A Neural Network Implementation on a Transputer System and Applications

by

Andreas Hadjiprocopis

Supervisor: Dr. P.M.Samwell

The City University

London

APRIL 1993

for the sun,

for the sea,

for the destiny,

and all the revolutions of the world.

NO

A NEURAL NETWORK IMPLEMENTATION ON A TRANSPUTER SYSTEM AND APPLICATIONS

Andreas Hadjiprocopis

Abstract

Neural computing, Parallel Distributed Processing, Neuromorphic systems are terms their use increases day by day (becomes more fashionable perhaps?). Their meanings and the scientific work they convey – not known to many – have been undoubtedly discussed widely:

– Fascinating? Yes.

– Mystical? Not so.

– Magical? Sort of.

– 'Space-age'ish? Certainly.

– Alchemical? Rather.

– Promising? Yes!! No!!

Artificial Neural Networks or simply ANN, are computational techniques intended to imitate some of the functions of the human brain using certain of its basic structures.

Theoretically, the many different models coming under the term exhibit some human brain characteristics such as **learning**, **generalisation**, **abstraction** and **association**. In practice, however, the performance of a neural network depends on the *nature* of the application and the *impartiality* of the judge.

This report describes an attempt to develop a powerful and flexible tool, both in **sequential** and **parallel** form, for studying and working with neural networks.

The advantages of using a highly parallel machine to load a computational scheme which is *by definition parallel* are explained, and the technical problems encountered and solutions suggested are considered.

Finally, the applications of the developed software in **medical diagnosis** and **pattern recognition** is described.

This project, in general, was not meant to search for new topologies and learning algorithms nor to modify existing ones. It must be understood that neither time nor resources were sufficient for such an attempt. However, a brief description of a proposed network configuration is given, based on our experience gained during this work.

To avoid any criticism based on the impression that this work was only a matter of writing software using ready-made formulae, I make it clear that an appreciable part of the project was spent in making a profound study of the theory behind neural nets.

Contents

List of Figures

Introduction

As the scientific evolution broadens and deepens, today will not be the same as tomorrow. Yesterday we seemed unbearably ignorant and yet the technological discoveries of our times are taken for granted.

The walk of man from the caves to the moon is full of falls and diversions, each one being a restatement of one of the most enduring fears of mankind: that of dangerous knowledge, that of Faust seeking knowledge not meant for man.

As a new century rises, the dream that the faustian motif gives way to a prosperous–through–science society has, so far, proved to be an **Utopia**. Once again Faust has been destroyed by his mephistophelean nemesis.

Science has been unjustly maligned and technology, a synonym of power, is seen as a means of imposition of the strong on the weak, of the rich on the poor. Technology advances at a rate which will not saturate the market and in the direction imposed by the multi-national corporations' interests, not excluding research in perfecting methods of eliminating man himself. Scientific research is led by businessmen and managers. Imagination, poetry, idealism and dreams are excluded *a priori*, as extremely unpredictable and unproductive.

Those who dream of a society where science and technology are used with imagination and wisdom exclusively to better *everybody's life* and protect our environment are confronted with lack of gravity and accused of being unrealistic and naive. Those who refuse to see science being reduced to a matter of economics, accounting and management are accused of being anachronistic.

If only people could believe that men working wonders do exist ...

The brain is the most complex computing device known to man. It has evolved in size and neurological complexity over many millions of years and still remains a mystery to man himself. All the great thinkers since the Greeks: Democritus, Plato, Aristotle through to Descartes and Broca, have advanced their own theories to explain its operation.

In the 1940's in a joint effort of **biology**, **cognitive studies** and **engineering**, the foundations of **connectionism** were laid down.

In 1943 **McCulloch** and **Pitts** proposed a model for a nerve cell using a threshold device. They showed that such a collection of neurons was capable of *calculating* certain logical functions.

In 1949 **Hebb** pointed the significance of the connections between synapses in the process of learning, and developed a basic learning rule.

It was not before 1958, however, that **Rosenblatt**, putting together the ideas of Hebb, McCulloch and Pitts, described the first operational model of neural networks, the **perceptron**. By arranging the neurons into a network of a particular topology and modifying the connections between synapses, the perceptron could *learn* to calculate logical functions.

This first success, no matter how great it was, should not be a reason for the *excessive* and *unjustifiable* optimism cultivated by the mass media which along with a rather *simplistic* explanation by many raised high hopes and expectations in people's minds which were never to be fulfilled.

It seemed for a time that the key to intelligence and 'thinking machines' had been found; reproducing the human brain was only a matter of constructing a large enough network.

Indeed the perceptron was a great achievement and a perspicuous example of the connectionist theory in practice. Though, it was still *early*; technology had not yet evolved to a degree which would allow certain technical problems to be overcome, and the work was incomplete; lacking strong theoretical background and intense analysis.

In 1969 this first direction of research in Neural Networks was abandoned almost completely following the work of **Minsky** and **Papert** who brought the limitations of the perceptron to light.

Discouraged researchers sought work in other areas of greater promise, government agencies redirected their funding, and interest in Artificial Neural Networks was eclipsed until, according to Minsky:

> "...some powerful convergence theorem will be discovered, or some profound reason for the failure to produce an interesting 'learning theorem' for the multilayered machine will be found."
>
> [Perceptrons, Minsky and Papert, 1969, pp231-2]

and that was not before 1989, some 20 years later.

Considering the enormous success of ANN, later on, Minsky's appraisal was proven *excessively pessimistic*, and no wonder with all that money, government support and recognition involved in research at that time.

At the same time with the evolution of connectionism, a new school of thought started adopting the **symbolic approach** to problem solving. Based on the hypothesis that processes of thought could be modelled using a set of symbols and applying a set of logical transformation rules, this approach had an enormous success in well-defined problem areas. From the beginning it seemed to be, along with connectionism, one of the main techniques towards Artificial Intelligence but after the first disappointment caused by the perceptron, it remained the only one.

However, this symbolic approach suffers from a number of limitations:

(i) Due to the sequential nature of the methods used, as the quantity of data increases the method may suffer from a *combinatorial explosion*.

(ii) The representation of knowledge is *localised* in the sense that one item of knowledge is represented by a precise object.

(iii) In a symbolic system it seems difficult, if not impossible, to simulate the learning process.

The connectionist approach overcomes all three of the above limitations:

(i) The fundamental operation of a neural network is parallel.

(ii) Knowledge representation is distributed in the sense that one item of knowledge is distributed over a number of processing elements.

(iii) The connectionist model exhibits learning as the network modifies its connections according to a predefined rule, using the examples presented to it.

Undoubtedly, the two approaches have their advantages and disadvantages depending on the nature of application. The ideal system must use both: the connectionist approach is good in low level functions such as image analysis and the symbolic approach to combine and supervise different networks at a higher level.

Chapter 1

Neural Networks fundamentals

In 1943 **McCulloch** and **Pitts** proposed a model for nerve cell or *formal neuron*, in great resemblance to the biological neuron. Since then a wide variety of Artificial Neural Networks has been developed. Despite the apparent diversity in network configurations and learning rules, the basic building block is more or less the same: the *artificial neuron*.

In this first chapter we will review some of the existing theories describing the nature of the biological neuron and give an account of a 'primitive' but nonetheless, essential learning rule proposed by Hebb.

Later we will make an attempt to model the biological prototype designing the artificial neuron and studying it closely and in detail.

1.1 The biological neuron

Nerve cells or neurons, are the fundamental elements of the central nervous system. Although their operation is a complicated and not fully understood process, it can be summarised as follows:

- The neuron accepts many inputs which are all added in some fashion.
- If enough active inputs are received at once, the neuron will be activated and 'fire'; if not, the neuron will remain in its inactive state.

As can be seen from figure 1.1 a neuron is built up of three parts:

- **Soma**: This is the *cell body*. It contains the nucleus of the neuron and carries out the synthesis of enzymes and other molecules necessary to the life of the neuron.
- **Dendrites**: These are long, irregularly shaped filaments (similar in shape to a tree whose leafless branches fork and fork again) attached to the **soma**. The dendrites act as the connections through which all the inputs to the neuron arrive.
- **Axon**: It serves as the **output** channel of the neuron. The axon is a non-linear, threshold device producing a voltage pulse (action potential) when the potential within the **soma** rises above a certain critical threshold. The axon terminates in contacts called **synapses**.

Neurons are connected one to another in a complex spatial arrangement. The connection between two neurons takes place at the synapses. In other words the axon of the neuron ends up to a synapse which couples the dendrite of the other cell.

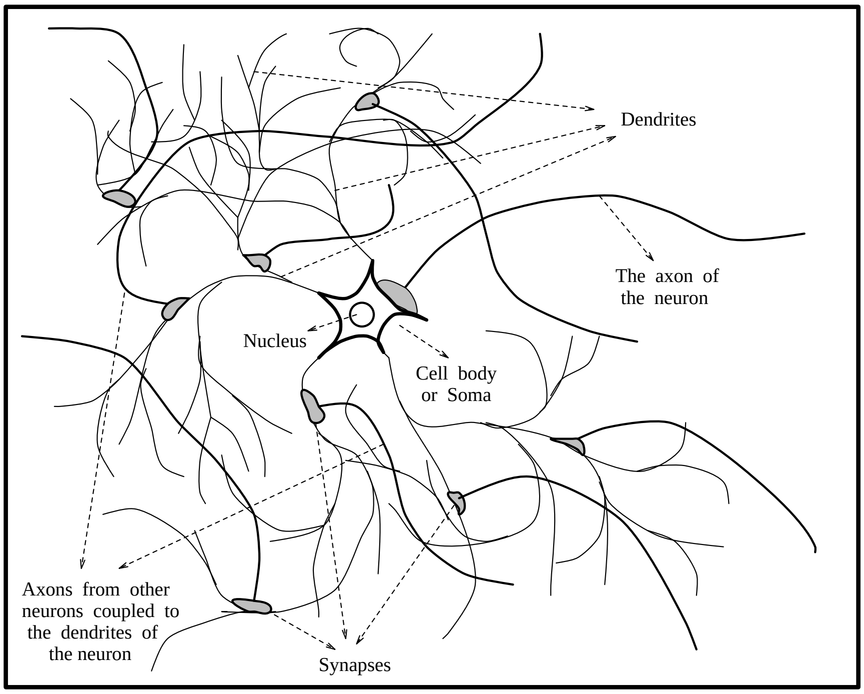

Figure 1.1: **The biological neuron.**

Note that there is no direct linkage across the junction; instead there is the *synaptic gap*. When there is need for transmission the synapse releases chemicals called *neurotransmitters* which diffuse across the gap causing charge ions to flow from the axon of the first neuron to the dendrite of the second.

A very interesting feature of the neuron operation is that some synapses excite the dendrite whilst others serve to inhibit it.

Learning is thought to occur by *modifying* the effective coupling between one cell and another at the synaptic junction. This is achieved by the release of more neurotransmitters, thus the flow of charges increases and the coupling between the two cells strengthens.

From the previous discussion the main features of the brain and its operation are:

- **A highly interconnected network**: Each neuron is connected to up to 10^{14} other neurons.
- **Massive parallelism**: Many neurons operate at the same time.
- **Distributed processing and representation of knowledge**: Information is *non-localised*, meaning that one fact may correspond to activity of a number of neurons. An item of knowledge may be stored in many synaptic gaps.
- **Fault tolerance**: Because of the distributed representation of information damage to a few neurons does not affect the operation of the brain significantly.

- **Learning**: It is exhibited by the adjustment of the coupling effect of two neurons.

1.2 Hebbian learning

On birth, a biological neural system, does not contain any knowledge or abilities[1]. A learning process that takes place over a period of time adds new information to the network – the system is said to be learning.

The question one may ask is:

– "How is new information added to the brain?", or simply,

– "How do we learn?"

An answer is given by a theory suggested by **Hebb** in 1949:

> "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"
>
> [D.O.Hebb, 'Organisation of behaviour', 1949, p50].

According to Hebb a model for *unsupervised* [2] training may be developed in which a synaptic strength between two neurons is increased if both cells

[1]Apart from those we called instinct.

[2]Hebbian rule can also be used in supervised training, slightly modified.

are activated at the same time. In this way often used paths in the network are strengthened. This can be expressed mathematically using the formula:

$$S_{ij}(t+1) = S_{ij}(t) + \beta O_i O_j \tag{1.1}$$

where,

$S_{ij}(t)$ is the synaptic strength between neurons i and j at time t,

O_i, O_j are the outputs of neurons i and j, and,

β is a constant of proportionality.

As an example which illustrates the basic idea behind Hebb's rule consider the classical experiment of **Pavlov**:

Figure 1.2 shows three simplified neurons **A**, **B** and **C**, part of the cerebral cortex of, say, a dog. Neuron **A** is stimulated by the sensory inputs of *sight* while **B** is stimulated by the sensory inputs of *sound*. Both neurons send their outputs to the third neuron, **C**, which is responsible for salivation[1] when the suitable activation is received via the synaptic junctions S_{AC} and S_{BC}.

Suppose that the system works properly in the sense that **C** is activated (and saliva produced) only when **A** is activated by the sight of food. Furthermore, when **A** is inactive, **C** is inactive no matter what the state of **B** is. Now let us allow **A** to fire, causing **C** to fire, by showing some food to the dog. While **C** is still firing, stimulate **B**, by ringing a bell. In this way,

[1] Saliva is produced before eating.

B (sound) participates in the excitation of **C**, although **B** by itself would be insufficient to activate **C**. As the experiment is repeated, the influence of **B** on **C** increases. According to Hebb's assumption, the synaptic strength between neurons **B** and **C** reinforces since they are both active at the same time.

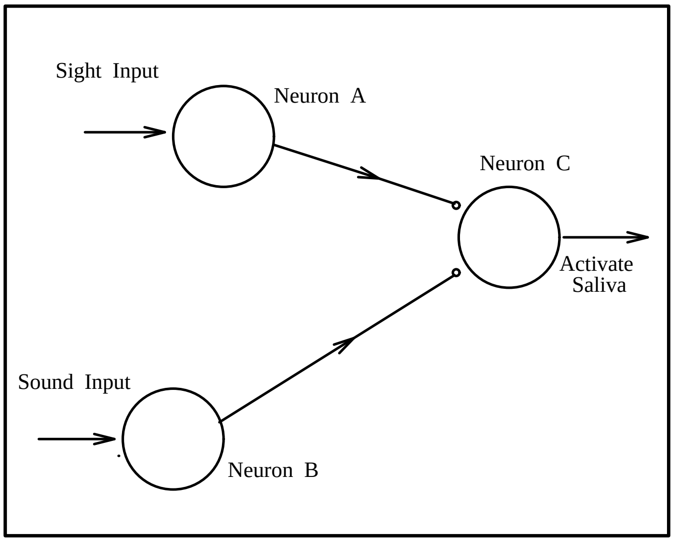

Figure 1.2: **Three simplified neurons used in the Pavlov's experiment.**

Finally the synaptic strength, S_{BC}, will be large enough to allow **B** to influence **C** without **A** being active. This results in the paradox that *saliva is produced when a bell is rung but no food is shown* which confirms Hebb's approach to learning.

1.3 The artificial neuron

From our previous examination of the brain we came to the conclusion that despite the fact that it is a very complex mechanism as a whole, it consists of relatively simple processing elements, the neurons, for which a lot of things are already known, as we have seen in Section 1.1.

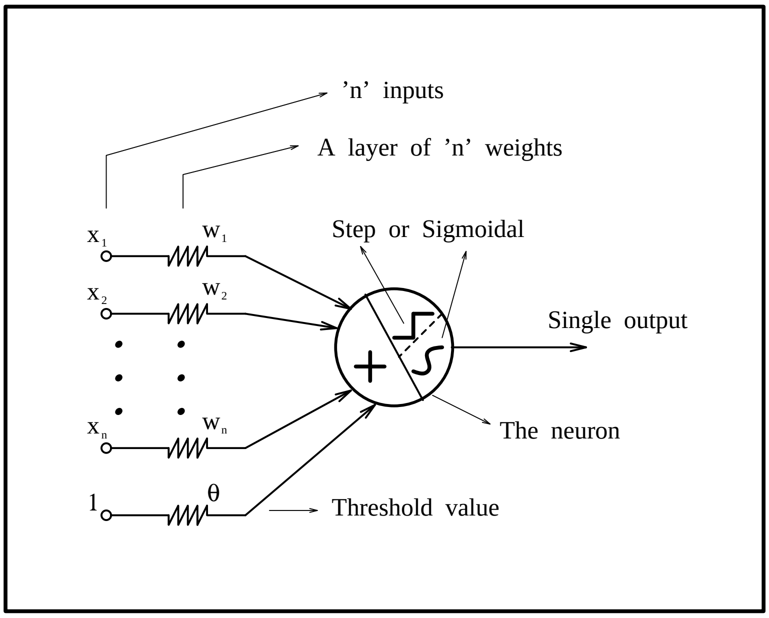

Figure 1.3: **A model of the artificial neuron.**

Following the main idea behind neural computing, which is that of modelling the major features of the brain and its operation, we can produce systems that exhibit similar properties. Our aim is to design a model that will capture the important features[1] of real neurons and at the same time ignore some smaller effects. In this way we hope to produce a simplified version of the brain cells preserving the same general behaviour.

As shown in figure 1.3, a set of n input values, $\underline{\mathbf{X}}$ (similar to the action potential in the biological model), is multiplied by a weight vector, $\underline{\mathbf{W}}$ (synaptic strength), and the product, $\underline{\mathbf{P}} = \underline{\mathbf{X}}.\underline{\mathbf{W}}$, is applied to the input channels (dendrites) of the neuron. The sum of all the weighted inputs is usually further processed by an activation function, $\mathcal{F}$(x), to produce the output signal, o, of the neuron:

$$o = \mathcal{F}(\sum_{i=0}^{n-1} x_i w_i + \theta) \tag{1.2}$$

θ is a threshold value added to the weighted sum, and in some cases is omitted, while in others is considered as the weight value whose corresponding input value is permanently set to 1. This is a more flexible representation which we will adopt,

$$0 = \mathcal{F}(\sum_{i=0}^{n} x_i w_i), \quad x_0 = 1, \ w_0 = \theta. \tag{1.3}$$

This simple model of the artificial neuron is based on certain properties and characteristics of its biological counterpart but ignores many others. For

[1]What is important and what is not is a complicated matter. I suspect that important features are those which when ignored the system changes its behaviour.

example it does not take into account time delays that affect the dynamics of the system. More important, it does not include the effects of synchronism or the frequency modulation functions of the biological neuron.

Another thing to note is that the biological neuron has binary output, it is either on or off. Our artificial neuron can have both binary or continuous output. This depends entirely on the activation function, $\mathcal{F}(\mathrm{x})$, used.

1.4 The activation function

The activation function is very important to the behaviour of a neural network since it defines the output of the individual neuron and therefore the way information is conveyed.

The neural model proposed by McCulloch and Pitts uses one of the simplest activation functions. It is a *step threshold* function and produces a binary output, on or off, similar to real neurons.

A step threshold function has associated a value (the threshold) according to which it classifies an input value as '0' if it is less than the threshold or '1' otherwise, as shown in figure 1.4. This type of function although compatible to the biological neuron's property of the *two-state output*, proves unsuitable when it comes to learning algorithms.

Although learning corresponds to strengthening the connections between active units, as explained in Section 1.2, using a step threshold activation function, it is impossible to strengthen the correct connections, by the correct amount. The reason being that the two-state output gives no indication of the scale by which we need to adjust the weights since we have no idea what the error is! In effect, this activation function *removes* the information needed for the network to adapt and the network is unable to determine what changes should be made in order to produce a better solution next time. This is known as the *credit assignment* problem, and is a characteristic difficulty encountered on binary neurons and architectures based on them.

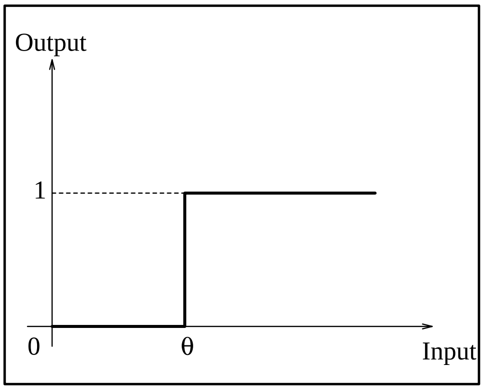

Figure 1.4: **The activation function used in the McCulloch and Pitts model.**

Although the Hebbian learning rule identifies *when* the connection between two neurons should be strengthened, it does not specify by what *amount* this should be done. In my opinion this is impossible in a two-state system unless the output is not just 'on' or 'off' once, but uses a trail of pulses in some kind of *encoding* or *modulation* to transmit all information concerning errors etc. But again this could be wrong because in a massive network, like the brain, some other factors should be taken into account and the binary output could prove more suitable.

On the other hand, the current approach of weight modification, although so far satisfactory, might be totally wrong[1].

The way around the difficulty imposed by the use of the step threshold function is to modify it by:

- eliminating the discontinuity at x $= \theta$, using the *linear* activation function as in figure 1.5(a),

- adding some non-linearity, a *sigmoidal* function for example, as in figure 1.5(b).

The first modification gradually leads us to the second one. The result is a neuron model whose output is normalised and related directly to its input value. This will allow us to calculate a measure of the error between expected and obtained outputs. Making decisions on how to modify various parameters in order to minimise the error is, therefore, easier and more flexible.

We will not deal with the linear function any further since we viewed it merely as a transition from the step threshold to the sigmoidal. Instead we will concentrate on the latter, as it is very widely and successfully used.

A wide range of functions have the properties of the sigmoidal function in its form:

[1] Many things in life suit us but they are not totally correct. For example it took a long time for people to get used to the taste of raw meat. When they finally kind of liked it, somebody clever invented hamburgers.

– Its output is continuous within -1 and 1 or 0 and 1.

– It is continuous differentiable.

– It is monotonic, non-linear, symmetric...

Of those functions, two are very popular:

$$\mathcal{F}(\mathrm{x}) = \frac{1}{1+e^{-x}}, \text{ and } \mathcal{F}(\mathrm{x}) = \tanh(\mathrm{x}).$$

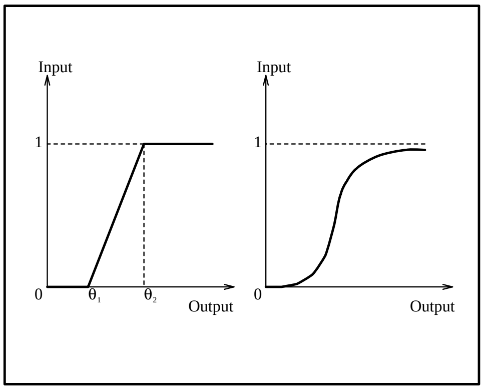

Figure 1.5: **The suggested activation functions: linear and sigmoidal.**

1.5 Supervised and unsupervised training

Training, in general, can be either *supervised*, where a target output is given for a set of inputs and success assumes that the student provides the correct output for the corresponding input, or *unsupervised* where only a set of inputs is supplied and the student has to classify them, extracting any statistical properties, according to some internal representation.

In supervised training each input vector is associated with a corresponding target (expected) output vector. The network is trained over a number of these training pairs using the following general algorithm:

- Present an input vector to the network
- Work out the real output vector
- Obtain a measure of the discrepancy between the target and real output values.
- Modify network parameters in order to decrease that error.

This procedure is often compared to a system where a teacher presents an input and following the above algorithm tries to imitate exactly that pattern of behaviour. No need to mention that this teaching mechanism is not likely to exist in a biological system. However, it bears great resemblance to the way a 'thing', a man, a dog, a parrot learns. Take for example a teacher who tries to teach a child the colours. He will present the colours (input vector) to a child a couple of times along with their names (expected output). The child is then expected to call the colours by their names, rewarded for correct answers and punished for incorrect ones.

On the other hand, an infant accomplishes *self-organization* in early development, without the need of any teacher. This leads us to the concept of unsupervised training.

It requires no target output vector, the training set consists solely of input vectors. The training algorithm, developed by **Kohonen** in 1984, modifies the network configuration, to group similar input vectors into classes by extracting the statistical properties of the training set. Another model based on the philosophy of unsupervised learning is the Hopfield network.

Chapter 2

The perceptron

The research of McCulloch and Pitts who defined a synthetic neuron and that of Hebb who proposed a simple rule to explain the process of learning, were sufficient to explain the functionality of the brain *by reference to its processing elements.* After that, researchers were faced with the challenge of trying to explain the functionality of the brain *by reference to its structure.*

The first successful model was proposed by **Rosenblatt** in 1959, in an attempt to explain visual perception from the perspective of a *neurophysiologist.* The machine he designed and built, called the **perceptron**, may be thought of as a pattern recognition device which is not built to recognize a specified set of patterns, but rather that it has some ability to *learn to recognize* the patterns of a set after a finite number of trials.

In this chapter we will explain the principles underlying the operation of the perceptron and discuss its limitations. We will also go through Minsky's arguments against it and try to give some answers.

2.1 Historical

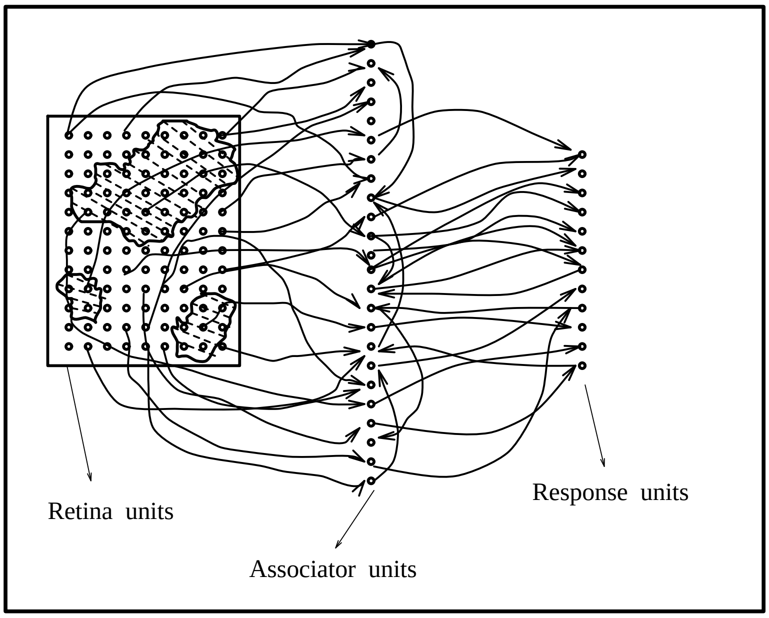

Figure 2.1: **The neural network model proposed by Rosenblatt**

Let us consider the principle elements of the model shown in figure 2.1:

(i) **The Retina**: It is a rectangular array of sensory units or photo-cells which convert a visual input of light and shadow to a suitable form, in an *all-or-nothing* manner according to a predefined threshold value.

(ii) **Associator units**: The sensory units are connected to the associator units which are formal neurons (following the definition of McCulloch and Pitts). These in turn, are connected to the response units[1].

(iii) **Response units**: These units also called **decision cells**, represent the output of the perceptron and operate in the same manner as the associator units. In the general perceptron model, connections between associator and response units can be made in both directions, adding *feedback* to the network.

When a stimulus is presented to the retina of a perceptron, impulses are conducted from the activated sensory units to the associator units. If the total signal arriving at an associator exceeds its threshold, the unit becomes active and sends impulses to the units which is connected. In the same manner, the response units will produce the output of the system to the given visual pattern.

2.2 Perceptron simplified

In order to simplify the study of the behaviour of a perceptron, we will use a more limited model than that defined in Section 2.1, as shown in figure 2.2.

[1] In the *cross-coupled* version, associator units may be interconnected.

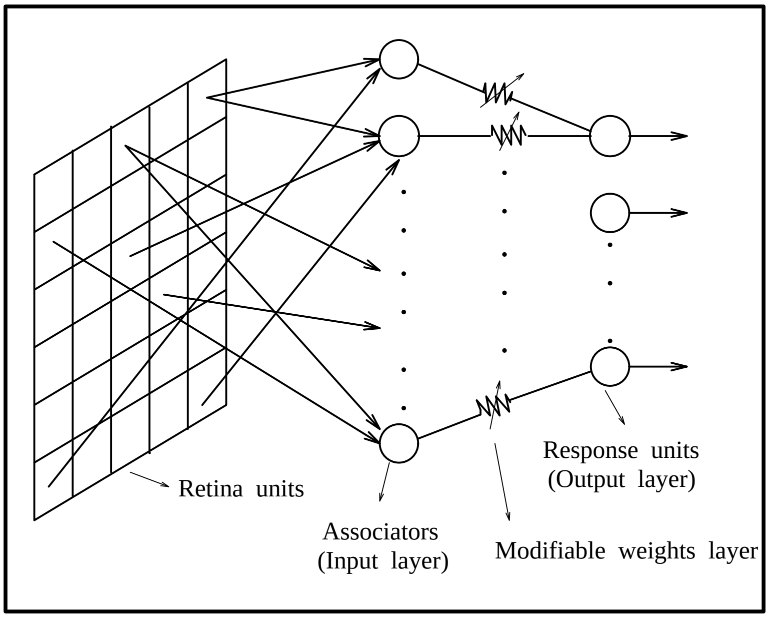

Figure 2.2: **A simpler neural network model based on the perceptron.**

We will make the following simplifications:

- Binary output values (0 and 1) are produced by all cells.
- The decision cells are threshold gates.
- No connections are permitted between two associator or response units.
- Connections between associator and response units are in a single direction (forward), no feedback is allowed.

- The only layer of modifiable weights is between the associator and response units.

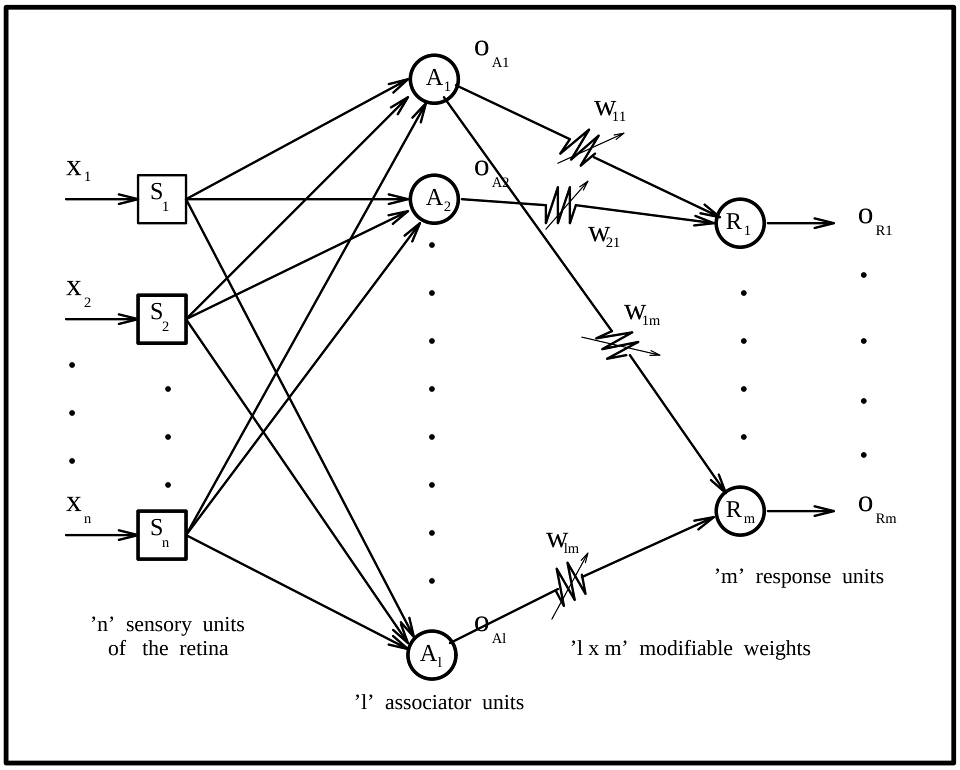

Figure 2.3: **Standard single-layer perceptron architecture.**

In order to construct a mathematical model and explain the operation of the *single-layer* perceptron we have to adopt a simple and flexible terminology as shown in the network of figure 2.3.

The training data set, $\underline{\mathbf{M}}$, consists of q pairs of training vectors ($\underline{\mathbf{X}}$, $\underline{\mathbf{T}}$),

$$\underline{\mathbf{M}} = \{(\underline{\mathbf{X}}^1, \underline{\mathbf{T}}^1), (\underline{\mathbf{X}}^2, \underline{\mathbf{T}}^2), \ldots, (\underline{\mathbf{X}}^q, \underline{\mathbf{T}}^q)\}$$

where,

$\underline{\mathbf{X}}^p$ is the p^{th} set of input values, and

$\underline{\mathbf{T}}^p$ is the p^{th} set of expected (target) output values.

At the beginning of the p^{th} training iteration, a set of input values, $\underline{\mathbf{X}}^p$, is applied to the sensory units of the retina, S_1, S_2, $\ldots$, S_n. which they distribute it to the associator units A_1, A_2, $\ldots$, A_l. The gain of all the paths connecting S and A units is **1**.

The associator units sum their inputs and produce an output vector, $\underline{\mathbf{O}}^p$,

$$\underline{\mathbf{O}}^p = \{o^p_{A1}, o^p_{A2}, \ldots, o^p_{Al}\}$$

the output of the j^{th} associator unit being:

$$o^p_{Aj} = \mathcal{F}(\sum_{i=1}^{n} x^p_i) \tag{2.1}$$

This signal is then transmitted to the response units, R_1, R_2, $\ldots$, R_m, after it has been multiplied by the corresponding weight value, w^p_{ij} connecting units A_i and R_j at the p^{th} training iteration.

The total output, o^p_{Rj}, given by the j^{th} response unit at the end of the p^{th} training iteration is

$$o^p_{Rj} = \mathcal{F}(\sum_{i=0}^{m} O^p_{Ai} W^p_{ij})^1 \tag{2.2}$$

Two things to note are:

- The activation function is a step threshold, that is,

$$\mathcal{F}(x) = \begin{cases} 0 & \text{if } x < \theta \\ 1 & \text{if } x \geq \theta \end{cases}$$

- The weights, w^p_{ij}, are updated at the end of the p^{th} training iteration[2] and stored in a two dimensional vector, $\underline{\mathbf{W}}$, of dimensions $l \times m$.

2.3 Training the perceptron

In order to train the perceptron we will need several iterations during which all training data will be presented. The output obtained will be compared to to the expected (target) output. The weight vector will then be modified in a fashion aiming to minimise the output error, that is the difference between the expected and obtained output values.

The perceptron learns by means of a *supervised learning process*. The main idea behind the algorithm is given below:

[1]The threshold value, θ, is incorporated in the expression as described by equation 1.3.

[2]Refer to Section 2.3 for the learning algorithm.

For the j^{th} response unit (R_j) and the p^{th} training iteration,

if the obtained output, o^p_{Rj}, is equal to the expected output, t^p_j, the weights, w_{ij}, connecting unit R_j and all activation units, A_i remain as they are.

If the two quantities are not equal then:

if $o^p_{Rj} = 0$ and $t^p_j = 1$, then

$$w_{ij}(t+1) = w_{ij}(t) + \beta o_{Ai} \tag{2.3}$$

or, if $o^p_{Rj} = 1$ and $t^p_j = 0$, then

$$w_{ij}(t+1) = w_{ij}(t) - \beta o_{Ai} \tag{2.4}$$

where β is a positive constant that controls the adaptation rate, called *rate of learning*.

In a simpler and more general form the perceptron learning rule can be summarised as follows:

For the j^{th} response unit (R_j) and the p^{th} training iteration,

$$w_{ij}(t+1) = w_{ij}(t) + \beta \delta^p o^p_{Ai} \tag{2.5}$$

where the term δ^p is termed *delta*, and is a measure of the error,

$$\delta^p = t^p_j - o^p_{Rj} = t^p_j - \mathcal{F}(\sum_{i=0}^{l} o^p_{Ai} W^p_{ij}) \tag{2.6}$$

2.4 Widrow–Hoff: A more advanced learning rule

The original learning rule described in Section 2.3 is not very practical and many times will never converge. This is because in equation 2.5 which describes the rule mathematically, there is no indication of *how wrong* the output was; instead there is a simple 'yes' or 'no' resulting from the two-state[1] activation function used.

In other words, in this training procedure, the network can not use the *discrepancy* between desired and actual output values to guide the learning process.

The innovation is the *delta rule* proposed by **Widrow** and **Hoff** in 1960. It addresses all these criticisms and has a number of interesting features, the main one being that it offers a way for the network to compute and utilize the discrepancies. This is done by calculating the difference between the *weighted input sum* and the expected output.

Equation 2.5 remains the same

$$w_{ij}(t+1) = w_{ij}(t) + \beta\delta^p o^p_{Ai}$$

but the delta term, δ^p, is modified,

$$\delta^p = t^p_j - \sum_{i=0}^{l} o^p_{Ai} W^p_{ij} \tag{2.7}$$

In this way, weight values change taking into account the *amount* of error and not only its *sign*.

[1] A more extensive discussion on activation functions is given in Section 1.4.

2.5 A visualisation of what the perceptron does

A typical problem given to a child could be the following:

> We have *six* kinds of fruits:
> { **oranges, grapes, apples, melons, peaches, bananas** }.
> We decided that in order to distinguish them, we will use the following *five* characteristic properties:
> { **taste, smell, shape, colour, texture** }.
>
> Assuming that the kid is familiar with all *six* of the fruits, we ask the following:
>
> > **Task one**: "If a combination of the *five* characteristic properties is given to you:
> > { **sweet and sour, nice, spherical, orange inside and outside, smooth** }.
> > Can you decide what fruit it refers to?"
> >
> > **Answer:** "Orange."
> >
> > **Task two**: "If a combination of the *five* characteristic properties is given to you:
> > { **sweet and sour, nice, spherical,** *red* **inside, orange outside, smooth** }[1].

[1]This combination is not completely strange to you, but there is something unknown. This could be either because you did not listen very well to the question or because you were simply not trained for that fruit – never came across it.

Can you decide what fruit it refers to?"

Answer: "I think that fruit looks like an *orange*."

A problem similar to the previous, can be assigned to a perceptron neural network in the following way:

Consider the *five-dimensional* vector space $\mathbf{R^5}$. Let the set $\underline{\mathbf{M}}$ contain *six* of its member s:

$$\underline{\mathbf{M}} = \{\ \underline{\mathbf{A}},\ \underline{\mathbf{B}},\ \underline{\mathbf{C}},\ \underline{\mathbf{D}},\ \underline{\mathbf{E}},\ \underline{\mathbf{F}}\ \}$$

where,

$$\underline{\mathbf{A}} = \{\ a_1,\ a_2,\ a_3,\ a_4,\ a_5\ \},\ \in \mathbf{R^5}$$
$$\underline{\mathbf{B}} = \{\ b_1,\ b_2,\ b_3,\ b_4,\ b_5\ \},\ \in \mathbf{R^5}$$
$$\vdots$$

We define a set, $\underline{\mathbf{O}}$, of *six* groups/classes as:
$\underline{\mathbf{O}} = \{$ class **1**, class **2**, class **3**, class **4**, class **5**, class **6** $\}$.

After a *finite* number of training iterations, during which all the *six* sets of data contained in $\underline{\mathbf{M}}$ and their corresponding classes were presented to the network, try the following:

Task one: "If the set belonging to $\underline{\mathbf{M}}$ is presented to the network, would it be able to identify which class it belongs to?"

Task two: "If the following set:

$$\underline{\mathbf{G}} = \{\ g_1,\ g_2,\ g_3,\ g_4,\ g_5\ \},\ \in \mathbf{R^5}$$

> not a member of the training set $\underline{\mathbf{M}}$ is presented to the network and bears resemblance to one of the members of $\underline{\mathbf{M}}$, $\underline{\mathbf{A}}$, will the network classify it in the same class as $\underline{\mathbf{A}}$[1] ?."

The **conclusion** is that the perceptron does nothing more than classifying data according to some experience gained through training over a set *representative of the input space.*

In a more formal way, **a perceptron – and any neural network in general – maps the n-dimensional vector space, $\underline{\mathbf{X}}$, onto the m-dimensional vector space $\underline{\mathbf{O}}$, where 'n' is the number of its inputs and 'm' the number of the output classes**[2]**.**

2.6 A visualisation of classification

As we saw in the previous section, the perceptron can be characterised as a *feature classifier.* Again, with a simple example, let us see how classification is done by the human brain.

> We are given three baskets. One contains apples and oranges all mixed up, while the other two are empty. We are told to separate apples from oranges. We do that by picking up one fruit from the first basket, examine it, classify it either as apple or orange and put it either in the left or the right basket.

[1]Assuming that the training set $\underline{\mathbf{M}}$ is representative of the space **R5** belongs to.

[2]The number of output classes, although depending on the number of outputs of the net, is better kept independent due to the many and different representation schemes.

Another example is the following:

> We have five peas and four beans thrown on a table randomly. We are told to separate them so that someone looking on the table could identify at once a group of peas and a group of beans. This can be done in two ways:
>
> 1. Take all peas to the left and all beans to the right[1], or,
> 2. Don't touch any of the peas/beans; instead draw a line in such a way that to the left of the line there are only peas and to the right of the line there are only beans.

We will study the latter method, although it is more difficult and kind of silly, because the perceptron follows a similar method to separate its output classes.

By drawing a line on the table we are *partitioning* the plane of the table, which is a 2-dimensional space, into two classes:

- The class of beans, and,
- the class of peas.

If we can achieve to separate completely beans from peas with a single straight line, as shown in figure 2.4(a), then the set of data[2] is called *linearly separable.*

[1] This, in my opinion, is a good example of how *self-organised* map work.

[2] It contains a set of input data which is the vertical and horizontal coordinates of the items, and a set of output data which is the property of the item (bean or pea).

Most of the time, however, such a line does not exist – the partitioning is impossible, as demonstrated in figure 2.4(b). In this case the data set is called *linearly inseparable*. Since we are so clever and resourceful, we can overcome the problem by using *many* straight lines or just a single curve. The *single-layer* perceptron is not that clever and fails[1].

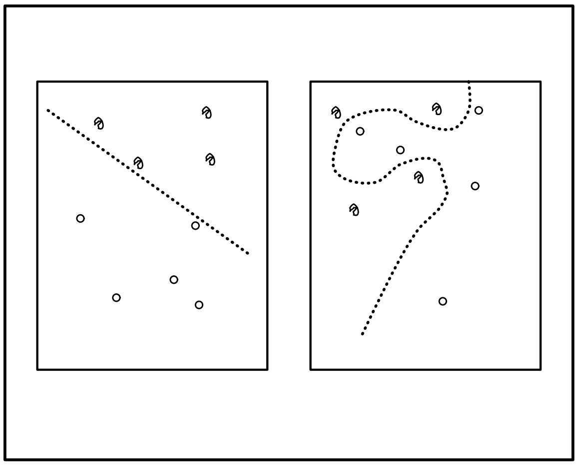

Figure 2.4: **A picture of linear separable and inseparable data sets.**

Now let us try to extend our 2-dimensional problem to the three dimensions, in our world that is. Looking at a lake, we can say that the class of fish under the water is separated from the class of ducks flying in the air by the *plane* of the water surface.

[1]This is due to having only a single processing element. It is like trying to implement a *sum of products* boolean expression with only one layer of logic gates: it is impossible.

Looking at a tennis match we can see that the winner is separated from the looser by the *plane* of the net, and both the players are separated from the spectators by the wire fence.

We can say, therefore, that in three dimensions, linearly separable data sets are separated by a *2-dimensional plane*.

In general, **an n-dimensional function or data set is called linearly separable if its output classes can be separated by the *hyperplane* of dimensions n-1**.

2.7 Limitations of the single-layer perceptron

Despite the impressive results achieved by the single-layer perceptron, there are some severe limitations which *cast down* the myth of 'thinking machines' following the initial success.

The single layer perceptron as proposed by Rosenblatt in 1959 will learn a set of input patterns only if they are selected to form a *linearly independent* set.

However, if this first constraint is ignored, then there is a chance for the net to converge to a solution, only if the training data pairs are *linearly separable*. To meet this last requirement, there must exist at least one set of weights which is used to generate all the desired outputs of the *linearly independent* data.

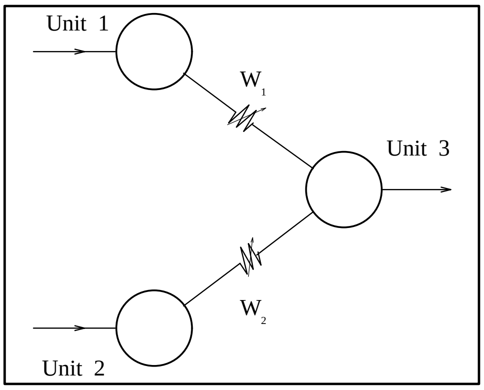

Figure 2.5: **A Neural Network trained for the XOR problem.**

Let us make this point clear using a couple of examples:

Figure 2.5 shows a two input, one output network, with only two weights. We will try to teach it to perform the **inclusive**, (OR), and **exclusive OR**, (XOR), boolean functions.

Example One

In the truth table of the OR boolean function presented in Table 2.1, we can see that the input pattern (1,1) is a linear combination of patterns (0,1) and (1,0). The first constraint, therefore, is violated.

The perceptron, however, will calculate this function perfectly because the second constraint is satisfied:

Apparently, the same weight vector that generates the desired output of patterns (0,1) and (1,0), 'happens' [1] to generate the desired output for pattern (1,1), as well.

Input		*Output*
unit 1	unit 2	unit 3
0	0	0
0	1	1
1	0	1
1	1	1

Input		*Output*
unit 1	unit 2	unit 3
0	0	0
0	1	1
1	0	1
1	1	0

Table 2.1: **Inclusive and exclusive OR truth table.**

Example Two

In the truth table of the XOR function presented in Table 2.1, we can again see that the last pattern is a linear combination of the previous two, and consequently the first constraint is not satisfied.

Unlike the previous example, the perceptron will not calculate the XOR function because the output of (0,1) and (1,0) patterns is 1 and therefore requires *high, positive* weights, while the output of (1,1) is 0 and requires *low or negative* weights. This means that it is impossible for the network to use the same weight vector for all three of the above input patterns.

[1] The reason for that is that the output for (0,1), (1,0), (1,1) is the same: 1.

From the perspective of digital logic design, the boolean expression A·B is a *minterm* itself and needs only one OR gate to be implemented. The expression A⊕B consists of two *minterms*: $\overline{A} \cdot B + A \cdot \overline{B}$. This clearly requires **two** layer of gates, one to implement the two product terms and one to implement the sum.

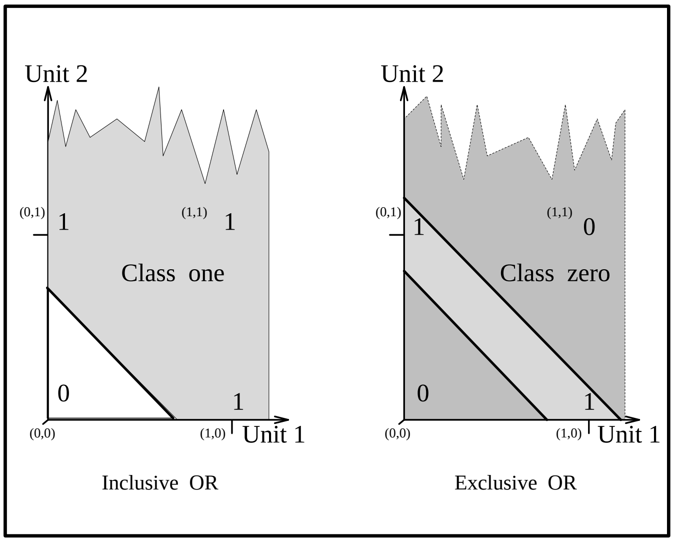

Figure 2.6: **Graphical representation of the OR and XOR data sets.**

To relate the two examples above, with the previous picture of classification, in Section 2.5, consider the diagrams of figures 2.6(a) and (b). They show a graphical representation of the training data used in the examples.

The input space is *2-dimensional* as there are only two inputs. We have two output classes '0' and '1'.

In the first diagram of figure 2.6, we can find *at least* one line that can partition the output space in two. Something similar for the second diagram is impossible with *only one* line.

A simple method to overcome the limitation of the single-layer perceptron, is to *add* some one more input to the network. In this way the data set which previously was linearly inseparable and dependent, now will be adequate, if the right values for the third input are selected.

Input			*Output*
unit 1	unit 2	unit 4	unit 3
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Table 2.2: **Adding one more input to the XOR gate.**

In this way, the problem of the XOR representation in the two dimensions can be solved if we move to the *3-dimensional* space, by adding one more input (unit 4), as shown in Table 2.2.

The input patterns are no longer linearly dependent and the perceptron with *three* inputs will solve the problem, as shown in Figure 2.7. A minor detail though is that we are not dealing with the XOR problem any more.

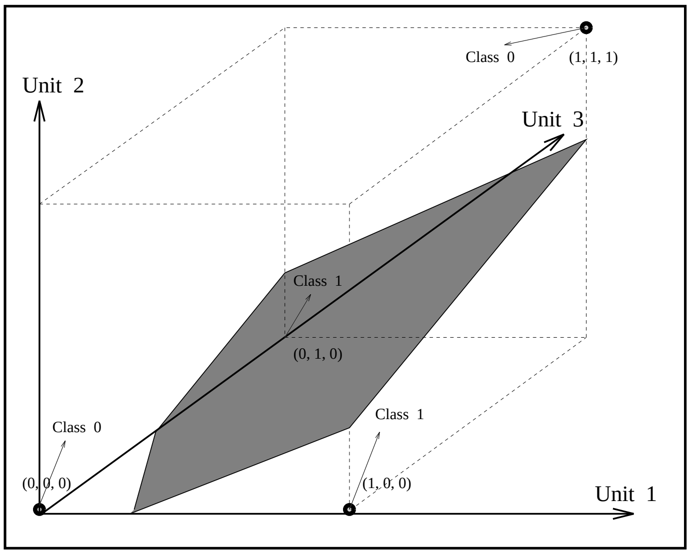

Figure 2.7: **The XOR problem in the 3 dimensions.**

The real solution is to use *two* lines in separating the two classes as in Figure 2.6(b). A *single-layer* network is not capable of doing this sort of partition. A more appropriate configuration is the *multi-layer* neural network which will be studied extensively in Chapter 3.

2.8 Scaling problems and a proposal

Many neural networks are demonstrated by solving problems that are only shadows of real world items – what Minsky and Papert called *toy* problems.

These demonstrations end up, usually successfully, with the same – almost a cliché – conclusion:

> "This is just a demonstration of neural nets using a very small model. If this is successful then think what a hundred times bigger network can do.... It is only a matter of time, therefore until such a huge network be a reality...".

Minsky and Papert claim that many networks suffer undesirable effects when scaled up to a larger size. This is recognized by other researchers too, but it is not a reason to defeat us.

An interesting idea was suggested by **James Freeman**, [1]:

> "It seems plausible to us that the brain is composed of many different parallel distributed systems, performing well defined functions, but under the control of a *serial processing* system at one or more levels. To address the issue of scaling we need to learn how to combine small networks and to place them under the control of other networks...".[pp 29-30]

Such a system will contain many small neural nets each one performing a different task, reporting to a central unit which will combine and analyze these individual results following standard AI techniques. This machine will have the *effectiveness* of a neural network and the *flexibility* of a serial processing AI system.

2.9 The end of an era

"Once upon a time two daughter sciences were born to the new science of cybernetics. One sister was natural, with features inherited from the study of the brain, from the way nature does things. The other was artificial, related from the beginning to the use of computers. Each of the sister sciences tried to build models of intelligence, but from very different materials. The natural sister built models (called neural networks) out of mathematical purified neurones. The artificial sister built her models out of computer programs.

In their first bloom of youth the two were equally successful and equally pursued by suitors from other fields of knowledge. They got on very well together. Their relationship changed in the early sixties when a new monarch appeared, one with the largest coffers ever seen in the kingdom of the sciences: Lord DARPA, the Defence Department's Advanced Research Projects Agency. The artificial sister grew jealous and was determined to keep for herself the access to Lord DARPA's research funds. The natural sister would have to be slain.

The bloody work was attempted by two staunch followers of the artificial sister, Marvin Minsky and Seymour Papert, cast in the role of the huntsman sent to slay Snow White and bring back her heart as proof of the dead. Their weapon was not the dagger but the mightier pen, from which came a book – Perceptrons."

[Seymour Papert: 'One Artificial Intelligence or many?', 1988, p. 3]

In the previous section, the weakness of the single-layer perceptron is that it is not capable of calculating *linearly inseparable* functions was exposed. The significance of this restriction, in practice, is enormous considering the fact that the proportion of linearly separable boolean functions in the set of n-dimensional functions *decreases exponentially* with n.

This was first presented by **Minsky** and **Papert** in 1969. In their book 'Perceptrons' they studied both the potential and limitation of network models[1], and using mathematics they analyze the kind of computation which could or could not be performed with the single-layer perceptron.

The focus of their criticism was that there are functions like *determining parity* in an input pattern or recognizing a *simply connected figure*, that can not be calculated by the network. Although Minsky and Papert realized that these problems could be solved by the addition of an intermediate – hidden – layer between the associator and response units, they raised further doubts:

(i) There are no training algorithms for a multi-layer network[1].

(ii) Even if a training algorithm could be found, there is no guarantee that a larger problem could be handled by simply increasing the

[1]Single-layer networks.

[1]That was in 1969.

size of the network[2].

As a conclusion they offered the totally subjective answer that research on multi-layer networks would be *sterile*.

The inability of networks to solve particular problems was for many investigators only symptomatic of a more general problem. This was regarded as a *mortal blow* to the area of neural networks and the majority of the scientific community walked away. Not much happened in the field after 1969. Research in neural networks paused for almost 20 years until in the late eighties a theoretical foundation was laid upon which the more powerful multi-layer networks of today are being constructed.

[2]The issue of *scaling problems* is dealt with in Section 2.8.

Chapter 3

Multi-layer Networks and Back Propagation

Following 20 years of near eclipse, neural networks revived with the development of *multi-layer* network configurations and powerful learning rules. The *multi-layer perceptron* is a network which is, nowadays, widely used in conjunction with the *Back Propagation* learning rule.

This specific architecture was known long before, it dates back to the late sixties, but the lack of an appropriate learning rule made its application impossible, although the particular merits of its use were recognized.

The Back Propagation algorithm was developed independently by three teams: **Werbos** in 1974, **Parker** in 1982, **Rumelhart**, **McClelland** and **Williams** in 1986. The work of the third team in 1986 signalled the renaissance of the whole subject.

Back Propagation is nothing more than a *generalisation* of Widrow–Hoff's *delta rule* for multi-layer networks. With the use of a *continuous differentiable sigmoidal activation function* instead of the step threshold, as the major innovation, this new learning rule overcomes the *credit assignment* problem[1].

In this chapter we will first make an introduction to the multi-layer networks and then try to visualise how the algorithm searches for the optimum solution.

Later on, the Back Propagation learning rule will be presented and ways of making its use more efficient will be discussed.

From this chapter onwards we are at present time. We are not watching Neural Networks evolving as spectators – we are participating.

[1] Refer to Section 1.4 for a description of the problem.

3.1 Multi-layer Neural Networks

A standard model of the N-layer network is shown in Figure 3.1. It has one *input layer* of n input units[1]. Its purpose is to distribute the input vector,

$$\underline{\mathbf{X}} = \{x_1, x_2, \ldots, x_n\}$$

to all the next layer units.

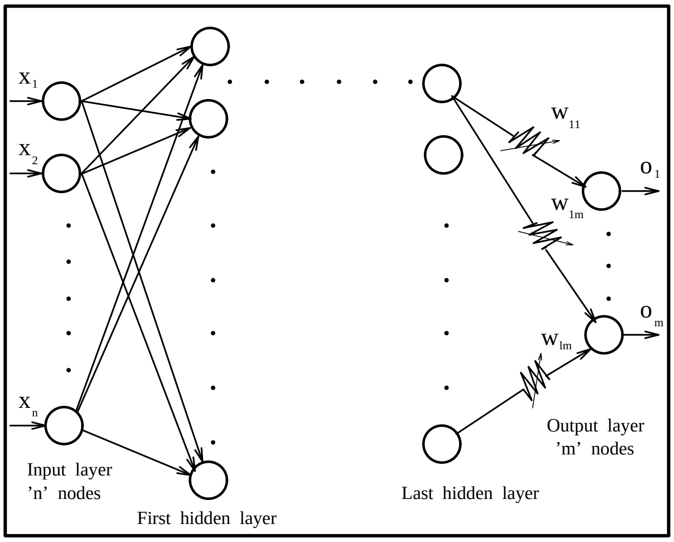

Figure 3.1: **A general multi-layer network.**

There are *N-2 hidden layers* each of them may have any number of units.

[1]Also called nodes.

Finally, there is the *output layer* of m output units which produces the output vector of the net,

$$\underline{\mathbf{O}} = \{o_1, o_2, \ldots, o_m\}.$$

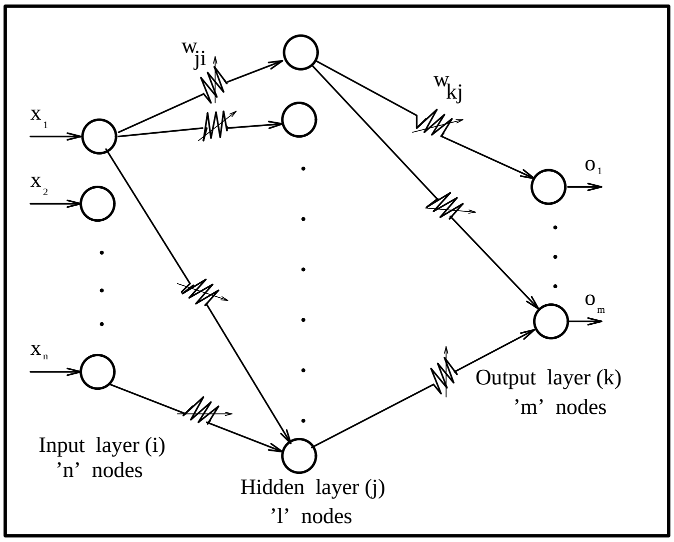

Figure 3.2: **A simpler feed forward network model.**

The network is *fully connected*, meaning that every node of the i^{th} layer is connected to *every* node of the $(i+1)^{th}$ layer. Nodes of the same layer, or nodes of two layers that are not adjacent are *not* connected.

Each node is a neuron as described in Section 1.3 with the difference that the activation function is a *sigmoid*, as suggested in Section 1.4. A *threshold value*, θ, and a *weight vector*, used to store all incoming weights, are associated with each node.

For convenience we will adopt the simplest multi-layer network: that with only one hidden layer, as shown in Figure 3.2. This network will be used in explaining Back Propagation training without any significant loss of generality. This is confirmed by the **Kolmogorov representation theorem**, according to which, *any* function[1] can be represented by a multi layer neural network of *no more than three layers*[2]. Therefore, whatever is done by a four or five layered network, can equally be done by a three-layer network. The theoretical significance of that theorem is that we can simplify our investigation using the three-layer model. In practice, however, a more-than-three layer network can, sometimes, converge faster.

3.2 Visualisation of the training process

Let us identify the problem we have to deal with when we train the network. We have a set of q training vector pairs (input, output),

$$\underline{\mathbf{M}} = \{(\underline{\mathbf{X}}^1, \underline{\mathbf{T}}^1), (\underline{\mathbf{X}}^2, \underline{\mathbf{T}}^2), \ldots, (\underline{\mathbf{X}}^q, \underline{\mathbf{T}}^q)\}$$

where,

[1] The term *function* is used in its more general meaning of a mapping of an input space onto an output space, as described in Section 2.5.

[2] That is input, output and a hidden layer.

$\underline{\mathbf{X}}^p$ is a vector of the *n-dimensional* space, $\mathbf{R^n}$, and represents an input pattern applied at the n input nodes of the network,

and,

$\underline{\mathbf{T}}^p$ is a vector of the *m-dimensional* space, $\mathbf{R^n}$, and represents the **expected** output values when $\underline{\mathbf{X}}^p$ is applied at the input, according to some functional mapping,

$$\underline{\mathbf{T}}^p = \Phi(\underline{\mathbf{X}}^p) \tag{3.1}$$

Our aim is to train the network so that it will learn the mapping of equation 3.1, provided that the training pairs are **selected properly** and are **sufficient**. For many reasons sometimes such a mapping is not possible and we compromise with the approximation:

$$\underline{\mathbf{Y}}^p = \Phi'(\underline{\mathbf{X}}^p) = \Phi(\underline{\mathbf{X}}^p) + \underline{\mathbf{E}}^p \tag{3.2}$$

where $\underline{\mathbf{Y}}^p$ is the obtained output vector and $\underline{\mathbf{E}}^p$ is the discrepancy between the expected and *obtained* output values, termed *error vector*. A measure of the error in a scalar form, which is more convenient, is given by:

$$E^p = \sum_{k=1}^{m} (t_k^p - y_k^p)^2 \tag{3.3}$$

training consists of nothing more than minimising this error quantity.

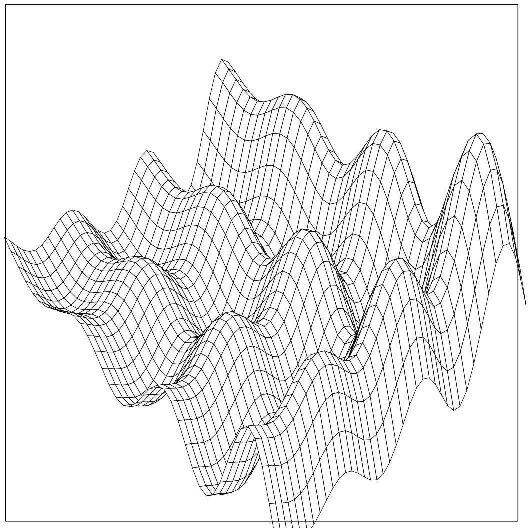

Figure 3.3: **A 3-dimensional error surface**

E^p depends on y_k^p which in turn, depends on the input vector $\underline{\mathbf{X}}^p$, which is arbitrary, and the current weight values stored in $\underline{\mathbf{W}}^p$. By modifying these weights we can increase or decrease the value of error, for a given training pair, as

$$E^p = \mathcal{F}(\underline{\mathbf{W}}^p) = \mathcal{F}(w_1^p, w_2^p, \ldots, w_r^p)$$

An *error surface* can be constructed, by plotting values of E^p with all the weights varying (each weight in one axis). For r weights we can have an *(r+1)-dimensional* error surface, in $\mathbf{R^{(r+1)}}$

This, however, is not very practical since we can not see what the surface looks like. Simplification is possible if we assume only two weights. In this case, the error surface would be 3-dimensional and the whole plot can be accommodated in three dimensions as shown in Figure 3.3

The error surface, $\xi(\underline{\mathbf{W}})$, is a rippling landscape of hills and valleys. Points of **minimum** error correspond to the **wells** and points of **maximum** error to the **peaks**.

Each possible solution is represented as a hollow or a *basin of attraction* in the landscape, and corresponds to a certain weight combination $(w_1, w2)$ which gives a correct output for a given input.

3.3 The gradient descent search

At this point it would be convenient to make one more simplification concerning the shape of the error surface, $\xi(\underline{\mathbf{W}})$, in three dimensional space. Although we understand that $\xi(\underline{\mathbf{W}})$ is different from application to application and its shape is very complicated, full of wells and mountains, let us assume for a moment that the error surface happened to have only one basin of attraction, and more or less resembles the shape of a *concave upwards, paraboloid*, as shown in Figure 3.4.

By assigning *random* weight values at the beginning[1] we get an initial error E_0 and a point on the surface of the paraboloid.

For the next weight selection we have an *infinite* choice of values corresponding to an infinite number of points on the error surface. The question is **"How will we select the appropriate weight vector combination**

[1] We do not care for a good first guess.

which will bring us closer to the solution?". In other words, we need to find a point which is closer to the centre of the basin of attraction and jump onto it.

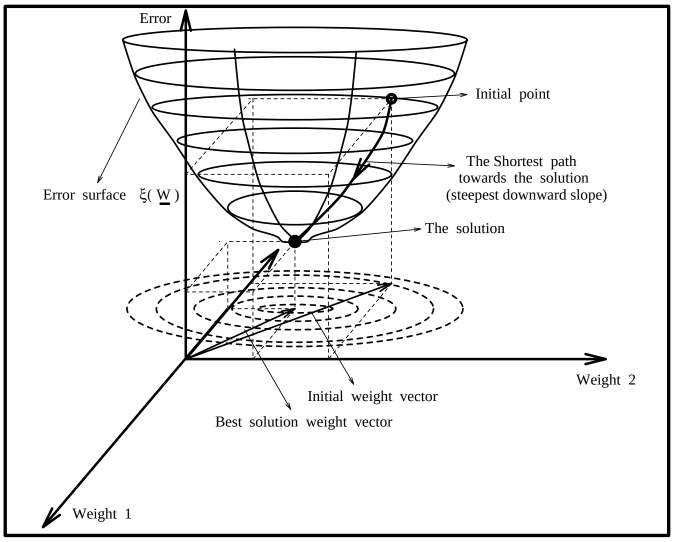

Figure 3.4: **A 3-dimensional error surface.**

If we could manage to select every time a set of weights such that we come closer and closer to the solution, the path followed would be directed *downhill*, like the path followed by a ball left to slide down on the inside surface of a deep bowl. As an *empirical* rule based on observation:

> "Given a point on the error surface, E_0, a point giving a smaller error value $E_1 < E_0$ would be one lying on the line connecting the previous point and the centre of the basin of attraction."

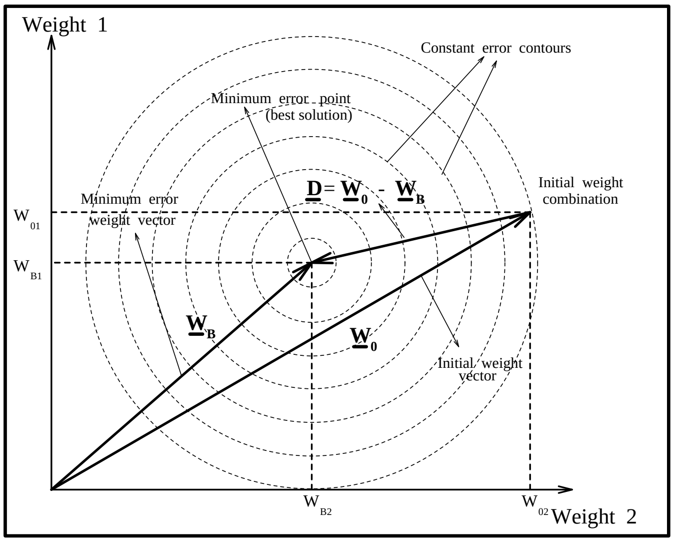

Figure 3.5: **Error contours graph.**

The direction of this line is the direction of the **steepest downward slope** or the negative gradient of $\xi(\underline{\mathbf{W}})$ at that point.

This can be verified from Figure 3.5 which shows the ground plan of the error surface of Figure 3.4. Each contour represents a constant error value. The error is minimum at the centre of the contours.

The direction of the vector connecting our initial point and the minimum error point is that of the steepest descent (the negative gradient of $\xi(\underline{\mathbf{W}})$). This vector, $\underline{\mathbf{D}}$, is. perpendicular to the contour lines and represents the distance between our initial point and the solution.

Any weight change should be in the direction of $\underline{\mathbf{D}}$:

$$\delta\underline{\mathbf{W}} = \beta \cdot \underline{\mathbf{D}}.$$

We know that $\underline{\mathbf{D}}$ is in the same direction as the negative slope of $\xi(\underline{\mathbf{W}})$, so

$$\underline{\mathbf{D}} = -\nabla\xi(\underline{\mathbf{W}})$$

then,

$$\delta\underline{\mathbf{W}} = -\beta \cdot \nabla\xi(\underline{\mathbf{W}}).$$

Therefore a better weight combination must be,

$$\underline{\mathbf{W}}(t+1) = \underline{\mathbf{W}}(t) - \beta \cdot \nabla\xi(\underline{\mathbf{W}}). \tag{3.4}$$

As a first conclusion:

> **From a given initial weight vector, a better solution is obtained by modifying the weights in the direction of the negative slope of the error surface at that point.**

As a second conclusion:

> **From a given initial weight vector, the distance to the next point will depend only on the value of β. if it is too big, our descent is *fast* but there is the danger that after a while we jump from one side of the error surface bowl to the other, missing the solution which is underneath. It is good practice to assign β a large value in the beginning and decrease it gradually as we move towards the solution.**

The above two conclusions although drawn from the simplified version of the minimum error search problem, can be used with any error surface, no matter how complicated it is, and any number of weights.

3.4 Back Propagation training

As the name suggests, Back Propagation trains the network on a predefined set of input–output example pairs using a *forward*[1] propagation of the input patterns and a *back propagation*[2] of the corresponding error signal.

This *two-phase propagate-adapt* cycle is summarised below:

Given a set of q training vectors,

$$\underline{\mathbf{M}} = \{(\underline{\mathbf{X}}^1, \underline{\mathbf{T}}^1), (\underline{\mathbf{X}}^2, \underline{\mathbf{T}}^2), \ldots, (\underline{\mathbf{X}}^q, \underline{\mathbf{T}}^q)\}$$

as defined in Section 3.2,

- An input pattern, $\underline{\mathbf{X}}^p$, is applied to the input layer of the network and is propagated in the forward direction through each layer until an output is generated.
- The output vector, $\underline{\mathbf{Y}}^p$, is compared to the target output, $\underline{\mathbf{T}}^p$, and an error signal, e_k^p is computed, for each output node.

[1] From the input to the output layer.

[2] From the output to the input layer.

- The error signals are then transmitted in the opposite direction. In this way every node in a layer will receive an error signal from all the nodes in the next layer, based roughly on the relative contribution the unit made to the original output.
- Each unit updates its weights, relative to the error signal received.
- The process is repeated with the next input pattern p+1.

Skipping the proof of the following formulae, which can be found in Appendix A, we move to the mathematical model of the Back Propagation algorithm applied to the network of Figure 3.2.

Apply an input vector

$$\underline{\mathbf{X}}^p = \{x_1^p, x_2^p, \ldots, x_n^p\}$$

to the input layer (layer i) nodes.

Propagate the signal up to the output layer using:

$$o_j^p = \mathcal{F}(\sum_{i=0}^{n} w_{ij} o_i) = \mathcal{F}(net_j^p), \quad for\ o_0 = 1\ \ and\ \ w_{0j} = \theta_j.$$

Compare the output vector $\underline{\mathbf{O}}^p$ with the target output $\underline{\mathbf{T}}^p$ and calculate the discrepancy, *delta*, for each output node,

$$\delta_k^p = (t_k^p - o_k^p)\mathcal{F}'(net_k^p)^1. \tag{3.5}$$

[1]

$$\mathcal{F}'(net_k^p) = \frac{\partial \mathcal{F}(net_k^p)}{\partial net_k^p}.$$

Propagate the delta value δ_k^p of each output node to the previous layer nodes **before updating the weights**.

For every node in a **hidden layer** calculate its delta value by,

$$\delta_j^p = \mathcal{F}'(net_j^p) \sum_{k=1}^{m} \delta_k^p w_{jk}^p \tag{3.6}$$

and propagate these values to the previous layer nodes, until the input layer[2].

Update weights using the *delta rule*:

$$w_{jk}(t+1) = w_{jk}(t) + \beta \delta_k^p o_j^p. \tag{3.7}$$

Repeat with the next training pattern p+1.

Note that for monitoring the progress of the network, a measure of the error for each training example pair is given by equation 3.3.

The first derivative of the sigmoidal activation function:

$$\mathcal{F}(x) = \frac{1}{1+e^{-x}} \tag{3.8}$$

is,

$$\mathcal{F}'(x) = \mathcal{F}(x)[1-\mathcal{F}(x)] \tag{3.9}$$

[2]The input layer does not participate in the process, it just distributes the input values to all the units of the first layer.

then,

$$\mathcal{F}'(net_k^p) = \mathcal{F}(net_k^p)[1 - \mathcal{F}(net_k^p)] \quad (3.10)$$

$$= o_k^p(1 - o_k^p). \quad (3.11)$$

3.5 Practical considerations

As with many aspects of neural net systems, *experience* and *intuition* is often the best guide. Here are some hints that might be useful when training a multi-layer network:

- **Poor performance** of the network could be due to the **poor internal representation**. Adding more units to the hidden layer will allow a better recoding of the input.

- Speed up of the convergence time can be achieved by introducing a *momentum term*, λ, as

  $$w_{jk}(t+1) = w_{jk}(t) + \beta\delta_k^p o_j^p + \lambda\{w_{kj}(t) - w_{kj}(t-1)\}, \quad 0 < \lambda < 1 \quad (3.12)$$

  This, however, requires additional storage space in each neuron, since we have to remember the (t-1) weight vector, in calculating the (t+1) vector.

- As demonstrated in Figure 3.6, sometimes the network falls on a *local minimum*[1] and gets trapped. Adding some random noise to the current

[1] Local minimum can be considered to occur when two or more disjoint classes are categorised as the same.

weights will knock the system out of the local minimum, most of the times.

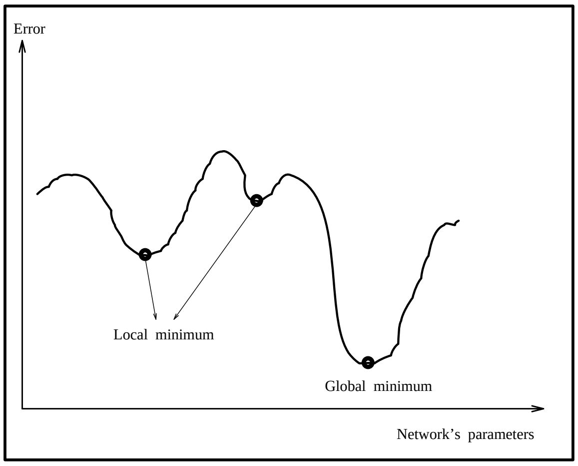

Figure 3.6: **Local and global minima.**

- During the training process, select the training pairs randomly from the data set. Avoid training the network with input vectors of one class and then switch to another class because in this way, the network will *forget* the original training. Imagine a tin sheet and someone with a hammer trying to give it a particular shape. If its knocks are not covering the whole surface *uniformly* but are directed only on specific points, then the surface will not be smooth but rather like a nail blanket.

Chapter 4

Other neural network architectures

In this chapter we move to *recurrent* network architectures and examine a number of different configurations such as the **Hopfield network**, the **Boltzmann machine** and **Kohonen's self-organised maps**.

We tried to give a flavour of how these networks and their training algorithms work without going into details, but at the same time avoiding over-simplification.

4.1 Recurrent and non-recurrent networks

In the previous chapter we studied single and multi-layer *feed-forward* neural networks along with their training algorithms. A common feature of those architectures is that *no output values* were fed back to the input nodes for the calculation of the next state output. For this reason they are known as **recurrent networks**.

The lack of feedback ensures that the network, no matter what the input values are, will produce an output. The system, therefore, is stable in every case.

Stability is a very important characteristic of the non-recurrent networks, but the absence of feedback *limits* the repertoire of their behaviour.

On the other hand *recurrent* networks have feedback paths from their outputs to their inputs. Once an input pattern is applied, the output is calculated and *fed back* as a new input. A new output is calculated again and the process repeats until the output becomes constant (a stable state).

Note that such a network may never find a stable state, *wandering interminably* from state to state never producing a constant value output.

Many theorems, however, have been developed which draw a line between stable and unstable recurrent networks, making their use possible and adding new momentum to the connectionist approach towards Artificial Intelligence.

4.2 Hopfield networks

One of the major contributions to the area of neural networks was made in the early eighties by **John Hopfield**. His contribution was more than a suggestion of a new model. His extensive analysis and study led his name to be associated with the network.

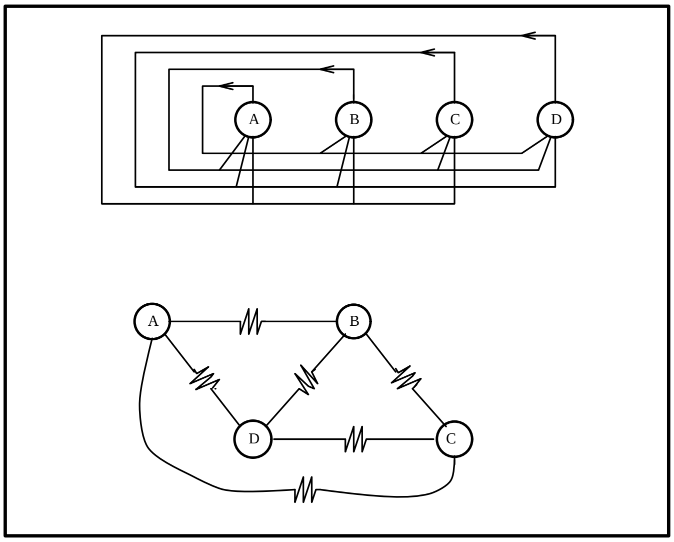

Figure 4.1: **A Hopfield network.**

A Hopfield network is a **content-addressable memory** in the sense that a previously memorised shape is looked up by stabilising the network's state, after giving it an incomplete description of the original shape as the input.

It consists of a number of nodes each connected to every other node via a single weight value. It is a *fully connected*, *symmetrically weighted* network, as shown in Figure 4.1(a) and (b).

The active units of the network are the neurons as described in Section 1.3. The use of a *step threshold* activation function suggests a two state nature of both inputs and outputs. Some models use binary (0 and 1) and others bipolar (-1 and 1) representation.

The major feature of the Hopfield network is that there are no input or output nodes! All nodes are the same, sometimes acting as input and sometimes as output units. Inputs to the network are applied to all nodes at the same time. The network will cycle through a succession of states until it converges to a stable solution, modifying every time the weighted connections according to a learning rule resembling that of Hebb.

The algorithm for storing patterns in *N-input* Hopfield network is given below:

Given a set of q bipolar or binary vectors, each of length N,

$$\underline{\mathbf{M}} = \{\underline{\mathbf{X}}^0, \underline{\mathbf{X}}^1, \ldots, \underline{\mathbf{X}}^{q-1}\}$$

where the p^{th} input vector is,

$$\underline{\mathbf{X}}^p = \{x_0^p, x_1^p, \ldots, x_{N-1}^p\}$$

- Assign connection weights using:

$$w_{ij} = \begin{cases} \sum_{p=0}^{q-1} x_i^p x_j^p & i \neq j \\ 0 & i = j \end{cases} \tag{4.1}$$

This implies that each weight value connecting node i and j is the sum of products of the i^{th} and j^{th} element of the input vector, for all input vectors.

The '$w_{ij} = 0$, for i=j' means that a node is not connected to itself. For this reason the weight matrix, $\underline{\mathbf{W}}$, has **zero diagonal**. Furthermore, connections between nodes are bidirectional, but with only a single value. This means that '$w_{ij} = w_{ji}$' and therefore the weight matrix is also **symmetric**. Networks not bearing these two properties may be unstable.

- Present an input vector to the network

$$y_i(0) = x_i, \quad 0 \leq i \leq N-1, \tag{4.2}$$

where $y_i(t)$ is the output of node i at time t and x_i is the i^{th} element of the input vector being presented.

- Iterate until convergence using:

$$y_i(t+1) = \mathcal{F}(\sum_{j=0}^{N-1} w_{ij} y_j(t)). \tag{4.3}$$

A Hopfield network is associated with an *energy function* given by:

$$E = -\frac{1}{2} \sum_i \sum_{j \neq i} w_{ij} x_i x_j + \sum_i x_i \theta_i \tag{4.4}$$

where, θ_i is the threshold value of unit i.

An energy surface[1] can then be constructed in which hollows represent minimum energy points and a good solution. Each node in the network decides which state it should switch into in order to *reduce* its energy.

A disadvantage of this simple version of the Hopfield net is that once the system falls in a local minimum there is no mechanism to take it out. A solution is discussed in the next section where we will introduce the *Boltzmann machine*.

4.3 The Boltzmann machine

The Hopfield model in its simplest version, as described previously, may converge to a local minimum, which may not give a satisfactory solution. A method that will help the network to escape from such a local hollow and move to a deeper well[2] is based on the *simulated annealing* process of metallurgy.

The idea behind can be found in those toys, kids use to play with:

> There is a transparent cube and inside there is a ball free to move on a landscape with wells and mountains, valleys and hills. The aim of the game is to move the ball from one place to another through the wells and mountains. The most difficult part is when the ball falls into one of the wells and you have to *shake* the cube very gently so as to free the ball but not to bring it to a worst

[1] Something very similar to the error surface of a feed-forward layer.

[2] A better solution or a *global* minimum.

location or back to the starting point.

Similarly in the energy landscape of a Hopfield net, a ball falling on a not-so-deep hollow should be given sufficient energy[1] so as to move about and possibly escape the local minimum, as demonstrated in Figure 4.2.

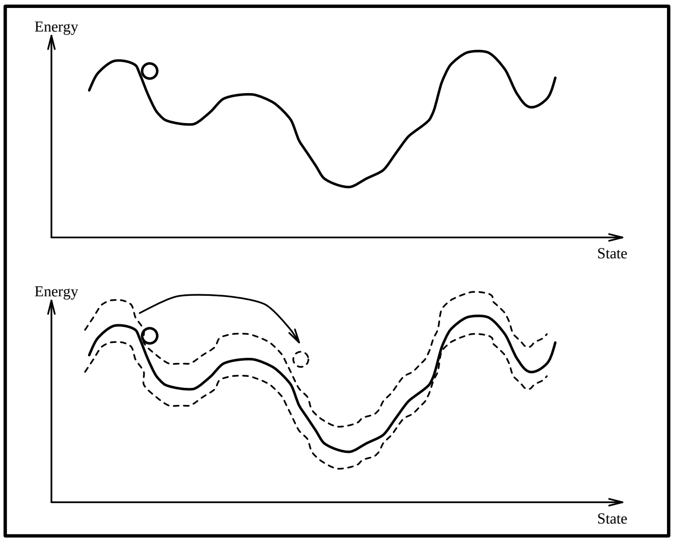

Figure 4.2: **Shaking the energy landscape**

This should be done carefully and the best way to achieve it is by providing a lot of energy at the beginning and gradually reduce it, as the network works its way towards the global minimum.

[1]To shake the energy landscape is a more popular term.

A very similar process is used in metallurgy to harden metals and is called *annealing*. At first the metal is raised to a very high temperature (it is given too much energy) and then it is left to cool down slowly so as its lattice becomes 'perfect' and impurities do not settle in its atomic structure.

A network of similar structure and learning algorithm to that of Hopfield but more flexible in escaping local minima, using the above method, was proposed in 1985 by **Ackley Hinton** and **Sejnowski**. It is called **Boltzmann machine** because each node is made to decide which state to follow with an uncertainty given by the Boltzmann probability distribution.

In a Hopfield net each node decides which state it should jump into in order to reduce the energy state of the system. In a Boltzmann machine each node decides which state it should jump into to achieve minimum energy, as before, but whether it will jump into that low energy state or into a *higher* one depends on the value of the probability function:

$$P_k = \frac{1}{1 + e^{-\frac{\Delta E_k}{T}}} \tag{4.5}$$

where ΔE_k is the energy gap between the present and future state, and T is a temperature factor which controls the rate of annealing. Increasing T, the probability of jumping into a higher energy state becomes larger, and therefore shaking is more effective.

With such a *probabilistic* learning rule the global minimum is usually reached since the system can escape from local minima by allowing, once in a while, jumps to higher energy states. However, the calculation power necessary to implement a Boltzmann network on a computer is vary large. This algorithm has been little used in real applications to date.

4.4 Self-Organised maps

In general, **self-organised maps** classify a pattern represented by an input vector, by organising the network, or map, according to the statistical properties of the input.

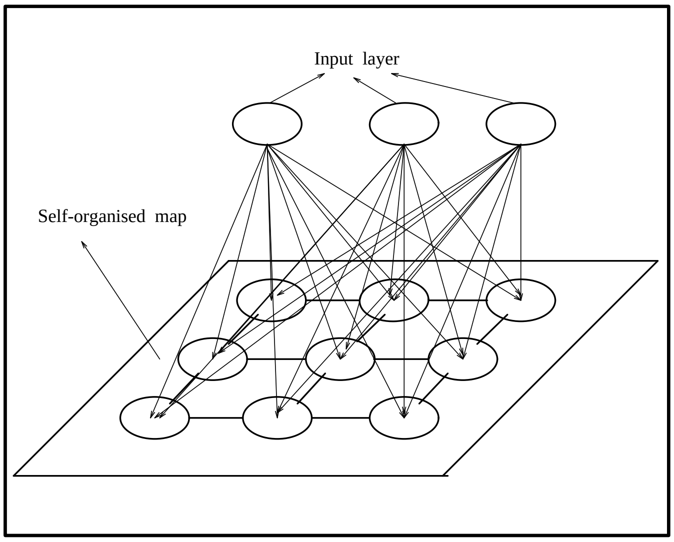

Figure 4.3: **A Self-organised feature classifier**

A typical Kohonen feature map is shown in Figure 4.3. There is only one layer of neurons in which interconnections are restricted to neighbouring nodes. All inputs are connected in all nodes of the map and each node on the grid is itself an output node.

The learning algorithm organises the nodes in the grid into local neighbourhoods that act as **feature classifiers** of the input data. The network is presented with a series of input patterns and must discover regularities in those patterns that can be used to divide them into elements of similar patterns.

Learning is called *competitive* in the case of self-organised maps because it assures that on a given trial just one unit will **win** and its output will be 1, while the output of all other units in the neighbourhood will be zero[1].

[1] A technique known as 'winner-take-all'.

Chapter 5

Characteristics and applications of Neural Networks

Despite the enormous variety of neural network architectures, they all tend to have the same general characteristics. In this chapter we will consider some of these features and also we will discuss some of the applications of neural networks.

5.1 Generalisation

One of the most important characteristic of Neural Networks is their capacity to *generalize*. If the network which has previously been trained to classify input patterns into particular classes, is given a *new* input pattern, it will typically respond in accordance with trained patterns to which is *similar*[1].

Neural Networks generalise by detecting features of the input pattern that have been learned to be significant, and so coded into the internal units: the similarity structure is implicit in the weight matrix. Therefore weights are the means of treating similar inputs similarly.

Such an approach to similarity is not sufficient and is often questioned. Because it is left, more or less, to luck whether the network will recognise as important the properties we recognise as important.

For example, suppose we have a set of pictures of people smiling. Some people have beard some others do not, some have hair others do not. When a human is shown all the pictures and asked to identify a common feature, we are pretty sure that he will say that they are all smiling.

Now suppose we train the neural network with these pictures. We **hope** that the network will recognise that they are all smiling, but in effect it is left to luck because along with the smiling feature, every picture carries the

[1]Similarity poses a notorious philosophical problem. Two objects cannot be said to be similar if, simply, they share some properties. According to *Nelson Goodman* (1955) any two objects are alike in an **infinite** number of respects. Assessing similarity **only** in terms of numbers of common properties is inadequate.

following information:

- two eyes,
- two ears,
- a nose, a mouth, etc.

We should not be surprised when the network responds with the answer that the faces in all the pictures have two eyes.

Why then it is so obvious for us, humans, that smiling is the common feature while for the neural network is so difficult?

In my opinion, this is due to the existence of an internal hierarchy mechanism which assigns data from our sensory units an importance factor depending on how often it appears, in what situation we are, and all the previous experience we have been gathering since the day we were born[1].

For a conventional neural network this is not the same. The smiling along with all the other information conveyed by the pictures are of equal importance if they are presented with equal significance. Despite this apparent weakness, this enormous generalisation ability allows a neural network to perform more successfully on problems where *rule-based systems* fail.

The most common application of the generalisation property is to train the network with a huge data set and then to ask it to predict the output

[1] Do not forget that the human face is the picture we see most often in our life.

for a given input situation, based on the previous experience gained. This approach could be very successful if we know all the input variables of the system we choose to model, but simply a catastrophe in any other case. This means that if we try to forecast the weather by training the network with sets of atmospheric pressure, humidity and wind velocity only, we will fail since we forgot the temperature input, which is obviously necessary.

However, for most real-world problems, nobody is sure what parameters are important and to what degree. For example, the financial market behaviour may depend on some factors that are blindingly obvious, such as unemployment or interest rates but it might also depend on whether the cat of Mrs Reagan died or if 'Mary Poppins' was shown on TV last night.

Applications vary from financial market predictions to meteorological forecasting, and from determining the life of a diesel engine to searching for life in the universe.

5.2 Classification

The human brain is very good at classifying data collected by its sensory units. We do this continuously, and most of the time unconsciously. Neural networks are also good at classifying input data. Training associates[1] some input patterns to a certain *output class*. The network performs a mapping

[1]Classification and Association are terms very close in meaning. The former, groups input and output data which is *given* by the user, while the latter, rather separates input data in a more intuitive way.

of n-dimensional input space onto an m-dimensional output space. which is very useful, especially when m is much smaller than n. In such a case large chunks of input data can be classified as *good*, *bad*, *beautiful*, *dangerous*, etc. This in *image processing* or *communications* is called **data compression**. The only difference being that in our case, assuming a well trained net, this compression is done in a more, kind of, **intelligent** way.

Classification by a neural network of an input pattern, once it has been trained, is done almost in zero time. The size of the input patterns affects the time taken for training, but not the response time, as it involves only a simple forward propagation of the input signal through all the layers to the output.

An interesting applications is in the compression of voice signals sent via the telephone lines. Figure 5.1 shows a network of 'n' units in its input and output layer and a hidden layer of m units (m $\ll$ n). It is trained with a very large data set of various people's voices, and the input is the same as the output. After training, the first half of the network is the **transmitter**. An n-bit input pattern is applied to it and an m-bit output is obtained from the hidden layer and sent over the telephone lines to the other half of the network. The hidden layer will receive that m-bit pattern and the output layer will generate the pattern applied at the input of the transmitter.

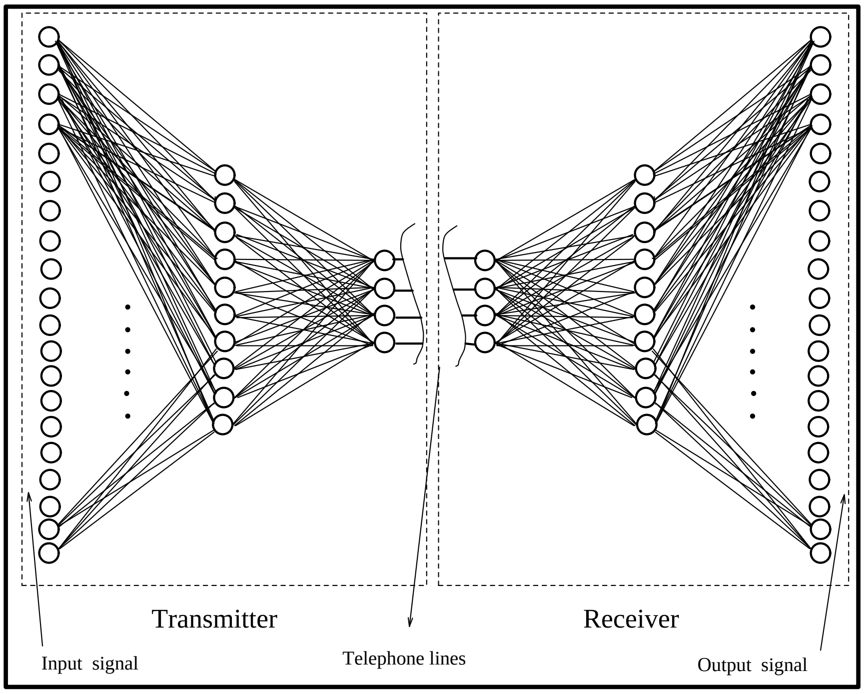

Figure 5.1: **Data compression of a voice signal to be sent in a telephone line.**

No need to tell that the system *is not a 100% reliable.* The main reason being the training and the size of the network, which in practice is not as large as it should be in order to achieve a correct internal representation.

In certain routine *diagnoses* in medicine, neural networks are taking the place of *expert systems.* They are much simpler, more flexible and with a different training can be used in another application.

In image processing, recovery of images from noise is a very common problem. A neural network which has the same number of inputs and outputs can be used in restoring the input image. Again its performance it is not very reliable, and is unsuitable for applications where accuracy is important.

5.3 Association

Human memory is often *associative*; one thing reminds us of another and another, based on a *chain of mental associations.* This associative ability is often used in memory organisation by establishing one end of an associative link and allowing our mind to connect it to the desired memory.

This feature of the brain is characteristic of the recurrent neural network models such as the **Hopfield network**. After training, a portion of the desired data is supplied and the full data is returned. This is called an associative memory and is in contrast to the ordinary *location addressable computer memory.*

Associative memories work in a fashion very similar to the human brain, and in the future, they will be used widely to construct 'intelligent' systems such as robots.

5.4 Fault tolerance

Neural Networks are intrinsically **fault-tolerant** since they are *distributed parallel processing* systems by definition. Each unit contributes only a small fraction to the final output response and, therefore, damage to a few of these nodes will have a minimum effect on the output. The distributed nature of the information means that damage has to be extensive before the network's response *degrades*[1] badly.

Neural Networks are also tolerant to noise due to their ability to generalise from taught examples to corrupted versions of the original patterns.

[1]This is in contrast to conventional machines which cease from operation **at once** if a component is damaged.

Chapter 6

Transputers

Machines which have physically separate processing and storage units, usually referred to as *von Neumann* architectures, exhibit a fundamental speed limitation.

The link between the two parts creates a **bottle-neck**, defining an *upper-bound* on performance. Furthermore, this two-part design produces extremely inefficient architectures, when the metric of efficiency is the utilisation of individual switching elements.

It is for this reason that Neural Networks implemented on a *von Neumann* machine will perform badly, especially when the number of interconnections and processing elements increases.

These limitations are not present in *array machines* or *MIMD* architectures. In this case, a processor equivalent to that used in a serial computer is duplicated a number of times. Each processor is given its own local memory and can perform an arithmetic operation in its local memory in the same time and in the same manner similar to that of a serial computer. Thus, a number of operations equal to the number of processors can be performed at the **same** time.

The signifigance of this is great, especially when communication between processors is good and efficient, while the communication needs of the program to be implemented are reduced to a minimum.

6.1 Overview of the transputer

The transputer is one of the several 32 bit VLSI processors that have been specifically designed for use in concurrent message-passing systems.

Its structure contains the three essential ingredients of a message-passing system:

- Processor
- Local memory
- Communications

and its operation is based on the idea of a group of compatible components which are able to communicate fast, reliably, and asynchronously, with the minimum of external logic.

The **IMS T800** transputer system features the following:

- **Processor**: At a speed of 20 MHz, it can achieve a performance up to 10 MIPS. It consists of a 32 bit central processing unit and a 64 bit floating point unit.
- **Memory:** There are available 4 KB of static RAM, while it can address up to 4 GBytes of memory space. The data transfer rate can be as high as 80 MB/sec.
- **Communication**: Messages between local memory and another transputer are transferred via the four hardware links. The link interfaces

and the processor operate concurrently, hence data transfer is not interrupting the processing.

Although each transputer may be assigned one or more processes, it can only execute one at a time. Process management is done by a scheduler held in microcode.

Communication between two processes that are co-resident occurs via single words of memory. Communication between two processes that reside on adjacent processors takes place via a serial link.

Chapter 7

Theory into practice

This chapter describes the practical part of the project, where theory was put into practice using the theoretical background and the principles given in the previous chapters.

Specifically, code was written for simulating a **multi-layer**, **back-propagation** network on the **T805-25** transputer system and a SPARC workstation.

The very simple and practical user interface in X-windows, allows the user not only to study how the network behaves, but also to test its performance in real-world applications.

We saw this project as a challenge to write highly portable and flexible code, allowing the user to simulate, networks of any size[1]. That was our main guideline ignoring almost completely, we have to admit, *communication overhead* and transputer network configuration. As a result the application

[1]Theoretically yes, but in practice it depends on the available RAM.

has only been tested on the root transputer.

From the beginning, time was a problem. We set, therefore, some priorities and followed this particular strategy up to the end. As a result, the program can run on any system that has a **parallel C** or **gnu C** compiler. There is a problem, however, with the parallel version, where it runs a bit slowly. Certainly one its cause is the heavy communication load. An additional reason might be that the program was only tested on the root transputer and, in effect, parallelism was inexistent.

We considered it proper to make some suggestions on how to make the simulation more efficient by adopting a certain configuration which will decrease communication overhead, although the proposed scheme was never tested in practice.

7.1 General programming concepts

The network is created in memory. Each neuron requires space to store certain parameters which are used in the propagation training. Processes are then created for each neuron and sit idle waiting for inputs. Once all the inputs are received, an output is calculated.

The most important data structure of the program is called 'Neuron' and has 10 fields described below:

(1) **layer**: An integer declaring the current layer.

(2) **node**: An integer declaring the current node.

(3) **inpnum**: An integer declaring the number of inputs.

(4) **outnum**: An integer declaring the number of outputs.

(5) ***weight**: A pointer to the array containing all the weights, in double precision format, connected to the input of the node.

(6) **threshold**: A double precision variable holding the threshold value of the node.

(7) **sigmoid**: A double precision variable holding the output of the node after it has been processed by the activation (sigmoid) function.

(8) **output**: A double precision variable holding the output of the node before it is processed by the activation function.

(9) **input**: A double precision variable holding the weighted sum of all the inputs to the node.

(10) **delta**:A double precision variable holding the delta error value for the back propagation.

There are three phases in the operation of the neural network

(i) **Create Neurons**: Each node of the network is associated with a **pointer** to the structure 'Neuron'. Input and output channel addresses are assigned to the node and weights are initialized to small random values.

(ii) **Start Training**: An input pattern is applied to the input layer which will distribute these values to the next layer. The nodes of the next layer will receive the input vector and will propagate the signal through and up to the output layer where error values are calculated. These error values are sent back to the previous layer using different channels[1]. A process called the 'error server' will receive the error signals and will monitor the performance of the network. The calculations are performed **in parallel** for nodes of the same layer and **sequentially** for different layers. No two layers perform a calculation at the same time.

(iii) **Testing**: It is the final stage. The network has been trained and

[1]Channels are **unidirectional.**

now test patterns are applied to evaluate the training. Step (ii) is repeated but this time, only a single forward pass is required.

7.2 Processes

Each node in the hidden and output layers is actually a process which sits there waiting for input. Only when **all** expected input values are received, the process will calculate and transmit the output value to the other processes it is connected to.

Sending output values is done in parallel. Therefore if a process is connected to n other processes (being the neurons of the next layer for example), then n other processes should be created with the same task: to send the output value to the address given. Then these processes are destroyed. This procedure is not at all efficient and a lot of time is wasted here.

Receiving input values is more complicated. Each process expects input from a constant number of channels. The receive procedure is not complete until all input channels got a value. Some synchronisation problems exist at this stage, as a single node may stop the network because something went wrong in calculating its output[1]. Care should be taken to use a safe communication protocol which ensures that an output has been received by the destination process. Otherwise, because all the channels are checked for

[1] Not at all fault tolerant!

input in a sequential manner (multiplexing), some output values may have been sent but not received, causing **deadlock**. We have to say, and we might be wrong, that the current version of parallel C compiler is not very reliable and extremely complicated. Debugging is practically impossible and problems may not be detected no matter how careful and methodical a user is.

Each hidden layer node process is independent of any other and does not share any global variables. The input and output layer processes do some message passing through global variables and care should be taken to assign them in the **same** processor at configuration.

7.3 Communication

Processes communicate through channels. **Software** channels when on the same transputer or **hardware** channels (links) when on different transputers. Global variables should not exist and all the information should be passed through the channels. This makes the Back Propagation algorithm very difficult to implement since for a fully connected network each node requires:

- Input channels as many as the nodes of the previous layer.
- Output channels as many as the nodes of the next layer.
- Error input channels as many as the next layer nodes.
- Error output channels as many as the previous layer nodes.

Taking, for example, a typical network of an input layer of 25 nodes, an output layer of 5 nodes and a single hidden layer of 200 nodes we will need for each node of the input layer 200 output channels so $200 \times 25 = 5000$ channels for the input layer.

Each node of the hidden layer requires 5 output and 5 error channels for the error, a total of $200 \times 10 = 2000$ channels

The whole network will need some 7000 channels. Considering the fact that each transputer has only 4 hardware links then a lot of multiplexing has to be done, slowing down the application considerably. A *structured network* approach[1] could prove more adequate for a multi-transputer implementation.

7.4 Facts

A transputer has only *four* hardware links to communicate with other transputers. In theory, however, a process residing on a transputer, can send and receive data from *any* number of channels. This obviously presupposes *multiplexing* channels onto the four available links – a solution which is not at all parallel.

For an efficient computational scheme which fully exploits parallelism in the existing machine, communication between transputers *should be reduced to a minimum.*

Furthermore, communication between transputers that have **no direct** connection, should be avoided by using an appropriate network configuration

[1]Refer to Section 7.4 and 7.5.

which will assign processes that do a lot of communication to the *same* or *adjacent* processors.

Communication between processes is made more difficult due to the **uni-directional** nature of channels.

As far as the neural network operation is concerned, we should note the following:

> Computation is **parallel** between nodes of the **same** layer and needs no communication whatsoever.

> Computation between layers is **sequential**, and at any time only one layer is active. For practical reasons, however, two layers must be active at the time of sending data since it takes two to communicate.

> A fully connected network has the greatest number of connections and requires great number of channels. We have a maximum communication overhead for such a network.

> Recent results[1] indicate that networks of smaller size can generalise better than fully connected networks. Many topologies can be constructed based on this observation, generally known as **Structured Neural Network architectures**.

[1] Baum and Haussler, 1989, in [10].

In addition, communication can be further reduced using the **receptive field** technique; where each input node is given only a **small** portion of the input pattern, and each hidden and output node is connected to only a **subset** of the units of the previous layer.

Weight sharing is often used to reduce the memory space required by the network: A single weight value controls several connections. In my opinion, there is an advantage in reducing the number of free parameters of the network, but this increases communication since weight values are calculated independently as before and further processing is needed to find the *average weight change*.

7.5 A proposed Neural Network architecture

Based on the philosophy of the **structured neural networks'** efficient network topology would be one of the following:

- Partition each layer of the network into **groups of nodes** , each one assigned to one transputer, and reduce communication between these groups using a receptive field technique, as shown in Figure 7.1a.
- Partition the whole neural network into groups of smaller neural networks each one assigned to one transputer. Again communication be-

tween these groups should be minimal. Such a network is shown in Figure 7.1b.

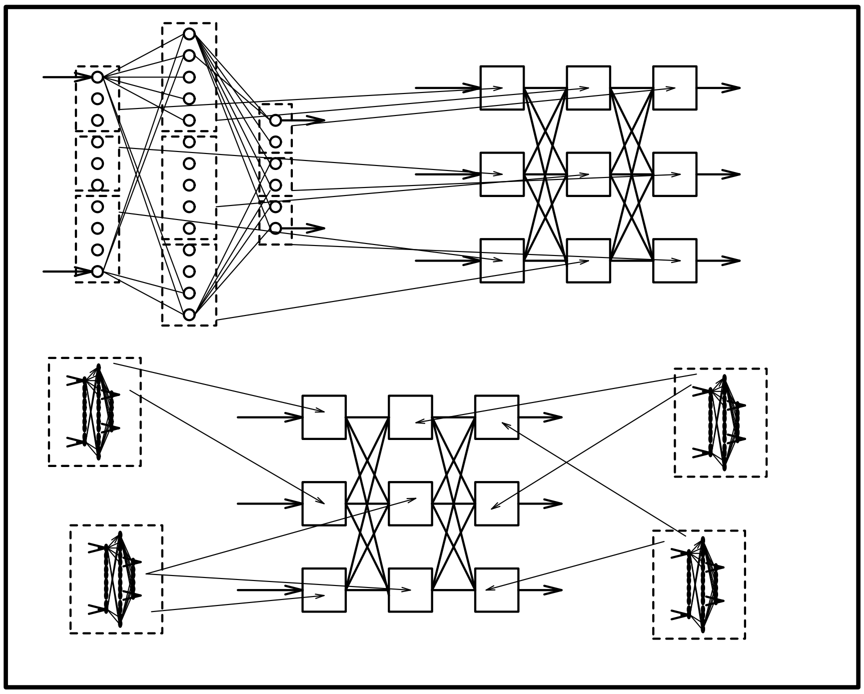

Figure 7.1: **Proposed Neural Network configuration.**

We find the latter idea more attractive because it resembles the operation of the brain.

7.6 Pattern recognition

Pattern recognition is the mapping of a specific pattern onto a more general pattern or class. It is fundamental to a variety of perceptual and cognitive tasks and therefore, to modelling human capabilities.

Neural Networks are devices for mapping one class of patterns onto another class by encoding the statistical regularities of the input in their weighted connections.

The claim, long ago, that they are highly suitable medium for performing pattern recognition is now a fact.

We could not resist trying them out. For this purpose we used a regular size network of :

- An input layer of 25 nodes.
- One hidden layer of 80–100 nodes.
- An output layer of 1–3 nodes, capable of handling 1–8 output classes.

to train it on two texture images.

The best technique to construct 25 bit input vectors from the texture images, forming the training data set, is to take 5×5 **squares** from the images and put them in a serial fashion. However, due to a technical problem, we had to compromise and take 25 bit **rows** from the image which is not very representative since in this way, only regularities on the vertical axis are present on the training set.

Despite this problem, the network after about 800 iterations, achieved correct classification of **unknown** input data with **80%** success.

When we tried with 8 texture images, the network **could not converge**.

7.7 Brain Tumours

Since its inception in the early seventies *X-Ray Computed Tomography* (CT) has been used extensively in the medical field[1].

The area of medicine concerned with the brain, its anatomy and its pathology benefited greatly from this new tool since it allowed the specialist to assess the extent of damage in some patients' brain, and to relate some functional impairments to anatomical sites and their deviation from 'normality'.

Over the past two decades radiologists developed *adhoc* methods to detect abnormalities in brain scans based on both their knowledge of the anatomy of the brain the and effect of tissue density on the attenuation of X-Ray beams. This has other applications besides the afore-mentioned analysis, e.g. *surgery planning*.

Current methods of analysis present many drawbacks:

– It requires the availability of an expert,

[1]Pioneering work by Hounsfield made it possible to view inside a volume of living tissue.

- Experts vary in their interpretation of data presented to them,
- Performance is hindered by the physical and mental state of the expert.

These drawbacks make the accurate measurement of geometry and density discrepancies difficult. First, the human eye can only discern about 64 grey shades. Second, although the eye is competent at the overall task of recognition, it is easily surpassed by computers at spectral analysis and quantifying differences.

This application is concerned with the detection and measurement of brain tumours in 'enhanced' CT scans. The enhancement is achieved through the administration of a contrast medium (e.g. Iodine).

In an enhanced scan abnormalities due to cancer growths (tumours) present distinctive features. The latter consist of a low density area (due to oedema) entertwined with high density fuzzy growth due to the cancer cells. The low density area can easily be delimited using unsupervised clustering e.g. ISODATA or K-Means etc. Unfortunately, oedema has about the same density as the ventricles[1].

This is where Artificial Neural Networks make their 'éntree'. The ISODATA clustering algorithm produces a map of the slice under consideration and a set of cluster centres. This is used as a guide to a simple algorithm which selects small blocks (5x5 pixels) from the original image to be fed to the ANN as an input pattern. The ANN would have been trained to recognize

[1] Cavities filled with fluid, hence low density.

enhancement textures[1]. Individual input patterns vote for their respective regions, and allow the separation of abnormalities from ventricular sections[2].

Preliminary results were promising, we could achieve a 90ANN with a data set taken from a small group of images hand-segmented by the physician.

Further investigation could be directed at overriding the ISODATA clustering when there is evidence of a better classification of a region (or indeed sub-region). Moreover, frequency analysis of normal and abnormal brain tissue (the image is viewed as a two-dimensional function producing the grey value of each pixel) could give clues beyond geometry, about the evolution of tumours. In addition, one could try to map symmetry, deformation of known features, etc. onto input vectors and train the ANN for the combined task of detection and quantification.

Another approach would be to train relatively small ANNs for different tasks and then combine their results to make a decision. Besides, training can be coordinated between the different ANNs.

[1]This is an obvious over-simplification, but worth a try!

[2]Ventricles also appear at the approximately the same position for a given slice depth.

Chapter 8

Conclusions

This report provides an introduction to the exciting field of neural networks. The subject is one of the most challenging I have ever come across. Not only because it combines four disciplines: **engineering, mathematics, psychology and neuro-physiology** in one, but also because it is a revolutionary alternative to conventional problem solving using computers.

At the same time, an attempt was made to exploit parallelism and existing technology of parallel machines in order to produce a more efficient and versatile neural network simulator. As has already been said, this was not very fast, one of the reason being that technology is not ready yet to meet the requirement of highly connected , simple processing elements. Instead we had to devise methods and tricks in order to produce something which is compatible with what was available.

In specific, when using arrays of transputers, *pseudo-parallelism* must be employed meaning that several processes live on a single transputer. Using this principle, structured neural networks were adopted and a particular architecture was proposed, but never tested in practice.

The last aim of the project was to evaluate the behaviour of neural networks, especially the multi-layer, back-propagation kind. For this purpose another program was written to simulate such a network on a sequential sequential machine – a SPARC station.

Two applications were selected from the field of pattern recognition and simulation carried out with satisfactory results[1].

The most impressive achievement, though, was the use of the simulator in identifying brain cancer.

As a last word, the future belongs to connectionism.

[1]Refer to Sections 7.6 and 7.7 for a more extensive description.

Appendix A: The Back Propagation Rule

Hidden and output layers update their weights in a different way.

Update of output layer weights

Let us define a measure of the error for all output nodes, k, at the p^{th} training example:

$$E^p = \frac{1}{2}\sum_{k=1}^{m}(t_k^p - o_k^p)^2, \tag{.1}$$

where,

t_k^p, is the *expected* output value of node k, at the end of the p^{th} training iteration.

o_k^p, is the *obtained* output value of node k, at the end of the p^{th} training iteration. Also given by $\mathcal{F}(net_k^p)$.

To achieve a smaller error value in the next iteration, we should change the weights proportionally to the negative gradient of the error surface, defined by E^p, at the current weight combination point:

$$\Delta\underline{\mathbf{W}} = -\beta \cdot \nabla E^p = -\beta \cdot \{\frac{\partial E^p}{\partial w_{k1}^p}, \frac{\partial E^p}{\partial w_{k2}^p}, \ldots, \frac{\partial E^p}{\partial w_{kj}^p}\}^t, \tag{.2}$$

where,

$$\frac{\partial E^p}{\partial w_{kj}^p} = \frac{1}{2}(0+0+\ldots-2(t_k^p-o_k^p)\cdot\frac{\partial o_k^p}{\partial w_{kj}^p}+\ldots+0) = -(t_k^p-o_k^p)\cdot\frac{\partial \mathcal{F}(net_k^p)}{\partial w_{kj}^p}. \tag{.3}$$

By Chain Rule,

$$\frac{\partial \mathcal{F}(net_k^p)}{\partial w_{kj}^p} = \frac{\partial \mathcal{F}(net_k^p)}{\partial net_k^p}\cdot\frac{\partial net_k^p}{\partial w_{kj}^p}, \tag{.4}$$

where,

$$net_k^p = \sum_{j=0}^{l} w_{kj}\mathcal{F}(net_j^p), \tag{.5}$$

so,

$$\frac{\partial(net_k^p)}{\partial w_{kj}^p} = \mathcal{F}(net_j^p). \tag{.6}$$

Finally, substituting into equation 3, we get the rate of change of the error with respect to the output layer weights as:

$$\begin{aligned}\frac{\partial E^p}{\partial w_{kj}^p} &= -\mathcal{F}(net_j^p)\cdot(t_k^p - o_k^p)\cdot\frac{\partial \mathcal{F}(net_k^p)}{\partial net_k^p} &(.7)\\ &= -(t_k^p - o_k^p)\cdot\mathcal{F}'(net_k^p)\cdot o_j^p. &(.8)\end{aligned}$$

Define an *error term* of the output layer as:

$$\delta_k^p = (t_k^p - o_k^p)\cdot\mathcal{F}'(net_k^p). \tag{.9}$$

The new weight connecting the k^{th} output and the j^{th} hidden layer node is given by:

$$w_{kj}(t+1) = w_{kj}(t) + \beta\cdot\delta_k^p\cdot o_j^p. \tag{.10}$$

Update of hidden layer weights

Let us first investigate which parameters the total output error depends on:

$$\begin{aligned}E^p &= \frac{1}{2}\sum_{k=0}^{m}\{t_k^p - \mathcal{F}(net_k^p)\}^2 &(.11)\\ &= \frac{1}{2}\sum_{k=0}^{m}\{t_k^p - \mathcal{F}(\sum_{j=0}^{l} w_{kj}\mathcal{F}(net_j^p))\}^2 &(.12)\\ &= \frac{1}{2}\sum_{k=0}^{m}\{t_k^p - \mathcal{F}(\sum_{j=0}^{l} w_{kj}\mathcal{F}(\sum_{i=0}^{n}(w_{ji}\mathcal{F}(net_i^p))))\}^2\ldots &(.13)\end{aligned}$$

Therefore, the output error depends on the weight and output values of **all** the nodes of the **previous** layers. The contribution of the j^{th} node of the hidden layer is given by:

$$\frac{\partial E^p}{\partial w_{ji}^p} = -\sum_{k=0}^{m}\{(t_k^p - o_k^p)\cdot\frac{\partial \mathcal{F}(net_k^p)}{\partial w_{ji}^p}\}. \tag{.14}$$

By Chain Rule,

$$\frac{\partial \mathcal{F}(net_k^p)}{\partial w_{ji}^p} = \frac{\partial \mathcal{F}(net_k^p)}{\partial net_k^p}\cdot\frac{\partial net_k^p}{\partial w_{ji}^p}, \tag{.15}$$

and,

$$\frac{\partial net_k^p}{\partial w_{ji}^p} = \frac{\partial\sum_{j=0}^{l} w_{kj}^p\cdot\mathcal{F}(net_j^p)}{\partial w_{ji}^p} \tag{.16}$$

$$= \sum_{j=0}^{l}\{\frac{\partial w_{kj}^p}{\partial w_{ji}^p}\cdot\mathcal{F}(net_j^p) + \frac{\partial \mathcal{F}(net_j^p)}{\partial w_{ji}^p}\cdot w_{kj}^p\}, \tag{.17}$$

clearly,

$$\frac{\partial w_{kj}^p}{\partial w_{ji}^p} = 0,$$

so,

$$\frac{\partial net_k^p}{\partial w_{ji}^p} = \frac{\partial \mathcal{F}(net_j^p)}{\partial w_{ji}^p}\cdot w_{kj}^p. \tag{.18}$$

Again, by Chain Rule,

$$\frac{\partial \mathcal{F}(net_j^p)}{\partial w_{ji}^p} = \frac{\partial \mathcal{F}(net_j^p)}{\partial net_j^p}\cdot\frac{\partial net_j^p}{\partial w_{ji}^p}, \tag{.19}$$

and,

$$\frac{\partial net_j^p}{\partial w_{ji}^p} = \frac{\partial\sum_{i=0}^{n} w_{ji}^p\cdot\mathcal{F}(net_i^p)}{\partial w_{ji}^p} \tag{.20}$$

$$= \mathcal{F}(net_i^p). \tag{.21}$$

Substituting back in the previous equations,

$$\frac{\partial \mathcal{F}(net_j^p)}{\partial w_{ji}^p} = \frac{\partial \mathcal{F}(net_j^p)}{\partial net_j^p}\cdot\frac{\partial net_j^p}{\partial w_{ji}^p}, \tag{.22}$$

and,

$$\frac{\partial net_k^p}{\partial w_{ji}^p} = w_{kj}^p \cdot \frac{\partial \mathcal{F}(net_j^p)}{\partial net_j^p} \cdot \frac{\partial \mathcal{F}(net_j^p)}{\partial w_{ji}^p}, \tag{.23}$$

and,

$$\frac{\partial \mathcal{F}(net_k^p)}{\partial w_{ji}^p} = \frac{\partial \mathcal{F}(net_k^p)}{\partial net_k^p} \cdot w_{kj}^p \cdot \frac{\partial \mathcal{F}(net_j^p)}{\partial net_j^p} \cdot \frac{\partial \mathcal{F}(net_j^p)}{\partial w_{ji}^p}. \tag{.24}$$

Finally,

$$\begin{aligned}
\frac{\partial E^p}{\partial w_{ji}^p} &= -\sum_{k=0}^{m}\{(t_k^p - o_k^p) \cdot \frac{\partial \mathcal{F}(net_k^p)}{\partial net_k^p} \cdot w_{kj} \cdot \frac{\partial \mathcal{F}(net_j^p)}{\partial net_j^p} \cdot \mathcal{F}(net_i^p)\} && (.25)\\
&= -\mathcal{F}(net_i^p) \cdot \frac{\partial \mathcal{F}(net_j^p)}{\partial net_j^p} \cdot \sum_{k=0}^{m}\{(t_k^p - o_k^p) \cdot \frac{\partial \mathcal{F}(net_k^p)}{\partial net_k^p} \cdot w_{kj}\} && (.26)\\
&= -\mathcal{F}'(net_j^p) \cdot o_i^p \cdot \sum_{k=0}^{m}\{(t_k^p - o_k^p) \cdot \mathcal{F}'(net_k^p) \cdot w_{kj}\}. && (.27)
\end{aligned}$$

The new weight connecting nodes of i^{th} and j^{th} layer is given by:

$$w_{ji}(t+1) = w_{ji}(t) + \beta \delta_j^p o_i^p, \tag{.28}$$

where,

$$\delta_j^p = \mathcal{F}'(net_j^p) \cdot \sum_{k=0}^{m} \delta_k^p w_{kj}^p \tag{.29}$$

is the error term of the hidden layer nodes,

and δ_k^p is the error term of the output layer as defined in equation 9.

Bibliography

[1] Freeman James. *Neural Networks, Addison-Wesley, 1991.*

[2] P.D. Wasserman. *Neural Computing: Theory and practice, Van Nostrand Reinhold, 1989.*

[3] R. Beale, T. Jackson.*Neural Computing: An introduction, Hilger, 1990.*

[4] M. Gluck, D.Rumelhart.*Neuroscience and connectionist theory, L. Erlbaum Associates, 1989.*

[5] J.McClelland, D.Rumelhart. *Parallel Distributed Processing, MIT Press, 1986.*

[6] W.Bechtel, A.Abrahamsen. *Connectionism and the mind, Basil Blackwell, 1990.*

[7] A.Anderson. *Minds and machines, Prentice-Hall, 1964.*

[8] M.Arbib. *Brains, machines and mathematics, McGraw-Hill, 1964.*

[9] R.Arnheim. *Visual thinking, University of California Press, 1969.*

[10] A.Stafylopatis. *Parallel Implementation of structures feedforward neural networks.*