherwise) whenever called upon (yet again!).

My supervisor, Prof. A.G. Sartori-Angus, for allowing me to pursue this avenue of research.

The Computer Science department staff for help and entertainment (and money) throughout my postgraduate career.

The FRD for providing much appreciated financial support.

All the proofreaders of this work — all remaining mistakes are mine alone.

ABSTRACT

Artificial neural networks (ANNs) were originally inspired by networks of biological neurons and the interactions present in networks of these neurons. The recent revival of interest in ANNs has again focused attention on the apparent ability of ANNs to solve difficult problems, such as machine vision, in novel ways.

There are many types of ANNs which differ in architecture and learning algorithms, and the list grows annually. This study was restricted to feed-forward architectures and Backpropagation-like (BP-like) learning algorithms. However, it is well known that the learning problem for such networks is NP-complete. Thus generative and incremental learning algorithms, which have various advantages and to which the NP-completeness analysis used for BP-like networks may not apply, were also studied.

Various algorithms were investigated and the performance compared. Finally, the better algorithms were applied to a number of problems including music composition, image binarization and navigation and goal satisfaction in an artificial environment. These tasks were chosen to investigate different aspects of ANN behaviour. The results, where appropriate, were compared to those resulting from non-ANN methods, and varied from poor to very encouraging.

Contents

CHAPTER ONE

1.1. INTRODUCTION

Artificial neural networks (ANNs), or connectionist models, have a fairly long and chequered history; more so than the recent revival of interest indicates. A brief history of ANNs and the workings of biological neural networks will be given. The general terms and concepts used will be defined as well as a discussion of some of the reasons that ANNs have become popular once again. Finally a few examples of applications using ANNs to tackle "real-world" problems successfully will be given.

There are two approaches for studying and modelling the brain: *bottom-up* and *top-down* [KSe89] (Introduction). The bottom-up approach studies the constituent parts of the brain, how these parts interact and tries to understand how these relationships achieve the observed processes. Thus it is a *data-driven* approach that proceeds from a detailed study of the biological hardware, and is usually based directly on experimental data. In ANN research, this approach is inherently parallel since natural neural nets operate in parallel [BP90]. Feldman *et al.* [FB82] have stressed an important point, usually known as the *100 steps constraint*: the brain is not fast enough to compute the functions required by the big AI models. A neuron performs simple calculations in a few milliseconds, while a complex task, such as face recognition, requires a few hundred milliseconds [Llo89]. Thus, assuming a serial process, only about one hundred discrete program steps can be executed before the goal is achieved. Therefore AI programs that require thousands of steps seem implausible for this reason alone.

The top-down approach is a *theory-driven* one. The question of how the information or how a particular computation could be carried out is first addressed. Once the problem has been characterized at a formal level, a particular algorithm is derived to solve the problem. Often the algorithm is designed to be in agreement with known anatomy and physiology and can thus be biologically plausible, although this is not a prerequisite. The bottom-up approach is generally used by neuroscientists, while the Artificial Intelligence community has concentrated on the top-down approach [KSe89].

An example might help to clarify the difference between the two methods. Consider the problem of separating an image into objects and background. The bottom-up approach would try to model the retinal cells, their interactions and the other cells and their interactions in the visual pathway. The top-down approach might use methods such as *grey level thresholding* or *multilevel thresholding* [Amm86] applied to the image, with less consideration of how these processes could be achieved biologically. A synthesis between the two approaches has naturally been taking place since both have limitations that are difficult to overcome. For example, understanding how the visual system computes binocular disparity requires a computational analysis of the inverse problem of recovering depth from two spatially displaced 3D images, which is a top-down approach, as well as a knowledge of the physiology of disparity-sensitive cells in the cortex, which is a bottom-up approach [KSe89].

1.2. A BRIEF HISTORY LESSON

What follows is a brief introduction to the history of ANN research. It is in no way complete and is meant as background, rather than a serious attempt to trace the roots of ANN research. Some of the earliest roots of ANNs can be found in work done by the neurologists Jackson (1869) and Luria [RM86]. Jackson, for example, was critical of the localized doctrines of the 19th century and argued for distributed, multilevel processing systems.

James (1890) was one of the first to publish a variety of facts related to the structure and function of the brain [ED90] (Introduction). For example, he stated some basic principles of correlation learning (his *Elementary Principle* amongst others) which are closely related to the concepts of correlation learning and associative memory. He also put forward the notion that a neuron's activity was a function of the sum of its inputs with past history contributing to the weights of the interconnections [Jam89].

McCulloch and Pitts [MP43] published one of the most famous papers in the neural network (NN) field. They derived models of neurons based on the physiological knowledge of the time and concluded with five points:

- The activity of the neuron is an *all-or-nothing* process.
- A certain fixed number of synapses must be excited within the period of latent addition in order to excite a neuron at any time, and this number is independent of the previous activity and position of the neuron.
- The only significant delay within the nervous system is synaptic delay.
- The activity of any inhibitory synapse absolutely prevents excitation of the neuron at that time.
- The structure of the net does not change with time.

The neuron described by these five conclusions came to be known as the *McCulloch-Pitts* neuron. Most of these assumptions are still used in the ANN models, but it has become increasingly clear that this type of model is too simplified. Biological neurons are in fact more complex than these assumptions state and act more like voltage-to-frequency converters than simple on-off logical devices [AR89] (introduction to [MP43] paper), [BKR78]. The importance of this paper was that using a massively parallel architecture they proved that a network of these neurons could represent any finite logical expression.

This was followed by Hebb's book [Heb49] which introduced the first method to update synaptic weights. Roughly, his rule states:

> "When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased." (p50)

His contributions were primarily [ED90]:

- Information is stored in the weights[1] of the synapses.
- The weights are symmetric.
- The putting forward of a connection weight learning rule where the weights are changed in proportion to the product of the activation values of the neurons.
- Postulating that if simultaneous activation of a group of weakly connected cells occurs repeatedly, these cells tend to coalesce into a more strongly connected assembly.

In 1958, Rosenblatt [Ros58] defined an ANN termed the *Perceptron.* It was the first precisely specified, computationally orientated neural network, and it made a major impact on a number of areas simultaneously [AR89] (p89). He also defined the *Perceptron Convergence Procedure*, a synaptic adaption rule, which was an improvement of that presented by Hebb. The Perceptron is discussed in the following chapter.

Widrow and Hoff [WH60] published *Adaptive Switching Circuits* in 1960, that, especially from an engineering viewpoint, was extremely important. They designed a neural network that they simulated on a computer and then implemented in hardware. Although similar to the Perceptron, the learning algorithm yields faster and more accurate learning [ED90]. Their neurons used a linear threshold activation function and were known as Adalines.

In the 1960s and 1970s, research into ANNs reduced greatly. One reason was the availability of serial computers which led AI researchers to consider serial-like algorithms and symbolic processing to be the most promising area of investigation. This shift away from ANNs was also partly due to the publication of *Perceptrons: An Introduction to Computational Geometry* by Minsky and Papert [MP88].This text demonstrated, in the course of trying to develop a theory of parallel computation, that the Perceptron is inherently limited due to its architecture.

1 The "weight" of a synapse is the efficiency of the synapse to propagate signals.

Minsky and Papert concluded their work with an intuitive judgment that the extension to multilayer perceptrons would be sterile. For the following decade the research and the associated models moved away from the strictly boolean framework to nets based on analogue and non-linear interactions [Lev89]. Kohonen [Koh72], for example, developed an associative memory neural network where the neurons were linear and continuous valued, rather than the binary McCulloch-Pitts neuron. Grossberg introduced numerous concepts that are still used in networks. Some of his work in this period include the *on-center off-surround* gain control system for a group of neurons and the sigmoid function, a new type of neuron activation function [ED90] (Introduction).

In 1982 Hopfield made two major contributions to the field [Hop82]: he described analogue neural network theories to mainstream scientists and suggested ways to build such networks in electronic circuitry. A learning rule that improved on the Perceptron learning rule was introduced in 1986 by Rumelhart *et al.* [RHW86]. Termed *Backpropagation* it allowed networks consisting of more than an input and output layer of neurons to be trained, and thus overcame the most serious limitation of the Perceptron. They showed that multilayer nets could accomplish some of the tasks that the Perceptron was incapable of. Although Rumelhart's discovery was preceded by Parker (1982) and Werbos (1974), they are generally credited with the discovery because they popularized ANNs and their two books [RM86] have come to be considered the "bible of ANN research" [Lev89]. Many practical systems using this rule were developed, the best known is arguably *NETTalk* developed by Sejnowski and Rosenberg [SR86] which learnt to pronounce English text. Other researchers, such as Grossberg and Carpenter [Cau89] and Fukushima *et al.* [FMI83], developed more complex and biologically more plausible ANNs with varying success.

Since 1987, the ANN field has expanded tremendously, and it is not feasible to summerize "all there is to know" about the current state of research [ED90].

1.3. WETWARE

Wetware is the term used to describe the soft tissue components of the brain; its biological and chemical circuitry. Guyton [Guy81] discusses the functioning of neurons, synapses and general brain function in detail. The following sections summarize and expand his discussion.

1.3.1. Structure

The basic building block in biological neural networks (BNNs) is the nerve cell or *neuron*. A neuron consists of:

- The *soma* which is the main cell body.

- A single *axon* which extends from the soma.
- *Dendrites* which are thin projections of the soma and axon.

The axon and dendrites usually have many branches, each which ends in a *synaptic knob.* The synaptic knob is close to, but not touching, the dendrites and soma of other neurons. The gap between the synaptic knob and the abutting neuron is known as the *synaptic cleft,* while the junction point from one neuron to the next is known as the *synapse* (fig 1-1).

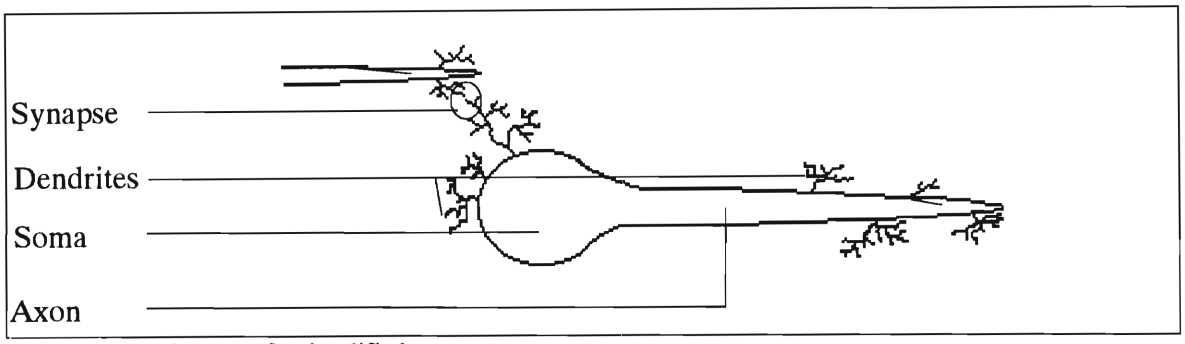


Figure 1-1 A diagram of a simplified neuron.

BNNs generally contain many different types of neurons. Neurons can differ in a variety of ways [Guy81]:

- The size of the soma.
- The length, size and number of dendrites. The length can be as long as 1m (the peripheral sensory nerve fibre).
- The length and size of the axon.
- The number of synaptic knobs, from a few to more than 100 000 (certain cells in the cerebral cortex for example) with an average of 1 000 to 10 000 per axon.

The different types of neurons react differently to incoming signals and therefore perform different functions. To give an indication of the complexity of the human neural network, the human brain contains about 10^{11} neurons in the cortex. Since each neuron has between 10^3 and 10^4 synapses, we have a total of approximately 10^{14} to 10^{15} synapses in the cortex. Figure 1-2 illustrates the complexity of BNNs compared to what can be achieved technologically [DAR87].

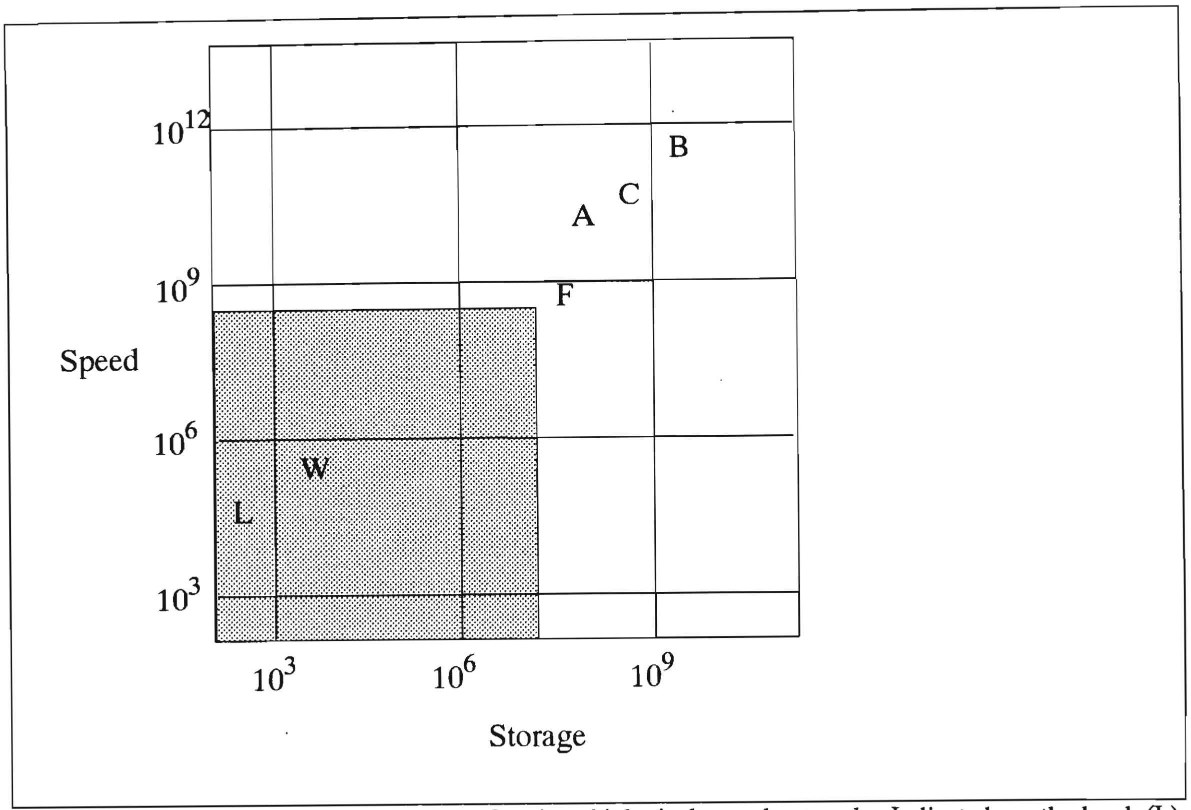


Figure 1-2 Comparison of the complexity of various biological neural networks. Indicated are the leech (L), worm (W), fly (F), Aplysia (A), cockroach (C) and bee (B). Humans do not appear in the diagram but would be found at coordinates (10^{14}, 10^{16}). The shaded area indicates the current limits of ANN technology. *Storage* is measured in the number of interconnections while *Speed* is measured in the interconnections processed per second.

1.3.2. Impulse propagation

The BNNs accomplish their tasks by propagating electrochemical impulses about the network.

1.3.2.1. Impulses in single cells

Electrical potentials exist across the membranes of essentially all cells in the body. Some cells, such as nerve and muscle cells, are capable of utilizing these impulses to transmit signals along the cells membranes.

Fluids on the interior and exterior of the cells are electrolytic solutions. Generally a very minute excess of negative ions accumulate immediately inside the membrane and an equal amount of positive ions accumulate on the outside of the membrane. This establishes a *membrane potential* between the inside and outside of the cell. There are two basic means by which membrane potentials can develop and be altered:

- Active transportation of ions through the membrane. The method in which this is accomplished is beyond the scope of this section.
- Diffusion of ions through the membrane as a result of ion concentration differences.

Depolarization (or *action potential*) occurs when the polarity at a spot in the membrane is reversed: the positive ions move to the inside of the membrane and similarly the negative ions move out of the cell. An action potential that occurs at one spot on the membrane usually excites adjacent portions of the membrane increasing the rate of active transportation, resulting in propagation of the action potential across the entire cell. This transmission of the depolarization impulse along a nerve fibre is known as a *nerve impulse*. There are a few important points to note about such impulses:

- There is no single direction of propagation of the impulse.
- All-or-nothing principle: the depolarization impulse travels over the entire membrane. This holds for all normal excitable tissue, such as muscle, except in certain very specific cases.
- The action potential normally only lasts almost the same length of time at each point along the fibre. Repolarization normally occurs first at the point of the original stimulus and propagates in a similar manner to the original depolarization. It follows the depolarization process a few 10 000th of a second later.
- The impulse reduces the concentration of the ion concentrations inside and outside the cell which can affect future impulses due to the ion concentration becoming depleted.

1.3.2.2. Myelinated and unmyelinated nerve fibres

The axon of a nerve fibre can be either myelinated or unmyelinated. The myelin sheath, which surrounds certain nerve fibres and is deposited by the *Schwann* cell, acts as an insulator. The myelin sheath is approximately as thick as the axon, and is interrupted once every 1mm by a junction known as the *node of Ranvier*. Myelin is an excellent insulator and increases the resistance to ion flow through the membrane approximately 5 000-fold. At the junction, small uninsulated areas remain where ion flow can take place with ease. In fact, these areas are 500 times as permeable as the membranes of some unmyelinated fibres.

The action potential is thus conducted from node to node rather than continuously along the entire fibre as in unmyelinated fibres. This process is known as *saltory conduction* and is valuable for two reasons:

- The depolarization process jumps long intervals along the axon. It greatly increases the velocity of the nerve transmissions.
- Energy is conserved because only the nodes need to depolarize, with several 100 times less loss of ions than for unmyelinated fibres.

1.3.2.3. Velocity of conduction

The velocity of impulses varies from as little as 0.5m/s in small unmyelinated fibres to 130m/s in very large myelinated fibres. The velocity increases approximately with the fibre diameter in myelinated fibres and approximately with the square root of the fibre diameter in unmyelinated fibres.

1.3.3. Presynaptic terminal and synaptic knob

The synaptic knobs are separated from the abutting soma and dendrites by the synaptic cleft, which is usually 200 to 300 Angstroms wide. The synaptic knob has two important internal structures:

- Synaptic vessicles contain the neurotransmitters which, when released, either excite or inhibit the abutting neuron.
- Mitochondria supply energy to synthesize the new transmitter substance (neurotransmitters).

Different neurotransmitters have different durations of stimulation. Another difference between the different transmitters is that some cause an increase in the rate of firing of the abutting neurons while others change the abutting neurons sensitivity to other transmitters.

1.3.4. Synapse

Synapses determine the direction that signals spread in nervous systems and can perform various selective actions:

- Blocking weak signals while allowing strong signals to pass.
- Selecting and amplifying certain weak signals.
- Channeling the signal in many different directions out of the cell rather than in one direction only.

1.3.5. Impulse propagation across the synapse

When an action potential spreads over the presynaptic terminal, the membrane depolarization causes the emptying of a small number of vesicles into the cleft and the released transmitters immediately change the permeability characteristics of the subsynaptic neuronal membrane to all ions. This leads to excitation or inhibition, depending on the type of transmitter substance released. A single neuron releases only one type of transmitter and releases it at all its synaptic knobs. Neurons can and do respond to different transmitters received by its dendrites and soma.

Some post-synaptic neurons respond to the release of the transmitters with a large number of impulses propagated along its soma while others respond with only a few. The stimulation of the cell must exceed a threshold before an action potential is generated and a nerve impulse results (the *firing* of the neuron). Usually ten or more knobs have to release their transmitters simultaneously, or in rapid succession, before the post-synaptic neuron will fire. This is known as *summation*. Neurons accommodate slowly increasing stimulae: as the action potential slowly increases, the neuron accommodates the increase by raising the threshold, rather than firing. The firing frequencies of neurons range from a few to a few hundred impulses per second [Fel82].

Some special characteristics of synaptic transmission deserve mention:

- Impulses are conducted through the synapses in only one direction.
- When excitatory synapses cause repeated stimulation at a rapid rate, the number of discharges by the neuron is at first very great, but becomes progressively less in succeeding milliseconds or seconds. This feature is called *fatigue* and, together with inhibitory circuits, is an important safety mechanism. For example, epileptic fits occur when a collection of neurons fire uncontrollable. The fits eventually subside partly because the neurons involved undergo fatigue.

1.3.6. Summation

There are two methods of summation which can result in a neuron firing:

- Spatial summation where a collection of knobs release their transmitters simultaneously.
- Temporal summation. Since most knobs can release transmitters repetitively in rapid succession only a few milliseconds apart and post-synaptic potential lasts up to 15ms, the rapid repeated release of transmitters by some knob can further increase the post-synaptic potential.

1.3.7. Learning

The synapses are the ideal site to control the signal transmissions. At a cellular level, it seems reasonable to try to adjust a neuron's firing by adjusting its threshold, given the appropriate inputs from abutting neurons. A number of suitable rules have been proposed with one of the first and best known due to Hebb [Heb49] [RM86]. Hebb proposed that if one neuron persistently causes another to fire, the connection between these neurons should be strengthened (as mentioned briefly in section 1.2). The connection can be strengthened by increasing the efficiency of the appropriate synapse transmission or by changing the threshold of the neurons. There is evidence that supports this rule [Guy81] (chapter 46).

Many learning rules proposed by animal learning theorists have, broadly speaking, been error correction rules: a change in the variable tracking the learning has been proportional to its difference from some target value or asymptote [McL89]. This type of rule is also favoured by human learning theorists [RM86].

The trend in neurobiological modelling of learning seems to be more in favour of a Hebbian-type rule. Neurobiologists don't deny the importance of error-correction — they think appropriate nets can be constructed using simpler Hebbian-type neurons. McLaren [McL89] showed that Hebbian-type neurons are unsuitable for this, but described a neuron that overcomes some of the difficulties, and which can be used to implement error-correction rules. His model was shown to be biologically plausible and so it would seem that error correction methods, as opposed to Hebbian learning, are a valid method for learning in NNs.

1.3.8. Distributed versus localized representation

Different functions are generally accomplished in different areas of the brain [Guy81]. This includes vertebrates as well as invertebrates such as the octopods [Cou79] (p165). This means that damage to the brain results in *graceful degradation* where the brain functions degrade but do not immediately cease.

The question of how memory is stored in the brain deserves investigation.There are essentially two ways of representing the data: the individual neurons could store the data or the interconnections between the neurons could accomplish this. The former method is sometimes known as the *grandmother cell* approach and originated with the hypothesis that there is an individual cell which, via connections to other cells in the visual pathway, is activated whenever an image of the subject's grandmother is detected.

The most important problem with this hypothesis is that if such a cell were irreparably damaged, it would be impossible for the subject's grandmother to be recognized. It should also be remembered that neurons are continually being destroyed and are generally not replaced [BL90].

The second method is generally used in ANNs. If the interconnections between neurons are the information store, the loss of a few neurons is less important, especially if a large number of interconnections are used to store each datum. Thus even after large portions of the net have been damaged, the creature is often still functional, albeit in a reduced manner [Guy81]. Also, if different collections of the interconnections represent different data, the same interconnection can represent parts of different data. This is further explained in [And90].

It is difficult to deduce a neural network's function solely from the determination of individual neurons receptive fields. Lekhy and Sejnowski [LS88], for example, used an ANN to model shape-from-shading: a model of how the visual system extracts information about a 3D object's shape from the shading of an image of the object. The receptive fields, which are the set of neurons from which other neurons receive inputs [HU91], developed by the artificial neurons were similar to the actual receptive fields of neurons observed in the visual cortex. The model reacted in similar ways to the natural neural network when bars of light where used as input to the network. The corresponding natural neural network was previously thought to achieve *bar* or *edge* detection given visual input and had never been associated with the detection of shape-from-shading information. This demonstrates the difficulty of deducing the function of a neural network by considering only the immediate receptive fields of the neurons and ignoring the effect of more distant or indirectly connected neurons.

1.4. WHAT ARE ANNs?

1.4.1. Definitions

Although there are many different definitions for an ANN, informally it can be considered to be a collection of simple processors that are connected in some manner. Each connection has a weight, or cost, associated with it which is applied to signals that are propagated along the connection. A definition used by the DARPA Neural Network Study [DAR87] (p396) was that "a neural network is a system composed of many simple processors — fully, locally or sparsely connected — whose function is defined by the connection topology and connection strengths. This system is capable of high-level functions, such as adaption or learning with or without supervision, as well as lower-level functions, such as vision and speech processing". Formally "a neural network is a dynamical system with the topology of a directed graph that can carry out information processing by means of its state response to continuous or episodic input" [Hec87a]. A more complete definition, partly due to Judd [Jud87] is that a neural network can be considered a directed, not necessarily acyclic, graph, where the arcs of the graph have some cost associated with them and the vertices perform some function of their inputs from other vertices in the graph.

The vertices of the graph are commonly referred to as *neurons*, *neurodes*, *nodes* or *units*. The arcs are known as *connections* or *interconnections* and the cost associated with a connection is known as the *weight* of that connection.

1.4.2. Features all ANNs possess

There are at least eight features that all neural network models possess [RM86] (p45):

- A set of processing units which correspond to the vertices of the graph as defined above. Generally these units perform simple operations, typically addition of the weighted inputs, as opposed to the transputer route where each unit is a powerful microprocessor. Typically units receive inputs from their neighbours, compute an output value and send the value to its neighbours. This process is inherently parallel with no executive/controller organizing the order or timing of each units operation.
- A state of activation for each unit where all the units activations represent the state of the system at time *t*. The state is usually represented by a vector of *N* real numbers corresponding to the activation values of the *N* units of the net. Different models restrict the units to different activation values, such as binary, discrete or continuous values.
- An output function for each unit. The output function is applied to the current activation state of the unit to determine the output value of that unit.
- Some pattern of connectivity among the units.
- A propagation rule for propagating input vectors through the net to produce output vectors.
- An activation rule for combining the inputs impinging on a unit, and that unit's current state, to produce a new level of activation for that unit.
- A learning rule to modify the behaviour of the net by modifying the connectivity patterns through experience.
- A representation of the environment in which the system is to operate.

The unit's processing power is usually limited to summing the inputs impinging on the unit and applying a function to the sum. Examples of commonly used functions are:

A linear function:
$f(net_i) = net_i$

The Heaviside (step) function:
$f(net_i) = 0$ if $net_i < t$
$f(net_i) = 1$ otherwise for a threshold t

The linear-threshold function:
$f(net_i) = 0$ if $net_i < t$
$f(net_i) = net_i$ otherwise for a threshold t

The *sigmoid* function:

$$f(net_i) = \frac{1}{1+e^{-net_i}}$$

where $net_i = \sum_{j=0}^{n_i} w_{ij} a_j$ is the net input to unit i,

n_i is the number of units connected to unit i, w_{ij} is the weight of the connection from unit j to unit i and a_j is the activation value of unit j.

The five conclusions of McCulloch and Pitts defining the McCulloch-Pitts neuron are still followed. However, using the McCulloch-Pitts model leads to properties not observed in BNNs [Llo89]:

- Neurons that excite some neurons and inhibit others.
- Neurons that merely change sign. For example, a neuron that accepts excitation from one neuron only and whose output produces inhibition in one neuron only.
- Neurons that connect to all other neurons of the same type.
- A neuron that singularly can cause another to fire. Although this does occur in the cerebellum, it is not certain whether it occurs in the neocortex. Available evidence suggests that it is uncommon.

Further differences between real and artificial neural networks include [Geo77] (p90-91):

- Actual neurons differ in size and firing rate.
- Actual nervous systems are accompanied by chemical changes when they fire, which may directly affect the overall properties of the system.
- Neurons in human beings are thought to fire across the cell bodies as well as along the axon.
- There might be electrical field affects operating in the nervous system.

Rumelhart *et al.* [RM86] (p73-74) have used units more complex than the McCulloch-Pitts neuron, termed *sigma-pi* units. With these units, output values of *n* units are multiplied together before the product enters into the summation. Such *multiplicative* connections allow units to *gate* each other. For example, if unit A and B form such a connection and unit A has a zero activation value, unit B's activation value has no effect in the summation, since the product of the two units activation is 0.

As other researchers have indicated [AR89] (p16) the McCulloch-Pitts neuron is too simple to accurately model real neurons: the McCulloch-Pitts neuron is a binary device while real neurons act more like voltage-to-frequency converters. Brodie *et al.* [BKR78] have proposed a more realistic model of a neural system. Taylor [Tay90] introduced temporal and probabilistic features found in biological neurons to the McCulloch-Pitts neuron and the resulting networks were capable of learning temporal sequences.

It has been suggested that complex valued weights be allowed (U.B. Sarma and C.Chen — pers comm). Although some consider this to be equivalent to real valued weights with additional inputs and weights (S.E. Fahlman, C.Chen — pers comm), others (Sarma — pers comm) think the analysis of such networks might be easier. This is currently an active research area.

Because neurons and synapse are now known to be more complex than the McCulloch-Pitts model, the knowledge partially justifies the Hebb rule, but also significantly extends it [DAR87] (p148).

The question of how accurate the models need to be is difficult and as yet unanswered [Dur89]. Miall [Mia89] notes that the only correspondence between real neurons and the current models of neurons is that both sum the activity of their inputs, weighted by synaptic efficacies and produce an output that is a function, non-linear or linear, of the summation. Miall further contends that, because of varied temporal properties of real neurons that the models lack, the current ANNs are unsuited to many time dependent tasks.

There seems to be two approaches to this problem:

- More accurate modelling of the neurons, their interconnections with the associated complexity. This is usually the approach favoured by neuroscientists.
- The applications/engineering approach which considers the distributed approach as being important, rather than the finer details of the individual units. Armstrong's *Adaptive Logic Networks* (pers comm) follows this route, where each unit can compute one of four logic functions, AND, OR, LEFT or RIGHT (where RIGHT(A,B) = B) of its two boolean inputs and the net is a tree-like structure. Adaptive Logic Networks (ALNs) are superisingly powerful with quick training and response times. Further ALNs have been applied to various tasks, eg. the estimation of fat in beef from ultrasound images [MT91].

Because of the current lack of knowledge about the functioning of neurons and their interactions, our models will, at best, be a rough approximation.

1.4.3. Learning algorithms

The learning algorithms can be grouped into three classes [Hec87a]:

- Supervised learning. The network requires a *teacher* which presents the network with input vectors and the associated desired output vectors. Through a process of error correction, the network tries to minimize the error between the desired output vectors and the actual output vectors computed. Various error functions can be used such as the sum of the square of the differences between the desired and actual output vectors or the Euclidian distance between the desired and actual output vectors.

- Self-supervised or graded learning. The network is presented with the input patterns and attempts to reduce an internally generated error measure. A method often used to reduce the error measure is to make small changes to the weights of the network. If the changes bring about a reduction of the error, then the changes were beneficial and similar changes should be made. If the changes increased the error measure, then the changes were *bad* and this direction of change shouldn't be followed. Another option is to have a *supervisor* who merely responds to the network's attempts as either *right* or *wrong* without giving an indication of the magnitude of the error, or how to correct it. Using the supervisor's response, the network reduces its error by adapting the weights of the connections.
- Unsupervised or self-organizing learning. For this type of learning no teacher is present. The input patterns are presented to the network and similar inputs are grouped into the same or similar classes. Note that *similar* often means the input patterns are grouped together in a way that is statistically significant.

One presentation of all the patterns making up the training data set is known as an *epoch* or *cycle*. The patterns used to train a net differ depending on the type of learning algorithm. Supervised algorithms require both input and target output pattern pairs so as to be able to calculate the error between the actual and desired outputs. Self-supervised and unsupervised algorithms only require input patterns since no target output is explicitly required by the algorithm.

1.4.4. Types of architectures

There are two classes of interconnection schemes:

- Layered networks consist of a series of layers of units (fig 1-3a). All the neurons in one layer share some common feature such as the same activation function or connections from the same set of cells.
- In non-layered networks the units are all considered to be in one layer (fig 1-3b), even though they may have different properties.

Layered networks can be furthered divided into:

- Feed-forward networks in which any unit receives inputs only from units on *earlier* layers, or from units on the same level (fig 1-3c). In this way, the input only *flows* in one direction through the net, from the input units to the output units.
- Recurrent networks allow later units to connect to earlier units, and thus feed their output back to act as input to earlier units (fig 1-3d). Non-layered networks are usually recurrent, although, given any recurrent net, a feed-forward net with identical behaviour over a finite period of time exists [RM86] as demonstrated by fig 1-3e and 1-3f.

without hidden units

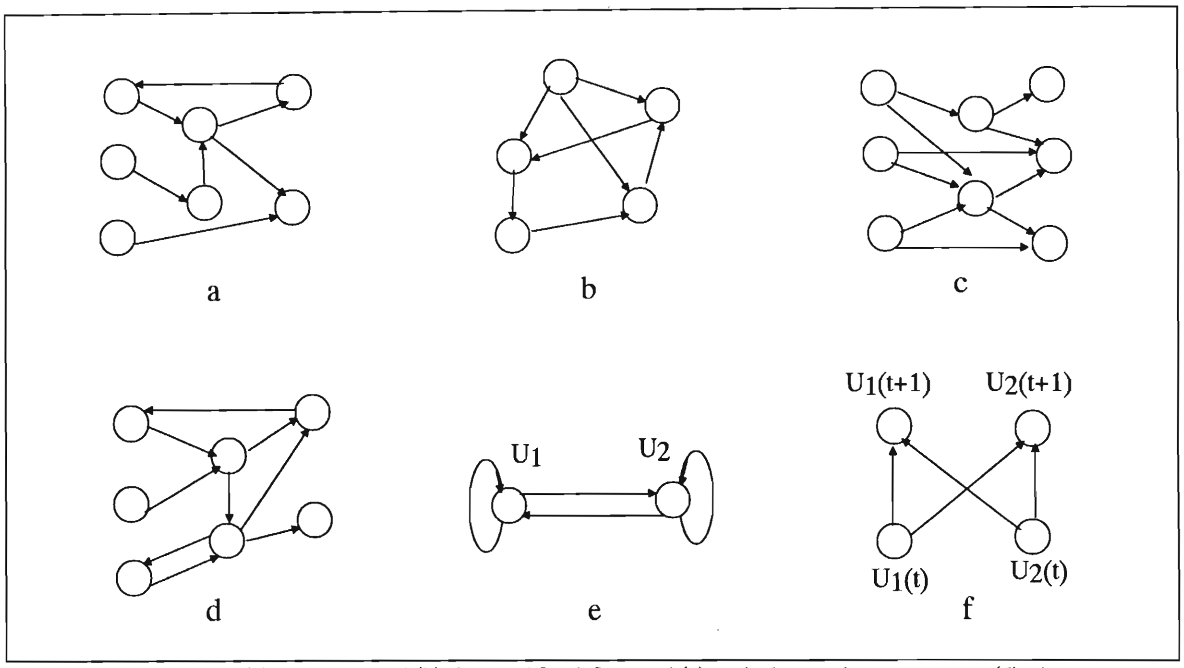


Figure 1-3 A layered (a), non-layered (b), layered feed-forward (c) and a layered recurrent net (d). Any recurrent net (e) can be replaced by a layered net (f) with identical behaviour to (e) at time *t*.

without hidden units.

1.4.5. The thirteen main ANN models

Hecht-Nielsen [Hec87a] [Hec88] lists the thirteen types of ANNs in common use:

NAME	YEAR	COMMENTS
Adaptive Resonance Theory (ART)	1978-1986	Unsupervised learning, autoassociative memory.
Avalanche	1967	Learn, recognise and playback of spatiotemporal patterns.
Backpropagation (BP)	1974-1985	The most popular supervised learning algorithm.
Bidirectional Associative Memory (BAM)	1985	Single stage heteroassociative nets.
Boltzmann Machine	1985-1986	Uses an "energy" function to find a global minimum of the cost function.
Brain State in a Box	1977	Single stage autoassociative net.
Cerebellatron	1969-1982	Learns the average of spatiotemporal patterns and replays these values on cue.

Counterpropagation	1986	Functions as a statistically optimal self-organizing look-up table.
Hopfield net	1982	Single stage autoassociative net with no learning.
Madaline	1960-1962	Trainable linear combiners.
Neocognitron	1978-1984	Multilayer hierarchical character recognition net.
Perceptron	1957	Trainable linear discriminants
Kohonen's Self-organizing Map	1980	Unsupervised learning. Forms a continuous topological mapping from one space to another.

The current learning laws can also be divided into six groups [Hec87a]:

- Grossberg: competitive learning of weighted average inputs (eg. ART).
- Hebb: correlation learning of mutually-coincident inputs.
- Kohonen: develops a set of vectors conforming to a particular probability density function.
- Kosko/Klopf: forms representations of sequences of events in temporal order.
- Rosenblatt: adjustment of perceptron linear discriminant device (eg. Perceptron).
- Widrow: minimization of mean squared error of a cost function (eg. Madaline).

ANNs are capable of a variety of operations. These include [Hec87a]:

- Mathematical mapping approximations where the ANN develops an approximation to a function $f: A \subset R^n \rightarrow B \subset R^m$ in response to a set of examples $\{(x_i, y_i)\}$, $x_i \in A$, $y_i \in B$.
- Probability density function estimation. A set of equiprobable *anchor points* are developed by self-organizing in response to a set of examples x_1, x_2, ... of vectors in R^n chosen in accordance with a fixed probability density function.
- Extraction of relational knowledge from binary data bases.
- Formation of topologically continuous and statistically conformal mappings.
- Nearest neighbour pattern classification.
- Categorization of data.

1.4.6. Features of ANNs

The connectionist approach is attractive because it exhibits many desirable properties including properties found in natural neural networks [BA91] [RM86]:

- Neural plausibility. Because ANNs were originally inspired by BNNs they seem more compatible than symbolic and other AI approaches with what is known about the nervous system. It must be remembered, however, that ANNs rarely include all the features found in BNNs, partly because of a lack of knowledge about the fine details of BNNs.
- Graceful degradation. Biological BNNs are extremely reliable. Although it is possible to exceed their limits, eg. by destroying too many neurons or connections or by supplying too many inputs, the BNN keeps on operating, albeit in a reduced, sub-optimal manner. This useful property is shared by most ANN models in contrast to traditional computational models.
- Capacity to learn from experience. A very attractive feature is that ANNs can learn their behaviour from examples. This contrasts with traditional computation models, such as expert systems, which are usually explicitly programmed or supplied with the relevant rules.
- Generalization. Many ANNs generalize from the training data to unseen data. Thus, the network's performance on unseen data is equivalent (or almost equivalent) to that on similar training data. This is a useful property since the net could possibly treat unseen and new situations correctly provided the training of the net was sufficient.
- Immunity to noise / fault tolerance. Noisy input data can be disastrous to traditional AI methods. ANNs are often less sensitive to *noisy* inputs and *soft constraints* (where no explicit rules are available or when many exceptions to the rules exist).

1.4.7. Criticisms of the ANN paradigm

There are at least two grounds on which ANNs can be criticized:

- Neurobiological. It is readily admitted that most of the models of neurons used in ANNs are greatly simplified. The McCulloch-Pitts neuron was a simplification of neurons and the processes involved as the process was understood in the 1940s. Although additional data has since arisen, many ANNs still use the McCulloch-Pitts neuron as a building block. Durbin [Dur89] raises the important point that, due to a lack of knowledge, it is not known how simplified the models can be made before important information is lost. It is important not to see a one-to-one correspondence between neurons and units: a group of units could possibly serve as a more accurate model of a neuron. Backpropagation, one of the most popular learning algorithms, is biologically implausible due to a variety of reasons including the fact that the propagation of signals both forward and backwards through the same synapses occurs, error signals are propagated backwards through the axon and the model is not a real time model as explained in [BB90].
- AI

- › Explanation facility. An important failing of ANNs is that it is difficult to extract information from a trained net. Expert systems and other symbolic AI paradigms have the rules explicitly encoded and thus an explanation of decisions can be given. Caudill [Cau90b] [Cau90c] and Hillman [Hil90] describe how hybrid systems consisting of ANNs and expert systems can be constructed. Such systems are capable of explaining the decisions taken. Methods for extracting rules from trained networks have been devised (S. Sestito — pers comm) thus enabling an explanation facility.
- › Hardware realization. Few of the ANN models have been implemented directly in hardware. Although it is a fairly simple task to implement a trained net in hardware, creating a net in hardware that can learn is more difficult. Accelerator boards and special neural network chips have been devised to increase the speed of software simulations of ANNs. Generative algorithms, which create new units and connections, also pose obvious problems for hardware implementation.
- › Symbolic processing. ANNs have achieved remarkable success in *low level* processes such as visual, auditory and other sensory processing and preprocessing. It is not clear, however, how higher cognitive processes can be successfully accomplished using ANNs. Rumelhart and McClelland [RM86] and Hinton [RM86] have applied ANNs to tasks such as the learning of English past tenses and the learning of relationships. Various responses have been raised to critics that insist on the need for symbolic representations and rules as opposed to ANNs [BA91]:
  - » *Approximationist* approach: the symbolic approach approximates a connectionist model and provides a less detailed amount of performance information than does a connectionist model.
  - » *Compatibilist* approach: investigators seek to implement the explicit rules in a connectionist network. They claim that crucial benefits accrue as a result of the connectionist implementation of symbolic models.
  - » *Externalist* approach [RM86] (chapter 14): ANNs may develop the capacity to interpret and produce symbols that are external to the network.
  - » Brooks [Bro91] has suggested that the reliance on representation disappears when approaching intelligence in an incremental manner with reliance on interfacing with the real world through perception and action, although this hypothesis was introduced with his *subsumption* architectecture which differs slightly from the connectionist approach.

1.5. APPLICATIONS OF ANNs

From an engineering and applications view, one of the strongest arguments in favour of ANNs is the success with which they have been applied to various problems. Many of the problems have yet to be solved using non-ANN approaches or the performance of non-ANN systems doesn't yet approach that of the ANN-based systems. This is especially true in those problems with a large number of inputs (eg. machine vision), many simultaneous constraints or fuzzy inputs and rules.

ANNs have been applied to many fields including:

- Vision
  - › Fukushima *et al.* [FMI83] developed a complex network of cascaded levels of different types of units that learns to recognize handwritten characters, even after considerable deformation, rotation and positional shifting of the characters.
  - › Backpropagation applied to hand-written zip-code recognition [LMB90]. The throughput from the camera to classification was more than ten characters per second. The image data was first preprocessed by binarizing the image, determining the location of the writing on the envelope and segmentation of the image. There was no feature extraction in the preprocessing stage. The net learnt to classify handwritten numerals of various styles accurately.

- Image processing
  - › Binarization of grey-level images [BYK90] which is discussed in detail in a later chapter.
  - › Adaptive boolean filters [YAN] which are similar to median filters and were used to remove noise from images. The results were comparable to those obtained using the median filter.
  - › Image compression [Bur90] [MRS90].
    Feed-forward ANNs can map inputs from an input space into a different sized output space. Also, in multilayer nets, the hidden layer(s) are usually smaller than the input layer. The net can be trained to map the images to themselves; i.e. the input and output layers have the same number of units and the input and desired output patterns are identical. With a smaller hidden layer, the images will have to be represented in a compressed form in the hidden layer.

    The image can be compressed by supplying it as input to the net and using the activation of the hidden nodes to represent the compressed image. The image can be restored provided the weights from the hidden to output units and the interconnections between the hidden and output units are known. The compressed values become the input vector of the decompression net (which is equivalent to the original net except that the original input layer has been removed and the original hidden layer is now considered the input layer) and the output vector corresponds to the original image.

    An added bonus is that if similar images are used when training the net (eg. a sequence of human eyes was used by [Bur90]), the net could learn to generalize from the images, so that if a slightly noisy compressed image were supplied as the image to be decompressed, a correct image would result from the decompression process.

› Segmentation of images [BWS87].

• Acoustic processing

› NETTalk [SR86].

English spelling is inconsistent. Although there are weak rules, many exceptions and qualifications exist. NETTalk learnt how to read English text: given a string of characters it produced the correct phonemes which served as input to a speech synthesizer. NETTalk had a similar aim as DECTalk, a text-to-speech system produced by Digital Equipment Corporation. DECTalk, however, used a complex rule based system developed over many years to apply phonological rules to the text and a look-up table to handle any exceptions to the rules.

NETTalk was a three layer net with 203 input units (26 characters, and three punctuation and space symbols with a window of seven characters used to provide context to the character in the centre of the window) and 26 output units (23 articulatory and three stress features). The training set consisted of isolated words and continuous speech dictated by a child (sixth grade) and required 12 CPU hours on a DEC VAX.

The accuracy of the system was 95% on the training set and 80% on the test sets. The system passed through various stages while learning: babbling, substitution of one vowel for all vowels and one consonant for all consonants (i.e. the net learned to differentiate between consonants and vowels) and recognition of word boundaries and stresses. The speech produced was understandable after ten passes through the training set.

› Kohonen phonetic typewriter [Koh90].

This unsupervised ANN learnt to generate a topological map of the phonemes present in Finish or Japanese speech. Phonemes that are closely related (i.e. those that are pronounced in a similar manner) were grouped close together in the map. Two hundred to 300 words were used to train the network. The map created for a standard (typical) user could be modified by the new speaker using 100 more words to *fine tune* the map. The accuracy of the network to spot and recognize arbitrary continuous speech typically varied between 80% and 90% depending on the text and speaker. After compensating for coarticulation effects using an expert system, the accuracy increased to 92% to 97%. The network has been implemented directly in hardware and the latest versions operate in real time with continuous dictation.

• Signal processing.

› Sonar signal processing [Sou89] [DAR87] (Appendix F).
A backpropagation-trained network was used to classify sonar returns from an undersea metal cylinder and a cylindrically shaped rock of similar size into one of two possible classes: rock or cylinder. Sonar returns were collected from suitable objects placed on a sandy ocean floor. Two hundred returns (111 from a metal cylinder) from the 1200 returns made were selected to serve as the training set. The net consisted of 60 input units (which had the FFT[1] of the sonar return normalized to the range [0.0, 1.0] as input), one hidden layer and two output units.

The training took 30 000 passes through the training set with a resultant accuracy of 99.8% on the training set and 90% on the test set. This compared favourably to the *nearest neighbour* method (82% on the test set) and trained humans (92% on the training set, 84% on the test set). The features discovered by the network were similar to the perceptual cues utilized by trained humans.

› Signal analysis in oil well drilling [MSV90].
Sigma-log curves provide information concerning the safety and economical optimization of oil well drilling. A backpropagation-trained net was used to discover and classify isolated and multiple discontinuities in sigma-log curves and performed better than the *nearest neighbour* classifier. The input layer consisted of 20 to 40 units which formed a window that was slid along the sigma-log curve, with the units taking on the value of the curve at that point. The units were fully connected between adjacent layers. Two output units, to classify the input patterns as: *increasing discontinuity*, *decreasing discontinuity* and *no meaningful discontinuity* were used. For isolated discontinuities, only two units in one hidden layer were used while for multiple discontinuities, two hidden layers of eight and four units respectively were used.

• Robotics and control theory.

› Satellite orbit control system [ADGP90].
An ANN-based control system to keep a geostationary satellite correctly orientated with respect to an inertial reference system rotating with the earth was designed. Traditional control systems use a theoretical model which ignores all non-linear terms in the equations governing the movement of the satellite and inertial frame. The model also ignores non-linearities caused by flexible appendages (such as solar panels or aerials). The BP-trained net was used in conjunction with a control system governed by the theoretical model with information passing between the two systems. The network was used mainly when a lack of rules was evident and for

1 Fast Fourier Transform.

refinement of the standard systems control signals. Preliminary results were a steady accuracy of 0.0001 radians on each attitude angle.

› Robot arm controller [SK91].
A six degree of freedom (DOF) robot arm was used with a camera in place of the gripper. The task was to place the camera directly above an object placed at an arbitrary position in the viewing field (i.e. 2D hand-eye coordination). The network adapted quickly, within tens of trials, and formed the correct mapping from the input to output domain.

Miller *et al.* [MGK90] attempted a similar problem where the task was to make the centroid of an object on a conveyer belt follow a specified trajectory on a monitor by appropriate movements of a six DOF robot arm equipped with a camera in place of the gripper. The inputs to this network were the length and centroid of the object as well as an image of the object on the conveyer belt.

› Autonomous navigation.
D. Pommerleau (pers comm) developed an ANN (termed ALVINN) which learnt to guide mobile robots using visual input. Because of its ability to learn, ALVINN could adapt to new situations and thus cope with autonomous navigation tasks. ALVINN learnt to control a van in less than five minutes by watching a human drive. The trained system could drive in a variety of circumstances (single paved and unpaved roads, multi-laned lined and unlined roads) at speeds of up to 55 m.p.h.

Nguyen and Widrow [NW90] trained an ANN to back a truck-trailer combination to a loading dock without any forward movements from an arbitrary starting position and orientation. The network learnt to solve sequential decision problems by moving into positions that, although not decreasing the displacement error, made the later task easier.

• Medicine.

› Monitoring a patient's breathing rate [Key91].
It is not possible to codify the rules of medicine exactly because of their fuzzy nature. ANNs cope with fuzzy rules remarkable well and seem suited to applications in medicine. Anesthesiologists use a capnograph to visually monitor a patient's cardio-pulmonary system and the anesthesia machine. A capnograph is a plot of concentration of carbon-dioxide versus time, and is usually similar to a square wave with values graduating between zero and one. Capnographs are routinely used in operating-rooms and can give details such as when the patient is fighting the ventilator (and thus the drug may be wearing off) or whether the machine is malfunctioning. The output of the capnogram was fed into two neural networks directly which identified the start and end of the wave. Other neural networks were then used to classify the

wave and draw a conclusion about the patient's state.

- Recreation.
  - › Networks that plays Backgammon [TS89].
    A neural network was developed to decide whether a move in Backgammon is *good.* Presented with an initial board configuration, the board after a legal move and short term features (such as whether the move made a *point*[1]), the net had to judge the quality of the move. It was only used for situations other than a simple race to the end since very little strategy is involved in such races. Backgammon was used because of the large number of choices available at each stage; other games, such as checkers, can be played well by a deep search through all possibilities up to *n* moves ahead. This is harder to do in Backgammon because of the element of randomness due to the dice roll and the large search space involved. The trained ANN was tested by giving unseen board positions as input and playing it against a human and computer program. Against the computer opponent the network won about 65% of the games while against the human expert it won 35-40% of the games.
  - › Music composition which is discussed in a later chapter.

1.6. CONCLUSION

A short historical introduction and the workings of biological neurons and nets have been discussed. It was noted that although the knowledge of neurons, their functioning and interactions has grown since ANNs were first used as models of neural networks, few of the current ANNs employ units that are more complex than the simple McCulloch-Pitts neuron. It is not certain what level of biological accuracy is required for the models to be realistic models of biological neural networks. The basic terms to be used throughout the remaining chapters were defined as were the basic assumptions and requirements of ANNs. Finally some successful applications of ANNs to a variety of tasks were briefly mentioned. Certain of these applications are examined and expanded in detail in following chapters, as is a more complete description of certain of the learning algorithms and network architectures.

1 A *point* is when at least two tokens belonging to a player are on a position.

CHAPTER TWO

2.1. INTRODUCTION

Due to the large number of neural network models the following chapters will only consider layered, feed-forward network architectures that are trained using Backpropagation and similar learning algorithms. These are in fact the most popular and studied ANNs. A network consisting of k+1 layers of units will be referred to as a *k-layer* network. For a k-layer net with connections only between adjacent layers there will be *k* layers of adjustable weights. The k+1 layers of units in a k-layer network will be labelled layer 0 (the input layer) through layer k (the output layer). All the algorithms discussed in this and following chapters only use information that is available locally at the weights that are being adapted and are known as *local techniques* [Jur].

2.2. PERCEPTRON ARCHITECTURE

The Perceptron, or single layer Perceptron, [Ros58] was one of the first ANNs designed to recognize simple patterns and to be trained with a simple supervised learning rule. Fig 2-1 illustrates the general architecture of a Perceptron net. The net consists of two layers of units (hence *single layer*): an input and output layer. The input and output vectors of the Perceptron are usually binary valued vectors with a single layer of adaptable weights.

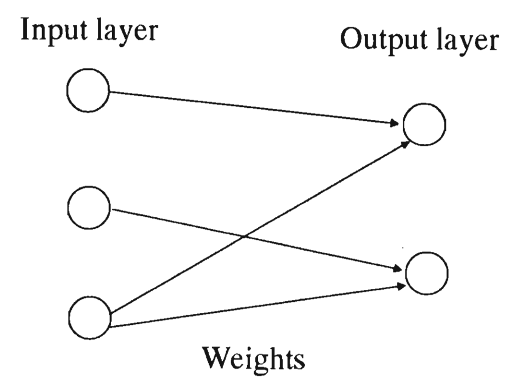


Figure 2-1 General architecture of the Perceptron. The single layer net consists of one layer of adaptable weights and an input and output layer of units.

The net input to an output unit i is defined as:

$$net_i = \sum_j w_{i,j} a_j - \Theta_i$$

where $w_{i,j}$ is the weight from input unit j to output unit i,
a_j is the activation value of input unit j,
and Θ_i is the threshold of unit i.

The activation value for an output unit j is given by:

$$o_j = f_h(net_j)$$

where f_h is the Heaviside function:

$f_h(x) = +1$ if $x <$ some threshold
$f_h(x) = -1$ otherwise

A simple net that classifies an input vector as belonging to one of two classes is shown in fig 2-2 [Lip87], where

$$y=f_h(\sum_{j=0}^{n} w_j i_j - \Theta) \text{ and}$$

$y = +1 \rightarrow$ the input vector belongs to class 0
$y = -1 \rightarrow$ the input vector belongs to class 1.

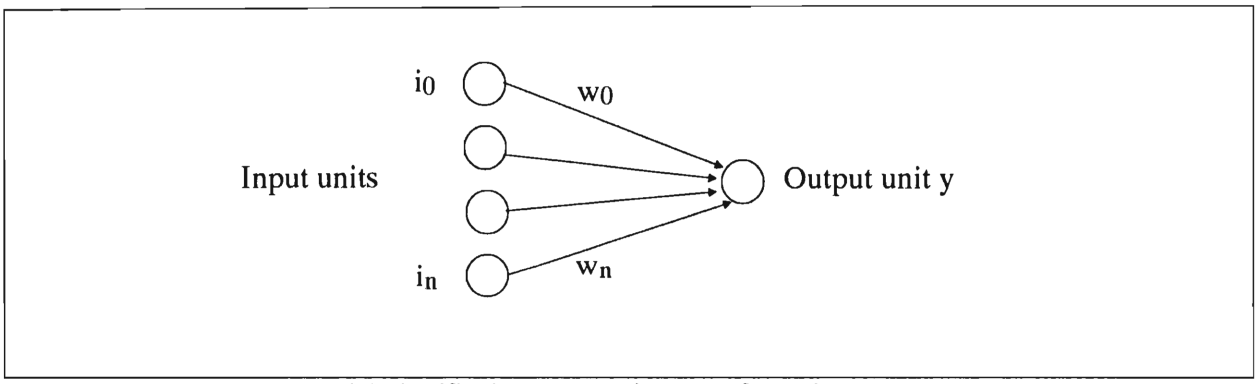


Figure 2-2 A perceptron net which classifies input vectors into one of two classes.

Generally only one output unit is active (+1) for a given input vector. Plotting a map of the decision regions created by the weights in the multidimensional space spanned by the input variables is a useful technique for analyzing the behaviour of the net [Lip87]. For figure 2-2, the two decision regions specify which input values result in class 0 and class 1 responses and the Perceptron determines a suitable set of weights that form two such decision regions separated by a hyperplane. For the case of two input units, the hyperplane is a boundary line (fig 2-3).

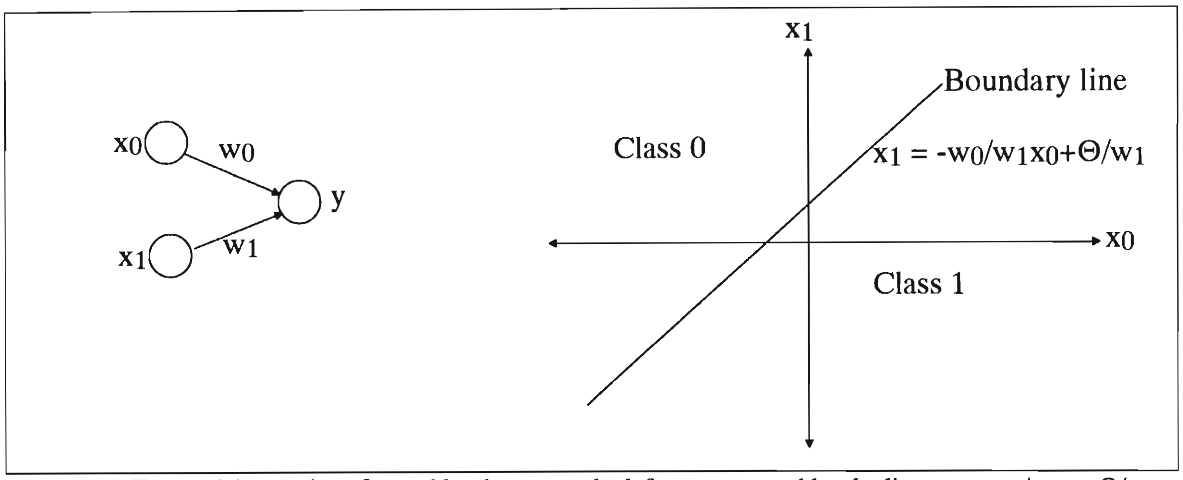


Figure 2-3 The decision regions formed by the net on the left are separated by the line $x_1 = -w_0/w_1x_0+\Theta/w_1$

The thresholds and connection weights can be adapted using various learning algorithms. The original *Perceptron Convergence Procedure* (or *delta rule*) developed by Rosenblatt [Ros58] for a Perceptron with N input units and M output units is [Lip87]:

STEP 0: Set Weights And Thresholds To Small Random Values
Set $w_{i,j}$ (0) and Θ_i to small random values.
$w_{i,j}$ (0) is the weight from input unit j to output unit i at time 0,
Θ_i is the threshold of the output unit i.

STEP 1: Present The New Input And Desired Output Vectors
Present inputs $x_0, x_1, ..., x_{N-1}$ and the desired output $d_0, ..., d_{M-1}$ to the net.

STEP 2: Calculate The Actual Output Vectors

$y_j = f_h(\sum_{i=0}^{N-1} w_{j,i}x_i-\Theta_j)$ for all output units j.

STEP 3: Adapt The Weights
$w_{j,i}(t+1) = w_{j,i}(t) + \eta(d_j(t)-y_j(t))x_i(t)$
where η is a positive proper fraction (the *learning rate*),
$d_j(t)$ is the j^{th} component of the desired output for the current input and
$x_i(t)$ is the activation value of input unit i.

STEP 4: If Learning Is Not Complete, Go To Step 1

Some points to note about the algorithm:

- The input vectors need not be binary valued, although they usually are. Steps 1 and 2 make no assumptions about the input vectors. Step 2 guarantees, however, that the output vectors are binary valued.

- The weights are only adapted if the actual output is not equal to the desired output. If the desired and actual outputs are equal, then d(t)-y(t)=0 and no weight change occurs.
- The learning rate determines the amount of influence the differences between the target and actual output vectors have on the weight change. Large values imply that the difference has a large influence on the new weight calculated while small values imply a smaller weight change. Faster convergence is achieved using large values since the weights then change by larger amounts. However, oscillation of weight values can occur. Using smaller values leads to more stable weight estimates with less oscillation. There is thus a conflict between fast convergence and stable weight estimates. Generally this parameter needs problem specific tuning.
- The thresholds can be considered to be the weights from an additional input unit, which always has an activation value of +1, to all the output units (fig 2-4). These weights can be adapted simultaneously with the other weights by step 3 of the algorithm.

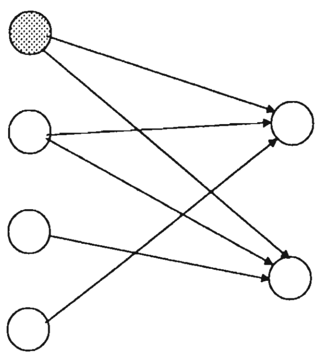


Figure 2-4 The thresholds can be replaced by an extra input unit (shaded) which has the value 1 for all input vectors. The weights from this unit will serve as the thresholds of the output units.

Rosenblatt, among others, proved the *Perceptron Convergence Theorem*:

> "If there exists a set of weights that separate the inputs into the desired classes then the delta rule will determine a suitable set of weights to accomplish this task".

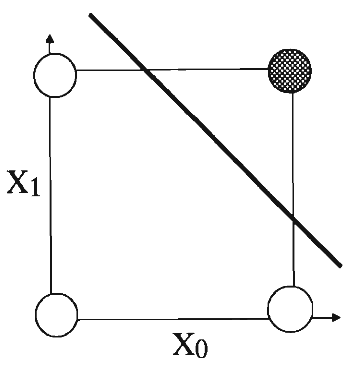
If such a set of weights exist, the problem is *linearly separable* or *separable.* A proof for the two-class case (fig 2-3), adapted from Minsky and Papert [MP88] (p168-170) appears in appendix A. The proof can easily be adapted to cover various other forms of the convergence procedure [MP88] (p175) including the case where **F** need not consist of unit vectors, can be a finite set or can satisfy an upper and lower bound of length. The theorem can be generalized to the case of more than two output classes.

2.3. CRITICISM OF THE PERCEPTRON

The Perceptron Convergence Procedure is guaranteed to find a solution to separable problems. However many input sets aren't separable. For example, the logic function AND (fig 2-5a)

X_0	X_1	AND
-1	-1	-1
-1	1	-1
1	-1	-1
1	1	1

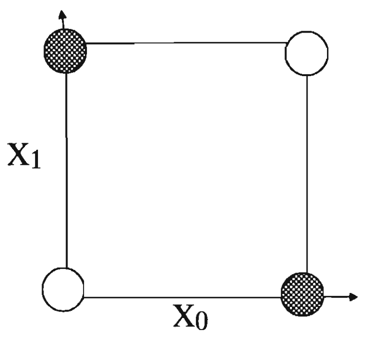
a



b

X_0	X_1	XOR
-1	-1	-1
-1	1	1
1	-1	1
1	1	-1

c



d

Figure 2-5 The truth tables for the logical AND and XOR appear in figures (a) and (c). Figures (b) and (d) are the diagrams associated with the truth tables. Shaded circles indicate a +1 while the others indicate a -1. The bottom-left coordinate of the boxes is (-1, -1) and the top-right is (+1,+1). From the figures it is easy to see that AND is linearly separable while XOR isn't.

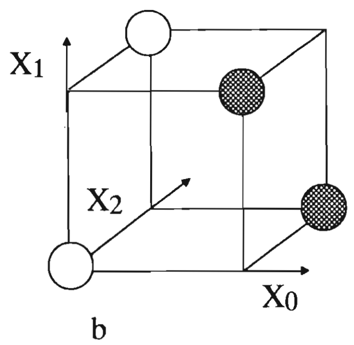
is clearly separable (fig 2-5b) while the XOR function (fig 2-5c) is not, as no line can be found that separates the two output classes (fig. 2-5d).

Non-separable sets, such as the XOR function, can sometimes be extended, by adding additional input units, so as to make the set separable by increasing the dimensionality of the weight space (eg. fig 2-6a & b). Pao [Pao89] extended this principle and his method is discussed later in this chapter.

Minsky and Papert analyzed the Perceptron architecture (with binary valued input vectors) thoroughly and their findings can be summarized as [MP88], [Pol88]:

X_0	X_1	X_2	eXOR
-1	-1	-1	-1
1	-1	1	1
1	1	-1	1
-1	1	1	-1

a



b

Figure 2-6 By adding an extra input (a), the XOR function is separable. Note that X2 does not determine the value of eXOR but is merely used to add an extra dimension to the weight space. The extended XOR function (b) is clearly separable using a plane.

- Their *Group Invariance Theorem* showed that linear threshold functions that are invariant under a permutation group can be transformed into a function whose coefficients depend only on the group.
- As various predicates scale, the size of the coefficients (weights) grows exponentially.

The *order* of a predicate was defined as the size of the largest conjunction in the minimal sum-of-products logical form of that predicate. The original Perceptron architecture of Rosenblatt [Ros58] consisted of an input *retina* with connections to a set of associator units (A-units) which had weighted connections to the output units (fig 2-7). The order as defined is equivalent to the number of inputs of the A-unit with the largest number of inputs [AM90] (p41).

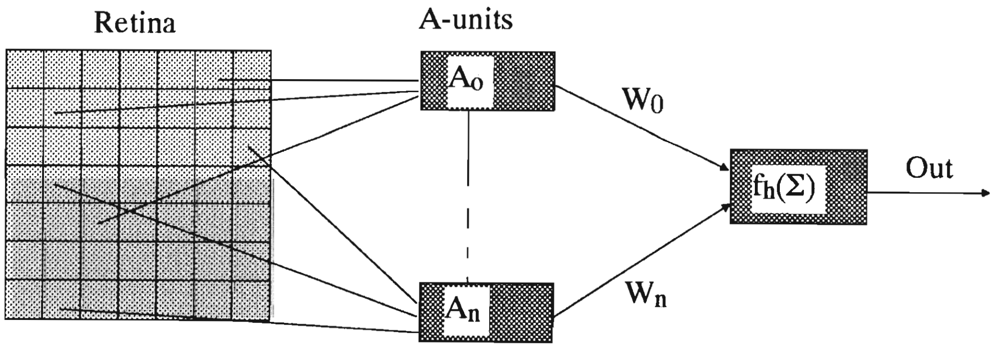


Figure 2-7 The original Perceptron architecture consisting of an input retina, the associator units (A-units), the modifiable weights (W_0 through W_n) which are multiplied by the signals passing through them, summed and a Heaviside function applied to derive the output (Out). The connections from the retina to the A-units are randomly chosen.

One of their central arguments concerning the inadequacies of the Perceptron is that the order should remain largely constant for any problem type regardless of the retina size [AM90].

This is analogous to the requirement that a conventional programs size should be largely invariant to the size of the task. For example, a sorting algorithm should not change in size when required to sort different quantities of numbers. But for tasks like the *parity* problem[1], the Perceptron is not of finite order. The order of this task is dependent on, and increases directly with, the size of the retina (the input vector) [AM90]. Other problems not solvable with a Perceptron of finite order include the *connectedness* problem[2] (fig 2-8). It is interesting to note that determining whether the two spirals of figure 2-10 are connected is difficult for human observers and requires cognitive processes rather than "pure" perception [Jul75].

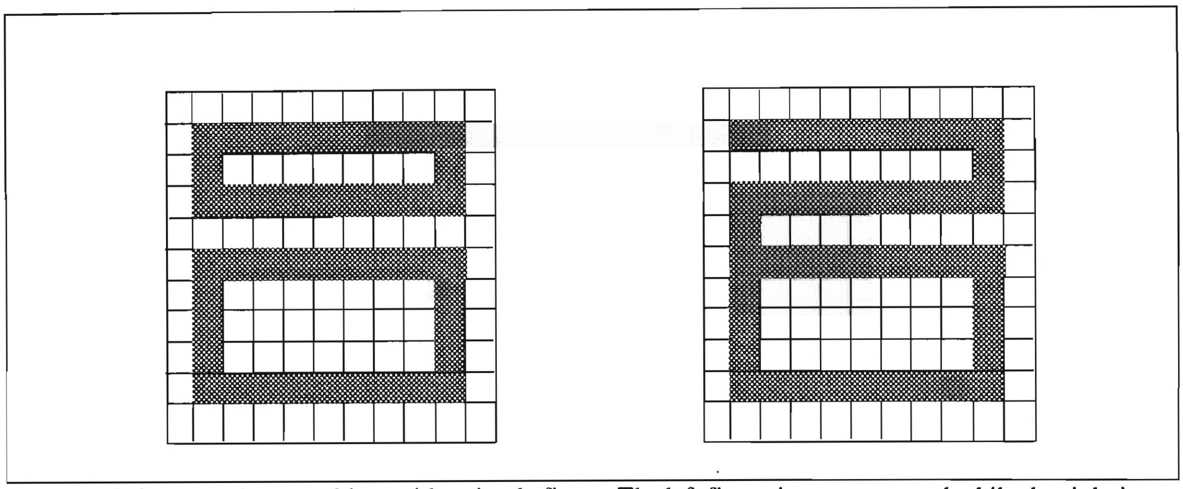


Figure 2-8 Connectedness problem with a simple figure. The left figure is not connected while the right is.

As Pollack has noted [Pol88], which is in agreement with Minsky and Papert, successful learning by the Perceptron (and in general by any ANN) is more dependent on **what** is to be learnt than on the **details** of the learning mechanism. A "law" often put forward by AI researchers is that a system must be able to potentially represent what it wants to learn. Minsky and Papert showed that for many tasks the Perceptron architecture could not represent the knowledge that was required to solve the problem. They argued that by adding layers of *hidden* units (fig 2-9) the representational properties of the net would increase and many of the problems associated with the Perceptron would be overcome [AM90] (p132). The hidden units recode the input patterns by forming an internal representation, and this recoding can support many required mappings from the input to output units [RM86] (p319). However, Minsky and Papert could not envisage a learning algorithm for a net containing hidden units.

Note, however, that Brooks [Bro91] has concluded from his research based on his *subsumption* architecutre that the reliance on representation disappears when approaching intelligence

1 Given a binary valued vector the task is to determine the parity of the vector.

2 Give a two dimensional pattern the task is to determine whether the pattern is continuous; i.e. whether the pattern consists of one continuous line.

in an incremental manner with reliance on interfacing with the real world through perception and action. He has put forward the fairly radical hypothesis that representation is the wrong unit of abstraction in the building of the bulkiest parts, such as vision, movement or obstacle avoidance, of intelligent systems.

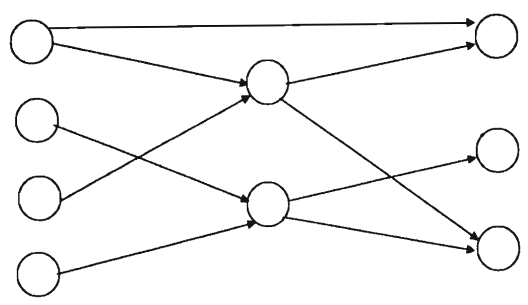


Figure 2-9 Feed-forward, non-recurrent net with one layer of hidden units.

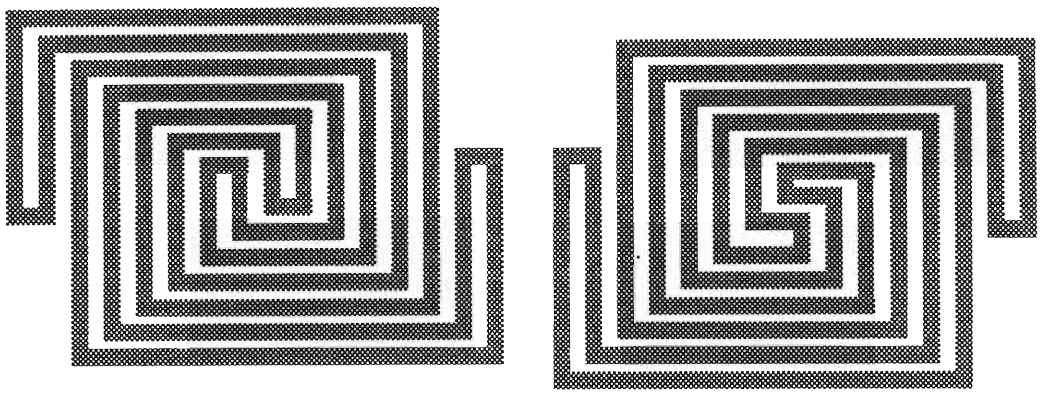


Figure 2-10 It is difficult to determine which of the two spirals is connected using "pure" perception. The right figure is connected.

If layers of hidden units are to be used it is clear that linear units (units having linear activation functions) should not be used:

Consider fig 2-9. Let:

> $\mathbf{I}_0$ denote the input vector,
> $\mathbf{W}_0$ denote the first layer of weights from the input to hidden layer,
> $\mathbf{I}_1$ denote the activation vector of the hidden layer units,
> $\mathbf{W}_1$ denote the second layer of weights from the hidden to output layer and
> $\mathbf{I}_2$ denote the activation vector of the output units.

Replacing Σiw by $\mathbf{W.I}$ (as in appendix A), note that

$$I_1 = \mathbf{W_0.I_0}$$
$$I_2 = \mathbf{W_1.I_1}$$
$$I_2 = \mathbf{W_1.(W_0.I_0)}$$
$$I_2 = \mathbf{W^*.I_0}$$

Clearly this can be extended to any number of layers of linear units and it can be seen that any number of such layers can be replaced by one layer of equivalent weights. Therefore, any hidden units should be non-linear.

Widrow and Hoff [WH60] developed one of the earliest trainable layered ANNs: the Madaline I [WL90]. The Adaline (Adaptive Linear Elements) units used are equivalent to the Perceptron with the threshold replaced by an extra unit which has an input permanently set to +1. By combining Adaline outputs using various logic functions, Madaline I was able to overcome certain of the Perceptron's problems. Fig 2-11 demonstrates how the XOR function can be

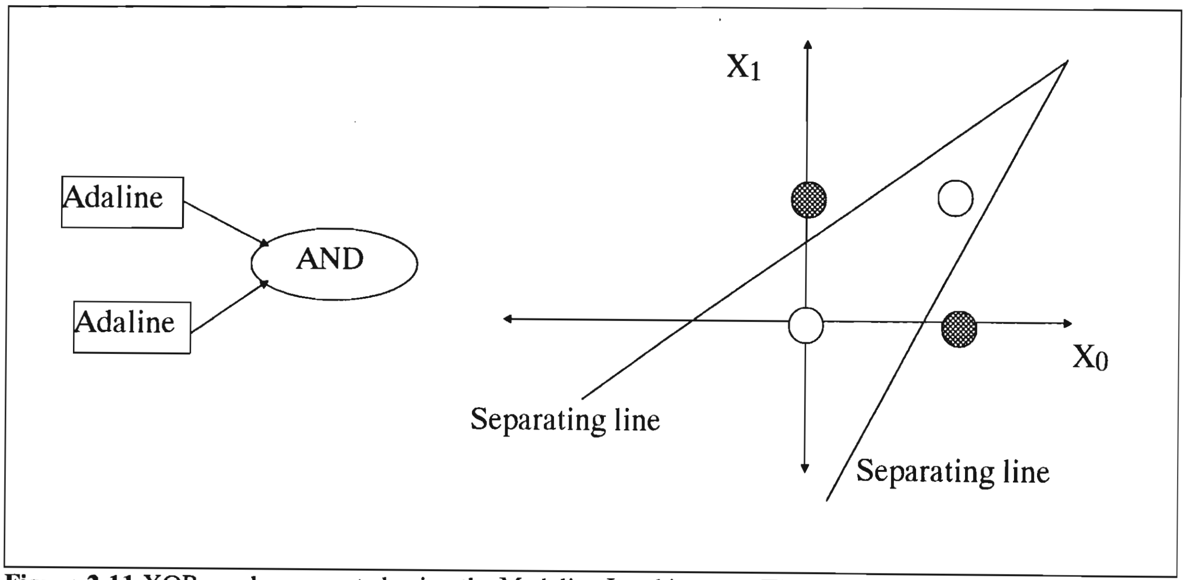


Figure 2-11 XOR can be computed using the Madaline I architecture. The two Adaline units compute partial solutions to the problem (the two separating lines) which are then combined (using an AND in this case) to derive the solution.

computed using one Madaline I net: the two Adaline units compute partial solutions to the problem which are then combined by ANDing the outputs to form the output of the Madaline.

Any logic functions, such as AND, OR and the majority-vote-taker, may be used to combine Adaline outputs. The second layer units are obviously fixed logic functions while the first layer units have adaptive weights. The Madaline I architecture was extended to the Madaline II which combined layers of Adalines while Madaline III used a sigmoid rather than Heaviside activation function and is equivalent to the Backpropagation algorithm [WL90].

2.4. BACKPROPAGATION

Rumelhart *et al.* [RMH86] developed a learning rule, termed *Backpropagation* (BP) or the *Generalized Delta Rule*, for an arbitrary layered feed-forward network (a multi-layer Perceptron). In such nets recurrent connections (connections within layers and from later layers) are not permitted but connections from earlier layers to later, non-adjacent layers are.

The total input to a unit x_i is the familiar summation used by the McCulloch-Pitts neuron defined as:

$$x_i = \sum_j w_{ij}\, y_j$$

where w_{ij} is the weight from unit j to unit i,
y_j is the activation value of unit j

For linear activation functions, the delta rule minimizes the squares of the differences between the actual and desired output values, summed over all the input-output patterns and output units. It can be shown (appendix B) that this corresponds to gradient descent on the error surface in weight space. For example, for a net with two weights the height of the error surface at a point (w_1, w_2) is equal to the magnitude of the error of a Perceptron with two weights having values w_1 and w_2 respectively. Without hidden units the error surface is paraboloid with a unique minimum, so gradient descent methods are guaranteed to find the minimum. However, it is not clear how to compute derivatives of the error surface, as required in gradient descent methods, when hidden units are present. Also, the error surface is not necessarily paraboloid in such cases and hence local minima can exist which implies that a gradient descent method can become caught in such minima.

Now, by restricting the units to those having differentiable, non-decreasing activation functions, the delta rule can be generalized to the Backpropagation rule [RM86] (p325):

Define (as in appendix B)

The net input to unit i for pattern p as:

$$net_{p,i} = \sum_j w_{i,j} o_{p,j} \qquad (1)$$

the output of unit j for pattern p:

$$o_{p,j} = f_j(net_{p,j}) \qquad (2)$$

and the error of all output units for pattern p:

$$E_p = \frac{1}{2}\sum_{j}(t_{p,j}-o_{p,j})^2 \qquad (3)$$

where $t_{p,j}$ is the jth component of the target output vector of pattern p, $o_{p,j}$ is the jth component of the actual output vector of pattern p and $w_{i,j}$ is the weight from unit j to unit i.

Let

$$E = \sum_{p} E_p \qquad (4)$$

be the error measure over all the patterns in the training set.

To correctly generalize the delta rule, set

$\Delta_p w_{j,i} \; \alpha \; -\partial E_p/\partial w_{j,i}$

$$\partial E_p/\partial w_{j,i} = \partial E_p/\partial net_{p,j} \,.\partial net_{p,j} /\partial w_{j,i} \qquad (5)$$

By (1)

$$\partial net_{p,j}/\partial w_{j,i} = \partial/\partial w_{j,i} \sum_{k} w_{j,k}\, o_{p,k} = o_{p,i} \qquad (6)$$

Define

$\partial_{p,j} = -\partial E_p/\partial net_{p,j}$

From (5)

$-\partial E_p/\partial w_{j,i} = \delta_{p,j}\, o_{p,i}$

Therefore to implement gradient descent in E, the weight change should be

$\Delta_p w_{j,i} = \eta \delta_{p,j} o_{p,i}$

as in the standard delta rule.

Now, $\delta_{p,j}$ can be determined using

$$\delta_{p,j} = -\partial E_p/\partial net_{p,j} = -\partial E_p/\partial o_{p,j} \,.\partial o_{p,j}/\partial net_{p,j} \qquad (7)$$

$\partial o_{p,j}/\partial net_{p,j} = f'_j(net_{p,j})$ from (2)

where f_j is the activation function of unit j and f'_j is the derivative of the activation function.

To compute $-\partial E_p/\partial o_{p,j}$ assume firstly that u_j is an output unit:

by the definition of E_p it follows that

$\partial E_p/\partial o_{p,j} = -(t_{p,j}-o_{p,j})$

which is the standard delta rule.

Substituting into (7)

$$\delta_{p,j} = (t_{p,j}-o_{p,j})f'_j(net_{p,j}) \qquad (8)$$

For u_j being any non-output unit:

$$\sum_k \partial E_p/\partial net_{p,k} \partial net_{p,k}/\partial o_{p,j} = \sum_k \partial E_p/\partial net_{p,k} . \ \partial/\partial\ o_{p,i} \sum_i w_{k,i} o_{p,i}$$
$$= \sum_k \partial E_p/\partial net_{p,k} w_{k,j}$$
$$= - \sum_k \delta_{p,k} w_{k,j}$$

Substituting into (7)

$$\delta_{p,j} = f'_j(net_{p,j}) \sum_k \delta_{p,k} w_{k,j} \tag{9}$$

Therefore, equations (8) and (9) give a recusive algorithm to calculate the δ's for all the units in the net.

Any activation function can be used provided it is non-decreasing and differentiable. Linear-threshold and Heaviside functions are not suitable and, as has been shown, linear functions are inadequate.

Rumelhart *et al.* [RHW86] used the *logistic* function:

$$o_{p,j} = \frac{1}{1+e^{-net_{p,j}}}$$
$$\partial o_{p,j}/\partial net_{p,i} = o_{p,j}(1-o_{p,j})$$

Thus, for the logistic function, the δ's are:

$\delta_{p,j} = (t_{p,j}-o_{p,j})o_{p,j}(1-o_{p,j})$ for output units (from (8)) and

$\delta_{p,j} = o_{p,j}(1-o_{p,j}) \sum_k \delta_{p,k} w_{k,j}$ for hidden units (from (9)).

2.5. BACKPROPAGATION ALGORITHM

The Backpropagation algorithm thus becomes [Lip87] [Wer90]:

STEP 0: Initialize Weights And Thresholds
Set all the weights and thresholds to small random values.

STEP 1: Present Input And Desired Output Vectors
Present the continuous valued input vector (x_0, x_1, .. x_{N-1}) and specify the desired outputs (d_0, d_1, ... d_{M-1}).

STEP 2: Calculate The Actual Output Vector
Using the logistic function and equations (1) and (2), calculate the activation values for all the units, including the output units ($y_0, y_1, \ldots y_{M-1}$), starting at the first non-input layer and moving, layer by layer, to the output layer.

STEP 3: Adapt The Weights
Starting at the output units and working back towards the input layer, adjust the weights using
$w_{i,j}(t+1) = w_{i,j}(t)+\eta\delta_i x_j$
where $w_{i,j}(t)$ is the weight from unit j to unit i at time t,
x_j is the output value of unit j,
$\delta_i = y_i(1-y_i)(d_i-y_i)$ if unit i is an output unit or
$\delta_i = y_i(1-y_i)\sum_k \delta_k w_{k,i}$ if i is any other unit.

STEP 4: If Learning Is Not Complete, Go To Step 1

A high-level algorithm of BP appears in appendix C.

It is important to start with small random values for the weights in step 0. If all the weights are initialized to the same value and unequal weights are required to solve the desired problem, the net would never learn [RM86] (p330). If the weights were initialized to the same value then, when the error is propagated back through the weights in proportion to the value of the weights, all hidden units directly connected to the output units would receive identical error signals and the weights after adaption would all be identical. This *symmetry* can be avoided by using small random starting weights.

As with the Perceptron, the thresholds can be replaced by adding an extra input unit, always having a value of 1, connected to all other non-input units. The weights from this unit can be adapted simultaneously with the other weights.

The updating of the weights (step 3) can occur at one of two times:

- After each pattern pair has been presented. This is known as *on-line* updating and generally gives faster convergence rates [KSV89].
- After each epoch and is referred to as *batch* updating. The errors for each unit are accumulated during the epoch and the weights are updated after the last pattern in the epoch has been presented.

2.6. CAPABILITIES OF MULTI-LAYER PERCEPTRONS

The capabilities of multi-layer nets are due to the non-linear activation function of the hidden units. Fig 2-12 [Lip87] illustrates the types of decision regions networks with 0, 1 and 2 hidden layers can construct using hard-limiting (i.e. Heaviside) activation functions. The single layer Perceptron forms decision regions separated by hyperplanes. A two layer Perceptron (having two layers of adaptable weights) can form any convex[1] decision region, possibly unbounded, in the space spanned by the inputs, by combining the hyperplanes formed by the first weight layer. A three layer net can combine these convex regions into regions which have, at most, as many sides as there are units in the first layer [Lip87]. Similar decision regions to those of fig 2-12 are formed for nets with multiple output units and sigmoid non-linearities. The behaviour of such nets is more complex because the decision regions are bounded by smooth curves and the analysis is thus more difficult.

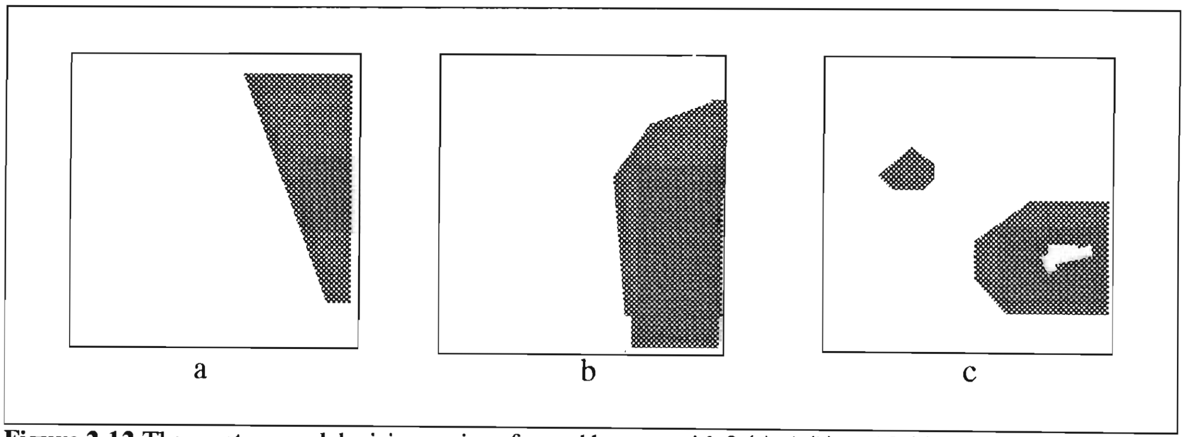


Figure 2-12 The most general decision regions formed by nets with 0 (a), 1 (b) and 2 (c) layers of hidden units, 2 input units and 1 output unit. Note that the activation functions for alll units are Heaviside functions.

Pao [Pao89] has criticized the view that hidden layers form combinations of hyperplanes to partition the input space into different regions. Because hyperplanes don't stop at the boundaries of the hypervolumes but may extend through the whole space, a hyperplane may serve in defining the boundary of one hypervolume but could extend into the interior of another hypervolume. An alternate viewpoint due to Nillson [Pao89] (p198) is that the successive layers carry out a sequence of mappings from one space to a space of another size until a representation (i.e. a mapping into a suitable space) is achieved where the desired separation is possible.

1 A *convex* region is a region where any line joining two points on the border of the region only passes through points within the region.

The *Kolmogorov Mapping Neural Network Existence Theorem* is a nonconstructive theorem by Kolmogorov [Lip87] [Sou89] (p77-79) which states that any continuous function of *N* variables can be computed using only linear summations and nonlinear, continuously increasing functions of only one variable.

Formally, Kolmogorov's Theorem can be stated as [GP89]:

> "There exist fixed increasing continuous functions $h_{pq}(x)$ on $I = [0, 1]$ so that each continuous function f on I^n can be written in the form
>
> $$f(x_1, \dots x_n) = \sum_{q=1}^{2n+1} g_q \left(\sum_{p=1}^{n} h_{pq}(x_p)\right)$$
>
> where g_q are properly chosen continuous functions of one variable."

In terms of networks this means that every continuous function of *n* variables can be computed by a network with two hidden layers where the hidden units compute continuous functions.

Hecht-Nielson [Hec87b] formulated the theorem as

> Given any continuous function, $\phi: I^n \rightarrow R^m$, ϕ can be implemented exactly by a two layer net with *n* input units, *2n+1* hidden units in one hidden layer and *m* output units.

Thus, a two layer net with N(2N+1) continuously increasing, nonlinear units can compute any continuous function of N variables.

Girosi and Poggio [GP89] have criticized the statement that the theorem *proves* that a network with two hidden layers is a usable or good representation for two reasons:

- A certain degree of smoothness is required by the units activation values. This is a prerequisite for the use of BP and similar gradient descent algorithms. However, a number of results [GP89] show that the inner functions h_{pq} of Kolomogorov's theorem are highly not smooth.
- Useful representations for approximation and learning are parametrized representations that correspond to networks with fixed units and modifiable parameters. Kolomogorov's theorem is not of this type: g_q is at least as complex, for instance in terms of the number of bits need to represent it, as f.

Thus, an **exact** representation of a function in terms of two or more hidden layers in a network seems doomed to failure. However, finding a good **approximation** using networks of two or more hidden layers is possible.

Cybenko demonstrated the existence of uniform approximations to any continuous function provided that the activation function is continuous [Jon90]. Jones [Jon90] extended this result to include any bounded sigmoidal activation function and developed an algorithm to construct such approximating functions. For the uniform approximation of discontinuous functions, two or more hidden layers may be needed (DeMers — pers comm).

Baum [Bau88] has shown that to implement an arbitrary dichotomy for any set of N points in general position in d dimensions, one hidden layer containing $\lceil N/d \rceil$ units will suffice. In fact, no smaller network can be used.

2.7. CRITIQUE OF BACKPROPAGATION

BP has several shortcomings. These include [Jur88]:

- Local minima. The Perceptron Convergence Procedure is guaranteed to find the global minimum, if one exists, because the error surface defined by the Perceptron architecture is a paraboloid in the space spanned by the weights and thus only one minimum exists. By introducing nonlinear and hidden units however, the error surface typically can contain many minima, not necessarily all global minima. Brady *et al.* [BRS89] have investigated cases where BP failed tasks that the Perceptron succeeded in separating. BP can fail to separate linearly separable problems when the unique minimum is constructed such that it fails to separate the input vectors into the desired classes. BP determines the minimum of the error surface and thus fails to separate the classes correctly. Other constructed examples used by Brady were problems where the local minima had large basins of attraction: there was thus a higher probability that BP would fall into such a basin and be trapped in a local minimum. Vogl *et al.* [VMR88] argue that in practical situations it may not be important or desirable to reach the global rather than a local minimum:
  - › The mathematical model used to model the system may describe the gross structure of the problem adequately but might be lacking in fine detail. Hence a local minimum might satisfy the gross structure but it is not possible to determine a global minimum satisfying the fine structure that isn't described by the model.
  - › It might be expensive computationally to determine the global minimum when a local minimum will suffice.
  - › The purpose of the search in weight space is to find a set of weights that satisfy the prescribed error criteria. A local minimum might satisfy the error criteria satisfactorily.
- Lack of convergence proof. It is easy to prove that if infinitesimally small weight adjustments are used, an optimal solution will be reached. However using infinitesi-

mally small adjustments is impractical. Therefore larger weight adjustments are used and convergence cannot be proved.

- Sensitivity to initial conditions. BP is a gradient descent method and due to the topology of the error surfaces (plateaus, local minima, etc.) the choice of initial weights affects the rate of convergence. Kolen and Pollack [KP90] have demonstrated BPs' extreme sensitivity to the initial weights and determined that the initial weights are a very significant parameter in the convergence time.
- Lack of incremental learning ability. Incremental learning occurs when patterns learnt previously aren't *unlearnt* or *lost* when new patterns are learnt. Because of the method used in BP to update all the weights when learning, nets trained with BP do not have this ability. This bodes ill for adaptation of a BP trained net to changing situations without having to relearn patterns learnt previously.
- Speed of convergence. This is possibly the best researched problem and is discussed in section 2.8.
- Scalability of the algorithm. Empirical and theoretical studies of BP have shown that the algorithm's performance scales badly as the problem size is increased [TJ88]. In chapter three, an algorithm that seems to scale better than BP is presented. An alternative possible solution is to train a set of small nets on subproblems of the task and combine them in some way to solve the task. Training the small nets is often faster than training one large net and the scalability problem doesn't affect the smaller nets as badly as it does the larger net. Lippmann [Lip89] discusses this approach as has been used in speech recognition tasks.
- Sensitivity to weight and input errors. When simulating nets by computers or implementing nets in hardware, the weights have to be represented in some manner. Generally 32 bit floating point values are used (Fahlman — pers comm). Hochfield and Fahlman (pers comm) have investigated the precision required using Fahlman's Cascade-correlation architecture and determined that at least 12 bits per weight were required. The accuracy of such representations and the number of significant digits could affect the performance of the net. Stevenson *et al.* [SWW90] noted that the sensitivity of a net to weight errors increases with the number of layers in the net and the percentage change in the weights during weight adaptation.

Various heuristic methods have been suggested to overcome some of these problems. For networks that don't learn the correct patterns (possibly by being trapped in local minima) Caudill [Cau91a] suggests:

- Reset the weights and retrain the network. This positions the start of the search at a different location on the error surface with the hope that the new position will eventually lead to a better solution. This is a last resort and generally not very feasible for large tasks with long learning times or real-time learning requirements.

- Perturbing the weights by adding small random amounts to the weights. This change in the weights moves the search to a different point on the error surface, thus hopefully escaping from the local minimum. Choosing very large perturbations can lead to slower learning since it might move the current point of the search far from a desirable solution. Similarly, small perturbations might not change the position in weight space sufficiently to escape from a local minimum especially if the minimum has a large basin of attraction.
- Don't use deeply layered networks. This leads to a great dilution of the error signal and is discussed in section 2.8.
- Add small amounts of noise (i.e. small random values) to the input patterns. This encourages generalisation by forcing the network to learn the prototypes of the patterns rather than the individual patterns. The problem is in choosing the amount of noise to add.
- Accept a larger error and be satisfied with the minimum reached.
- Use a larger number of units in the hidden layers by increasing the number of hidden units by 10% if necessary.

Jurik extended these heuristics [Jur88]:

- Start with more hidden units than there are input and output units. Retrain the network with progressively less hidden units until further reductions cause a bad degradation in the performance of the network. This too encourages generalisation. For tasks that require large learning times this option is rarely feasible.
- Have as many different input-output pattern pairs as possible to force the network to extract the prototypes of the patterns rather than learning the patterns themselves.
- Back-propagate only those errors with a magnitude that exceeds some threshold thus speeding up the learning process by only considering "large" errors. This method was used in the training of NETTalk.
- Average the error over a number of patterns before adapting the weights by BP. This is a mixture of the batch and on-line weight updating schemes.
- In each epoch present the "difficult" patterns more than once. For example if the network is not learning pattern 2 correctly, pattern 2 can be presented more than once per epoch.
- When adding noise to the weights during adaptation, the noise can be gradually reduced as learning progresses. Large amounts of noise early in the learning progress allow large steps through weight space while smaller steps allow finer searching of the area which hopefully contains a minimum.

2.8. RATE OF CONVERGENCE

BP was defined so as to be a gradient descent method. Gradient descent is equivalent to hill-climbing techniques on the negative of the error surface. However hill-climbing, and thus gradient descent, has obvious problems [Win84] (p94-95):

- The *foothill* problem occurs when there are secondary peaks in the search space which draw the hill-climbing procedure away from the global maximum.
- The *plateau* problem occurs when large "flat" spaces separate the maxima and the hill-climbing procedure can "wander" about such spaces aimlessly.
- The *ridge* problem occurs if the directions in which the method searches for a maximum are limited and thus potentially good search directions are overlooked.

The problems are greatly increased when the number of dimensions in the search space increases due to extra parameters (weights).

Minsky and Papert [MP88] (epilogue) note that hill-climbing works for "toy" problems but becomes impractical for larger and more complex problems. BP is merely a recursive way of calculating the gradients with less computational effort than "normal" gradient descent methods and doesn't alleviate the basic problems associated with hill-climbing procedures. Nothing has been proven about the range and types of problems for which BP works efficiently and dependably.

A problem with the gradient descent technique of BP is the rate at which it converges to a minimum. Numerous improvements to the algorithm have been suggested including:

- Adding a *momentum* term to the weight update rule [RHW86] [RM86] (p330) where a fraction of the previous weight update is added to the current weight update. So step 3 of the algorithm is replaced by:
  $w_{i,j}(t+1) = w_{i,j}(t) + \eta\delta_i x_j + \alpha(w_{i,j}(t) - w_{i,j}(t-1))$
  where α is the momentum constant and is usually chosen such that $0 \leq \alpha \leq 1$. Increasing the learning rate, η, can lead to faster learning but also to oscillation in the error performance of the net during learning similar to what is found when increasing the learning rate in the Perceptron rule. Including the momentum term can increase the rate of convergence without oscillation. The choice of the parameters η and α is difficult and problem dependent with the choice depending on the topology of the local error surface. Ideally the values of η and α should be able to change during the search of the error surface. Methods to accomplish this are given in a later chapter.
- *Stuck* units. Units can be turned off (have an activation of 0) early in the learning stage and stay stuck in this zero state. This problem is due to the points where the derivative of the sigmoid function approaches 0. Fahlman [Fah88] has termed such points *flat*

spots. In BP as the error is back-propagated through the net, the error seen by each unit j is multiplied by the derivative of the sigmoid function. The derivative of the logistic function is $o_j\,(1-o_j)$ where o_j is the activation value of unit j. This derivative approaches 0 as o_j approaches 0 or 1. So, even if the output represents the maximal possible error (eg. the output is 1 instead of 0 if j is an output unit), only a small fraction of the error is passed back through the net. Theoretically such units will eventually recover but this can take a long time. When simulating a net, round-off error might cause such units never to recover. Fahlman [Fah88] added a constant, 0.1, to the derivatives of the output units to overcome the flat spots. This made a dramatic difference with the learning time reducing by almost half in the 10-5-10 encoder benchmark[1]. Chen and Mars suggested ignoring the derivative at output units and consider the output units to be linear [Tve91]. For such a case it was suggested that a second learning rate, $\eta_2 = \eta/10$, be used when adapting the weights of the units in the hidden layers.

- Non-linear error function. The flat spots at the output units can be eliminated by using a non-linear error function that approaches infinity as the difference between the desired and actual output approaches +1 or -1. Fahlman [Fah88] used the hyperbolic arctangent of the difference between the actual and desired output which had a modest effect (±25% improvement) on the convergence rate when tested on the 10-5-10 encoder benchmark.
- Changing the range of the logistic function. Consider an input unit j having an activation of 0. Now the weight from this unit to the ith hidden unit is changed in proportion to the activation value of unit j and the error signal back-propagated from unit i (step 3 of the algorithm). Because the activation value of unit j is 0 no change is made to this weight. Two ways to remedy this are:
  - › Use inputs of 0.1 and 0.9 instead of 0 and 1 respectively in the input vectors. In such cases, all input units will always have an activation value and weight updating, although possibly small updates, will take place.
  - › Change the range of the logistic function to be symmetrical about 0 with lower and upper bounds of -0.5 and +0.5 respectively by changing the logistic function to be $f(x) = 1/(1+e^{-x}) - 0.5$. This approach was used by Stornetta *et al.* [SH87] with a 10% faster convergence rate and 20% smaller variance (the variance was due to averaging the results over a number of trials with different starting weights) resulting. They claim that in general problems this method would decrease the convergence time by 30% to 50%.
- Heuristic adaptation of the learning rate parameter. Vogl *et al.* [VWR88] proposed two heuristic methods:

1 Given a binary string of length N which has only one component with the value 1, the task is to teach the net to map the string to itself using M units in the single hidden layer. This is known as the N-M-N encoder problem.

› The learning rate, η, should be varied according to whether the iteration decreases the total error for all the patterns. If the weight update does reduce the total error it is assumed that the search direction is a good direction and η is multiplied by a constant φ_1.

› If the step produces an error larger than n% (usually 1-5%) of the previous error then η is multiplied by a constant β_1, the momentum parameter is set to 0 and the step is repeated with the last weight change ignored. When a successful step is taken, the momentum parameter is reset to its original value and training continues as normal.

These methods lead to a significant improvement on the 3x3 T-C problem[1] (2119 epochs down to 826 epochs) and the 7x7 T-C-L-X[2] (30 000+ epochs down to 1 000 epochs).

- Gain parameter. Kruschke and Movellan [KM91] introduced a gain term to the BP equations:
The activation of a unit i is defined as
$a_i = f(g_i net_i)$ where g is the real valued gain for unit i and net_i is the net input to unit i as defined previously.

$\Delta w_{i,j} = \varepsilon_w \delta_i a_j$ is the weight change where ε_w is the learning rate for weight updates and

$\delta_i = (\sum_k \delta_k w_{i,k}) f'(g_i net_i).g_i$ as in standard BP.

Gradient descent on the error with respect to the gains can be computed in a manner similar to gradient descent on the error with respect to the weights, giving a rule ($\Delta g_i = \varepsilon_g \delta_i net_i / g_i$)to update the gain values. It is easy to incorporate the gain calculation in the standard BP algorithm since all quantities are locally available to the weight being updated. It was shown that the gain was beneficial for speed and generalisation while simulations confirmed significant speed improvements. When units are learning the gain term speeds up the learning. However, with very few hidden units there might be

1 The task is to differentiate "T"s and "C"s that appear in one of four orientations on a 3x3 grid. The T's and C's are constructed in such a manner that the T and C of the same orientation differ in at most two positions.

2 The task is similar to the T-C problem except that "T", "C", "L" and "X" are used on a 7x7 grid. Again the letters of the same orientation are constructed so as to differ in at most two positions.

no successful units and in such cases the gain term won't significantly improve the performance of the net [KM91].

- Choice of initial weights. The initial weights are an important factor in the speed and success of convergence. It seems intuitively obvious that starting with a good set of weights will enhance the performance of the algorithm. However, determining a set of good weights is the whole purpose of BP and it is not clear that good sets can be determined in one step. Kramer (pers comms) suggests using Principle Component Analysis to choose a set of promising initial weights.
- Shortcut connections. "Fully connected feed-forward net" is usually taken to mean that each layer is fully connected to the next (later) layer. However, the BP algorithm is defined for general non-recurrent nets. Therefore BP is suitable for nets with *shortcut*[1] connections. Fahlman uses the term to mean that each unit receives incoming signals from all units in all earlier layers (pers comm). Adding such connections could increase the convergence rate since such nets give the hidden units clearer signals to follow (Fahlman — pers comm). Although shortcut connections increase the dimensionality of the weight space (by adding extra weights) Kempka (pers comm) has empirical evidence that dramatic decreases in learning times occur when using shortcut connections and, in addition, when using activation functions other than the logistic function, the generalization ability of the net increases.
- Dilution of error signal. Clearly the error is relevant when adapting the weights leading to the output units but as the error is back-propagated (and attenuated by the derivative of the activation function) the error bears less and less information to the earlier units (Fahlman — pers comms). Thus, in a deeply layered net, the error is *diluted* and bears little information for updating the weights of the earliest layers.
- Enhancing the input representation. As mentioned previously (e.g. fig 2-6) by extending the input pattern representation it is possible that a simpler net may solve the problem. Pao [Pao89] (chapter 8) has extended this to his *Functional Link Net.* By including *higher-order* connections, by enhancing the input representation, generally results in a dramatic decrease in the learning time. Pao used two models to enhance the input pattern representation:
  - › Function expansion model. Each component of the input vector is acted on by some set of functions to yield the enhanced representation. For example, given the input $(i_0, i_1, \ldots i_n)$, the enhanced input would be $(f_0(i_0) \ldots f_k(i_0) \ldots f_0(i_n) \ldots f_k(i_n))$. A set of suitable functions, for example, include
    $f_0(x) = x$

1 Shortcut connections are connections from earlier, not necessarily adjacent, layers to later layers.

$f_1(x) = x^2$

$f_k(x) = \sin \pi x$

› Outerproduct model. This is a special case of the functional expansion model. Each component of the input vector is multiplied by the entire input vector to form an enhanced representation. This amounts to forming the outerproduct between each input vector and itself.

Both models can be used recursively, i.e. repeated on the enhanced representation, to further enhance the resulting representation. In practice it is important to *prune* the enhanced representation to remove any duplicate units, especially in the outerproduct model, to reduce the input dimensionality. For example, using the outerproduct model to enhance the input vector (x_1, x_2, x_3) the pruned input pattern used would be (x_1, x_2, x_3, x_1x_2, x_1x_3, x_2x_3, $x_1x_2x_3$). Once the input representation has been enhanced, Pao found that the Perceptron often was adequate to solve the resulting problem. For example, using the outerproduct model on the XOR problem makes it solvable using the Perceptron (fig 2-13).

X_0	X_1	X_0X_1	XOR
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

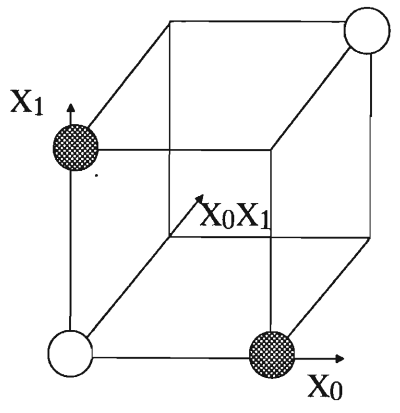


Figure 2-13 The XOR problem can be solved by a perceptron when the input representation is enhanced using the outerproduct model. The truth table (left) indicates the extra input (X_0X_1) which has no affect on the XOR values. It does, however, make the problem linearly separable (right figure).

2.9. CONCLUSION

The Backpropagation algorithm is a gradient descent algorithm designed to find a minimum of the error surface defined in weight space. BP is a generalisation of the Perceptron Convergence Procedure which was found to be limited to the solution of separable problems. Although BP has shown some remarkable success it is plagued by many problems. One of the most serious problems is that of the rate of convergence. Many improvements to the algorithm have been suggested with varying success.

The question remains whether the improvements are merely improvements to an inadequate algorithm or whether BP is less limited than the Perceptron. Minsky and Papert state [MP88] (prologue) that the theorems on the limitations of the Perceptron can be generalised to BP with little modification which would seem to suggest that BP, like the Perceptron, is flawed. However, from an applications point of view, BP can be remarkably successful although little theoretical guidance exists as to which tasks BP is suited to solving.

CHAPTER THREE

3.1. INTRODUCTION

Tesauro and Janssens [TJ88] conducted an empirical study of BP learning times to examine the *scalability*[1] of BP, using the parity benchmark. It was found that the learning time was proportional to 4^n where *n* was the number of input units ($2 \leq n \leq 8$) which is consistent with recent theoretical analysis of similar algorithms [TJ88]. Hinton has shown [Hin89] that the training time for BP is $O(n^3)$ on serial machines ($O(n^2)$ on parallel machines with one processor for each weight) and at least $O(n)$ training patterns should be used, where *n* is the number of weights in the net.

From this it can be seen that the learning time increases exponentially as the size of the net, and thus the number of weights, increases linearly. This is known as the *scalability problem.* It is necessary to find solutions to the following problems:

- Learning speed. In the previous chapter various methods were suggested to increase the learning speed of BP. However, these solutions often add additional processing or storage requirements to the basic BP algorithm with only a marginal improvement.
- An efficient algorithm, bounded by a polynomial in the input size, is required so as to reduce the effects of the scalability problem.
- An algorithm that always finds the global minimum or is rarely trapped in local minima.

In this chapter two algorithms will be discussed: *Quickprop* (QP) is an algorithm that seems to scale well and is faster than BP while *Backpercolation* (Perc) seems to avoid local minima and is faster than BP.

3.2. CONJUGATE GRADIENT TECHNIQUES

In step 3 of the BP algorithm the weight is updated by:

$$w_{ij}(t+1) = w_{ij}(t) + \eta\delta_i x_j$$

1 The scalability of a neural net algorithm is a measure of how the time required to converge increases with increasing input sizes.

Consider the more general problem of minimizing a function f(x) where $x = (x_1 \dots x_N)^T$ is a vector to be determined. In the case of ANNs, x is a weight vector that solves the problem. Clearly BP is a method for solving such a problem where f(x) is the error function as defined in the previous chapter. By estimating a minimum $x^{(0)}$, the estimate can be improved iteratively [Osb85] [Wal75]:

$$x^{(t+1)} = x^{(t)} + \alpha^{(t)} d^{(t)} \qquad (1)$$

where $\alpha^{(t)}$ is a scalar step length and
$d^{(t)}$ is the search direction.

$\alpha^{(t)}$ can be determined by a line-search as in the Steepest Descent algorithm. Choosing

$$d^{(t)} = -\nabla f(x^{(t)})$$

gives the steepest descent direction and by identifying $w_{ij}(t)$ with $x^{(t)}$, η with $\alpha^{(t)}$ and $\delta_j x_i$ with $d^{(t)}$, BP can be seen to be similar to gradient descent methods.

However gradient descent methods converge slowly [Sha90]. BP differs from (1) in that the learning rate, η, is a constant unlike $\alpha^{(t)}$.

Using Newton's direction defined by:

$$d^{(t)} = -(\nabla^2 f(x^{(t)})^{-1} \nabla f(x^{(t)})) \qquad (2)$$

where $\nabla f(x^{(t)})$ is the gradient of f at $x^{(t)}$ and
$\nabla^2 f(x^{(t)})$ is the matrix of second partial derivatives of f at $x^{(t)}$.

can lead to faster convergence. However, this requires second order information which requires a global procedure (and is thus not available to units as local information) and greater storage requirements. In addition, it is not easy to parallelize this method, unlike gradient descent [Sha90].

Quasi-Newton methods approximate (2) by

$$d^t = -H^{(t)} \nabla f(c^{(t)}) \qquad (3)$$

where $H^{(t)}$ is the positive definite matrix approximation of $(\nabla^2 f(x^{(t)}))^{-1}$ obtained from first order information.

These methods have the advantage that the second order partial derivatives are not needed but $H^{(t)}$ is usually a totally dense matrix which increases the storage requirements.

Conjugate gradient methods determine $d^{(t)}$ as a linear combination of the current gradient vector and the previous search direction. By starting with

$$d^{(0)} = -\nabla f(x^{(0)})$$

the sequence $d^{(t)}$ chosen as

$$d^{(t+1)} = -\nabla f(x^{(t+1)}) + \beta^{(t)} d^{(t)} \tag{4}$$

where $\beta^{(t)}$ is a scalar parameter chosen so as to ensure that the sequence of vectors, $d^{(t)}$, satisfy a mutual conjugacy condition.

One method to calculate $\beta^{(t)}$ is the Polak-Rebiere rule [KS89]:

$$\beta^{(t)} = ((g^{(t+1)}-g^{(t)})^T.g^{(t+1)})/((g^{(t)})^T.g^{(t)})$$
where $g^{(t)} = \nabla f(x^{(t)})$

which will calculate the successive search directions. From (4) it is clear that conjugate gradient techniques require little extra storage and are easily parallelizable [Sha90].

From this information the deficiencies of the BP algorithm can be highlighted [KS89]:

- Because the momentum term is constant, the search direction may not be in the direction of steepest descent and in such cases any weight change will increases the total error.
- If the search direction is in the descent direction, the learning rate may be large enough to move from one wall of the "valley" to the opposite wall leading to oscillation and slower learning.

Finding values for the momentum and learning rates is thus difficult and problem specific.

Conjugate gradient methods, in general, share all the desirable properties of steepest descent methods: low storage requirements, easy implementation and parallelization but, when implemented properly, converge more quickly and generally are as quick as quasi-Newton methods [Sha90]. However, care must be taken in the line-search when determining $\alpha^{(t)}$ and a *restart* procedure may be necessary to restart failed line searches by using the negative of the current gradient as the new search direction after n or $n+1$ weight updates, where n is the number of weights in the net [Atk89] [Sha90].

The disadvantage of conjugate gradient methods is the number of function evaluations required (up to twice that of Newton methods and, depending on the line-search algorithm, up to twice as many gradient evaluations). Hence there is a trade off between the number of function evaluations required and speed of convergence. Moller [Mol90] has noted that as a minimum is approached, the rate of convergence of conjugate gradient methods decreases.

Moller [Mol90] developed a *scaled* conjugate gradient method (SCG) which uses second order information but only requires O(N) memory (where N is the number of weights in the net), has no user dependent parameters and avoids the time consuming line-search by using a step size scaling mechanism. Tsioitsias (pers comm) notes that conjugate gradient methods are more susceptible to rounding errors than other direct methods, may have erratic behaviour and may cause the weights to grow very large very quickly.

3.3. QUICKPROP

To recap, BP uses the partial first derivatives of the error function with respect to each weight to perform gradient descent in weight space so finding a minimum of the error function. Using infinitesimal steps down the gradient guarantees that a minimum will be reached. Clearly infinitesimal steps are impractical. Empirically it has been determined that for many cases the minimum reached is a good enough solution for most purposes [Fah88]. However, the set of partial first derivatives carries little information about the curvature of the error surface. For this information, higher-order derivatives need to be used.

Two approaches have been used to try and account for the curvature of the error surface:

- By adjusting the learning rate heuristically based on previous weight updates. Momentum achieves this in a limited sense as do algorithms such as the *Delta-Bar-Delta* algorithm [Tve91]. All such methods improve the convergence rate of BP to some extent.
- By using the second partial derivatives of the error with respect to each weight which supply the required higher-order information. Conjugate gradient and Newton methods use this approach but generally require global calculations to compute the true second derivatives. Therefore, some approximation of the second derivative is usually used.

Quickprop (QP), developed by Fahlman [Fah88], is a second order method based loosely on Newton's method. It is similar to standard BP using batch updating but a copy of the error derivative, $\partial E/\partial w(t-1)$, calculated during the previous training epoch, is kept as are the differences between the current and previous weights. As in standard BP, $\partial E/\partial w(t)$ is available at the time of weight updating.

Two assumptions about the error surface are then made:

- The error vs. weight curve for each weight can be approximated by a concave parabola.
- The change in the error curve, as seen by each weight, is not affected by all the other weights that are changing simultaneously.

For each weight, independently, the previous and current derivative of the error slopes and the weight change between the points at which these slopes were computed is used to construct a concave parabola. This construction is trivial and only uses information local to the weight being updated. A "jump" is then made to the minimum of the constructed parabola:

$$\Delta w(t) = \frac{S(t)}{S(t-1)-S(t)}\Delta w(t-1)$$

where S(t) is the current value of the slope $\partial E/\partial w(t)$, and $\Delta w(t-1)$ is the previous weight change.

Clearly the new weight value is only an approximation of the optimal value but, by applying the method iteratively, the method is very effective. The momentum term added to standard BP is not required although the learning rate parameter is retained. The advantage is that with QP large steps that jump to the minimum of the parabola are allowed rather than the smaller steps down the side of the parabola towards the minimum as in BP.

When using the QP update rule three situations can arise:

- The current slope is smaller than the previous slope but in the same direction. The QP rule causes a step further down the wall of the parabola to be taken.
- The current slope is in the opposite direction of the previous slope. This implies that the minimum of the parabola has been passed. The QP rule places the new position somewhere between the current and previous positions, which will be closer to the minimum.
- The current slope is in the same direction as the previous slope but is of the same or larger magnitude. Following the QP rule explicitly would cause an infinite step or a step moving up the current slope and away from the minimum to be taken.

Fahlman introduced an extra parameter, the *maximum growth factor* (μ) which limits the size of any weight change that may take place, as a heuristic measure to overcome the last situation. Therefore, no weight change is allowed to be larger than μ(previous weight change). If the step computed by the QP rule is infinite, too large or moves up the current slope, the step used is μ(previous weight change). If μ is chosen as too large, chaotic behaviour results in the net with slow or no convergence. From experiments, a μ value of about 1.75 works well for a range of problems.

A high-level algorithm of QP appears in appendix C.

3.3.1. Quickprop features

Because the QP rule uses previous slopes when calculating the weight update, there must be a method to *bootstrap* the process when first starting or when the step size previously was 0 but a non-zero slope has been computed for the current epoch. For both these cases the previous slope will not be available. Intuitively gradient descent based on the current slope and some learning rate parameter, η, seems suitable. Fahlman tried two methods for this bootstrapping process:

- Whenever the previous weight step fell below some fixed threshold, the gradient descent method was used in place of the QP rule. This method worked but for larger problems suspicious behaviour occurred in the vicinity of the threshold.
- A gradient descent term was always added to the value computed by the QP rule. This worked well except for cases when the minimum of the parabola had been overstepped. In such cases the QP rule accurately located the minimum but the gradient descent term forced the new position past this minimum. The solution was always to add η(current slope) to the value calculated by the QP rule except when the current slope was of opposite sign to the previous slope. In such a case only the value computed by the QP rule was used.

QP can suffer from problems similar to those of BP; specifically the problem of flat spots. This can be trivially solved using the same methods applied to BP. From experimental evidence, only the learning rate, η, needs problem-specific tuning and the tuning needn't be too careful for reasonable results.

The disadvantage of QP is the extra storage space required. To store the slopes and weight differences of the previous epoch each require O(N) storage where *N* is the number of weights in the net.

3.3.2. Quickprop performance

Fahlman tested QP on a series of *tight* encoder problems[1] with the results as given in [Fah88]. The improvement in learning time was almost an order of magnitude over standard BP and almost a factor of four over the modified BP (using 0.1 added to the derivatives and a non-linear error function to overcome flat spots).

For the tight encoder problems 4-2-4, 8-3-8, 16-4-16, 32-5-32, 64-6-64, 128-7-128, 256-8-256, the learning time grew slower than logN where *N* was the number of patterns to be learnt. On a serial machine the actual clock time grows by a factor between N^2 and N^2logN. On a parallel machine, the time grows by a factor of between N and NlogN. It is not clear whether these results hold for larger or other problems.

1 An N-M-N encoder task is *tight* if $N = 2^M$.

Fahlman experimented on the 8-M-8 encoder problem to see how the number of hidden units (*M*) affected the training time. The time decreased monotonically with increasing *M*, even when *M* was much larger than 8. Previously it had been suggested that the learning time would slow down after some point — probably because more hidden units force the output units further into the flat spots. The performance of QP for this task isn't necessarily an indication of its scalability since BP could arguably also perform better given the solutions to the flat spots problem.

On the XOR benchmark[1] *Delta-Bar-Delta* [Tve91] required 250.4 epochs, standard BP 1538.9 epochs and QP 24.22 epochs to correctly learn the function. QP is faster than the standard and enhanced versions of BP as well as conjugate gradient methods [Jur]. Empirical evidence seems to indicate that QP scales well as the number of patterns to be learnt increases. However, this needs to be established for a larger variety of tasks. It also needs to be established how QP scales as a function of the number of weights to be able to compare it realistically to BP.

3.4. BACKPERCOLATION

Backpercolation[2] (Perc), introduced by Jurik [Jur], is an algorithm for training multi-layer perceptrons. Due to the proprietary nature of Perc only the general principles of the algorithm may be explained while the actual mathematics needs to be kept confidential. Perc assigns each unit its own activation error thereby assigning each unit its own error surface. Jurik claims that experimental evidence indicates that weight changes that reduce the local activation error permit "tunneling through" the global error surface which leads to faster convergence and a better probability of reaching an optimal minimum. BP, on the other hand, tries to decrease the global error by descending along gradients of the global error surface.

Perc claims to satisfy the following constraints:

- The training stability doesn't degrade with many hidden layers.
- Only local computations are required.
- The training algorithm doesn't add any units to the network.
- The weights converge quickly to attain arbitrary accurate output.

1 The task is to implement the exclusive-or logic function.

2 The Backpercolation algorithm is proprietary to Jurik Research, PO 2379 Aptos, CA 95001 USA. Non-profit research is encouraged. All commercial use requires a license from Jurik Research. Proprietary technical report available.

The third constraint, as will be seen, is in fact somewhat restricting and there are good reasons for allowing algorithms to change the network architecture by adding or removing weights or units.

Iteration, as used by Jurik, means the presentation of one pattern and the updating of the network parameters, including the weights. Throughout this section, the notation used will be identical to that used by Jurik.

The basic Perc algorithm:

STEP 0: Assign Random Values To The Initial Weights.

STEP 1: Present The Input And Desired Output Vector.

STEP 2: Determine A Global Error For The Given Signal.

STEP 3: Back-propagate The Error Gradient To All Units But Those Of The First Hidden Layer.

STEP 4: All Units, Other Than Input Units, Are Assigned Their Own Activation Error.

STEP 5: The Weights Are Adjusted.

STEP 6: The Error Magnification Parameter, λ, Is Adjusted.

STEP 7: If Learning Is Not Complete, Go To Step 1.

Steps 1, 2 and 3 are identical to those of BP. Note that Perc uses the hyperbolic tangent as the non-linear activation function. The bias term, which is equivalent to the thresholds used in BP, is implemented by adding another input unit which permanently has a value of one.

A high-level algorithm of Perc appears in appendix C.

Each unit is assigned its own activation error, ε, which indicates the amount to change the activation, μ, of the unit by. In general, $\varepsilon = \mu^{*}-\mu$ where μ^{*} is the target value of μ. The calculation of the activation error is accomplished by exploiting the duality of the inner product of the signal and weight vectors. Units may thus "request" a change to the incoming signal which is broadcast to the unit providing the signal. The request is interpreted by the originating unit as an output error. The signal can then be changed, at the originating unit, by changing one or more weights. This causes each weight to have its own error surface to descend.

3.4.1. Backpercolation mathematics

Define (as for BP):

$$\mu_k = \sum_j w_{kj}\varphi_j$$

where w_{kj} is the weight from unit j to unit k,
φ_j is the activation value of unit j (i.e. the signal from unit j) and
μ_k is the total incoming signal to unit k.

Assume an arbitrary unit has an internal activation error ε. For each unit impinging on this unit, the activation value (or signal), φ, of the impinging unit is multiplied by the weight of the connection between the two units, w, giving the inner product $\varphi.w$. The duality of the inner product suggests that steepest descent may be used to reduce ε in two ways:

$$\Delta w = -\partial\varepsilon/\partial w \text{ and } \Delta\varphi = -\partial\varepsilon/\partial\varphi$$

This indicates how much adjustment is needed in the weight or signal to reduce the error ε. To change φ, the request for the change is sent to the unit providing the signal. This occurs on a layer by layer basis starting with the output layer and proceeding towards the input layer which is similar to BP's backpropagation of the error.

All the requests to a unit to alter a signal are combined to form an optimal value which then determines the internal activation error of that unit. After all units have been assigned internal activation errors, the units weights can be adapted to reduce their own activation errors thus giving each unit the equivalent of its own error surface.

Thus two phases are involved in the backpercolation process; message creation (MCR) and message optimization (MOP). The output layer performs MOP to determine the internal activation error for these units. The output units then send requests to change the incoming signals to the last hidden layer (MCR) which then perform MOP on these request to determine an optimal internal activation error. This process continues until the input units have performed MOP.

Due to the proprietary nature of the algorithm, and by the conditions of the agreement undertaken, the details of MOP and MCR may not be disclosed. However, they are clearly given in [Jur].

3.4.2. Backpercolation features

Perc does not use a learning rate parameter as BP does, but uses instead a self-adjusting error magnification term, λ. The *Averaged Absolute Error* (AAE) is defined as the absolute value of the difference between the desired and actual output, averaged across all the training patterns in one epoch. λ is assigned an initial value, generally less than 1, and is adjusted whenever the AAE is below a threshold, typically 0.8. Again the details of this adjustment may not be disclosed but is performed so as to stop oscillations that may occur if λ is too large by reducing λ.

As was mentioned in chapter 2, the initial weights do influence the convergence rate of the network. Jurik introduced an *auto-kickstart* strategy: the weights are initially set to $\pm\varphi/N_\varphi$ where N_φ is the number of connections to the unit (the *fan-in* of the unit) and φ is a global parameter. Jurik suggests starting with $\varphi = 2$. Whenever overall weight decay is detected, φ is doubled, the weights are reset and training restarts. Jurik justifies this by noting that if φ is too small, all units of a Perc net will have values within a small region about 0. The whole net will have linear behaviour and might not be able to achieve a non-linear mapping. Therefore if a non-linear mapping is required to solve the problem, the weights will decay towards zero. Clearly φ should be increased in such cases.

By normalizing the input data the performance of Perc and BP seems to yield better performance [Jur]. This is accomplished by dividing the difference of the mean of each input component (over all patterns) and each input component by the standard deviation between each component and the mean.

3.4.3. Backpercolation performance

Jurik compared Perc's performance to the reported results of various algorithms using the parity, encoder, linear channel[1], multiplexer[2] and symmetry[3] benchmark tasks. For most of the tests Perc converged faster than the other algorithms and was equalled by Quickprop in

1 The N-N-N Linear Channel task is replicate any vector pattern of N continuous input values in the N output values. The single hidden layer has N units.

2 The input layer has $N+2^N$ units and only one output unit. The N input units represent a binary encoding of any of the remaining 2^N input units. The desired output of the net is the input of the designated input unit.

3 The net has 2N binary-valued input units and 1 output unit. The task is to indicate whether the first N inputs are identical, in reverse order, to the second N inputs.

the 8-3-8 encoder task and bettered by Quickprop in the 10-5-10 encoder task. From Jurik's results it seems that Quickprop and Perc have similar convergence rates.

It is not clear how much of Perc's superior performance is due to the approach of assigning each unit its own activation error:

- As was shown in chapter two, changing the sigmoid function to the range (-0.5, 0.5) instead of (0, 1) led to a large decrease in convergence times. Since Perc uses the hyperbolic tangent, which has a range (-1, 1), it seems obvious that an increase in the rate of convergence will result from this alone.
- Due to the normalization of the input patterns, it seems suspicious to compare Perc's results directly with other algorithms which don't apply this normalization to the patterns.
- In the Perc source code distributed by Jurik, the weights and errors are normalized throughout the feed-forward and updating process. What affect this has on the convergence rate is not discussed by Jurik.
- Perc uses on-line updating where the weights are adjusted after each pattern has been presented. Many of the algorithms Perc was compared to perform batch updating. There is empirical evidence that on-line updating leads to quicker convergence than batch updating [KSV89].
- As has been discussed by Fahlman [Fah88], there is no set of benchmarks that test all aspects of an algorithm and different researchers use different criteria to measure success and error. Therefore it is difficult to compare the results of reported tests unless the identical conditions (e.g. measures of success, error thresholds, initial weights) are used. No details are given on how failures are treated. Fahlman [Fah88] has detailed various alternate strategies when reporting learning times that included failures. It is not clear whether Perc succeeded in all the benchmarks or if the high convergence rate was balanced by a higher probability of failure.
- Possibly Perc's choice of initial weights, where each weight has one of two possible values, including the auto-kickstart feature, affects the rate of convergence.

It is instructive to compare the scalability of Perc to that of QP as determined by Fahlman [Fah88]. This was accomplished by using the encoder problem in two ways: by using a set of tight encoders (4-2-4, 8-3-8, 16-4-16, 32-5-32, 64-6-64, 128-7-128) and the encoder 8-M-8 with $M \in \{2, 3, 4, 5, 6, 8, 16\}$ and comparing the learning times required. Where results were available for Perc it was found that they differed, sometimes significantly, from those reported here.

This can be possibly explained by two factors: the tests reported here did not include the auto-kickstart method of reinitializing the weights during training and an initial learning rate (λ) of 0.2 as opposed to 1.0 was used. Using λ=2.0 delivered results that were more comparable to those reported by Jurik. It is not clear whether these reasons adequately explain the difference in performance. However, if they do, it would seem to indicate that Perc is dependent on the initial values of the parameters and is thus a *brittle* or *delicate* algorithm [Fel82]. Two different implementations of Perc were used in the tests, one of which is distributed by Jurik, and thus it seems unlikely that implementation details caused the difference in performance.

The task was judged to be complete when all output units over all patterns had activation values that were within some neighbourhood of the desired value. This neighbourhood was determined by the *error threshold.* Jurik suggests that an error threshold double that used for algorithms with an activation function in the range (0, 1) should be used when comparing Perc's performance to such algorithms due to the larger activation range (-1, 1) of the units in the Perc algorithm. Since Fahlman used an error threshold of 0.4, a threshold of 0.8 was used in the Perc tests.

	Perc(λ=0.2)			**Perc(λ=2.0)**			**QP**		
Size	Trial	Mean	S.D	Trial	Mean	S.D.	Trial	Mean	S.D.
4-2-4	10	33.8	8.08	10	6.2	2.53	100	15.93	6.2
8-3-8	10	78	40.09	10	23.1	5.69	100	21.99	5.6
16-4-16	10	100.8	30.44	10	77.2	24.02	100	28.93	5.4
32-5-32	5	187.4	19.35	5	316.4	106.2	25	30.48	3.1
64-6-64	5	440.8	31.16	5			10	33.90	2.9
128-7-128	**FAILED**			**FAILED**			5	40.20	2.8
256-8-256	**FAILED**			**FAILED**			5	42.00	1.6

Table 3-1 Comparison of the scalability of Perc (for λ=0.2 and λ=2.0) and QP for various tight encoder problems. The mean indicates the mean number of epochs required to satisfy the desired error criterion.

A trial was classified as a failure if the required error criterion was not satisfied within 750 epochs. Due to limited computational resources, less trials were conducted for the Perc scalability tests. The results of the two tests are detailed and compared to those reported in [Fah88] in tables 3-1 and 3-2 and graphically in figures 3-1 and 3-2. As can be seen, Perc scales worse than QP in both cases, especially in the first test.

Perc requires O(N) extra storage to store the internal activation errors of the N units. This compares favourably to QP's storage requirements. However, Perc requires two backward passes through the net when updating weights as opposed to QP's one. Since Perc uses on-line updating this could be a time consuming exercise, especially for large or deeply layered nets.

	Perc(λ=0.2)			Perc(λ=2.0)			QP		
Size	Trial	Mean	S.D	Trial	Mean	S.D.	Trial	Mean	S.D.
8-2-8	10	365.5	75.04	10	168.3	36.42	25	102.8	37.7
8-3-8	10	79.8	46.32	10	24.7	4.64	100	21.99	5.6
8-4-8	10	40.8	6.44	10	11.4	3.57	100	14.88	2.8
8-5-8	10	38.7	14.24	10	8.4	1.07	100	12.29	2.0
8-6-8	10	33.2	7.36	10	7.9	1.29	100	10.13	1.6
8-8-8	10	30	5.033	10	6.3	1.16	100	08.79	1.3
8-16-8	10	18.6	1.43	10	4.4	0.52	100	06.28	1.0

Table 3-2 Comparison of the scalability of Perc (for λ=0.2 and λ=2.0) and QP for the 8-M-8 encoder task.The mean indicates the mean number of epochs required to satisfy the desired error criterion.

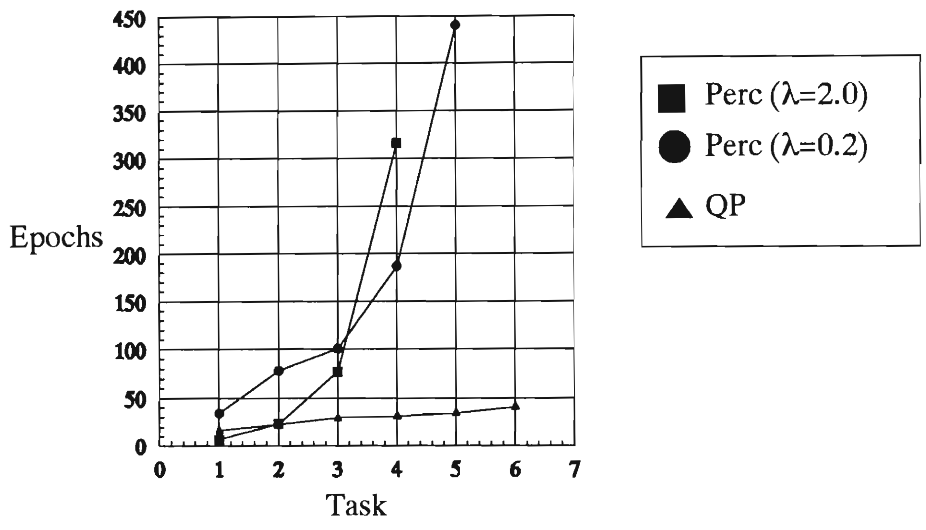


Figure 3-1 Graphical comparison of the scalabilty of Perc and QP detailed in table 3-1. *Task* indicates the encoder task (4-2-4, 8-3-8, 16-4-16, etc.) while *Epochs* indicates the number of epochs required to successfully learn the task.

A simple test using the 10-5-10 encoder was used to test the generalization ability of a Perc trained net. By reducing the number of different patterns used to train the net, the net's generalization ability could be tested. As Fahlman [Fah88] has noted, the encoder-decoder task actually punishes generalization since similar patterns are associated with different output patterns.

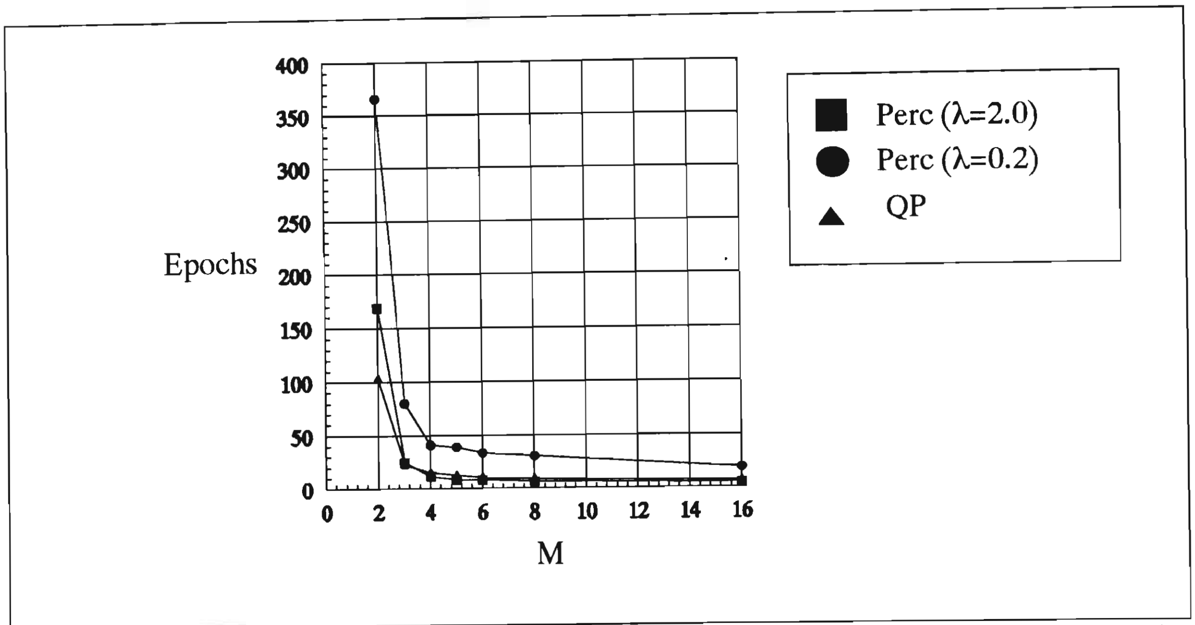


Figure 3-2 Graphical comparison of the scalabilty of Perc and QP detailed in table 3-.2. *M* indicates the number of units in the single hidden layerwhile *Epochs* indicates the number of epochs required to successfully learn the task.

Number of training patterns	Mean number of bits wrong	Mean number of patterns wrong
10	0	0
9	1	1
8	2	2
7	3	3
6	4	4
5	5.1	5
4	7.9	6
3	12.9	7
2	20.4	8
1	22.4	9

Table 3-3 Results of training the net with varying number of patterns to the test generalization ability of the net. Note that in the worst case (training with only 1 pattern) only 22.4 out of a possible 100 bits were incorrect.

Table 3-3 details the results achieved when the net was trained with a different number of patterns. In all cases, the patterns to be excluded from the training set for a specific trial were chosen at random from the ten available patterns. The results were compared using the number of patterns incorrect (out of a possible 10) and the total number of bits wrong (out of a possible

10.10=100) which were averaged over ten trials. The tests were repeated with nets trained using BP and QP and comparable results were achieved.

Finally tests were carried out to determine the robustness of the net to arbitrary damage to the weights in the net. Again the 10-5-10 encoder was used. A fully trained net was subjected to arbitrary amounts of damage and the resulting performance of the net was noted. The number of weights damaged ranged from 0 to 100% while the amount of damage ranged from 0 to 100% of their final values. Thus in the most severe case, all the weights were damaged by 100% of their final values; thus either becoming 0 or doubling in value. Results were averaged over ten trials each. Even when the all the weights were changed by 100%, on average only 25 bits were incorrect. Again nets trained using BP and QP were subjected to the same test with similar results.

3.5. CONCLUSION

This chapter dealt with two algorithms that seem to provide substantial improvement over BP. QP seems to scale well when dealing with the encoder task although further evidence is needed before extending this conclusion to other tasks. QP seems to have a similar rate of convergence to Perc. Although Perc promises a large decrease in the convergence time and a better probability of a global minimum being found, it is not clear how much of this is due to the assignment of local error surfaces as opposed to the various heuristics used in the algorithm. From the initial tests conducted on Perc, it seems that either the algorithm is "brittle" and its performance can be adversely affected by the choice of parameters or the auto-kickstart method plays a larger role in its "improved" performance than noted by Jurik. It would be interesting for the above tests to be repeated by Jurik so that his results could be compared to those obtained. From these results however, it seems that QP is faster than Perc (certainly for the encoder benchmark) and scales significantly better. Further investigation of Perc is necessary which would be accelerated by releasing the algorithm into the public domain. Given the improvement in learning times of these two algorithms, it seems well worth the extra storage space required.

CHAPTER FOUR

4.1. INTRODUCTION

The BP algorithm, and algorithms derived from it, are generally easy to understand and implement. Such algorithms have been used in solving a variety of problems that are often difficult to solve using more traditional approaches. However, BP-like algorithms do have certain shortcomings. Many of these were discussed in the previous two chapters with various possible solutions suggested. In this chapter it will be noted that the learning problem for this type of algorithm is *NP-hard* and the implications of this will be discussed. Finally generative algorithms that alter the net architecture while learning will be discussed.

4.2. NP-COMPLETENESS AND LEARNING

An algorithm is *efficient* if it is a polynomial in the size of the problem, *n*, or is bounded by a polynomial in *n*. A computational problem is *intractable* if there is no efficient algorithm for solving it. *NP-complete* problems belong to the class of problems for which it is not known whether they are tractable or intractable. NP-complete problems are essentially all the same: a efficient solution for solving one guarantees an efficient solution to solve all of them. Conversely, if one is intractable then all are [GJ79] [Meh84].

A *decision problem*[1], π, consists of all the instances of the problem, D_π, and a subset, $Y_\pi \subset D_\pi$, of all the yes-instances. The class of *NP* problems consists of all decision problems that can be solved by nondeterministic algorithms which consist of a *guessing* and *checking* stage. The guessing stage of the algorithm proposes a structure C(I) for an instance I of the problem and the checking stage determines efficiently (i.e. in polynomial time) whether I and C(I) yield a *yes* answer [GJ79].

The class of *P*, or deterministic, problems are those decision problems for which efficient algorithms exist. Clearly P$\subset$NP: if a determinstic algorithm exists for solving a problem, the algorithm can be viewed as a nondeterminstic algorithm with the *guessing* stage removed.

1 A decision problem is one whose answer is either yes or no.

The class of *NP-Complete* problems are all those NP problems Π with the property that all other problems in NP can be transformed polynomially[1] to Π. The SATisfiability problem [GJ79] was the first problem shown to be NP-Complete [Meh84][GJ79].

One of the aims of the research into ANN learning algorithms is to develop efficient learning algorithms. All the algorithms considered in the previous chapters are designed to train a net to associate pairs of input/output patterns (supervised learning). While many of the algorithms are successful when applied to small networks, it is important to be able to increase the problem size (i.e. size of the input/output patterns) to tackle more complex problems. BP scales badly as the input size increases [TJ88]. Some algorithms, like QP, seem to scale better but, as will be shown, this class of learning problem is NP-hard[2].

Judd [Jud87] defines the *loading* problem to be:

> A net architecture and task to be learnt are given. The loading process is the assignment of appropriate activation functions to every unit in the net so that the derived mapping includes the given task. If no such configuration exists the algorithm must report this fact.

As can be seen, this includes any algorithm that requires a fixed, feed-forward net architecture and the mapping of a set of input patterns to a set of output patterns. This includes all the algorithms discussed previously.

The loading problem is clearly a search process which Judd [Jud87] transformed to a decision problem. It was then shown that the resulting decision problem is NP-complete by a polynomial transformation from SAT [Jud90]. Because the decision problem is no more difficult than the search problem this implies that the loading problem is NP-complete. Evidence suggests that $P \neq NP$ (i.e. NP problems are intractable) and the best general result at present is [GJ79]:

> "If $\Pi \in NP$, then there exists a polynomial p such that Π can be solved by a deterministic algorithm having time complexity $O(2^{p(n)})$."

1 A problem Π_1 can be transformed polynomially to a problem Π_2 if there exists an efficient algorithm, f, that transforms every instance I of Π_1 to an instance $f(I)$ of Π_2 so that $I \in Y_{\Pi 1} \Leftrightarrow f(I) \in Y_{\Pi 2}$.

2 A problem Π is NP-hard if it is as hard as the SAT problem, i.e. SAT α Π[GJ79].

Proving a problem to be NP-Complete does not imply that the problem is unsolvable, merely that it is unlikely that an efficient algorithm exists. Therefore, trying to find efficient algorithms should be given a low priority and rather the following could be considered:

- Trying to obtain efficient algorithms for special cases of the general problem.
- Relax the conditions to generate a new problem that may be solved efficiently.
- Find efficient algorithms that obtain *nearly optimal* solutions.

Garey and Johnson [GJ79], for example, note that the time complexity measure is a *worst-case* measure. Therefore, a inefficient algorithm merely means that there is at least one instance of the problem that requires that much time. Most problem instances could require far less time than the stated limit. This seems to hold for several well-known algorithms such as the simplex algorithm for linear programming and branch-and-bound algorithms for the knapsack problem [GJ79].

4.3. IMPLICATIONS OF NP-COMPLETENESS

4.3.1. Negative implications

There are many response to this result that are inappropriate. Such responses include:

- The use of massive parallelism in the simulation and implementation of large nets. This tries to solve the exponential growth in the learning times by dividing it by a linear function and is clearly inadequate.
- The use of shallow nets, i.e. nets with only a few layers of adjustable weights. The proof of the theorem holds for nets with only two layers of weights. Although empirical evidence shows that BP works well with nets of a shallow depth, a possible explanation for this was given previously.
- The use of different learning rules. Since no assumptions about the learning rule used appears in the statement of the loading problem (other than the fact that the net architecture is not altered during learning) it can be seen that all supervised learning algorithms for feed-forward nets of a fixed architecture are NP-complete.
- The use of different unit activation functions. Judd [Jud87] [Jud90] has shown that the theorem holds for linearly separable functions and that for quasi-linear functions (such as the linear threshold or sigmoid functions) the loading problem is NP-hard.
- The architecture does not form part of the learning algorithm. This merely implies that the learning task is even harder for algorithms that have only local knowledge of the net architecture, rather than the global knowledge used in the proof of the theorem.

4.3.2. Positive implications

- The theorem is a statement about supervised learning in nets in general. There might thus exist a subclass of net architectures or algorithms, restricted by some design structure, that might always learn tasks or certain classes of tasks in polynomial time.
- The proof of the theorem only holds for non-recurrent nets and nets that don't alter their architecture during learning. Therefore using recurrent nets and allowing the architecture to be altered during learning could negate this result. Minsky suspects (pers comm) that all the theorems developed in "Perceptrons" [MP88] can be extended to feed-forward nets but do not hold for recurrent nets.

4.4. GENERATIVE ALGORITHMS

Generative algorithms allow the network architecture (the units and their interconnections) to be altered as a function of experience. This differs from the algorithms in previous chapters that only allow the adaption of the weights of the interconnections in a fixed architecture. The use of generative algorithms is motivated by [HU91]:

- The NP-Completeness proof doesn't hold for algorithms that change the net architecture while learning [Jud87] [Jud90].
- Rapid learning and the ability to adapt to a changing environment.
- Robustness when presented with noisy or incorrect data.
- The ability to create efficient internal representations of the environment in which the net operates.
- Incremental learning; i.e. allowing the net to modify internal representations when faced with a changing environment without the loss of previous learned information.
- No explicit net architecture needs to be given, with compact nets (nets having few extra units or weights) being constructed by the learning process.
- The choice of the number of hidden units to use when training a net using BP and similar rules is often an arbitrary choice. Using too many hidden units can lead to *overfitting* of the data and thus poor generalization [HU91] [RM86] while too few hidden units can lead to slow learning times or a failure to learn the task.
- Learning in biological nets involves the growth of new synapses [Guy81]. It is not known whether the brain adds new neurons while learning, but it seems plausible that previously underused and unused neurons are recruited during learning.

Honavar and Uhr [HU91] extended the definition of connectionist models to a *Generalized Connectionist Model* (GCN). A GCN is a graph of linked units with a particular topology, Γ, which can be partitioned into three sub-graphs: the *behaviour/act* (Γ_B), *evolve/learn* (Γ_λ) and *coordinate/control* ($\Gamma_{\mathbf{K}}$) sub-graphs. The nodes in the sub-graph compute different types of functions: **B** (behave/act), λ (evolve/learn) and **K** (coordinate/control). Therefore

$$GCN=\{\Gamma, \mathbf{B}, \lambda, \mathbf{K}\}.$$

Non-generative connectionist models are typically specified as [HU91]:

- $\Gamma_{\mathbf{B}}$, the sub-net that behaves, is left unspecified. A complete description of Γ is necessary to completely specify the connectionist model.
- The same activation function is typically used for all units.
- The changes in the weights are computed rather than the changes required in the sub-nets to bring about these changes.

Non-generative algorithms learn the patterns by modifying only the weights of the fixed net architecture. However, there are powerful alternatives [HU91]:

- Learning that modifies the activation functions of individual units.
- Learning by modification of the connectivity of the network by the addition or deletion of interconnections or units.
- Learning that modifies the learning structures, such as the actual learning rules themselves.
- Learning that modifies the control structures, such as modifying the controls that determine the type of learning rule to be used.

Throughout the remainder of this chapter, algorithms that modify the connectivity of the net will be discussed. An algorithm that adds new units as necessary can learn any function by constructing a look-up table. Yet such a look-up table would be an inefficient use of memory since each training example would have to be stored in a unique combination of units [MP88]. Also, the generalization ability of such a net would be minimal and restricted only to patterns stored in the table. This is discussed in more detail later in this chapter.

Honavar and Uhr [HU91] have noted that there are good reasons to restrict the *fan-in* (i.e. the number of weights to a unit) of the units. By restricting the fan-in, however, deeply layered nets are required to learn complex functions and, as noted in chapter two, BP and related algorithms perform poorly when training deeply layered nets, partly due to the dilution of the error signal.

4.5. REQUIREMENTS OF GENERATIVE ALGORITHMS

Generative algorithms require that various sub-tasks are performed. These sub-tasks should preferably be performed by local subsets of the net and include [HU91]:

- Deciding when to generate a new unit, as opposed to modifying a weight say.
- Deciding where to generate a new unit.
- Choosing the pattern to be encoded by the new unit.
- Choosing the connectivity of the new unit.
- Choosing a method to modify the weights.
- A method to evaluate the units and weights, including pruning or re-organization of the network.

These requirements are clearly discussed in [HU91] and will be briefly summarized.

4.5.1. Choosing when to generate a new unit

The choice of when to generate a new unit can be decided in a variety of ways, including:

- The generation of new units at predetermined or tunable rates, possibly as a function of the time required to learn the task.
- Error-driven generation which occurs when the feedback indicates that the net produced an incorrect output.
- Error-driven generation when the feedback indicates an incorrect output has been produced over a sufficiently long period.
- The probability of adding a unit decreases with the time taken to train the net.

4.5.2. Choosing where to generate a new unit

The choice of where to generate the new units is non-trivial. In layered nets, there is little guidance as to which layers should be preferred when generating units.

- The simplest case is to generate units at any layer in a multi-layer net whenever the criteria for generation is satisfied.
- An alternative is to fill the earlier layers first by
  - › Starting with the first layer, fill each layer to some pre-determined capacity before moving to a later layer.

› The probability of adding a unit to a layer decreases with the distance of the layer from the first layer.

4.5.3. Choosing the pattern to be encoded by the new unit

A variety of options exist including:

- Choosing a pattern at random. This is achieved by choosing the weight vector for the new unit at random, and thus encode a random pattern. Clearly the random weights can be modified using some learning rule such as BP.
- Choosing a pattern by crossover or mutation. The new unit's weights are chosen to be some mutation or crossover of the weights of other units involved in similar patterns. This is similar to the crossover and mutation processes used in genetic algorithms [Aus90] [Gol89].
- Choosing a pattern by extraction. The new unit has its weights chosen such that the new unit responds optimally to some specific sub-pattern of the input pattern.

4.5.4. Choosing the connectivity of the new unit

The choice of the method of establishing the output connectivity of the new unit determines the manner in which the new weights are initialized and modified. Some possibilities include:

- The new unit is connected to all the output units with the new connections being assigned random weights. Some weight modification rule, such as BP, is then used to alter the weights. This is the strategy used by the Cascade-correlation algorithm.
- The new unit is connected to a selected set of output units. This set can be determined by the feedback supplied by the training pattern that was used to generate the new unit.

4.5.5. Choosing a method to modify the weights

Any suitable weight modification rule is applicable. Honavar and Uhr [HU91] discuss a method of generalizing the extraction of sub-patterns which can form a method of weight modification.

4.5.6. A method to prune and re-organize the net

It is often necessary to *prune* units or weights that are not contributing to the network's performance. The estimation of the "usefulness" of a unit may be determined by its information-content (determined by the weights on its output connections) or by allowing competitive interaction between added units. It is important that pruning of the generated net be considered so as to make the net more compact and thus derive a near-optimally sized net. Smaller nets can introduce fault-tolerance and encourage generalisation by replacing single units with a set of units that encode the same information [HU91]. By reducing the number of units and weights, the time required to propagate signals through the net, during and after training, is similarly reduced.

Methods that alter the architecture of a net trained with BP have been suggested. Marchesi *et al.* [MOP90], for example, describe a method for removing connections during training. The method starts with a fully connected, recurrent net that is trained with a BP-like rule. A synapse/connection is removed whenever its *Percent Average Synaptic Activity* is less than some threshold. The threshold is adjusted according to some function of the number of epochs of training and this *threshold function* is problem dependent. This method differs from most generative algorithms in that the net starts as a fully connected net from which weights are then removed. Since the gross net topology (i.e. number of units and layers) must be predetermined when using this algorithm, this method still requires the user to determine the initial topology of the net before learning commences.

Brent [Bre91] developed an algorithm to construct a decision tree suitable to solve a given task from which an equivalent ANN can then be derived. The construction of the decision tree is a recursive process with no choice of the number of hidden layers or weights having to be made. This algorithm achieves results at least as good as a multi-layer net trained with BP while the training times are much faster. Note that on serial machines it is more efficient to use a decision tree rather than the equivalent ANN since the classification of an input vector relies only on tests on the path from the root of the tree to the leaf node corresponding to the region. This is invariably faster than propagating the signal through a net [Bre91].

4.6. GENERATION AND GENERALIZATION

4.6.1. Vapnik-Chervonenkis dimension and generalization

The work by Vapnik and Chervonenkis (1971) provides tools to determine whether the examples (training patterns) convey enough information for the net to accurately determine the desired input-output function. Abu-Mostafa [Abu89] discusses this in detail. A brief summary of the relevance of the Vapnik-Chervonenkis dimension (V-C dimension) to ANNs is in order.

Assume that a function f has to be determined from a set of examples. Initially a number of hypotheses, G, can be guessed. The task of the learning algorithm is to use the training data (the examples) to select a function $g \in G$ which has a "good" performance on the training data.

Depending on the characteristics of G, it can be predicted how the performance of the net on a set of test data will generalize. This generalization aspect is captured by the V-C dimension, d.

If the training set is sufficiently large with respect to d, generalization can be expected. In general, the more flexible or large G is, the larger the V-C dimension. For example, for feed-forward nets, d grows in proportion to the net size.

4.6.2. Generalization ability of generative algorithms

Standard circuit complexity arguments show that generative learning algorithms can solve, in **polynomial time**, any learning problem that can be solved in polynomial time by **any** algorithm [Bau89]. In this sense, these types of algorithm are universal learners that are capable of learning any learnable class of concepts.

Blummer *et al.* [Bau89] have given conditions (which involve the V-C dimension) which suffice to demonstrate that a learning algorithm that can grow representations in a large class of target functions will converge to yield good generalization.

The danger in the use of generative algorithms is that the algorithms might keep adding units until the training set is learnt sufficiently well. It is better to encourage the net to extract the prototype from the training patterns by restricting it in some way to keep the number of added units to a minimum. One method, which works well for Cascade-correlation, is to add a weight decay term which weakens the weights in the net thus encouraging the net to strengthen existing weights rather than add new units. Another method is to increase the severity of the criteria used to determine when to add new units.

A rule of thumb to encourage generalization, mentioned by Hinton [Hin89], is that the number of training patterns should exceed the number of weights in the net. Of course for a generative algorithm, it is difficult to know the number of weights that will eventually be required to solve the problem, unless the total number of weights is restricted in some manner.

4.7. THE CASCADE-CORRELATION ALGORITHM

The Cascade-Correlation algorithm (CCA) was developed by Fahlman and Lebiere [FL90] as a generative algorithm that would overcome some of BP's failings; specifically the slow learning rate and the need for arbitrary choices of interconnection strategy and the number of hidden units and layers. They identified two major problems with the BP algorithm that leads to slow learning: the *moving target* and the *step-size* problem.

4.7.1. The moving target problem

Units in the interior of the net, i.e. hidden units, try to evolve into feature detectors that will play a useful role in the net's overall computation. This task is, however, greatly complicated by the fact that all the hidden units are simultaneously changing via modification of their weights. Since the hidden units in a specific layer do not interact, but only have access to their inputs and the error propagated back from later layers, the problem as defined by the error being backpropagated is constantly changing. Therefore the units do not quickly assume a useful role in the net as feature detectors but go through many changes before settling down to a useful feature detector.

BP slows down dramatically as the number of hidden layers is increased. This is partly due to the dilution of the error signal but also because of the moving target problem. Jurik claims that Perc overcomes this moving target problem with the weights undergoing little unnecessary change [Jur] but the evidence presented is not conclusive.

Fahlman and Lebiere [FL90] note that the *herd effect* is a common manifestation of the moving target problem. Suppose that there are two separate computational tasks, A and B, that are to be performed by the hidden units. If A generates a larger error, all the hidden units will change their weights so as to reduce A's error. However, as soon as B's error is larger, the hidden units will then try to reduce B's error. This could cause task A to generate a larger error, especially if all the hidden units try to solve task B. This process will oscillate between trying to reduce A's or B's error as each becomes larger when the error of the other is reduced by weight modification. Eventually both errors will become sufficiently small, it is hoped, so that learning will cease. In most cases the *herd* of hidden units will split up and deal with both tasks, but this split may take a long time. A similar, but more complex, problem exists when more than two tasks are to be learnt.

A solution is to allow only a small set of units or weights to be altered for a certain task. CCA uses an extreme version of this by allowing only one hidden unit to change at any given time. Although this seems counter intuitive and would seem to slow learning, empirical evidence suggests that learning speed is actually increased. An additional advantage is that the algorithm creates new hidden units when required, and thus relieves the experimenter from having to define the net topology.

4.7.2. The step-size problem

This problem occurs because BP only computes the first order partial derivatives which do not carry any information about the curvature of the space being searched. Higher order derivatives are needed for this. Solutions to this problem have been extensively discussed in chapter three.

4.7.3. Cascade-Correlation algorithm

Cascade-Correlation combines two ideas: the *cascade architecture* and the learning algorithm. The cascade architecture begins with a set of input and output units which are determined by the problem and the representation of the input and output patterns used. All the input units are connected to every output unit with an addition *bias* input which implements the threshold function as discussed in chapter two. The output units may produce a linear sum of their inputs or some non-linear activation function may be applied to the weighted sum. When precise analog output values are desired, linear output units may be used, while for binary output units sigmoid units would be the better choice.

Hidden units are added to the net singularly. Each new unit is connected to all previous non-output units (the original input units and existing hidden units). The input weights of the added unit are "frozen" (i.e. are unmodifiable) when the unit is added and only its output weights are trained. Thus, in general, one new hidden layer is added to the net when adding a new unit. Powerful high-order features are created in this way, but it can lead to deeply layered nets with units having a high fan-in since each hidden layer generally contains only one unit.

Starting with a net with no hidden units (equivalent to a Perceptron), the weights are trained on the patterns of the training set using an appropriate rule such as the Delta rule (since there is no need to backpropagate through hidden units at this stage). Fahlman and Lebiere [FL90] use the QP rule for weight updating since it is equivalent to the Delta rule for a net with no hidden units but converges faster.

At some point the learning may reach an asymptote with no decrease in the error resulting after a certain number of training cycles. If the net's performance is satisfactory the training can stop. If not, there must be some further (residual) error that can be reduced by adding a new hidden unit to the net. The new unit is added to the net, its input weights "frozen" and its output weights, which are connected to all the output units of the net, are trained using a suitable weight update rule (QP normally). This is repeated until the error criterion is satisfied.

To add a new unit, a *candidate* unit is created having connections from all non-output units already existing in the net. The output of the candidate unit is not connected to the net. A number of passes through the training set are used to adjust the candidate unit's input weights, initially chosen as random values, with the goal being to maximize the sum, S, over all the output units *o* of the magnitude of the correlation[1] between V, the candidate unit's value, and E_o, the residual output error observed at unit o. S is defined as:

$$S = \sum_{o} \left| \sum_{p} (V_p - \overline{V})(E_{po} - \overline{E}_o) \right| \tag{1}$$

where *o* is the network output at which the error is measured and *p* is the training pattern. $\overline{V}$ and $\overline{E}_o$ are the values of V and E_o averaged over all the patterns.

To maximize S the partial derivative of S with respect to each candidate unit's incoming weight, $\partial S/\partial w_i$, is calculated. In a manner similar to the derivation of the BP rule, expanding and differentiating S gives:

$$\partial S/\partial w_i = \sum_{p,o} \sigma_o (E_{p,o} - \overline{E}_o) f'_p I_{i,p}$$

where σ_o is the sign of the correlation between the candidate's value and output *o*, f'_p is the derivative of the candidate's activation function with respect to the sum of its inputs for pattern p, and $I_{i,p}$ is the input to the candidate unit from unit *i* for pattern *p*.

Having computed $\partial S/\partial w_i$ for each incoming connection, S can be maximized by performing gradient descent. Again only one layer of weights is being trained. Once S ceases improving, the new candidate is installed in the net, its input weights are frozen and its outputs are connected to all the output units of the net and initialized with random weights.

The candidate unit only cares about the magnitude of its correlation with the error at a given output and not the sign of the correlation due to the absolute value in (1). Generally if a positive correlation with the error at a given unit is determined, a negative connection weight to that unit will be developed, and so will attempt to cancel some of the error. Similarly for a negative correlation.

1 Because the formula for S leaves out some normalization terms, S is a covariance rather a true correlation measure.

An obvious improvement is to use a set, or *pool*, of candidate units. Each unit is trained as above with the candidate with the highest correlation being added to the net as the new unit either after a certain number of cycles or when no further improvement in the correlations of all the candidates in the pool is noted. Note that there is no interaction between different units in the pool and all receive the same inputs and residual error feedback. Because of this, all the candidates must be trained in parallel.

Using a pool of candidates is useful in many ways:

- The risk of installing a candidate unit that got stuck during training is reduced.
- Because many parts of weight space can be simultaneously searched, due to the random assignments of initial weights to the candidates in the pool, learning speed may be increased.
- There is no need to limit the candidate units to the same activation function. A mixture of non-linear activation functions, such as sigmoidal or Gaussian functions, can be used by the different units in the candidate pool. This mixture of activation functions in the net may lead to more compact or efficient solutions being found.

4.8. CRITIQUE OF CCA

4.8.1. Advantages

- Cascade-Correlation learns fast. Numerous benchmarks performed by Fahlman and Lebiere and other researchers indicate its improved performance. For the problems investigated by Fahlman and Lebiere, CCA's learning time grew roughly as NlogN, where N was the number of hidden units in the successful net.
- Only one layer of weights is trained at any time. Therefore a simple rule, like the Delta rule, will suffice for weight updating. This is one of the factors leading to the faster learning times.
- Error signals are never propagated backwards through various layers of units. Signals are only propagated in one direction through connections which is biologically more plausible than BP.
- The candidate units do not interact with each other except when a winner is selected. This makes the algorithm attractive for parallel implementation.
- Incremental learning is supported since once a feature detector (hidden unit) is created, its input weights are never changed and thus the features it detects aren't altered as new patterns are learnt.
- It is not necessary to guess the size, depth and connectivity of the net in advance.
- A net consisting of a mixture of units with different non-linear activation functions may be constructed.

4.8.2. Disadvantages

- Hardware implementation of the algorithm seems complicated since there is no upper limit to how large the net may grow. Similarly in a changing environment, many units may be added leading to a slowing down of the net when simulated by software.
- Fahlman (pers comm) has noted that CCA works well for continuous input/binary output problems but that the generalization ability decreases with continuous output problems, with the training patterns being learnt point by point. This seems to be due to the correlation measure that CCA tries to maximize: the units are encouraged to overshoot the residual error rather than match the error precisely. This is not serious when binary output values are required but is not the correct scheme for continuous valued output. Crowder (pers comm) has reported that CCA seems to yield poorer generalization than BP. However, by adding a weight decay as detailed by Krogh and Hertz [KH92], the generalization ability of CCA can be improved (Fahlman — pers comm).
- CCA requires a number of parameters to be specified:
  - › Learning rate.
  - › Maximum growth factor.
  - › Weight range of the random initial weights.
  - › Weight-decay to avoid possible floating-point overflow in computer simulations.
  - › The size of the candidate pool.
  - › The amount of training of the input and output weights of candidate units before adding a new unit from the pool of candidates to the net (the *patience* parameters) and the amount of training of the weights in the net before deciding when to generate a new unit.

The first four parameters determine the performance of the QP weight update rule while the rest determine the rate at which new units are added to the net. Yang and Honavar [YH91] have investigated the effects of the various parameters on the performance of CCA for four tasks[1] and have suggested the following heuristics:

- For the QP parameters, the use of larger maximum growth factors generally gave better results as did the use of smaller weight ranges, the use of the hyperbolic arctangent error function and the addition of a constant to the derivative of the sigmoid function to overcome the flat spots.

1 Classification of king-rook vs. king-pawn endgames in chess in win/no win, classification of audiology data, classification of soybean diseases and classification of iris plants.

- Large candidate pool sizes generally gave better results (since this allowed a large area of the weight space to be searched) while the patience parameters had little effect over the ranges that were examined.

However, they do caution that additional studies are required since the interaction of the various parameters can be complex.

4.9. ENHANCEMENTS TO THE ALGORITHM

Yang and Honavar [YH91] have investigated several enhancements to CCA using the tasks mentioned previously. These include the use of two different error metrics, different candidate unit selection strategies and the effect of allowing the input weights of candidate units, once added to the net, to be modified.

Error metric I used the magnitude of the error between the desired and actual outputs to determined whether a pattern was correctly classified while error metric II classified a pattern as correct if the output unit with the largest activation value corresponded to the correct category. The new unit was chosen from the candidate pool in one of two ways: the unit with the highest correlation with the residual error as in CCA (method I) or at random (method II). The results on the chess, audiology and iris data sets had almost the same characteristics regardless of the choice made. The number of epochs required was larger when using error metric I on all data sets other than the soybean data. Using method I to select the new unit from the pool of candidates resulted in a significantly smaller number of epochs to achieve success: CCA constructed nets with 1.1±0.3, 0.3±0.5, 0 and 0.5±0.5 hidden units for the chess, audiology, soybean and iris data sets respectively. As can be seen, the decision surfaces for these problems are relatively simple (due to the small number of hidden units required) and sweeping conclusions should thus not be drawn. It seems obvious, however, that selecting a unit by random from the candidate pool would not give faster learning times in general.

CCA only modifies the output weights of a new unit once it has been added to the net. Yang and Honavar [YH91] studied the effects of modifying the input weights in two ways. Method I trained input and output weights of new units while method II unfroze the input weights of all the hidden units once some proportion of the patterns (95% in their study) were correctly learnt. The number of training epochs decreased when input weights were allowed to be modified but the performance of the net (the classification accuracy) was found to be worse on the audiology and chess data sets. Note that the amount of operations per epoch is greater since the input weights also need to be modified and that the Delta rule will not suffice for weight updating. Method II would seem to increase the performance of the net while there is a danger that method I could reintroduce the herd effect.

4.10. CONCLUSION

Learning an arbitrary task in a feed-forward, non-recurrent net is a NP-complete task. This result might not hold for recurrent nets or algorithms that modify the net topology as learning progresses. Generative algorithms alter the architecture of the net as a function of experience and have many advantageous properties. The Cascade-Correlation algorithm is an example of a generative algorithm that relieves the user from making a choice of the number of hidden units and layers and is faster than most of the algorithms of the previous chapters. Because only one layer of weights is being modified at any stage, any suitable weight updating scheme may be used, including the Delta rule, BP, QP or Perc. CCA seems to be robust in that the choice of the parameters values does not affect the rate of learning too greatly. CCA constructs a compact, though not necessarily optimal, sized net. However, it is not clear whether the algorithm can be easily implemented in hardware and deep nets with a large fan-in may be constructed by the algorithm.

CHAPTER FIVE

5.1. INTRODUCTION

In this chapter the results of applying ANNs to four tasks will be discussed. The four tasks were the calculation of decompression stops when SCUBA diving, decision making in a simple environment, composition of music and binarization of images. The four tasks were selected to demonstrate and test different features of ANNs. All of the ANNs were trained using the Cascade-Correlation algorithm (CCA) described in chapter four. CCA was considered the best algorithm to use since it is a generative algorithm, thus relieving the user of determining the best network structure, and it learns faster than BP, QP and Perc, the other learning algorithms considered.

5.2. CASCADE-CORRELATION PARAMETERS

CCA has many parameters that are involved in the learning process. It was found, however, that many of the parameters could be set at certain values, regardless of the task, and that this did not affect the performance significantly. This is similar to what Yang and Honavar [YH91] found in their study of CCA. The most important factor in these tasks was to train a net to successfully accomplish the task, with the speed of learning a secondary consideration. Due to the limited computational facilities available, most of the tasks were limited to only thousands of epochs of training before being considered a failure. However, for most of the tasks, CCA learnt the required tasks in less than 1 000 epochs.

The parameters found to be of importance were:

- The amount of decay the output weights underwent. As Fahlman has noted [Thr91] it is sometimes useful to increase the amount of decay of the output weights which encourages the formation of more hidden units than is normally required. This is especially useful when small training sets are used. It was found that increasing the output decay from 0 to 0.1 always caused more hidden units to be generated which sometimes resulted in increased learning times. However, the resulting nets always performed better for those tasks with small training sets.
- The "patience" parameters which determine how long the candidate units input and output weights are trained before the unit is added to the net. It was found that the patience parameters had little affect on the constructed net, with only a few extra hidden units being added to the net and no significant change in the learning speed.

- The type of error measure used to determine when training should cease. Two different error measures were used:
  - › The sum-of-squares error criterion averaged the sum of the squares of the differences between the actual and desired outputs over all the patterns and judged learning to be complete when this average was smaller than some threshold. This error criterion can encourage generalization since smaller errors produced from some of the training patterns can be traded off against larger errors from other patterns since the errors from all the patterns need not all satisfy the error cirterion [Fah88]. Generally this method achieved nets with better generalization capabilities.
  - › The number of bits that were incorrect were totalled and training was complete when no bits were incorrect. This method can only be used for binary output vectors, as was the case for most of the tasks. A bit was considered correct if the absolute difference between the desired and actual output vector's component was less than some threshold.

5.3. CALCULATION OF DECOMPRESSION STOPS

5.3.1. Introduction

When SCUBA diving, nitrogen (N_2) is absorbed by the fat and muscle tissue from the blood stream. On ascent, N_2 flows out of the tissues into the blood and then to the lungs where it is exhaled. Because the blood supply is small compared to the amount of tissue in the body, only small amounts of N_2 will diffuse from the tissues to the bloodstream and be exhaled in any period. Any excess N_2 in the tissues will form bubbles and pain, and often death, can result [Cou79] [SAU].

This absorption of N_2 usually only occurs when diving to deep depths for prolonged periods or when repetitive dives are undertaken, increasing the total amount of time spent submerged. To avoid the formation of bubbles, certain prescribed rates of ascent with strict "decompress-ion stops" must be followed which allows the N_2 to be exhaled normally without any build-up in the tissues [SAU].

5.3.2. Aim of task

Table 5-1 is the section of the decompression table used for the task. The use of the table is not important for understanding the task the net was required to learn. The task was to learn the table, which can be trivially solved by an ANN by constructing a look-up table containing the relevant details. However, this would be an inefficient use of memory [MP88] with faster access times resulting from traditional storage methods such as a two dimensional array.

Depth (m)	Duration until commencement of ascent (min)							
	8	9	10	11	14	15	17	20
27	0	0	0	0	0	0	0	0
30	0	0	0	0	0	0	0	0
33	0	0	0	0	0	0	0	1
36	0	0	0	0	0	1	1	1
39	0	0	0	0	1	1	3	3
42	0	0	1	3	3	3	4	4
45	0	1	1	3	3	3	5	5

Table 5-1 The section of the decompression table used. The digits in the table indicate different decompression stops.

Therefore, to test the generalization ability of the ANN, only a subset of the available data was used to train the net. It seems intuitively obvious that the generalization ability of the net should increase if the training data contained the boundary cases of the classes to be learnt rather than just arbitrary points in the table. It was hoped that the net would extract certain simple rules when learning the required mapping. Such rules could include, for example, the fact that no class number is ever less than the class number to its left.

5.3.3. Training sets and performance

Three data sets were used to test the generalization ability of the net. In each case the net was constructed starting with the Perceptron architecture, thus not utilizing the incremental learning capability of CCA. This was done so as to compare learning times given different sets of starting weights and to prevent a possibly inferior solution due to local minima that might have been achieved by previously trained nets.

In all cases the sum-of-squares error measure with a threshold of 0.35 was used to judge when training was complete. For all the following tables *1* indicates that the data in table 5-1 at the corresponding position was part of the data set while *0* indicates that that data point was excluded from the training set.

A *unary* encoding system was used to represent the input and output data: each output class was represented by a unique output unit and similarly each component of the input vector was represented by a unique unit. The output unit with the largest activation value was assumed to be the activated unit (i.e. the corresponding component of the output vector had a value of *1*) with all other output units assumed to have no activation (i.e. the corresponding component of the output vector had a value of *0*).

5.3.3.1. Stage 1

Table 5-2 details the table entries making up the first data set. Only 23 patterns were used with an entry forming part of the training set only if it indicated the beginning of a new class when moving to the right in a certain row of the table. It was hoped that the net would learn the boundaries of the classes and possibly interpolate the classes between the boundaries correctly.

Depth (m)	**Duration until commencement of ascent (min)**							
	8	9	10	11	14	15	17	20
27	1	0	0	0	0	0	0	1
30	1	0	0	0	0	0	0	1
33	1	0	0	0	0	0	0	1
36	1	0	0	0	0	1	0	0
39	1	0	0	0	1	0	1	0
42	1	0	1	1	0	0	1	0
45	1	1	0	1	0	0	1	0

Table 5-2 First data set

With the output weight decay (*output decay*) parameter set to 0.1, two hidden units were added to the net which required 778 epochs to learn the training data. The net correctly learnt the 23 training patterns but 35 errors resulted when tested on the unseen data. The net usually responded with "class 1" in these error cases. Certain "patterns" were noted in the errors:

Depth (m)	**Time (min)**	**Response (class)**
27-39	9-10	1
27-39	11	3
27-39	14	1

It has been noted previously that the training set should contain O(n) patterns, where *n* is the number of weights in the final net. Since the table contains a total of 64 data points, and the simplest ANN (the Perceptron) would contain approximately 64 weights, it seems reasonable to assume better performance would result with a larger training set.

Retraining the net with the same data and parameters but setting the patience parameters to 100 epochs, the constructed net had two hidden units and learnt the task in 170 epochs. Thirty-five errors resulted with three of these being from the training data and 32 errors from unseen data. The errors on the unseen data were exactly the same as those given above.

5.3.3.2. Stage 2

By increasing the training data set to 38 patterns (table 5-3) the following occurred:

- When using the sum-of-squares error measure, an output decay parameter of 0.1 and a patience parameter of 500 epochs, 341 epochs were required to learn the task with three hidden units being added to the net. Eleven errors resulted when tested with the unseen data. Again a pattern in the errors was noticed:

Depth (m)	**Time (min)**	**Response (class)**
33-39	9-10	5

- Using the number of incorrect bits as the error measure (with a threshold value of 0.2) and the other parameters as above, 409 epochs and 4 hidden units were required. Twelve errors resulted, all from unseen data, with the following pattern emerging:

Depth (m)	**Time (min)**	**Response (class)**
27-39	10	5

5.3.3.3. Stage 3

Finally the training set was increased to include all the entries appearing in table 5-4. Specific entries were included to alleviate the errors that occurred in the other tests. For example, for depths of 27 to 36 metres and times of 9 to 11 minutes, the previous nets had consistently misclassified the input. Various choices were made with respect to the value of parameters, with varying results:

Patience	Output decay	Error measure	Epochs[1]	Hidden units	Errors[2]
500	0.1	s-s[3]	283	3	5
250	0.1	bits[4]	420	5	9
250	0.01	s-s	338	4	8

Results of testing the net trained using training set 3.

1. The number of epochs required to learn the task.
2. The total number of test and training patterns incorrectly classified.
3. The sum of the squares of the differences between the desired and actual output vectors had to be less than 0.2.
4. The number of bits (within 0.2 of their desired value) had to be 0 before training was judged to be complete.

Depth (m)	**Duration until commencement of ascent (min)**							
	8	9	10	11	14	15	17	20
27	1	0	0	0	0	0	0	1
30	1	0	0	0	0	0	0	1
33	1	0	0	0	0	0	1	1
36	1	0	0	0	1	1	0	1
39	1	0	0	1	1	1	1	1
42	1	1	1	1	0	1	1	1
45	1	1	1	1	0	1	1	1

Table 5-3 Second data set

Depth (m)	Duration until commencement of ascent (min)							
	8	9	10	11	14	15	17	20
27	1	1	0	1	1	0	0	1
30	1	0	1	0	1	0	1	1
33	1	0	1	0	1	0	1	1
36	1	1	0	1	1	1	0	1
39	1	0	0	1	1	1	1	1
42	1	1	1	1	0	1	1	1
45	1	1	1	1	0	1	1	1

Table 5-4 Third data set

5.3.4. Conclusion

This task seems to demonstrate quite clearly the importance of a well chosen training set. It is not sufficient to use an arbitrary collection of data points. Including the boundary cases in the data set is important since the decision regions to be formed by the hyperplanes are then better defined. Encouraging the addition of hidden units to the net constructed by CCA, via the output decay parameter, resulted in a better generalization capability while using the *global* sum-squares error measure seemed to increase the generalization as well. The reason for the improved generalization when using this error measure has been explained previously.

In all the tests, it was found that many of the parameters required by CCA are problem independent or, at worst, require a little "tuning" for acceptable performance. This agrees with the findings of Yang and Honavar [YH91]. Although this task can be trivially solved using traditional methods, it can be seen that ANNs often generalize from trained to unseen data acceptably.

5.4. BOB

5.4.1. Introduction

Attempts to use ANNs in robotics has, until recently, concentrated on *low level* tasks such as vision and appendage movement. It is not clear whether ANNs can perform *higher level* tasks, such as goal satisfaction, adequately or better than traditional methods. Because of the parallelism involved in ANNs, they would seem suited to tasks requiring a decision to be taken given many simultaneous inputs. Dayhoff [Day90] described “artificial critters” which are “creatures” (ANNs) that are designed to perform some simple task, such as surviving in a simple environment or moving away from threatening situations. *BOB*1 is an artificial creature which tries to survive in a simplified environment where BOB’s decisions are supplied by an ANN given the current stimuli from the environment as input.

5.4.2. Previous approaches and BOB

Arbib [Arb87] argues that the solution to complex tasks should be broken into functional units, such as schemas. Neural nets as structural units intermediate between structures subserving schemas and small neural circuits can then be used. Myers [Mye91] simulated a food-finding creature to demonstrate learning with delayed reinforcement while Nicole [Nic90] modeled the defensive reflex of Aplysia using a hand-crafted net that underwent no training.

Sutton [Sut90] describes three methods for a robot to decide which action to follow:

- By planning. The best action is deduced by considering the current goals and the robot’s world model. This is limited by the computational complexity and the accuracy of the world model.
- By reacting. The planning is completed in advance and the results are stored as a set of situation-action rules.
- By learning a set of good reactions by trial and error.

Sutton combined all three methods in his *Dyna* model with promising results.

Bechtel and Abrahamsen [BA91] have noted that

> “Suppes, for example, offered a proof that for any finite automaton there is a stimulus-response model that converges to the automaton. Hence, any model having rules, plans or other higher-order entities can be reduced to a computationally equivalent stimulus-response model”.

Building on Sutton and Bechtel’s results, BOB is a neural network that reacts to the current situation. The reactions are pre-computed by training the net with an appropriately constructed training set. The important difference between BOB and the other approaches is that identical

1 BOB will be referred to as a male for no particular reason.

units and layers make up the net that controls BOB, as opposed to Arbib's approach of using a variety of nets and units for control, and the training set was constructed so as to allow BOB to extract relevant rules and thus be able to react to unseen situations.

5.4.3. BOB's environment, actions and rules

The simplified environment consisted of three stimuli: light, prey and obstacles. No fixed geography was imposed on the environment with BOB only able to detect the stimuli at his current position. The input to BOB was thus supplied by three sensors:

- A light sensor which indicated the strength of the light directly in front of BOB.
- An object sensor which determined the distance of any obstacle directly in front of BOB.
- A prey sensor which calculated the distance between BOB and any edible goodies directly in front of him.

The sensors all returned values in the range [0,1] which indicated the strength of the appropriate signal, with 0 being the weakest and 1 being the strongest. How the responses are generated or the type of preprocessing required is not considered important since the object was to simulated goal planning rather than the low level processing.

BOB has three possible actions:

- Move forward.
- Turn left.
- Turn right.

These actions can be combined, so that BOB can turn left and then move forward, accomplishing a diagonal movement. BOB has no memory of previous actions or states, as well as having no internal measures such as an indication of hunger. The action to be taken is represented by a binary valued output vector of three components. As in the previous task, the output unit with the largest activation value was assumed to be the active unit with the other units being assigned a value of 0.

BOB's behaviour is governed by a few simple rules:

- Move towards the strongest light source.
- Move away from objects.
- Move towards prey.

When the rules are in conflict, such as moving towards a light source and away from an object, the following apply:

- BOB prefers moving towards prey. This is true even in the presence of a strong conflicting light stimulus.
- If there is no light or prey stimulus or BOB gets too close to an object, he turns. BOB turns left if there is no light stimulus or he has encountered an object, otherwise he turns right.
- BOB does not actively back away from objects. Only when he gets too close does he try to find a better direction to take.

As can be seen, the environment and BOB's possible actions are quite limited. It was felt, however, that these would be sufficient to simulate various different situations and allow enough possible reactions without a large computational overhead.

5.4.4. Results

CCA was used to train the net. It was found that the Perceptron architecture was inadequate and hidden units were always required. The number of hidden units, and thus hidden layers, required ranged from five to eight. The training data set consisted of 36 patterns of three input and three output values. A comparison of the effect the the two error criteria, used in the previous task, on training times and performance was made.

It was found that the net required longer training times using the second error criterion (the total number of bits wrong had to be zero) and training could often only be suspended by increasing the threshold. The first error criterion did, in fact, cause the net to generalize better with better reactions by BOB.

BOB performed as was expected given the above rules. What is significant is that the rules weren't explicitly encoded into the net's representation. Thus, the net generalized from the training set to extract the prototypes and thus its own rules. This is normally possible with a suitably designed training set. This approach can be a significant improvement over *hard-wiring* the rules in a formal system such as an expert system or programmed solution since the rules don't have to be explicitly enumerated and only suitable examples need to be given as training data. This is useful in tasks in which rules are not available or when the rules are too numerous or unclear to enumerate. Of course, care must be taken to include representative examples of many situations in the training data.

5.4.5. Improvements

BOB's overall performance was adequate. However, because BOB had no memory of previous states or actions or any internal signals, his behaviour was occasionally stupid. For example, he would always move towards available prey which is not something normally seen in natural creatures. Adding an internal measurement of his hunger state could possibly alleviate this problem.

Because of the lack of memory, BOB would often overshoot his target by turning away from the best direction briefly while still moving forward and thus move away from the desired target (fig 5-1). Adding memory of previous states would alleviate this problem. A simpler solution would be to increase the viewing field size of the detectors. Thus, instead of only detecting a light source, say, directly in front, BOB would be able to see the light intensity across a larger viewing field. Whether this would be sufficient remains to be seen but it is suspected that some sort of memory of previous actions or states would still be required.

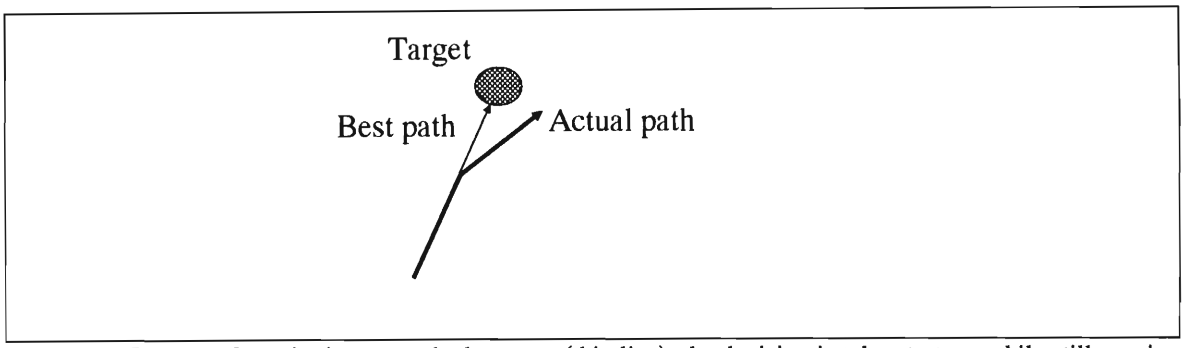


Figure 5-1 Instead of continuing towards the target (thin line), the decision is taken to turn while still moving forward (thick line). This causes BOB to follow the path away from the target.

5.4.6. Conclusions

ANNs can be used for higher level process such as goal satisfaction. Although the task was to determine whether BOB would formalize the rules inherent in the training set, it does not seem that this approach would be appropriate for all circumstances. For a simple environment such as that faced by BOB, a programmed solution or expert system would undoubtable perform at least as well with less time required to implement a working system. However, for environments with many input variables, or when the reactions rules are ill-defined, this method could have numerous advantages over traditional methods.

The training data must be chosen carefully. It is not sufficient to merely give as much training data as possible; it is important to include the boundary cases in the training set since this allows the net to generate the appropriate hyperplanes needed to separate the decision space correctly.

Alternative methods to consider when trying to simulate natural movement or decision making in robotic applications are those proposed by the *Artificial Life* community where biological creatures' decision making or movement are modeled by simple methods such as cellular automata with the required behaviour emerging [Lan88] [Lai88].

Brooks' approach [Bro91], using the subsumption architecture, to intelligent behaviour deserves mention. No explicit world model is constructed, but the creature interacts directly with the environment by perception and action. This is similar to the approach used by BOB where the inputs to the net, the perception of the world, are acted upon to produce some action by BOB in the environment. There are important differences, however, between Brooks' approach and connectionism [Bro91].

5.5. MUSIC COMPOSITION

5.5.1. Introduction

ANNs have been applied to a variety of musical related tasks, including determining the optimal fingering for stringed instruments and tonal analysis [ED90]. A commercial musical package has been released by NEC which uses neural networks to create harmonies for a user supplied musical composition [Gea91].

The automatic composition of music has been explored using traditional approaches such as rule-based systems [Cha80] [Jea68]. When using a rule-based system, determining the rules can be time consuming and non-trivial. It is also not clear if there are rules that determine when music sounds pleasant. Part of the beauty of music is the occasional randomness often exhibited; the music doesn't always "follow the rules" but follows occasional unexpected directions.

An alternative approach to the rule-based system would be to present an ANN with examples of the style of music which is desired and, hopefully, the net will generalize from the examples and compose similarly styled music. Because rules won't have been explicitly encoded, a degree of randomness might result in the output of the net and thus of the composed music.

5.5.2. ANNs and temporal sequences

When learning a temporal sequence, it seems that the net should have some memory of previous states, which would enable it to generate the appropriate response given its current input and past history. To incorporate such memory requires either a recurrent architecture or the inclusion of time delays in mutlilayer perceptrons. Eberhart and Dobbin [ED90] discuss various recurrent net strategies, which include recurrent connections from output to input units and from input units to themselves, that have been employed when learning temporal sequences while Lippmann [Lip89] discusses time delay multilayer perceptrons and recurrent nets as applied to speech processing. Backpropagation has been extended to deal with the training of recurrent nets by Almeida [Alm87] amongst others.

Since non-recurrent feed-forward nets were being studied, it was decided to use a simple strategy for providing the net with some sort of context in which the musical notes appeared. The strategy is similar to that employed by Eberhart and Dobbin [ED90] and by Sejnowksi and Rosenberg in their NETTalk experiment [SR86]. Consider the notes of the desired training song to be a linear list. A *window* of fixed size is slid along the note list with the notes appearing in the window serving as the input to the net. The note to the immediate right of the window (i.e. the note that will appear in the window next) is taken to be the desired output of the net. Figure 5-2 illustrates the process. NETTalk used a similar system where the window contained a sequence of letters and the net was trained to pronounce the letter in the center of the window as the window was passed across the entire training text.

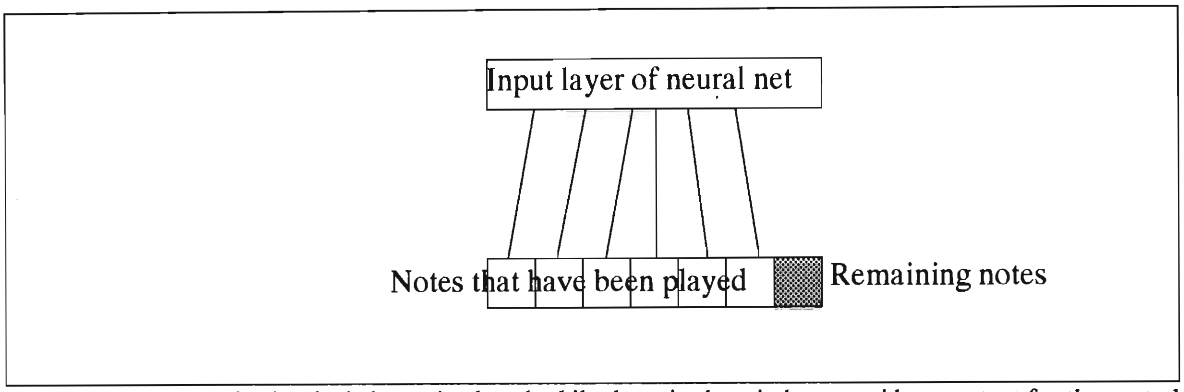


Figure 5-2 The note in the shaded area is played while those in the window provide a context for the central note.

A decision has to be made on the size of the window used. The term *n notes deep* will denote a window that encompasses *n* notes; i.e. the input vector will contain *n* notes. The larger *n*, the more complex the net will be with a resulting decrease in learning speed. However, having more notes in the input vector will provide more context to the note and should lead to less variability in the music. It was found that training three to five notes deep generated a net which composed the more pleasing music which is in agreement with Eberhart and Dobbin's results [ED90].

5.5.3. Musical representation and training data

Two ways to represent the musical training data input to the net were considered:

- The notes can be presented as transitions. Each unit represents one magnitude of the transition between one note and the next. The magnitude can, for example, be measured in musical half-steps. For example, the note sequence A B C would be represented as A +2 +1: B is two half-steps (A A# B) above A and C is one half-step (B C) above B. This method has the advantage that a large number of notes can be represented with relatively few units. The biggest disadvantage, however, is that any drift will become magnified; i.e. if a wrong note is produced, all the following notes would be transposed.

- Each unit represents a different note. So, for the sequence A B C, units representing these notes would be activated. This has the disadvantage that more units are required in the input layer, but alleviates the transposed notes problem.

The second method was used because of its inherent robustness.

The output units can be considered to represent one of two cases:

- Time slices where each iteration of the net represents a time slice. An extra *note-begin* unit would be required to indicate whether a note starts in the current time slice or is a continuation of the note in the previous time-slice. However, a decision then has to be made of how to handle a note, in the current time-slice, of a different pitch than the note in the previous time slice when the note-begin unit remains inactivated.
- Individual notes where each output unit represents a different note. Extra output units are required to represent the duration of the output notes.

Although the second method requires a larger number of output units, it is the more robust of the two and was thus used. Due to the representation of the notes the input and output vectors are clearly binary valued vectors.

Three tunes of a similar style were used to train the net: “Oh Susanna”, “She’ll be comin’ ’round the mountain” and “Yellow rose of Texas”. These tunes were chosen due to the simplicity of the music, the availability of the tunes and the fact that Eberhart and Dobbin [ED90] had used these tunes in their experiments and the composed music could thus be compared to the music produced by their trained ANN. The three tunes were combined into one training set (in the order “Oh Susanna”, “She’ll be comin’ ’round the mountain” and “Yellow rose of Texas”) as done by Eberhart and Dobbin. By using tunes of a similar style it was hoped that the trained net would compose music similar in style to the training tunes. Using tunes from different styles could result in a mixture of styles (a type of “cross-over” style) or a very unpleasant noise!

5.5.4. Composing music with a trained net

Once the net has been trained to a satisfactory degree of accuracy, it may then be used to compose music. This is achieved by seeding the net with the first *n* notes (where the net has been trained *n* notes deep). The output produced by the net can then be copied back to serve as input for the next note to be generated. Clearly the input vector must be updated in such a way that the first note in the vector is discarded, all the other notes are shifted and the generated note is appended to the input vector. Figure 5-3 illustrates this process.

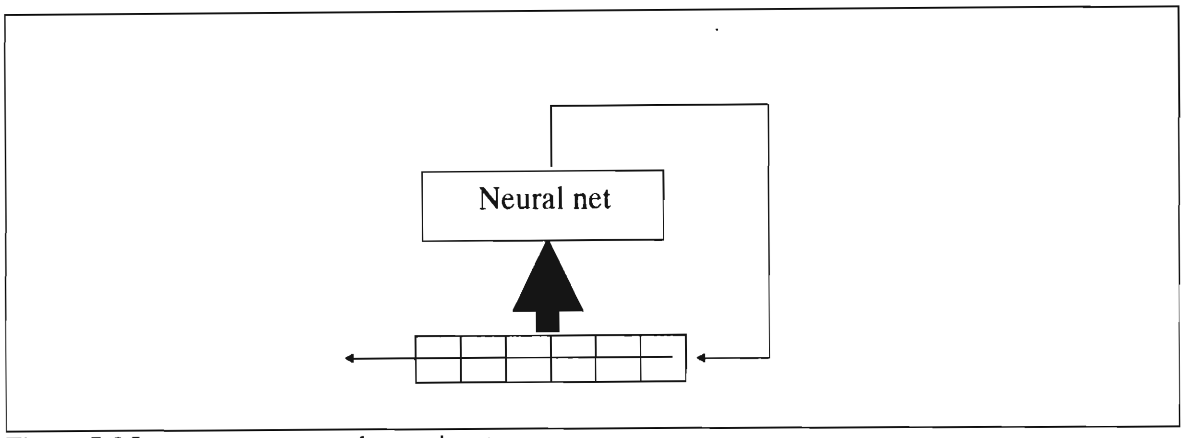


Figure 5-3 Input - note generated - new input

Two methods of copying the generated note to the input vector were considered:

- The output vector (and thus the generated note) was copied directly to the relevant position in the input vector. Note that the input vector should be binary valued but there are no guarantees, and indeed it is unlikely, that the output vector's components would consist of only two values. In general the output vector would be a real valued vector and the unit with the largest value would be considered to be "on" while all others would then be considered "off", thereby giving a binary valued vector. By copying the output vector to the input vector directly, the input vector would be a real valued vector after at least *n* copying processes (if the net was trained *n* notes deep).
- The output vector was converted to a binary valued vector, by considering the unit with the largest activation value to be "on" and all other units "off", and this converted vector was then copied to the relevant position in the input vector.

Eberhart and Dobbin [ED90] discuss other methods which use a probabilistic approach to convert the output vector to a binary vector, but it was found that the above two methods were adequate.

5.5.5. Results

With an application of this type, there is little or no qualitative measure by which to judge the results. The results are therefore judged subjectively and different people will disagree on the quality of the composed music and thus on the amount of generalization that has taken place.

Three nets were trained: three notes, four notes and five notes deep. CCA was again used as the learning algorithm. No hidden units were generated in any of the nets which indicates that the data is linearly separable and can thus be learnt by the Perceptron. Hidden units are often credited with the generalization ability of ANNs by forming higher order feature detectors [FL90]. To test this, CCA was forced to generate hidden units by adjusting the patience and output decay parameters as in previous tasks. No appreciable difference was noticed in the quality of the music composed by the nets with hidden units when compared to the music composed by the nets without hidden units.

The two error measures were compared and it was found that the net ceased to learn, or learnt very slowly, when the "number of bits wrong" measure reported that 70 bits were incorrect. Moreover, the composed music using a net trained with this error measure was awful. Learning was considered complete when the sum-of-squares error reached a certain threshold, which was varied from 0.25 to 0.45 in 0.05 size steps. The training time required for the net trained four notes deep appears in table 5-5. Note that the rate of convergence slows down when the error threshold is decreased. For example, decreasing the threshold from 0.3 to 0.25 causes the number of iterations required to solve the problem to more than double. This is a characteristic of gradient descent techniques [Mol90] such as that employed in Cascade-Correlation. It would seem that allowing a weaker condition (larger error threshold) to terminate the learning would add more variability to the composed music, while stricter conditions (smaller error threshold) would lead the net to memorize the training data better. By requiring the net to reach a smaller error measure the weights are forced to change in such a way that the training patterns are better remembered. This could cause the prototypes in the training examples not to be learnt. This was borne out by the results.

Error threshold	Epochs
0.25	72
0.30	29
0.35	21
0.40	17
0.45	15

Table 5-5 Training times for different error thresholds.

Both methods of copying the generated output note to the input were compared. Converting the output vector to a binary valued vector before the copying process resulted in better composed music. Four different seeds were used to start the composition process: the first *n* notes of each of the training songs and *n* randomly chosen notes (where *n* is the number of notes in the input vector). Again little difference was noted in the composed music although the random seed often led to music that was surprisingly different from the training tunes yet of a similar style.

The net trained five notes deep led to an almost complete memorization of the training tunes. Even when seeded with a random starting seed, the net quickly started producing one of the training songs. The net produced the three tunes with similar frequency with no combination of the tunes resulting and few incorrect notes. This was somewhat unexpected but can be explained by examining the training tunes: if a certain sequence of notes only appears in one of the tunes it seems probable that the net would respond with the next note in the sequence of that song when presented with that part of the sequence. This process should continue as long as only one of the tunes has such a sequence of notes. Therefore, provided there aren't too many note sequences that are identical in the different training songs, the net should be able to deliver the trained response. What was interesting was that this net must have learnt the training songs (almost) perfectly since any slightly ambiguous output note could introduce enough noise so that the next note output wasn't from that trained sequence. Thus the net had learnt to recall a temporal sequence even though no temporal information was explicitly provided by the net architecture. Were the training data to have more sequences of notes in common the net could confuse them and thus not produce the trained songs perfectly, if at all.

The net trained four notes deep gave the best results with a mixture of memorized note sequences (five or six notes) and "new" note sequences. The "three notes deep" net seemed to produce the most randomized, and unpleasant, music.

5.5.6. Improvements

Eberhart and Dobbin [ED90] discuss various ways in which to inject randomness into the music. As was seen, nets trained with a smaller sized input vector tended to result in more random music being composed. Thus combining nets with different sized input vectors and choosing the generated note from one of the nets in some way, could result in better music being composed. However this method is computationally expensive since it requires the training of more than one net. Alternatively, the unit considered as the note to be played next could be chosen in some other way rather than always choosing the unit with the largest activation value to be the next note.

The manner in which the composed music is played is important. The program used to play back the composed music selected the unit with the largest activation value as the generated note which was then played through the internal speaker of an IBM PC. However, when more than one unit had the same high activation value, the unit corresponding to the higher note was selected. Therefore, occasionally high notes were heard amongst a sequence of low notes. There seems no general procedure for handling such cases except by using some probability measure to choose between units with the same activation value.

The composed songs generally do not come to a natural end. This was expected since, among all the note sequences, there were only three sequences that represented the end of a tune as the training set was composed of three tunes. Thus the net had only three examples of suitable ending sequences compared to the many non-ending sequences. This problem might be overcome by including more examples of song endings in the training set.

5.5.7. Conclusion

Composing music based on rules is difficult because of the lack of a qualitative measure to classify the composed music. An ANN has been successfully applied to the task of music composition. Although the composed music is adequate, the preferred method would be for the human composer to use the ANN to compose the music while making changes to the composition where necessary.

A significant result is the ability of a feed-forward net to seemingly learn temporal sequences without having to resort to recurrent or time delay connections.

5.6. IMAGE BINARIZATION

5.6.1. Introduction

ANNs have been applied to image processing with some success:

- Yin *et al.* [YAN] used ANNs to remove noise from images with results which compared favourably with those obtained using the median filter. However, their net was specifically designed for the task and underwent little training.
- Bilbro *et al.* [BWS87] designed an ANN, trained with the Boltzmann algorithm, to segment images to extract the object in the image.

The task is to train an ANN to binarize an image by determining an adaptive threshold.

5.6.2. Image binarization

Image binarization is the process of converting a grey-level image to a bi-level one by selecting a threshold t such that

$p(x,y) = 1$ if $p(x,y) < t$
$p(x,y) = 0$ otherwise,
where $p(x,y)$ is the value of the pixel at position (x,y) in the image.

Binarization is important for a variety of reasons:

- The differences between objects and the background in images may be clearer in two tone images. This eases feature extraction.

- Image processing requires the processing of large amounts of data. By representing the images in bi-level, the amount of storage required for each image can be reduced. For example, an image of 256x256 pixels where each pixel can store 256 shades of grey requires 524 288 bits, while the equivalent binary image would require 65 536 bits; 87.5% less.
- Image skeletonization can be performed on bi-level images [Amm86].

Determining an appropriate threshold can be a difficult problem. The threshold is usually determined by performing various operations on the histogram of grey-levels of an image (i.e. a histogram that reflects the distribution of the pixel values in the image). If the image consists of mostly two grey levels, such as black text on a white background, the histogram will normally have two peaks (bi-modal). For such images, choosing an appropriate threshold is less complicated as the threshold is usually chosen to fall somewhere between the two peaks.

Rosenfeld and Kak [RK82], for example, give three different methods to choose a threshold. A popular method is the *p-tile* method [RK82]: if it is known what fraction of pixels should be above the threshold, the threshold *t* can be chosen appropriately. If 1-p of the pixels should be above the threshold, the threshold is chosen to be at the p-tile of the image's histogram. *Optimal thresholding* [GW87] selects the "best" threshold using statistical information contained in the histogram. The image is assumed to contain two principle brightness regions (i.e. the histogram is bi-modal), and thus the histogram can be considered as an estimate of the brightness probability density function [RK82]. The resulting algorithm for determining the optimal threshold is, however, non-trivial to implement. Both of these methods require bi-modal histograms. For non-bi-modal histograms or for images for which no *a priori* information is available, the problem of determining a threshold is more complicated.

A promising method is Otsu's *Discriminant And Least Square Binarization* (DLSB) [BYK90]. DLSB has no need for *a priori* information but determines the optimal threshold in the sense that the chosen threshold minimizes the square error between the binarized and original image.

If a set of images with the best threshold predetermined in some way is available then at least two methods are available:

- *Mean threshold binarization* and *median threshold binarization.* The mean or median of the sample thresholds may be used as a threshold to binarize other images. For these methods to work adequately, it is important that the example set of images and thresholds are chosen with care to be representative of all images that will need to be binarized using this method. The selection of such an example set of images could be difficult and time consuming. The biggest disadvantage of this method would be that the threshold determined for new images is not adaptive.

- The images and thresholds may be used as training data for an ANN where, given the histogram of the image as input, the net would determine the desired threshold. This might allow an adaptive threshold to be determined. This is the method followed and is an extension of the work done by Babaguchi *et al.* [BYK90].

5.6.3. Babaguchi's approach

Babaguchi *et al.* [BYK90] had two guiding principles when designing their network:

- The net was intended not to average the training images thresholds but rather to use the complete histogram data to determine an appropriate threshold. To encourage this generalization, they kept the training data set as small as possible. This would hopefully cause the net to extract the prototypes rather than just memorizing the training data.
- They wanted to produce binary images that were visually satisfying. Therefore their criteria for judging the binary images were based on visual appearance rather than a statistical measure which clearly influenced the choice of the thresholds. Although the selection of the thresholds then becomes a subjective process, the thresholds might produce visually more satisfying bi-level images.

Their method, termed *Connectionist Model Binarization* (CMB), consisted of two stages: a learning and binarizing stage. The first stage used the training data to teach the net to select appropriate thresholds given the histograms of the images as input and the user determined thresholds as the target outputs. The second stage was the prediction (or test) stage where unseen images were presented to the net and it supplied the thresholds.

The CMB network was a three-layer network with *N* input and output nodes, where *N* is the number of points in the histogram. The input to the net was the grey-level histogram. Because the net expected inputs in the range [0,1], the histograms entries had to be normalized. This was a simple task of determining the largest entry in the histogram, and dividing all the histogram entries by this value; i.e. the normalized histogram *h(s)* was

$$\mathrm{h(s)} = \frac{H(s)}{L} \qquad (0 \leq s \leq N-1)$$

where H(s) was the original histogram's value, and L = max(H(0), H(1), ..., H(N-1)).

The number of units in the hidden layer was varied to determine the effect this would have on the net's performance. The number of hidden units used were 8, 16, 32 and 64. The output layer consisted of a binary vector that indicated the threshold to be used given the input histogram with the component with the largest value indicating the threshold to be used.

The teacher signal *t(s)*, which corresponded to the desired threshold, was presented in three ways:

- Impulse type:
  $t(s)=1$ for $s=s'$
  $t(s)=0$ otherwise
- Step type:
  $t(s)=1$ for $s'-\varepsilon < s < s'+\varepsilon$
  $t(s)=0$ otherwise
- Gaussian type:
  $t(s)=e(-0.1562(s-s')^2)$ for $s'-\varepsilon < s < s'+\varepsilon$
  $t(s)=0$ otherwise
  where s' is the desired threshold.

Experimental results indicated that the impulse type delivered the best results.

Their aim was to train the net using similar images. Thus, images of newspaper (NP) and technical papers (TP) were used. The images were 840x840 pixels and the pixel values were limited to a grey-level of 64; i.e. 0 through 63. Therefore the input and output layers had 64 units each. Twenty images of each type(NP and TP) were captured, and 10 selected at random from each group to be used as training data. The thresholds for the training images were determined interactively. The training images were presented to the net in four different ways:

- $(NP1, TP1, ..., NP10, TP10)^1$
- $(NP1, ..., NP10, TP1, ..., TP10)^1$
- $(NP1, TP1, ..., NP10, TP10)^k$
- $(NP1, ..., NP10, TP!, ..., TP10)^k$

where NP1, NP2,...NP10 and TP1, TP2,...TP10 denote the NP and TP training images respectively, and $()^k$ indicates that each of the samples parenthesized were repeated *k* times in the order specified. It was found that the second presentation method achieved the best learning speeds.

5.6.4. Extending the problem

It was decided to extend the range of problems that Babaguchi tried to solve. The images were all of printed matter but no assumptions were made about the size or style of the typeface, lighting conditions, the angle at which the image was captured or whether the image contained text or diagrams. The images included graphs and simple pictures as well as images with the foreground and background colours reversed (e.g. white writing on a blue background rather than the familiar black-on-white).

The images were restricted to 128x128 pixels due mainly to computational limitations and because the digitizer used was only capable of 512x512 pixel images and there were defective pixels located in one section of the digitizer's retina. By choosing the images' position carefully, these pixels were avoided. Babaguchi *et al.* used images of Chinese text which generally consist of finer lines where the choice of threshold must be made carefully to avoid blurring of the lines. Due to the lack of availability of their original images, it was decided to use images of English text. The effect of the smaller images and style of writing could cause the histograms to be less obviously bi-modal. In fact, the histograms of the images used were generally not bi-modal, certainly less bi-modal than the histograms illustrated in [BYK90]. The twenty training images and seven of the test images appear in appendix D.

The visually most satisfying thresholds were determined interactively for the twenty training images by trial-and-error and these thresholds are given in table 5-6. When testing the net, the output unit with the largest activation value was assumed to be the threshold determined by the net.

Image	Threshold	Image	Threshold	Image	Threshold	Image	Threshold
1	35	6	23	11	29	16	39
2	35	7	39	12	28	17	39
3	63	8	56	13	35	18	38
4	60	9	23	14	40	19	32
5	40	10	35	15	42	20	34

Table 5-6 User-specified thresholds of the 20 training images. The mean and median of these thresholds is 38.25 and 37.0 respectively.

5.6.5. Results

5.6.5.1. CMB results

Babaguchi *et al.* limited their BP algorithm to 50 000 iterations. They determined experimentally that there was no difference in performance when using 32 or 64 hidden units, while there were only two threshold samples that differed when testing a net that used eight hidden units. A learning rate of 0.5 and momentum of 0.9 were used as the training parameters of BP with the performance of the net being superior to mean threshold binarization (MTB). MTB showed no stable performance even for images in the same set while DLSB had the tendency to cause blurring.

5.6.5.2. Extended problem results

Choosing a threshold too small causes blurring while a threshold too large causes a loss of detail. Learning ceased when the sum-of-squares error was less than 0.35. This value was a rough rule of thumb which have given good results in other tasks. Table 5-7 details the effects of different parameter values on learning times and the number of hidden units added to the net. As can be seen, the output decay dramatically increases the number of hidden units added with an increase in the learning time resulting.

Patience	Output decay	Number of epochs	No. of hidden units
8	0.01	60	0
3	0.01	77	1
4	0.01	130	6
8	0.1	1720	33
5	0.1	1637	44
3	0.1	1440	49

Table 5-7 Details of the effects of different parameter values on learning time and number of hidden units

The performance of the net was tested with a set of different images and comparisons were made between the results obtained and those obtained when using the mean and median of the training set, as well as thresholds determined interactively. It was found that the nets with a large number of hidden units (those trained with the output decay parameter set to 0.1) determined thresholds that were better than those determined by the other nets (for which the output decay parameter was set to 0.01).

Appendix E contains the results obtained when using some of the test set images to test the net while appendix F details the results obtained using the training data to test the net. The effect of hidden units on the net's performance was studied by training two sets of nets: one group with less than twenty hidden units and the other with more than twenty hidden units. The number of hidden units added can be influenced by the choice of the output decay term as detailed above.

It was found that the nets with more than 20 hidden units generally performed better in all the tests. However, there was no appreciable difference in binarization quality between members from the same group. The third and fourth row of images in appendix E demonstrate the difference in performance between the two groups of trained nets: the first group of nets, with less than 20 hidden units, were unable to binarize the image in such a way that the fine detail was classified as foreground while the second group correctly binarized the image.

When using the training set to test the trained net, it was found that the training set had not been memorized perfectly, with the nets sometimes failing to binarize certain training images.

In all such cases, using the mean and median threshold failed too. As can be seen in some of the binarized images, the thresholds determined by the net were occasionally too large which caused blurring. This occurred in images that had a large amount of foreground colour in one region; such as when the image contained only a portion of a letter in a large font. However, in general the adaptive threshold determined by the net was equal or superior to the mean and median, which obviously failed to take the test image's distribution into affect when determining a threshold.

5.6.6. Conclusions

The task of determining a threshold to binarize an image is well suited to being solved using a neural net. The advantage of using a net is that it forms an adaptive method of determining the threshold. Further, the threshold determined need not be statistically significant but can be chosen interactively to suite whatever purpose the experimenter has in mind. The method is not limited to bi-modal histograms and no *a priori* information about the images are required. To prevent the net from merely memorizing the data or calculating the average of the thresholds it is important to encourage generalization by including a number of hidden units.

Babaguchi's work was extended to include varied lighting conditions and viewing angles, using images with various combinations of foreground and background colours and the use of images of differing styles of text. The trained ANN performed adequately in all cases, including when presented with (pseudo) 3-D text of which no examples were included in the training set, often outperforming the traditional methods considered and almost matching the threshold determined interactively for the test images.

An obvious area to further explore would be the application of the process to images of 3-D objects and *n-arization* where the grey-level image is converted to an image of *n* grey levels.

5.7. Conclusion

ANNs trained using CCA were applied to four problems. Although successful in all four problems, it was seen that traditional methods could give superior results (such as in the calculation of decompression stops). By increasing the complexity of some of the problems, such as increasing the input size of BOB's sensors, the advantage of neural networks would become clearer.

It was also shown that ANNs could be used in fields such as musical composition with a certain degree of success while very encouraging results were obtained in the image binarization problem. Thus ANNs are in fact a useful technique, the theoretical aspects and pitfalls not withstanding.

CONCLUSION

The resurgence of interest in artificial neural networks has resulted in models that overcome many of the difficulties experienced by the earlier models, are more powerful and thus can learn more complex tasks. This thesis has examined the non-recurrent, feedforward architecture first used by the original Perceptron. The Backpropagation rule, a generalization of the Perceptron rule, and its properties were examined. It was noted that although the BP algorithm has a slow rate of convergence, there are many simple modifications that can be made to the algorithm to increase the learning speed.

It was noted that BP scaled badly; as the size of the task is increased linearly, BP's learning time required often scales exponentially. The quickprop algorithm, a semi-Newton method, derived from BP, was examined. It was found to be significantly faster than BP and seemed to scale better than BP. The presence of local minima in the error surface searched by BP can lead to the net reaching an inferior solution. Backpercolation is an algorithm that assigns each unit in the net its own activation error and thus enables each unit to try to descend a local error surface. Perc had the additional feature that it is faster than BP while QP compares favourably to it. Due to the proprietary nature of the Perc algorithm little research by investigators other than its inventor has taken place. Therefore the claims of its performance must be regarded skeptically. Various possible reasons for its improved rate of convergence were mentioned and it remains to be seen whether the improvements are due to the new approach of assigning local error surfaces to each unit or due to these other factors.

The task of learning an arbitrary task using an arbitrary fixed feedforward architecture is an NP-hard problem. Therefore, it seems unlikely that an efficient algorithm exists that can accomplish such tasks. This result has not been extended, however, to generative or recurrent nets. As noted, many of the theorems relating to the Perceptron seem to be extendible to multilayer feedforward nets but do not apply to recurrent nets. The basic requirements of generative algorithms were investigated and their advantages over the other algorithms noted. Cascade-Correlation, a generative algorithm developed by Fahlman, was discussed.

Finally four tasks were solved using neural nets. Different criteria were used in selecting the tasks. These included the testing of the generalization ability of a CCA trained net using the task of calculating decompression stops required by SCUBA divers and the learning of rules using a suitable designed training data set for training an artificial critter. The quality of the solutions varied from adequate (such as for the calculation of the decompression stops) to very promising (such as the binarization of images). Tasks such as the composition of music and binarization of images indicated the possibility of solutions to problems that are difficult to solve using traditional methods. An unexpected result was the learning of temporal sequences by a non-recurrent net trained by CCA: it was previously assumed that recurrent or time delay connections would be required for such tasks.

Various conclusions may be drawn:

- All the models studied only required local information; information that is locally available to the unit that is changing the weights associated with it. Even though there is no explicit controlling unit as in traditional AI systems (e.g. expert systems) synchronization of the units must still be supplied by an external unit. For example in BP, the algorithm must broadcast to the units when to propagate a signal forward and when to update weights.
- ANNs are a useful method that may be applied to a variety of problems. However, they should not be considered as a catch all method that can solve all problems. Many problems can be better solved using traditional methods. An avenue to be further researched is the synthesis of traditional methods and ANNs into hybrid systems. Caudill [Cau90b] [Cau90c] [Cau90d], for example, has discussed various methods of combining expert and fuzzy decision systems with ANNs.
- Given the result that learning in non-recurrent feedforward nets is NP-hard it must be determined what restrictions on the tasks, architectures and algorithms may lead to efficient solutions as suggested by Judd.
- The investigation of whether the use of recurrent and generative algorithms will overcome the NP-hard result is required.
- The design and use of generative algorithms should be given more attention due to the advantages inherent in such algorithms.
- The use of recurrent nets for tasks involving temporal sequences and as a method to overcome the possible limitations of the results of Minsky and Papert [MP88], if these are extendible to multilayer nets, should be considered. Recurrent nets have their own set of unique problems however. One important problem would be that of the introduction of chaotic behaviour which often appears in non-linear, feedback systems [Gle88]. Pollack [Pol88], for example, discussed methods in which chaotic behaviour could be utilised to represent certain types of AI-style data structures while Doya [Doy92] discusses methods to overcome the problem of bifucations during learning in recurrent nets.
- ANNs can accomplish adaptive tasks, such as image binarization, in image and signal processing.
- The need to give a net as much help as possible during training was found to be important. This could include the encoding of spatial or temporal information in nets required to solve tasks that include such details.
- The importance of local minima should be quantified. From an applications point of view local minima may be safely ignored provided the trained net performs satisfactorily. From a theoretical view, it is important that methods that allow the reduction of the

effect local minima have on the search for an adequate solution and on the quality of the final solution be determined.

- ANNs perform well in tasks that have few fixed or quantifiable rules, or tasks that require some type of generalization (musical composition is a task where this is required).
- The use of ANNs in domains that require a deal of resistance to damage or to fuzzy and noisy input is highly recommended.
- The use of genetic algorithms to develop new or improved algorithms, search methods or nets should be investigated. Caudill [Cau91b] and Austin [Aus90] discuss this for example.
- The use of a collection of smaller nets instead of one large net to solve a large or complex problem. This has the advantage of avoiding or reducing the scaling affect inherent in many of the popular algorithms while still allowing a good solution to be found. Lippmann [Lip89], for example, discusses this in connection with speech recognition using ANNs.
- Algorithms that reduce or overcome the scalability problem are required.

REFERENCES

[Abu89] Abu-Mostafa Y.S. (1989). The Vapnik-Chervonenkis Dimension: Information versus Complexity in Leanring, *Neural Computation* 1, p312-317.
Oops.

[ADG90] Apolloni B., Donzelli P., Gianelli G. & Peverelli A. (1990). Neural Networks for Satellite Orbit Control Systems. In *Third Italian Workshop in Parallel Architectures and Neural Networks*, ed. Caianiello E.R., p281-293, London: World Scientific.

[Alm87] Almeida L.B. (1987). Backpropagation in Perceptrons with Feedback. In *Neural Computers*, eds. Eckmiller R. & v.d. Malsburg C., p199-208, Berlin: Springer-Verlag.

[AM90] Aleksander I. & Morton H. (1990). *An Introduction to Neural Computing*, London: Chapman and Hall.

[Amm86] Amman C.J. (1986). A New Thinning Algorithm for Binary Images. *M.Sc Thesis*, appendix C, pC22-C25, Durban: University of Natal.

[And90] Anderson J.A. (June 1990). Data Representation in Neural Networks, *AI Expert*, p31-37.

[AR89] Anderson J.A. & Rosenfeld E. (1989). *Neurocomputing: foundations of research*, Cambridge: MIT Press

[Arb87] Arbib M. A. (1987). Schemas and Neurons: two levels of neural computing. In *Neural Computers*, eds. Eckmiller R & v.d. Malsburg C., p311-320, Berlin: Springer-Verlag.

[Atk89] Atkinson K.E. (1989). *An introduction to Numerical Analysis*, chapter 8, New York: Wiley.

[Aus90] Austin S. (March 1990). An Introduction to Genetic Algorithms, *AI Expert*, p49-53.

[BA91] Bechtel W. & Abrahamsen A. (1991). *Connectionism and the Mind: an introduction to parallel processing in networks*, Oxford: Basil Blackwell Ltd.

[Bau88] Baum E.B. (1988). On the Capabilities of Multilayer Perceptrons, *Journal of Complexity* 4, p193-215.

[Bau89] Baum E.B. (1989). A Proposal for More Powerful Learning Algorithms, *Neural Computation* 1, p201-207.

[BB90] Brngio Y. & Bengio S. (1990). *Learning a synaptic learning rule*, Technical Report #751, Universite de Monreal.

[BKR78] Brodie S.E., Knight B.W. & Ratliff F. (1978). The Response of the *Limulus* Retina to Moving Stimuli: a prediction by Fourier synthesis. *Journal of General Physiology* 72, p129-154, 162-166.

[BL90] Brunak S. & Lautrap B. (1990). *Neural Networks: computers with intuition*, London: World Scientific.

[BP90] Bonori M. & Paolini A. (1990). Notes on the Design of an Artificial Neural Network. In *Third Italian Workshop in Parallel Architectures and Neural Networks*, ed. Caianiello E.R., p199-214, London: World Scientific.

[Bre91] Brent R.P. (1991). Fast Training Algorithms for Multilayer Neural Networks, *IEEE Transactions on Neural Networks* 2 (3), p346-354.

[Bro91] Brooks R.A. (1991). Intelligence Without Representation, *Artificial Intelligence* 47, p139-159.

[BRS89] Brady M.L., Raghavan R. & Slawny J. (May1989). Backpropagation Fails to Separate where Perceptrons Succeed, *IEEE Transactions on Circuits and systems* 36 (5), p665-674.

[Bur90] Burrascano P. (1990). Multilayer Perceptron for Image Compression. In *Third Italian Workshop in Parallel Architectures and Neural Networks*, ed. Caianello E.R., p309-317, London: World Scientific.

[BWS87] Bilbro G.L., White M. & Snyder W. (1987). Image Segmentation with Neural Computers. In *Neural Computers*, eds. Eckmiller R & v.d. Malsburg C., p71-80, Berlin: Springer-Verlag.

[BYK90] Babaguchi N., Yamada K., Kise K. & Tezuka Y. (June 1990). Connectionist Model Binarization. In *Tenth International Conference on Pattern Recognition*, vol 2, p51-56, Los Alamitos: IEEE Computer Society Press.

[Cau89] Caudill M. (August 1989). Neural Network Primer VIII, *AI Expert*, p61-67.

[Cau90a] Caudill M. (June 1990). Using Neural Networks: fuzzy cognitive maps, *AI Expert*, p49-53.

[Cau90b] Caudill M. (July 1990). Using Neural Networks: making an expert network, *AI Expert*, p41-45.

[Cau90c] Caudill M. (September 1990). Using Neural Networks: diagnostic expert nets, *AI Expert*, p43-47.

[Cau91a] Caudill M. (January 1991). Neural Network Training Tips and Techniques, *AI Expert*, p56-61.

[Cau91b] Caudill M. (March 1991). Evolutionary Neural Networks, *AI Expert*, p28-33.

[Cha80] Chamberlin H. (1980). *Musical Application of Microprocessors,. Rochelle Park: Hayden Book.*

[Cou79] Cousteau, J.Y. (1979). *The Ocean World,* 1985 edition, New York: H.N. Abrams Inc.

[DAR87] DARPA (1987). *DARPA neural network study: Oct 1987- Feb 1988,* New York: AFCEA International Press.

[Day90] Dayhoff J. (1990). *Neural Networks Architecture: an introduction,* New York: Van Nostrand Reinhold.

[Doy92] Doya K. (1992). Bifurcations in the Learning of Recurrent Neural Networks, to appear in *Proceedings of 1992 IEEE International Symposium on Circuits and Systems.*

[Dur89] Durbin R. (1989). On the Correspondence between Network Models and the Nervous System. In *The Computing Neuron*, eds. Durbin R., Miall C. & Mitchison G., p1-10, Menlo Park: Addison-Wesley.

[ED90] Eberhart R.C. & Dobbin R.W. (1990). *Neural Network PC tools: a practical guide*, chapter 14, London: Academic Press.

[Fah88] Fahlman S.E. (September 1988). *An Empirical Study of Learning Speed in Back-Propagation Networks*, Carnegie-Mellon University Technical Report CMU-CS-88-162.

[FB82] Feldman J.A. & Ballard D. (1982). Connectionist Models and their Properties, *Cognitive Science* 6, p205-254.

[Fel82] Feldman J.A. (1982). Dynamic connections in neural networks, *Biological Cybernetics 47* (1), p27-39.

[FL90] Fahlman S.E. & Lebiere C. (February 1990). *The Cascade-Correlation learning algorithm*, Carnegie-Mellon University Technical Report CMU-CS-90-100.

[FMI83] Fukushima K., Miyaka S. & Ito T. (1983). Neocognitron: a neural network model for a mechanism of visual pattern recognition, *IEEE Transactions on Systems, Man and Cybernetics* 13(5), p826-834.

[Gea91] Geake E. (9 Nov 1991). Neural Networks Bring Harmony to Hummers, *New Scientist*, p27.

[Geo77] George F.H. (1977). *The Foundations of Cybernetics*, London: Gordon & Breach Science Publishers.

[GJ79] Garey M.R. & Johnson D.S. (1979). *Computer and Intractability: a guide to the theory of NP-Completeness*. chapters 1-3, 5, 6, San Francisco: W.H. Freeman and Co.

[Gle88] Gleik J. (1988). *Chaos: making a new science*, London: Heinemann.

[Gol89] Goldberg D.E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*, Menlo Park: Addison-Wesley.

[GP89] Girosi F. & Poggio T. (1989). Representation Properties of Networks: Kolmogorov's Theorem is Irrelevant, Neural Computation 1, p465-469.

[Guy81] Guyton A.C. (1981). *Textbook of Medical Physiology*, Philadelphia: W.B. Saunders Co.

[GW87] Gonzalez R.C. & Wintz P. (1987). *Digital Image Processing*, second edition, p354-367, Menlo Park: Addison-Wesley.

[Heb49] Hebb D.O. (1949). *The Organization of Behaviour*, pxi-xix, 60-78, New York: Wiley

[Hec87a] Hecht-Nielsen R. (1987). Neurocomputing Applications. In *Neural Computers*, eds. Eckmiller R. & v.d. Malsburg C., p445-453, Berlin: Springer-Verlag.

[Hec87b] Hecht-Nielson R. (1987). Kolmogorov's Mapping Neural Network Existence Theorem, *Proceedings of the IEEE First International Conference on Neural Networks* 3, p11-14.

[Hec88] Hecht-Nielsen R. (1988). Neurocomputing: picking the human brain, *IEEE Spectrum*, p36-41.

[Hil90] Hillman D.V. (June 1990). Integrating Neural Networks and Expert Systems, *AI Expert*, p31-37.

[Hin89] Hinton G.E. (1989). Connectionist Learning Procedures, *Artificial Intelligence* 40 (1), p143-150.

[Hop82] Hopfield J.J. (1982). Neural Networks and Physical Systems with Emergent Collective Computational Abilities, *Proceedings of the National Academy of Sciences USA* 79, p2554-2558.

[HU91] Honavar V. & Uhr L. (1991). *Generative Learning Structures and Processes for Generalized Connectionist Networks*, Technical Report #91-02 (Department of Computer Science, Iowa State University).

[Jam90] James W. (1890). *Psychology (Briefer Course)*, chapter XVI, New York: Holt, p253-279.

[Jea68] Jeans J. (1968). *Science and Music,* New York: Dover Publications Inc.

[Jon90] Jones L.K. (1990). Constructive Approximations for Neural Networks by Sigmoid Functions, *Proceedings of the IEEE* 78 (10), p1586-1589.

[Jud87] Judd J.S. (1987). Learning in Networks is Hard, *IEEE First International Conference on Neural Networks*, p685-692.

[Jud90] Judd J.S. (1990). *Neural network design and the complexity of learning*, Cambridge: MIT Press.

[Jul75] Julesz B. (1975). Experiments in the Visual Perception of Texture, *Scientific American* 232 (4), p34-43.

[Jur] Jurik M. *Backpercolation: assigning local error in feedforward perceptron networks*, Proprietary Report: Jurik Research and Consultants.

[Jur88] Jurik M. (1988). Back error propagation: a critique, *Thirty third IEEE Computer Society International Conference,* Washington: Computer Society of IEEE, p387-392.

[Key91] Keyes J. (1991). *Getting Caught in a Neural Network*, AI Expert, p44-49.

[KH92] Krogh A. & Hertz J.A. (1992). *A Simple Weight Decay can improve Generalization*, UCSC Technical Report.

[KM91] Kruschke J.K. & Movellan J.R. (1991). Benefits of Gain: Speeded Learning and Minimal Hidden Layers in Backpropagation Networks, *IEEE Transactions of Systems, Man and Cybernetics* 21 (10), p273-280.

[Koh72] Kohonen T. (1972). Correlation Matrix Memories, *IEEE Transactions on Computers* C-21, p353-359.

[Koh90] Kohonen T. (1990). The Self-organizing Map, *Proceedings of the IEEE* 78 (9), p1464-1477.

[KP90] Kolen J.F. & Pollack J.B. (1990). Backpropagation is Sensitive to Initial Conditions, *Complex Systems* 4, p269-280.

[KS89] Kramer A.H. & Sangiovanni A. (1989). Efficient Parallel Learning Algorithms for Neural Networks. In *Advances in Neural Information Processing Systems I,* ed. Toeretzky D.S, p40-48, Morgan Kaufmann.

[KSe89] Kock C. & Segev I. (eds) (1989). *Methods in Neuronal Modeling: from synapses to networks*, Cambridge: MIT Press.

[Lai88] Laing R. (1988). Artificial Organisms: history, problems and directions. In *Artificial Life*, ed. Langton C.G., p1-48, New York: Addison-Wesley.

[Lan88] Langton C.G. (1988). Artificial Life. In *Artificial Life*, ed. Langton C.G., p49-62, New York: Addison-Wesley.

[Lev89] Levine D.S. (1989). The Third Wave in Neural Networks, *AI Expert*, p27-31.

[Lip87] Lippmann R.P. (1987). An Introduction to Computing with Neural Networks, *IEEE ASP Magazine*, p4-22.

[Lip89] Lippmann R.P. (1989). Review of Neural Networks for Speech Recognition, *Neural Computation* 1, p1-38.

[Llo89] Lloyd D. (1989). *Simple Minds*, Cambridge: MIT Press.

[LMB90] Le Cun Y., Matan O., Boser B. *et al* (1990). Handwritten Zip-code Recognition with Multilayer Networks. In *The Tenth International Conference on Pattern Recognition* 2, p35-40, Los Alamitos: IEEE Computer Society Press.

[LS88] Lekhy S.R. & Sejnowski T.J. (1988). Network Model of Shape-from-shading: neural function arises from both receptive and projective fields, *Nature* 33 (2), p452-454.

[McL89] McLaren I. (1989). The Computational Units as an Assembly of Neurones: an implementation of an error correcting learning algorithm. In *The Computing Neuron*, eds. Durbin R., Miall C. & Mitchison G., p160-179, Menlo Park: Addison-Wesley.

[Meh84] Mehlhorn K. (1984). *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*, eds. Brauer W., Rozenberg G. & Salomaa A., chapter VI, New York: Springer Verlag.

[MGK90] Miller W.T., Glanz F.H. & Kraft L.G. (1990). CMAC: an Associative Neural Network Alternative to Backpropagation, *Proceedings of the IEEE* 78 (10), p1561-1567.

[Mia89] Miall C. (1989). The Diversity of Neuronal Properties. In *The Computing Neuron*, eds. Durbin R., Miall C. & Mitchison G., p11-24, Menlo Park: Addison-Wesley.

[Mol90] Moller M.F. (1990). *A Scaled Conjugate Gradient Algorithm for Fast Supervised Learning*, preprint PB-339 (Computer Science Department, University of Aarhus, Denmark).

[MOP90] Marchesi M., Orlandi G., Piazza F., Pignotti G. & Uncini A. (1990). Dynamic Topology Neural Network. In *Third Italian Workshop in Parallel Architectures and Neural Networks*, ed. Caianello E.R., p107-115, London: World Scientific.

[MP43] McCulloch W.S. & Pitts W. (1943). A Logical Calculus of the Ideas Immanent in Nervous Activity, *Bulletin of Mathematical Biophysics* 5, p115-133.

[MP88] Minsky M.L. & Papert S.A. (1988). *Perceptrons: an introduction to computational geometry*, expanded edition, Cambridge: MIT Press.

[MRS90] Marsi S.; Ramponi G. & Sicuranza G.L. (1990). Image Compression Using a Perceptron. In *Third Italian Workshop in Parallel Architectures and Neural Networks*, ed. Caianello E.R., p355-364, London: World Scientific.

[MSV90] Malferrari L., Serra R. & Valastro G. (1990). Using neural networks for signal analysis in oil well drilling. In *Third Italian Workshop in Parallel architectures and neural networks*, ed. Caianiello E.R, p345-353, London: World Scientific.

[MT91] McCauley J.D., Thane B.R. & Whittaker A.D. (1991). *Fat Estimation in Beef Ultrasound Images Using Texture and Adaptive Logic Networks*, A&M University Technical Report.

[Mye91] Myers C. (1991). Learning with Delayed Reinforcement Through Attention-driven Buffering, *International Journal of Neural Systems* 1 (4), p337-346.

[NW90] Nguyen D. & Widrow B. (1990). The Truck Backer-upper: an example of self-learning in neural networks. In *Neural Networks for Control*, eds. Miller W.T., Sutton R.S. & Werbos P.J., p286-299, Cambridge: MIT Press.

[Nic90] Nicole S. (1990). A Network Model of Habitation in *Aplysia*. In *Third Italian Workshop in Parallel Architectures and Neural Networks*, ed. Caianello E.R., p149-158, London: World Scientific.

[Osb85] Osborne M.R. (1985). *Finite Algorithms in Optimization and Data Analysis*, chapter 3, New York: Wiley.

[Pao89] Pao Y.H. (1989). *Adaptive pattern recognition and neural networks*, chapter 8, p197-222, Menlo Park: Addison-Wesley.

[Pol] Pollack J.B. *Implications of Recursive Distributed Representations*, Laboratory for AI Research (Ohio State University) Technical Report.

[Pol88] Pollack J.B. (1988). *Book review: Perceptrons: An introduction to computational geometry, expanded edition* (to appear in the Journal of Mathematical Psychology).

[RHW86] Rumelhart D.E., Hinton G.E. & Williams R.J. (1986). Learning Representation by Back-propagating Errors, *Nature* 323 (9), p533-536.

[RK82] Rosenfeld A. & Kak A.C. (1982). *Digital Picture Processing* 2, p61-66, New York: Academic Press.

[RM86] Rumelhart D.E., McClelland J.L. & the PDP Research Group (1986). P*arallel Distributed Processing: Explorations into the microstructure of cognition*, vol 1: Foundations, Cambridge: MIT Press.

[Ros58] Rosenblatt F. (1958). The Perceptron: a Probabilistic Model for Information and Organization in the Brain, *Psychological Review* 65, p386-408.

[SAU] South African Underwater Union (1990). *Snorkel/one star sport diver: course manual*, RSA Litho (pty) Ltd, third revision.

[SH87] Stornetta W.S. & Huberman B.A. (1987). An Improved Three-layer Backpropagation Algorithm, *Proceedings of the IEEE International Conference on Neural Networks*, p637-644.

[Sha90] Shanno D.F. (1990). Recent Advances in Numerical Techniques for Large-scale Optimization. In *Neural Networks for Control*, eds. Miller W.T., Sutton R.S. & Werbos P.J., p171-178, Cambridge: MIT Press.

[SK91] van der Smagt P.P. & Kröse B.J.A. (1991). A Real-time Learning Neural Robot Controller, *Proceedings of the 1991 International Conference on Artificial Neural Networks*, p351-356.

[Sou89] Soucek B. (1989). *Neural and Concurrent Real-time Systems: the Sixth Generation*, New York: Wiley.

[SR86] Sejnowski T.J. & Rosenberg C.R. (1986). *NETtalk: a parallel network that learns to read aloud*, John Hopkins University Technical Report (JHU/EECS-86/01).

[Sut90] Sutton R.S. (1990). First Results with Dyna, an integrated architecture for learning, planning and reacting. In *Neural Networks for Control*, eds. Miller W.T., Sutton R.S. & Werbos P.J., p179-189, Cambridge: MIT Press.

[SWW90] Stevenson M., Winter R. & Widrow B. (1990). Sensitivity of Feedforward Neural Networks to Weight Errors, *IEEE Transactions on Neural Networks* 1 (1), p71-80.

[Tay90] Taylor J.G. (1990). New Paradigms for Neural Networks. In *Third Italian Workshop in Parallel Architectures and Neural Networks*, ed. Caianello E.R., P19-43, London: World Scientific.

[Thr91] Thrun S.B. *et. al.* (1991). *The MONK's Problems: A performance comparison of different learning algorithms*, Technical Report CMU-CS-91-197.

[TJ88] Tesauro G. & Janssens B. (1988). Scaling Relationships in Backpropagation Learning, *Complex Systems* 2, p39-44.

[TS89] Tesauro G. & Sejnowski T.J. (1989). A Parallel Network that Learns to Play Backgammon, *Artificial Intelligence* 39 (3), p357-390.

[Tve91] Tveter D. (July 1991). Getting a Fast Break with Backpropagation, *AI Expert*, p36-43.

[VMR88] Vogl T.P., Mangis J.K., Rigler A.K., Zink W.T. & Alkon D.L. (1988). Accelerating the Convergence of the Backpropagation Method, *Biological Cybernetics* 59, p257-263.

[Wal75] Walsh G.R. (1975). *Methods of Optimization*, chapter 4, New York: Wiley.

[Wer90] Werbos P.J. (1990). Backpropagation through time: what it is and how to do it, *Proceedings of the IEEE* 78 (10), p1550-1559.

[WH60] Widrow B. & Hoff M.E. (1960). Adaptive Switching Circuits, *1960 IRE WESCON Convention Record*, p96-104, New York: IRE.

[Win84] Winston P.H. (1984). *Artificial Intelligence*, second edition, London: Addison-Wesley.

[WL90] Widrow B. & Lehr M.A. (1990). 30 years of adaptive neural networks: Perceptron, Madaline and Backpropagation, *Proceedings of IEEE* 78 (9), p1415-1439.

[YAN] Yin L., Astola J. & Neuvo Y. *Adaptive Boolean Filters for Noise Reduction*, Technical Report (Electrical Engineering Department, Tampere University of Technology.

[YH91] Yang J. & Honavar V. (1991). *Experiments with the Cascade-Correlation Algorithm*, Iowa State University Technical Report #91-16.

APPENDIX A

PERCEPTRON CONVERGENCE THEOREM

Definition:

Assume the input patterns are from a space which has two classes, $\mathbf{F}^+$, $\mathbf{F}^-$. The perceptron must respond with +1 for $\mathbf{F}^+$, -1 for $\mathbf{F}^-$.

Consider a set of input values $\underline{x}_i$ as vectors in i-dimensional space called **X** and a set of weights $\underline{w}_i$ as another vector in the same space, denoted by **W**. Assume that all vectors are of unit length.

Replace $\sum_{i=0}^{n-1} w_i x_i(t)$ by **W.X**

Algorithm:

```
Start: Choose any value for $\mathbf{W}$ ($\neq 0$)

Test: Choose $\mathbf{X}$ from $\mathbf{F}^+ \cup \mathbf{F}^-$
if $\mathbf{X} \in \mathbf{F}^+$ and $\mathbf{W.X} > 0$ then goto Test
if $\mathbf{X} \in \mathbf{F}^+$ and $\mathbf{W.X} \leq 0$ then goto Add
if $\mathbf{X} \in \mathbf{F}^-$ and $\mathbf{W.X} < 0$ then goto Test
if $\mathbf{X} \in \mathbf{F}^-$ and $\mathbf{W.X} \geq 0$ then goto Sub

Add: $\mathbf{W} = \mathbf{W} + \mathbf{X}$
goto Test

Sub: $\mathbf{W} = \mathbf{W} - \mathbf{X}$
goto Test
```

The algorithm can be simplified to:

```
Start:Choose any value for W

Test: Choose X from $F^+ \cup F^-$
if $X \in F^-$ then change the sign of X
if $W.X > 0$ then goto Test else goto Add

Add: $W = W + X$
goto Test
```

However, if we define $\mathbf{F} = \mathbf{F}^+ \cup \mathbf{F}^-$, the algorithm can be further simplified:

```
Start: Choose any value for W

Test: Choose any X from F
if $W.X > 0$ then goto Test else Add

Add: $W = W + X$
goto Test
```

THE PERCEPTRON CONVERGENCE THEOREM

PROOF

Let **F** be unit-length vectors. If there exists a unit vector $\mathbf{W}^*$ and a number $\delta > 0$ which partitions the space then the algorithm will only execute "Add" a finite number of times. I.e. the algorithm will determine a suitable vector **W** which will partition the space correctly. Needless to say, the vector determined need not be $\mathbf{W}^*$.

Assume that there is a unit vector $\mathbf{W}^*$ which partitions the space and a small positive constant δ such that $\mathbf{W}^*.\mathbf{X} > \delta$ for all **X** an element of **F**.

Define

$$G(\mathbf{W}) = (\mathbf{W}^*.\mathbf{W})/(|\mathbf{W}|) \tag{1}$$

Note that G(**W**) is the cosine of the angle between **W** and $\mathbf{W}^*$.

Since $|\mathbf{W}^*| = 1$, $G(\mathbf{W}) \leq 1$.

Consider the behaviour of G(**W**) through succesive passes through *Add.*

First consider the numerator:

$$\mathbf{W}^*.\mathbf{W}_{t+1} = \mathbf{W}^*(\mathbf{W}_t + \mathbf{X})$$
$$= \mathbf{W}^*.\mathbf{W}_t + \mathbf{W}^*.\mathbf{X}$$
$$= \mathbf{W}^*.\mathbf{W}_t + \delta$$

Hence, after the nth application of *Add* :

$$\mathbf{W}^*.\mathbf{W}_n = n\delta \qquad (2)$$

Now, consider the denominator. Since $\mathbf{W}^*$.**X**must be negative (for *Add* to be executed)

$$|\mathbf{W}_{t+1}|^2 = \mathbf{W}_{t+1}.\mathbf{W}_{t+1}$$
$$= (\mathbf{W}_t + \mathbf{X}).(\mathbf{W}_t + \mathbf{X})$$
$$= |\mathbf{W}_t|^2 + 2\mathbf{W}_t.\mathbf{X} + \mathbf{X}^2$$

However, **W.X** must be negative and $|\mathbf{X}|^2 = 1$, so

$$|\mathbf{W}_{t+1}|^2 < |\mathbf{W}t|^2 + 1$$

and after the nth application of *Add*

$$|\mathbf{W}_n|^2 < n \qquad (3)$$

Combining (2) and (3)

$$G(\mathbf{W}_n) = (\mathbf{W}^*.\mathbf{W}_n)/|\mathbf{W}_n|^2 > n\delta/(n^{1/2})$$

But G(**W**) $\leq$ 1, so $n \leq 1/\delta^2$

So, regardless of δ, *n* (the number of times *Add* is executed) will always be finite. Thus the algorithm will partition the space correctly if such a partioning exists.◊

APPENDIX B

DELTA RULE AS A GRADIENT DESCENT METHOD

From [RM86] (p322-324)

The delta rule for updating the weights following the presentation of input-output pattern pair *p*

$$\Delta_p w_{jk} = \eta(t_{pj}-o_{pj})i_{pk} = \eta\delta_{pj}i_{pk} \tag{1}$$

where t_{pj} is the j^{th} component of the target output for pattern p, o_{pj} is the j^{th} component of the actual output for pattern *p* and i_{pi} is the i^{th} component of the input vector for pattern *p*.

minimizes the squares of the differences between the actual and desired output values summed over all the output units and input-output pattern pairs, provided that a linear activation function is used. One method to show this is to show that the derivative of the error measure with respect to each weight is proportional to the weight change dictated by the Delta rule with a negative constant of proportionality.

PROOF

Define

$$Ep = \tfrac{1}{2}\sum_j (t_{pj}-o_{pj})^2 \tag{2}$$

to be the error measure on input-output pattern p and

$$E = \sum_p E_p$$

to be the total error measure.

To show that the Delta rule implements gradient descent on E, it must be shown that

$$-\partial E_p/\partial w_{jk} = \delta_{pj} i_{pk} \alpha \Delta_p w_{jk}$$

With no hidden units and using the chain rule

$$\partial E_p/\partial w_{jk} = \partial E_p/\partial o_{pj} . \partial o_{pj}/\partial_{jk} \qquad (3)$$

From (2)

$$\partial E_p/\partial o_{pj} = -(t_{pj} - o_{pj}) = -\delta_{pj} \qquad (4)$$

But, since linear activation functions are used

$$o_{pj} = \sum_k w_{jk} i_{pk}$$

i.e. $\partial o_{pj}/\partial w_{jk} - = i_{pk}$ (5)

Substituting (4) and (5) into (3)

$$-\partial E_p/\partial w_{jk} = \delta_{pj} i_{pk} \qquad (6)$$

as desired.

Noting that

$$\partial E/\partial w_{jk} = \sum_p \partial E_p/\partial w_{jk}$$

the net change in w_{ji} after one epoch is proportional to $\partial E/\partial w_{jk}$ and thus the Delta rule performs gradient descent in E.◊

The above proof is only strictly true if the values of the weights are not changed during the epoch. By keeping the learning rate sufficiently small, on-line updating can be used and the departure from true gradient descent will be negligible. Using a small enough learning rate the Delta rule will determine a set of weights that minimize this error measure.

APPENDIX C

The high-level algorithms for Backpropagation (BP), Quickpropagation (QP) and Backpercolation (Perc) are given in this appendix. Note that only the "bare bones" of the algorithms are given. For the Perc algorithm, much detail has had to be omitted due to the proprietary nature of the algorithm. The algorithms have been written so as to increase legibility rather than for efficiency.

Notation:

Unless otherwise specified the following notation will be used:

- $Weight_{i,j}$ is the weight from unit j to unit i.
- X_i is the ith component of the input vector,
- $Y_{i,j}$ is the ith unit in the jth layer.
- O_i is the ith component of the output vector.
- D_i is the ith component of the desired output vector.
- The sigmoid non-linearity is $f(x) = 1/(1+e^{-x})$
- The threshold is considered to be the weight from an additional input unit which has a value of 1 as mentioned in chapter 2 and 3 .
- Assume L layers of weights (including the input layer) numbered 0, 1, ... L-1.

BP ALGORITHM

This algorithm excludes the momentum term and assumes on-line updating is used (i.e. the weights are updated after each pattern pair has been presented). It is a trivial task to accommodate these in the given algorithm. For example, batch learning can be accomplished by accumulating the changes required for the weights in an array (each weight having one unique entry in the array) and then to update the weights after the epoch is complete using the accumulated values.

Algorithm

```
/* Step 1: Initialize all weights and offsets (thresholds) to small random numbers. */
/* Step 2: Present the input vector and the desired output vector. */
/* Step 3: Use the sigmoid non-linearity to calculate the actual output vector. */

for (K = 1; K ≤ L; K++)
{
    /* For each layer, starting at the first hidden layer and moving to the output layer. */
    for (I = first unit in layer K; I ≤ last unit in layer K; I++)
    {
        Sum = 0;
        for (J = first unit in layer (K-1); J ≤ last unit in layer (K-1); J++)
            Sum += $Weight_{I,J}$;
        $Y_{I,K}$ = f(Sum);
    }
}
/* Copy the appropriate values to the output units. */
for (I = 0; I ≤ M; I++)
    $O_I = Y_{I,L-1}$;

/* Step 4: Use a recursive method starting at the output layer and working back to the first
hidden layer to adjust all the weights. */
for (K = L-1; K > 0; K--)
    for (I = first unit in layer K; I ≤ last unit in layer K; I++)
        if (K = L-1) /* i.e. the output layer. */)
            $\delta_I = O_I(1 - O_I)(D_I - O_I)$;
        else
        {
            Sum = 0;
            for (K2 = first unit in layer K+1; K2 ≤ last unit in layer K+1; K2++)
                Sum += $\delta_k$ $Weight_{K2,I}$;
            $\delta_I = Y_{I,K}(1 - Y_{I,K})$S um;
        }
for (J = first unit in layer K-1; J ≤ last unit in layer K-1; J++) /*Update the weights. */
    $Weight_{I,J} = Weight_{I,J} + \eta\, \delta_I\, Y'_J$;

/* Step 5: if learning isn't complete go to step 2. */
```

QP ALGORITHM

This algorithm uses the weight update rule as specified by Fahlman and given in chapter three. Note that a gradient descent term is always added to the value calculated by the Quickprop rule except when the conditions outlined in chapter three are satisfied.

Because QP is an extension of BP, only the weight update portion of the algorithm has been given. The only other change to the BP rule is the requirement that the slope values and the weight changes of the previous epoch be available for all the weights must be available at weight update time. μ is the maximum growth factor that determines how large any weight may become.

When the weight update algorithm below is executed, it is assumed that each weight has the following information available to it: its current and previous slope (calculated using the error derivative method in the BP algorithm) and its previous weight change. Therefore it is quite acceptable to update the weights of units in the earliest layers first.

Algorithm

```
/* Weight update step. */

ShrinkFactor = μ/(1.0+μ);
for (K = 1; K ≤ L-1; K++)
{
   for (I = first unit in layer K; I ≤ last unit in layer K; I++)
      for (J = first unit in layer K-1; J ≤ last unit in layer K-1; J++)
      {
         W = Weight_{I, J};
         DW = LastWeightChange_{I, J};
         S = CurrentSlopes_{I, J}+Decay*W;
         /* Decay is a weight decay term used to prevent the weights from increasing too quickly. */
         P = PreviousSlopes_{I, J};
         NextWeightStep = 0;



         if ((S == 0) || (P == 0))
            NextStep += LearnRate*S;
         ellse
```

```
        if ( ( (S ≤ 0) && (P > 0) ) || ( (S < 0) && (P ≤ 0) ) )
        /* The slopes are opposite in sign so use the quickprop rule alone. */
           NextStep += S/(P-S)*D;
        else /* The slopes are in the same direction. */

           if (abs(S) = abs(P)) /* Guard against taking too large a step. */
              NextStep += μ*D+LearnRate*S;
           else /* All ok so use the quadratic and linear terms. */
              NextStep += S/(P-S)*D+LearnRate*S;

        LastWeightChangeI, J = NextStep;
        WeightsI, J = W+NextStep;
     }
}
```

Perc ALGORITHM

Due to the proprietary nature of the Perc algorithm only the barest details may be revealed here. As can be seen it is very similar to BP with an extra step added — the back percolation to assign each weight its own error surface to descend.

Algorithm

Perc can be viewed as an extended BP algorithm.

Framework of Perc

```
/* Step 1: Initialize the weights to small random values. */

while (Training is not complete)
{
   R esetErrorsTo0;
   /* Reset the error counters to 0. */

   for (Pattern = 1; Pattern  number of patterns; Patterns++)
   {
      GetNextPattern(Pattern);
      /* Get the next I/O pattern in the training set. */
      ForwardPass;
```

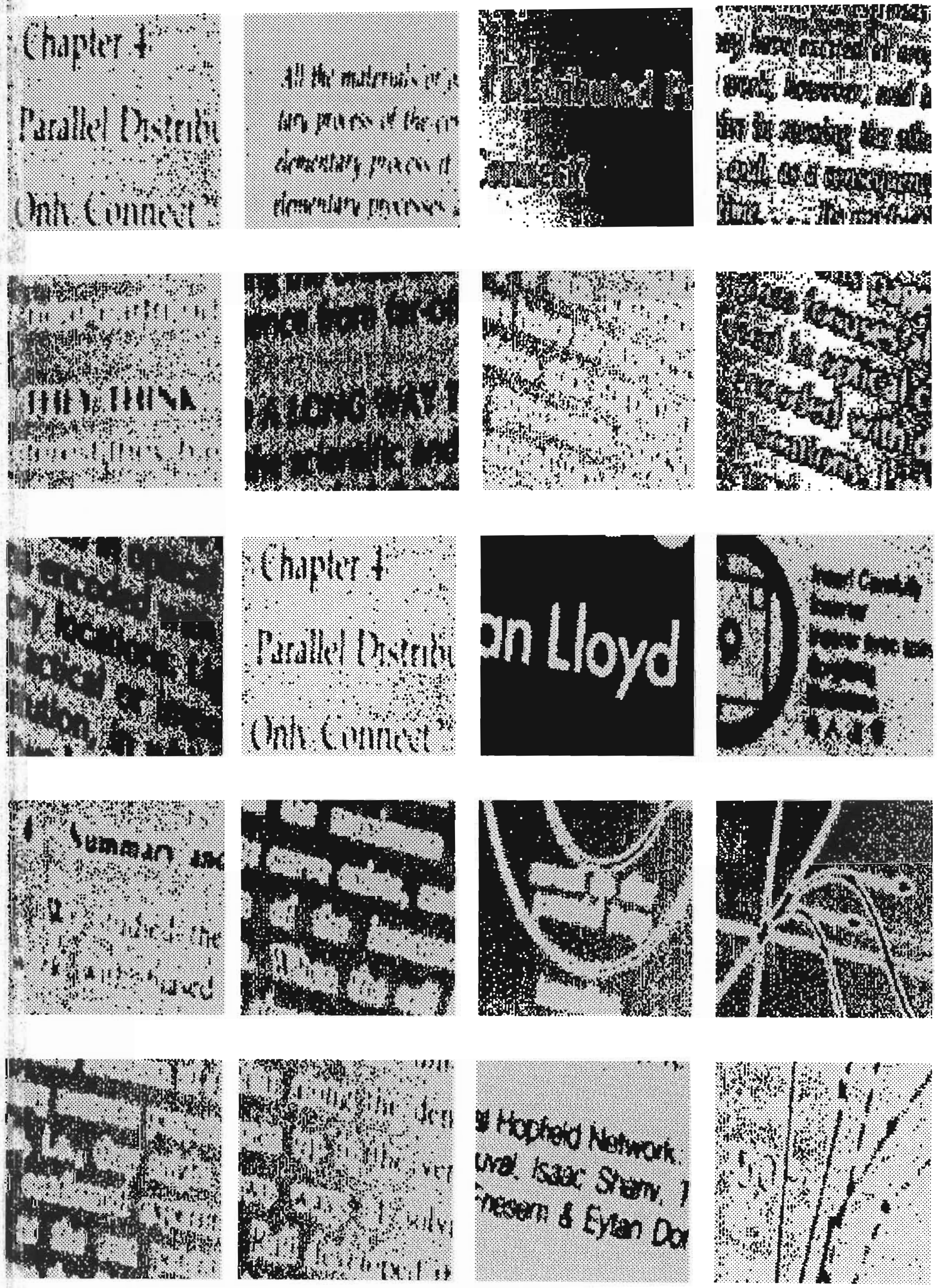
```
        /* Pass the data forward through the net as for BP. */
        RecordErrors;
        /* Record the error statistics used in the back percolation step. */
        BackProp;
        /* Back propagate the error gradients as for BP. */
        BackPerc;
        /* Back percolate the error measures and adjust the weights. */
        AmplifyLambda;
        /* Increase the learning rate if the required conditions are met. */
    }
}
```

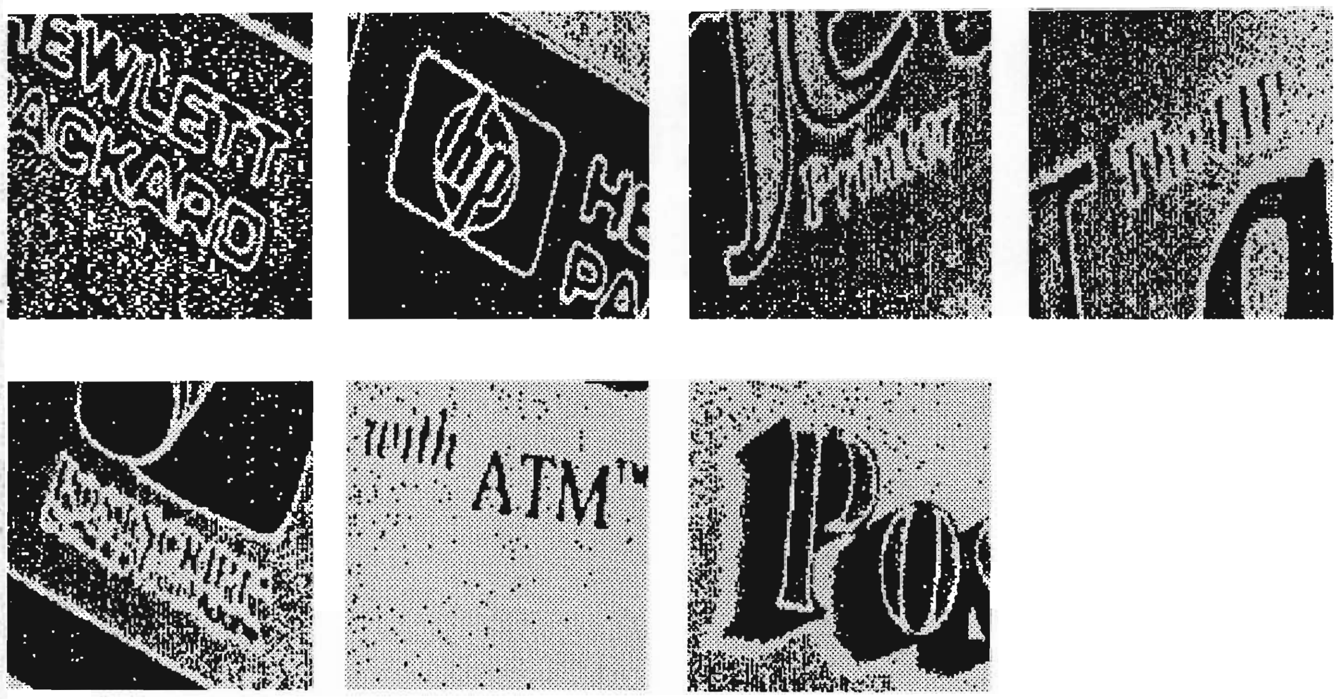
The *BackPerc* step

The only difference in the Perc and BP algorithm is the extra step, *BackPerc.* This step accomplishes the following:
There are **four basic steps** for all units in a layer:
1. Evaluate the optimal internal activation error of each unt.
2. Normalize each units' internal activation error.
3. Evaluate & post the δμ error messages to earlier units.
4. Evaluate & post the δweight weight changes of each unit.

The δμ and δweight error messages are used to calculate the new weight values. Unfortunately due to the proprietary nature of the algorithm, further details may not be revealed.

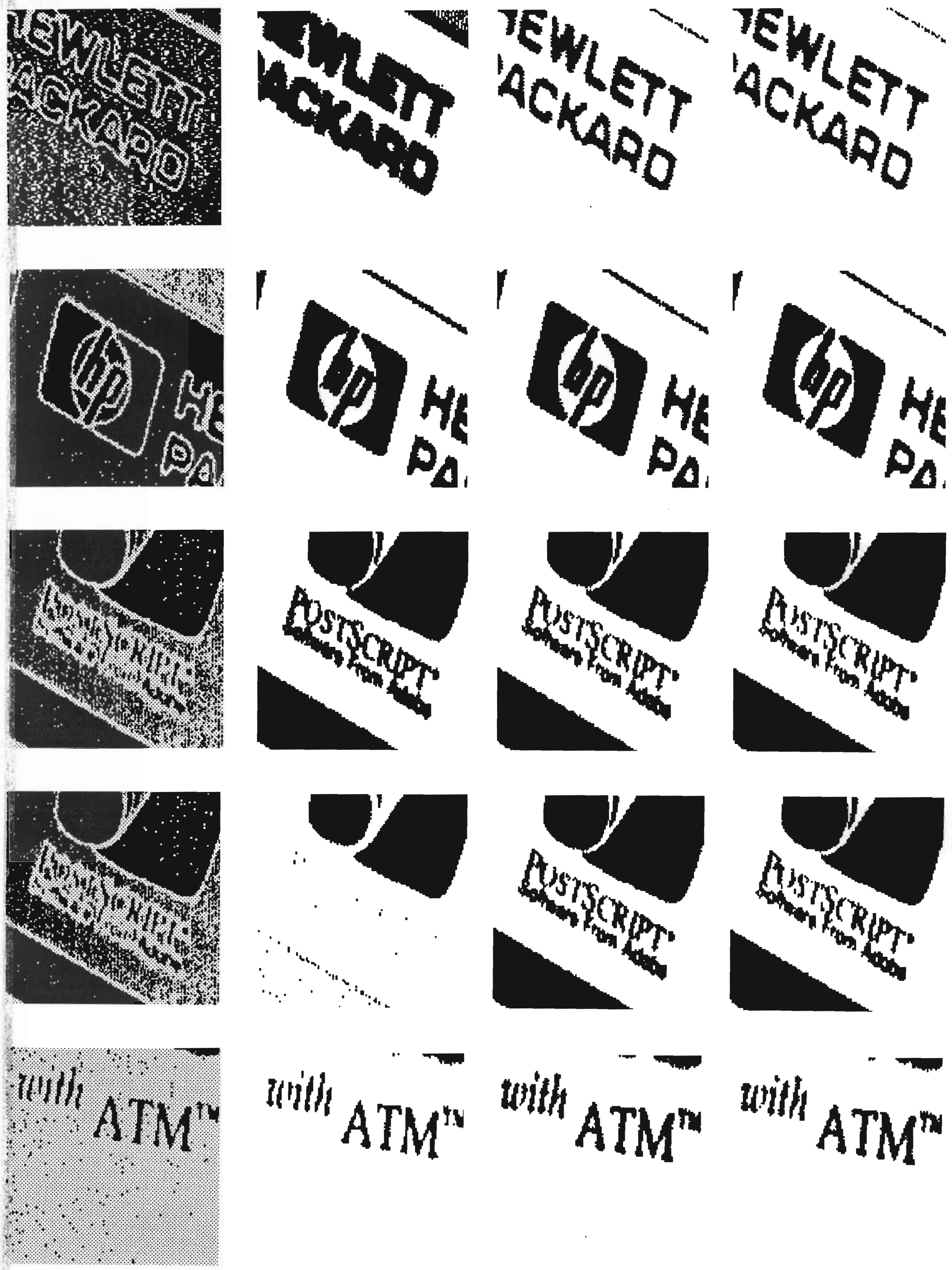
APPENDIX D

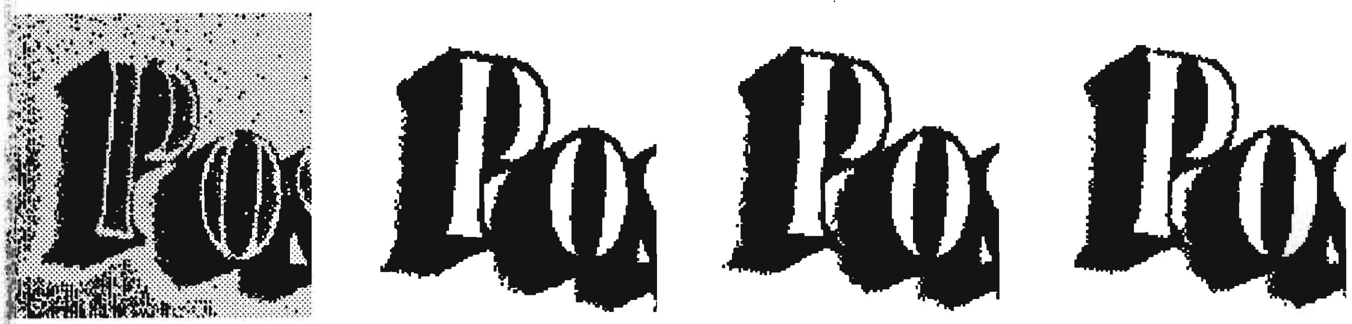

Chapter 4
Parallel Distribu
Only Connect
All the materials or
Distributed P
Chapter 4
Parallel Distribu
Only Connect
an Lloyd
Summary
Hopfield Network
uval, Isaac Shamy
nesem & Eytan Dor



The twenty images making up the training set and seven of the test images.

APPENDIX E

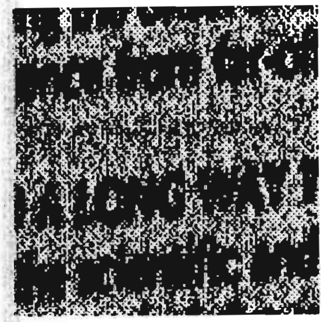
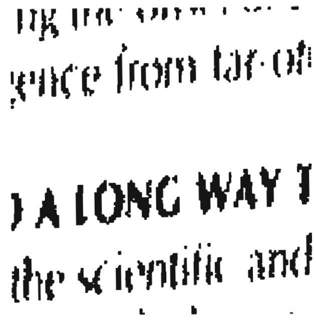

HEWLETT PACKARD
hp
POSTSCRIPT
Software From Adobe
with ATM™



The results of applying the threshold determined by the net (second column), the mean (third column) and the median threshold (fourth column) to the original image (first column) of some of the seven test images appearing in appendix D.

Rows three and four on the preceding page demonstrate the effect the number of hidden units have on the net's ability to binarize images. The image on the third row is successfully binarized because the net used has more hidden units (49) as opposed to that used on the image in row 4 (11 hidden units).

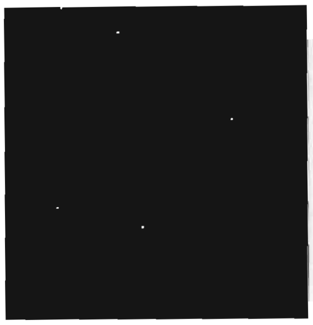
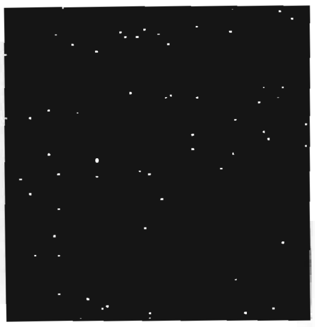
APPENDIX F

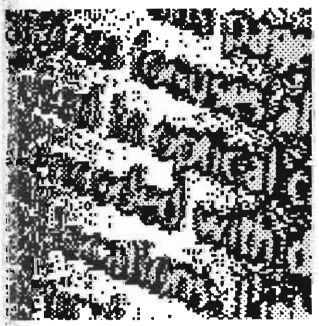



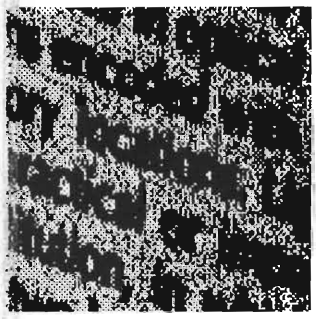
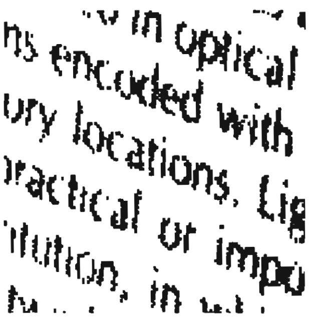
gence from far-of

A LONG WAY T

the scientific and



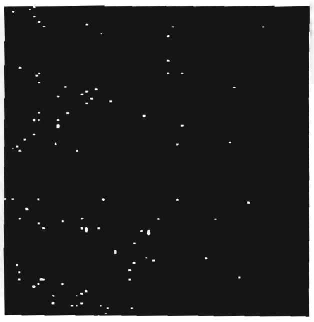
in optical

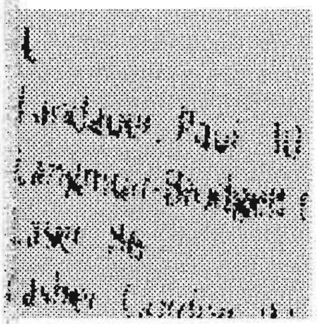
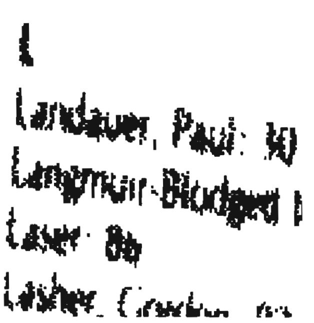
ns encoded with

ury locations. Lig

ractical or impo

itution, in



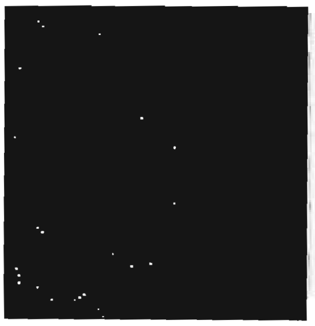
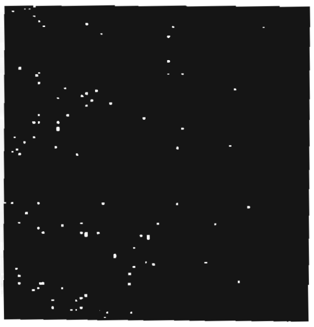
L

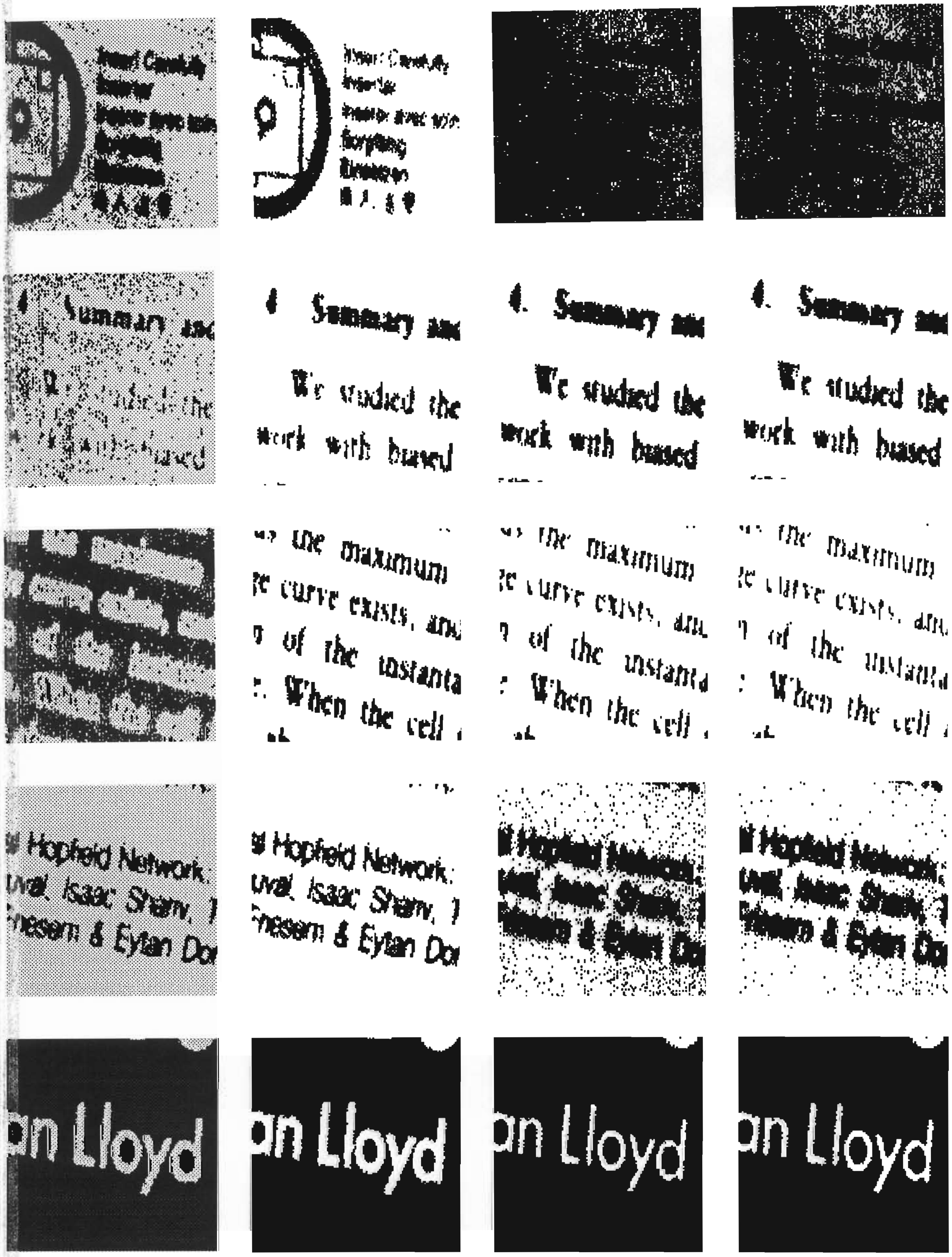
Landauer, Paul: 30

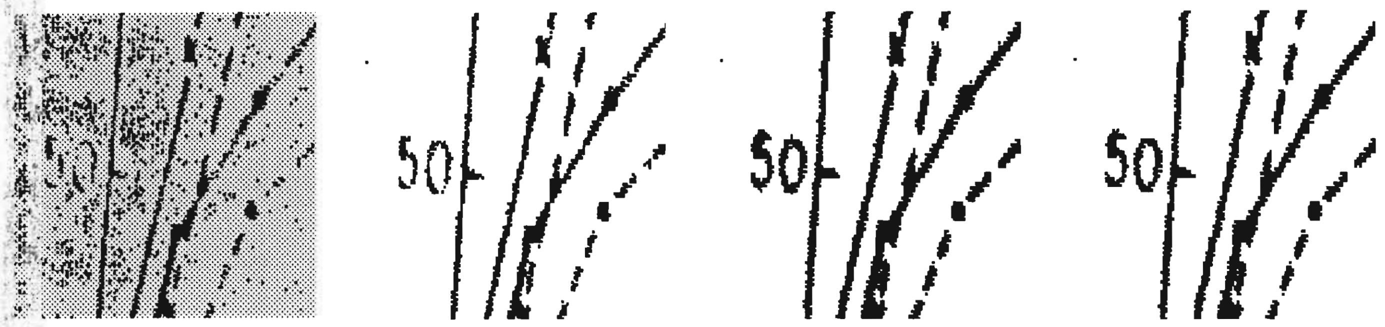
Langmuir-Blodgett

Laser: 86

Lasher, Gordon


4. Summary and
We studied the
work with biased
the maximum
curve exists, and
of the instanta
When the cell
Hopfield Network,
uval, Isaac Shany,
hesem & Eytan Do
an Lloyd



The results of applying the threshold determined by the net (second column), the mean (third column) and the median (fourth coulmn) to the orginal image (first column) of some of the training images given in appendix D. Notice how all three methods fail to binarize the third training image on page F-1.