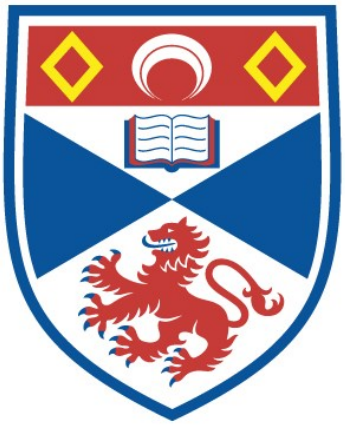
# MODELLING CONTINUOUS SEQUENTIAL BEHAVIOUR TO ENHANCE TRAINING AND GENERALIZATION IN NEURAL NETWORKS

Chen Lihui

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



1993

# MODELLING CONTINUOUS SEQUENTIAL BEHAVIOUR TO ENHANCE TRAINING AND GENERALIZATION IN NEURAL NETWORKS

A thesis presented by

Lihui Chen, BEng.

Department of Mathematical and Computational Science

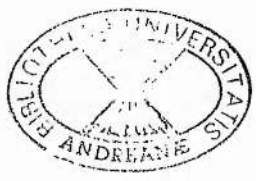
University of St. Andrews

to the

University of St. Andrews

in application for the degree of Doctor of Philosophy

June 1992



## DECLARATION

I, Lihui Chen, hereby certify that this thesis has been composed by myself, that it is a record of my own work, and that it has not been presented in partial or complete fulfilment for any other degree or professional qualification.

I was admitted to the Faculty of Science of the University of St. Andrews under Ordinance General No. 12 and as a candidate for the degree of Ph.D in February 1988.

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

Signed

Date 24/06/92

## Certificate

I hereby certify that the candidate, Lihui Chen, has fulfilled the conditions of the Resolution and Regulations appropriate to the Degree of Ph.D.

Signature of supervisor : Date 18/6/92

## ACKNOWLEDGEMENTS

I am greatly indebted to my supervisor, Dr. Mike K. Weir, for his persistently guidance, encouragement and invaluable assistance towards the completion of this thesis.

My sincere thanks to all friends for being there, and especially to my beloved husband David for his understanding and the hearty support during my research and preparing the thesis.

Finally, I also wish to express my appreciation to the Overseas Research Student Award (ORS/8734011) and the University of St.Andrews for granting me the scholarship to study in the United Kingdom.

## ABSTRACT

This thesis is a conceptual and empirical approach to embody modelling of continuous sequential behaviour in neural learning. The aim is to enhance the feasibility of training and capacity for generalisation.

By examining the sequential aspects of the passing of time in a neural network, it is suggested that an alteration to the usual goal weight condition may be made to model these aspects. The notion of a goal weight path is introduced, with a path-based backpropagation (PBP) framework being proposed.

Two models using PBP have been investigated in the thesis. One is called Feedforward Continuous BackPropagation (FCBP) which is a generalization of conventional BackPropagation; the other is called Recurrent Continuous BackPropagation (RCBP)which provides a neural dynamic system for I/O associations. Both models make use of the continuity underlying analogue-binary associations and analogue-analogue associations within a fixed neural network topology.

A graphical simulator **cbptool** for Sun workstations has been designed and implemented for supporting the research. The capabilities of FCBP and RCBP have been explored through experiments. The results for FCBP and RCBP confirm the modelling theory. The fundamental alteration made on conventional backpropagation brings substantial improvement in training and generalization to enhance the power of backpropagation.

# CONTENTS


## CHAPTER 1 INTRODUCTION

1.1 Introduction....1

1.2 A view of historical development....3

1.3 Sequential processing in neural networks....7

1.4 Basic concepts....9

1.5 Outline of the thesis....17

## CHAPTER 2 SEQUENTIAL PROCESSING USING STATE-BASED BP MODELS

2.1 Introduction....20

2.2 time representation in neural networks....20

2.2.1 the spatial representation of time....21

2.2.2 dynamic system approaches – specific topologies....23

2.2.3 dynamic system approaches – specific dynamic rules....24

2.3 Major approaches....26

2.3.1 time-delay networks....26

2.3.2 partially recurrent networks....28

2.3.3 discrete backpropagation through time....33

2.3.4 forward propagation....36

2.3.5 teacher forcing networks....40

2.3.6 dynamic recurrent networks....41

2.3.7 moving targets method....43

2.4 conclusions....45

## CHAPTER 3 A PATH-BASE FRAMEWORK


3.1 Introduction ..... 46

3.2 I/O, weight state, role of hidden units and training feasibility ..... 46

3.2.1 The role of hidden units ..... 47

3.2.2 How various conditions can be used for justifying goal weight existence 48

3.2.3 Whether training feasibility is a problem ..... 55

3.2.4 Concluding remarks ..... 59

3.3 Required features for time-dependent signal processing (TDSP) ..... 59

3.4 Common problems using SBP ..... 60

3.5 A path-based framework ..... 66

3.5.1 An abstract machine analogy ..... 67

3.5.2 The role of goal weight states in neural convergence ..... 68

3.5.3 Major specific features of the path-based approach ..... 69

3.6 Conclusions ..... 71


## CHAPTER 4 A FEEDFORWARD CONTINUOUS BACKPROPAGATION MODEL


4.1 Introduction ..... 72

4.2 FCBP model ..... 72

4.2.1 the role of hidden units in FCBP ..... 73

4.2.2 the relationship among input, weight and output ..... 74

4.3 Training speed and generalization capacity ..... 75

4.3.1 Training ..... 75

4.3.2 Generalisation ..... 77

4.3.3 The outline of FCBP ..... 82

4.3.4 The trade offs in FCBP ..... 83

4.4 The FCBP Training Schemes..... 84

4.5 The FCBP Generalization Schemes..... 85

4.6 Conclusions..... 86

CHAPTER 5 RECURRENT CONTINUOUS BACKPROPAGATION MODEL

5.1 Introduction..... 88

5.2 The RCBP model ..... 88

5.2.1 The internal state model ..... 89

5.2.2 How transitions can be designed to achieve dynamic behaviour in rcbp 90

5.2.3 The relationship between the network variables..... 96

5.2.4 Summary features of the RCBP approach :..... 97

5.3 The RCBP Training Scheme..... 98

5.4 The RCBP Generalization Scheme..... 100

5.5 Conclusions..... 102

CHAPTER 6 EXPERIMENTS AND ANALYSIS ON FCBP AND RCBP

6.1 Introduction..... 104

6.2 What FCBP can do without hidden units ..... 104

6.2.1 CXOR task description..... 105

6.2.2 Training CXOR without hidden units ..... 106

6.2.3 Training CXOR to explore the feature of stepping stones..... 108

6.3 FCBP training and generalization with hidden units..... 110

6.3.1 Task description and discussion..... 110

6.3.2 Training results and analysis..... 112

6.3.3 Generalization and results analysis....113

6.4 Comparison of FCBP and SBP....115

6.4.1 Training....116

6.4.2 Generalization....120

6.5 FCBP and single goal approach....124

6.5.1 Task description....125

6.5.2 Training....125

6.5.3 Results and analysis....126

6.6 RCBP training and generalization....128

6.6.1 Task description....128

6.6.2 Training and results analysis....130

6.6.3 Generalization results and analysis....132

6.7 Electrocardiogram (ECG) addressable memory....133

6.7.1 Task description....133

6.7.2 Training and results analysis....135

6.8 Conclusions....139

CHAPTER 7 CBPTOOL DESIGN CONSIDERATION & OVERVIEW

7.1 Introduction....141

7.2 Design Considerations....142

7.2.1 Design Purposes....142

7.2.2 Design features....142

7.2.3 Choice of the design environment....145

7.3 Interface Outline....145

7.3.1 Window, Menu and Mouse Interfaces....146

7.3.2 File interface....148

7.3.3 Function interface ..... 149
7.3.4 Special dumping interface ..... 150
7.4 Main Internal Representations and Implementations ..... 151
7.4.1 Considerations ..... 151
7.4.2 Implementation ..... 153
7.5 Main simulator parts ..... 160
7.5.1 The Design module ..... 160
7.5.2 The Build module ..... 161
7.5.3 Parameter setting module ..... 162
7.5.4 The Training module ..... 162
7.5.5 The Performance module ..... 162
7.5.6 The Display module ..... 163
7.6 Discussions ..... 163
7.6.1 System environment limitations ..... 163
7.6.2 Flexibility and Speed ..... 164
7.6.3 Graphic screen dumping ..... 165
7.6.4 Access speed ..... 165
7.6.5 Format or language ..... 166
7.6.6 further future facilities ..... 166
7.7 Conclusion ..... 166

CHAPTER 8 CONCLUSION AND RECOMMENDED FUTURE WORK

8.1 General remarks ..... 168
8.1.1 General achievements ..... 168
8.1.2 Limitations ..... 169
8.2 Recommended future work ..... 170

## APPENDIX 1 THE CBPTOOL USER MANUAL

1. Introduction .....172

2. Accessing the simulator.....172

3. Main Facilities and Parameters.....173

3.1 Windows .....173

3.2 Aid-Tools.....175

3.3 Menus.....179

4. Forms of the Specification Files .....197

4.1 Initial-Weight state Specification File .....198

4.2 Training-Pattern Specification file.....198

4.3 Path-Based-Performance-Pattern Specification file.....200

4.4 Time-Based-Performance-Pattern Specification file.....202

5. Display and Store the Results .....203

5.1 Displays.....203

5.2 Store into files.....208

## APPENDIX 2 EXAMPLE

The OR problem .....211

## APPENDIX 3 THE DATA

The training data of the ECGs.....213

REFERENCES .....217

# CHAPTER 1

# GENERAL INTRODUCTION

## 1.1 Introduction

• What are neural networks?

It is well known that the human brain contains massive numbers of rather slow processing elements called "neurons" (on the order of $10^{10}$ and $10^{11}$) that are richly interconnected (single cortical neurons can have average about $10^3$ to $10^5$ connections per neuron) (see, for example, Chapter 4 Rumelhart, *et al.*, 1986). The operational speed of each neuron is estimated to be only between 100Hz and 1000Hz. However, the human brain is capable of recognizing a visual scene and issuing a reaction within a fraction of a second, i.e., within about 100 cycles. It is also noted that on complex cognitive tasks the human brain easily performs better than the central processor unit of even the most powerful contemporary serial computers, which operate at frequencies of tens of MHz.

To many people the solution to this puzzle seems a "massive parallelism" and learning ability. Though individual components of the human brain are inherently slow, the system as a whole operates quickly, since many computations are carried out in parallel. In addition, the human brain has the ability to learn from examples. These are properties which distinguish it from an ordinary serial computer system, which has to be programmed to perform a meaningful task using one or a few central processor units.

In the past few decades, biologists and neurophysiologists have greatly improved their understanding of the organizational principles of brains. In the picture which has emerged, neurons are cells that can amplify and conduct electrical pulses. From the main body of each neuron a long fibre, the *axon*, emanates branching into a number of *dendrites*, which end on or near the bodies of other neurons. The coupling( or synaptic junction) between the dendrite and the next cell body may be such that an arriving nerve pulse has an

excitatory or an inhibitory effect on the recipient neuron. The facts that biological computation is so effective and the human brain can solve very complicated cognitive problems suggest that it may be possible to attain and emulate similar capabilities in artificial devices based on the design principles of neural systems.

As a subclass of neural system research, computer scientists are busy investigating the properties of parallel models of "computation" that can be embodied in *artificial neural networks*, sometimes called *parallel distributed processing* (PDP). This is to build neural networks based on simplified neural system features and emphasize computational power rather than biological fidelity.

Generally speaking, an *artificial neural network* is a dynamic, information processing system composed of a large number of simple processing elements called *artificial neurons* or *units* that interact one another using weighted directed connections called *links* which co-operate to solve a computational task. The adjective "artificial" will be dropped from now on and taken to be understood. A simplified general structure of the units is reviewed later in §1.4.

• Why neural networks?

A simplified biological model and neural networks share a common mathematical formulation as a system. Depending on which form of the interactions is embodied, this results in different network models. These are characterised by differences in their neuron behaviour, network topology and learning rules.

From a computer scientist's point of view, there are two main reasons put forward here for investigating neural networks:

(1) Firstly, as discussed above, these networks resemble the human brain much more closely than conventional computers. Even though there are many differences between artificial neurons and real neurons, neural networks are an analogy taken from the human brain. A deeper understanding of the computational properties of connectionist networks

may reveal some general principles that can be applied to a whole class of devices of this kind, including the brain. In this way it looks as though neural network models have a good chance at capturing some significant complexity of cognitive systems. Thus the study of neural networks often involves trying to understand the complex phenomenon called intelligence or to see how intelligence is embodied in brains and may be embodied in machines.

(2) Secondly, the recent technological advances in VLSI (very large scale integration) and computer aided design make it much easy to build massively parallel machines. Neural networks are massively parallel, so that computations can be performed efficiently with these networks making a good use of parallel hardware.

• What is this chapter about ?

In this chapter, a brief historical review of some of the earlier work on neural networks is given in §1.2 which gives a general picture of the development in this field. In §1.3, there is an explanation as to why sequential processing is investigated for parallel processing systems. In §1.4, some basic concepts related to the aspects of learning are reviewed. Finally, in §1.5 the thesis structure is outlined.

## 1.2 A view of historical development

The landmark paper of Warren McCulloch and Walter Pitts (1943) is often taken as the starting point of neural network research. McCulloch and Pitts considered networks with two state threshold elements and proved that every logical function could be implemented using these kinds of neurons (1943). Their results imply that any finite state machine can be simulated by a network of such neurons (Arbib, 1987).

In 1949, Hebb's learning scheme paved the ground for many neural network models of learning. The learning scheme is for formal neurons like those of McCulloch and Pitts in

which connections between neurons were strengthened whenever the neurons 'fired' jointly.

Enthusiasm for neural networks peaked for the first time when in the late 1950s Frank Rosenblatt and his colleagues at Cornell University invented the Perceptron which is a single layer neural network. Networks with multiple layers, however, were poorly understood at that time.

Further study of artificial neural systems almost stopped in the mid 1960s after Marvin Minsky and Seymour Papert, two pioneers of artificial intelligence, convincingly pointed out that the Perceptron was incapable of solving simple, yet important classification problems such as the well known "exclusive-or" problem (Minsky and Papert, 1969). By showing that **XOR** cannot be learnt through any single layer perceptron, Minsky and Papert had conclusively demonstrated a fundamental inadequacy of single layer perceptrons in representing general I/O mappings. However, as Minsky and Papert knew, it is always possible to convert any unsolvable I/O mapping problem into a solvable one in a multiple layer perceptron.

As the original perceptron learning procedure does not apply to more than one layer, in order to solve I/O mapping problems with intermediate layers, containing a kind of units called hidden units which are units not having direct network inputs or outputs, a new learning procedure is needed to make weights along input links for hidden units learnable. Minsky and Papert focused on the question of what preprocessing must be done by the units in intermediate layers to allow a task to be solved and believed that no general procedure could be found for learning with hidden units. Their book led to a fading of interest in neural networks generally.

Although reduced in intensity, research on neural networks was not abandoned totally. Undeterred by the lack of interest and support from the rest of the world, a few researchers in the areas of neurophysiology, biological control theory, cognitive psychology and

artificial intelligence still carried on their research into neural networks. Some achievements are listed in the Table 1.1.

After a period of time, training algorithms were developed for multiple layer networks (Werbos,1974; Le Cun, 1985; Parker, 1985; Rumelhart *et al.*, 1986), which could solve the type of the well known "exclusive-or" problems. These algorithms were generalisations of those for single layer networks. Since this breakthrough, together with other developments such as the dynamics of restricted classes of neural networks, neural networks have been theoretically analysed and better understood (see, e.g., Rumelhart, *et al.*, 1986; Hopfield, 1982, 1984; ). The interest in neural networks revived again and has reached the high level we can see today. Reprints of more than 40 important original research articles scattered among diverse journals were assembled by Anderson and Rosenfeld (1988).

As a selective review, three aspects related to the development in I/O associations in multiple layer networks with state-based backpropagation (BP) approach (see §1.4 about BP) should be mentioned:

• The generality and sufficiency of three layer feedforward neural networks for continuous function mapping have been theoretically analysed, recognized and proved (Hornik, *et al.*, 1989; Cybenko, 1989; Funahashi, 1989).

• The dynamics of recurrent networks have been explored (e.g. Almeida, 1987; Pineda,1987; Rohwer and Forrest, 1987; Robinson and Fallside, 1987; Williams and Zipser, 1988; Pearlmutter, 1989; Rohwer, 1990).

• The importance of sequential processing in neural networks has been realised by some workers and the approaches have been investigated (e.g. Jordan, 1986; Tank and Hopfield, 1987; Waibel *et al.*, 1987; Elman, 1988).

Only a small part of the achievements in neural network learning has been listed here. This gives us though a general idea about the major development of this field since 1943. For

more details about the historical development see those listed in Table 1.1 below in addition to the book (Rumelhart, *et al.*, 1986).

Table 1.1 Highlights in the development of neural networks (1943 ~ 1987)

| year | names | development |
|---|---|---|
| 1943 | McCulloch, W. & Pitts,W. | threshold logic neuron |
| 1949 | Hebb, D. | neuron learning rule |
| 1955 | Selfridge, O.G. | pandemonium pattern recognition model |
| 1957 | Kolmogorov, A.N. | function representation theorem |
| 1959 | Rosenblatt, F. | Perceptron |
| 1960 | Widrow, B. & Hoff, M. E. | Widrow-Hoff learning rule, ADALINE |
| 1961 | Steinbuch, K. | Learnmatrix |
| 1966 | Neumann, J.von | cellular automata |
| 1969 | Grossberg, S. | instar,outstar,avalanche,ART |
| 1969 | Minsky, M. & Papert, S. | theory of Perceptrons |
| 1970 | Anderson, J.A. | associative memory |
| 1973 | Anderson, J.A. | Brain-State-in-the-Box (BSB) |
| 1973 | Malsburg, C.von der | self-organization |
| 1974 | Kohonen, T. | associative memory |
| 1974 | Werbos, P. | error back-propagation (PhD thesis) |
| 1980 | Fukushima, K. | Neocognitron |
| 1982 | Hopfield, J.J. | stochastic binary Hopfield net |
| 1982 | Kohonen, T. | Kohonen feature map |
| 1983 | Kirkpatrick, S.; Gelatt, C. & Veccji, M. | simulated annealing |
| 1984 | Hopfield, J.J. | deterministic 'graded' Hopfield net |
| 1985 | Cun, Y.L. | learning in asymmetric nets |
| 1985 | Parker, D. | error back-propagation (rediscovery) |
| 1985 | Ackley, D.; Hinton, G. & Sejnowski, T. | Boltzmann machine |
| 1986 | Rumelhart, D.; Hinton, G. & Williams, E. | multilayer Perceptron |
| 1986 | Szu, H. | Cauchy machine |
| 1987 | Carpenter, G. & Grossberg, S. | Adaptive Resonance Theory (ART) II |
| 1987 | Hecht-Nielsen, R. | generality of three-layer network |

## 1.3 Sequential processing in neural networks

There are a number of areas in which neural networks can provide adequate models. One area I would like to enhance the capability of neural networks in is modelling sequential behaviour such as time dependent signal processing.

A common task for the Perceptron was to carry out visual character recognition. The sequence underlying the presenting of characters is not intended affect the recognition of those characters. Consequently, this kind of task does not strongly suggest that time should be incorporated into the neural framework. However, it is also clear that the processes of the human brain are not only highly parallel but also sequential. Hearing, for example, is a different task showing a basic human capacity where time sequences are more clearly involved. It is unlike the earlier visual recognition task, since hearing input sequences involves dependence between current and past inputs, and so temporal structure is involved in the hearing processing. This example suggests that time may need to be imposed in neural networks in such a way that assists and enhances the capabilities of sequential processing. Processing involving temporal structure will be called sequential processing henceforth throughout this thesis.

Attention has been paid to the problems of learning sequences in neural networks (e.g. Elman, 1988). However many of the associated problems remain unsolved or at least not fully investigated. It will be put forward that this is at least partly because BP-based learning has typically taken a single weight state to be the result of learning regardless of whether the learning is to approximate an I/O function, dynamic system or I/O associations with an underlying temporal structure.

The question being raised is what constitutes general justification for the weight state approach and whether it is justified in all cases. One justification arises from the general principle that with enough hidden units a single weight state can provide I/O mappings of any complexity (Cybenko, 1989; Funahashi,K., 1989; Hornik, K., Stinchcombe, M., and White, H., 1989).

Neural knowledge represented through a set of weights as parameters embodied within the system exists together with the system structure. Such knowledge as a set of parameters may be fixed for the long term and so not be intended as a function of time. If the knowledge is to be accessed at any time, this brings a random access feature to the system. Theoretically then, the single weight state approach has the capability to represent any complex long term knowledge and provide a system with random access and a generalization capability.

However, consider Simpson's definition of neural convergence (Simpson, 1989):" *if the mapping converges to a fixed value, or to some fixed set, then the learning procedure is properly capturing the mapping.* "

It would seem from the above that the single goal weight state approach in neural learning is only one of several approaches which might be able to solve tasks. In particular, there may be good reasons not to stick with the single goal weight state approach for solving various kinds of sequential processing tasks.

For example, when only sequential access is required after learning, neural performance may be viewed as drawing upon different knowledge sequentially. Each piece of knowledge can then be adapted to act as a set of parameters for a certain moment in sequence. In this way, an account of the temporal and sequential aspects of processing is used. In this case, there is no reason to restrict the system to find a single instantaneous knowledge state as the goal of learning.

In this thesis, efforts are made to explore a temporally based and analogue processing capacity in neural networks based on gradient descent methodology. A simulator to support the research has been designed and implemented. An approach with a more complete account of the temporal as well as the sequential aspects of signal processing will be shown conceptually and experimentally to improve the effectiveness and feasibility of neural simulations.

## 1.4 Basic concepts

Fundamental concepts of neural networks used in the thesis are briefly reviewed here. For more details see the papers and books (e.g., Lippmann, 1987; Rumelhart et al, 1986; Crick,1989; Kinoshita et al, 1987; Tazelaar, 1989).

### • An artificial neuron

In a neural network, each unit generally has an output activity that is determined by the input received from other units in the network.

There are many possible variations within this general framework. One common, simplified assumption is that the combined effects of the rest of the network on a unit are mediated by a single scalar quantity. This quantity, which is called the excitation of a unit is usually taken to be a linear function of the output activity of the units that provide input to the unit:

$$x_j = -\theta_j + \sum_i y_i w_{ji} \qquad (1.1)$$

where $x_j$ is the total input to unit $j$; $y_i$ is the output activity of unit $i$; $w_{ji}$ is the weight value on the connection from unit $i$ to unit $j$ ; $\theta_j$ is the threshold of unit $j$. The threshold term can be substituted by giving every unit an extra input connection from a common unit called *bias* whose activity level is fixed at 1. For unit $j$, the weight on this special connection is the negative of the threshold $\theta_j$. An artificial neuron is shown in Fig.1.1.

The output activity $y_i$ of unit $i$ can be defined to be a linear or nonlinear function of its total input excitation $x_i$. For units with discrete states and a threshold $\theta$, this function typically has value 1 if the total input is great than $\theta$ or 0 otherwise. For units with real-valued states, a typical linear input-output function is: $y_i = x_i$, a typical nonlinear input-output function is the logistic function: $y = f(x) = 1/(1+e^{-x})$ (Fig. 1.2). Because the latter function is monotonic and "S-shaped", it is often referred as a sigmoid function.

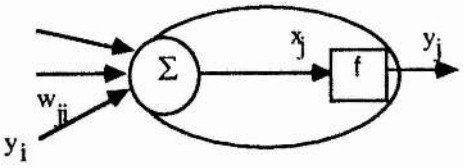



Fig. 1.1 An artificial neuron

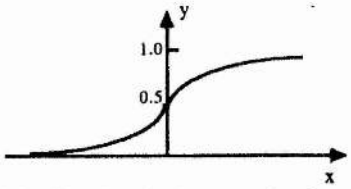



Fig. 1.2 The logistic transfer function

• Perceptron procedure

This is a simple learning algorithm for the Perceptron (Rosenblatt,1962). If the output of a perceptron unit $i$ is $y_i$, we have:

$y_j(t) = f_h [\Sigma_i w_{ji} y_i(t)]$

where $f_h$ is a step function $f_h(x)$ is 1 if $x > 0$; $f_h(x)$ is 0 otherwise. We adapt weights $w_{ji}$ at time $t$+1 according the following rules ($d_j$ denotes the target value of unit $j$):

| | |
|---|---|
| if $(y_j = d_j)$ | then $w_{ji}(t+1) = w_{ji}(t)$; |
| if $(y_j = 0$ but $d_j = 1)$ | then $w_{ji}(t+1) = w_{ji}(t) + y_i(t)$; |
| if $(y_j = 1$ but $d_j = 0)$ | then $w_{ji}(t+1) = w_{ji}(t) - y_i(t)$; |

• Delta rule

This rule, proposed by Widrow and Hoff in 1960, uses the difference between the desired, or *target*, activity and the obtained activity, which is called the *delta*, to drive learning in a network. The idea is to adjust the strengths of the weight connections so that they will tend to reduce this *difference* or *error* measure. The rule in its simple form can be written as:

$$\Delta w_{ji} = \varepsilon \, \delta_j \, o_i \qquad (1.2a)$$

where $\varepsilon$ is a constant of proportionality called the learning rate, $o_i$ is the output of unit $i$ which is input to the link $l_{ji}$. $\delta_j$ is the delta for a linear unit $j$ given by :

$$\delta_j = t_j - o_j \qquad (1.2b)$$

which is the difference between the teaching activity $t_j$ (to output unit $j$) and its actual activity value $o_j$. There is also though the feature of many patterns being learnt by the network so that the rule becomes:

$$\Delta w_{ji} = \varepsilon \Sigma_p \, \delta_{jp} o_{ip} \qquad (1.2c)$$

where $\delta_{jp}$ and $o_{ip}$ are defined for each pattern $p$ , where the index $p$ ranges over the set of the input patterns, and $i$ refers to the source unit for the link $l_{ji}$.

The delta rule is very similar to the Perceptron procedure for networks with threshold units, the differences are only that units with real-valued outputs instead of linear threshold units are used in training.

In the perceptron, the error signal is used in the calculation of the modification of weights and is equal to the binary difference between the weighted input sum *after* thresholding and the desired result. There is no distinction made between the performance and the learning rule as far as thresholding is concerned.

In the delta rule, there is a distinction. During training, the error signal is equal to the difference between the weighted input sum *before* thresholding and the desired output. During performance, the same linear thresholded units are used as in the perceptron, with an output of +1 if the weighted sum of its inputs is larger than the threshold or -1 otherwise. In other words, the delta rule modifies connection weights when the weighted sum of the inputs of the neuron is not exactly equal to the binary target, even if the thresholded response is correct.

• Gradient descent

In general, when a function $f$ depends on one or more independent variables $v_i$ ($i$=1,2,...$n$), gradient descent as a minimisation method can be used to find the values of those variables where $f$ takes on a minimum value. The method makes changes in the $v_i$

proportional to the negative of the derivative of the function $f$ with respect to the variables $v_i$:

$$\Delta v_i := - k \frac{\partial f}{\partial v_i} .$$

In neural networks, gradient descent is used to find a minimum value for the error function $E$, which depends on some independent variables. In techniques such as BP, we want to find the values of weight variables $W$ where $E$ takes on a minimum value.

• Least-Mean-Square procedure

LMS is an error measure which has been applied by Widrow and Hoff (1960) to give a version of the delta rule. The procedure makes use of the ideas of the delta rule, LMS error and gradient descent for learning and adjusting connection strengths.

The total error of such a one layer linear network can be defined by a simple error function as in Eq.1.2b or by a quadratic LMS error function such as :

$$E = \sum_p E_p = \sum_p \sum_i (t_{pi} - o_{pi})^2 \qquad (1.3a)$$

where $t_{pi}$ denotes the target value of output unit $i$ for pattern $p$. Gradient descent can be then used to find a weight state that minimises the function $E$. The procedure is that after each pattern has been presented, the error-weight derivatives for that pattern are computed. The total error-weight derivatives of the patterns are used for making each weight moving down the error gradient toward its minimum value for all the patterns. The learning rule is:

$$\Delta w_{ij} = - k \frac{\partial E}{\partial w_{ij}} = -k \frac{\partial E}{\partial o_i} \frac{\partial o_i}{\partial w_{ij}} = 2k \sum_p (t_{pi} - o_{pi}) o_{pj} \qquad (1.3b)$$

when $o_i = \sum_j o_j w_{ij}$ (for a one layer linear network).

• Network topology

The functional ability of a neural network is largely dependent on its net topology, i.e. the number and arrangement of interconnections between the units. Depending on their functions within the network, units can be grouped into three types which are: the *input*

*units*, which receive inputs from the network's environment; the *output units*, which have associated teaching or target patterns; and the *hidden units*, which neither receive environment inputs directly nor give direct output to the environment.

Depending on the linking relationship among the units, there are two major different types of networks: feedforward networks and recurrent networks. Networks may not necessarily have all the above three kinds of units.

• Feedforward networks and recurrent networks

A feedforward network is a network with all links being feedforward. This implies that all units are linked from units in a lower layer to units in a higher layer (closer to the output layer), no links between units within the same layer. The structure of a fully connected general feedforward network is shown in Fig. 1.3a. Where each unit of each layer is linked feedforward to the units in all the higher layers. Feedforward networks may not necessarily have all those links.

When a feedforward network's unit of one layer is only linked to the units in the next layer and not to any units in the other higher layers, the feedforward network is a strictly layered network. As an example, the structure of a strictly layered network is shown in Fig. 1.3b.

There are no above linking limitations in recurrent networks. In recurrent networks, there is at least one unit whose output can directly or indirectly feed back into the unit. The structure of a recurrent network is shown in Fig. 1.3c for an example.

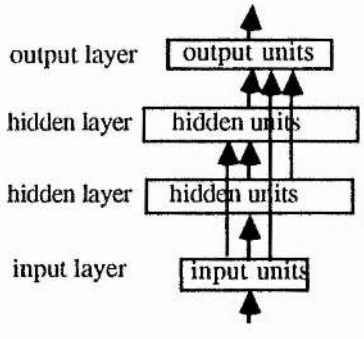



Fig. 1.3a A general feedforward network

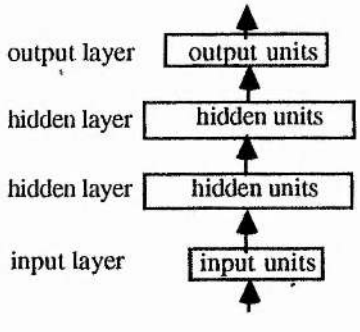



Fig. 1.3b A strictly layered feedforward net

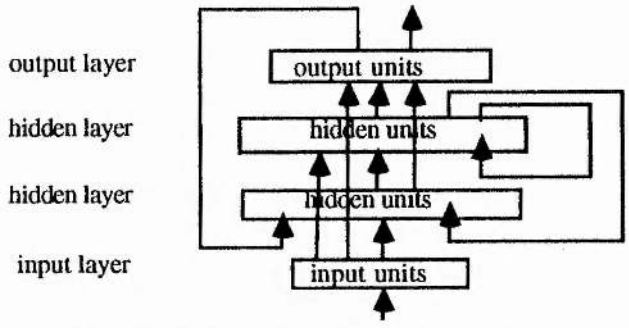



Fig. 1.3c A recurrent network structure

• Backpropagation learning procedure

The "backpropagation" (BP) learning procedure is a generalization of the LMS procedure that works for non-linear networks which have layers of hidden units between the input and output units. The basic idea of the learning method is to combine a nonlinear perceptron-like system capable of making decisions with the objective error function of LMS and using gradient descent method to minimise the error by finding a set of suitable weight variables.

Variants of the BP procedure were discovered independently by Werbos (1974), Le Cun (1985), Parker (1985) and Rumelhart, Hinton & Williams (1986). They are going to be referred as conventional BP throughout this thesis, all approaches using the BP idea based on a single goal weight state approach will be termed as state-based backpropagation (SBP).

• Batch and on-line

The "batch" version of BP sweeps through all the cases of I/O training tuples accumulating the measure of the derivative of the error function with respect to any weights in the network $\partial E/\partial W_{ij}$ before changing the weights. This is guaranteed to move in the direction of steepest descent at the current weight state.

The "on-line" version, which requires less memory, updates the weights after each input-output case. This version can be made as an arbitrary close approximation of the steepest

gradient descent after a complete sweep through all the cases provided each of the weight changes is sufficiently small.

• Local and global (non-local)

These two terms are used to point out what kind of information are needed when a computation is carried out.

Local in computation implies that each unit requires information only from other units to which it connects. When these information are called local information, global in computation implies that some non-local information are needed.

• Error- weight space and error surface

The LMS learning procedure has a simple geometric interpretation. An $n+1$ multiple dimensional "error-weight space" can be constructed in this way where $n$ is the total number of the weight links in a network : there is an axis for each weight, and one extra axis corresponding to the error measure.

For each combination of weights, there is one weight state in weight space. The network will have a certain error for current inputs which can be represented by the height of a point above a weight state in weight space. These points form a surface called the error-weight surface.

• I/O tuple, I/O path and training position

An I/O tuple is a list of input and output values. Each of the values is associated with an output activity of an input or output unit in the network.

An I/O path, as used throughout the thesis, is comprised of an infinite sequence of I/O states. Each of the states is an I/O tuple at a certain time. At an instant in time, an I/O tuple in the path is seen whose values are the current associated signal values at that instant.

Each of the individual I/O tuples occurring at the same evolved fractional distance in time along each of a number of I/O paths are at the same training position. The fractional distance is said to constitute a training *position* in the I/O paths' state sequences. There are infinite number of the I/O states along each path but only finite number of them are tested or used, which consist of the sequence of the I/O states.

A diagram in Fig. 1.4 shows the relationship between training positions and I/O paths. Suppose the training positions are at *t1*, *t2* and *tk* along the time axis. The input axis shows the input values of the paths at each moment associated with three fixed outputs along the paths respectively.

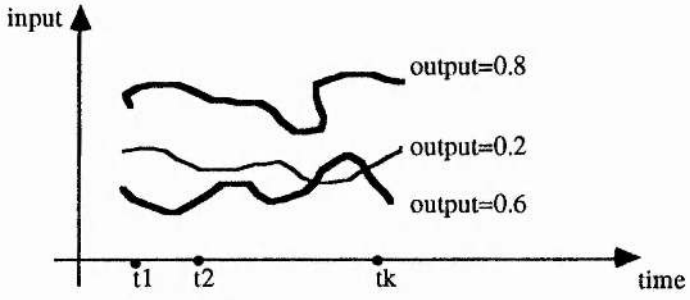



Fig. 1.4 A diagram of showing the relationship between paths and time

• Fixed-point and non-fixed-point algorithms of recurrent networks

Fixed-point algorithms enable kinds of dynamic behaviour which are designed to have the networks converge to some stable fixed-points in error-weight space. In particular, when an input pattern is given either as an initial condition (when a network has no input units) or as a constant external input, the response of the network is taken to be the output state of the network once it has reached its fixed-point.

Non-fixed-point algorithms on the other hand are able to learn non-fixed-point attractors in time and to produce desired sequential behaviour over a bounded interval.

• Time dependent signal processing

In this thesis, time dependent signal processing is defined as processing which involves arbitrary approximation of I/O associations chosen along a number of I/O paths.

• Weight state path and goal weight state path

The weight state path, as used in the thesis, is an infinite sequence of weight states consisting of finite goal weight states and infinite interpolated weight states. Each of the state is a set of weight values; each value is associated with each link in the network.

Each goal weight state belonging to the weight path is associated with each trained I/O training position. Such a state enables the network to act as a machine which produces the correct outputs for any inputs associated with the position. Each interpolated weight state belonging to the weight path is associated with each untrained position, it also enables the network to act as a machine which approximates the correct outputs for any inputs associated with the position.

The goal weight state path is an ideal weight state path, each state along the path is the goal weight state of the associated training position.

• Temporal association

Temporal association is that a particular output sequence is produced in response to a specific input sequence.

### 1.5 Outline of the thesis

This thesis includes the review, analysis, design, implementation and empirical assessment of learning models for solving problems involving sequential I/O associations where analogue values may be involved. The models involve multiple layer networks based on backpropagation as the core part of the learning method. The efforts made in the

approaches are purely conceptual and methodological using artificial neural networks, no other biological or psychological plausibilities are considered.

In Chapter 2, a general review is given to assess the capability of the existing neural models in learning sequences. A detailed discussion on ten related models is presented. This gives an insight of what is the inherent sequential processing capacity of those models.

In Chapter 3, It is argued that a new framework is needed. This is through both reviewing related features of signal processing and analysing the inherent infeasibilities in training and generalization for arbitrary approximation I/O signal associations underlying continuous analogue functions using SBP approach. The philosophy of a new path-based framework investigated in the thesis is presented.

In Chapter 4, a new approach using the new path-based framework called feedforward continuous back-propagation (FCBP) is presented. The aim of the FCBP approach is to provide a means for achieving arbitrary approximation of analogue signal associations within a fixed neural topology. The notion of goal weight sequences is introduced and applied; the training and generalisation capabilities of FCBP are analysed; the training and generalization schemes of FCBP are given.

In Chapter 5, another path-based approach called recurrent continuous backpropagation (RCBP) is presented. RCBP is a kind of path-based dynamic system. The notion of activity sequences is introduced; the design and implementation details of the model are described; the training and generalization schemes of RCBP are given.

In Chapter 6, several experiments based on FCBP and RCBP are presented and the results are analysed. These experimentations are chosen not only to show the features of the two models but also to demonstrate the capabilities and benefits for training and generalization by employing the concepts and methodologies embodied in the two new models.

In Chapter 7, a simulator called **cbptool** is introduced here. This chapter is a guide of how the design of **cbptool** has been evolved from conception and requirements for

implementation. A detailed description of the functions, designing, internal representations and user interfaces of the tool is given. Possible improvements in the design of the tool are also discussed.

Finally in Chapter 8 a general conclusion is made based on the thesis. This includes reviewing the general points of the research and recommending some future work.

# CHAPTER 2

# SEQUENTIAL PROCESSING USING STATE-BASED BP MODELS

## 2.1 Introduction

As it is discussed in §1.3, this thesis is about investigation of the capability of neural networks in dealing with sequential processing. Many approaches have been investigated to strengthen this capability in neural networks. Three main kinds of approaches have been used to explore the problem of learning sequences using neural gradient descent methods. The first one is a simple spatial approach. This is to represent time as an explicit part of inputs (§2.2.1). The second kind is to impose dynamic features in networks through special network topology (§2.2.2). The third kind of approach is to employ complex dynamics to train networks to be dynamic systems which can continuously react to the sequences of inputs for temporal associations (§2.2.3).

This chapter is arranged to review the existing approaches related to those aspects. Firstly, in §2.2, various general approaches related to time representation are outlined. Secondly, in §2.3, several specific models which attempt to realise learning sequences are reviewed. It is hoped that this review gives an insight into what the inherent sequential processing capacity is of those models. Finally, a conclusion is presented in §2.4.

## 2.2 TIME REPRESENTATION IN NEURAL NETWORKS

Neural networks are parallel distributed processing systems. Because of the parallelism, learning sequences implies a time representation problem (Elman, 1988).

In the conventional serial processing computer system, the question of how to represent time interacting with sequential tasks generally does not arise. There is no such question

because sequential tasks are processed step by step in succession where sequences of processing represent sequenced events. Compared with the traditional serial computer system, neural networks have a major difference in dealing with sequences. For traditional computer systems, a finite internal state representation is automatically present. In neural network models, a time representation problem needs to be solved explicitly for learning tasks which involve internal state sequences. In general there are three types of tasks related to the problem of learning time sequences in neural networks, which are sequence recognition[1], sequence reproduction[2] and temporal association. Temporal association is a general case, it includes pure sequence generation and the previous two cases as special cases.

Many methods for implementing the above three tasks in neural networks have been investigated. In this thesis only the approaches based on the SBP framework will be concerned. It can be seen that the way of representing time in those SBP based models is various. This section will discuss the three major methods mentioned in §2.1: representing time spatially; imposing internal states using special network topology; embodying more complicated dynamic features in fully recurrent networks.

### 2.2.1 THE SPATIAL REPRESENTATION OF TIME

This method is to treat the effect of time as an explicit part of the inputs, in other words to represent the effect of time as additional inputs. Typically this approach uses a pool of input units for the event presented at time **t**, another pool for **t+1**, and so on in what is

---

[1] Sequence recognition is that a network produces a particular single output value when a specific input sequence is presented.

[2] Sequence reproduction is that a network is able to generate the rest of a sequence itself when it sees part of the sequence.

often called a 'moving window' paradigm. Time is represented explicitly by associating the serial order of the pattern with the dimension of the pattern vector. The first temporal event is represented by the first element in the pattern vector, the second temporal event is represented by the second element in the pattern vector and so on. Each entire window is thus processed as a single parallel input tuple by the model (Fig. 2.0). A model called time-delay neural network (TDNN) has been investigated based on this idea which will be reviewed in §2.3.1.

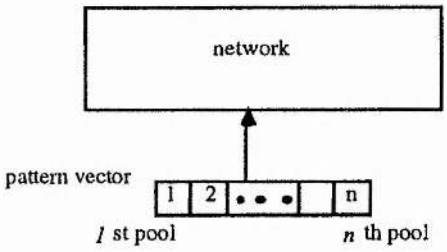



Fig. 2.0 The diagram of spatial representation model

This spatial representation method may be used for solving some sequence recognition problems as it can produce a particular output pattern for a number of input sequences with a fixed length. This method has been applied to speech recognition (e.g., McClelland and Elman, 1986; Cottrell, Munro & Zipser, 1987; Waibel et al., 1989 ). However, the system needs to memorise an entire fixed length sequence which has proven itself to be unsatisfactory in solving sequence recognition problems in general. This is because many tasks do not know the length of all related sequences, or do not have all sequences with a fixed length. As Elman (1988) pointed out, such implementation is not psychologically satisfying, and it is also computationally wasteful since some unused pools of units must be kept available for the rare occasions when the longest sequences are presented. Therefore, the models based on this time representation can only be applied to a limited number of real tasks. This representation is thus not a necessarily natural or practical method for analogue temporal structure tasks in neural networks.

### 2.2.2 DYNAMIC SYSTEM APPROACHES – SPECIFIC TOPOLOGIES

There are other two major methods used for learning time sequences. Both are trying to give processing systems some designed dynamic properties to represent temporal states or to represent time by the effect which time has on processing. This is to have the effect of time implicitly in a network by introducing some internal states for the network.

One of the methods is to approximate dynamic systems through specific network topologies that have internal states to represent time sequences. The other is to approximate dynamic systems by providing the processing system with more complicated dynamic rules than the topology approach.

As an example, the following shows how internal states can be imposed and used effectively in a specially designed network. Suppose a two input unit network is required to do sequential binary addition for two binary input sequences. The structure of the network has been designed and shown in Fig. 2.1, with a suitable learning algorithm described in §2.3.2.1. In this network, two inputs are used for representing the current input values from the two binary sequences, the previous inputs and their effects in time can be aggregated in a carry unit $U_c$ by its state so that the inputs from the immediate past are available as well as current inputs; that is, delayed inputs are available. The system thus decides the output at time $t$ according to the two explicit inputs at $t$ and the implicit input from the internal state, which is the state of the carry unit.

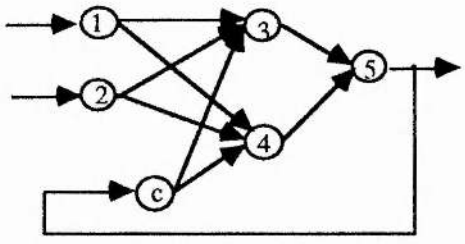



Fig. 2.1 The network for two string binary addition

From the understanding of what the model can be used to learn to be instead of how to learn, some discussions can be made based on this kind of neural model.

From the topology structure used in the above example, it is noticed that networks with a limited set of carefully chosen recurrent connections are very interesting. When output from a network system is fed back into the network as an extra input, this may enable the network to learn and to choose the next output in a way at least partly based on the previous one. These recurrent links (here the link from $U_5$ to $U_c$ ) can help to provide internal states for the networks and hence a dynamic system even when the simple learning rules applied in the associated feedforward networks are used. Some models which have been investigated for this purpose will be discussed in §2.3.2.

### 2.2.3 DYNAMIC SYSTEM APPROACHES – SPECIFIC DYNAMIC RULES

As mentioned in the last section, another method for dealing with temporal structure problems is to further study the dynamic features embodied in fully recurrent networks. This is one of the steps towards representing more complicated temporal structures in neural networks.

The following is a review of why a recurrent network is needed for this dynamic system approach. If a network is able to do one-many[3] associations, it implies there are some internal states in the network. If we view the activity value of each neuron in a network as the only source of the internal states of the network, when the network can do one-many associations, this implies that some neurons' activity states have many different values associated with the same input tuple. According to the conventional rule of evaluating activity values in feedforward networks, each of the neurons will always have the same

---

[3] Here we define that an one-many association in neural networks is that for a single input tuple, the networks have different output tuples associated with it.

activity value for the same weight state when the inputs of the network are the same, so that there is no such kind of internal state in any feedforward networks, no feedforward networks can have one-many associations in SBP.

However in SBP networks can have some internal states when there are some extra inputs beside the inputs from input neurons to networks. Some dynamic features may be imposed into networks with recurrent links to have this capability. The major features of recurrent networks can be concluded as follows:

A logistic gradient-descent approach in Cartesian coordinates for recurrent networks is an approach based on non-linear system with dynamic features. Exploration of recurrent networks is to study dynamic systems of neural network sort. Recurrent links in a network may allow the network to have internal states and produce complicated, time-varying outputs in response to a simple input. So recurrent networks may be used to approximate not only a formal automaton but also more complicated potential phenomena.

Note that recurrent networks intuitively can be used for embodying internal states and representing time dynamically through imposing on a suitable dynamic rule. However the network topology is not the sufficient condition of having a dynamic system. Pineda (1987), Almeida (1987), Rohwer and Forrest (1987) have independently derived an equivalent algorithm called **recurrent back-propagation** for fixed-point recurrent networks. This is a learning algorithm based on recurrent networks, and a step to show that conventional back-propagation can be extended to arbitrary networks. Because this algorithm is only for recurrent networks which can converge to stable states, this model cannot be used for problems related to learning time sequences. It can be seen that the dynamics imposed on a network will decide the capabilities of the network.

Two existing approaches mentioned in §2.2.2. and §2.2.3 can be concluded as this: the first one is a simple rule based dynamic system approach which is based on some specific designed network topologies; the second one is to impose various dynamic rules and study complex dynamic features by making full use of the inherent dynamic properties in a fully

recurrent network. In other words, the first is a topology based approach which embodies internal states through specially arranged recurrent links and context units. The second one is a rule based approach which is to control the approximation of target trajectories by imposing different ways of error propagation through time.

One common feature of the above three approaches is that they are all trying to represent time sequences instead of to model the sequences in the parallel processing systems. Models based on the three approaches for learning time sequences have been proposed and investigated. Next section is arranged to review those typical models.

## 2.3 MAJOR APPROACHES

Particular methods according to the types described in §2.2 are reviewed here.

There are two major common features in all these existing approaches reviewed here: (1) After training finished, all the networks converge to a single weight state; (2) gradient-descent is used as an error correction method. Differences between these approaches include: different dynamic rules are used by those models; some of the approaches are on-line and some are on batch; some of the approaches are based on discrete time and some based on continuous time; some are local and some are global. The major features of each of these approaches will be reviewed next.

### 2.3.1 TIME-DELAY NETWORKS

As reviewed in §2.2.1, the simplest way to perform sequence recognition is within the paradigm of spatial representation: this is by turning the temporal sequence into a spatial pattern on the input layer of a network. A feedforward network and the backpropagation learning algorithm can then be used to learn and recognize sequences.

A resulting model called a time-delay neural network (TDNN) has been used by many people for speech recognition (e.g. Waibel *et al.*, 1987; Lang & Hinton, 1988). TDNN

can have two dimensional time representation, one from the spatial representation of input patterns and another from the whole pattern shifting $k$ times.

For example, the values $x(t_i)$, $x(t_i-\Delta)$, $x(t_i-2\Delta)$, ... $x(t_i-(k-1)\Delta)$ from a signal $x(t)$ will be presented simultaneously at the input of a network with $k$ time delay as one pattern for training. In a practical network, these values could be obtained by feeding the signals into a temporal window with a fixed size $k$ and position for each time slice as shown in Fig. 2.0.

A TDNN can be implemented in a replicated feedforward network trained under constraints. The replication implies that each input unit has multiple copies (each associated with a particular time step) and uses a separate weight from each of the copy to each hidden unit, each link carries information about activation at a particular time step, to impose a kind of temporal window on the system — i.e. on the size of the number of time delay links. The network can be trained under constraints which ensured that the multiple copies of each unit applies the same set of weight to each part of the temporal window.

During training, all weights associated with different time frame but the same time delay will have the same set of values, this constrained training in a replicated feedforward network can be achieved through using a regime very similar to the conventional backpropagation (e.g. Lang & Hinton, §4.3, 1988; Lang, 1989) or the regime applied in the *backpropagation through time* (§2.3.3). Since a hidden unit applies the same set of weights at different time frames after training, it can then produce similar responses to similar patterns that are shifted in time.

For example, a set of sequences is with 12 time steps, suppose the number of the time delay is 3 (or the size of the temporal window), then 10 training patterns need to be trained for each of the sequences in TDNN using a multiple layer replicated feedforward network shown in Fig.2.2 using constrained training algorithm, three sets of weights between input and hidden units can then be applied in performance.

Although the power of this model was demonstrated by showing a better performance than all previously tried techniques on some speech applications (Waibel, 1987), several drawbacks to this general approach to sequence recognition were also reported (Mozer, 1989).

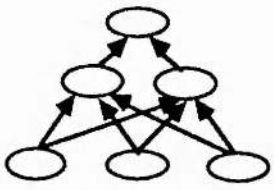


Fig. 2.2 The constrained weights feedforward net

The length of the delay must be chosen in advance to accommodate the longest effective sequence because the length determines how many context information can be provided for recognition. This model cannot be used for sequence recognitions with arbitrary length of context. And perhaps most important in signal processing, the input signal must be properly registered in time and arrives at the exactly correct rate. This model may thus only be used for solving some particular sequential recognition problems.

### 2.3.2 PARTIALLY RECURRENT NETWORKS

Another way to recognize and reproduce sequences is using partially recurrent networks. These networks use special topologies as described in §2.2. Three models are reviewed here.

#### 2.3.2.1 Jordan networks

Jordan (1986) described a network containing the feature of internal states in a specially designed partially recurrent network.

The general structure of this kind of network is shown in Fig. 2.3a; a network showing more unitary details of the structure in Fig.2.3b ( not all links are shown). It can be seen that the basic structure of the Jordan networks is similar to a feedforward layered neural

network. The additional part is to augment the basic feedforward layered structure at the input level with some additional units called *state units* which provide limited recurrence. The number of the *state units* is equal to the number of the output units so as to buffer the output state of the output units. This is achieved when each *state unit* receives a connection with a fixed weight of 1.0 from its corresponding output unit. At the same time, the *state units* also send connections with learnable weights to all hidden units and with a chosen weight of $\mu$ to all *state units* (including self-connection) to provide states. Except those specially arranged links, all the other links of the network are learnable and the weights can be modified according to the conventional back-propagation learning rule.

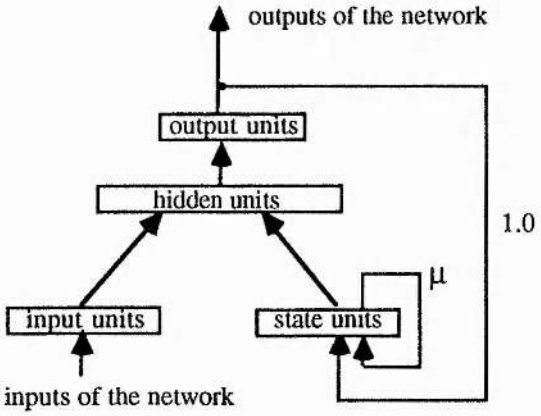



Fig. 2.3a A Jordan Recurrent Network

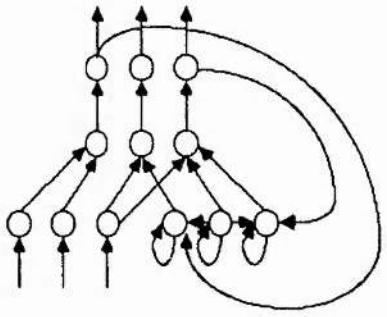


Fig. 2.3b The structure of the Network

In the network, the output states of the *state units* are derived from their own outputs and those of the output units on the previous time cycle. This allows the current state to depend on the previous state and on the previous output. The state at time $t$ is given by:

$$s_t = \mu\, s_{t-1} + y_{t-1} \tag{2.1}$$

where $s_t$ and $y_t$ denotes respectively the state vector and the network output vector at time $t$; $\mu$ ($\mu<1$) is the strength of the self connections. Iterating Eq. (2.1), we have:

$$s_t = y_{t-1} + \mu y_{t-2} + \mu^2 y_{t-3} + \ldots \mu^{t-3} y_2 + \mu^{t-2} y_1 + \mu^{t-1} y_0 = \sum_{r=1}^{t} \mu^{r-1} y_{t-r} \tag{2.2}$$

It can be seen that the self connections give *state units* themselves some individual memory. According to Eq. (2.2), $\mu$ value helps to define the state as an exponentially weighted average of past outputs, so that the arbitrarily distant past has some representation in the state. The value of $\mu$ decides the decay rate of past information. By making $\mu$ closer to 1 the memory can be made to extend further back into the past. In general the value of $\mu$ should be chosen to suit the features embodied in the input sequences (Stornetta *et al.*, 1987).

In this way *state units* memorise outputs of the network at the previous time cycle; they act as internal states imposed in the network. The known internal states which can be learnt during training then can be used in the network. The current network outputs are dependent on not only the current inputs but also the internal state — the previous state. This enables the subsequent behaviour to be shaped by previous responses.

This kind of network can be used to solve certain tasks which involve learning and representation of information contained in sequences. Some successes have been reported when the network is trained to generate a set of output sequences with a fixed input pattern; to prompt different output sequences with different input patterns (Jordan, 1986); to distinguish different input sequences (Anderson et al., 1989).

In this model, the internal states to be retained by the network across time must be manifested in the desired outputs of the network after a certain time. This is because the internal states are produced through the actual outputs of the network (see Eq.(2.1)). This means that only a certain type of temporal structures can be represented in the networks.

#### 2.3.2.2 Elman simple recurrent network

Another kind of partially recurrent network architecture has been studied based on Jordan's approach. In 1988, J. Elman investigated what he called a *simple recurrent network* (SRN). A general structure is shown in Fig.2.3c.

The basic structure of SRN is also similar to a feedforward layered neural network. At the input level there are also some additional units which provide limited recurrence similar to that in Jordan networks, but they are called *context units*. The number of the *context units* is equal to the number of the hidden units so as to act as a buffer with the copy of the output state of the hidden units. Each context unit receives a connection with a fixed weight of 1.0 from its corresponding hidden unit and sends connections with learnable weights to all hidden units. This network copies the output values of the hidden units into the set of *context units* to encode sequential structure in the network.

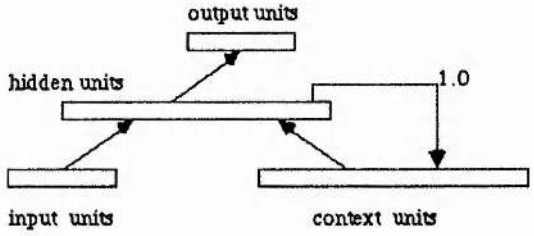



Fig. 2.3c Simple Recurrent Network

In Jordan's network the state of the *context units* consists of the outputs of output units of the previous time cycle and self-connections. In SRN hidden units are used to represent a compressed form interior structure. This is that the state of the *context units* was derived from the outputs of hidden units on the previous time cycle. In contrast to the output units, as the hidden units are not taught to assume specific values, this means that they can develop representation, in the course of learning a task, which encodes the temporal structure of the task.

When using this kind of neural network architecture, the output values of the context units at time **t+1** are the same as the output values of the hidden units at time **t**. Thus the context units are able to remember the previous internal state. The context units provide the network with memory by developing internal representations which are sensitive to temporal context. The hidden units have the task of mapping both an external input and the previous internal state to some desired output.

The SRN model has been applied to some tasks which involve sequence recognition. It has been shown in D. Schreiber's paper (Servan-Schreiber *et al.*, 1988) that a SRN could learn to be a finite state recogniser for a grammar. This is because the encoding of sequential structure depends on the fact that back-propagation enables hidden layers to encode task-relevant information (using hidden units to compress temporally interior structure). In the network, internal representations encode not only the prior event of the network state, but also the relevant aspect of the representation that was constructed in predicting the prior event from its predecessor. When fed back as inputs, these representations provide information that allow the network to maintain prediction-relevant features of an entire sequence. The hidden unit patterns can possibly achieve an encoding of the entire sequence of events presented with finite length. Therefore with enough hidden units, like some other approaches, this model can be applied to train a network to be a finite state automaton.

However, it is noted that the number of time steps of history being maintained relies on the number of the hidden units. So the question with SRN is whether the error from the history that has been cut off is significant. This question can only be answered respect to a particular task. This implies that in SRN the computational expense per time step scales linearly with the number of time steps of history being maintained, because the greater the number of steps maintained, the more hidden units are needed. This can cause problems for both the training feasibility or the amount of storage in SRN. So that it is very likely that the accuracy of approximation is gradually traded off against storage and computation in this kind of network.

#### 2.3.2.3 Stornetta network

Fig. 2.3d shows an architecture designed by Stornetta (1987), that can also perform sequence recognition tasks without changing the conventional learning rule. Note that the different between this approach and Jordan network is that now the inputs to the context

units are the external inputs and themselves, hence network external inputs only reach the rest of the network via the context units. This implies that the inputs to the network are preprocessed by the context units. This preprocessing serves to include past features of the inputs into the present context values, hence letting the network recognize and distinguish different sequences.

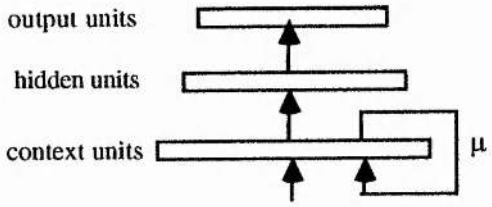



Fig. 2.3d A structure of Stornetta's network

Some other architectures along the line of the specific network topology approach have also been investigated. Because the scope of this thesis, more discussions about those models can be referred to the review paper given by Shimohara *et al.* (1988).

It is concluded that: through employing a set of specifically arranged recurrent connections, the partial recurrent networks can show some important features for imposing time structure into neural networks without changing the conventional feedforward learning rule. Similar behaviour can probably be obtained through any of the three partial recurrent networks discussed above. This is regardless of whether the feedback is from a hidden, a output or a context layer, though particular problems might be better suited to one rather than the other.

### 2.3.3 DISCRETE BACKPROPAGATION THROUGH TIME

The networks reviewed above for sequential processing are either feedforward or partially recurrent networks. Now let us see the models based on fully recurrent networks, in which each unit may be connected to any units.

It is retained that all models based on fully recurrent networks have synchronous dynamics; most are using discrete time; and the appropriate update rule is :

$$y_i\,(t+1) = \mathrm{f}(\,x_i(t)\,) = \mathrm{f}\,(\textstyle\sum_j w_{ij}\, y_j(t) + I_i(t)\,) \qquad (2.2)$$

where $y_i(t)$ is the output of the unit $i$ in the network at time $t$; $x_i$(t) is the excitation of the unit at $t$; $I_i(t)$ is the external input to unit $i$ at $t$; $f$ is the logistic function; $w_{ij}$ is the weight value linking from unit $j$ to $i$.

A general framework for learning in fully recurrent networks was laid out by Rumelhart, Hinton and Williams (1986), who unfolded the recurrent network into a multiple layer feedforward network which grows by one layer on each time step. This shows that the behaviour of a recurrent network within a certain time $n$ can be attained at the cost of duplicating the topology $n$ times over for the feedforward version of the network. Both the recurrent network and the feedforward network will behave identically for $n$ time steps. Using this model to produce temporal association in a small maximum length $n$ as an example, the outline of the model is reviewed as follows.

Supposed we are interested in sequences of a length $n$, for sequences spanning the time steps $t=1,2,\ldots\ n$, we simple duplicate all units in the recurrent network $n$ times to have an associated feedforward network. For each layer of the feedforward network, each unit in the recurrent net has a copy in the layer. The source and destination of each link is unchanged for any links associated with any two units in the recurrent network. However, the links are from the units belonging to the layer $i-1$ to the current layer $i$ in the forward network. In this way, a separate unit $U_i^t$ in the feedforward network holds the state $U_i(t)$ of the equivalent recurrent network at time $t$. Fig. 2.4b shows the idea for a general two unit recurrent network (Fig. 2.4a) for 4 time steps. Note that the weight value $w_{ij}$ from $U_j^t$ to $U_i^{t+1}$ is independent of $t$. That is, the same weight values must be used in each layer.

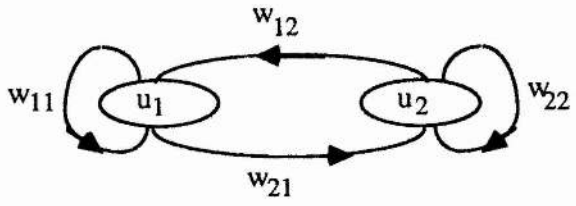



Fig. 2.4a The recurrent network

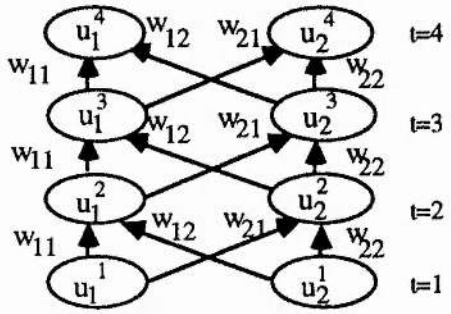



Fig. 2.4b The equivalent net of Fig.2.4a net.

The resulting unfolded network is strictly feedforward and can be trained by a slight modification of the conventional BP rule. This is that the input and output specified for unit $i$ at time $t$ is applied to unit $U_i^t$. In the forward pass phase, the activity values of the units at layer $t$ are evaluated based on the activity values at the layer $t-1$. Because the output units can adopt the appropriate states during the forward iteration, the errors are assessed not only at the final layer of the multiple layer feedforward network, but also at each time-step by comparing the remembered actual states of the output units with their desired states. The errors can then be injected backward from the layer in which they are produced. Therefore the errors, delta values and the activity values at appropriate time should be used in keeping track of the changes estimated for each weight at each level and then each of the weights are changed according to the sum of these individual prescribed changes. This implies that there is a weight change proposed for each link in existence at each time step. These weight changes are then added to give an overall weight change for each link at the last time step which is the actual weight change made.

The technique based on this model called **backpropagation through time**. Note that once a task has been trained in the unfolded version, the network may be used for the temporal association. Nowlan (1988) also obtained good results on a constraint satisfaction problem. This shows that the *backpropagation through time* algorithm is applicable to those tasks where enough information about the time structure of the tasks is known. This information is needed to restrict the layers to a reasonable number. However,

many tasks are not of this type. So if the requirement for memory corresponding to that for a large feedforward equivalent network is one of the limitations of the model, being aware of the time structure of the tasks is another one as is the fact that a delta value tend to zero as the number of layers increase. The *backpropagation through time* model though has not been widely applied, it has been largely superseded by the other approaches in recurrent networks.

### 2.3.4 FORWARD PROPAGATION

This is a learning algorithm for continually running recurrent networks, derived by Robinson & Fallside (1988) and later rediscovered independently by others (Williams & Zipser 1988; Gherrity 1989). The learning algorithm is very similar to the *discrete backpropagation through time* approach reviewed above, but instead of unfolding the recurrent network into a multiple layer feedforward network, an on-line technique with no additional hard copies of network topology but using non-local knowledge in computation is provided. The non-local in computation means that each unit must have knowledge of the complete weight matrix and error vector. Here a review of this model is presented—the dynamics of the network; the way to adapt the weights in improving the performance of the network over time; the advantage and limitations of this model.

• **The dynamics and weight adapting rules**

The dynamics of a *forward propagation* network is the same as that of *discrete back-propagation through time* described in Eq. (2.2). Let $y_k(t)$ denotes the output activity of unit $k$ at time $t$, and $x_k(t)$ denotes the external input signal to the unit $k$ at time $t$. Also define $z_k(t)$ to be the output of unit $k$ obtained :

$$z_k(t) = \begin{cases} x_k(t) & \text{if } k \in I \\ y_k(t) & \text{if } k \in U \end{cases} \qquad (2.3.1)$$

Here $I$ denotes the set units which have external inputs, and $U$ denote the set units which do not have external inputs in the network.

The dynamics can be described by the following two equations. Let $s_k(t)$ denote the net input to the $k$th unit at time $t$, for $k \in U$, where $U$ denotes the set of output units; $I$ denotes as the set of input units.

$$s_{k\,(t)} = \sum_{l} w_{kl} z_l(t) \tag{2.3.2}$$

where $l \in U \cup I$ and the external input at time $t$ does not influence the output of any unit until time $t+1$:

$$y_k(t+1) = f_k\,(\,s_k\,(t)) \tag{2.3.3}$$

Define a time-varying error signal for each unit $e_k$ by:

$$e_k(\tau) = \begin{cases} d_k(\tau) - y_k(\tau) & \text{if } k \in T \\ 0 & \text{otherwise} \end{cases} \tag{2.3.4}$$

where $T$ denotes the set of output units.

The network error at time $\tau$ is denoted as:

$$J(\tau) = \frac{1}{2} \sum_{k \in U} [e_k(\tau)]^2 \tag{2.3.5}$$

Now we want to minimise the total error over the trajectory when the network runs from time $t_0$ to time $t_1$:

$$J_{\text{total}}\,(t_0, t_1) = \sum_{\tau=t_0+1}^{t_1} J(\tau) \tag{2.3.6}$$

$$\text{Hence } J_{\text{total}}\,(\,t_0, t+1) = J_{\text{total}}\,(t_0, t) + J\,(t+1) \tag{2.3.7}$$

Suppose $\nabla_{\mathbf{w}} J$ denotes the gradient of a total error measure $J$ in weight space $\mathbf{w}$, according to the gradient rule: $\nabla(J_1 + J_2) = \nabla J_1 + \nabla J_2$ and Eq.(2.3.7), the gradient satisfies the relationship:

$$\nabla_{\mathbf{w}} J_{\text{total}}\,(t_0, t+1) = \nabla_{\mathbf{w}} J_{\text{total}}\,(t_0, t) + \nabla_{\mathbf{w}} J\,(t+1) \tag{2.3.8}$$

so we can simply accumulate the values of the vector $\nabla_{\mathbf{w}}J$ at each time step until the final time step. After the network has run through the $t_0$ to $t_1$ trajectory, each weight $w_{ij}$ can be altered by :

$$\Delta w_{ij} = \sum_{t=t_0+1}^{t_1} \Delta w_{ij}(t) \tag{2.3.9}$$

$$\text{According to : } \Delta w_{ij}(t) = -\alpha \frac{\partial J(t)}{\partial w_{ij}} \tag{2.3.10}$$

where $\alpha$ is some fixed positive learning rate.

We now need an algorithm to compute the $\frac{\partial J(t)}{\partial w_{ij}}$ at each time step $t$. As we know from Eqs. (2.3.4) and (2.3.5):

$$-\frac{\partial J(t)}{\partial w_{ij}} = \sum_{k \in U} e_k(t) \frac{\partial y_k(t)}{\partial w_{ij}} \tag{2.3.11}$$

Now let us see what is implied in Eq.(2.3.11). First of all it can be seen that, from Eq.(2.3.4), $e_k(t)$ is a term associated with target values. At each time $t$, $e_k(t)$ can be calculated for each $k \in U$. Next, the term $\partial y_k(t)/\partial w_{ij}$ measures essentially the quantity of the output value of the unit $k$ at time $t$ to a small change in the weight value of $w_{ij}$ along the entire trajectory from $t_0$ to $t$. It is assumed that the initial output state of the network $Y(t_0)$, the inputs over $[t_0, t)$, and the remaining weights are not altered when determining this term. Since this term is only taking into account the effect of such a change, it does not depend on any target values, it is only associated with internal actual values.

We understand from Eq. (2.3.11) that weight modification in *feedforward propagation* is associated with two kinds of terms. The two terms actually separate two features from each other. One feature is the term associated with target values and the other is the term only associated with internal actual values. Because both term could be calculated directly from the network's actual operation, this separation makes weight modifications without backward pass possible in this algorithm.

The following is about finding a way to compute the factor $\frac{\partial y_k(t)}{\partial w_{ij}}$ at time step $t$. To compute this term, we differentiate Eqs. (2.3.3) and (2.3.2), and at the same time consider two important assumptions: the input signals do not depend on the network weights and the initial state of the network is independent of the weights:

By introducing an auxiliary variable $p^k_{ij}(t) = \frac{\partial y_k(t)}{\partial w_{ij}}$ for all $k \in U$, $i \in U$ and $j \in U \cup I$ at time $t$, Williams and Zipser *et al.* have worked out that

$$p^k_{ij}(t+1) = f_k'(x_k(t)) \left[ \sum_{l \in U} w_{kl}\ p^l_{ij}(t) + \delta_{ik} z_j(t) \right] \tag{2.3.12}$$

where $\delta_{ik}$ denotes the Kronecker delta: $\delta_{ik} = \begin{cases} 1 & \text{if } i=k \\ 0 & \text{otherwise} \end{cases}$ ;

with initial conditions: $p^k_{ij}(t_0) = 0$ (2.3.13)

So the precise algorithm consists of computing at each time step $t_0$ to $t_1$ the quantities $p^k_{ij}(t)$ using Eqs. (2.3.12) and (2.3.13); getting the $e_k(t)$ using Eq. (2.3.5), and then computing the weight changes:

$$\Delta w_{ij}(t) = -\alpha \frac{\partial J(t)}{\partial w_{ij}} = \alpha \sum_{k \in U} e_k(t)\, p^k_{ij}(t) \tag{2.3.14}$$

The overall correction between $t_0$ and $t_1$ to be applied to each weight $w_{ij}$ in the network then is given by the Eq. (2.3.9).

Instead of using Eq. (2.3.9) as above to integrate the overall correction between $t_0$ and $t_1$ during the simulation and make a weight state transition, the weights can be continuously updated according to Eq. (2.3.14) under the assumption that each on-line weight change does not as an individual affect the trajectory taken very much when integrating Eq. (2.3.14) between $t_0$ and $t_1$. Since the auxiliary quantities $p^k_{ij}(t)$ have given initial boundary conditions (zero at the start of time), all the computations can then be carried out forward in time.

• **The summary**

This is an on-line and non-local algorithm. Networks can run continuously in the sense that they sample their inputs on every update cycle, and any unit can have a training target on any cycle. The computation time on each update cycle is also completely determined by the network topology, no relaxation or anything similar is required. Any weight value can be updated according to the Eq. (2.3.14). As the storage and computation time on each step are independent of length of training sequences and are completely determined by the size of the network, so no prior knowledge are required of the temporal structure of the task being learnt.

However this model is computationally very expensive. If there are $n$ units (non-input units) and $n^2$ weight-links in the network, as there are $n^3$ auxiliary variables, a total of $O(n^4)$ calculations need to be updated at each time step.

Efforts have been made to have a faster on-line technique and reduce the complexity of the algorithm (Zipser,1990). Recently work has been done on an exact, stable variant of the algorithm, which requires only $2n+n^2$ auxiliary variables so that they can be updated in just $O(n^3)$ time (Toomarian & Barhen, 1991). This model may become a feasible technique for on-line learning, or a model for further investigation of neural dynamic system features.

### 2.3.5 TEACHER FORCING NETWORKS

Williams and Zipser (1988) derived the **Teacher Forcing** algorithm which is an interesting variant of the *forward propagation* algorithm discussed above. This method can also be applied to other BP algorithms such as conventional backpropagation.

The algorithm is essentially the same as the earlier one, with one major alteration: using the teacher-forced state (desired output values) to compute future activity in the network; thus the teacher forces all the output units to have the correct states, even as the network runs.

According to the idea, the new definition of $z_k(t)$ is :

$$z_k(t)=\begin{cases} x_k(t) & \text{if } k \in I \\ d_k(t) & \text{if } k \in T(t) \\ y_k(t) & \text{if } k \in U\text{-}T(t) \end{cases}$$

rather than the one in Eq.(2.3.1); here the $T(t)$ is the set of indexes $k \in U$ for which $d_k(t)$ exists. The dynamics of the network is also modified based on this alteration. In implementation, network performs essentially the same computations as in §2.3.4 except that at each time $t+1$ the associated auxiliary variable $p^k_{ij}(t)$ is set to zero for all $k \in T(t)$ before computing any of the $p^k_{ij}(t+1)$. This is because now $d_k(t)$ is used as $z_k(t)$ so that $\partial y_k(t)/\partial w_{ij} = 0$ for all $k \in T(t)$.

The first version of this teacher forcing algorithm can be only applied to discrete time, clocked networks (Williams & Zipser, 1988). In a subsequent version, the model can be applied to temporally continuous networks (Pearlmutter, 1990). Although others using teacher forcing on networks with a large number of hidden units reported difficulties (Pearlmutter, 1988), this model can learn to approximate some oscillation functions.

### 2.3.6 DYNAMIC RECURRENT NETWORKS

Pearlmutter (1989) derived a solution using the dynamics of recurrent networks which with appropriate approximations[4] may serve as a basis of the learning algorithm for temporal association. The model is a continuous time version of *backpropagation through time*.

Similar to the *feedforward propagation* approach, this model is also based on a technique of computing $\partial E/\partial W_{ij}$ by doing gradient descent in the weight state $W$ so as to minimise the error $E$, where $E$ is an error function which measures the total error differences over the temporal trajectory of the states of a continuous recurrent network. But it is a local and

---

[4] Using $dy_i/dt \approx \{ y_i(t+\Delta t) - y_i(t) \} / \Delta t$ to approximate derivative $dy_i/dt$.

batch model. An error metric is introduced which measures how much a small change to $y_i$ at time $t$ affects $E$ when this change is propagated forward through time and influences the remainder of the trajectory. The metric has the form:

$$\tilde{z}_i(t) = \partial^+ E / \partial y_i(t) \tag{2.4.1}$$

where $\partial^+$ denotes an ordered derivative, with variables ordered here by time and not unit index.

The main idea of this technique is that a weight state transition is based on looking ahead a few time steps to see the errors associated with the training patterns based on the current weight state first and then decide the adjustment to the weight state according to $\partial E/\partial W_{ij}$. Pearlmutter has worked out the formulas for the calculations:

$$-\frac{\partial E}{\partial w_{ij}} = \int_{t_0}^{t_1} y_i f'(x_j)\, z_j \, dt \tag{2.4.2}$$

where $z_j$ is $\frac{\partial E}{\partial y_j}$ and is also the limit of $\tilde{z}_i$ as $\Delta t$->0.

$$y_i(t+\Delta t) = (1-\Delta t)\, y_i(t) + \Delta t\, f(x_i(t)) + \Delta t\, I_i(t) \tag{2.4.3}$$

$$z_i(t) = \Delta t\, e_i(t) + (1-\Delta t)\, z_i(t+\Delta t) + \sum_j w_{ij}\, f'(x_j(t))\, \Delta t\, z_j(t+\Delta t) \tag{2.4.4}$$

and $z_i(t_1) = 0$

After the network has learnt in the manner indicated above, a task-dependent internal representation has been set-up, so it is possible to approximate some time-dependent functions with the internal states' help.

This approach is also related in various ways to many other approaches in training fully recurrent networks. Pearlmutter's algorithm is a generalization of those algorithms derived for *fixed-point backpropagation through time* in discrete time recurrent networks.

It is reported that this type of model seems particularly suitable for temporally continuous domains such as signal processing tasks(Pearlmutter, 1990). Overall the Pearlmutter approach is the best for temporal association using an SBP approach.

### 2.3.7 MOVING TARGETS METHOD

Rohwer (1990) has proposed the *moving targets* learning algorithm which provides a training regime for discrete-time networks with arbitrary feedback links. This algorithm is also applicable for dealing with tasks associated with processing of inputs presented naturally in sequence.

The algorithm is similar to conventional BP in which an error function is minimised using a gradient-based method. However the optimization is different to the one used by conventional BP. The optimization in the moving targets algorithm is based on using an 'activation deficit' error function:

$$E_{ad} = \frac{1}{2} \sum_{(it) \in T} \{x_{it} - X_{it}\}^2 \qquad (2.5.1)$$

where a unit-time pair $it$ is called an 'event', $x_{it}$ is denoted as the excitation value or "activation" of an event, $X_{it}$ denoted as the target excitation for an event, $T$ is denoted as the set of *target* events.

The basic idea of the approach is to treat the hidden unit activation as variable target activation. The method seeks to achieve independent control over the changes in activation which are dependent on the weight changes in standard back-propagation. Suppose the dynamic variable $x_{it}$ changes with time according to the rule:

$$x_{it} = \sum_{j} w_{ij} f(x_{j,t-1}) \qquad (2.5.2)$$

and the output of each unit is :

$$y_{it} = f(x_{it}) \qquad (2.5.3)$$

The moving targets algorithm derives a modified form of Eq. (2.5.1) as a function of weights and moving targets:

$$E = \frac{1}{2} \sum_{(it) \in T \cup H} \left\{ \sum_{j} w_{ij} f(X_{j,t-1}) - X_{it} \right\}^2 \qquad (2.5.4)$$

where $f(x)$ is the logistic function, $w_{ij}$ represents the connection strength from unit $j$ to unit $i$. $H$ denotes the set of *hidden* events. The independent control over the activation changes mentioned above are instigated by substituting target excitation values for actual excitation values for both hidden and output units.

The substitution means that when the target excitations are chosen, the error is fully dependent on the $w_{ij}$ explicitly displayed in the equation. On the other hand, fixing the weights makes the error fully dependent on the target excitation. There are no a prior desired values for the $X_{it}$ with event $(it) \in H$ but any values for which weights can be found that make the error vanish would be suitable.

The equations for the minimum, $\partial E/\partial w_{ij} = 0$, form a linear system, these equations can then be used to define the weights as functions of the moving targets, the solution of which provides the optimal weights for any given set of moving targets.

For a set of suitable targets achieved using $\partial E/\partial X_{it}=0$, the network is able to control the units so as to be not too far from their targets which is achieved through $dE/dw_{ij} = 0$.

There are two phases in the training, in one phase the targets of hidden units are improved so that if the targets are attained better performance would be achieved; in the other phase the weight state is modified so that each unit comes close to attaining its activity target. The equations for modification of moving targets and weights have been derived by Rohwer, which are the derivatives with respect to the moving targets $\partial E/\partial X_{it}$ and the optimal weights for any given set of moving targets $\partial E/\partial w_{ij}$ (Rohwer, 1990).

This model, because it caters for recurrent networks, provides an alternative method for learning time representation. It has a control on the evolution of both activity and weight during the learning of weights.

The primary disadvantage of this model is: Each I/O tuple to be learnt must be associated with a set of targets for the hidden units. Because the targets need to be learnt as the weight state is, each tuple needs to be seen at least twice. This makes the technique inapplicable for on-line learning. Another drawback of using this method is: when more I/O training data is involved, more moving target values need to be trained as more hidden units are required. This implies that the number of the variables is increased as the number of the training steps which the network caters for is increased. This may heavily decrease the training speed.

## 2.4 CONCLUSIONS

In this chapter, major existing SBP neural models which can be used to solve the problem of learning time sequences have been reviewed. This review will be used in the next chapter to show why a new framework is needed in neural networks.

# CHAPTER 3

# A PATH-BASED FRAMEWORK

## 3.1 Introduction

In this chapter a new framework called path-based backpropagation (PBP) is presented. PBP is evolved from the state-based back-propagation (SBP) framework and deals especially with signal associations where temporal and sequential features are involved. The aim of PBP is to provide a means for achieving arbitrary approximations of I/O signal associations within a fixed neural topology with or without recurrent links using gradient descent.

In §3.2 the role of hidden units and its relation with training feasibility in feedforward networks using SBP are discussed. This gives an insight into the inherent infeasibilities both in training and generalization for arbitrarily close approximation of continuous functions in neural networks using the SBP framework. In §3.3, the common problems in time-dependent signal processing using existing SBP models are summarized. Features of these problems provide the basic requirements for a new framework. In §3.4 the philosophy of the new framework is given, the notion of goal weight sequences is introduced, and the PBP framework for time-dependent signal processing is proposed. Finally a conclusion is presented in §3.5.

## 3.2 I/O patterns, weight state, role of hidden units and training feasibility

This section is arranged to discuss the relationship between the number of I/O patterns and feasible training by asking the following two main questions: What are the necessary conditions under which a solution weight state exists for an I/O training set in single layer feedforward networks? What is the resultant essential requirement for the design of the network topology in multilayer non-linear networks? As the section name suggests, the I/O patterns, solution weight state and the hidden units will be shown to be interrelated one

another and are closely associated with the training feasibility which may become a problem when a large number of I/O training patterns are involved.

### 3.2.1 THE ROLE OF HIDDEN UNITS

The aim of training a neural network to realise I/O mappings in feedforward networks using SBP is to search for a goal weight state. Training will be fruitless if the goal weight state does not exist.

As Minsky and Papert (1969) pointed out, by showing that **XOR** cannot be learnt through any single layer perceptron, there is a fundamental inadequacy of single layer perceptrons in representing general I/O mappings. However, it is always possible to convert an unsolvable I/O mapping problem for a single layer perceptron into a solvable one using a multiple layer perceptron which gives rise to a need for hidden units.

This indicates that hidden units play a very important role to ensure that a solution weight state exists for a set of I/O patterns. The aim of training a neural network using SBP is to search for a goal weight state. Although hidden units may not always be needed for a goal weight state, they are still closely related with the existence of a goal weight state in many cases. That is to say that enough number of hidden units is one of crucial factors for realising training in SBP. The number affects training feasibility in two aspects. One is to ensure a network is trainable in the first place and the other is about the training speed when it is trainable. In this thesis, training infeasibility means that either the number of hidden units required for finding a solution weight state becomes so large, memory resources are exhausted; or the number becomes so large that training becomes infeasible in terms of time.

Training feasibility is particularly important in arbitrarily closely approximating time-dependent signal processing because a large number of analogue I/O patterns may underlie the processing which tends to infinity in the continuous limit. This last point gets more and

more of a grip as the error tolerance is decreased to accommodate approximation of more and more I/O patterns.

An analysis on the tendency of the training feasibility is needed for networks realising tasks such as time-dependent signal processing.

### 3.2.2 How various conditions can be used for justifying goal weight existence

To ensure a set of I/O patterns is trainable using a network, we need to investigate the existence of a goal weight state in the network. If the weight state does not exist, more hidden units are needed to have a new network with the existence of a goal weight state. Four conditions to justify the sufficiency of solution weight state existence, abbreviated by LPC, EE, LI, LS are discussed in this thesis. The subsequent analysis based on these four conditions gives a better understanding of the role of hidden units and also helps to estimate the number of hidden units in multi-layer networks.

- **Linear Predictability Constraint (LPC)**

McClelland and Rumelhart (1986) have given a necessary and sufficient condition for the existence of a goal weight state in a single layer linear network for arbitrary I/O mappings. This is called the **linear predictability constraint**: *"Over the entire set of patterns, the external input to each unit must be predictable from a linear combination of the activations of every other unit"* ( the external input here should be interpreted as the target output of the output units).

More specifically, a set of formal equations showing the constraint for the various I/O patterns to each output unit is given in Eq.(3.1a). Thus, for a set of I/O patterns there must exist a set of solution weight states $w_{kj}$ for every output unit $k$ in the network for all I/O training patterns $p$. That is

$$t_{kp} = \sum_j w_{kj}\ i_{jp} \qquad (3.1a)$$

where $t_{kp}$ is the target activity value for output unit $k$ for I/O pattern $p$; $i_{jp}$ is the input value for input unit $j$ for I/O pattern $p$ and is also the input activity value for output unit $k$ from unit $j$. Eq.(3.1a) can be represented in a form of matrix equation:

$$T = I\,W \tag{3.1b}$$

where each row of the activity matrix $T$ is the set of target activity values for an I/O mapping pattern. The number of the columns of $T$ is the same as the number of the output units in the network; each row of the input activity matrix $I$ is one input pattern. The number of the rows of matrix $I$ is equal to the number of the I/O mapping patterns; the number of the columns is equal to the number of the input units in the network.

Note that the LPC constraint can also be applied to non-linear networks if the excitation-output function of each unit is one to one. The Sigmoid function is an example of such a function.

Conclusion: LPC or a variant of it needs to be obeyed as a general condition if there is to be a solution weight state for an arbitrary I/O training set (including binary output mappings; analogue output mappings) for single layer linear and sigmoidal non-linear networks.

- **Excitation equations (EE)**

To represent further implications arising from LPC, here a set of *excitation equations* of output units is used as a way to reveal the relationship between I/O mapping patterns and weight link values in feedforward networks. This helps to show mathematically why hidden units are necessary for solving some mapping problems.

Suppose the network receives an input pattern $p$, we have

$$ex_{ip} = \Sigma_j\, w_{ij} \cdot a_{jp} \tag{3.2}$$

where $a_{jp}$ is the activity value of input unit $j$ for the pattern $p$; $ex_{ip}$ is the excitation value of the output unit $i$ for pattern $p$, and $w_{ij}$ is a weight link value linking from unit $j$ to unit $i$.

Eq.(3.2) is an excitation equation of an output unit $U_i$ for achieving I/O mapping between an output pattern $O_p$ and input pattern $I_p$ through a single weight state based on a single layer linear or non-linear feedforward network. To simplify the analyses in this section, the bias unit will not be considered here.

There are three points implied in the excitation equation sets to be noted, which are

1) The goal weight state has to be the solution of a number of simultaneous equation sets.

If a network is able to achieve a number of exact value I/O mappings in the network using a single solution weight state, a set of excitation equations in the form of Eq.(3.2) must be satisfied simultaneously for each output unit in the network with regard to all the I/O mapping patterns.

2) The solution weight existence can be examined by the rank of matrices.

LPC is obeyed if and only if each of $ex_{ip}$ in Eq.(3.2) is one to one associated with the $t_{ip}$. This is regardless of how $t_{ip}$ is produced by various networks. An important case for this thesis will be where $f(ex_{ip}) = t_{ip}$ for a single but non-linear layer network (where $f$ (x) is an one-one non-linear function such as the logistic or sigmoid function). For each output unit the Eq.(3.2) can be represented in a matrix form which is

$$Ax=b \tag{3.3a}$$

where $A$ is the coefficient matrix with $m*n$ input pattern elements ($m$ rows – number of patterns and $n$ columns – number of dimensions of each input pattern); $x$ is the variable vector with $n$ weight elements; $b$ is constant vector with $m$ target output elements. An augmented matrix of the equation set is matrix $C$, which is given by:

$$C=\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} & b_1 \\ a_{21} & a_{22} & \dots & a_{2n} & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} & b_m \end{bmatrix} \tag{3.3b}$$

A solution $x$ exists if and only if the rank of matrix $A$ and the rank of its augment matrix $C$ has a relation that

$$r(A)=r(C) \leq n \tag{3.3c}$$

This implies that the number of the independent equations is dependent on both values in matrix $A$ and the vector $b$. Only if the number of the independent equations is equal to or less than the number of the variables $x$, will one or more solution values for $x$ exist.

3) There are important cases that can be discussed using EE — binary output mappings and analogue output mappings

— Analogue output mappings: these output mappings have potentially zero error tolerance and are called exact value mappings in the thesis. In this kind of mapping, the target excitation values in Eq.(3.2) is a set of fixed exact values rather than a range of values. Finding a solution weight state means solving the equation set exactly.

— Binary output mappings: this kind of mapping has a broad error tolerance and is called an inexact mapping in this thesis. Each target excitation value can be set to be any value within a range (or within the error tolerance), the excitation equation set is associated with a set of inequalities for each output unit. Finding a solution weight state means solving the set of inequalities.

**• Linear Independence condition (LI)**

LI is an important corollary of LPC for input sets with arbitrary associations in single layer networks. The LI condition implies that an arbitrary output pattern $O_p$ can be correctly associated with a particular input pattern $I_p$ without ruining associations between other I/O pairs only if $I_p$ is not a linear combination of the other input patterns. This means that in order to ensure arbitrary associations to each set of input patterns is learnable, the input patterns must form a linearly independent set (McClelland & Rumelhart, 1988).

Single-layer networks

For single layer networks, LI can be required directly for arbitrary associations for exact value mappings in general for a given number of input and output units; and indirectly after reduction in individual mapping cases.

Reduction means that some I/O training patterns may be eliminated from the associated excitation equation set to give a reduced equation set. More specifically, for a particular set of I/O patterns, there are the associated matrices *A* and *C* defined in Eq.(3.3). Reduction needs to be carried out if an excitation equation associated with an I/O pattern can be eliminated from the equation set without affecting the rank of r(A) and r(C). As part of the rank condition, Eq.(3.3c), if the matrix A now has linearly independent rows there is a solution weight state.

The direct application of LI means that LI is applied to the input training set without reduction because the worst case of no reduction has to be included amongst the arbitrary associations. LI should be applied to the input patterns associated with the reduced set in individual mapping cases because a particular case may be reducible.

Multi-layer networks

Based on the discussion of Eq.(3.3), for a multi-layer network, LI is directly relevant to hidden units. This is because a multi-layer network can realise an I/O training set if and only if a linearly independent reduced set of input values can be found for each output unit, where the input values are derived from network processing.

A connected consequence to the discussion is that a training set of input patterns for an output unit can be made linearly independent if and only if the number of linearly independent columns of input values for each input appearing in the excitation equations is at least as great as the reduced number of the excitation equations.

Now that the relevance of linear independence to multi-layer networks has been established, we may reach a conclusion about the role of hidden units for finding a single goal weight state for multi-layer networks. Firstly, if the number of LI columns of input values in the EE for an output unit is less than the number of excitation equations in the minimum reduced set achievable over all weight states, then the associated reduced set of input patterns for the output unit will always be linearly dependent. This fact leads to the conclusion that there is then no solution weight state. For suppose this fact is the case, i.e. that the rank of matrix $A$, r($A$), is $m$ and that $m<n$, where $n$ is the number of linear independent equations. For the matrix $C$ defined in Eq.(3.3b), r($C$) = $n$ since there are $n$ linearly independent equations and hence r($A$)≠r($C$). According to the rank condition in Eq.(3.3), there is therefore no solution weight state.

The introduction of further non-linear hidden units feeding in directly or indirectly to the output unit may provide a solution weight state in two ways. One is to achieve a sufficient degree of linear independence by adding extra linearly independent columns to matrix $A$. The other is to achieve a sufficient degree of reduction by increasing the linear dependence amongst the rows of matrix $C$.

**• Linear Separability condition (LS)**

LS is linear separability condition for justification of weight state existence for binary output mappings in single or multi-layer networks. LS makes conclusions which are consistent with that of LI.

LS applies directly in single layer networks when considering networks as binary output mappers. A necessary and sufficient condition for the existence of a solution weight state is given by:

$$\sum_{j=1}^{n} w_{ij} \cdot i_j = \theta_i \qquad (3.4)$$

where $\theta_i$ is the threshold value of each output unit *i*. Eq.(3.4) defines a hyperplane so that all of those inputs of associations with the output value 0 are on one side of the hyperplane and those with output of 1 are on the other side.

Functions with such a hyperplane are called linearly separable. These functions define the class of problems that can be solved by a single layer perceptron (Rumelhart *et al.*, 1986).

In multi-layer networks, LS gives a geometric view of the role of hidden units in realising binary output mappings: When no hyperplane associated with an output unit exists in the input space formed by input patterns which consist of the input values fed into the output unit, hidden units are needed to give another recoded input space to ensure the existence of a solution hyperplane.

- **Example**

XOR is a binary output mapping problem. Both the LI and LS as conditions can be applied to see whether a solution weight state exists in a single layer network. This example shows that LI and LS will make the same conclusion for justifying the existence of a goal weight state for a set of I/O patterns.

There is no single hyperplane in the input space which can separate the two classes of input patterns. This implies that the set of I/O does not satisfy the LS given by Eq.(3.4). There is no solution weight state for realising the I/O set using a single layer network.

The non-existence of a solution in a single layer network can be also checked through the LI after reduction. The following matrix equation: $\mathbf{I}^*\mathbf{W} = \mathbf{X}$ needs to be satisfied simultaneously to achieve XOR using a single weight state W:

$$\begin{bmatrix} I_1 \\ I_2 \\ I_3 \\ I_4 \end{bmatrix} \begin{bmatrix} w_{31} \\ w_{32} \end{bmatrix} = \begin{bmatrix} t_{31} \\ t_{32} \\ t_{33} \\ t_{34} \end{bmatrix} \qquad \text{where} \begin{bmatrix} I_1 \\ I_2 \\ I_3 \\ I_4 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$

The equation associated with the input (0,0) and $t_{31}$=0 can be eliminated without affecting the rank of r(A) or r(C). There are three further irreducible equations, given the ranges of excitation for $t_{32}$, $t_{33}$ and $t_{34}$. LI should be applied to the input patterns associated with the reduced set. Because there is linear dependence in the input set such as $I_4 = I_2+I_3$, but $t_{34} \neq t_{32}+t_{33}$, this mapping cannot be realised using a single layer network. The conclusion of applying LS or LI is therefore the same.

- **Summary of §3.2.2**

When designing a network topology for a particular I/O training set, a sufficient number of hidden units is needed to provide a rich enough weight state description for accommodating the I/O structure of a problem.

### 3.2.3 WHETHER TRAINING FEASIBILITY IS A PROBLEM

All three conditions discussed in §3.2.2, which are LPC, LS, LI, have been used to review the existence of a solution weight state. It remains to find what the relationship is between the number of I/O patterns and feasible training in general.

It has already been proved in theory that as long as there are enough hidden units, a multi-layer feedforward network is able to realise any I/O mappings, not only for the binary output mappings using multi-layer perceptrons (Minsky & Papert, 1969) but also for continuous mappings using non-linear perceptron-like networks (Cybenko, 1989; Funahashi,K., 1989; Hornik, K., Stinchcombe, M., and White, H., 1989). Funahashi (1989) has also proved that any arbitrary discrete output mapping can be done by using a three layer feedforward network. However those theories do not address an important point — that is, how many hidden units are really needed for a solution weight state existence.

Actually the number of hidden units is crucial in justifying SBP training feasibility in terms of training speed as well. To see how the number of hidden units is closely related to these two aspects of training feasibility, we examine two particular cases. This analysis will have important implications for the thesis where a large number of I/O training patterns are involved in arbitrarily close approximation of I/O mappings.

1) Network design intended for universal binary output mappings

As discussed in §3.2.2, for a particular binary output training set on a chosen topology, the hidden units are added to accommodate the existence of a goal weight state for a minimum achievable reduced EE or to achieve the LS condition.

From the geometrical view of LS it can be said that there are three interrelated factors in each binary output mapping. Those are the number of hidden units, the decision regions and I/O training set. The I/O training set determines the complexity of the decision regions in terms of LS; By complexity here is meant an informal idea of number and linear shape of distinct binary output regions. It is related to the degree of recoding needed for each output unit to achieve LS of the input fed into it. The more complex the regions, the more hidden units are needed for the I/O set. There have been more formal definitions of complexity using simpler networks (e.g.: Lippmann, 1987; Baum, 1988).

Lippmann (1987) has pointed out geometrically the relationship among the I/O training set, number of hidden units and decision regions using linear threshold units for binary output mappings, i.e. a perceptron. As explained in Khanna's book (1990), for each output unit, a single layer perceptron forms a half-plane decision region in the input space; a two layered perceptron (with one hidden units layer) forms any convex region (including unbound ones) in the space; a three layered perceptron forms any set of decision regions in the space.

Baum (1988) has established various bounds for estimating the number of hidden units in universal binary output mappings using multi-layer perceptrons. One result is that a

network with one hidden layer requires a minimum number of *[N/d]* hidden units to implement an arbitrary dichotomy ( 2-valued output) for $N$ I/O values and $d$ input units. Another more general result is that a multi-layer perceptron requires at least $O(\sqrt{N\,e\,/\,\log_2 N})$ units where $N$ is as before and $e$ is the number of output units. It can be seen from the last expression that the more I/O training patterns there are, the more hidden units are needed. Note that these results are also based on networks with linear threshold units, they cannot be straightforwardly applied to sigmoidal networks. However, while the more detailed conclusions of Lippmann and Baum cannot be used for sigmoidal networks, the general theoretical trends for hidden units in terms of complexity are acknowledged here.

The conclusion is: for arbitrary binary output mappings using sigmoidal networks, if more I/O patterns form more complicated decision regions, the more hidden units are needed, the less feasible the training is likely to be.

2) Network design intended for particular continuous mappings

Similar to the discussions for binary output mappings, for a particular analogue output mapping training set on a chosen topology, we need to make sure that the number of hidden units is enough for accommodating the existence of a goal weight state for realising a minimum achievable reduced EE or LI.

As concluded above, the number of hidden units is related to the number of the linearly independent EE in the reduced sets. Consider a given topology which is successful for a given training set of I/O patterns. When the number of I/O training patterns increases, the number of EE that are linearly independent will tend to increase as well and thus eventually prevent there being a solution weight state. In turn, the number of hidden units needed has to be increased to ensure the increased amount of linear independence needed among the input patterns for each output unit that will ensure the existence of a solution weight state.

In principle, this type of mapping requires more hidden units compared with that of binary output mappings for the same number of I/O training patterns. This is because, in EE, the target values associated with exact value mappings are fixed exact values. This is unlike binary output mappings where each target can be chosen flexibly within a certain range.

In order to keep the minimum number of hidden units unchanged, a match is required between the excitation equation of a new I/O pattern and linear combinations of those equations associated with other I/O patterns. This match is increasingly unlikely for an analogue output as the error tolerance is shrunk for a closer approximation. Hence the number of the independent equations will not be further reduced with less flexible target values than in binary output mappings. An example of the OR problem is shown with this feature in Apdx 2.

The more arbitrary I/O patterns to be trained, the more hidden units may be needed. Hence the number of hidden units may become infeasible for large number of such I/O patterns and infinite in the continuous limit.

From the geometrical explanation of LI it can also be seen that the number of hidden units can only remain unchanged if the hyperplane in weight space associated with a new input pattern has a common intersection with those hyperplanes associated with other I/O patterns in weight space.

The conclusion is: the more arbitrary related I/O patterns there are in the training set and more equations in the reduced EE, the more hidden units are likely to be needed, and hence the less feasible the training is likely to be. This is consistent with Baum's theory for threshold networks and the theoretical analysis mentioned above for continuous mappings where Funahashi (1989) assumed that a continuum of hidden units are needed in the mapping, while Cybenko (1989) and Hornik *et al.* (1989) assumed that a large enough number of hidden units are used.

### 3.2.4 CONCLUDING REMARKS

For binary output mappings, the more complex the decision regions are, the more hidden units are needed for finding a solution weight state.

For analogue output mappings, such as those underlying time-dependent signal processing, the more I/O patterns that are chosen for training along the paths, for a closer approximation, the closer the mapping is to an exact value mapping. The more arbitrary I/O training patterns there are to be trained exactly, the more hidden units may be needed to ensure linear independence for a larger reduced set of input patterns to output units.

This shows a theoretical trend of the minimum number of hidden units against the number of independent I/O training patterns: the memory resources needed for hidden units and the speed of training may become infeasible for large I/O mappings. This has implications for complex binary output mappings and arbitrarily close approximation of analogue signal processing.

## 3.3 Required features for time-dependent signal processing (TDSP)

There are many problems which need to be solved in neural networks in order to mimic the nature of time-dependent signal processing in the real world. In this section, features of time-dependent signal processing which are related to further explanation of the infeasibilities of the existing models are reviewed. The features discussed below mainly explain why a new type of neural framework is needed. Only those features which are both important in the investigation of the analogue models and at the same time are related to the work which has been tackled in the thesis are presented.

(1) Time representation

A general network design for representing temporal structures should be independent of particular tasks as much as possible. It will be shown that this implies that there should be as little influence as possible on the training feasibility when the desired approximation accuracy to a time-based signal is increased.

(2) Continuous function

An I/O path can often be described by a continuous function in a coordinate system which consists of infinite number of states, where each state consists of analogue signal values. This means that analogue neural network systems should be able to approximate the I/O associations chosen along continuous functions to any variable finite accuracy within a feasible time.

(3) Multiple I/O path associations

I/O associations based on more than one I/O path may need to be responded to by a single system. For example, where an I/O path mapping is comprised of 9 I/O paths, one system should be designed to achieve these associations rather than 9 separate systems.

During system performance, I/O associations may be based on different sequential features of various paths. This implies that the system needs to learn not only all the instantaneous I/O associations within a feasible time but also the sequential features embodied in the I/O associations such as which of the input paths the network is performing along at a certain moment.

### 3.4 Common problems using SBP

All the SBP models discussed in §2.3 embody some methods for representing the effect of time in a neural network and have certain capacities in learning time sequences with the help

of gradient descent techniques. There are some problems in those SBP models in dealing with sequential processing in general and TDSP in particular. In this section, some of the limitations of SBP approaches for implementing some common features of time-dependent signal processing in neural analogue systems are discussed. An outline of those limitations which indicate the exploration of a new framework would be fruitful are presented.

(1) The design of a net topology is imprecise generally

In SBP models, the network structures are very crucial for the existence of a solution weight state. For example, in the *Simple Recurrent Network* (SRN) (refer to §2.3), when a SRN is used to approximate a finite state automaton or to do simple sequential recognition, there may be failure in sequential aspects if the number of hidden units is not enough. For example, the network may not be able to distinguish two very similar but different sequential paths of characters. Suppose identical output follows from a particular finite length of input path, where this memory length is determined by the number of hidden units. If the paths are different initially but identical over the current memory length being used for output determination, then identical output will be given where different ones should occur. For all types of networks, the smaller the problem solved at any time, the easier it is to arrive at a solution topology. More precision would be available if the sequential aspects did not rely on the design of the number of hidden units needed.

(2) A training feasibility problem

For all existing SBP models, the network is trained to find a single final weight state. When a network is used to approximate a continuous function, it means that a single goal weight state needs to be found for the network through a finite number of I/O analogue training patterns chosen from the function.

One important aspect in achieving this target is about training feasibility. As discussed in §3.2, an adequate number of hidden units are needed to provide internal representations in

order to get a single goal weight state for certain number of I/O training patterns in SBP models.

TDSP in neural networks involves the recognition of an underlying continuous function after training enough number of I/O training patterns along the function. As analysed in §3.2, the more I/O training patterns an I/O training set has, the more internal representations may be required and the more hidden units are needed. The number of the hidden units will be increased together with the number of the training patterns. The more hidden units applied, the more complex the error-weight surface a network has and hence the less feasible is the training. So training speed is very likely to be decreased as the number of hidden units increases. This is borne out empirically (e.g. Sutton, 1986; Hinton, 1987; Fahlman, 1988b). The potentially infinite number of I/O patterns in analogue signal processing therefore may pose a training feasibility problem in TDSP.

(3) Generalization capacity is limited

In the existing neural network models, training is supposed to be able to discover the underlying relationships amongst the I/O training patterns. For example, there may be a family tree relationship underlying a certain number of satisfactorily chosen training patterns which represent somebody's ancestry. The relationship may also be an abstract mathematical function underlying a set of training patterns.

Generalization in neural networks is to have an ability to do the correct associations for untrained input patterns which are based on the same relationship as that of the training patterns. If the underlying relationship has been learnt through the samples of the relationship during training, the network should therefore have capability for generalization. If the underlying relationship is partially or completely not captured during training, the associations to the untrained patterns will be either very poor or meaningless. When there is no generalization capacity, the trained network is just a look-up table for the trained patterns (Hinton, 1987). This is not what we expect from neural networks. For a formal discussion of generalisation in learning, see Valiant (1984).

However as Chauvin (1990) pointed out: "generalization and interpolation properties of non-linear networks are still theoretically obscure". In SBP, it is not theoretically clear both as to how the underlying relationship is captured during training and how to control the accuracy in the generalization. Various practical attempts measuring the generalization obtained for particular problems report some success which is nevertheless limited (e.g. Yu, 1990).

Obtaining outputs for untrained inputs in SBP is not a problem, it is the fact that the outputs may not correspond to those given by the trained network that is the problem. It may be readily appreciated that an I/O path may be approximated by setting a finite number of I/O pairs distributed over the path as data points and interpolating the rest of the path. In neural terms, this corresponds to approximating the I/O path by setting a finite number of weight states to reproduce the I/O data points through training and interpolating the weight states to produce the rest of the I/O path through generalization. For SBP the number of weight states set is 1. Therefore the SBP approach has limitations in generalization in at least two aspects: (1) the accuracy of generalisation is limited by the inherent weight interpolation scheme, only a constant interpolation scheme through a single solution weight state during generalisation; (2) the interpolation is also limited for a given network architecture because a given network can only cater for a finite number of I/O data points— these two aspects bring a problem termed as the Untrained Output (UO) problem in this thesis.

An I/O mapping may be thought of geometrically as a contour map where the contour height corresponds to output value and the horizontal plane corresponds to the input values. A geometrical interpretation of the UO problem for I/O mappings is that a contour map of the generalisation I/O set in input space may be different to the map associated with the training I/O set. The contour map analogy applies to both analogue and binary I/O mappings.

The resolution of the mismatch is to increase the number of I/O data points and/or to increase the order of interpolation.

In SBP, there is no variability in the order of interpolation that can be used to tackle the UO problem through weight state. The mismatch can only be tackled by increasing the number of I/O data points. According to the discussion in §3.2 this implies adding hidden units in the SBP approach.

So in SBP the Untrained Output problem may occur either due to the inherent limited interpolation capability or due to an inadequacy in the size of the given network. Hence the generalization capability is limited.

(4) Modelling time

In SBP, a single weight state is used to serve as a spatial parameter for all I/O associations which are presented over a period of time. This means that SBP treats time over a period for training purposes as if it were a spatial whole instead of modelling the effect of time sequentially. This makes time representation in networks often over complicated in dealing with some relatively simple sequential processing.

Approximating tasks which involve temporal associations may not always implies that a neural network should approximate a dynamic system. It may only be required that a network approximates a I/O function.

If a path such as one of those found in sequential I/O functions contains one-many I/O associations over a period of time then a dynamic system approach is forced for the SBP approach. This is because a single weight state in a feedforward network only allows many-one associations over any period of time. Dynamic systems approaches are more complicated than I/O function approaches and so the forced change in category of approach may be over complicated.

Therefore a framework with a different approach to modelling the effect of time may simplify solving some tasks involving time structure.

(5) Finite networks and infinite I/O associations

As discussed in §3.2, in arbitrarily accurate approximation of a continuous function, SBP uses finite units to achieve representation of an underlying infinite structure.

With SBP, the entire I/O approximation is attempted using a single weight state of finite dimension. The effect is analogous to attempting multiplication with a Finite State Machine (FSM).

For an FSM, as the multiplicands grow in the number of digits, the internal states grow in number correspondingly to act as sequential memories for past operations.

For SBP, as the accuracy of an approximation to a continuous function is increased in the number of chosen training patterns, the weight state has to grow in dimension to produce the increasingly complex I/O association.

A given finite mechanism is unable to fully anticipate the great complexity of its environment no matter how well it is designed. As far as the dynamic computation is concerned, both FSM and the analogue to FSM in neural networks have to provide infinite temporal extension through what is naturally a finite space.

One approach which provides an infinite extension feature for neural networks has been made, which is to dynamically add hidden units during training as the extra need arises (Ash, T., 1989). The ability to find a solution topology for a training set is thereby enhanced. This spatial augmentation is analogous to adding new internal states to a FSM.

As has already been discussed in (3) above, generalization requires not only a suitable number of data points but also a suitable order of interpolation. Hence Ash's approach has a limited potential for feasibly improving generalization.

(6) Sequential processing and recurrent networks

When recurrent networks are used to provide one-many I/O associations, the extra internal activity provides an internal state derived from past activity. What are the limitations using the activity based approach?

One limitation is that the training surfaces are still superposed. Training is as slow (if not slower) with recurrent networks as with feedforward networks (Sutton, 1986; Hinton, 1987; Fahlman, 1988).

A second limitation is that generalization is improved but still limited. Recurrent activity, as internal network activity, may be considered for substitution for providing generalisation through variable interpolation. Linear or higher order interpolation through varying recurrent activity will be more accurate than constant or zero order interpolation with a fixed or zero recurrent activity. However, no matter what order of interpolation is used, the degree of approximation is inherent limited in two ways: One is that the weight state is fixed. Hence the approximation is still a point approximation to a path in weight terms. The second way is that further training data chosen cannot be added without a possible increase in the number of hidden units being required.

The above is intended to show some inherent weaknesses in the single weight state framework. A path-based framework investigated for a resolution of the weaknesses is introduced below.

### 3.5 A path-based framework

In order to deal with the above weaknesses directly, it would seem possible to give equal status to the trained weight state as to the internal activity state, that is to allow weights to vary over time. Simpson's definition of neural convergence allows this as a possibility: " *if the mapping converges to a fixed value, or to some fixed set, then the learning procedure is*

*properly capturing the mapping.* " (Simpson, 1989). This equality of status leads to a new framework for backpropagation. So in order to resolve the problems mentioned above, a new framework based on dynamic extension in time is proposed. It will be seen later on that such extension in time will lead to extension in space as well.

### 3.5.1 AN ABSTRACT MACHINE ANALOGY

For present purposes, the aim is to not only increase the number of data points that can be captured but also to interpolate the data points to various degrees.

An abstract machine analogy is that a kind of Interpolating Turing Machine (ITM) may be created. The extra feature relative to a Turing Machine (TM) is that there is more than one symbol associated with each tape square. Suppose each tape square has a data point number at the centre of the square. Other numbers are then able to be associated with other points along the centre of the tape through interpolation (Fig. 3.1). Hence there is infinite interpolation as well as infinite discrete extension.

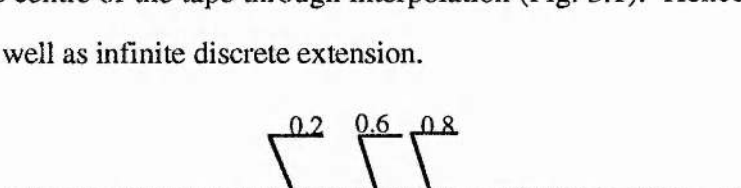



Fig. 3.1 An Interpolated Turing Machine tape

A neural dynamic approach which has the mechanism of using dynamic interpolated memories as the analogue to an infinite interpolated TM tape and embodies time sequences in the system is investigated in this thesis.

Important points to this new approach to note are: (1) In SBP, finite representation concentrated into a single moment of random access time is forced to correspond to an infinite mapping over a period of sequential time. (2) A dynamic interpolated memory approach provides, by contrast, one-one correspondence with an infinite sequential mapping. Such a correspondence provides an analogue of the Interpolated Turing Machine tape which evolves the FSM to cope with problems of infinite extension in time such as

those mentioned earlier. (3) The more natural correspondence with time will be shown to produce significant computational benefits with respect to the problems outlined in the previous section.

The specific goal of the new approach: (1) to have a conceptual exploration to find a resolution of the problems that occur in training potentially infinite analogue I/O associations where there are underlying continuous functions; (2) to explore the neural realisation of the relationships empirically.

### 3.5.2 THE ROLE OF GOAL WEIGHT STATES IN NEURAL CONVERGENCE

In general, the aim of training a neural network to realise I/O associations is to search for a goal as a condition for neural convergence. A machine is thereby obtained which provides the desired I/O mappings in performance. This aim does not imply that there can necessarily only be a single weight state as the goal of training. If the goal of the training is to find more than one weight state or a single weight path, and during performance these goal weight states at each moment can be associated with a particular I/O mapping accurately, this also satisfies the above aim.

The goal of neural convergence in SBP framework is a single goal weight state. The benefits of the approach has been briefly reviewed in §1.3. The infeasibility of this approach is also discussed in §3.2 and §3.4. A question being asked here is whether there is any other goal of neural convergence which can be applied for the same aim but without the training feasibility problem.

The new approach begins by appreciating that sequential access is suited to many types of TDSP. This is because for each next step in time, a correct response is only required to the next input value along one path and not to any other predecessors or successors.

The framework explored in the thesis is to define the goal condition to be a sequence of weight states, a weight path rather than a single weight state. This trades off access to

desired output in performance against the number of hidden units needed for training in such cases.

The conclusion of this section is: a weight path approach as an alternative goal of neural convergence is proposed for approximating the production of I/O associations.

### 3.5.3 MAJOR SPECIFIC FEATURES OF THE PATH-BASED APPROACH

In the path-based backpropagation framework PBP, instead of a single weight state as the simultaneous solution for all the I/O training patterns as the goal of training, a sequence of weight states, a weight path, is found and used. In general a weight path instead of a weight state is used as the goal of training.

Each weight state in the goal path provides random access to just those individual I/O patterns occurring at the same evolved fractional distance in time along each of a number of training I/O paths. The fractional distance will be said to constitute a *position* in the I/O paths' state sequences. Travel along the weight path in time during performance allows sequential access to all the desired values at each *position* along the I/O paths.

More technically, suppose we wish to train $m$ sequential I/O paths with $n$ positions. Let the $i^{th}$ sequence be $S_1^i, S_2^i, ..., S_n^i$, where $S_n^i$ is denoted as the pair of I/O at position $n$ along the $i$ path. Then it is desired to find a sequence of goal weight states $W_1, W_2, ..., W_n$ where $W_j$ realises the set $\{ S_j^1, S_j^2, ... S_j^i, ... S_j^m \}$ at sequential position $j$. Fig. (3.2) is a diagram of showing the relationship between I/O and W for a 2-orbits problem. The one orbit starting at the North has a binary target output of 0. The other starting at the West position has a binary target output of 1. The two sets of end points of the straight lines are shown in Fig.(3.2), each line constitutes a training position and is associated with a goal weight state.

In practice, this framework trades the storage needed for storing the goal weight path and restricted access to desired outputs in performance against the variable number of hidden units needed for training.

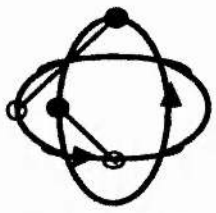


Fig. 3.2 The two training positions along 2-orbits

The following points should be born in mind and pursued throughout the design of the new framework: (1) Representation capability will be independent of the number of data points used sequentially as much as possible; for example the representation capability should not change whatever the length of a chosen curve represented in coordinates will be; (2) The network topology will be finite and relatively smaller than for SBP; (3) Accuracy in generalization is controllable, the Untrained Output problem can be resolved to a significant extent; (4) A single network can be trained to do I/O associations for multiple I/O paths. During performance, I/O associations should be able to switch from one path to another.

The computational benefits of this approach lie in both the feasibility of training and the provision of a trained weight path for use in generalization. These features will be further analysed and discussed together with two models based on PBP.

The two PBP models are specially implemented for investigation of approximations of sequences of I/O associations chosen from continuous functions or complex analogue-binary mappings The first one is for feedforward networks, and is called Feedforward Continuous Back-Propagation (FCBP). FCBP is a first step within PBP. It solves at least some problems associated with time-dependent signal processing. However, the dynamic capacity is still limited in FCBP. In order to further explore the dynamic systems of path-based sort, another model called Recurrent Continuous Back-Propagation (RCBP) based on both the weight path and a sort of internal state path approach on recurrent networks has

also been preliminarily investigated. For more details about the PBP and the models FCBP and RCBP, please refer to the chapter 4 and chapter 5 respectively.

## 3.6 Conclusions

The basic question raised in this chapter is whether or not SBP-based neural network models are suitable for analogue sequential processing in general or time-dependent signal processing in particular.

It has been pointed out that infeasibility is aggravated in training arbitrarily close approximation of continuous functions and the associated Untrained Output problem for generalization in SBP may be severe. Most importantly, analogue infinite I/O or complex binary associations are not always easily represented by finite neural topologies with SBP models. A dynamic interpolated memory approach is proposed instead resulting in the use of goal weight paths.

# CHAPTER 4

# A FEEDFORWARD CONTINUOUS BACKPROPAGATION MODEL

## 4.1 Introduction

In this chapter a new approach evolved from conventional back-propagation called feedforward continuous back-propagation (FCBP) is presented.

FCBP is one of the two approaches which have been developed in the thesis using a path-based backpropagation framework. The aim of the FCBP approach is to provide a means for achieving arbitrarily close approximations of I/O mappings along paths within a fixed neural topology.

In §4.2, a general introduction of FCBP, the roles of hidden units and weight states in FCBP are reviewed. In §4.3 the notion of goal weight sequences introduced in PBP is applied in FCBP to see if the additional sequential properties bring significant benefits in training and generalization using feedforward networks. Then in §4.4 and §4.5 respectively, the training and generalization schemes of FCBP are given. Finally a conclusion is given in §4.6.

## 4.2 FCBP model

This is the first model developed based on the PBP framework for feedforward networks. The philosophy of the path-based approach framework PBP has been introduced in §3.4, here the FCBP itself is studied.

As reviewed in §3.3, the training required in time-dependent signal processing is to have a network produce desired output paths in reaction to independent switches in the input paths during performance. Sequential access can be seen as a natural way of modelling the effect of time in this case.

In FCBP the sequential access to I/O mappings is modelled by extending the use of goal weight states over time. This is to apply the PBP framework in a feedforward network, which implies that the goal weight condition is a sequence of weight state transitions, a weight *path* rather than a single weight state.

The aim of training in FCBP is to search for a goal weight path which can yield a sequence of internal states within a network and provide I/O sequential mappings in performance. Each goal weight state in the weight path only needs to enable the neural network to act as an abstract machine at a certain moment which produces the correct outputs for the associated inputs at that *position*. The abstract machine represented by the neural network then changes as the weight state changes. It can be seen that training in FCBP is to find each goal weight state for each training position using conventional BP, and then store each the goal weight state indexed by position.

The aim of generalization in FCBP is to approximate the underlying continuous goal weight path through the discrete sequence of goal weight states associated with the training positions.

Following is a more detailed discussion on the role of hidden units in FCBP and the relationship amongst I/O and weights in training and performance.

### 4.2.1 THE ROLE OF HIDDEN UNITS IN FCBP

Now that each weight state is associated only with I/O values at its associated position in the paths instead of all I/O patterns at all positions in SBP, hidden units are no longer needed in single I/O path cases. However, hidden units are still needed in FCBP if the target I/O functions to be realised contain more than one I/O path.

At each position, training is performed in the same way as that of SBP. Hidden units are needed to solve the simultaneous equations (Eq. (3.2) ) corresponding to the I/O values which are from different paths but at the same position. In other words, the hidden units

provide a single weight state for all the I/O values along the set of I/O paths that occur at the same position, so that each weight state can be adapted to all the target I/O values associated with each position. In this way, a single goal weight path may be used for the I/O associations of many different I/O paths. The number of the goal weight states, which consist of the goal weight path, is equal to the number of training positions.

In FCBP as the number of hidden units has to cater for only one position at a time, this number is completely determined by the number of training sequence paths rather than the number of training I/O patterns along the paths. The number of the paths are known before training and is relatively much smaller than the number of the training patterns in general, this is why that much less hidden units are needed for training in FCBP compared with that in SBP.

It is also noticed because the number of I/O values at each such training position remains the same and the number of hidden units required in FCBP remains fixed no matter how many positions are chosen to train on to approximate the I/O paths, the size of networks required does not need adjusting. Hence the problem of infeasibility in SBP training is tackled by changing the goal of training and results in altering the role of hidden units.

### 4.2.2 THE RELATIONSHIP AMONG INPUT, WEIGHT AND OUTPUT

In SBP, hidden units enable a single weight state to be the goal of training. During performance, access to any output values is always associated with the learnt single goal weight state and an individual input value. This single weight state offers random access to any trained individual output values, as the output values can follow independent switches in the input values.

In FCBP, the outline of the relationship among input, weight state and output is: Each weight state in the goal weight state path provides random associations to just the I/O values at the associated training position both in training and performance.

In training, each weight state is driven by the errors used in conventional BP through the I/O patterns at the position. The cost of the memory for storing the weight state path can be calculated based on the positions involved.

In performance, the synchronization of sequences is involved between I/O and weights. If, at an instant, an input pattern in the I/O paths is seen whose output pattern contains the current associated signal values at that instant, the output signal values of the inputs can be only seen when the weight signal values associated with that instant of time are presented synchronously when the signal values of the input pattern are presented. Fig.4.0 is a general picture of the I/O and weights association in FCBP.

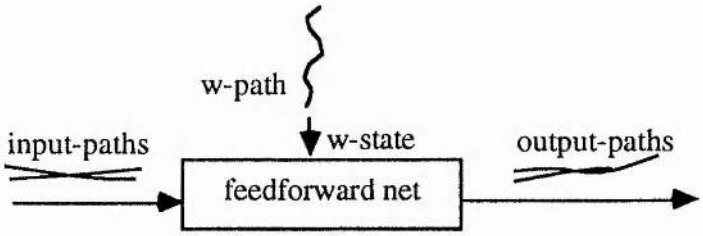



Fig.4.0 An associator using a trained weight path to process a set of I/O paths

## 4.3 Training speed and generalization capacity

This section will address the FCBP training and generalization capability and explore why FCBP in principle can speed up training and has a powerful capacity for generalization.

### 4.3.1 TRAINING

In conventional BP, a major cause of the slowness in training lies with the use of component error-weight surfaces for individual I/O patterns. Weights are changed either after the presentation of each I/O pattern value or after a cycle of all such values. In the first case, the component error-weight surfaces for the individual I/O pattern values are used separately to drive the training. In the second case the component surfaces are summed to provide an overall error-weight surface for training.

A zigzag tendency thus occurs whether it is seen as travel along ravines in an overall error-weight surface or as travel between separate surfaces. For more detailed discussions about ravines see, for example, Hinton (1987), Sutton (1986). Travel to a single goal weight state is likely to become an ever more lengthy process as the number of I/O patterns which are treated as independent increases. This is because the number of surfaces with opposing gradients at any weight states increases.

In FCBP, the training path itself may now form a valid goal weight condition when it contains a sequence of weight states yielding the desired I/O values.

The goal weight path approach considerably eases the ravine problem. For arrival at a goal weight path, all that is required is that a separate goal weight state needs to be found for I/O pattern values at a given position along the I/O target paths. If there is only one I/O target path, there will be only one surface at each position. If there is more than one I/O target path at each position, the number of the component surfaces summed at any position is still much less than those associated with all the I/O values along all the paths. Therefore, in FCBP each goal weight state should be found relatively quicker since only a number of component surfaces local to the current I/O position are used to find the weight state and the necessary network size is also relatively small.

Furthermore, when the size of the discrete training set of I/O values is increased in better approximation of the underlying continuous I/O paths, the difference between consecutive I/O values is decreased and hence the closeness between the consecutive I/O surfaces can be increased to speed up training. Once the first goal weight state has been found, the next goal weight states may be expected to be close together in their sequence. In FCBP, the speed up in training is based on converting a single set of I/O pairs into many smaller and closely connected sets.

Therefore the behaviour of a network may be described as: training moving along the target I/O paths, where there is an underlying continuum, produces a sequence of suitable

weight states for the desired I/O values along the paths. A change from the current input to another input along the I/O continuum may be associated with a corresponding continuous change along the goal weight path. This feature will be referred as the property of continuity. This means: if there is a continuous I/O mapping from I/O at time $t$ to I/O at time $t$+1, it is reasonable to suppose that there is a continuous weight mapping from weight at time $t$ to weight at time $t$+1.

A sequential problem is one where the order in which the input occurs has to be learnt as well as the individual associated outputs. Another important feature in the PBP approach then, is the order given by the training cycle for FCBP in which I/O associations are trained. The order of access after training may differ from the training order provided the training time is used as a coordinate index in space rather than time. See Fig.4.1. for details.

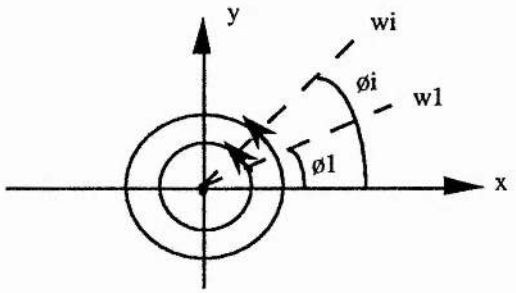



Fig.4.1 Two sequential training paths in two dimensional space

The direction of the training order in Fig.4.1 is indicated by the arrows along the paths. There is a weight state associated with each phase position $\boldsymbol{\phi_i}$. The $\boldsymbol{\phi_i}$ may be used either as describing the training time or as a coordinate index to access the weight state after training. This feature can be very useful after training (see further discussion in §4.3.2).

### 4.3.2 GENERALISATION

Through the property of continuity described above, the analogue nature of neural processing in conventional BP often provides the feature that an input will produce an

output near to the outputs of the neighbouring inputs (in coordinates). This feature will provide an underlying basis for production of outputs of untrained inputs in FCBP.

The aim of generalisation after training using the FCBP approach is to use a goal weight path representing a transition sequence of internal states within the same network which provides the desired I/O mappings for untrained as well as trained inputs in performance. The untrained patterns may include the untrained patterns along the set of I/O training paths or even outside the paths as discussed in §4.3.1.

More specifically, generalization in FCBP can consist of both spatial and temporal aspects when an appropriate space-time scheme is introduced. This is a scheme whereby input values are assigned unique times (which will be used later to refer to *time slices*). The space and time are measured relative to the bounds of the input values and the period of the training cycle respectively, so that each position corresponds to a specific moment of the simulated time, each input conjoins with the unique goal weight state occurring at the assigned time along the goal weight path. Any input value fitting the space-time scheme will then always get the same response from the trained network through the weight state which is associated with the position related to that particular time slice. Each time slice corresponds to a set of all possible inputs related to the particular time, some of the inputs may be along one of the training paths, some are not.

With the evolution of time some inputs are assigned to the time slices which correspond to *trained positions*, some to *untrained positions*. As an example, the diagram in Fig.4.2 shows a picture of assigning any input pattern represented in two dimensions (x, y) within the ranges of $x$, $y$: $[-a, a]$ a time value $t$ according to $t = \sqrt{x^2+y^2}$. Three time slices are shown in Fig.4.2, each is formed a circle in the input space. For example all input values assigned to $t=a1$ and $t=a3$ could be associated with two trained time slices and those assigned to $t=a2$ are associated with an untrained time slice.

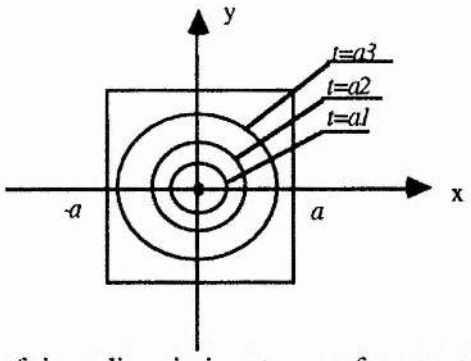



Fig. 4.2 The picture of time slices in input space for a certain space-time scheme

Within a particular trained time slice, there are only two kinds of recognition patterns, one kind is the trained pattern chosen from the paths; the other is the untrained pattern outside of the paths. The generalisation properties to those untrained inputs here are the same as those of SBP due to generalisation being based on a single weight state in both cases.

Between those trained discrete time slices, there are also two kinds of untrained inputs. One is the untrained input chosen from the paths; the other is the untrained input outside of the paths. For the first kind of untrained input, continuous variation of outputs corresponding to the continuous inputs needs to be considered. A continuous production of desired output can be attempted by having the goal weight path as a set of continuous signal sequences rather than the trained discrete values for each weight link. Such a continuous goal weight path may be simulated through interpolation of the trained discrete sequence of the goal weight states. The interpolation of the sequence of goal weight states constitutes the second form of generalisation in FCBP. This approximates the continuous goal weight path underlying the discrete goal weight states associated with trained positions, hence provides a suitable weight state for each moment in continuous time. The interpolated weight states determine outputs for this kind of untrained input.

In digital simulation, FCBP uses the continuous mapping ability of a goal weight path by training a finite number of discrete I/O chosen along the sets of I/O paths to produce a sequence of the goal weight states. An approximation to the arbitrary continuous I/O paths can then be produced by interpolating the rest of the goal weight path using the learnt goal

weight states. In this way, a neural network trained on a finite number of I/O positions can draw on a potentially infinite number of weight states to do generalisation for a potentially infinite number of untrained inputs and desired output values along the I/O paths. Also as indicated above, the accuracy of approximation may be increased by increasing the number of training positions to get more goal weight states to approximate the goal weight path. As training can be achieved with a fixed sized network, this implies that the accuracy of approximation can be increased in FCBP without having to increase the number of hidden units in the network.

For the second kind of untrained input outside training paths and associated with a untrained time slice, the approximated goal weight state associated with the time slice is also used for generalization. The generalisation properties to those untrained inputs are also similar to those of SBP, the generalisation is based on a single approximated weight state in this case.

- **The fundamental difference in generalization**

In FCBP, generalization of untrained inputs within trained time slices is the same as that of SBP, but very different in dealing with untrained inputs between trained time slices.

The following is a more detailed view of generalization based on the untrained inputs in FCBP. There may be the Untrained Output problem if untrained inputs are chosen within an untrained time slice. A method has been imposed to resolve the problem in FCBP.

As discussed above, in FCBP each I/O value in the input space can find the associated time slice whether it belongs to trained I/O paths or not. The I/O can be realised by either finding a learnt weight state associated with the time slice or working out a weight state based on the associated learnt weight states according to the time slice. For the latter case, an approximation to the weight state associated with the untrained time slice is needed. Generalisation for untrained inputs may be viewed as getting the I/O associations through a trained network by interpolation.

For an interpolation between inputs and outputs, an interpolation mechanism and the related parameters are needed. There are many existing interpolation methods, related to many ways for choosing the parameters for the method in order to have a good approximation. For example, one approximation method makes use of the nearest neighbouring I/O as its parameter, another method uses the average value of a set of neighbouring I/Os instead.

For untrained inputs along the training paths, generalization is also related to the way of selecting parameters and approximation mechanisms. Based on SBP and FCBP, one question to be asked is which of the two approaches have imposed parameter choices and flexible interpolation mechanisms in the model to perform a better generalization in principle for untrained inputs.

Learnt weight states may be viewed as acting as a set of parameters for a chosen interpolation method. In the SBP model, the networks have only a single set of parametric values, but in FCBP models, the networks have a sequence of such sets. In the SBP model, only constant interpolation can be used for approximation of a suitable solution, but in FCBP many orders of variable interpolation are available. These inherent features of the two models provide a theoretical basis for expecting the FCBP model to perform better generalization.

Refer to §3.3.2, another view of the fundamental differences between SBP and FCBP in realising untrained patterns is that the mismatch of the contour maps has to be solved using a new network topology or by adding more hidden units in SBP. This will be achieved in FCBP by increasing the number of training I/O patterns chosen from the paths without changing the network topology.

It can be seen that the difficulties and limitations of the Untrained Output problem mentioned in §3.3.2 using a single weight state may be resolved in general by FCBP approach.

### 4.3.3 THE OUTLINE OF FCBP

Replacing a single weight state by a weight path as the goal of training, FCBP not only removes the limitation of SBP for arbitrarily close approximating continuous functions in feedforward networks, but also increases the speed for training a large number of I/O sequential target values by using an underlying sequential nature.

A method of control over the number of hidden units is described. Using this method, each weight state needs only to produce correct responses required by the I/O values along any sequential paths at the related time. Each goal weight state should be trained relatively quicker since the associated problem size for each weight state and the network size are both relatively small.

In generalization, the Untrained Output problem can be resolved to a significant extent and many existing approximation methods can be explored and introduced into the FCBP model as the tools of generalization for untrained inputs. A suitable interpolation method chosen from a generalization tool kit can then be applied to solve different problems.

The specific features of FCBP can be summarised as follows:

(1) It is a feasible approach for training networks to approximate a kind of mapping when a large number of sequential I/O target values or complex binary mappings are involved; this is especially useful for approximation of I/O analogue output mappings chosen along continuous functions with arbitrary approximation accuracy.

(2) Because the number of the hidden units are decided by the number of the I/O target paths, no adjusting of the network topology is needed when the number of training data chosen along the paths is increased.

(3) A method has been imposed in FCBP to resolve the problem of Untrained Output:

Any higher order interpolation methods for weight state approximations can be employed by FCBP for realization of some untrained inputs.

The mismatch of the contour maps can be resolved through adding more training data along the training paths without changing the network topology or increasing the number of hidden units.

(4) FCBP also suggests a reasonable analogous method within SBP to decide the order of choosing I/O training patterns even in search of a single weight state as the goal of training in the SBP models. The path idea may speed up the training for finding a single weight state if there is an underlying continuum.

### 4.3.4 THE TRADE OFFS IN FCBP

It can be seen that there are some trade offs in FCBP:

(1) Many small stored weight states are used in FCBP instead of a single large weight state used in SBP. Since the total memories required in FCBP can be more than the amount needed in SBP, there is trade off between memory to store the whole sequence of the learnt weight states for the number of hidden units and the time needed for training.

(2) Although FCBP is a software technique, it is an approach which may be used as a hybrid of neural and digital computer architecture. There may therefore in practice be random access and automatic synchronization to the correct weight state for any input through the indexing of the weight matrix. In strict neural terms in FCBP, there is limited random access allowed to trained individual output values along paths during performance. Input must synchronise with weights in time for each correct I/O association in FCBP.

Conclusion: The aim of the FCBP approach is to provide a means for achieving arbitrarily close approximations of I/O mappings along paths within a fixed neural topology. In

FCBP this can be achieved with a fixed sized network with the help of using dynamic interpolated memories.

### 4.4 The FCBP Training Schemes

Two training regimes of FCBP have been explored in the thesis. The implementation of the regimes has been carried out using simulated neural network digital computer programs with the simulator **cbptool** which will be described in Chapter 7. Experiments based on the regimes can be found in Chapter 6.

#### .Regime 1

In regime 1, a single weight state transition is made for every change in input using the error-weight gradient and learning rate as in conventional BP. Training continues with many traversals of the I/O paths until there is a complete traversal of the paths with the associated weight state for each position along the paths generating an error below a fixed universal tolerance. A goal weight path has then been achieved.

#### .Regime 2

The direction and amount of each weight state transition is also computed using conventional BP. The difference from regime 1 is that consecutive weight state transitions are repeatedly made at each particular I/O position until the errors of the I/O patterns chosen from all the I/O paths at the position are below a fixed universal tolerance. Only then the training moves to the next I/O position. The last weight state resulting from the sequence of weight state transitions made for each training position is taken for the overall training termination test at that position. This test has the training under this regime continuing until the ends of the I/O paths. The last weight states at each position then form the goal weight state path.

The two regimes may be characterised thus: the first regime is aimed to merging its weight path with a goal path over a number of I/O cycles while the second regime is to keep its path intersecting the goal path over a single I/O cycle. The empirical issue raised by the two regimes is which one provides a more effective training.

### 4.5 The FCBP Generalization Schemes

Generalization in FCBP uses a suitable interpolation method to produce approximations of continuous functions by using the sequence of the trained goal weight states as the parameters. All the standard interpolation techniques can be considered and explored for weight interpolation. Two main factors need to be considered in choosing the interpolation techniques. One is the time expense in computation in working out the approximated weight state, the other is about the approximation accuracy. For example, if we aim to spend as little time as possible in the approximation of a goal weight state, a neighbourhood approach may be applied. This means that the goal weight state associated with the last trained position is used to approximate all the goal weight states associated with the untrained positions until another trained position is met. It can be seen that this approach spends no time in computing the approximated goal weight state. However this approach may lack sufficient accuracy in approximation for some cases. For higher accuracy, the standard Fourier interpolation may be a more complete but time consuming approach comparing with the neighbourhood approach. This is to approximate a goal weight states based on the Fourier analysis of the sequence of learnt weight states. By taking the $n$ goal weight states associated with the $n$ trained time slices in the goal weight path, Fourier analysis is used to generate the approximated goal weight states for the untrained time slices.

In the thesis, a simple and standard linear interpolation technique (LIT) which has both a relatively high degree of accuracy and limited computation involved has been explored.

The implementation of the LIT regime has been carried out in the simulator **cbptool**. Experiments using the LIT regime are shown in Chapter 6.

**Regime Linear Interpolation Technique (LIT) approach**

This generalization regime uses a conventional linear interpolation technique to approximate the weight state of a untrained pattern. The approximation is based on a linear interpolation of two learnt weight states associated with two trained time slices. The trained time slices are the neighbouring trained slices associated with the time slice of the untrained pattern.

For the $i^{th}$ untrained I/O patterns amongst $k$ such patterns regularly spaced between two neighbouring trained time slices, its weight state can be calculated using the two learnt weight states $W_1$ and $W_2$ associated with the two time slices through LIT. Each component weight has the form: $w_1 + \frac{i}{(k+1)} (w_2 - w_1)$, where $w_1$ and $w_2$ are component values of the weight states $W_1$ and $W_2$.

## 4.6 Conclusions

In this chapter, the FCBP model has been introduced and the reasons for why FCBP may have better training and generalization results comparing to that of SBP have been analysed. It is clear that FCBP is evolved from conventional BP but is a very different approach in both concept and methodology.

In general, FCBP is designed for the investigation of temporal associations and sequential signal processing in a parallel processing system. FCBP offers an extension of the feasibility of the back-propagation approach to training; and better approximation in generalization. FCBP is able to use temporal structure to allow variable approximation within a small and fixed sized network. The major features of FCBP itself have been presented in §4.3.3. It is concluded that:

(1) FCBP is a neural system with a way of modelling the effect of time which is appropriate for dealing with the temporal and sequential signal problems.

(2) More than one sequential path can be trained and generalized within a single system as long as two constraints are obeyed, which are: each of the single paths has the same number of the training positions, at each of the positions if input values chosen from different paths have the same input values, they will be associated with the same output values.

(3) Training is feasible for arbitrarily close approximation of I/O mappings where there is an underlying continuity. The more I/O training data chosen along paths, the closer a network will provide approximation of I/O mappings along the paths. In FCBP this can be achieved with a fixed sized network with the help of using dynamic interpolated memories.

(4) The Untrained Output problem is less severe in FCBP compared with that in SBP. The interpolation of weight states provides additional generalization power through a theoretical and feasible basis for resolving the Untrained Output problem for finite sets of I/O paths. Also, generalization can be achieved with a variety of approximation methods. Compared to the zero order weight interpolation in SBP, a much higher order approximation can be employed by FCBP.

# CHAPTER 5

# RECURRENT CONTINUOUS BACKPROPAGATION MODEL

## 5.1 Introduction

Another new approach called recurrent continuous back-propagation (RCBP) is presented in this chapter. Like FCBP, RCBP is also an approach based on the path-based backpropagation (PBP) framework using recurrent links to embody temporal and sequential capacity in neural networks. It provides a neural dynamic system for I/O associations by generating sequences of internal activity states and weight states.

This chapter is arranged to discuss what RCBP is, how its features compared with other approaches, and how to define internal states in terms of the activities. In order to get a clear picture of RCBP, detailed features are given with emphasis on how internal states can be embodied in PBP at each position.

In §5.2 the desired features of RCBP are presented and the role of the activity states of neurons in recurrent nets is reviewed. The notion of activation sequences is introduced to see if the additional activity dynamic brings significant additional power for the path-based approach. Then in §5.3 and §5.4 respectively, the training and generalization schemes are given. Finally a conclusion is presented in §5.5.

## 5.2 The RCBP model

• Why is RCBP needed?

A dynamic system differs from an I/O function in providing one-many I/O associations through varying internal states. In general FCBP is not always powerful enough to deal with the one-many associations needed for the following two reasons:

1) At each training position, FCBP is the same as SBP. There is a single weight state at each position for the feedforward network.

2) The source of internal state variation in FCBP through the weight state path is therefore not able to provide one-many associations at each position because the internal state at each position is constant.

• Connections between RCBP and SBP

If internal states independent of inputs and weights can be dynamically induced in a network, the network can become a dynamic system with inputs. Some investigation has been made in SBP (see §2.3). It will be shown here how a recurrent network, with a suitable dynamic rule for achieving the dynamic feature, can also support such dynamics in PBP when the three features of PBP — time modelling, weight state sequence and fixed neural network topology are incorporated in the new approach RCBP.

The subsequent subsections are to discuss four aspects related to the design of RCBP. The first one is about how internal states can be embodied in PBP at each position. This shows that a suitable dynamic rule is needed for recurrent networks to represent the internal activity states. The second is about how such a dynamic rule is established in RCBP and how the activity sequence works. The third reviews the relationship amongst I/O pattern, weight state and activity state in RCBP. An outline of the RCBP model is then presented in the last subsection.

### 5.2.1 THE INTERNAL STATE MODEL

As internal variation both weight states and activity states can be viewed as internal states in a neural network, this section requires a specific view of internal states as provides of one-many associations.

Now let's consider embodying a dynamic system's internal states at each training position in PBP and in particular whether there are any means which can make outputs be driven not only by current inputs and the weight state but by something which is related to the effect of past history.

Because PBP uses a single weight state at each training position, the requirement for the internal states is similar to that of solving the one-many association problem in SBP. As shown in §2.3, many approaches for achieving the associations have been explored in SBP using recurrent networks and suitable dynamic rules. According to the similarity of PBP and SBP, this implies that there should be some ways to embody the internal states in PBP as well.

In this thesis a similar solution as for SBP is attempted in PBP, namely the prearrangement of one-many I/O associations through a recurrent network and its activity states. This is to say that dynamic activity states are used for modelling the dynamic system's internal states in PBP at each position.

Since a recurrent network has been chosen as the topology for the approach, the next question that arises is how to design the dynamic rule for the network.

### 5.2.2 HOW TRANSITIONS CAN BE DESIGNED TO ACHIEVE DYNAMIC BEHAVIOUR IN RCBP

The general dynamic features that we wish to implement in RCBP include: embodying the internal states; without losing the continuum features among I/O paths and weight states path.

1) The control of activity transitions is needed

Compared with feedforward networks, the main additional problem in recurrent networks is how to control the recurrent activity during training and performance. The reviews given in §2.3 have shown us many existing approaches. Some approaches may have control

over the activity in indirect ways, such as the forward propagation model (Williams and Zipser, 1988), which compensates for poor activity changes through succeeding weight changes. More direct methods directly controlling the activity changes during training are also possible, for example the moving targets model (Rohwer, 1990).

In SBP, when recurrent networks are made to be at equilibrium, the equilibrial activity state is completely determined by the current I/O and weight state. That is, a single weight state recurrent network at equilibrium, as demonstrated in papers such as that in Almeida's (Almeida, 1989b), can only provide an I/O function. This relationship is therefore unable to provide activity states as the required internal states mentioned above.

In RCBP, non-equilibrial activity states are needed to provide one-many associations through internal states which are independent of the current I/O and weight state. The inherent dynamics of such networks needs to be designed, which entails the design of the activity state and weight state transition rules.

The review of the existing models in §2.3 reminds us that different transition rules will provide very different network behaviours, therefore before designing the dynamic rules for networks, the desired type of network behaviour must be established.

2) The desired type of network behaviour for RCBP

The desired type can be viewed in terms of two aspects:

In performance— a goal weight state together with a generated activity state associated with each training position enables recurrent networks to act as a machine with internal states to produce the correct output for its given input at the particular position.

In training— a set of activity sequences is generated together with a goal weight sequence to be found according to the set of I/O training tuples chosen from the training paths;

3) The existing dynamic approaches in SBP

In SBP, an approach will be referred to as *static RBP* when the aim is for a recurrent network to provide a I/O function through achieving equilibrium and as *dynamic RBP* when the aim is for a recurrent network to provide a dynamic system. The latter is the main focus of attention here.

As reviewed in §2.3, dynamic RBP has a homogeneous time delay scheme whereby every link has an one-step time delay.

Although dynamic RBP can train an I/O set and generate an activity sequence in the trained order for the set, this kind of approach is not suited for incorporation into RCBP. This is for the following reasons:

• When a recurrent network is not at equilibrium, the activity state is not only dependent on the weight state and input values but is also dependent on the activity values at the previous moment. In this way, the change in the non-equilibrial recurrent activity is only partially dependent on the change in environment input. When the activity is far away from equilibrium, a sudden large change in recurrent activity may occur for arbitrarily small change in environment input.

• Such sudden large change in activity undermines the property of continuity (see §4.3.1) used by PBP to speed up training and enhance generalization. In particular, dynamic RBP has no control over recurrent activity between positions through the continuity underlying I/O training patterns. Training is undermined because the consecutive goal weight states are far apart. Generalization is undermined because the weight data points are far apart and the intermediate variation may be wild.

• Interpolation of weight values is improved in PBP by exerting control through continuous weight mappings. For this reason, dynamic RBP will not be used in RCBP.

4) The scheme employed in RCBP

A scheme is needed which makes effective use of the continuity underlying the I/O training sequence to control the evolution of the sequence of activity and to have a continuous mapping underlying the goal weight state sequence.

• The one-step delay scheme

The constraint for RCBP is to have a dual time delay scheme with one scheme for the recurrent links and another for feedforward links.

Two kinds of links may be made between a pair of neurons, one labelled feedforward, the other recurrent (if each exists).

The recurrent links have a one-step time delay where the output from a source neuron takes one step in time to travel along the link and become input to the destination neuron. The step length is given by the interval between environmental input changes. This scheme ensures that present recurrent input is determined by previous recurrent output so that there is both independence from current environment input and a dependence on past history.

The scheme is designed to fit with non-recurrent SBP at each training position except the first (see below). The weight changes are calculated from information attached to each moment in isolation. The feedforward links have zero time delay so that weight changes are calculated from information attached to each moment in isolation as in standard (non-recurrent) back-propagation.

The activity states at the previous training position along any recurrent links are taken together with current I/O and weights to evaluate the activity values at the current training position. The dynamics of the activity states in the recurrent networks can be described as:

$$x_i(t) = \sum_{(i,j) \notin RL} w_{ij}(t) y_j(t) + \sum_{(i,j) \in RL} w_{ij}(t)\, y_j(t\text{-}1)$$

$$\text{and } y_i(t) = f(x_i(t)) \qquad (5.1)$$

where f(x) is the logistic function; RL denotes a set of index pairs (i,j) for recurrent links $l_{ij}$ linking from unit $j$ to unit $i$; $t$ is the moment associated with training positions;$w_{ij}$ (t) is the weight link values linking from unit $j$ to unit $i$; $x_i$ (t)and $y_i$(t) are the excitation and activity value of unit $i$. It remains to decide the initial conditions for Eq.(5.1).

• The static RBP employed at the first training position

The initial condition is that static RBP is employed at the first training position to establish recurrent activity near equilibrium for each I/O pattern. Equilibrium will be used for the first training position to set the initial activity values in order to establish control over the recurrent activity and have a continuous weight mapping (see discussions in 5) below).

The single weight state and the static recurrent network at the first position dictate that only an I/O function is to be targeted at this position.

5) The dynamic behaviour of RCBP

At the first position, an activity $A$ is obtained when the network is at equilibrium. By sitting at an equilibrium state, the set of activity states of the network at the training position is fully determined by the set of I/O values and the learnt weight state at that position.

The non-equilibrial evolution rule Eq.(5.1) is applied to calculate activity for all the positions barring the first one. By setting the network at equilibrium for the I/O patterns at the first training position and the second position's I/O patterns close to the first I/O values, according to Eq.(5.1) the two associated activity states will be close as well. This will form a basis for evolving the activity path with a certain level of control through underlying I/O continuity.

The input and weight state values are thus more loosely coupled with the equilibrium activity state at each position after the very first one. Input and weight changes might appear to move the activity states further away from equilibrium at successive positions. However the activity transitions accompanying positional change also allow the activity to

move towards a new nearby equilibrium for each new input. Provided the input changes are small, the activity should remain reasonably close to each new equilibrium. This in turn controls the size of the activity transitions to keep them small enough to aid training and generalisations discussed earlier (§4.3.1).

The loose coupling of $I$ and $W$ with $A$ means that the dynamics of $A$ is only partially determined by $I$ and $W$. $A$ is also determined by the dynamics of $A$ itself except at the first position. The intention is for $A$ to be able to provide the required internal states while remaining under continuous control.

As far a training is concerned, at each training position (except the first one), the recurrent network may be viewed in a feedforward fashion. The recurrent links for each unit can be seen simply as extra inputs, whose values are those of the previous outputs from the source neurons. The excitation of each unit in the net is then calculated as that in an ordinary feedforward net. For example, at position $p_t$ associated with time $t$, if a unit $U_i$ has two feedforward links from units $U_{j1}$, $U_{j2}$ and three recurrent links from units $U_{k1}$, $U_{k2}$, $U_{k3}$ respectively shown in Fig. 5.1a, the evaluation of activity $U_i$ in the recurrent network as shown in Fig. 5.1a is equal to the evaluation based on the equivalent feedforward network at that time shown in Fig.5.1b.

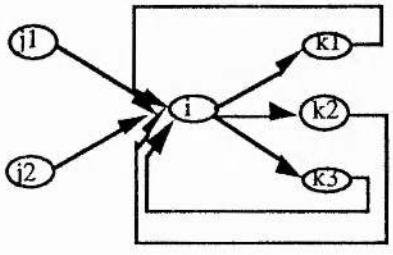



Fig.5.1a A recurrent net

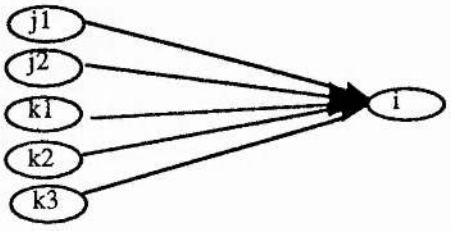



Fig. 5.1b A feedforward view of $U_i$

The scheme for changing the weights in the network is therefore the same as that for an equivalent feedforward network using conventional backpropagation at each of those training positions.

### 5.2.3 THE RELATIONSHIP BETWEEN THE NETWORK VARIABLES

At each position, there is one weight state along the weight state path acting in the system, which together with a set of activity states at the position along each activity path corresponding to each I/O path provide the desired I/O associations. The goal weight path amd activity paths therefore act to provide a sequential store of I/O machines with internal activity states. Fig.5.2 shows a picture of the relationship in RCBP.

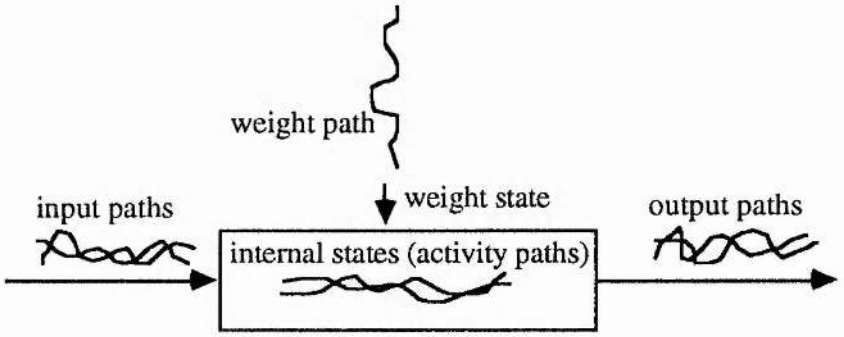



Fig. 5.2 The relationship among the I/Opaths, w-path and activity-path in RCBP

Along the paths, continuous control is aimed at by using equilibrium at the first training position and the Eq.(5.1). The property of continuous control means that a small change from the current input to another input along an I/O continuous path may be associated with a small continuous change along the goal weight path and the activity path.

RCBP like FCBP is also able to closely approximate I/O associations along continuous paths by training on a finite number of discrete I/O tuples chosen along the paths. The generalization in RCBP is based on an approximation through interpolation in both the weight state sequence and the activity state sequence. As for FCBP, various interpolation techniques may be applied.

In RCBP, the minimal number of hidden units are again determined according to the number of I/O paths. However due to extra inputs to neurons provided by the recurrent links the solution weight state may exist for a given number of I/O paths without as many

hidden units as FCBP requires. As for FCBP, the number of hidden units is relatively much smaller than the number for all the I/O training tuples in all the I/O paths. The same number of hidden units is used for each different position.

### 5.2.4 SUMMARY FEATURES OF THE RCBP APPROACH :

The approach serves as an example to show the basic concept and methodology of the path-based approach with internal activity states. The aim is to make a contribution to the control of recurrent activity through the property of continuity.

• The effect of time appears through the sequential change of position:

During training, the set of activity states at the current position are evolved from the set of activity states at the previous position together with the learnt weight state and I/O set associated with current position. The first goal weight state is found using RBP. The rest of the goal weight states, each associated with one position, are found using feedforward SBP with the activity states treated as extra inputs at a position. The effect of past history is represented through the activity state transitions.

• Each activity state at the previous position contributes to determining suitable internal states at the current position. This provides a dynamic system with one-many associations.

• Continuous control over the activity path may be derived from underlying continuity amongst the I/O. The activity path may be evolved under a degree of continuous control. This is achieved through equilibrium and the dynamic rule described in Eq.(5.1).

• The sequential nature of I/O associations has been further developed in the RCBP approach compared with that in the FCBP approach through the activity sequence. RCBP should be considered as an approach to providing path-based dynamic models in neural networks. Although RCBP only has a single internal state at the first training position, the technique nevertheless provides a path-based dynamic system for dealing with one-many associations

### 5.3 The RCBP Training Scheme

An RCBP training regime has been designed based on the theory in the previous section. The implementation of the regime has been carried out using the simulator **cbptool** and will be described in chapter 7. Some experimental results based on the regime can be found in Chapter 6.

In RCBP, the regime can be used for training more than one I/O target path. The error-weight gradient, learning rate and momentum coefficient values are all calculated as in conventional backpropagation. The direction and amount of each weight state and activity state transition however are computed by one of two methods depending on the two kinds of training positions: the first training position or the others.

The strategy of the weight state transitions in RCBP is similar to that used by the regime II of the FCBP approach. That is, consecutive weight state transitions are repeatedly made at each particular I/O position until the errors are below a fixed universal tolerance. At each position the last weight state therefore results from a sequence of weight state transitions which are made for the position. The termination of overall training on the position is tested by this last weight state. Only then does the training move on to the next I/O position. This test has training under the regime continue until there is a complete traversal of I/O positions on the paths with satisfactorily low error. All the last weight states at each position then form the goal weight state sequence.

The activity state transition and the calculation of the gradient descent along the error-weight surface for finding a goal weight state can be described in the following way: At the first training position, a goal weight state is found by using the fixed-point recurrent network learning algorithm RBP (see Pineda, 1987; Almeida, 1987; Rohwer, 1987). The set of current activity states are obtained for the network at equilibrium. An activity state is trained for each training path at the first position. This forms the first set of activity states for the generated activity sequences. For the second and all subsequent positions, the set

of generated activity states at the previous training position and the I/O values at the current position act as parametric indexes for the error-weight surface used for finding the goal weight state at the current position. When a goal weight state has been found, a correct set of activity states for producing the desired outputs is generated by the set at the previous position, together with the current environment input. This set of activity states is then taken to be the set of generated activity values for the position.

Suppose we are to train I/O values along $m$ paths, where each of the paths has $n$ training positions. Let the $k^{th}$ of such sequences be:

$S_1^k, S_2^k, \ldots S_n^k$,

where $S_t^k$ denotes the combination of input value $I_t^k$ and output value $O_t^k$ at $t^{th}$ training position along the $k^{th}$ sequence. We wish to find a sequence of learnt goal weight states

$W_1, W_2, \ldots W_n$,

and a learnt goal activity states sequence

$A_1^k, A_2^k, \ldots A_n^k$.

A more formal representation can be written showing the relationship of input, target output, activity and weight through two sequences of sets below in Fig.5.3. Each set in both sequences corresponds to each training position. The first is the sequence of variable sets for training, the second represents the sets of results which show the goal of weight state and the generated activity state when the training has completed at the position during training.

In Fig.5.3, $A_{init}^k$ and $W_{init}$ denotes the initial activity state and weight state before training commences respectively. Each learnt goal activity and weight state is carried forward to the next training position.

It can be seen that training in RCBP finds each goal weight state for each training position using conventional BP. The technique then computes the set of generated activities at the current position and stores these weight and activities indexed by position.

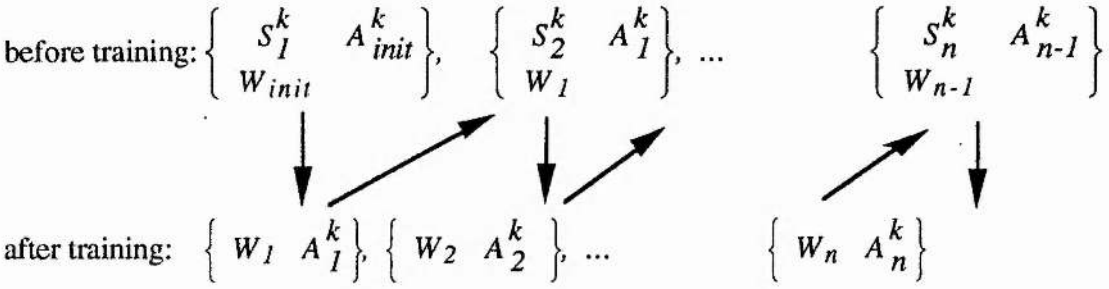



Fig. 5.3 A picture of RCBP training

During training, at each position the set of generated activity states from the previous position need to be present so that current activities can be computed. During performance, only the set of activity states at the first position together with the sequence of the goal weight states need to be stored. The whole sequence of the activity state sets does not need to be stored because this sequence can be generated from the very first set with the help of the learnt goal weight state sequence and I/O sequences during performance.

## 5.4 The RCBP Generalization Scheme

A major aim of the RCBP approach is to use a goal weight path representing a transition sequence of I/O machines within the same network with a generated activity path as internal states to provide the desired I/O associations for untrained as well as trained inputs. In principle, generalization in RCBP is very similar to that in FCBP. All the interpolation techniques applied in FCBP for approximating a goal weight state path can be considered and employed here for approximating a goal weight state path and activity state path.

Generalisation in RCBP can also be considered in both spatial and temporal aspects when an appropriate space-time scheme is introduced (refer to the discussions in §4.3.2).

However, discussion of these aspects would be very similar to FCBP. Consequently, discussion of generalisation technique is restricted here to interpolation aspects.

The linear-interpolation technique (LIT) as a generalization regime described in FCBP (§4.5) has also been explored and implemented in RCBP. The implementation of the LIT regime has been carried out in the simulator **cbptool** (§7.5.5).

.Linear interpolation technique for RCBP

For approximating a goal weight state for an untrained I/O pattern associated with an untrained position, a linear interpolation is employed. This uses two learnt weight states associated with two trained positions which are the neighbouring trained positions of the position associated with the untrained pattern along the training or generalization path.

The approximated weight state for the untrained pattern is carried out by using the learnt weight states. If the untrained pattern is the $i^{th}$ untrained pattern for recognition out of $k$ such patterns regularly spaced between two neighbouring trained patterns, the approximated weight state can be calculated where each component of the weight has the form: $w_1 + \frac{i}{(k+1)} (w_2 - w_1)$, where $w_1$ and $w_2$ are the values of corresponding component weights in the two trained weight states.

For approximating an activity state associated with an untrained position, a slightly more complex way of approximating can be done. The untrained activities will be computed in the same way as the trained activities. That is, in order to approximate the activity associated with the untrained position $(i+k)$, $0<k<1$(where $k$ is the $k^{th}$ untrained pattern for recognition out of $n$ such patterns regularly spaced between two neighbouring trained activity states $A_i$ and $A_{i+1}$), the previous activity state at position $(i-1+k)$ is used in conjunction with the weight state and environmental input at $(i+k)$. A recursive approximation of the previous activity would involve all previous moments and hence be time consuming. Instead, an interpolation similar to that for the weights is used based on

the trained activities at $(i\text{-}1)$ and $i$. That is, the activity state at $(i\text{-}1+k)$ is calculated according to :

$$a_1 + \frac{k}{(n+1)} (a_2 - a_1)$$

where $a_1$ and $a_2$ are the values of the corresponding component activity values in the two trained activity states at $(i\text{-}1)$ and $i$.

This rule of finding suitable training positions can be applied to all the untrained positions except the positions between the first and second training positions. For these positions, there is no previous activity information available. For approximating any untrained activity states between the first and second training positions, the generated learnt activity states at the first and second positions are used to directly interpolate the required activity state using the above expression.

### 5.5 Conclusions

The RCBP approach has been introduced in this chapter. Through a close look and discussion of the approach from the structure design to the inherent features of the model, it is clear that RCBP is similar to the FCBP approach in the sense that both approaches are based on paths, but RCBP is a very different approach in comparison with FCBP, by controlling the dynamics of activity states of units in networks, RCBP provides an approach of path-based dynamic system.

The major features of RCBP have been summarised in §5.2.4. A comparison with the features of PBP and FCBP is as follows:

(1) Like FCBP, RCBP is a sequential technique which allows arbitrary approximations of a set of continuous I/O associations within a given topology. RCBP provides a dynamic system with input in contrast to FCBP's I/O function.

(2) A trained network in RCBP produces multiple I/O associations in the form of desired output paths when the two constraints similar to that in FCBP are obeyed. The first

constraint is that each of the paths has the same number of the training positions. The second is that there are no one-many associations involved at the first training position. The desired output paths can be produced in reaction to independent switches in the input paths and associated internal state paths during performance though it is much more restricted in the independence than that in FCBP. The restriction on the independence switches is that in RCBP the switch cannot happen at each moment but after complete traversal of a training path.

(3) In providing a version of a path-based approach with dynamic system features, RCBP offers an extension of the FCBP approach. The temporal structure and internal states allow variable approximation within a controlled and fixed size network with controllable internal states.

# CHAPTER 6

# EXPERIMENTS AND ANALYSIS ON FCBP AND RCBP

## 6.1 Introduction

In this chapter, several experiments based on both the FCBP and RCBP models are presented and their results are analysed. These experimental examples are not only chosen to show the concepts and methodologies of both new models in practice, but are also used to explore the basic features associated with the particular capacities of training and generalization in dealing with temporal and sequential signal processing using the models.

In §6.2, a simple example is chosen to demonstrate what FCBP can do within a fixed topology with enough memory while SBP needs increasingly many hidden units and hence takes a longer time. This gives an insight into the analogue and sequential aspects of FCBP. In section §6.3, more capabilities of FCBP are shown in exploring generalization and features of the hidden units in FCBP. In §6.4 comparisons of the training speed and generalization accuracy of both FCBP and SBP models are presented. In §6.5 a set of experiments are carried out to test if the concepts embodied in the FCBP will also help to speed up training when the goal of training is a single weight state. In §6.6 an example shows that training on RCBP can be carried out and used to approximate complicated signal processing tasks which are not achievable by applying FCBP. Finally conclusions are presented in §6.7.

## 6.2 What FCBP can do without hidden units

Here we present how the training by FCBP is fundamentally different from the SBP approach.

### 6.2.1 CXOR TASK DESCRIPTION

A feedforward neural network will be trained on a task to exemplify the analogue and sequential aspects of FCBP. The general task is to associate correctly the inputs from particular positions around the edges of an unit square in a 2-D input space with desired output values. The network has two input units and one output unit. Suppose the associated output value $\delta$ is based on a continuous function of the form:

$$\delta = (1\text{-}2\varepsilon) \mid input_1 - input_2 \mid + \varepsilon \qquad (6.1)$$

where $\varepsilon$ is a constant within a range such that $0 < \varepsilon < 0.5$. According to this equation, when the paired combinations of input values are chosen along the edges of the unit square, the output value $\delta$ will either increase or decrease linearly from one extreme value, $\varepsilon$, to the other, $1\text{-}\varepsilon$, along each side of the unit square. The I/O relationship is shown in Fig. 6.1.

The four input pairs at the corners of the unit square are (0,0), (1,0), (1,1) and (0,1). According to Eq. (6.1) the output values have the extreme values of $\varepsilon$ or $1\text{-}\varepsilon$. Both values are found at the corners in alternation around the unit square starting with the value $\varepsilon$ associated with (0,0). It can be seen that this problem is equivalent to the well known XOR problem. Because the XOR problem can be considered as a special case within the continuous unit square problem, we name this task a *Continuous XOR problem* (CXOR).

CXOR is a kind of problem which involves continuous analogue I/O mappings based on using a given discrete I/O mapping to approximate the underlying continuous one. There are two major reasons for selecting this general task. Firstly, the incorporation of the XOR problem will allow us to show results which are comparable to those familiar in conventional BP. Secondly, since there is only a single I/O path associated with the task, it helps to show clearly the difference between FCBP and SBP both in training and generalization at a simple level.

Several specific tasks are derived from this general task by varying the training set. In the following two sections, training is carried out based on the I/O tuples chosen from the square using a simple network structure shown in Fig. 6.2. This problem serves as an introduction to FCBP without hidden units. The experimental results show some fundamental differences between FCBP and SBP.

### 6.2.2 TRAINING CXOR WITHOUT HIDDEN UNITS

Three experiments are carried out on two sets of I/O training tuples which are all chosen along the edges of the unit square but with two different step sizes between tuples; the associated output values are assigned based on the I/O relationship described in Eq. (6.1) with $\varepsilon$ =0.2. One set of the input training tuples are chosen along the edges with step 1.0 (4 tuples) and the others with step 0.25 (16 tuples). All three experiments are based on 100 different initial weight states initialised randomly within the range of -1.0 and 1.0. The output tolerance $\beta$ is chosen to decide the universal error tolerance $0.5\beta^2$ in training.

The learning rates displayed in the results were found as follows: A preliminary investigation shows that the interval of [5,15] yields fixed learning rates producing successful training for most cases. This interval was explored by randomly searching for the first successful integer learning rate. If no integers in the interval produced successful training, failure was recorded. In all successful cases, the learning rates of either 0.5 above or below the successful integer learning rate value were tried out to make a final improvement in finding the best of the three learning rates for each experiment.

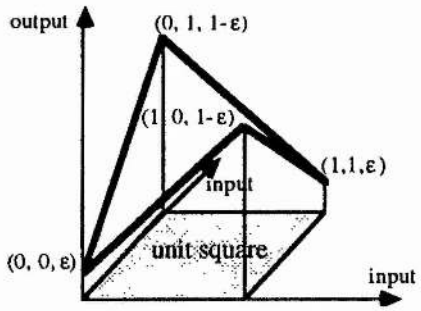



Fig. 6.1 The I/O relationship of CXOR

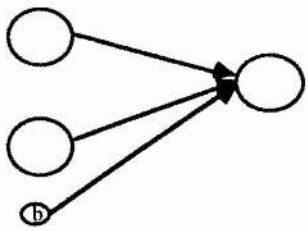



Fig. 6.2 The 2-1 network

According to the chosen training tuples and parameters described above, the training results based on both FCBP and SBP approaches are shown in Table 6.0 using the network shown in Fig. 6.2 and regime FCBP-II.

Table 6.0 Training on CXOR using FCBP

| index | training step size | output tolerance | learning rate | number of weight transitions | successful trials |
|---|---|---|---|---|---|
| @1 | 1.00 | 0.2 | 11.0 | 4.96 | 100 |
| @2 | 0.25 | 0.2 | 11.0 | 16.01 | 100 |
| @3 | 0.25 | 0.05 | 10.5 | 21.83 | 100 |

The experiment in @2 is based on chosing more discrete I/O analogue tuples along the square compared with that in @1. The experiment in @3 is based on increasing the accuracy required at each training position.

The results in Table 6.0 show that (1) when based on the same output training tolerance 0.2, the average number of weight transitions per position (i.e. the inverse of the training speed) is slightly decreased when the number of the training tuples is increased, but the total number of transitions is significantly increases; (2) Compared with those in @2, the results in @3 show that the accuracy of the training can be substantially increased without an excessive increase in training time.

The results confirm the theory that: 1) no hidden units are needed in these training cases; 2) the accuracy of approximation to the underlying continuous function can be raised by increasing the number of I/O tuples without changing the structure of the network (see §4.2.1).

The above training problems cannot be tackled using the conventional SBP approach if no hidden units are used. A number of hidden units are needed for the SBP approach because the 4 or 16 training tuples are linearly inseparable without hidden units for any output training tolerance (See §3.2). Therefore the training of the network shown in Fig. 6.2 fails when based on the SBP approach. However, training succeeds and is fast when based on the FCBP approach.

### 6.2.3 Training CXOR to Explore the Feature of Stepping Stones

Another set of experiments on CXOR is made for testing the feature of 'stepping stones' in FCBP using the network shown in Fig. 6.2 and regime FCBP-I. Each 'stepping stone' is an extra training position chosen along I/O paths and it is employed to play a role in the training process but is not used in testing the goal weight state for stopping training. It is used to explore experimentally what the relationship is between adding extra training tuples and the training feasibility with various training accuracies. The accuracy is the output tolerance in training.

All experiments are based on 100 trials and a set of suitable learning rate values, which both are chosen or found in the way described in §6.2.2. Here experiments tested training feasibility through three groups of recognition and training sets. Each recognition set consists of the tuples which are the desired I/O after training. These tuples are selected along the edge of the unit square with a certain training step size. Repeatedly halving the training step size relative to the recognition step size produces the related training sets in the group. The training tuples which are not in the recognition sets are extra tuples, they act as stepping stones for speeding up the training. The total number of weight transitions is

recorded when the first loop of the learnt weight states for the associated recognition tuples has been found. Table 6.1. shows the results.

Table 6.1 Training on FCBP-I

| index | recognition step size | training step size | output tolerance | learning rate | number of weight transitions | successful trials |
|---|---|---|---|---|---|---|
| @1 | 1.00 | 1.00 | 0.2 | 14.5 | 5.278 | 36 |
| @2 | 1.00 | 0.50 | 0.2 | 7.8 | 16.04 | 100 |
| @3 | 1.00 | 0.25 | 0.2 | 15.0 | 16.00 | 100 |
| @4 | 1.00 | 0.125 | 0.2 | 15.0 | 32.00 | 100 |
| @5 | 1.00 | 0.0625 | 0.2 | 15.0 | 64.00 | 100 |
| @6 | 0.50 | 0.50 | 0.1 | - | - | 0 |
| @7 | 0.50 | 0.25 | 0.1 | 10 | 19.31 | 100 |
| @8 | 0.50 | 0.125 | 0.1 | 13 | 32.18 | 100 |
| @10 | 0.25 | 0.25 | 0.05 | - | - | 0 |
| @11 | 0.25 | 0.125 | 0.05 | 13.5 | 36.01 | 100 |
| @12 | 0.25 | 0.0625 | 0.05 | 15.0 | 64.25 | 100 |

It can be seen that training is very fast in most cases. But there are also some poorer results and complete failures. The failures here all happened when the training step size is the same as that of the recognition step size. In those cases, training was either mostly or entirely unsuccessful over the learning rate range [5,15] (in @1, @6 and @10).

It is suggested that the above failures arise because FCBP-I requires a degree of precision in the single weight changes between error-weight surfaces. There is a need to land at a weight state for the current I/O tuple which has a suitable error-weight gradient to generate an immediately successful next weight state for the next I/O tuple. As this need is generally uncertain to be satisfied for all the consecutive I/O tuples around the I/O cycle, this regime's training may take a longer or infeasible time as in the cases with more I/O tuples in the training set (such as that in @6 and @10).

In all other cases, the results show that : (1) There were no training failures in all those cases with stepping stones; (2) The average number of transitions per position was not increased but decreased with the increase of the number of the training positions. This suggests that extra training tuples bring extra intermediate error-weight training surfaces which benefit the smoothness of training.

## 6.3 FCBP training and generalization with hidden units

In §6.2, the experiments have shown that hidden units are not needed in some situations by FCBP where SBP needs them. However hidden units still play an important role in the FCBP approach. Details of the role of hidden units in FCBP models have been discussed in section §4.2.1. Here experiments show the results for the training and generalisation involved in more complex problems where hidden units are needed in FCBP. It is hoped that this set of experiments can help us to explore and see the FCBP capability more completely. The training results for FCBP are also intended to show the trade-off between doing several searches in a number of relatively small weight space against doing a single search in a relatively large weight space.

### 6.3.1 TASK DESCRIPTION AND DISCUSSION

A 4-spirals problem is used to show how the training and generalization are carried out using the FCBP approach where hidden units are required. In this section a description of the task is given.

The 4-spirals problem is a sequential variant of Wieland's 2-spirals problem (Lang & Witbrock, 1988). That is, there is a natural set of 4 I/O paths given by the 4 spirals. In the 4-spirals problem, there are four nested spirals used as the 4 paths, the ones starting in the north and south positions have an associated analogue target output range between 0.0 and 0.4 interpreted as 0, and the other two starting in the east and west positions have a target

range between 0.6 and 1.0 interpreted as 1 (Fig. 6.3). A single system is needed to approximate the associations of the 4-spirals.

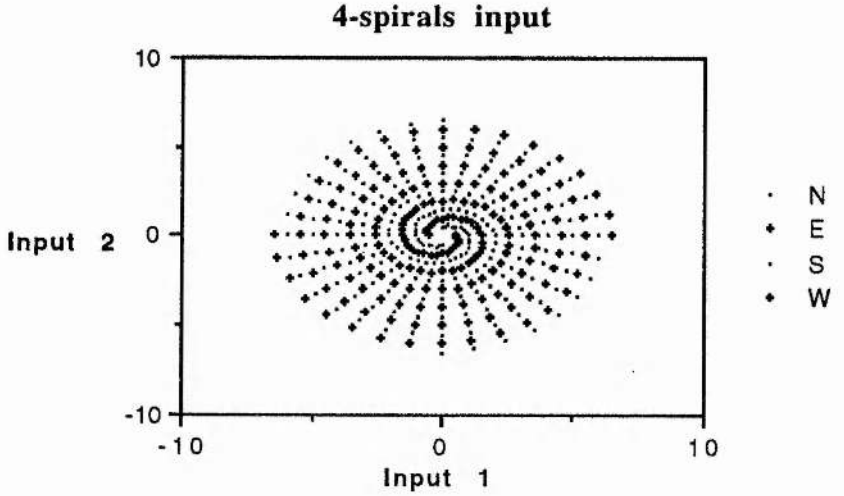



Fig. 6.3 The 4-spirals picture

As mentioned in §4.6, FCBP can be applied to approximate a single system for associations along several I/O paths as long as the I/O paths are subject to the two restrictions summarised in §4.6.

In the 4-spirals problem, each position consists of I/O tuples at the same distance along their respective spirals. At each position there are four linearly inseparable I/O values along the different spirals which need to be assigned the same goal weight state. The four inseparable values entail an architecture for FCBP with a power similar to that needed for solving the XOR problem using conventional BP. That is, at each training position the training is similar to solving the XOR problem. So the 4-spirals problem requires the use of hidden units in FCBP.

Two kinds of network topologies for the 4-spirals problem have been examined. One has a single layer of 2 hidden units (Fig. 6.4) which, being equivalent to solving XOR in SBP, is one of the smallest strictly layered architectures for training the 4-spirals problem in FCBP. The other is a fully connected (§1.4) feedforward network with three hidden layers as shown in Fig. 6.5 which is not just an example of an over rich structure for training the

problem in FCBP but also is the structure used by Lang and Witbrock's SBP approach for the 2-spirals problem (1988). It was chosen since it will help to do comparisons with SBP. All experiments on the 4-spirals problem use these two network topologies.

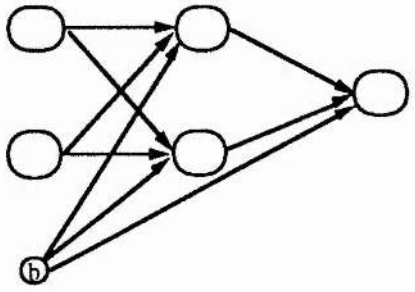



Fig. 6.4 The 2-2-1 Network

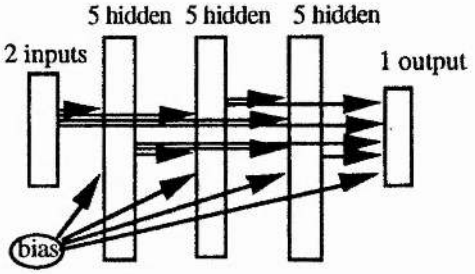



Fig. 6.5 The 2-5-5-5-1 network

### 6.3.2 TRAINING RESULTS AND ANALYSIS

Training on the 4-spirals problem was made using the two network topologies mentioned in the previous section. Both topologies were tested for a good pair of learning rate and momentum coefficient values over 100 trials where the initial weight states were chosen randomly within the range -0.1 and +0.1. As suggested by Fahlman (1988), the learning rate and momentum coefficient values were preliminarily investigated within the range of [0,1], in the way described in §6.2.3. The average number of weight state transitions taken for a successful trial was computed where a goal path was found within 20,000 and 40,000 weight state transitions at each training position for the 2-2-1 and 2-5-5-5-1 topologies respectively.

The training regime used is that of regime II for each 97 training positions. The training positions start from the inside and go towards the outside of the spirals with a constant angle of change. The results of training the 4-spirals problem using FCBP are shown in Table 6.2.

Table 6.2 Training results for the 4-spirals problem

| index | network topology | learning rate | momentum value | successful trials | average total wt transitions | ave. wt transitions on first position |
|---|---|---|---|---|---|---|
| @1 | 2-2-1 | 1.0 | 0.9 | 89 | 10,600 | 7,973 |
| @2 | 2-5-5-5-1 | 0.15 | 0.9 | 100 | 22,916 | 20,906 |

Here output tolerance = 0.2;

As discussed in chapter 4, an important feature of FCBP is that when the input training data are chosen with an underlying continuity, there is the inherent feature of closeness between weight states associated with consecutive training positions. This again implies that the FCBP approach can make use of this feature in training to achieve the goal weight sequence quickly. The training speed should increase significantly after training has been completed at the first training position.

The data obtained is in support of this hypothesis. The succeeding weight states are obtained very much more quickly compared with getting the weight state for the first position — the average number of the weight state transitions for the first I/O position is 7,973 and 20,906 for the two test cases, and the total weight state transitions are much less than 7,973*97 and 20,906*97 respectively.

It is also noted that the number of weight state transitions at each training position is affected by the number of hidden units. Training with a single strict layer of hidden units based on a simple network topology is much faster than that based on the more complicated topology. This can be seen from the data recorded for the total average weight state transitions in @1 and @2 in Table 6.2. These results are consistent with those of Samad (1988) for XOR using SBP and those of Plaut, Nowlan and Hinton (1986).

### 6.3.3 GENERALIZATION AND RESULTS ANALYSIS

As discussed in §4.3.2, generalization in FCBP is aimed at approximating continuous paths. The testing of generalisation ability involves assessing the tuples belonging to a performance set[1] defined in each task with a given tolerance after training on a finite training set[2]. It is then supposed that, given a small enough distance between consecutive test members, all members of the actual trained continuous sequential functions are close

---

[1] Each element of the set is an input tuple used for testing of associations, no matter if the tuple has been trained or not.

[2] Each element of the set has been trained and a suitable weight-state has been found for the I/O association.

enough to the target functions. In FCBP, a linear interpolation technique is applied to approximate a goal weight state using a discrete sequence of learnt weight states for the generalization. If the difference between the actual output and its target value for each input pattern to be performed is less than a given tolerance, this is a satisfactory generalisation. An absolute output tolerance for generalisation independent of any tested finite performance set is used to define a particular level of approximation. This is called generalization tolerance throughout this thesis. Suppose the tolerance is 0.2, this means that actual outputs within the range of ($\delta$-0.2, $\delta$+0.2) will be acceptable, where $\delta$ is the desired target output value.

For the 4-spirals problem, generalization is carried out by choosing a performance set based on the training set described in §6.3.2. That is, three extra positions were interposed between each pair of consecutive trained I/O positions for performance. So a total of 385 I/O positions, i.e. a total of 1540 I/O tuples, makes up the performance set for generalization. As for the output tolerance for training, the generalization tolerance is 0.2 here.

Two sets of generalization experiments have been made based on the same performance sets using two different network topologies.

In the first experiment, the generalization is tested on the 89 successfully trained goal weight state sequences based on the 2-2-1 network; the second is based on the results of the 100 successfully trained goal weight sequences based on the 2-5-5-5-1 network. The results are shown in Table 6.3 below.

Table 6.3 The generalization of 4-spirals problem

| index | network topology | No. of average successful generalisation tuples |
|---|---|---|
| @1 | 2-2-1 | 1540 |
| @2 | 2-5-5-5-1 | 1534 |

Here No. of trained tuples = 388; No. of performance tuples =1540; generalisation tolerance =0.2;

The data obtained show that the generalization accuracy is decreased for training on the more complicated topology. From the results in @1, there is no failure in the generalisation based on the 2-2-1 network. However, for each trial there are about 0.5% ( i.e. 6/(1540-388) ) untrained tuples which cannot be approximated within the tolerance 0.2 when the generalization is performed based on the 2-5-5-5-1 network. These results are consistent with the analyses on the relationship between the number of hidden units and the generalization accuracy made by Hinton (1989, 1991).

### 6.4 Comparison of FCBP and SBP

Besides showing the features of FCBP, one of the other reasons for choosing the two problems in the experiments is that the problem types are well known as benchmarks in the SBP approach This allows us to compare the FCBP results with those existing results for SBP in both training and generalization.

There are two kinds of comparisons throughout the set of experiments. One comparison is based on what will be called *general comparison*, where each experiment has its own suitable training parameters. The training speed and generalization accuracy of both the FCBP and SBP approaches can be compared based on each one's suitable network topology and the other parameters. The other comparison is called *strict comparison* in the sense of both training being carried out within the environment of having common features for the two approaches.

The first comparisons are made for the XOR problem. When the step size is 1.0, the CXOR problem is to train a single path with 4 training positions in FCBP, and this becomes the XOR problem in SBP. Two kinds of comparisons have been made, each involving experiments based on the two approaches respectively. Here two strict comparisons are made using the parameters and topology which are suited to SBP and FCBP respectively.

Another set of comparisons are made for the spirals problems. Because the 2-spirals problem is only a simple problem for FCBP — a single layer network without hidden units can solve it, the 4-spirals problem is used instead to show a problem in FCBP where hidden units are needed. The training and generalization results have been compared with those obtained in the 2-spirals problem. Results have also obtained from the two different problems using the two approaches: 2-spirals in SBP and 4-spirals in FCBP each based on two kinds of topology: one is in favour of FCBP, the other is in favour of SBP.

### 6.4.1 TRAINING

- **Training of the CXOR problem**

The method employed for the strict training comparison is to find the optimum results for each approach using two sets of parameters: for two different topologies, learning rates and momentum coefficients.

The version of SBP used is the same as that of conventional BP on feedforward networks described by McClelland and Rumelhart (Jones and Hoskins, 1987) except in the termination criteria which are altered to be the same as for FCBP-II at each training position (§4.4). The results are shown in Table 6.4 below.

Table 6.4 The comparison of training on CXOR problem

| index | topology | learning rate | momentum | regime | weight-transitions | training failures |
|---|---|---|---|---|---|---|
| @1 | 2-1 | 11.0 | 0.0 | FCBP-II | 4.96 | 0 |
| @2 | 2-2-1 | 0.9 | 0.9 | FCBP-II | 22.82 | 0 |
| @3 | 2-2-1 | 0.9 | 0.9 | SBP | 815 | 9 |
| @4 | 2-2-1 | 10.0 | 0.0 | FCBP-II | 4.80 | 0 |
| @5 | 2-2-1 | 10.0 | 0.0 | SBP | 852 | 23 |

Here training step =1.0; output tolerance =0.2; Maximum no. of weight transitions = 2000

The best set of training results on FCBP are shown in @1 and @4 for the two different topologies. @1 shows the result of FCBP without hidden units. This can only be used for a general comparison with the best result for SBP. The best set for SBP is shown in @3. The strict comparisons use a 2-2-1 structure with two sets of momentum coefficients and learning rates, each set's parameters are made to suit one of the approaches.

Note that FCBP training here is significantly faster than SBP. It can also be seen that training speed and training failure vary in their weight state transitions according to the learning parameter values. FCBP's training speed slows down as the learning rate is less suitable but training is still always successful. For SBP, the training speed slows down but also the training failures increase for less suitable learning parameters. This suggests that the FCBP training is more robust, at least for this problem, compared with SBP.

**• Training of the spirals problem**

The 4-spirals problem in FCBP is similar to the 2-spirals problem in SBP in the sense of hidden units being needed in both cases.

For the SBP approach the 4-spirals problem is just a similar but more complicated problem than the 2-spirals problem. More training tuples and more complex decision regions are involved in finding a single weight state in the 4-spirals problem. Thus it is very likely that more hidden units and a longer training time may be needed compared with the resources needed in solving the 2-spirals problem (refer to §3.2 for more discussions on the relationship between the number of hidden units and I/O training tuples). A effort has been made for solving the problem using SBP in order to see the training feasibility.

There is no precise formula for how to design a topology for sizeable problem (§3.2). Lang and Witbrock (1988) use a formula of 1 bit per I/O pair and 1.5 bits of information per link in designing their network for the 2-spirals problem. The network used here is designed as a fully connected feedforward network with two input units and one output

unit in the input and output layer respectively, three hidden layers with 5, 14 and 5 units at each layer. Together with the links from the bias units, there are total of 264 links in the network and 396 bits of information. This is sufficient to meet the requirement of a total of 97*4 i.e. 388 bits of information for training in the 4-spirals problem when there are 97 training tuples chosen along each spiral. An experiment with three trials has been made to train the 4-spirals problem on the designed network topology with learning rate 0.001 and momentum coefficient value 0.5 using the SBP approach. For each trial, the initial weight states were randomly chosen within the range of -0.1 and 0.1 as suggested by Lang and Witbrock.

All experimental results are shown in Table 6.5. From @1 to @4, a successful trial is the one which has found a goal weight state for each of the 97 I/O positions within 20,000 weight state transitions. For @5, a successful trial is one finding a goal weight state for the 388 (i.e. 97*4) I/O tuples within 50,000 epochs which is 19,400,000 weight state transitions.

Table 6.5 A comparison of training on 4-spiral and 2-spirals problems

| index | topology | problem | training failures | learning rate | momentum | regime | weight-transitions |
|---|---|---|---|---|---|---|---|
| @1 | 2-1 | 2spirals | 0/100 | 0.45 | 0.0 | FCBP-II | 714.46 |
| @2 | 2-5-5-5-1 | 2spirals | 0/3 | [0.001,0.002] | [0.5,0.95] | SBP | 3,666,600 |
| @3 | 2-2-1 | 4spirals | 11/100 | 1.0 | 0.9 | FCBP-II | 10,600 |
| @4 | 2-5-5-5-1 | 4spirals | 0/100 | 0.15 | 0.9 | FCBP-II | 22,916 |
| @5 | 2-5-14-5-1 | 4spirals | 3/3 | 0.001 | 0.5 | SBP | >19,400,000 |

Here output tolerance is 0.2 for FCBP and binary for SBP.

@1 and @2 show the results of training based on the FCBP and SBP for the 2-spirals problem. This tell us that in FCBP the 2-spirals problem can be solved without hidden units in an average of 714.46 total weight state transitions for each trial. This is compared with 3,666,600 total weight state transitions for the same number of I/O positions but half

the number of I/O tuples for the best result reported by Lang and Witbrock (1988) with the SBP approach.

Also, it is noted that FCBP needs less hidden units for training but more weight states to be stored comparing to the SBP approach. In FCBP the number of hidden units is dependent on the number of the independent training tuples at each position. This number is much less than the number required by the SBP approach for solving the problem. However, extra memory is required to store the learnt goal weight states sequence in FCBP. In a particular cases, this is dependent on the number of training positions and the size of the network topology. For the 2-spirals problem, FCBP uses 291 weight values (i.e. 97*3 links) whereas Lang and Witbrock uses 138 weight values (1 weight state * 138 links) against speeds of 714.46 for FCBP and 3,666,600 for SBP, this gives an example of the trade off between the memories and the training speed in FCBP.

As described in §6.3.2, in FCBP the 4-spirals problem can be solved using the 2-2-1 structure in an average of 10,600 total weight state transitions for each trial. This is also compared with the 3,666,600 weight state transitions reported for the 2-spirals problem using SBP. According to Lang and Witbrock's formula, FCBP uses 873 (i.e. 97*9 links) whereas SBP uses 264 weight values against speeds of 10,600 for FCBP and at least infeasible for SBP.

The 4-spirals problem is also trained in FCBP using the more complicated topology, employed by Lang and Witbrock's SBP approach for the 2-spirals problem to compare the training speeds. Results in @4 show that the network topology is significant for the FCBP training. The richer topology doubles the training time, this time is still significantly less though than that of SBP's training for the 2-spirals problem (in @2). Finally the data in @5 shows that no training has been successfully completed within 50,000 epochs for 4-spirals problem in SBP. This gives an indicator as to SBP's infeasibility for the 4-spirals problem.

### 6.4.2 GENERALIZATION

In FCBP, generalization involves interpolating weight states for the untrained tuples through the learnt weight state sequence. In SBP generalization, as a single learnt weight state is used for all the untrained tuples, it is then, in weight space, a constant interpolation, i.e. of order 0. The comparisons for generalization presented here are based again on the CXOR problem and the spiral problems.

**• Generalization of the CXOR problem**

Several experiments have been made for both showing generalization for the FCBP approach itself with various topologies and parameters and for comparisons based on both the SBP and FCBP approaches.

Both general and strict comparisons are made for generalization (§6.4). The training parameter equalities in strict comparison especially are intended to eliminate unwanted interference from training differences on generalization.

Within the FCBP approach, differences are examined by having the training accuracy $\alpha$ vary so as to reveal trends in the generalizations based on different learnt goal weight state sequences.

For comparisons made between the two approaches, generalization differences are explored based on learnt goal weight sequences obtained by having a suitable network topology and training parameters with different training accuracies. This is to see if the topology and other training parameters are a factor in the generalization results. FCBP and SBP have been compared for their approximation accuracy to the continuous target function described in Eq. (6.1) when the performance step generating both trained and untrained inputs is 0.25 using those goal weight sequences which obtained when the training step size is 1.0. The network topology is shown in Fig. 6.4. Three sets of training parameters with various training tolerances were used.

The results are shown in Table 6.6. The best set of the generalization results on FCBP are shown in @1, @2 and @3 with three different training tolerances. The data shows that the total error in generalization is significantly decreased when the training tolerance is reduced in FCBP. The associated network topology without hidden units is not able to learn a single goal weight state for CXOR in SBP. Hence no strict comparison may be made based on this topology.

The strict comparisons with SBP are shown from @4 to @9 using a network topology which is suitable for SBP. @4 to @6 show the results using a set of good training parameters (learning rate and momentum coefficient) for SBP, @7 to @9 show those using a set of the good parameters for FCBP. The results of the FCBP approach are significantly more accurate than SBP regardless of which set of training parameters are taken in @4 to @9.

Table 6.6 Comparison of generalization on CXOR

| index | network topology | learning rate | momentum value | training output tolerance | SBP average total error | FCBP average total error |
|---|---|---|---|---|---|---|
| @1 | 2-1 | 11.0 | 0.0 | 0.20 | - | 0.003 |
| @2 | 2-1 | 11.0 | 0.0 | 0.10 | - | 0.0015 |
| @3 | 2-1 | 11.0 | 0.0 | 0.05 | - | 0.0007 |
| @4 | 2-2-1 | 0.9 | 0.9 | 0.20 | 0.006 | 0.0046 |
| @5 | 2-2-1 | 0.9 | 0.9 | 0.10 | 0.005 | 0.0009 |
| @6 | 2-2-1 | 0.9 | 0.9 | 0.05 | 0.005 | 0.0004 |
| @7 | 2-2-1 | 10.0 | 0.0 | 0.20 | 0.016 | 0.0034 |
| @8 | 2-2-1 | 10.0 | 0.0 | 0.10 | 0.014 | 0.0004 |
| @9 | 2-2-1 | 10.0 | 0.0 | 0.05 | 0.014 | 0.0002 |

For each trial: No. of training tuples=4; No. of performance tuples=16;

As an aside, it was found that the final learnt weight states vary slightly in their closeness to the ideal goal weight state path according to the learning parameter values. However, the

major trend is shown in the table and is that FCBP's error per I/O performance tuple decreases rapidly with respect to that of SBP as the training tolerance is reduced. At the smallest training tolerances shown, the difference of both approaches in error of is 1(in @6) or 2 (in @9) orders in magnitude. This suggests that the loops of ideal weight variation between consecutive trained inputs for SBP are relatively significant for producing accurate output whereas the non-linear variation in the ideal weight states for FCBP is relatively insignificant (see Fig. 6.6 for two intuitive pictures of the ideal goal weight paths). Also the data in the two columns showing the total average errors suggest that the training parameters for SBP are relatively significant for producing an accurate generalization whereas the parameters for FCBP are less relatively significant. This is shown in the following two graphs in the Fig. 6.7a and Fig. 6.7b.

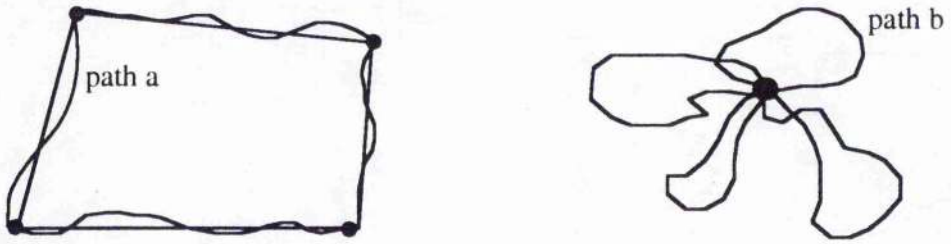



Fig. 6.6 An intuition picture of the ideal goal weight path for FCBP (path a) and SBP (path b). The filled circles represent the learnt weight states. The polygon shows the weight path produced by linear interpolation for FCBP.

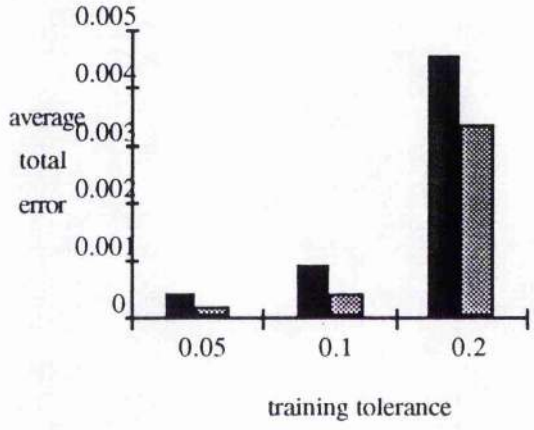



Fig. 6.7a The differences on FCBP

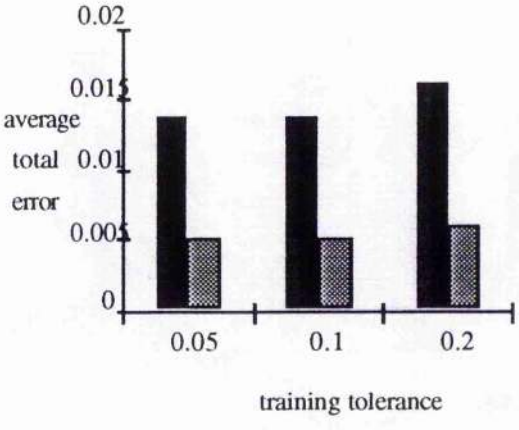



Fig. 6.7b The differences on SBP

In the above pictures, the black filled bars represent the errors when the less suitable training parameters are used and the grey bars represent the errors with suitable parameters.

**• Generalization of the spirals problem**

The comparisons made show the generalization capability and accuracy of FCBP when the associated network has hidden units.

The 2-spirals problem is already a complicated problem for SBP, and considerable effort is needed in order to design a suitable network and to implement the training for the 4-spirals problem in SBP approach. A limited attempt at the 4-spirals problem with SBP training has been made without success, therefore no strict comparison can be made with SBP. However, the 4-spirals problem in FCBP may show a significantly better generalization compared with that of SBP for the simpler version of a spiral problem — the 2-spirals problem. This comparison is still a useful indicator of FCBP's relative power.

Table 6.7 Comparison of generalization on the spirals problem

| index | net topology | learn. rate momentum | generalizatio n tolerance | problem | regime | no. incorrect perform. tuples |
|---|---|---|---|---|---|---|
| @1 | 2-1 | 0.45/0.0 | 0.2 | 2-spirals | FCBP-II | 0 |
| @2 | 2-2-1 | 1.0/ 0.9 | 0.2 | 4-spirals | FCBP-II | 0 |
| @3 | 2-5-5-5-1 | 0.15/0.9 | binary | 2-spirals | SBP | 56 |
| @4 | 2-5-5-5-1 | 0.15/0.9 | binary | 4-spirals | FCBP-II | 3 |

Here No. of performance tuples is 770 for 2-spirals problem and 1540 for 4-spirals problem.

The experiments have been made based on three network structures. The first two structures ( in @1, @2) are satisfactory for carrying out the 2-spirals and 4-spirals problems respectively in FCBP but are insufficient for successful training in SBP. The other structure is that chosen by Lang and Witbrock for the 2-spirals problem in SBP approach. Although this is an over-rich structure for the 4-spirals problem using FCBP, it is also used for the 4-spirals problem in FCBP to aid comparison by removing differences

due to topology. This is because the differences may be explored by having different number of hidden units that reveal trends of the generalization in FCBP.

Three extra positions are interposed between each pair of consecutive trained I/O positions for generalisation. The total of 1540 I/O tuples are used as the performance tuples for testing 4-spirals generalization. The results presented by Lang and Witbrock in their paper are used here for comparison with FCBP. Lang and Witbrock also interpose three extra positions between each pair of consecutive trained I/O positions for performance. This gives total of 770 I/O tuples for testing 2-spirals generalization.

From the results in @1 and @2, it can be seen that there is no failure in generalization with a tolerance of 0.2 over the performance set for the 2-spirals or 4-spirals problems in FCBP. This can be compared with the 56 failures reported in @3 for the 2-spirals problem in SBP approach with binary classification as the generalization criteria. As binary classification is less accurate than using 0.2 as the generalisation tolerance, the number of the failures in SBP is very likely increased if the criteria is 0.2.

The result of the 4-spirals problem in FCBP is shown in @4, for all the successful trained trials, 3 out of a total of 1540 performance tuples are misclassified in FCBP according to binary interpretation when the over-rich topology 2-5-5-5-1 is used. This is compared with 56 binary misclassifications out of 770 performance tuples for the best results reported by Lang and Witbrock for the 2-spirals problem in SBP. Again, the results show the significant improvement of generalization accuracy by FCBP.

### 6.5 FCBP and single goal approach

Some basic experiments to explore the relationship between SBP and a hybrid of FCBP with the single goal weight state approach (FCBP-SBP) are presented here.

### 6.5.1 TASK DESCRIPTION

The idea is to see if the stepping stones technique used in FCBP (refer to §6.2.3) can also speed up the training in finding a single weight state. A number of extra training tuples are used to help in finding a single weight state for those recognition tuples. As defined in §6.2.3, the extra training tuples are employed in the training process but are not used in testing the goal weight state for stopping training.

A feedforward neural network has been used by solving the CXOR problem to explore the training of finding a single weight state using the stepping stones method. The task is: training on 32 training tuples chosen along the edges of the unit square but where the single goal weight state applies only to 4 of the tuples, those consisting of the XOR problem, out of the 32 tuples. The 28 extra tuples in the 32 training tuples function as stepping stones. Each of them is one of 7 training tuples between any two recognition tuples.

The aim is to see if these stepping stones can be used to speed up the training for finding a single goal weight state by smoothing the weight transitions and using a regime similar to FCBP-I in making one weight change per I/O position, but which continues until a single goal weight state is found or failure occurs.

### 6.5.2 TRAINING

In neural terms, this task requires a network having two input units, some number of hidden units, and one output unit.

The simple network structure used for this set of experiments is shown in Fig. 6.8. The experiments are carried out with two training approaches. One is the SBP approach; the other is the modified single goal weight state approach described in §6.5.1, which is denoted as FCBP-SBP here.

The FCBP-SBP regime is a combination of the FCBP-I regime and SBP. At each training position, a single weight state transition is made for all inputs at the position using the error-weight gradient, learning rate and momentum coefficient as in conventional BP using the batch method (§1.4). A test for the single goal weight state is made each time a complete traversal of the all training positions occurs. Training continues until the testing for the goal weight state is successful or failure occurs. The success implies that the weight state has errors below a fixed universal tolerance for all training tuples. A single goal weight state is then found.

For the experiment on the stepping stones method described in §6.5.1 using the FCBP-SBP approach, the training set consists of 32 training tuples along the edges of the unit square with a fixed step size 0.125 and has the output associations based on the same I/O relationship described in Eq. (6.1) with training tolerance 0.2. Both SBP and FCBP-SBP use a learning rate and momentum coefficient pair over 100 trials where the initial weight state were chosen randomly within the range -1 and +1. The learning rate and momentum coefficient values were found again within a suggested range of [0,10] in the way described in §6.2.3. For the two approaches, the average number of weight state transitions taken for a successful trial was computed where a goal weight state has to be found within 10,000 weight state transitions.

### 6.5.3 RESULTS AND ANALYSIS

The two sets of training results based on the FCBP-SBP and SBP approaches are shown in Table 6.8.

Table 6.8 Training on CXOR using FCBP-SBP and SBP

| index | training step size | recog. step size | regime | learning rate | momentum value | weight transitions | successful trials |
|---|---|---|---|---|---|---|---|
| @1 | 1.0 | 1.0 | SBP | 10.0 | 0.0 | 377.84 | 100 |
| @2 | 0.125 | 1.0 | FCBP-SBP | 0.9 | 0.5 | 4192.32 | 100 |

Here training output tolerance =0.2; No. training trials= 100; topology 2-5-1.

Both results are collected for the best set of training parameters for the approaches respectively.

The experimental result shows that the single weight state can be found through the FCBP-SBP approach. It also shows that this approach may be much slower compared with that of the SBP approach using the same network. This implies that the stepping stones used in FCBP may not be helpful for finding a single weight state. A possible explanation follows:

If we consider the weight state set for each I/O training tuple, the size of the weight state set depends on both the network architecture and the error tolerance in recognition. Finding a single weight state as the goal of training is to find a single weight state within the common goal weight state set intersecting all the goal weight state sets associated with each recognition I/O tuple. The common weight state set is decided by the recognition tuples, the network topology and the error tolerance, and it is independent of the existence of the sets associated with the stepping stones.

During training, the extra stepping stone tuples may prolong the training because the extra tuples may drive the current weight transition away from goal weight states located within the common goal weight state set. An intuitive picture is given as follows in Fig.6.9:

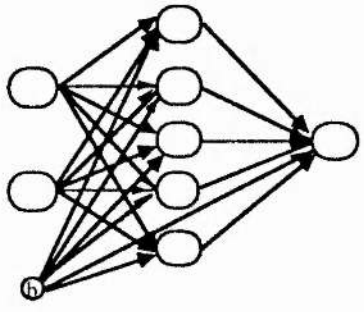


Fig. 6.8 The 2-5-1 network

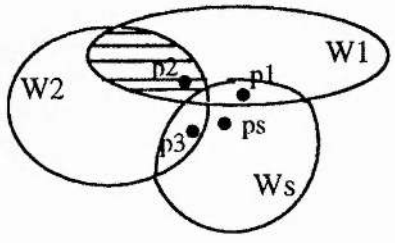



Fig. 6.9 The picture of weight state sets

Suppose $W_1$, $W_2$ and $W_s$ correspond to the three sets of goal weight states associated with the two training positions and one stepping stone position respectively; $p_i$ denotes a goal weight state within a weight state set. The shaded part in Fig. 6.9 is the common weight

state set of the sets $W_1$ and $W_2$. Without the stepping stone method, the weight transitions in weight space may form a route of $p_1$ toward $p_2$. As $p_2$ is within the common weight state set, the training has succeeded. With the stepping stone method, the weight transitions may form another route, which is from $p_1$ to $p_s$ and then on to $p_3$ instead of $p_2$. As $p_3$ is not in the common set, further transitions to the common set are needed before the training can stop. This effect appears to have outweighed the extra smoothness of weight state transitions inherent in the stepping stone methodology.

Conclusion: Adding a number of intermediate training tuples may well make the finding of a single goal weight state more difficult.

## 6.6 RCBP training and generalization

In this section, an example is presented to illustrate the features of the Recurrent Continuous Back-Propagation model (RCBP). It will be shown that problems which cannot be solved by FCBP are amenable to solution through this path-based dynamic model.

### 6.6.1 TASK DESCRIPTION

The task here is to correctly recognize inputs from particular positions around each of 4 elliptical orbits in an I/O space.

All the orbits are of the same shape but different in their direction or orientation (see Fig. 6.10). Each orbit is a path in I/O space which has a projection corresponding to its input path within the input square in 2-D dimensions. Each orbit's input path starts from a different position and moves in either a clockwise or anti-clockwise direction. All the orbits' I/O paths lie on a plane with a unique orientation and gradient.

In more detail, the input paths all have the form of an ellipse with centre (0.5,0.5) and major and minor semi-axes of 0.5 and 0.25. This can be described as follows:

Let $\alpha$ be the polar angle of an input value from the centre of an orbit's input path; **r** the polar radius of an input value from the centre of an orbit's input path :

$$\mathbf{r} = (\mathbf{a}*\mathbf{b}) / \sqrt{(b\cos\alpha)^2+(a\sin\alpha)^2} \tag{6.2}$$

where **a** and **b** are the semi-axes of the ellipse. Then each orbit can be defined by the two variables **x** and **y** in Cartesian coordinates:

$$\mathbf{x}=0.5+\mathbf{r}\cos\alpha \text{ and } \mathbf{y}=0.5+\mathbf{r}\sin\alpha \tag{6.3}$$

The relationships of the orbits' input paths and output paths $\mathbf{z_i}$ (i=1,2,3,4) can be defined as four continuous functions by Eq. (6.4a) to Eq. (6.4d):

$$\mathbf{z_1} = 0.2*(1-\mathbf{x})+0.6 \tag{6.4a}$$

$$\mathbf{z_2} = 0.2*\mathbf{y}+0.6 \tag{6.4b}$$

$$\mathbf{z_3} = 0.2*\mathbf{x}+0.6 \tag{6.4c}$$

$$\mathbf{z_4} = 0.2*(1-\mathbf{y})+0.6 \tag{6.4d}$$

for the four orbits respectively. We name this task an *intersecting orbits problem.*

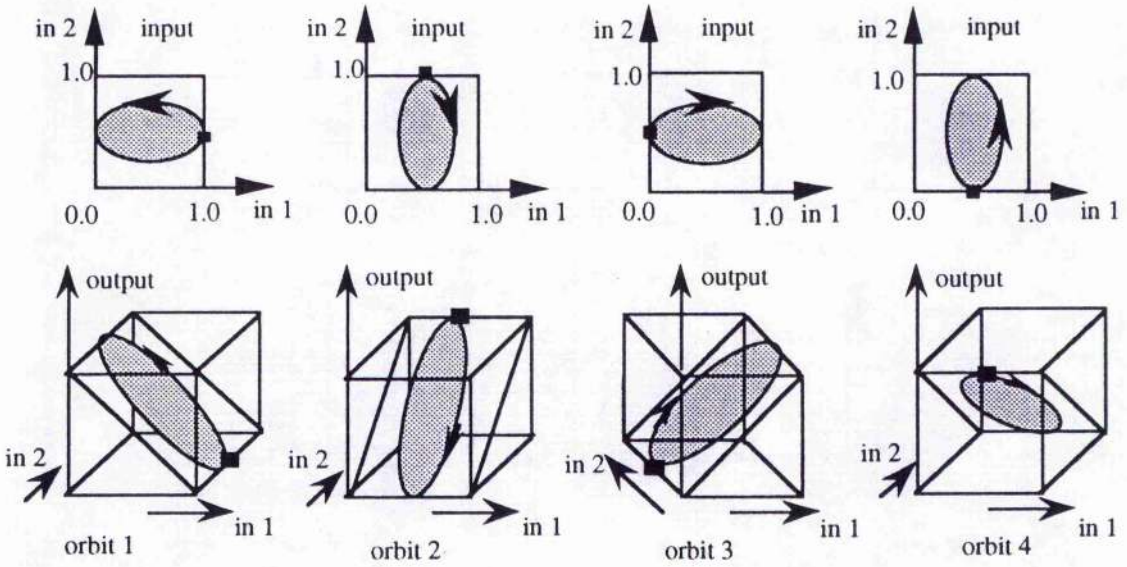



Fig. 6.10 The I/O relationship of the intersecting orbits problem

The motivation in selection of the orbits problem is: it is difficult enough to be interesting because hidden units and internal activity states are involved and hence shows the power and capability of RCBP.

The intersecting orbits problem cannot be solved by employing FCBP because some one-many associations are required at some training positions along the paths — different output values are required at the orbits' input path intersections (such as when x=y in input space). One-many I/O associations in themselves do not require a recurrent network for PBP as they do for SBP. It is the fact that the orbits intersect at some training positions and have different desired output values.

### 6.6.2 TRAINING AND RESULTS ANALYSIS

Training based on I/O tuples chosen from the 4 orbits are carried out. This serves as an introduction to see how the RCBP works.

In neural terms, the intersecting orbits problem requires the network to have two input units, some number of hidden units, one output unit and some connections with recurrent links. A simple recurrent network has been chosen to meet this requirement and shown in Fig. 6.11.

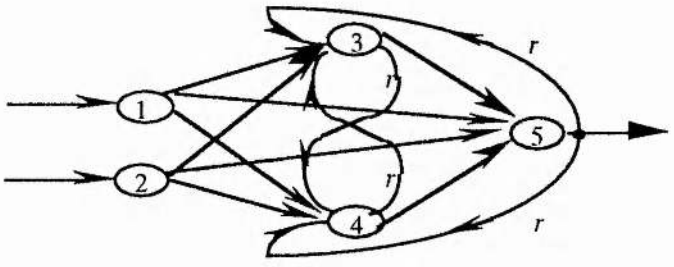



Fig. 6.11 The 2-2-1 recurrent network

The training was carried out over 30 trials where the initial weight states were chosen randomly within the range [-1,1]. The learning rate and momentum coefficient values were

explored within the range of [0,1] in the way described in §6.2.3. The average number of weight state transitions taken for a successful trial was computed where a goal weight state was found within 40,000 weight state transitions at each training position.

The input tuples in training set are chosen along the edges of the ellipses with a constant step size for the phase $\phi$ of the orbital period. That is, the set of input training tuples are chosen along each of the ellipse edges with step size $\phi = \pi/40$, which entails 80 training positions along each ellipse. The corresponding output values **z** were derived from the four I/O relationships described in Eq. (6.4). The position number starts from 1 and is increased along the direction described in Fig. 6.10 for each of the paths. The training results are shown below.

Table 6.9 Training results of 4-orbits problem

| index | net topology | $\theta$ step | learning rate | mom. | successful trials | max_wt transitions | ave. wt transitions | ave. wt trans on first |
|---|---|---|---|---|---|---|---|---|
| @1 | 2-2-1r* | $\pi/40$ | 0.7 | 0.9 | 25 | 40,000 | 11853 | 642 |

Here No. of the training trials = 30; output tolerance=0.1 * see Fig. 6.11

From the data, two important conclusions emerge. The first is that the RCBP approach is viable. Secondly, the training speed is acceptable though the control through continuity may be not as good as that in FCBP.

At the first training position, although training speed is slower compared to that of the most of the subsequent positions, the continuity feature of finding weight states along the training paths is not as good as that in the FCBP approach. In some training positions, training takes even longer than training at the first training position. This may be explained as that the weight state is now indexed not only by the underlying continuum of the I/O training tuple values but also by the activity values of the network. The latter may show sizeable changes in transition despite the attempts at control through continuity which are inherent in the design of RCBP. However, the RCBP approach still makes significant use of the continuity feature in training to achieve the goal weight sequence. When the input

training data are chosen with underlying continuity, the average training speed over all training positions is still much faster than that at the first training position. This can be seen through the number of the average weight transitions needed at each position. The number is 642 at the first training position and is about 142 at each of the other training positions. The inherent feature of closeness between consecutive weight states associated with consecutive training positions plays an important role in RCBP training.

A final aside is that it is more difficult to find a set of suitable training parameters for satisfactory training in RCBP compared with that in FCBP. Hence the error-weight surface may be relatively complex for this problem.

### 6.6.3 GENERALIZATION RESULTS AND ANALYSIS

As presented in §5.4, generalization in RCBP is in principle very similar to the generalization in FCBP. It is aimed at approximating continuous functions within a given tolerance by training on a finite number of tuples chosen along the functions. All the interpolation techniques applied in FCBP for weight states can also be applied here.

In this RCBP experiment, a linear interpolation technique is applied to the learnt goal weight state and activity state sequences to carry out the generalization. If the difference between the actual output and its target value for each input pattern in performance is less than a given tolerance, this is taken to be a satisfactory generalisation.

For the 4-orbits problem, generalization is carried out by choosing a performance set based on the same rule for choosing a training set. That is, three extra positions were interposed between each pair of consecutive trained I/O positions for generalisation. So for each trial, a total of 317 I/O positions or total of 1268 I/O tuples makes up the set for generalization. The same output tolerance is used as for training, i.e. the generalization tolerance is 0.1 here.

The generalization is carried out based on the 25 successfully trained weight states and activity state paths based on the 2-2-1 recurrent network. The results are shown in Table 6.10 below.

Table 6.10 The generalization of 4-orbits problem

| index | no. of performance tuples per trial | no. of successful generalization tuples per trial | failure rate of untrained tuples |
|---|---|---|---|
| @1 | 1268 | 1261 | 0.74% |

Here generalization tolerance = 0.1; No. of trained trials=25;

The above data shows that the generalization is good. For 25 successfully trained trials, for each trial, there are 4 paths; each path has 317 generalization test tuples (80 trained tuples and 237 untrained tuples); the average number of incorrectly performing tuples is only 0.52, 0.6, 1.7 and 3.3 for the four paths respectively. The average failure rate of the untrained tuples is about 7/948 for each training trial.

## 6.7 Electrocardiogram (ECG) addressable memory

In this section, a simple example is chosen to view the features of PBP in dealing with real world problems.

### 6.7.1 TASK DESCRIPTION

The task here is to associate each of 4 subjects' analogue ECGs with a particular different constant output signal through a discrete approximation of the ECGs. The length of the ECGs was taken to be one complete period of the subject (DYH) with the lowest frequency. The graphs of those ECGs are shown in Fig.6.12, the chosen data are shown in Apdx 3.

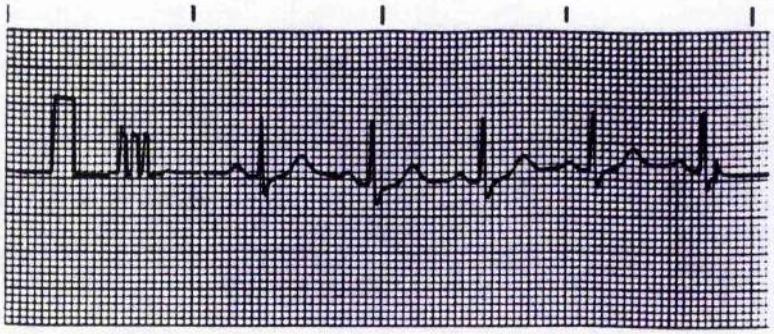


Fig.6.12a The ECG of the subject (MKW)

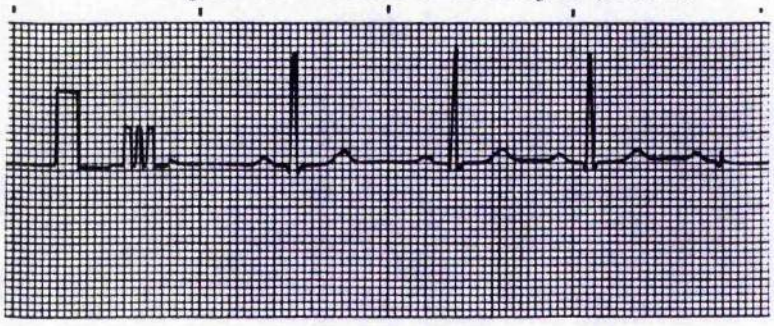


Fig.6.12b The ECG of the subject (LHC)

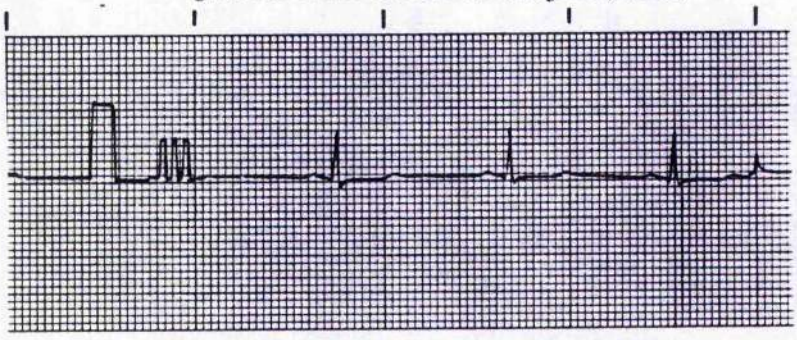


Fig.6.12b The ECG of the subject (VW)

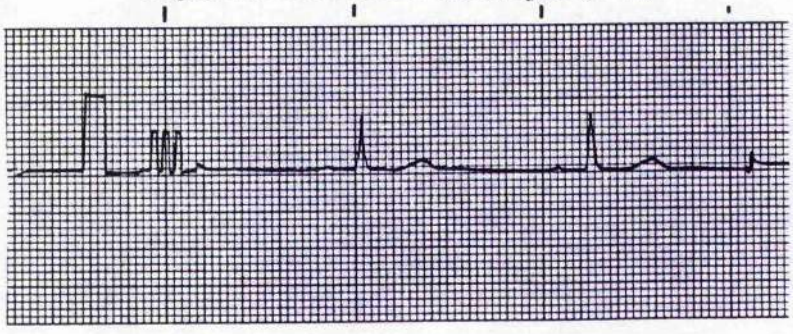


Fig.6.12b The ECG of the subject (DYH)

There are four 1-D input training paths, where each path consists of a subject's ECG and four 1-D output paths, one for each input path, where each output path consists of a signal at a different constant analogue value. In more detail, the output paths $\mathbf{z_i}$ (i=1,2,3,4) can be defined as four constant analogue values, which are $\mathbf{z_1}$ = 0.2, $\mathbf{z_2}$ = 0.4, $\mathbf{z_3}$ = 0.6, $\mathbf{z_4}$=0.8 for the four ECGs numbered 1,2,3 and 4 respectively in Fig.6.12. We name this task the *ECG addressable memory problem.*

The motivation in selection of this problem is that as a real world problem it is small but difficult enough to be interesting because hidden units and real analogue data are involved.

### 6.7.2 TRAINING AND RESULTS ANALYSIS

Experiments have been carried out on four sets of input training tuples which are all chosen along the paths of the ECGs either with or without normalisation of the raw real world data. The input tuples in the training set are chosen along the paths of the ECGs with each path value at a position corresponding to a common moment in time and with various time steps between positions. The latter variation occurs because the set of input training tuples are chosen along each ECG to ensure that all the peaks and troughs of each ECG are chosen. This entailed 27 training positions where each position contains at least one peak or trough value. The position number is increased in the direction of increasing elapsed time for each ECG path. The associated output values $\mathbf{z}$ were assigned based on the I/O relationship described above.

In neural terms, the ECG problem requires a network to have one input unit, some number of hidden units and one output unit. According to the raw data, there are no identical inputs with different outputs involved in the ECG problem. In principle then, a feedforward network and the FCBP approach should be able to be used for solving the problem.

Each of the four experiments was carried out over 30 trials where the initial weight states were chosen randomly within either the range [-0.01,0.01] or [-0.1,0.1]. The learning rate

and momentum coefficient values were explored within the range of [0,1] in the way described in §6.2.3 using a strictly layered feedforward network with 3 hidden units. The average number of weight state transitions taken for a successful trial was computed where a goal weight state was found within a given number of weight state transitions at each training position.

Training has been attempted using FCBP with both the raw data and three other sets of data which are different normalisations of the raw data. The training results on the raw data are shown below.

Table 6.11 Training results of the ECG problem on raw data

| net topology | range of initial weights | learning rate | mom. | successful trials | max_wt transitions | no. successful trials at the first position |
|---|---|---|---|---|---|---|
| 1-3-1 | [-0.1, 0.1] | 0.7 | 0.9 | 0 | 120,000 | 30 |

Here no. of the training trials = 30; output tolerance=0.1.

The result shows that although the FCBP approach should be able to train the ECG problem in principle, all attempts at training failed. Some important conclusions need to be made. Specifically, there are three major aspects to be noted here which can undermine the training feasibility of FCBP.

Firstly, large jumps should be avoided in input values between consecutive training positions within an input training path. If there are sizeable irregular changes in consecutive input training values, the FCBP approach cannot makes significant use of the continuity feature underlying the design of FCBP to achieve the goal weight sequence. Training may be undermined because the consecutive goal weight states are far apart.

Secondly, closeness should be avoided between input values among training paths within the same training position. In principle, RCBP is only needed in preference to FCBP in dealing with true one-many associations where many identical input values are associated with different output values at a training position. In practice, when inputs are very close and the associated outputs are significantly different, the goal weight states are so extreme

that training is infeasible. Hence a recurrent network may be required instead to generate the I/O associations more easily.

Finally, the goal weight states should not be too large. This is because if they are, even a small input change between positions may lead to large excitation and output changes under the final goal weight state for the previous position. If the desired output at the new position does not match these changes, it may be very different from the actual output which implies that large weight changes are needed, thus making training difficult.

By having the input values regularly distributed throughout a range of $[-r,+r]$ for $r>>0$, this may prevent goal weight states having large weight values. This is because the goal weight states can be put reasonably close to the origin in the weight space. The training feasibility is increased as a consequence. Fig.6.13a and b for example show two different goal weight states in 2-D weight space. In Fig.6.13a, the lines $l_1$ and $l_2$ are associated with the I/O tuples $I_1/O_1$ and $I_2/O_2$ respectively; in Fig.6.13b, $l_1$ and $l_2$ are associated with $I_1/O_1$ and $-I_2/O_2$. The pair of $l_2$ lines have gradients which are the negative of each other, while the pair of $l_1$ lines are the same as each other. It can be seen that the goal weight state of the second mapping is closer to the origin compared with that in the first one.

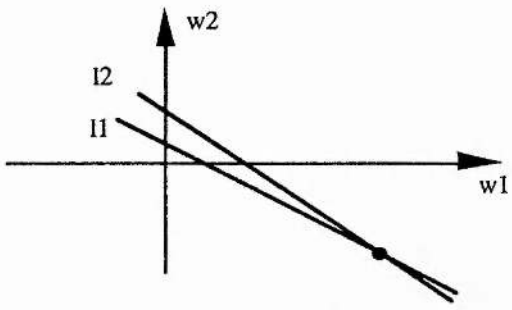



Fig.6.13a Goal weight state with large values

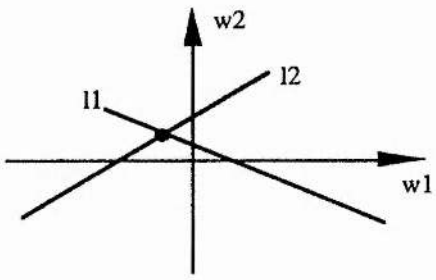



Fig.6.13b Goal weight state close to origin

The first and third aspects mentioned above were focussed on first in the present problem to see if FCBP could be made to work with normalised data. The first aspect is significant here since taking the peaks and troughs as the primary sampling determinants leads to weak continuity in the sample data. The third aspect is seen to be significant from inspection of the weight states successfully trained in the first attempt described above.

Table 6.12 Training results of ECG using log(x)

| net topology | learning rate | mom. | successful trials | max_wt transitions | ave. weight transitions | ave. wt transitions on the first position |
|---|---|---|---|---|---|---|
| 1-3-1 | 0.9 | 0.9 | 21 | 120,000 | 83,417 | 25,449 |

Here no. of the training trials = 30; output tolerance=0.1; initial weight range: [-0.01,0.01]

Table 6.12 shows the results of an initial attempt at smoothing the raw data to attend to the first aspect mentioned above by applying the logarithmic operator to each raw input training data $x$.

Although the failure rate is 30%, this experiment shows that training feasibility is improved compared with using raw data. Inspection of the failures showed that the failure is not now caused by the irregular variations in inputs but large values in goal weight states. Compared with finding a goal weight state containing smaller values, training is more sensitive to learning rate and momentum values and becomes infeasible when relatively significant weight changes are required as suggested above.

Table 6.13 shows the results after further normalisation of the data according to avoid the third aspect discussed above. This experiment was carried out by changing each input training data $x$ into $a_i * \log(x)+b_i$ where $a_i$ and $b_i$ are chosen values (see Table 6.14).

Table 6.13 Training results of ECG on normalisation

| index | net topology | learning rate | mom. | successful trials | max_wt transitions | ave. weight transitions | ave. wt trans. on the first position |
|---|---|---|---|---|---|---|---|
| @1 | 1-3-1 | 0.9 | 0.9 | 28 | 40,000 | 17,646 | 769 |
| @2 | 1-3-1 | 0.25 | 0.9 | 30 | 40,000 | 3,686 | 2772 |

Here no. of the training trials = 30; output tolerance=0.1; initial weight range: [-0.1,0.1]

Table 6.14 Chosen values for two normalisations

| | first normalisation (@1) | | second normalisation (@2) | |
|---|---|---|---|---|
| no. path | $a_i$ | $b_i$ | $a_i$ | $b_i$ |
| path1 | -1 | 5 | -0.5 | 3.5 |
| path2 | 1 | -3 | 0.5 | -2 |
| path3 | -1 | 4 | -0.5 | 3 |
| path4 | 1 | -2 | 0.5 | -1.5 |

$b_i$ is chosen for both distinguishing the four input paths more strongly and making the input training data large enough to have smaller weight values in goal weight states. $a_i$ is chosen to regularly distribute the input values throughout a range [-$r$, $r$] in the first case (@1) and to also further smooth the data in the second case (@2). The results show that the better normalisation further substantially improves the training feasibility. The high success rates meant that a recurrent approach was not needed after all.

Conclusion: In dealing with real world problems, normalisation of data is important for PBP as it is for SBP. All normalisation techniques used by conventional BP should be taken into consideration in case they can help with the three major aspects mentioned above for PBP.

## 6.8 Conclusions

Several experiments have been done to show:

(1) The features of FCBP are exemplified: the role of the hidden units; the stepping stones feature; the simpler network structure needed in solving problems; the trade-offs for training speed and memories. These features and the results show that FCBP offers the possibility of extending SBP's feasibility through the use of continuous temporal structure to control the network size and improve training and generalisation.

(2) Comparison of FCBP with SBP: the experiments show significant differences in training speed and generalization accuracy between the two approaches in solving the same problems. It is also noticed that the values of the training parameters in the FCBP approach are not as sensitive as those in SBP regarding the speed and accuracy of training and generalization.

(3) The search for a single goal weight state by using extra intermediate I/O tuples. The goal achievable but training may not be faster than that using SBP.

(4) The features of RCBP are exemplified: the need for recurrent links; the role of the internal states; the training speed; the generalization capability. These features and the results show that RCBP extends the capability of the FCBP approach.

(5) The capability of PBP in solving real world problems is tested. It is realised that normalisation of data is important for PBP as it is for SBP.

# CHAPTER 7

# CBPTOOL DESIGN CONSIDERATION & OVERVIEW

## 7.1 Introduction

A simulator called **cbptool** is designed and implemented in the thesis. **cbptool**, supported by a graphic user interface, is a neural network tool written in 'C' that supports path-based back-propagation (PBP) and the conventional back-propagation learning models.

The **cbptool** provides a user-friendly tool for investigating the back-propagation of neural networks, especially the new learning models of PBP. Useful facilities for PBP research and development are built into an easy-to-use environment that takes advantage of the *SunView*[1] graphical interface. All user interfaces of the tool are, wherever and whenever possible, mouse and menu driven in the window environment. The graphical interface enables a user to design a neural network interactively on screen and helps the user to further study the PBP models and observe the dynamic behaviour of the networks. It can also be used to investigate both SBP and PBP-based applications and carry out associated researches.

The **cbptool** works on all *Sun* (from series 3 up) workstations running 'C' under *Unix* and using the *SunView* window management environment.

---

[1] If the user wishes to know more about the SunView environment, please refers to the *SunView System Programmers Guide* and the *SunView Programmers Guide.*

This chapter is a guide to how the design of **cbptool** has been evolved from conception and requirements for implementation. A detailed description concerning about the functions, designing, internal representations and user interfaces of **cbptool** is presented in this chapter. In §7.2, the design purpose, design features, and the chosen environment are reviewed. In §7.3, the user interface is outlined. In §7.4, the major internal representations and implementation of the tool is presented. In §7.5, the major simulator parts are introduced; In §7.6 the discussion of some design issues is summarised. Finally in §7.7 the conclusion is presented.

## 7.2 Design Considerations

### 7.2.1 DESIGN PURPOSES

The fundamental aim of the tool is to provide an environment in which a user, who may not necessarily have any knowledge of either computer programming or the internal principles involved in neural networks, may design and simulate the actions of a neural network in performing learning and generalization. Such kind of neural network operations are supported by FCBP, RCBP and conventional BP learning models.

The tool is not intended, however, merely for beginners but should be sufficiently flexible as a research tool by users who have sufficient knowledge of neural networks. It is hoped that **cbptool** can be used as a handy tool for users to further explore the new learning framework PBP, to develop some interesting applications; or as a tool to teach the basic concepts of neural networks.

### 7.2.2 DESIGN FEATURES

The following features will be pursued throughout the design of the tool:

• **Graphic interactive interfaces**

A simulator is easier to use if graphical images and interfaces are provided. In the meantime, it is important to keep the CPU cost as low as possible if a graphical interface is involved. It was thus decided that the tool should contain only as many graphical interfaces as necessary for providing both a good graphic tool and a relatively low CPU cost. More concrete points in **cbptool** are as follows:

(1) mouse and menu driven

The tool will be, wherever and whenever a user interface is needed, mouse and menu driven. This allows a great degree of easiness and user-friendliness. This will help users to carry out user-machine conversations easily and set up all necessary parameters for any operation.

(2) network scrolling on the screen

The user will be able to design a network larger than the screen or window in which he is working. To this end the design window will be a mobile virtual mapping onto a world coordinate system. With the mouse and menus, the user can not only design a network topology on screen but also save a network or load in a previously created network into the tool.

(3) graphical output display

Users can display output data graphically. What output results are displayed and in what form will be user-definable at a certain stage. This allows for maximum flexibility. The following two main points are relevant for users to represent or interpolate results:

• The tool depicts the data which would provide the user with a good overall representation of the most important dynamic behaviour during the operations of the network.

• The tool records other less important but relevant data produced by the tool operations into some working result files in a textual form. This enables the user to have possible access afterwards.

• **A Backpropagation environment**

The training and generalisation regimes used to exemplify the path based approach all involve variants of BP. In order to provide the necessary completeness, flexibility and richness for BP, it was decided that only the PBP and conventional BP models are implemented in the tool. Training and generalisation facilities are provided for feedforward and recurrent networks.

• **Flexibility and simplicity**

Most functions and parameters used by the operations of the tool can be chosen from a menu which is either designer defined or entirely user defined. The user can provide his own code fragments which can be linked into the main program, which allows maximum flexibility.

The initial weight values associated with links can be set using mouse, menus or through a file interface. Some of the values are observable dynamically. The user can communicate with the tool in a relatively simple way.

• **Choice of running ground**

To separate the parameter setting from the performing environment will be one of the features which makes a more powerful simulator in order to fulfil different requirements by different users.

The simulator operation can be run with or without the interactive window environment after the required parameters have been set up. This implies that when the user has

completed his own network topology design, set up all required initial values and chosen his operation, the ground of running the operation can be selected as well. It enables the operations to be run in the background without occupying a window system if desired. This feature enables the user to save CPU expenses on graphics especially when the user wants to apply the simulator to an application or some calculation intensive research which does not require any graphic interfaces. It also enables the user to run several **cbptool** processes at the same time on one machine, in other words, to save computer resources in general.

### 7.2.3 CHOICE OF THE DESIGN ENVIRONMENT

All *Sun* workstations from series 3 up have both sufficient processing power and memory available to cope with this simulator. Also the *Sun* workstations provide the *SunView* window environment which is compatible with 'C' language.

The *SunView* is a good design environment for supporting interactive, graphics-based applications running within windows; it is also relatively easy to be understood and used due to its powerful and high-level functions.

Another reason for choosing *SunView* rather than other environments was that *SunView* is a well-implemented product and the necessary documentation of it was readily accessible. Recently, newer more portable software has been developed such as *X-windows*. It is entirely feasible for future development to change the environment if desired.

## 7.3 Interface Outline

The user interface is one of the most important features of any application. No matter how outstanding the underlying code, if the user interface were not well designed, the user would suffer, and the application would be non-productive and even be worthless in the worst case. This means if the application designer could not distance the user from the

complexity of the underlying code, and not provide a user-friendly environment, the application would be lifeless. Considerable effort has been made in achieving user-friendly interfaces. Four major types of interface have been provided by **cbptool** and they are described one by one below.

### 7.3.1 WINDOW, MENU AND MOUSE INTERFACES

(1) Design Canvas:

This is the main window of the tool. Within this window, the user can design the actual topology of the network.

The canvas is implemented as a virtual *SunView* canvas. As the design canvas is only a viewing window onto a larger virtual canvas, it is necessary to include two scrollbars: one horizontal and one vertical to allow the user access the whole of the canvas.

A balance between the canvas dimensions and the memory allocation has to be considered in the design of the interface. If the canvas is too small, the dimension of neural networks is limited; if the canvas is too large, the memory allocation for the canvas would be wasted. Currently, the user is limited to a design canvas of 20 by 20 elements i.e. a maximum of 400 neurons. This number can be altered easily.

(2) Menu Bar

This window is implemented as a *SunView* panel, which allows the user to choose from a number of menu options. Each menu option deals with a different area of the tool.

(3) Mouse Panel

The main function of this interface is to display some text messages detailing the function of each mouse button. The text messages are updated each time according to the current choice of tools.

(4) Popup windows

Popup window is a common feature of the tool. It is mostly used as an interface to access files (described below). For example, the user can specify the directory and filename of a file through this interface. This kind of interface also allows the user to select or tailor some operation parameters to suit various needs.

For more details of the above four windows please refer to §Apdx 1.3.

(5) Output display windows

There are three output windows designed for display, representation and graphical interpolation of the data produced by the **training** or **performance** operation currently underway.

The concern here is to provide as much useful data as possible without overburdening the user with superfluous information nor taking too much computing time. This implies:

- Depicting the data in a good manner. The user should find it easy to extract any meaningful information and see the dynamic changes as quickly and clearly as possible.
- Avoiding losing some information which needs to be referred by the user after the simulation has finished.

There are three display windows named as: the Display Frame, Display Equaliser and Display Plotter. Each of them has a specific function. Display Frame is used for setting operation parameters and displaying on-line information in text; Display Equaliser and Display Plotter are for displaying network error in two different ways after a certain period of time. Further details are described in §Apdx 1.5.

Some other windows were also designed in the tool. For example, an on-line help window provides information and help to the user to efficiently use the tool; a TTY window is implemented in order to provide the user with an environment incorporated with direct access to the *Unix*. All these can be referred in details in §Apdx 1.3.

### 7.3.2 FILE INTERFACE

Full use of the I/O facilities provided by the 'C' language and *Unix* operating system has been made so as to have a good interface for users. A file interface based on the *Unix* file system has been adopted to enable an easy I/O conversation between the user and simulator. This interface can improve the communication flexibility. For example the user can prepare some input data in an environment outside the tool or some operation results of the tool may be saved into a file which is accessible in any *Unix* environment. The following text explains through an example of loading training tuples why a file interface is needed by the tool.

loading training tuples

One of the user's motivations to use the tool is to speed up and simplify work on neural networks. Therefore the tool was designed carefully in order to reduce the user's work as far as possible.

First of all, suppose a tool can produce all the training tuples automatically as long as a mathematical expression is provided, the tool would be a perfect tool for those users who know the mathematical expression of their training tuples.

However, most of the training tuples are chosen from experimental results. It may be infeasible to represent the experimental results in a single mathematical equation, especially in a $n$ dimensional space. Moreover when the training tuples can be described as a real $n$ dimensional mathematical equation, the single equation may still not sufficient enough for the tool to generate the training tuples which are needed by the user. For example, when the user wants to train some tuples which should be chosen from a continuous function within a certain range along a certain path in a certain direction, it is impossible to expect the tool to generate these training tuples if only the overall function is provided. Some

other information such as the training direction and range are needed in order to get the desired tuples.

From the discussion above, it can be seen that it is not essential and appropriate to embody a facility such as a universal training tuple generator inside the tool. Instead the user is required to prepare a complete list of the training tuples as real-valued I/O tuples in a working file before the user is able to do any **training**. The user can prepare this working file in the background and write a small program to do it if it is necessary.

The above example explains some reasons why a file interface is needed and provided in the tool.

The user can also use the file interface to load, save and print out a network topology. I/O tuple files can contain a list of training or generalization tuples. The user can also load a list of initial weight states, a learnt weight or activity path through the file interfaces. At the same time, the tool can put some operation results into one of those working files. This ensures the user can reach the operation results from *Unix* files when the operation is completed.

The fact that both the 'C' language and *Unix* allow a line length and the size of a file to be unlimited enables working files to contain a great deal of information. For example, the dimensions for the training tuples are included in the file itself.

For more details about the file formats and specifications refer to the relevant section in §Apdx 1.4.

### 7.3.3 FUNCTION INTERFACE

This interface is designed to provide the tool with a degree of power and flexibility. It provides an opportunity for the user to design their own procedures for setting some

operation parameter values. Also it enables the tool designer to develop a new version of the tool with only a few custom-made new modules.

As an example, the function interface may be used as a parameter interface for calling a self-adaptive learning rate algorithm to set a suitable learning rate parameter for any three layer feedforward network during training. In the tool, an algorithm developed by Dr. M. K. Weir (1990) for this purpose has been built into it. This means that the algorithm has been compiled and combined into the tool. The user can apply the function, through the function interface to some of the training regimes provided, to calculate the training parameter value **learning rate** during the training.

By re-writing the procedure and re-compiling the tool, both the user and the designer can get a different version of the tool if it is necessary.

The same interface has been provided for setting another training parameter **momentum coefficient**. This interface has been reserved in **cbptool** in order, for example, to allow the user to write their own self-adaptive procedure for the momentum attribute in the future.

Besides being used for setting parameters, the function interface may also be used to choose one of the training and generalisation regimes. For example, the interface can be used to choose one of the interpolation regimes.

### 7.3.4 SPECIAL DUMPING INTERFACE

This is a kind of indirect interface in the sense that facilities obtained through this kind of interfaces involve using system utilities.

This is a facility to dump a graphic screen to a printer, which enables a user to get a hard copy of a network graphic image. This facility is only available indirectly (see the discussion in §7.6.3 ). The procedure is as follows:

- Firstly *Sun* rasterfiles are ported across the network: the *Unix* command *screendump* is used to get the screen image and store into a file; it is then ported across the computer network.
- Secondly the rasterfile formatted graphic files are converted into a **Macintosh** compatible bitmap image: using a convert application package to convert the *Sun* raster file into a *Mac* bitmap file. For example, using *convert* of *Mac*'s package to do the conversion.
- The bitmap file is then read into a **Macintosh** graphic package such as **SuperPaint**;
- Finally this image can be dumped to the **Apple LaserWriter** which supports **PostScript**.

## 7.4 Main Internal Representations and Implementations

As for design and implementation of all other computer applications, the tool designer has given a consideration of how to abstract, represent, handle and store the great number of data and relationship involved throughout the design and implementation. The discussions about neural network concepts in general or PBP in particular have been described in chapters 2,3,4 and 5. This section describes some of the major issues and methods related to the internal representations and the data structure design in **cbptool**. The discussion is focused in particular on the internal representations of a neuron, a network topology, and the main data structures.

### 7.4.1 CONSIDERATIONS

Data structure design is one of the most important tasks in any computer application. Many factors have to be taken into account, for example, the data structure should be simple enough for a quick, easy and efficient access; powerful and flexible enough to store all the necessary information in order to support all the operations which are likely to be performed on the logically related data structures in the application. Also a certain degree

of expansion of the data structures should be taken into consideration in order to be compatible, with any further extension of the application.

In **cbptool**, attention has been paid to the following three major aspects.

• **Dynamic memory management**

Because a network topology is user-defined, the size of many simulator variables which support the user-designed network is unknown at the time of compiling or generating the tool. It is both wasteful and difficult to pre-assign memory for these variables when generating the tool. Fortunately, the 'C' language provides with functions supporting dynamic memory allocation. These functions allow the application designer to allocate storage at running time and are particularly useful for implementing data structures of unknown size. However it should be remembered that 'C' language is not one of those languages, such as PS-ALGOL or LISP which have built-in routines to recognize and automatically release garbage, so that both the appropriation and release of dynamic memory need to be designed and implemented in *cbptool.* All the data structures used in the tool have been created using *calloc* and released using *cfree.*

• **Trade off between memory size and speed of operation**

It is important to design data structures with consideration of both the method of internal representation and speed of operation. This is especially important for the design of a neural network simulator, because a neural network tool requires both a high operation speed and large data storage. For example, the more complicated a network topology is, the more numerous the weight links are in a big neural network, and so the more data storage is needed. The way of represent the data can make a big difference to the requirement of storage memories and the speed of operation.

In order to support a large user-defined network, or a user-chosen operation with a great number of I/O operation tuples involved, it is important to consider the tool design in both

ways: saving memory and fast search. There may be a trade off between memory size for internal representation and speed of operation.

- **Support operations**

There is also another balance between the number of data structures and the speed of operations. With reference to this, a certain number of redundant data structures may be needed to support the speed of some operations. For example, a vector is used to store the number of input links for each unit in the network after the network is designed. This vector is not necessary as the information can be found through searching each *I_LINK* list of each unit, but it does help to speed up major operations.

### 7.4.2 IMPLEMENTATION

To go into exhaustive details about the step by step design and implementation of all the data structures used in the tool would be a time-consuming business. Here only the implementation approach to the main issues mentioned previously are described. For all further implementation details the user can refer to the commented simulator code included separately.

- **Data Structures of Networks**

Essentially, there are four main data structures which have been used in implementing a user-designed neural network. The following two pictures show the relationship between a network and its internal representation. The example of the user-designed network topology is shown in Fig.7.1 and a picture of a simplified data structure of the relevant network is shown in Fig.7.2.

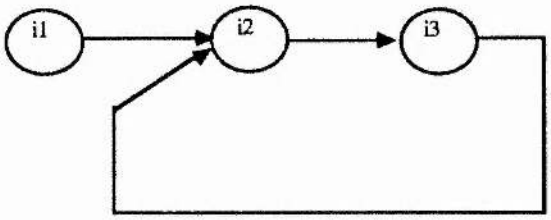



Fig. 7.1 A part of a user-designed network. Three neurons in the network as shown. There are two incoming links leading to the neuron i2, one link is from neuron i1 and another from neuron i3. There is only one outgoing link from the neuron i2. For neuron i3, there is one incoming and one outgoing link respectively.

A picture of a simplified internal representation of this network is in Fig.7.2:

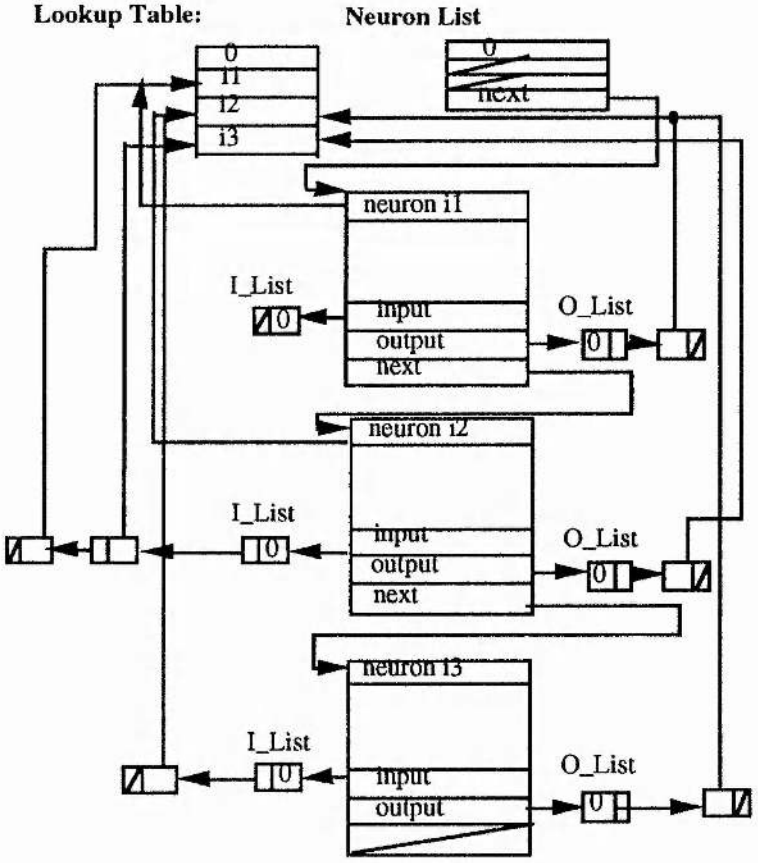



Fig. 7.2 Picture of a simplified data structure of the network

Fig.7.2 shows the internal data structures of the neural network in Fig.7.1. The four types of the structures are: *Lookup_Table*, *Neuron_List*, *I_List*, *O_List* which are the structures for internal representation of any network in the tool. In this example there is one *Lookup_Table*, one *Neuron_List* with 4 elements, 4 *I_List*, and 4 *O_List* lists respectively

with various numbers of elements in each *I_List* and *O_List*. The first element in each structure is a sentinel value, a zero structure. Each type of data structure is described below.

(1) Lookup_Table list

This is used as a table for holding all the neuron IDs of a network. For each neuron in the network, its ID needs to be stored in this table, then the table can serve as a key index table in order to refer to any neurons of the network by the IDs.

Although referred to as a table, this data structure is implemented as a linked list with each node representing an integer neuron identifier ID. Every other structure that has a reference to a neuron ID points to the *Lookup_Table*.

The main design motivation here is it will be easier to keep all the data interrelated and unanimous with the centralized ID management. The main benefit of this technique is that if a change to a neuron ID is necessary then a simple change of the ID in the *Lookup_Table* ensures that all references to that neuron have been changed.

(2) Neuron_List list

This data structure is designed for storing a number of *neuron structures* each of which consists of some information about the neuron (such as ID, activation value etc) and the interfaces associated with representation of the inter relation of the network to which the neuron belongs to.

In essence the internal representation of the neurons in a user-designed network is a linked list, each element of the list associated with a neuron is implemented as a *structure* of the 'C' language. Each *structure* consists of five types of fields. One is used for the relevant information about the associated neuron, the other four are used for pointers pointing to the other structures for representing the associated network. The following points give the five major fields one by one:

(i) ID-pointer field: This field is used to store the pointer pointing to the associated ID which is stored in the *Lookup_Table.*

(ii) Neuron-information field: This field is used to store a certain number of variables associated with the neuron such as error_value, output_value etc.

(iii) Input-link-pointer field: Points to the *I_LIST* of the neuron. See description below about the I_LIST list.

(iv) Output-link-pointer field: pointing to the *O_LIST* of the neuron. See description below about the O_LIST list.

(v) Next-link-pointer field: This field is used to store the pointer pointing to the next neuron structure in the *Neuron_List.* The value will be NULL for the last neuron structure in the *Neuron_List.*

(3) I_LIST list

This data structure is provided to support some operations which may be necessary in searching all the neurons feeding into a given neuron.

As described above within every *neuron structure* associated with each neuron, one field called *Input-link-pointer* is defined and pointing to the associated *I_LIST.*

Each neuron has its own I_LIST list. Each *I_LIST* is a linked list. The number of the elements of each I_LIST is dependent on the number of the units linking toward the unit associated to the I_LIST. Each element contains two pointer fields. One field is either pointing to the *Lookup_Table* or is 0 for the first element in each *I_LIST.* The other field is either empty or points to the next element of the *I_LIST.* All incoming neuron IDs of a neuron can be found by searching through the associated *I_LIST* of the neuron. Therefore for each given neuron, its *I_LIST* can be used to find out all the neurons leading to it. The direction of each link is from one of the neurons in the *I_LIST* list to the neuron which owns the I_LIST.

(4) O_LIST list

Similar to the *I_LIST* list, *O_LIST* list is provided to support some operations which may be necessary in searching all the neurons receiving output value of a given neuron.

For each neuron, one field called *output-link-pointer* is defined in its *neuron structure*. This is to point to its own *O_LIST*.

As for the *I_LIST*, each unit has its own O_LIST. The number of the elements of each O_LIST is dependent on the number of the units linking from the unit associated to the O_LIST. The structure of O_LIST is implemented as a linked list with two fields. For a given neuron, its *O_LIST* can be used to find out all the neurons which associate with the given neuron. But in *O_LIST*, the direction of each link is from the given neuron to one of the neurons in the *O_LIST*.

• **Weight structure**

Two neuron IDs are needed to refer uniquely to the strength of the link if there is a connection between any two neurons. Several alternatives were considered in order to design the weight data structure, e.g. a quadratic array or a directed graph. A balance is needed among the memory space requirement and the accessing time as well as the programming complexity. It was finally decided that the weight structure should be implemented as a ragged array in which the depth of the array is equal to the number of the locations required.

Comparing with a quadratic array or a directed graph, a ragged array technique maximises storage efficiency, and has a reasonable accessing speed.

The technique works by allocating a single vector for the network first, and then a variable size vector for each element of the single vector is allocated.

The number of the elements in the single vector is equal to the number of neurons in the network. The size of each variable vector depends on each associated neuron. It reflects the number of connections leading into the neuron. The contents of each element in each variable size vector is a pointer to an abstract data type which stores the following information:

(1) The ID of a neuron;

(2) The information associated with that link. For example: the weight value; change in weight per training tuple presentation; previous change in weight; some data to support the learning rate function, etc.

As an example, Fig. 7.3a is a simplified picture to represent the internal structure of a part of the network shown in Fig. 7.3b. There are 3, 2, 2 and 4 incoming links respectively for the neuron 3, 4, 5 and 6 of the network. Fig. 7.3b shows the ragged array of the 4 neurons and the related abstract data structures.

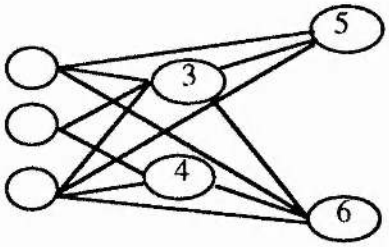



Fig. 7.3a The topology

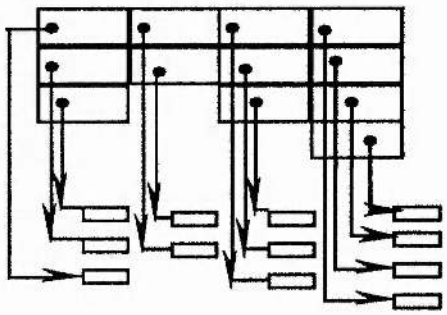


Fig. 7.3b The data structure

Thus a ragged array is used to store the weight data which requires only two index values. For example, the weight value between unit 6 and 2 will be found by the following procedure:

(i) Locate the position associated with the unit 6 in the first row of the ragged array; there is a various size vector associated with this position;

(ii) Go through each element of the vector and do (iii) until either a correct abstract data structure is found or a failure is reported, which is that no required data type is found after going through all the elements associated with the vector;

(iii) Access the content of the element to get the pointer, which points to the abstract data type defined for each link;

(iv) Use the pointer to check the content of the ID field of the associated abstract data type. The search succeeds when the number in the ID field stores the ID of the required unit. Then the weight value between the two units can be found in the associated data field.

• **The internal representation of the tuples**

Using the file interface mentioned above, the tool is able to access the training or performance I/O tuples through working files specified by the user. A data structure is needed to support the interface and store the data in the system. It was decided that the tuples should be stored in an array as shown below. Thus, the horizontal length of the array is the sum of the number of the input and output neurons, the depth of the array is the number of the tuples.

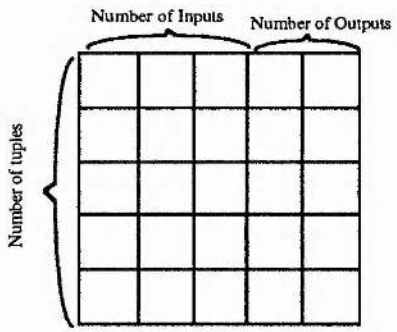



Fig. 7.4 A simplified picture of data structure of the tuples

• **Other data structures**

Some of the data structures used for the main variables of the operations such as **training** and **performance** are mentioned below. The sizes of these data structures depend on the

number of the neurons of a user network so that they vary from one network to another. More details about the working data structures can be found from the enclosed code. The more important structures are described as follows:

(1) Input vector: This is used for storing the external input information for each neuron of a network.

(2) Activation vector: This is used for dealing with the activation value of each neuron.

(3) Delta vector: This is used for the delta value for each neuron during the **training**.

(4) Target vector: This is used to store the target value of each neuron.

(5) Pattern_Error vector: This is used to store the pattern error value of each neuron when performing an operation such as *training*.

(6) Link_number_to vector: Each element in the vector is associated with each neuron to store the number of the links leading to the associated neuron.

## 7.5 Main simulator parts

All the tool facilities can be divided into the following six distinct aspects: Design, Build, Train, Performance, Batch, and Display.

### 7.5.1 THE DESIGN MODULE

This is a module to support the design of a user network in a mouse and menu driven environment. The function of the module includes enabling the user to directly access neurons and connection weights, to design a network with arbitrary interconnection between all the neurons. A feedforward or recurrent network can be designed through this module. For more details please refer to the User Manual in the appendix §Apdx 1.3.

### 7.5.2 THE BUILD MODULE

This is to support the automation of the construction of the internal representation of a user designed network.

In order to fit with the whole system design requirement and without loss of freedom in designing the user's network, a method is necessary for reorganising a newly designed network before passing it to the **training** or **performance** operation for the reasons:

(1) the training or performance operation expects the network topology to be in a fixed format (i.e. all input neurons precede all hidden neurons, all hidden neurons precede all output neurons, all output neurons precede the bias neuron);

(2) a most recently created neuron is always guaranteed to have the highest ID number;

(3) neurons of any given type can be positioned anywhere by the user on the canvas during the design of a network so that some neurons may be redundant (i.e. unconnected to any other neurons in the network).

Before any operation taking place on a user-designed neural network, it is also important to ensure the validity of the network topology. Some common problems occurring in validating a network for example are the existence of unconnected neurons within the network, the existence of an invalid connection between two neurons.

It can be seen that procedures are needed to enable the screen topology to be both valid and tidied up and then to create a real internal topology for manipulation during performance. This module brings more freedom and on-line help for the user to design his networks. For more details please refer to the User Manual in the appendix §Apdx 1.3.3 Menus D.

### 7.5.3 PARAMETER SETTING MODULE

Some interfaces are needed for the user to set up specific parameters for the operations. For example, one interface is provided for the user to load in the initial weight states for a batch training operation; another enables the user to decide whether the operation is carried out in background. For more details refer to the User Manual in the section §Apdx 1.3.3C.

### 7.5.4 THE TRAINING MODULE

This module is designed to support the training operation on a user-defined network through one of training regimes. The module has a total of six training regimes, of which five of them are based on the PBP approach for training feedforward or recurrent networks; one is based on SBP— the state-based BP for training feedforward networks. For more details please refer to the User Manual in the appendix §Apdx 1.3.3 E.

### 7.5.5 THE PERFORMANCE MODULE

This module is designed to do generalization for any trained feedforward or recurrent networks through one of the regimes.

Two kinds of regime have been designed here. One is a path-based regime which implies that the I/O tuples must be chosen from a certain training path. The other is a more general time-based regime. It means that the performing tuples can be arbitrarily chosen from a trained surface but a training time must be provided explicitly as an additional parameter. Two path-based regimes have been implemented in **cbptool**. One is for feedforward and the other is for recurrent networks. One time-based regime for feedforward networks has been also implemented.

For the three regimes above, a linear interpolation technique has been employed to perform generalization. Two extra function interfaces are also provided for the user to design their

own interpolation techniques. For more details refer to the User Manual in the section §Apdx 1.3.3F.

### 7.5.6 THE DISPLAY MODULE

This module deals with the display of all outputs generated by the network operations. Two types of information need to be displayed on windows provided by the simulator. One is about the internal dynamic results or final results of an operation. The other is general messages about the operation errors or warning prompts.

In the simulator, all general messages are displayed through a popup window. Operation results are shown in two ways. One is to interpret, represent and display graphically the data produced by the training operation. This kind of output can give the user a good overall representation of the dynamic changes occurred within the training operation. The other is to store some operation results into files through a file interface. These output data are available in a textual form which enables the user to access the data when it is necessary. For more details please refer to the User Manual in the section §Apdx 1.5.

## 7.6 Discussions

As for any tool, limitations in the design or implementations need to be discussed. Some issues encountered throughout the implementation of the user interface or data structure design will be discussed below. It is hoped that with additional consideration or modifications, the tool could work even better.

### 7.6.1 SYSTEM ENVIRONMENT LIMITATIONS

(1) *SunView* limitation

Although *SunView* is a very powerful tool in many aspects, it is not an industry standard tool and so portability is limited. There are also some other limitations in *SunView*. One

example is subsequent access to any panel item which has been dynamically created. Although the dynamic allocation of panel items is possible, difficulties were encountered to allocate static handles dynamically. It is very problematic in attempting to access a dynamically created panel item within *SunView* at a later stage, as it is necessary to refer a panel item through its associated handle, allocated to the item statically. This limits flexibility of the window interface in the tool.

(2) File system limitation

There is a constraint on the maximum number of open files allowed for a single user in the file system of the ***Unix***. This is a major problem encountered throughout the implementation of the tool user interface. The current system (Sun's release 3.0 system) has a limit of 30 file descriptors per *Unix* process. As each active frame in the *SunView* environment is represented by an open file, it means that the tool is limited to a maximum of 30 windows active at any time. When file interfaces are needed to access external files for certain operations, the limitation on the number of the active windows at any time becomes increasingly severe during operation. This problem has been overcome by destroying windows or files as soon as they are not in use and then recreating them when they are needed.

The second limitation is that the file formats depend very much on ***Unix***, they naturally restrict portability to non-***Unix*** systems.

### 7.6.2 FLEXIBILITY AND SPEED

There is a trade off between providing a flexible interface for the user to choose an operation from and the tool speed. Many memory sizes of data structures used in the tool are unknown before compiling the tool. This is why so many memories are assigned dynamically. Some associated operations provided by the tool can operate on the same data structure if the operations are executed one after another. But in order to increase the

flexibilities for the user in choosing operations, the memory has to be released after completing one operation and re-allocated before performing a new operation. Although this is not a major problem, it does waste time and causes the tool to run more slowly than would otherwise be the case.

### 7.6.3 GRAPHIC SCREEN DUMPING

This facility is only available indirectly (refer to the description in §7.3.4 ) in **cbptool.** The user can get a hard copy of his network image on screen by dumping the graphic image from the screen to a printer. However, the complexity and relatively low significance for the thesis of actually implementing this feature meant that direct implementation is currently unavailable.

### 7.6.4 ACCESS SPEED

As the data structure of NEURON is designed as a linked list, and a sequential search on the linked list has been implemented, this could slow down the access speed for a big network.

Some future effort could be made to improve the access speed for locating a neuron in a big network. For example, creating a two dimensional array that maps onto the grid coordinates of the canvas, each element in the array would contain either a pointer to a neuron or NULL. The biggest disadvantage of this design is that it would then be extremely wasteful in memory space for a small network user with most of the array spaces being empty. One resolution of the problem might be to use a hash array to achieve both a faster access speed and less waste of memory space.

### 7.6.5 FORMAT OR LANGUAGE

A great deal of further time and effort could be expended to ensure that the data entered through the file interfaces is valid. For example, all file formats in the file interfaces would be much less rigid if a function called **'strtok'** provided by 'C' language had been used. However, a large amount of machine time is needed to do the on-line check if there is a large amount of data in the file. No matter how much effort is exerted in verifying input information, it is still impossible to protect a system from invalid data. So data verification is left to user.

A suggestion for improvement in this aspect is that the file interface for this purpose be replaced by implementing a kind of language in the tool, which can then be used by the user to describe the data more conveniently.

### 7.6.6 FURTHER FUTURE FACILITIES

At present there are no short cuts to design a network without specifying the units one by one; no way of running the tool in a completely batch mode without starting in SunView or running the tool without interacting with *SunView*, where all parameters can be described in a login file in text. This later facility could be useful if the user wants to set up long experiments without *SunView*.

## 7.7 Conclusion

The tool **cbptool** firstly has been used as an introductory tool to the PBP based models. Secondly it has been used as a research tool to complete all the experiments on the basic research and case studies for FCBP and RCBP. It has been found that the **cbptool** is an easy to use, flexible and graphic based tool which works satisfactorily.

The **cbptool** provides a solid base for the user to work on. A lot of thought and consideration has been put into the design and implementation in order to have a good balance among different requirements which may be requested by different users. **cbptool** has also been designed with strong expansion possibilities. It currently functions as a valuable research tool with further possible development in the future.

# CHAPTER 8

# CONCLUSION AND RECOMMENDED FUTURE WORK

## 8.1 General remarks

The objective of the research work is to understand and further investigate the way artificial neural networks may deal with learning sequential processing. A major learning algorithm — backpropagation, is chosen to explore the possibility of further extending the strength of the standard model to achieve this goal. The thesis is a primarily conceptual and fundamental approach. The design, theoretical and empirical understandings are concluded here.

### 8.1.1 General achievements

New concepts and methodologies were evolved from the notion of using the continuity underlying a path in design and the implications for training and generalisation of converging to a goal weight path. These aspects together show how that the fundamental capabilities of SBP may be generalised.

The two path-based models FCBP and RCBP have been explored in this thesis. They suggest considerable advantages over SBP for sequential and other problems. In particular:

(1) They allow arbitrarily close approximation of a continuous function or dynamical system within a fixed topology.

(2) Problems involving sequential I/O associations with underlying continuity can be trained as accurately as required more feasiblely by using the FCBP or RCBP training models.

(3) Interpolation techniques may be applied through the weight or activity states for generalisation. In generalization, the fundamental change made is for various interpolation techniques to be appliable to the states. This increases the level of approximation of goal weight states for untrained I/O associations from zero order (SBP) to a higher orders (PBP) depending on the order of the approximation and the type of the approximation method used.

(4) Algorithms explored for the SBP approach in a forward or recurrent network may be employed as sub-tools by the two path-based models at each training position. This allows a general extension of SBP approaches.

The features of FCBP and RCBP are exemplified through experiments, and the results support the above concepts and analysis. Training results show the trade-off between doing several searches in a number of relatively small weight spaces against doing a single search in a relatively large weight space. Generalisation results show significant improvement in accuracy when models can make use of higher order interpolation methods.

Applications in complex analogue-binary classification, signal processing or, in general, problems where analogue I/O associations or complex binary classifications are involved, may be investigated by using these new models. To support this, in addition to the exploration of the new models, the work that has been done in this thesis also includes the design of a path-based simulator **cbptool**. The simulator provides a convenient tool with a graphic and user friendly interface for workers in this area to further explore PBP and related applications.

### 8.1.2 LIMITATIONS

The PBP approach is intended to offer a higher level extension to the SBP framework. It has been designed to deal with aspects of sequential and temporal signal processing in

artificial neural networks. However, as a first approach of its type, there are limitations in applying the models to solve some real world problems. The limitations can be listed as follows:

(1) In both models, the number of the training positions along each of the I/O training paths must be the same.

(2) During performance, correct associations rely on the weight signals synchronising with the associated input signals and internal state signals. Since BP is a software technique, synchronization is straightforward in practice. However, synchronization becomes a future issue with analogue hardware implementation.

(3) In RCBP, one-many associations cannot be targetted at the first training position in the current regime employed by the RCBP approach.

### 8.2 Recommended future work

Some work can be done to further the exploration of the path-based framework:

(1) Applying the PBP approach.

The basic intention of the PBP approach is that tasks involving sequential problems (refer to §4.3.1) can be investigated using the PBP models. A sequential problem is one where the order in which the inputs occur has to be learnt as well as the individual associated outputs. Also though, as a software simulation, any tasks which can make use of the time and weight index feature described in §4.3.1 can also be investigated using PBP.

(2) Improvement of the RCBP capability:

Some other dynamic system associated with finding a single weight state in recurrent networks might perhaps be considered to try to eliminate the limitation on the first position in RCBP.

(3) It is relatively straightforward to consider incorporating some other state-based approaches into FCBP to speed up the training at each training position.

(4) Compared with the linear interpolation used in the thesis, higher level interpolation techniques may be used to produce a better approximation for weight state or activity state paths.

(5) Improvements may also be made in the design and implementation of the **cbptool** simulator, such as to increase the processing speed, save more computational resources, improve the user interface, develop more facilities, etc.

(6) A neural mechanism for inputs and weight state associations could be explored.

# APPENDIX 1

## THE CBPTOOL USER MANUAL

### 1. Introduction

The user manual serves as a guide for the use of the simulator **cbptool**. It describes the facilities provided and how the **cbptool** works. It has four parts: (1) How to access the simulator; (2) What the main facilities and parameters of the simulator are; (3) What the forms of the specification files are; (4) How and in what forms to store and display the results.

### 2. Accessing the simulator

To access the simulator, the user needs first to get into the working directory and then ensure that the executable file **cbptool** exists before running the simulator. The following routines explain the details:

• To get into the correct working directory: use the Unix command **cd** to locate the directory under which the simulator is located.

• To check the executable file **cbptool**: if the executable file **cbptool** exists under the working directory, it means that the simulator has already been built and is ready to run. If not, the simulator needs to be built.

• To build or recompile the simulator: the simulator can be built and recompiled or updated. To do so, the user should check if a working file **makefile**, related to the ***Makefile*** in the Unix system, is present in the user's working directory and the contents of the **makefile** are satisfactory, then carry out the Unix commands **touch** and **make**. More details are as follows:

(i) Check the contents of the **makefile** file using: **vi makefile**.

(ii) Type in the command: **touch neural.c**

The Unix command touch will alter the time-stamp on the simulator main program ***neural.c*** and ensure the associated Unix command **make** builds the simulator.

(iii) Type in the command: **make**

The Unix command **make** will initiate the **makefile** program and build the simulator by compiling the 'C' source code and linking to the necessary libraries. This process normally takes a few minutes, depending on the system. If any errors occured at this stage, the user should refer to the Unix ***Makefile*** manual.

• To run the simulator: in order to run the simulator, the user should be in the SunView environment. This can be achieved by typing **`suntools`** or **`sunview`** and then **cbptool** to run the simulator. The simulator itself will take about 10 seconds to come up, as many SunView windows, menus, panel items need to be created and the memory of the canvas need to be allocated during this time. A number of windows, menus, and tools should then appear on the screen.

• To quit the simulator: to quit the simulator, simply choose the **quit** option in the **`Quit`** menu and confirm the action.

### 3. Main Facilities and Parameters

All the facilities provided by the simulator can be divided into three different aspects: Windows, Aid-Tools and Menus. Each aspect includes a number of particular facilities. Each of the facilities in turn is described and their basic features are outlined.

3.1 WINDOWS

The whole Sun workstation screen is divided into six different windows providing a design environment with menus and aid tools for the user to work within the simulator. A picture of the screen is shown below.

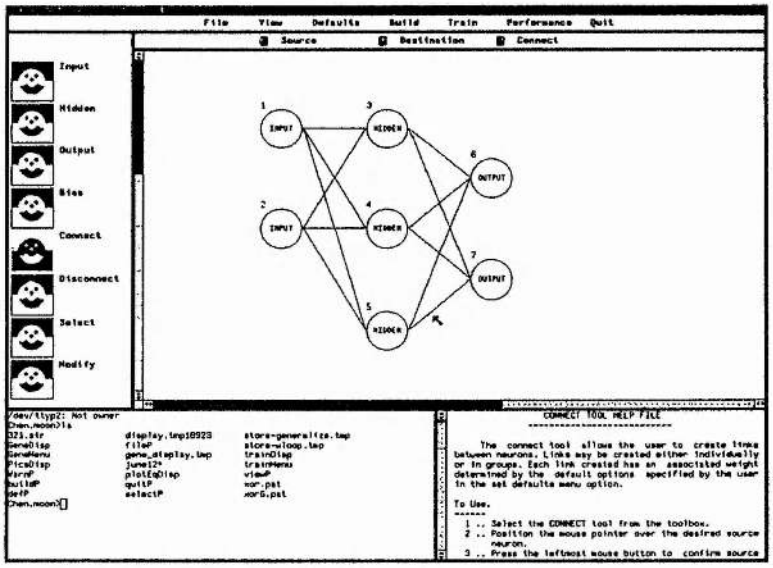



Fig. Apdx 1.1 The over view picture of **cbptool**

**A. The Canvas Design Window**

This is at the centre of the screen, occupying the most screen space. It is also the main window of the simulator on which the user can design his network using the tools in the Toolbox. This window is scrollable in both horizontal and vertical directions for networks larger than the window. Currently, the user is limited to a design canvas of 20 by 20 elements. This means that the maximum number of the neurons in one user-designed network is 400.

**B. The Terminal Emulator**

On the left side of the screen just below the design window, the Terminal Emulator window enables the user to interact with the Unix operating system while remaining in the simulator. Essentially this is a shell, in which the user can perform any tasks which could performed within the operating system completely independent of the simulator.

**C. The Text Window**

This window is immediately on the right side of the Terminal Emulator window. It is used to display text messages such as the on-line help files associated with each tool. The window is scrollable for large text files to be displayed without loss of any contents.

**D. The Toolbox**

The Toolbox is on the left side of the Design Canvas Window. It is a box containing all the tools needed for the design and creation of a neural network topology in the simulator. Each tool is represented by an icon with an associated label. All the functions of each tool will be described in details in section §Apdx 1.3.2.

**E. Menu Bar**

Across the top of the screen frame is the Menu Bar. It contains a number of menus, each dealing with a special part of the simulator operations. Each of the menus and associated options will be described in depth in the section §Apdx 1.3.3.

**F. The Mouse Panel**

This is a thin horizontal box below the Menu Bar starting from the right of the Toolbox. It provides the user with an indication of what each of the mouse buttons does for a given tool. The Mouse Panel contains

three image icons representing three specific mouse buttons. Each button has different functions for each chosen tool. Each icon is associated with a text message which is updated according to the current choice of a tool from the Toolbox. As an example, the following texts describe what are the messages for some of chosen tools.

(1) For Input, Output, Bias, and Hidden tools:

Left button: place neuron; Right button: delete neuron.

(2) For Connect, Disconnect, and Modify tools:

Left button: source neuron; Middle button: destination neuron; Right button: confirm.

(3) For Select tool:

Left button: place neuron; Right button: confirm

3.2 AID-TOOLS

All the aid tools are presented on the screen within the Toolbox window which has been described in §Apdx 1.3.1D. The following text discusses the general use of the tools and the functions of each tool.

**A. To select a tool**

In order to use a tool in the Toolbox, the user must choose the tool first. This can be done by placing the mouse cursor over the tool icon and then pressing the leftmost mouse button. When a tool is selected, the tool is highlighted, and any previous tool choice is deselected. This ensures that only one tool is active at a given time. An on-line help message associated with the tool will be displayed in the Text Window (refer to §Apdx 1.3.1C ).

**B. Tools and their functions**

(1) Input neuron Tool

The **Input** tool allows the user to create or delete input neurons. If a user wants to create an input neuron, the user should position the mouse cursor on a free space of the design canvas and press the leftmost mouse button. A new neuron will be created at the grid position which is the closest to the chosen point on the canvas with the chosen parameters specified by the user through the menu option: **`set parameters`**.

If a user wants to delete an input neuron, the user should position the mouse cursor over the neuron on the design canvas and press the rightmost mouse button. The target neuron body and all associated links to other neurons from the target neuron will be removed.

Note: A new neuron will not be created if a neuron already exists at the chosen point on the design canvas.

(2) Hidden Neuron Tool

This **Hidden** tool allows the user to create or delete hidden neurons. The guide-line to create or delete hidden neurons is the same as for the input neurons.

(3) Output Neuron Tool

This **Output** tool allows the user to create or delete output neurons. The rules to create or delete output neurons are also the same as for the input neurons. The **Output** tool will only delete output neurons.

(4) Bias Neuron Tool

This **Bias** tool allows the user to create or delete the bias neuron. The rules are very similar to the rules for input neuron except that only one bias neuron is allowed in each network.

(5) Connection Tool

This **Connect** tool allows the user to create feedforward or recurrent links between two neurons. Links can be created either individually or in group.

To create a link (or links), the user must position the mouse cursor over an icon of a desired source neuron on the design canvas to select the source neuron by pressing the leftmost mouse button. A number of the source neurons can be selected in this way.

Similarly, to select a destination neuron, the user must position the mouse cursor over an icon of a desired destination neuron on the canvas by pressing the middle mouse button to confirm the selection of the destination neuron. A number of the destination neurons can be selected in this way.

Once all of the source and destination neurons in a group to be fully connected have been selected, links between them can be made by pressing the rightmost mouse button. All valid links will be drawn on the design canvas.

Each link created has an associated weight, the value of which is determined by the chosen parameter options specified by the user through the **`set parameters`** menu option.

Note:

• In order to get a nice or clear topology picture on the screen, the user should have a well-prepared topology before drawing it on the screen. A careful design that avoids any overlaps among links is necessary, because links can be hidden if an overlap exists.

• The validity of all possible links is displayed below. Here **v** denotes that the connection is valid, **Inv** denotes invalid.

| Source Neurons \ Destination Neurons | Input | Hidden | Output | Bias |
|---|---|---|---|---|
| Input | Inv | v | v | Inv |
| Hidden | Inv | v | v | Inv |
| Output | Inv | v | v | Inv |
| Bias | Inv | v | v | Inv |

Fig. Apdx 1.2 The valid relationship of links

(6) Disconnect Tool

This **Disconnect** tool allows the user to disconnect links between two neurons. It works on exactly the same principle as the **Connect** tool, except it deals with disconnecting existing links rather than creating links.

In order to keep the integrity of the network, checks are made to see if the target links to be disconnected exist. If the links exist, they are disconnected, otherwise nothing happens.

(7) Select a Neuron Tool

The idea of having the **Select** tool is to allow the user to view and modify existing neuron attributes. Any confirmed changes or settings made to the neuron are stored within the network topology and will be applied during any operations associated with the neuron. In **cbptool**, this tool is only provided for viewing the attributes.

In order to select a neuron, the user must position the mouse cursor over the icon of the target neuron on the design canvas by pressing the leftmost mouse button to confirm the selection of the target neuron. A popup display, shown in Fig.Apdx 1.3, will appear on screen, which shows the neuron attributes as well as details of links to and from the neuron.

(i) The detailed information associated with the features of the target neuron, e.g. its ID and type is displayed in the centre panel. Except that the two latter values are unalterable, all other displayed attributes are user-alterable within the panel.

All the functions associated with the neuron can be selected in any of the two ways: one is to use the SunView cycle facility to cycle through the available options; the other is to select through the function menu. The first method is by clicking on the appropriate function option with the lefthand mouse button when cycling through all the options with the mouse button; the second method is to press the righthand mouse button to bring up the menu.

In order to select some other attributes such as all the values listed in the centre panel, the user should do the following things:

• Position the text cursor near to the appropriate text item using the mouse;

• Edit the text in the normal manner to get a new valid value for the associated attribute.

(ii) Values for the incoming and outgoing links to the target neuron are displayed in the left and right side panels respectively.

These values are for display only and cannot be altered through editing the text on the screen. The **Modify** tool allows the user to alter the link attributes.

Each link related to the target neuron is represented by a neuron ID and the associated weight value, where the ID can be a source or destination neuron ID associated with the link. Each of the two panels has an associated scrollbar, it enables the user to view all of the links without missing any links in case that the target neuron has too many incoming or outgoing links for the panel windows.

(iii) When a user has made any changes on the alterable items, it is the time for the user to save or discard the changes. The changes to the topology will be saved if clicking on the confirm button, or discarded by clicking on the cancel button. In both cases, the neuron popup will disappear after clicking.

(8) Modify Weights Tool

This **Modify** tool enables a user both to view and modify any weights on existing links. The last change made to the weights will be stored within the network topology and will be applied by any operations where the weights are needed.

In order to modify a weight, the user must select the source neuron of the link by pressing the leftmost mouse button, confirm the destination neuron with the middle mouse button and press the rightmost mouse to bring up the weight popup. See Fig. Apdx 1.3 below.

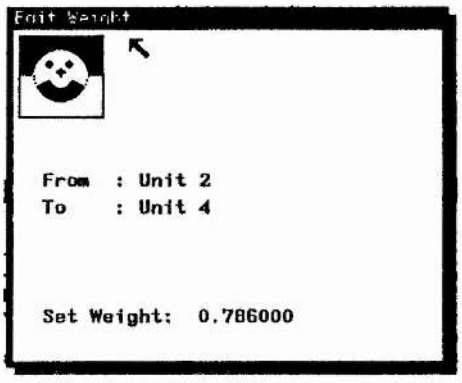



Fig. Apdx 1.3 The *Modify* weights popup

The weight popup displays the weight value between the two linked neurons. The user can modify weight within this popup.

All methods of using the **Modify** tool are very similar to those of the **Connect** tool described above, except in the following three aspects:

• The user is only able to access one link or modify one weight value at each time;

• The modification cannot be cancelled after the user has modified the weight. In other words, there is no closing of the dialogue box without saving the modification.

• No effort has been made in the simulator to ensure if the data entered are valid, the verifying of data is left to the user.

3.3 MENUS

This is the end-user interface of all operations provided in the **cbptool**. There are seven menu items in the Menu Bar. Each item contains various numbers of optional sub-items in order to deal with an operation. The following text will illustrate the usage and function of each item in the Menus Bar window.

**A. File Menu**

This menu contains a number of options dealing with file manipulation. There are three sub-items for the user to create, load and save a network topology. A picture is shown in Fig.Apdx 1.4a.


Fig. Apdx 1.4a The *File* menu.

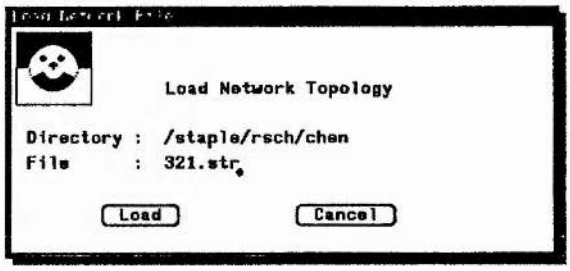



Fig. Apdx 1.4b The loading popup.

(1) New

By choosing this sub-item, the user can design a new network topology on the design canvas. The previous contents on the design canvas will be cleared and discarded.

(2) Load Topology

This is designed to load in a previously stored network topology from an external file into the design canvas. The previous contents on the design canvas will be cleared and discarded.

When this item is chosen, a load topology popup will be called automatically in the simulator and displayed (see Fig. Apdx 1.4b). The popup enables the user to specify both the directory and filename of the topology file.

To load the topology file, the user should type in both the directory and filename correctly. When it is checked that both the names typing are correct, the operation can be confirmed by clicking on the load button. If the file does not exist or is of incorrect format, an error message will be displayed.

There are two things to remind ourselves in this kind of operation:

• The default directory is the current working directory;

• The Terminal Emulator is available to verify whether the file exists;

(3) Save Topology

This is designed to store a network topology currently on the design canvas to an external file.

Again when this item is chosen, a popup for saving topology will be called automatically in the simulator and displayed on the screen. The popup enables the user to specify both the target directory and filename for saving the topology file.

The operation will be confirmed by clicking on the save button. The file will be stored in a textual form.

If the directory specified by the user does not exist, an error message will be displayed. Also the Terminal Emulator can be used to verify whether the topology file exists after the operation.

**B. View Menu**

This is a menu for a user to deal with all the display canvases designed in the simulator. The menu contains three sub-items, each of which enables the user to clear a canvas.


Fig. Apdx 1.5 The *View* menu

(1) Clear Design Canvas

Identical to the *new* menu option in the **File** menu, this option allows the user to clear the design canvas, and enables the user to design a network topology within a clear window.

(2) Clear Plotter Canvas

The plotter canvas in the simulator is provided to plot dynamically a graph which shows the network error at a certain training time when some operations are performed on a training network. This option enables the user to clear the plotter canvas.

(3) Clear Equaliser Canvas

The equaliser canvas in the simulator is provided to display graphically the dynamic changes of error for each training tuple when some operations such as **FCBP1** and **DBP** training regimes are performed on a network during training. This option enables the user to clear the equaliser.

For further details about the functions of the Plotter and Equaliser canvases, the user should refer to sections §Apdx 1.5.1.B and C.

### C. Set Parameters Menu

This menu contains only one item. The related function is to enable the user to set some parameter values in the simulator. See Fig. Apdx 1.6a below.


Fig. Apdx 1.6a The *Defaults* menu

When this item is chosen, a popup will be displayed on the screen (Fig. Apdx 1.6b). The popup enables the user to customize the parameter values to suit his own needs. By modifying the contents shown in the popup and confirming the modifications, the associated attributes will be set in the simulator.

Note that the parameter values will only affect neurons and links created after the parameters have been set. The user should do the parameter setting before creating his new network. The following text describes each parameter in turn.

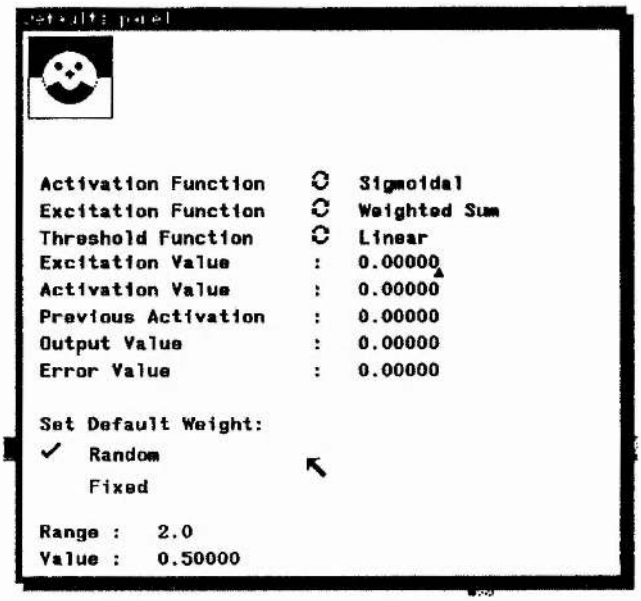



Fig. Apdx 1.6 The *Default* popup

(1) Activation Function

This is a function interface which enables the user to choose an activation function from a list of functions. The activity value of a given neuron in the network will be calculated according to the chosen function. The default is the **Sigmoidal** function. Other functions listed are the reserved interfaces for possible future implementations.

(2) Excitation Function

This interface enables the user to choose an excitation function from an excitation function list. The excitation value of a given neuron will be calculated according to this choice. The default is the **Weight Sum**. As for the Activation Function interface, the other functions listed are the interfaces for future implementation.

(3) Threshold function and values

The user can decide the function and values through the menus. In **cbptool**, these interfaces listed are reserved only for future implementation.

(4) Initial-Weight-Setting Parameter

The user has to decide in which way to set the initial weights for his network when applying the training operation. There are two options for setting this parameter value: **RANDOM** or **FIXED**.

If the user chooses **RANDOM** option, the initial weight values will be randomly set within a desired range. The desired range is a range decided by the user by typing the associated value within the item **RANGE**. For example, if a range of 1.0 has been decided by the user, then the simulator will produce all initial weight values randomly within the range (-1.0,+1.0).

If the **FIXED** value option is chosen, fixed initial weight values will be set by the user through either another facility: the **Modify** tool during creation of the network or the file interface provided together with training during training.

Note:

When the user chooses the **FIXED** option as a default, care should be taken to avoid to have a symmetrical network. This is because the network will be unable to be trained properly for neural reasons (McClelland and Rumelhart, 1988).

**D. Build Menu**

Facilities are provided as aids to the user to make sure if a network topology is valid and the internal representation of the network is tidy. This menu contains two options *tidy* and *valid* which help the user to tidy up a user-designed topology and ensure the user-designed topology is logically valid.

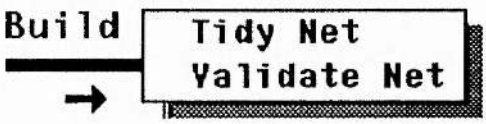



Fig. Apdx 1.7 The *Build* menu

(1) Tidy Net

As a menu option, the user can tidy up his own network design by applying this option to the network currently on the design canvas. This procedure enables the screen topology to be tidied up and then goes on to create a real internal topology for manipulation during performance.

(2) Validate Network

Similar to the **Tidy Network, Validate Network** is to do with the validity of the network as a whole. The aim of this facility is to check the topology currently on the design canvas to ensure if the topology is correct and suitable for the simulator operations.

The criteria for a valid network are as follows: (i) A network must have at least one input and one output neuron. (ii) For each type of neuron, there must exist several certain valid connections.

This means that an input neuron must have a connection to other neurons and there is no incoming connections from any neurons. A hidden neuron must have both connections from input and output neurons. The bias neuron is only able to link to the hidden or the output neurons (Fig. Apdx 1.2 shows the valid relationships among the units).

**E. Training Menu**

This menu contains many options for training a user-designed network and displaying dynamically the data produced when performing the training operation(Fig. Apdx 1.8a).


Fig. Apdx 1.8a The *training* menu

(1) Open Display Plotter

This is to open the plotter window which enables the user to see dynamically the graph of the summed error of a training network over certain number of weight state transitions when performing a chosen training operation. See section §Apdx 1.5.1B for more details.

(2) Open Display Equaliser

This is to open the equaliser window which enables the user to see dynamically the graphs of the error change for each training pattern of a network during some training operations (see section §Apdx 1.5.1C for more details).

(3) Open TRAIN-DISPLAY-FRAME

This is to open the training display popups, to set some parameters and see dynamically the on-line messages of a training operation in a textual form. The messages help the user to trace operation results and final reports e.g. the number of the weight state transitions; or whether the network has been trained successfully (see section in §Apdx 1.5.1A for more details).

(4) Training Network

This is the option for setting and performing **training** on a user-designed network. Two main popups will come up in turn and help the user to set up all the training attributes necessary before any user-designed network can be trained.

(i) TRAIN-PATTERN POPUP

First of all, the training option brings up the TRAIN-PATTERN POPUP (Fig. Apdx 1.8b) which allows the user to specify the desired training patterns through a file interface provided by the simulator. This popup interface enables the simulator to receive a number of Input/Output training tuples specified by the user and perform training based on both a user-designed network and user-designed training tuples.


Fig. Apdx 1.8b The loading file popup

The popup works in the similar way to the LOAD TOPOLOGY POPUP discussed in section §Apdx 1.3.3A. Once the directory and filename of the file for the training patterns have been typed in by the user, an integrity check will be made by the simulator to see if the contents of the pattern file are compatible with the current network topology. If the contents are incompatible with the current network topology, an error message will be displayed on the screen. Otherwise the file will be loaded into the simulator, and two popups related to training will be brought up next. The format of the training pattern file will be specified in §Apdx 1.4.2. The three popups are described below.

(ii) TRAIN-DISPLAY-FRAME and other working windows

Once the training patterns have been loaded, the TRAIN-DISPLAY-FRAME will be opened. Another two display windows: DISPLAY-EQUALISER and DISPLAY-PLOTTER will also be opened automatically. But these two will be closed automatically if the chosen training regime is not going to use them during operation.

DISPLAY-EQUALISER and DISPLAY-PLOTTER allow the user to view the training dynamically. They will be discussed in §Apdx 1.5.1C and B.

The TRAIN-DISPLAY-FRAME is a main training window. On top of the frame, there is a specification menu which allows the user to tailor the training attributes to suit his needs before training, the lower section of

the TRAIN-DISPLAY-FRAME enables the user to see some data or training messages displayed during the **training** operation. For descriptions about the messages displayed within the section of the TRAIN-DISPLAY-FRAME, refer to §Apdx 1.5.1A.

The following text will specify each item in turn within the section of the specification in the TRAIN-DISPLAY-FRAME in order to help the user to set up the training parameters.

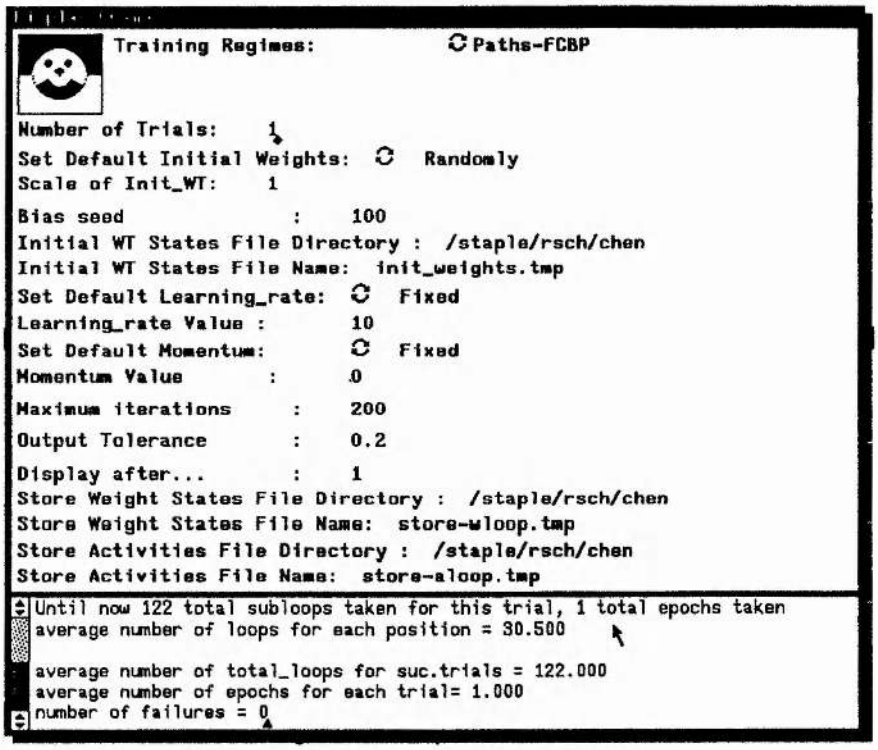



Fig. Apdx 1.9 The training display popup

(1) Training Regimes

There are six different training regimes available in the simulator. The user can choose one regime either by cycling through the options available or by clicking on the right regime with mouse.

The regimes provided here are all based on backpropagation. The direction and amount of each weight change in the weight state transition is estimated using gradient descent method. Each regime has though a different way to control the training. A brief introduction of each training regime is described below. More details about the regimes refer to Chapter 4 and Chapter 5.

• FCBP1

In the training regime FCBP1, one single weight state transition is made for each change in input training tuple as it is in standard backpropagation without momentum coefficient. Training continues until there is

a complete traverse of the training path with the associated weight state for each I/O training tuple having an error within a certain tolerance. A goal weight state path is then achieved.

The message of failed training will be displayed if the training is stopped when a critical value of the training time is exceeded.

• FCBP2

In the training regime FCBP2, the direction and amount of each weight change in the weight state transition is also estimated using standard back-propagation without momentum coefficient. The difference from FCBP1 is the rule of the weight state transition.

In FCBP2 when training an I/O tuple, consecutive weight state transitions are repeatedly made until either the error of the associated weight state is within a certain tolerance or the training is compulsorily terminated (the tuple cannot to be trained) if the training time is exceeded. It is the last weight state resulting from the sequence of weight state transitions made for an I/O training tuple that is taken for the training termination test.

Upon coming across a tuple along the training path that cannot be trained, the training will be stopped and a training failure message will be displayed on the TRAIN-DISPLAY-FRAME.

In the normal case, training under this regime continues until there is a complete travesal of the I/O tuples along the training path with low error.

• DBP

DBP stands for a discrete backpropagation regime for feedforward networks, it is a slight variation on the conventional backpropagation (Rumelhart *et al*, 1986) in the rule of the tuple error test: there are two I/O passes for each training tuple. It is the error of the second pass resulting from applying the current weight state made for the I/O tuple that is taken for the test.

• PATHS-FCBP

PATHS-FCBP is a regime very similar to the FCBP2 except it deals with training based on multiple training paths and at each position, DBP is applied to find a single goal weight state. If the user is interested in the details about the concepts of the multiple training paths, refer to the relevant section in Chapter 4.

• FCBP-SBP

This is a regime provided for finding a single goal weight state based on the PBP approach. FCBP-SBP denotes: feedforward continuous back-propagation for a single weight state. This is to use two different training and recognition sets to guide training. A regime like a multi-paths based FCBP1 is applied for each

training loop (and varies the training direction one after another loop) based on the training tuples in the training set until a single weight state is found according to the testing tuples in the recognition set or failure occurs. More details about this approach can be found in §6.5.

• RCBP

RCBP is the regime used for the recurrent network in the PBP approach. More details about the regime refer to the chapter 5.

(2) Number of Trials

This attribute allows the user to decide the number of the training trials and enables the user to do some operations in batch. Different training trials start from different initial weight states, initialised in a user-specified way, but all the other training parameters are the same. The default value of this attribute is 1, and a suggested number is 100.

(2) Bias Seed

This attribute allows the user to generate the same sequence of data which may be used as a set of randomly chosen initial weight values. This enables the user to have different experiments using the same randomly chosen initial weight values.

(4) Set Parameter for Initial-Weight states

This attribute allows the user to initiate the initial weight states for the training in batch ( For the training on a single trial, see section §Apdx 1.3.3F ). There are two types of options: randomly and user-defined. They are described as follow:

• Randomly

This means the simulator to set the initial weight state for each training trial. Each weight link within the network will be set randomly within a range which is decided by an attribute provided through the field: **scale**.

• User-Defined from a prepared file

This is to set the initial weight state in a user-defined way when performing the training in batch. Clearly in order to train more than one trial based on a network with the user-defined initial weight states, a file interface should be used. The specifications can be obtained through the file interface in a certain format. The directory and file name of the file should be provided through the following two fields: **File Directory of Initial-Weight states, File Name of Initial-Weight states.**

• User-Defined from screen setting

This is to set the initial weight state through screen setting. It is only valid when training is on a single trial. This is only to provide one more interface for the user to set up his initial weight.

(5) Scale of Initial weight

This is used to set the range of the initial weight values. Each weight link within the network will be set randomly within the range such as within (-0.1, 0.1).

(6a) File Directory of Initial-Weight states

This field allows the user to specify the a directory under which there is an external file containing all the user-defined initial weight states. The default directory is the current working directory.

(6b) File Name of Initial-Weight states

As described above, this field allows the user to specify the file name.

(7) Set Parameter for Learning rate

There are two different parameter settings available in the simulator. One is to set the parameter as FIXED and the other is as FUNCTION.

•FIXED

If the user wants to have a fixed *learning rate* set by himself before training and keep it fixed during the whole training procedure, then the user should choose this option and type in the real *learning rate* value in the very next attribute field LEARNING RATE-VALUE.

•FUNCTION

If the user wants to have a self-adaptive *learning rate* during training, he should choose this item. The parameter will be calculated through an associated function. Any values typed by the user in the LEARNING RATE-VALUE field will be discarded then. An algorithm to adapt the *learning rate* for any three layer feedforward networks, designed and implemented by Weir (Weir, 1990 ), has been compiled into the simulator as an example.

(8) Set Parameter for Momentum Coefficient

As for the learning rate setting, there are two different parameter settings available in the simulator. One is to set the parameter as FIXED and the other is FUNCTION.

•FIXED

If the user wants to have a user-decided *momentum* parameter, the user should choose this option and type in the *momentum* value into the very next attribute field MOMENTUM-VALUE.

•FUNCTION

In comparison to learning rate parameter, this is just a function interface in the simulator, no example function procedure itself has been implemented. It is an extension interface provided for the user to write his own self-adaptive procedure for the *momentum* attribute if it is necessary in the future. If he chooses this item, any values typed in the MOMENTUM-VALUE field will be discarded, and the *momentum* value will be calculated by a user-designed function.

(9) Output Tolerance

This is one of the training parameters which is used to decide the training accuracy. The default value is 0.2.

(10) Maximum Number of Weight state Transitions

This is a variable to decide when the training will be terminated. The number of the weight state transitions is a critical training attribute because a training to some networks, for various reasons, would last infeasibly. Using this parameter, if the network has not been trained by the time when the critical attribute is reached, the training can be aborted.

Note:

• It is impossible to say in general what a prior value is for this training parameter. It depends on the network being trained, the training regime being applied and the other training attributes being chosen. The user should choose this parameter with a great care, and not set it too small or too large.

• The parameter has different way to explain the critical training attribute for the different kinds of regime.

(i) In FCBP1, DBP or FCBP-SBP regime, this number controls the total number of loops. When training along one training loop has been completed, the total number of the weight state transitions of this loop is equal to the number of the training positions. Therefore the maximum number of the loops, which is equal to the number of the training positions divides the maximum number of the weight state transitions, will be used as the critical attribute.

(ii) For regime FCBP2, PATHS-CBP, RCBP: this number controls the number of weight transitions at each training position. The maximum number refers to the total weight state transitions for all the tuples at

each position. The number of the weight state transitions at each training position will be the number of the training positions divides the maximum number of the weight state transitions.

(11) Display after

This allows the user to control the display. The graphs can be display after every various number of weight transitions. In **cbptool**, the default number is one.

(12a) File Directory of Stored-Weight states

As described in (6a), this attribute allows the user to decide a directory under which an external file containing all learnt weight states will be saved.

(12b) File Name of Stored-Weight states

This attribute allows the user to decide the name of the external file mentioned above.

(13a) File Directory of Stored-Activations

Similar to (12a), this attribute allows the user to decide a directory under which an external file containing all learnt activation values will be saved. This operation is only valid when the RCBP training regime applied.

(13b) File Name of Stored-Activations

This attribute allows the user to decide the name of the external file mentioned in (13a).

Once all of the training attributes in the TRAIN-DISPLAY-FRAME have been set up, training operation can be initiated by clicking the top left icon within the popup. When started, the operation cannot be stopped until the network either is trained as desired or its critical training attribute (see **Maximum Number of Weight state Transition**) is reached. It is therefore very important to make sure that all the attributes set up are satisfactory before starting the operation.

A popup is provided for the user to decide in which way to run operations. There are only two options for this parameter: **BACKGROUND** and **FOREGROUND**. The default is **FOREGROUND**. If the user chooses **BACKGROUND** option, the operation will be performed in the background, and **cbptool** no longer occupies the Sun window system. Otherwise the chosen operation will be performed within the window environment. When the running ground has been chosen, operation starts.

When the training has been completed, it should be noticed that the weight state related to the network topology is not altered permanently by training. This means that the weight state of the network before

training is reinstated automatically when training is completed. This allows the user to run subsequent trials on exactly the same network, for example, to train on the same network with the same initial weight state but different other training attributes.

Note:

(i) The background facility is mainly provided for a user who wants to do a chosen operation in batch mode. This enables the user to save CPU expenses on graphics and window system when he wants to apply the simulator with intensive calculations.

(ii) If the chosen operation is running in the background, the user has actually released most resources of the simulator, and the user will automatically be forced to quit the simulator when the operation finished. The results can be either placed in an external file if the file interface has been applied before the operation or output directly onto the screen. If the user needs to use some other facilities of the simulator, he will need to access and run the simulator again.

**F. Performance Menu**

This menu contains three options for the **performance** on a trained network and displaying dynamically the data produced during the operation. (Fig. Apdx 1.10)

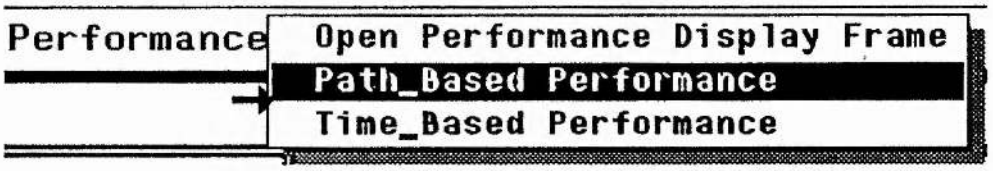



Fig. Apdx 1.10 The *Performance* menu

(1) Open Performance Display Frame

This facility is provided to open the display popup, set some operation parameters and see dynamically the on-line messages of the operation in a textual form. The messages help the user to trace operation results and final reports e.g. the number of the tuples which are generalized successfully. See section §Apdx 1.5.1A for more details.

Another two menus are for setting and performing two kinds of **performance** operation for a trained-network. One operation is based implicitly on the training time, it is a path-based performance. The other is based explicitly on the training time, it is a time-based performance. The both operations can be either a supervised or an unsupervised. If the user wants a supervised operation, the target output tuples should be included in the **performance-pattern specification file** (refer to §Apdx 1.4.3, §Apdx 1.4.4). The description about how these two performances differ from one another is presented below together with more details about the two operations.

(2) Path-Based performance

This is the option to do the **performance** along a training path. All patterns for the performance must be either trained patterns or the patterns close to one of the trained patterns along the training path. The first pattern of the performance must be a trained pattern. All patterns are provided by the user in a file served as the **performance-pattern specification file**.

The operation continues, while reading in a performance pattern and applying a suitable weight state which, is carried out through one of the generalization regimes, to calculate the actual network output until there is a complete travesal of the performance patterns.

(3) Time-Based Performance

This is the option to do the **performance** based on a real time. Each pattern together with an associated time is specified by the user. The pattern is not necessarily along any of the training paths, but simply on the trained space with real time.

The training time is needed for generating an associated weight state by the generalization regimes. This is a real index value of a training order based on the training position. For the trained tuples in the first training position, the value is 1.0; for those in the second position, the value is 2.0 and so on. The integral part of the value indicates the closest training position associated with the performance tuple, and the fractional part indicates the distance from the position. For example, if a performance tuple is halfway between the training positions 2 and 3, its training time will be 2.5.

Whatever the option chosen by the user, the following two main popups will come up in turn and help the user to set up all the necessary attributes before **performance**.

(i) LOAD PERFORMANCE-PATTERN POPUP

Similar to the **training** operation, **performance** options bring up the LOAD PERFORMANCE-PATTERN POPUP (Fig. Apdx 1.11) which allows the user to load in the desired patterns through a file interface.

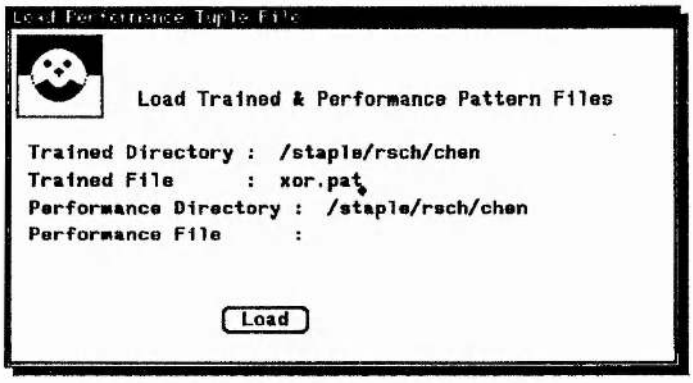



Fig. Apdx 1.11 The loading popup for generalization

This enables the user to specify a number of performance tuples to the simulator. The popup works in a similar way to the LOAD TOPOLOGY POPUP discussed in section §Apdx 1.3.3A. Once the directories and filenames of the training pattern file and the performance pattern file have been typed in by the user, an integrity check will be made to see if the contents in the two pattern files are compatible with each other, with the current network topology. A check is also needed to ensure if the file is compatible with itself. An incompatible of example is incompatibility: when the user wants to do a supervised operation but there are no output tuples in the file. If the contents of the loaded pattern files are incompatible with the topology, the training patterns or the context, an error message will be displayed on the screen. Otherwise the file will be loaded into the simulator, and the popup PERFORMANCE-DISPLAY-FRAME will be brought up next. The format of the performance pattern file will be described in §Apdx 1.4.3 and §Apdx 1.4.4.

(ii) PERFORMANCE-DISPLAY-FRAME

Once the performance patterns have been loaded, the PERFORMANCE-DISPLAY-FRAME will be opened (Fig. Apdx 1.12). This is a main conversation window for the **performance** operation. On top of the frame, there is a specification menu which allows the user to tailor the operation attributes, the lower section of the PERFORMANCE-DISPLAY-FRAME enables the user to see some data or operation messages displayed during the performing. The following text will discuss each item in turn within the section of the specification menu to help the user to set up the operation parameters, the descriptions about the data and operation messages displayed within the lower section of the frame refer to §Apdx 1.5.1A.

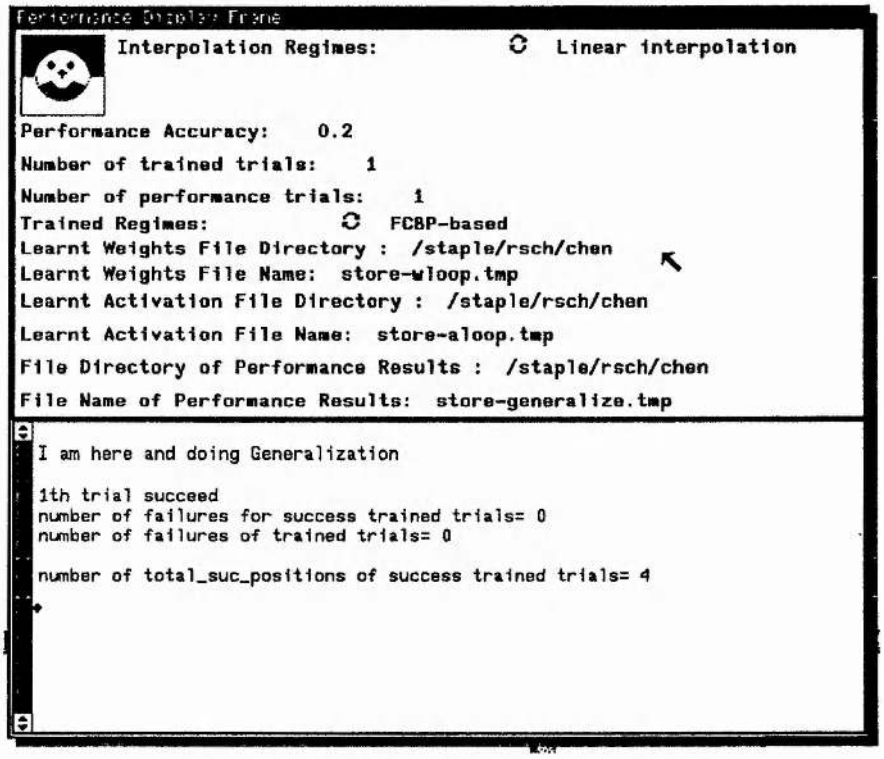



Fig. Apdx 1.12 The generalization disply popup

(1) Interpolation Regimes

There are three functions for three different generalization regimes in the simulator. The user can choose any regime either by cycling through the options available or by clicking on the right regime with mouse. The default regime is the linear interpolation: LINEAR-INTERPOLATION. The interfaces of the other two functions have been reserved for the future implementation of the associated procedures.

• LINEAR-INTERPOLATION

LINEAR-INTERPOLATION refers to generalization based on the linear interpolation technique. This regime means that the generalization to the untrained patterns is based on a linear interpolation of the two neighbouring weight states of two trained patterns along the training and generalization path. The definition of the neighbourhood is described above.

• NEIGHBOURHOOD

NEIGHBOURHOOD regime is one of generalization regimes. The interface of this procedure is implemented but the procedure itself is for future implementation.

• FOURIER-INTERPOLATION

FOURIER-INTERPOLATION refers to generalization based on the Fourier analysis on the learnt weight states. The interface of this procedure is implemented but the procedure itself is for future implementation.

(2) Performance Accuracy

This is a parameter associated with the training tolerance, used to decide the accuracy of individual activities in a supervised performance. The default value is 0.2.

(3) Number of Trained Trials

If the network has been trained in batch, a **performance** operation in batch is then possible. This parameter helps the simulator to decide what the maximum number of the operation trials is allowed if the user wants to do **performance** in batch.

(4) Number of the Performance Trials

This specifies the number of the trials in batch. The valid number of the performance trials must be less or equal to that of the trained trials.

(5) Trained Regimes

There are three types of regimes. Again the user can choose a regime either by cycling through the options available or clicking on the right regime with the mouse.

• FCBP-BASED

FCBP-BASED stands for the performance based on the two FCBP-based training.

This regime means that the performance is based on a weight state path. The performance for the patterns will be done by applying the associated weight state based on the learnt path which has been obtained by applying one of those continuous backpropagation training regimes.

• DBP-BASED

SBP-BASED stands for the performance based on the Discrete BackPropagation training.

This regime means that the performance is based on a single weight state. The operation patterns will be performed by applying a single learnt weight state which has been obtained through training.

• RCBP-BASED

RCBP-BASED stands for the performance based on the RCBP training.

This regime means that the performance is based on both a weight state path and a activity path. The linear interpolation technique is then applied to both the learnt weight state and activity state paths.

(6) File Directory of Learnt-Weight states

This enables the user to specify the directory of the file which contains all the learnt weight states of a trained network. It is those weight states that enable the user to do the performance.

(7) File Name of Learnt-Weight states

Similar to (6), this allows the user to specify the name of the file.

(8) File Directory of Learnt-Activity states

This enables the user to specify the directory of the file which contains all the learnt activity states of a trained network.

(9) File Name of Learnt-Activity states

Similar to (8), this allows the user to specify the name of the file.

(10) File Directory of Performance-Results

This allows the user to specify the directory of the file which will contain all performance results of the chosen operation: the actual network input and output values generated by the **performance** operation.

(11) File Name of Performance-Results

This allows the user to specify the name of the file described above.

Once all of the attributes in the PERFORMANCE-DISPLAY-FRAME have been set up, the operation can be initiated by clicking the top left icon within the PERFORMANCE-DISPLAY-FRAME. A popup will be provided to set the running ground (this is the same as that in §Apdx 1.3.3E). When started, the operation cannot be stopped until it has completed. Therefore it is very important to make sure that all the attributes set up are correctly before starting any operation.

### G. Quit Menu

This menu (rather a button) provides for the user to exit the simulator (Fig. Apdx 1.13).

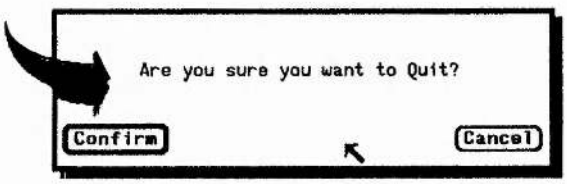



Fig. Apdx 1.13 The *Quit* menu

When this menu is selected, a QUIT POPUP will be displayed. If the user wishes to remain in the simulator at this stage, the cancel button should be selected, otherwise confirm the operation by clicking on the confirm button.

## 4. Forms of the Specification Files

Four different specification files are used in the simulator as file interfaces for **training** and **performance** operations. As discussed earlier, the file formats used in the simulator depend on the Sun operating system and are very rigid. If the format is not adhered to, the simulator will not complain but will try and operate on the erroneous pattern values, it will totally nullify the results of the operations. Therefore carefully and correctly specifying the working files to the simulator is a primary condition to be met for running any simulator operations. The following sections will describe the formats for the four files: initial-weight state file, training-pattern file, path-based-performance-pattern file, and time-based-performance-pattern file.

4.1 INITIAL-WEIGHT STATE SPECIFICATION FILE

This is the interface that enables the user to do training based on a set of user-designed initial weight states.

As an example, a corresponding specification file, formatted for setting the initial weight states for two training trials in a designed network, would look like the following:

```
w31, w32, w36,
w41, w46,
w53, w54, w56,

0.10012, 0.21000, 0.70021,
0.08001, 0.19251,
0.01003, 0.52009, 0.11027,
```

Fig. Apdx 1.14a The specification file

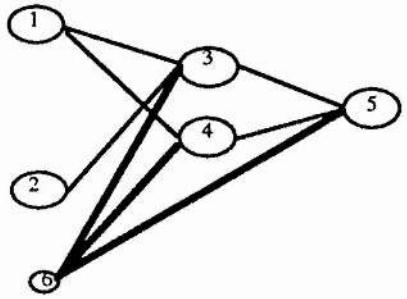



Fig. Apdx 1.14b The user-designed network

Fig. Apdx 1.14a shows the information about the specification file for the user-designed network topology shown in Fig. Apdx 1.14b: the two groups information correspond to two training trials. The contents of the first group information is an example to show the format and the details of each item for one trial, the second group is as a real example for setting initial weight state for a training trial. The new-line in the file is represented by a short horizontal dashed line.

The file format can be described in text as follows:

(1) For each training trial, There are a single new line and a number of **lists** associated with it in the specification file.

(2) The number of the **lists** depends on the total number of the hidden and output neurons related to the user-designed network. The order of the **lists** is decided by the neurons' ID.

(3) The length of each **list** depends on the number of the incoming links to the neuron. The items of the **list** are the weight strengths of associated incoming links to the current neuron, ordered according to the associated neuron ID related to the incoming links, and separated by comma.

4.2 TRAINING-PATTERN SPECIFICATION FILE

This is the file which enables the user to do training based on a set of user-designed training I/O tuples.

```
Number_of_I/O_Tuples:    4
Number_of_I/O_Inputs:    2
Number_of_I/O_Outputs:   2
Number_of_Paths:         2
---
Path_1
Inputs
0.000, 0.000, 1.000, 1.000,
0.000, 1.000, 0.000, 1.000,
Outputs
0.100, 0.900, 0.900, 0.100,
0.100, 0.900, 0.900, 0.100,
---
Path_2
Inputs
1.000, 1.000, 0.000, 0.000,
1.000, 0.000, 1.000, 0.000,
Outputs
0.100, 0.900, 0.900, 0.100,
0.100, 0.900, 0.900, 0.100,
```

Fig. Apdx 1.15a The specification file

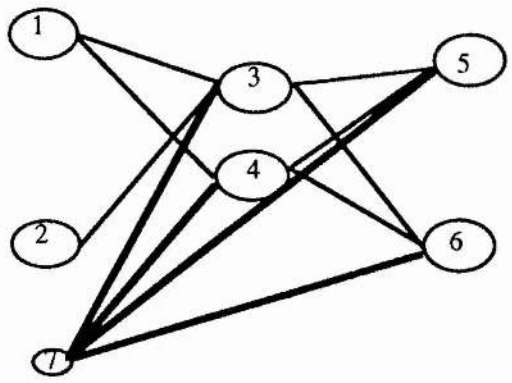



Fig. Apdx 1.15b The network topology

The above picture Fig. Apdx 1.15a shows a real example of the specification file for specifying four training tuples for the user-designed network topology shown in Fig. Apdx 1.15b. The new-line in the file is represented by a short horizontal dashed line. The two groups correspond to two training paths.

In a general case, there are seven parts in the specification file in order to specify the training. More details are given below.

(1) Basic information. This part provides the basic information for the training. There are four items, each item is specified in an individual line in the file. The form is all the same for the four lines: a string follows an integer number. The specification string starts from the first column and must not include any space among the string; at least one space is needed to separate the number and the string. The four numbers specify:

- The number of the I/O training tuples.
- The number of the input neurons in the network.
- The number of the output neurons in the network.
- The number of the training paths in the training.

(2) A space line. This is a space line in the file, used to separate the basic information described above and the others in the file.

(3) A Path prompter. This is a special line in the file. It is used as a prompter to specify the number of the path in the file.

(4) An Input prompter. This is another used line in the file. It can be used as a prompter to specify the input tuples which will be provided next in the file.

(5) Specify the input tuples.

This part consists of as many **lists** as the number of the input neurons. The order of the **lists** depends on the input neuron's ID. Each **list** is a line in the file.

The items in each **list** are the real input values of the training tuples, ordered according to the order of the training tuples. The number of the items in each **list** depends on the number of the training I/O tuples. All items in each **list** are separated by comma.

(6) An Output prompter. This is another used line as a prompter to specify the output tuples will be provided next in the file.

(7) Specify the output tuples.

Very similar to the Part (5), this part provides the information for the output tuples, consists of as many **lists** as the number of the output neurons. The order of the **lists** depends on the output neuron's ID. Each **list** is a line in the file.

The items in each **list** are the real target output values of the training tuples, ordered according to the order of the training tuples. The number of the items in each **list** depends on the number of the training I/O tuples. All items in each **list** are separated by comma.

Both an example and description of each part of the specification file have been given above. It is clear that the part (3) to part (7) associate with only a single training path in the file. If a ***group*** is defined and referred to the part(3)-part(7), the number of the ***group*** depends on the number of the training paths. Therefore each training specification file consists of the part (1), part(2) and a number of the ***group***.

### 4.3 PATH-BASED-PERFORMANCE-PATTERN SPECIFICATION FILE

This file enables the user to do **performance** based on a set of user-designed performance tuples. All the tuples provided by the user must be along a training path. Both the format and the contents of the file provided by the user are very similar to those of the Training-Pattern Specification file.

Here is an example, the formatted file enables the user do an unsupervised performance on a trained network based on the second training path for eight performance tuples. The corresponding specification file appears like the following:

```
Number_of_I/O_Tuples:   8
Number_of_Inputs:       2
Number_of_Outputs:      2
Paths_no.:              2
-----
Performance_Tuples:
1.0,1.0,1.0,0.5,0.0,0.0,0.0,0.5,
1.0,0.5,0.0,0.0,0.0,0.0,0.5,1.0,1.0,
Outputs:
0.8, 0.8, 0.8, 0.8, 0.8, 0.8, 0.8, 0.8,
0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2,
```

Fig. Apdx 1.16a The specification file

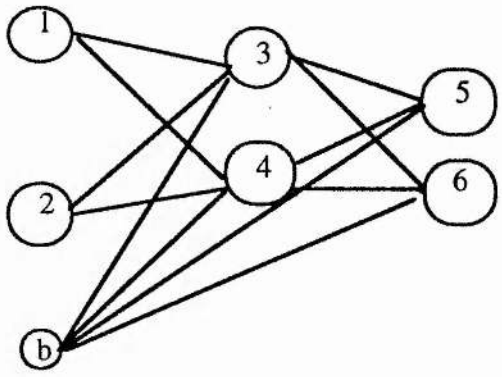



Fig. Apdx 1.16b The user-designed network

Fig. Apdx 1.16a shows the real example of the specification file for the user-designed network topology shown in Fig. Apdx 1.16b. The new-line in the file is represented by a set of short horizontal dashed line.

In general, there are four or six parts (depending upon if it is a supervised operation) in the specification file in order to specify the **performance**. This is listed as follow:

(1) Basic information. This part provides a set of basic information for the operation. There are four items, each item is specified in an individual line in the file in the same form as that described in Apdx 1.4.2 (1). The numbers represent:

• The number of the performance tuples;

• The number of the input neurons in the network;

• Information about the targets: if this is a supervised performance, the value of this field should be the number of the output neurons, otherwise should be zero.

• The number of the path (it starts from the Xth column): this value indicates the operation is based on which training path.

(2) A space line. This is a space line in the file, used to separate the basic information described above and the others in the file.

(3) A Performance prompter. This is used as a prompter.

(4) Specify the performance tuples. This part is very similar to the part(5) in the ***training-pattern-specification file***, it is used to specify the performance tuples.

(5) An Target prompter. This is another prompter line.

(6) Specify the target tuples. This part is very similar to the part (7) in the ***training-pattern-specification file***, and is used to specify the target tuples when performing a supervised performance.

Note:

There are two main different points comparing this specification file with the ***training-pattern-specification file***:

• As each performance is only based on a single training path, there is no more than one ***group*** defined in the ***training-pattern-specification file***.

• In the performance specification file, the part (5) and part (6) are only needed when the user wants to do a supervised performance and provide the targets to the output neurons. This is implied by representing the real number of the output neurons in the third line of the part (1). When doing a unsupervised performance, this value is set to zero, part(5) and part(6) do not exist then.

## 4.4 TIME-BASED-PERFORMANCE-PATTERN SPECIFICATION FILE

This file enables the user to do performance based on a set of user-designed performance tuples together with the associated training time. Both the format and the contents of the file provided by the user are similar to those of the ***path-based-performance-pattern-specification file***.

As an example, a file has been formatted in order to ensure the user to do a supervised performance on a trained network which has two input, two output neurons, with eight performance tuples chosen from a training surface and each one together with a training time specification. The corresponding specification file appears like the one shown in Fig. Apdx 1.17a below.

```
Number_of_I/O_Tuples:    8
Number_of_Inputs:        2
Number_of_Outputs:       2
——
Performance_Tuples
1.0, 1.0,1.0, 0.5, 0.0, 0.0, 0.0, 0.5,
1.0, 0.5,0.0, 0.0, 0.0, 0.5, 1.0, 1.0,
Training-Times
1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5,
Outputs
0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2,
0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2,
```

Fig. Apdx 1.17a The specification File

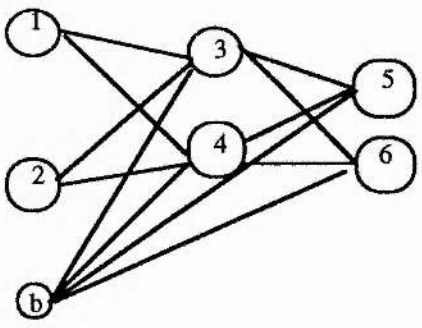



Fig. Apdx 1.17b The user-designed network

Fig. Apdx 1.17a shows a real example of the specification file for the user-designed network topology shown in Fig. Apdx 1.17b. The new-line in the file is represented by a set of short horizontal dashed line.

In general case, there are six or eight parts (depending upon if it is a supervised operation) in the specification file in order to specify the **performance**. More details can be described below.

(1) Basic information. Except the last item in the part (1) of the ***training-pattern-specification file***, the other three items in this part are the same as those in the ***training-pattern-specification file*** :

• The number of the performance tuples.

• The number of the input neurons in the network.

• Information about the targets.

(2) ~ (4) Refer to the associated parts in the ***path-based-performance-pattern-specification file.***

(5) A Training_Time prompt.

(6) Specify the training_time of the tuples. This is similar to the part (7) in the ***training-pattern-specification file***, but it is used to specify the training time of the associated performance tuple.

(7) A Target prompt. This is a line as a prompter.

(8) Specify the Target tuples. This part is very similar to the part (7) in the ***path-based-performance-pattern-specification file,*** it is used to specify the target tuples when performing a supervised **performance.**

## 5. Display and Store the Results

This section describes the facilities used to display, store the internal or final operation results and general messages. This also explains to the user how and in what forms to use the facilities.

### 5.1 DISPLAYS

As a whole, four types of display subwindows have been provided in the simulator. Each serves as a separate specific function. Three of them are to display the outputs generated by the network in graphics. The other one is to display general messages. They are: Display Frame, Display Plotter, Display Equaliser and General Message Popup. The following text describes the features of each facility in detail.

### A. Display Frame

In **cbptool**, both operation: **training** and **performance** have their own display frame popup. There are two functions for each of the display frame popups. The first function of the popups is to set up the operation parameters, which has been discussed in section §Apdx 1.3.3E and §Apdx 1.3.3F. Here the second function of the popups is discussed which is to display some operation information generated by the two operations in a textual form. They are shown in the lower section of the frames (see Fig. Apdx 1.9).

In the **training**, the text information consists of the information about each of the training tuples. For example, the number of the weight state transition taken or the failure information for the trail. Once the operation has been completed, general information is displayed to report the performance of the operation. For example, the number of the successfully trained trails, the failure number, the average weight state transitions. etc.

In the **performance**, the information shows that the **performance** results (see the lower section of the frames shown in Fig. Apdx 1.12). For example, the number of the successful generalized trials.

All text shown in the second section of the training or performance **FRAME** will be saved into the external files: ***train.display.tmp**** or ***perform.display.tmp**** respectively for further perusal when the user quits the simulator. The * here denotes an integer number generated according to the number of the current procedure. This implies that the user will get different file name when he/she executes the operations each time. The exact file name of the current procedure is displayed in the Suntool window.

The display frames can be closed as any SunView windows: moving the mouse pointer to the window label, pressing the righthand mouse button and selecting the Done option. The frames can be re-displayed by either choosing the **open display frame** option in the **Training** or **Performance** Menu, or performing the associated operation.

### B. Display Plotter

Fig. Apdx 1.18a and Fig. Apdx 1.18b show the picture of Display Plotter and Display Equaliser respectively. The Plotter popup window is provided to plot a graph of the error changing dynamically against a certain number of the weight state transitions in the **training**. Only three training regimes: **FCBP1, DBP,** and **FCBP2** have been supported to plot the graph during the operation.

The graph is drawn on an extensible window, the horizontal width of the window can be larger than the actual physical window displayed, hence there is a scrollbar along the bottom. During the **training**, the plotter will automatically track the graph as it is drawn, allowing the user to view the progress of the dynamically changing of the training on the network.

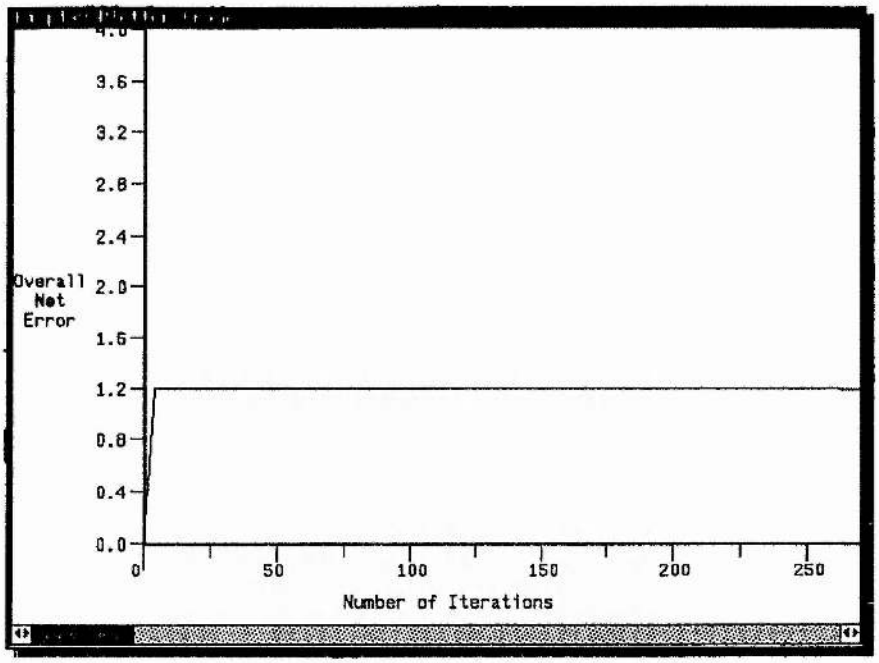



Fig. Apdx 1.18a The Plotter window

Both of the plotter axes are calibrated; a number is shown along the horizontal axis every certain number of the weight state transitions and the error is shown along the vertical axis. More descriptions on the two axes scale follow:

• Each scale represents two different number of the weight state transitions in the three kinds of the training regimes:

(1) X-calibration for regime **FCBP1** and **DBP**

For **FCBP1** and **DBP**, one scale in the x-axis is one training loop. This means that between any two adjacent scales, the number of the weight state transitions is equivalent to the number of the training tuples along one training path.

(2) X-calibration for regime **FCBP2**

For regime **FCBP2**, one scale in the x-axis is one weight state transition.

• The error value shown in the y-axis also have two different meanings for three kinds of the training regimes:

(1) Y-calibration for regime **FCBP1** and **DBP**

For **FCBP1** or **DBP**, the error values shown in the y-axis are the overall network errors. Therefore the y-axis calibration is scaled depending on the total number of tuples being presented to the network for training.

(2) Y-calibration for regime **FCBP2**

For **FCBP2**, the error values shown in the y-axis are the errors of the current training tuple. Therefore the y-axis is calibrated between 0.0 and 1.0 as any individual pattern error will never exceed 1.0.

Similar to the display-frames, the **Display Plotter** can be closed just as it is done on any SunView windows, and can be re-displayed by choosing the **Open Display Plotter** option in the **Training** Menu and cleared by selecting the **Clear Plotter Canvas** option in the **View** Menu.

### C. Display Equaliser

This popup window (Fig. Apdx 1.18b) is provided to display dynamically the changing error of individual training pattern.

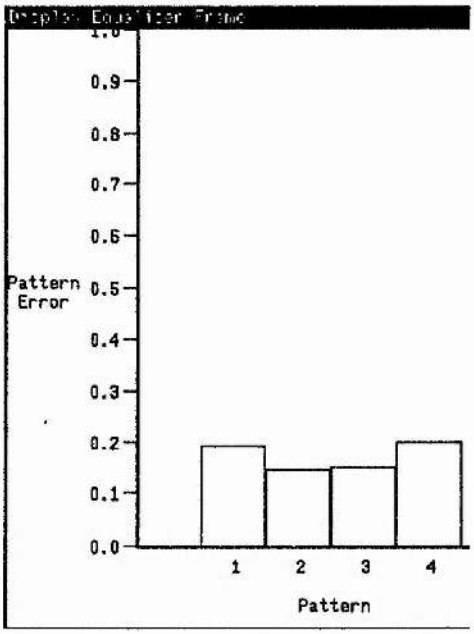



Fig. Apdx 1.18b The Equaliser window

Each pattern error is displayed in the form of a bar graph which changes size dynamically on the screen, somewhat similar to a graphic equaliser, hence been named as **Display Equaliser**.

Similar to the **Display Plotter**, only two training regimes: **FCBP1** and **DBP** have been supported to plot the graph during **training**. For both regimes, the **equaliser** shows the changing error of individual pattern once a training loop.

Again, both of the plotter axes are calibrated, the number of the bars associated with the tuples presented to the network will be shown on the horizontal axis and the individual pattern error shown on the vertical axis. It is noticed that the equaliser does not need scrollable along the vertical direction and the vertical axis can be calibrated between 0.0 and 1.0 as any individual pattern error will never exceed 1.0. Because the number of the patterns presented to the network is determined at running time, it is necessary to make the equaliser scrollable along the horizontal, the graph is drawn on a horizontal window larger than the actual physical window displayed. Thus if there are more patterns than can be displayed in the physical window, the user can scroll along the horizontal window to view all of the bar graphs.

Very similar mechanisms can be applied in **Display Plotter** to close, re-display and clear the **Display Equaliser**. Following in Fig. Apdx 1.19 is an overview picture of three windows in training: plotter, equaliser and training display frame.

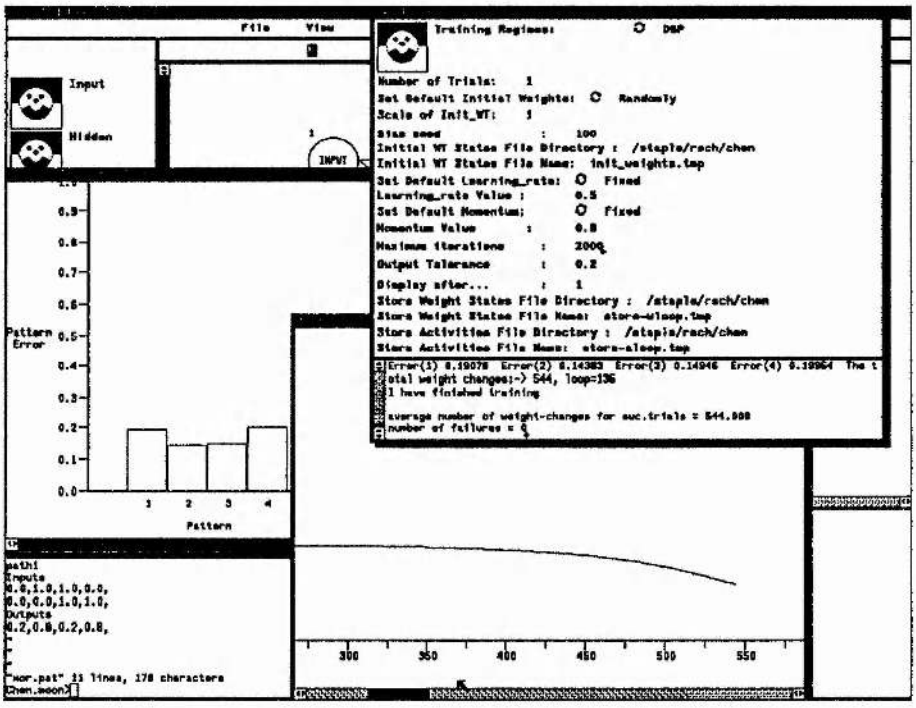



Fig. Apdx 1.19 The Plotter, Equaliser and training frame windows

### D. General Message Popup

The last type of popup window is the **General Message Popup** window which is used for displaying warning or error messages. The message is always displayed at the centre of the popup, when this happens,

processing cannot continue as normal unless the popup is closed by clicking on the warning icon. An example is shown below.

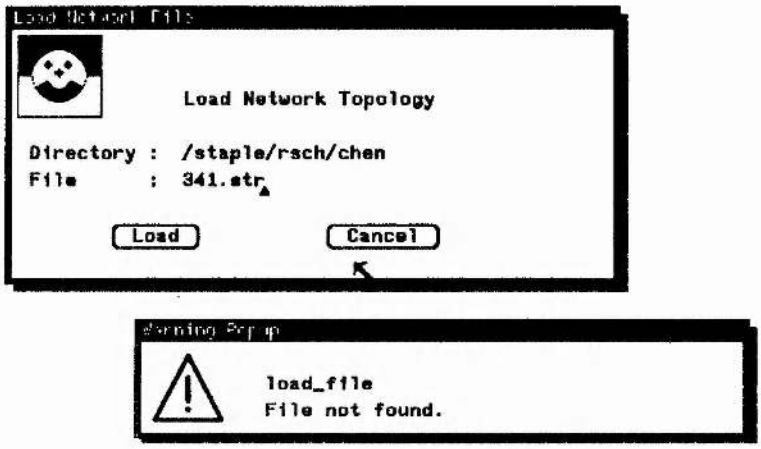



Fig. Apdx 1.20 A warning popup for a loading file operation

5.2 STORE INTO FILES

This section discusses the forms of files which used to store operation results in order to help the user to interpret the information recorded in a textual form.

Four types of information are stored into external files: the text display part of the display frames; the learnt weight states or the learnt weight sequence; the learnt activation values; and the performing inputs and outputs in the performance operation. The following text will describe all these files in detail.

**A. Store the Display Text**

Two designed working files: ***train.display.tmp**** and ***perform.display.tmp**** are used to record the text displayed on the two related display frames: **TRAIN-DISPLAY-FRAME** and **PERFORMANCE-DISPLAY-FRAME**. This enables the user to further perusal the information after the **training** or **performance**.

The formats of these two files are the same as what have been displayed in the display-frames. The user can easily interpret the information.

**B. Store the Learnt Weight-Sates**

Learnt weight state is one of the most important training results, it is not only needed by the **performance**, but also helps the user to further analyse the **training**.

As mentioned before, each set of the learnt weight states is the results of one successfully trained trial. The number of the learnt weight states is equal to the number of the training positions along one or several chosen training path(s). All these weight states can be recorded in a user-defined external file.

The form of the recording is very similar to the form of the file for setting initial weight states through the **Initial-Weight state Specification file** described in section §Apdx 1.4.1. There are only two different aspects in the two files: the number of the **groups** for each trial is different; one additional line is needed in the **Learnt-Weight states file**. These will be described below.

(1) The number of the **states** for each training trial is different in the both files.

• In the Initial-Weight state Specification file, for each training trial, there is only reserved line and one **group** consisting of a number of **lists**.

• In the Learnt-Weight states file, each training trial no longer corresponds to only a single **group** defined above. The number of the **groups** for each training trial is equal to the number of the training positions. Each **group** still starts from one and only one new-line for either a new training trial or a new training position.

(2) The additional line.

Compared to the **Initial-Weight state Specification file**, each **Learnt-Weight states file** has an additional line at the bottom of the file to tell the user which trial has been trained successfully.

The length of the line depends on the number of the training trials. In other words, the number of the items in the line separated by comma is the same as the number of the training trials. Each item value is either 1(successful) or 0 to tell whether the related trial has been successfully trained or not.

### C. Store the Learnt-Activation-Values

Besides the learnt weight states, the learnt activation-values is another operation result to be stored when the **RCBP** training is performed. Together with this information, the user is able to further analyse the **RCBP** operation and perform the generalization operation.

The learnt activation-values are also corresponding to successfully trained trials. All these activation values are recorded in a user-defined external file in a simple format similar to the learnt weight states file without the additional line. The format can be described as: for each learnt trail, there is a **package** of activition values. Each **package** consists of as many **groups** as there are I/O training positions. Each of the **group** has as many **lists** as there are I/O paths. Each of the **list** starts with a new-line and follows a line

of information. The information provide the activition values associated with each unit except the input units in the units' ID order and separated by a comma.

**D. Store the Performance Outputs**

This is another user-defined external file to keep all the I/O dynamic information stored in a textual form for the **performance** operation. In other words, the related operation results can be either observed through the display-frame to get some general information or to get more details through the file interface.

The form of the files is simple. It consists of a number of lines with an equal length. The number of the lines is dependent on the number of the performing tuples. Each line represents a pair of the I/O information of the **performance**: '$X_{i1}$, $X_{i2}$, ... $X_{in}$, $O_{i1}$, $O_{i2}$, ... $O_{im}$,' where $X_{ij}$ ($j$=1, 2... $n$) denotes for performing tuple $i$ the real value of the $j_{th}$ input neuron and $O_{ij}$ ($j$=1, 2... $m$) denotes for performing tuple $i$ the real network output value of the $j_{th}$ output neuron.

# APPENDIX 2

## EXAMPLE: THE OR PROBLEM

Following is an example which shows the role of hidden units in the two kinds of particular I/O mappings. This shows the similarity and difference between finding a solution weight state in the two particular cases.

### The OR problem

The requirement of hidden units can be seen through the Boolean **OR** function in two different types of particular mappings in a sigmoid-unit single layer network.

1) A binary output mapping OR

The network (Fig. Apdx 2.1) consists of two input units $\mathbf{U_1}$ and $\mathbf{U_2}$, and one output unit $\mathbf{U_3}$.

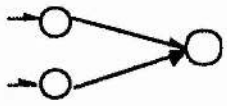


Fig. Apdx 2.1 The network topology

For the topology, the following matrix equation: $\mathbf{A*W = X}$ shows the set of excitation equations that needs to be satisfied simultaneously to achieve **OR** using a single weight state **W**:

$$\begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} w_{31} \\ w_{32} \end{bmatrix} = \begin{bmatrix} t_{31} \\ t_{32} \\ t_{33} \\ t_{34} \end{bmatrix} \qquad \text{(Apdx 2.1)}$$

where **A** is the input activity matrix and **X** is an excitation matrix of the output unit $\mathbf{U_3}$.

For binary output mappings, the target excitation values for the four I/O mapping patterns are $t_{31} \in [0,\theta]$, $t_{32} \in (\theta,1]$, $t_{33} \in (\theta,1]$ and $t_{34} \in (\theta,1]$ respectively. The output unit is turned on when the excitation $t_{ip}$ is above the threshold $\theta$ or is turned off otherwise. The derived inequalities $w_{32} > \theta$ in the second row, $w_{31} > \theta$ in the third row, and $w_{31}+w_{32} \geq \theta$ in the fourth row, yield a solution leading successful learning possible.

The above conclusion can also be deduced through the set of reduced excitation equations in a matrix form. Suppose $\theta$ = 1/3; Eq.Apdx 2.1 can be associated with a matrix equation with a set of chosen target excitation values which satisfies the OR binary mapping condition:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} w_{31} \\ w_{32} \end{bmatrix} = \begin{bmatrix} 1/2 \\ 1/2 \end{bmatrix} \text{ or } Ax=b\text{; Augment matrix is } C \text{ denoted in Eq.(3.3)} \qquad \text{(Apdx 2.2)}$$

It is clear in Eq.(Apdx 2.2) r($A$)=r($C$)=2, there are two weight variables too, so the weight state exists.

2) An analogue output mapping OR

Now let us see if a solution weight can also be found for an analogue exact value OR mappings. Suppose the target excitation values for the four I/O mapping patterns are $t_{31}$ =0.0; $t_{32}$ =1.0; $t_{33}$ =1.0; and $t_{34}$=1.0. It can be seen through solving the set of reduced equations in Eq.(Apdx 2.1) with the specific set of target excitation values in a matrix form:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} w_{31} \\ w_{32} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \text{ or } Ax=b\text{; Augment matrix is } C \text{ denoted in Eq.(3.3)} \qquad \text{(Apdx 2.3)}$$

This derived equations $w_{32}$=1 in the first row, $w_{31}$=1 in the second row, and $w_{31}+w_{32}$=1 in the third row, yield a contradiction leading successful learning impossible.

The result can also be deduced through checking the ranks of the matrix $A$ and $C$ in Eq.(Apdx 2.3). It can be seen that r($A$) is 2; r($C$) is 3; there are three equations in the reduced equations set with only two weight variables $w_1$ and $w_2$. From a mathematical point of view, in order to find a weight state as the solution of the equations set, more independent weight variable terms are needed, otherwise the set of weights does not exist.

This analogue output OR problem shows that sometimes analogue value mappings may need hidden units where binary mappings do not.

From the OR problems discussed above, one example has suggested that the number of hidden units needed in binary-analogue mappings is more than that needed in the binary-binary mappings when the same number of I/O patterns is based on.

## APPENDIX 3

## THE DATA: THE TRAINING DATA OF THE ECGS

| mkw | lhc | v w | dyh |
|---|---|---|---|
| 7 | 3 | 4 | 2 |
| 6 | 3 | 4 | 2 |
| 5 | 3.5 | 4 | 2 |
| 13.5 | 3.6 | 4 | 2 |
| 3 | 2.8 | 4.2 | 2 |
| 5 | 2.7 | 4.3 | 2 |
| 5.2 | 2.6 | 3.7 | 2.5 |
| 5.5 | 1.5 | 3.5 | 2 |
| 6 | 18 | 3 | 2 |
| 8 | 3 | 10 | 2 |
| 7 | 3 | 3.1 | 2 |
| 5.7 | 3 | 3.2 | 9 |
| 5.5 | 3 | 3.2 | 2 |
| 5.2 | 4 | 3.2 | 1.9 |
| 5.8 | 5 | 3.3 | 1.7 |
| 6 | 4 | 3.5 | 1.5 |
| 5 | 3 | 4 | 1.8 |
| 4.5 | 3 | 4 | 2.1 |
| 4.2 | 3 | 4 | 2.5 |
| 13 | 3 | 4 | 2.7 |
| 1.5 | 3 | 4 | 2.5 |
| 3.5 | 3 | 4 | 2 |
| 4.1 | 3 | 4 | 1.8 |
| 4.2 | 3 | 4 | 1.8 |
| 5 | 3 | 4 | 1.8 |
| 7 | 3 | 4 | 1.8 |
| 5.5 | 3.8 | 4 | 1.5 |

Apdx3.1 The raw chosen data

| log(mkw) | log(lhc) | log(vw) | log(dyh) |
|---|---|---|---|
| 0.845 | 0.477 | 0.602 | 0.301 |
| 0.778 | 0.477 | 0.602 | 0.301 |
| 0.699 | 0.544 | 0.602 | 0.301 |
| 1.130 | 0.556 | 0.602 | 0.301 |
| 0.477 | 0.447 | 0.623 | 0.301 |
| 0.699 | 0.431 | 0.633 | 0.301 |
| 0.716 | 0.415 | 0.568 | 0.398 |
| 0.740 | 0.176 | 0.544 | 0.301 |
| 0.778 | 1.255 | 0.477 | 0.301 |
| 0.903 | 0.477 | 1.000 | 0.301 |
| 0.845 | 0.477 | 0.491 | 0.301 |
| 0.756 | 0.477 | 0.505 | 0.954 |
| 0.740 | 0.477 | 0.505 | 0.301 |
| 0.716 | 0.602 | 0.505 | 0.279 |
| 0.763 | 0.699 | 0.519 | 0.230 |
| 0.778 | 0.602 | 0.544 | 0.176 |
| 0.699 | 0.477 | 0.602 | 0.255 |
| 0.653 | 0.477 | 0.602 | 0.322 |
| 0.623 | 0.477 | 0.602 | 0.398 |
| 1.114 | 0.477 | 0.602 | 0.431 |
| 0.176 | 0.477 | 0.602 | 0.398 |
| 0.544 | 0.477 | 0.602 | 0.301 |
| 0.613 | 0.477 | 0.602 | 0.255 |
| 0.623 | 0.477 | 0.602 | 0.255 |
| 0.699 | 0.477 | 0.602 | 0.255 |
| 0.845 | 0.477 | 0.602 | 0.255 |
| 0.740 | 0.580 | 0.602 | 0.176 |

Apdx3.2 The log(x) of each chosen value $x$

| n1(mkw) | n1(lhc) | n1(vw) | n1(dyh) |
|---|---|---|---|
| 4.155 | -2.523 | 3.398 | -1.699 |
| 4.222 | -2.523 | 3.398 | -1.699 |
| 4.301 | -2.456 | 3.398 | -1.699 |
| 3.870 | -2.444 | 3.398 | -1.699 |
| 4.523 | -2.553 | 3.377 | -1.699 |
| 4.301 | -2.569 | 3.367 | -1.699 |
| 4.284 | -2.585 | 3.432 | -1.602 |
| 4.260 | -2.824 | 3.456 | -1.699 |
| 4.222 | -1.745 | 3.523 | -1.699 |
| 4.097 | -2.523 | 3.000 | -1.699 |
| 4.155 | -2.523 | 3.509 | -1.699 |
| 4.244 | -2.523 | 3.495 | -1.046 |
| 4.260 | -2.523 | 3.495 | -1.699 |
| 4.284 | -2.398 | 3.495 | -1.721 |
| 4.237 | -2.301 | 3.481 | -1.770 |
| 4.222 | -2.398 | 3.456 | -1.824 |
| 4.301 | -2.523 | 3.398 | -1.745 |
| 4.347 | -2.523 | 3.398 | -1.678 |
| 4.377 | -2.523 | 3.398 | -1.602 |
| 3.886 | -2.523 | 3.398 | -1.569 |
| 4.824 | -2.523 | 3.398 | -1.602 |
| 4.456 | -2.523 | 3.398 | -1.699 |
| 4.387 | -2.523 | 3.398 | -1.745 |
| 4.377 | -2.523 | 3.398 | -1.745 |
| 4.301 | -2.523 | 3.398 | -1.745 |
| 4.155 | -2.523 | 3.398 | -1.745 |
| 4.260 | -2.420 | 3.398 | -1.824 |

Apdx3.3 The first set of data after the further normalization

| n2(mkw) | n2(lhc) | n2(vw) | n2(dyh) |
|---|---|---|---|
| 3.077 | -2.261 | 2.699 | -1.349 |
| 3.111 | -2.261 | 2.699 | -1.349 |
| 3.151 | -2.228 | 2.699 | -1.349 |
| 2.935 | -2.222 | 2.699 | -1.349 |
| 3.261 | -2.276 | 2.688 | -1.349 |
| 3.151 | -2.284 | 2.683 | -1.349 |
| 3.142 | -2.293 | 2.716 | -1.301 |
| 3.130 | -2.412 | 2.728 | -1.349 |
| 3.111 | -1.872 | 2.761 | -1.349 |
| 3.048 | -2.261 | 2.500 | -1.349 |
| 3.077 | -2.261 | 2.754 | -1.349 |
| 3.122 | -2.261 | 2.747 | -1.023 |
| 3.130 | -2.261 | 2.747 | -1.349 |
| 3.142 | -2.199 | 2.747 | -1.361 |
| 3.118 | -2.151 | 2.741 | -1.385 |
| 3.111 | -2.199 | 2.728 | -1.412 |
| 3.151 | -2.261 | 2.699 | -1.372 |
| 3.173 | -2.261 | 2.699 | -1.339 |
| 3.188 | -2.261 | 2.699 | -1.301 |
| 2.943 | -2.261 | 2.699 | -1.284 |
| 3.412 | -2.261 | 2.699 | -1.301 |
| 3.228 | -2.261 | 2.699 | -1.349 |
| 3.194 | -2.261 | 2.699 | -1.372 |
| 3.188 | -2.261 | 2.699 | -1.372 |
| 3.151 | -2.261 | 2.699 | -1.372 |
| 3.077 | -2.261 | 2.699 | -1.372 |
| 3.130 | -2.210 | 2.699 | -1.412 |

Apdx3.4 The second set of data after the further normalization

## REFERENCES


**Almeida**, L. B., 1987, "A Learning Rule for Asynchronous Perceptrons with Feedback in a Combinational Environment", In Proceedings, 1st International Conference on Neural Networks, Vol. 2 p609-618, San Diego, CA, June 1987. IEEE.

**Almeida**, L. B., 1989, "Backpropagation in Non-feedforward Networks", Neural Computing Architectures, Chapter.5 p75-90.

**Almeida**, L. B. and Neto, J. P., 1989, " Recurrent Backpropagation and Hopfield Networks", Neural Computing Algorithms, Architectures and Applications, Springer-Verlag, NATO ASI Series.

**Anderson**, J. A. and Rosenfeld, E., (eds) 1988, "Neurocomputing: Foundations of Research", MIT Press/Bradford Books, Cambridge MA.

**Anderson**, S., Merrill, J.W.L. and Port, R., 1989, "Dynamic Speech Categorization with Recurrent Networks", in Proc. of the 1988 Connectionist Models Summer School (Pittsburg, 1988), eds, Touretzky, D. et al, p398-406.

**Arbib**, M. A., 1987, "Brains, Machines, and Mathematics", Springer-Verlag Heidelberg.

**Ash**, T., 1989, "Dynamic Node Creation in Backpropagation Networks", Univ. of California, San Diego, ICS Report 8901.

**Baum**, E. B. and Haussler, D., 1989, "What Size Net Gives Valid Generalization?", Neural Computation 1, p151-160.

**Baum**, E. B., 1988, "On the Capabilities of Multilayer Perceptrons", Journal of Complexity, 4 p193-215.

**Brady**, M., 1988, "BP Fails to Separate to Separate Where Perceptrons Succeed", IEEE ICNN 1988 Vol.I p649-656.

**Brady**, M., Raghavan, R. and Slawny, J., 1988, "Gradient Descent Fails to Separate", In Proceedings of the IEEE International Conference on Neural Networks, 1 p649-656.

**Chauvin**, Y. , 1990, "Generalization Performance of Overtrained Back-Propagation Networks", Lecture Notes in Computer Science 412 (Springer-Verlag, 1990), p46-55.

**Cottrell**, G.W., Munro, P.W. and Zipser, D., 1987, "Image compression by back propagation: A Demonstration of Extensional Programming. In N.E. Sharkey (Ed.) Advances in Cognitive Science, Vol.2 Chichester, England: Ellis Horwood.

**Crick**, F., 1989, " The Recent Excitement about Neural Networks", Nature, Vol 337 p129-132.

**Cybenko**, G., 1989, "Approximation by Superpositions of a Single Function", Mathematics of Control, Signals and Systems, 2, p303-314.

**Cybenko**, G., 1990, "Complexity Theory of Neural Networks and Classification Problems", Lecture Notes in Computer Science 412 (Springer-Verlag, 1990), p26-43.

**Elman**, J. L., 1988, "Finding Structure in Time", Tech. Report CRL-8801, Center for Research in Language, Univ. of California, San Diego, April, 1988.

**Elman**, J. L., 1989, "Representation and Structure in Connectionist Models", CRL Tech Report 8903, Univ. of California, San Diego, August 1989.

**Fahlman**, S.E., 1988a, "Faster Learning Variations on Back-propagation Networks", Computer Speech & Language, 2 p205-218.

**Fahlman**, S.E., 1988b, "An Empirical Study of Learning Speed in Back-Propagation Networks", CMU-CS-88-162.

**Funahashi**, K., 1989, "On the Approximate Realisation of Continuous Mappings by Neural Networks", Neural Networks, 2 p183-192.

**Gherrity**, M., 1989, "A learning algorithm for Analog, Fully Recurrent Neural Networks", in Proc. of IEEE IJCNN 1989 Vol. 2 p643-644.

**Hebb**, D.O., 1949, " The Organisation of Behaviour", Wiley, New York.

**Hertz**, J. A., R. G. Palmer, and Krogh, A. S., 1991, "Introduction to the Theory of Neural Computation", Addison-Wesley, 1991.

**Hinton**, G. E., 1987, "Learning Translation Invariant Recognition in a Massively Parallel Networks", Lecture Notes in Computer Science, 258, (Springer-Verlag, 1987) p1-13.

**Hinton**, G. E., 1989, "Connectionist Learning Procedures", Artificial Intelligence 40 p185-234.

**Hinton**, G. E., 1991, "Lectures on Neural Networks", Easter Course, April 18-19 1991, St. Andrews University, UK.

**Hopfield**, J.J., 1982, " Neural networks and Physical Systems with Emergent Collective Computational Abilities", Proc.Natl.Acad,Sci.USA, Vol.79, p2554-2558.

**Hopfield**, J.J., 1984, "Neurons with Graded Response Have Collective Computational Properties Like Those of Two-state Neurons", Proc.Natl.Acad,Sci.USA, Vol.81 p3088-3092.

**Hornik**, K., Stinchcombe, M., and White, H., 1989, "Multilayer Networks are Universal Approximators", Neural Networks, 2, p359-366.

**Jones**, W. P. and Hoskins, J., 1987, "Back-propagation", BYTE, Oct, 1987 p155-162.

**Jordan**, M. I, 1986a, "Serial Order: A Parallel, Distributed Processing Approach", Technical Report 8604, University of California, San Diego, Institute for Cognitive Science.

**Jordan**, M. I, 1986b, "Attractor Dynamics and Parallelism in a Connectionist Sequential Machine", in Proceedings of the Eighth Annual Conference of the Cognitive Science Society p531-546.

**Khanna**, T., 1990, "Foundations of Neural Networks", Addison-Wesley Publishing Company, 1990.

**Kinoshita**, J. and Palevsky, N.G., 1987, "Computing with Neural Networks", High Technology May, 1987, p24-31.

**Lang**, K. J. and Hinton, G.E., 1988, "A Time-Delay Neural Network Architecture for Speech Recognition", Carnegie Mellon University, CMU-CS-88-152.

**Lang**, K. J. and Witbrock, M.J., 1988, "Learning to Tell Two Spirals Apart", Proceedings of the 1988 Connectionist Models Summer School, Morgan Kaufmann, p52-59.

**Le Cun**, Y., 1985, "A Learning Scheme for Asymmetric Threshold Networks", in Proc. Cognitive, 85 Paris, p599-604.

**Lippmann**, R.P., 1987, "An Introduction to Computing with Neural Nets", IEEE Acoustics, Speech, and Signal Processing Magazine, April 1987 p4-22.

**Lippmann**, R.P., 1989, "Review of Neural networks for Speech Recognition", Neural Computation, Vol. 1 p1-38.

**McClelland**, J.L. and Elman, J., 1986, "Interactive Processes in Speech Perception: The TRACE Model", in Parallel Distributed Processing, Vol.2 Chap.15.

**McClelland**, J. L. and Rumelhart, D.E., 1986, "A Distributed Model of Human Learning and Memory", in Parallel Distributed Processing, MIT Press, p170-215.

**McClelland**, J. L., Rumelhart, D.E., and the PDP Research Group, 1986, "Parallel Distributed Processing: Explorations in the Microstructure of Cognition", Vol. 2: Psychological and Biological Models. Cambridge: MIT Press.

**McClelland**, J. L. and Rumelhart, D.E., 1988, "Explorations in Parallel Distributed Processing, Cambridge: MIT Press.

**McCulloch**, W.S. and Pitts, W. H., 1943, " A Logical Calculus of the Ideas Immanent in Nervous Activity", Bulletin of Mathematical Biophysics, Chicago: University of Chicago Press, Vol. 5 p115-133.

**Minsky**, M.L. and Papert, S.,1969, "Perceptrons: An Introduction to Computational Geometry", Cambridge, Mass MIT Press.

**Mozer**, M.C., 1989, " A Focused Back-Propagation Algorithm for Temporal Pattern Recognition", Complex Systems Vol.3 p349-381.

**Nowlan**, S. J., 1988, "Gain Variation in Recurrent Error Propagation Networks", Complex Systems, Vol. 2 No.3 p305-320.

**Parker**, D.B., 1985, "Learning-logic", Technical Report TR-47, Sloan School of Management, MIT Cambridge MA, 1985.

**Pearlmutter**, B. A., 1988, "Learning State Space Trajectories in Recurrent Neural Networks", Carnegie Mellon University, CMU_CS_88_191.

**Pearlmutter**, B. A., 1989, "Learning State Space Trajectories in Recurrent Neural Networks", Neural Computation, 1(2) p263-269, 1989.

**Pearlmutter**, B. A., 1990, "Dynamic Recurrent Neural Networks", Technical Report CMU-CS-90-196, Carnegie Mellon University, Pittsburgh, 1990.

**Pineda**, F.G., 1987, "Generalization of Back-Propagation Recurrent Networks", Physical Review Letters, Vol. 59, No. 19. p2229-2232, 1987.

**Plaut**, D. C., Nowlan, S.J. and Hinton, G.E., 1986, "Experiments on Learning by Back-propagation", Technical Report CMU-CS-86-126, Carnegie Mellon University, Pittsburgh PA 15213.

**Robinson,** A.J. and Fallside, F., 1987, "Static and Dynamic Error Propagation Networks with Application to Speech Coding". In Anderson, D.Z. editor, Neural Information Processing Systems, p632-641, New York 1987, American Institute of Physics.

**Rohwer**, R. and Forrest, B., 1987, "Training Time-Dependence in Neural Networks", In Proceedings IEEE ICNN San Diego, 1987 p701-708.

**Rohwer**, R. and Renals, S., 1988, "Training Recurrent Networks", Proc nEURO 88

**Rohwer**, R., 1990, "The Moving Targets Training Algorithm", In Touretzky, D. S. editor, Advances in Neural Information Processing Systems 2, p558-565, San Mateo, CA, 1990. Morgan Kaufmann.

**Rosenblatt**, F., 1962, "Principles of Neurodynamics", Spartan Books, New York, 1962.

**Rumelhart**, D. E., Hinton, G. E., Williams, R.J., 1986a, "Learning Representations by Back-Propagating Errors", Nature 323 p533-536.

**Rumelhart**, D. E., Hinton, G. E. and Williams, R. J., 1986b, "Learning internal representations by error propagation", In Parallel distributed processing: Explorations in the Microstructure of Cognition, Vol 1. Chapter 8, Cambridge MIT, 1986.

**Rumelhart**, D.E., McClelland, J. L., and the PDP Research Group, 1986, "Parallel Distributed Processing: Explorations in the Microstructure of Cognition", Vol. 1: Foundations. Cambridge: MIT Press.

**Samad**, T., 1988, "Back-Propagation Is Significantly Faster if the Expected Value of the Source Unit is Used for Update", INNS, Boston.

**Servan-Schreiber**, D., Cleeremans, A. and McClelland, J. L., 1988, "Encoding Sequential Structure in Simple Recurrent Networks", Carnegie Mellon University, CMU_CS_88_183.

**Shimohara**, K., Uchiyama, T. and Tokunuga, Y., 1988, "Backpropagation Networks for Event-driven Temporal Sequence Processing", Proc. of IEEE International Conference on Neural Networks, Vol.1 p665-672.

**Silva**, F.M. and Almeida, L.B., 1990, " Acceleration Techniques for the Backpropagation Algorithm", Lecture Notes in Computer Science, No.412 (Springer-Verlag, 1990) p110-119.

**Simard**, P.Y., Ottaway, M.B. and Ballard, D.H., 1989, "Analysis of recurrent backpropagation", The University of Rochester, Computer Science Dept., Rochester, NY 14627, Technical Report 253.

**Simpson**, P. K., 1989, "Artificial Neural Systems", Pergamon Press, 1989.

**Sontag**, E.D. et al , 1988, "BP Separates when Perceptrons Do", Report Rutgers Center for Systems and Control RP SYCON-88-12.

**Stornetta**, W., Hogg, T. and Huberman, B., 1987, "A Dynamic Approach to Temporal Pattern Recognition", in Proc.IEEE Conf. Neural Systems, Denver, 1987.

**Sutton**, R. S., 1986, "Two problems with backpropagation and other steepest-descent learning procedures for networks", Proc. 8th annual conference of Cognitive Science Society, p823-831, Hillsdale, NJ: Lawrence Erlbaum Associates.

**Tank**, D.W. and Hopfield, J. J, 1987, "Neural Computation by Concentrating Information in Time", Proc.Natl. Acad. Sci. USA Vol.84 p1896-1900.

**Tank**, D.W. and Hopfield, J. J, 1988, "Collective Computation in Neuronlike Circuits", Sci.American Vol. 257 No.6 p62-70.

**Tazelaar**, J. M., 1989, "Neural Networks", BYTE Aug. 1989.

**Touretzky**, D. S. and Pomerleau, D.A., 1989, "What's Hidden in the Hidden Layers", BYTE Aug.1989.

**Valiant**, L.G., 1984, " A Theory of the Learnable" , Communication of ACM, Vol. 27 No.11 p1134-1142.

**Waibel**, A., Hanazawa, T., Hinton, G., Shikano, K. and Lang, K., 1987, "Phoneme Recognition Using Time-Delay Neural Networks", Technical Report TR-I-0006, Advanced Telecommunications Research Institute, Japan.

**Werbos**, P.J., 1974, "Beyond regression: New Tools for Prediction and Analysis in the Behavioural Sciences", PhD thesis, Harvard University, Cambridge, MA.

**Weir**, M. K., 1990, "A Method for Self-determination of Adaptive Learning Rates in Back-propagation", Neural Networks, Vol.4 No. 3, 1991.

**Weir**, M. K., and Chen, L.H., 1991, " Neural Training and Generalization of Sequences using Continuous Temporal Structure", IJCNN, 1991 Singapore Vol.3 p2027-2031.

**Weir**, M. K., and Chen, L.H., 1990, " Training and Generalization using Continuous Backpropagation", Technical Report CS90/90/12, St. Andrews University, Computational Science Division, Scotland.

**Widrow**, B. and Hoff, M., 1960, " Adaptive Switching Circuits", In Western Electronic Show and Convention, Convention Record, Vol.4 p96-104. Institute of Radio Engineers.

**Williams**, R. J. and Zipser, D., 1988, "A Learning Algorithm for Continually Running Fully Recurrent Neural Networks", Tech. Report ICS Report 8805 Univ. of Califormia, San Diego, La Jolla, CA 92093, Nov., 1988.

**Yu**, Y. H. and Simmons, R. F., 1990, " Descending Epsilon in Backpropagation : Better Generalisation" , Technical Report AI90-130. AI Laboratory, Univ. of Texas at Austin, USA.

**Zeidenberg**, M., 1990, "Neural Networks in Artificial Intelligence", Ellis Horwood Series in AI, 1990.