Neural Networks for Perceptual Grouping

Sarbjit Singh Sarkaria

Doctor of Philosophy

The University of Aston in Birmingham

September 1990

The University of Aston in Birmingham

Neural Networks for Perceptual Grouping

Sarbjit Singh Sarkaria

Doctor of Philosophy

September 1990

Summary

A number of researchers have investigated the application of neural networks to visual recognition, with much of the emphasis placed on exploiting the network's ability to generalise. However, despite the benefits of such an approach it is not at all obvious how networks can be developed which are capable of recognising objects subject to changes in rotation, translation and viewpoint. In this study, we suggest that a possible solution to this problem can be found by studying aspects of visual psychology and in particular, perceptual organisation. For example, it appears that grouping together lines based upon perceptually significant features can facilitate viewpoint independent recognition.

The work presented here identifies simple grouping measures based on parallelism and connectivity and shows how it is possible to train multi-layer perceptrons (MLPs) to detect and determine the *perceptual significance* of any group presented. In this way, it is shown how MLPs which are trained via backpropagation to perform individual grouping tasks, can be brought together into a novel, large scale network capable of determining the perceptual significance of the whole input pattern. Finally the applicability of such significance values for recognition is investigated and results indicate that both the MLP and the Kohonen Feature Map can be trained to recognise simple shapes described in terms of perceptual significances.

This study has also provided an opportunity to investigate aspects of the backpropagation algorithm, particularly the ability to generalise. In this study we report the results of various generalisation tests. In applying the backpropagation algorithm to certain problems, we found that there was a deficiency in performance with the standard learning algorithm. An improvement in performance could however, be obtained when suitable modifications were made to the algorithm. The modifications and consequent results are reported here.

Keywords : Neural Networks, Perceptual Organisation, Backpropagation, Multi-layer Perceptron, Recognition.

To
my parents.

Acknowledgements

I wish to express my thanks to the following people :

Dr. Alan Harget for his support and guidance.

Dr. Ela Claridge for seeing me through the early stages of my research and for much needed encouragement.

To my friends and colleagues in the department.

And finally to SERC for providing the necessary funding.

List of Contents

Title page 1
Summary 2
Dedication 3
Acknowledgements 4
List of Contents 5
List of Figures 10
List of Tables 12

1. Introduction 13

2. Neural Networks 16
2.1 Introduction and Brief History 16
2.2 The Multi-Layer Perceptron 19
2.2.1 The Error Backpropagation Algorithm 21
2.2.1.1 Formal Analysis 21
2.2.1.2 Applying the Backpropagation Rule 26
2.2.1.3 The Activation Function 27
2.3 The Hopfield Net 28
2.3.1 Content Addressable Memory 28
2.3.2 Solving Optimisation Problems 29
2.4 Competitive Learning 31
2.5 The Carpenter and Grossberg Net 32
2.6 The Boltzmann Machine 33
2.7 The Kohonen Self-Organising Feature Map 35
2.7.1 Feature Maps in the Brain 35
2.7.1.1 Evidence for Feature Maps 36
2.7.1.2 How are Feature Maps Formed? 36
2.7.2 Lateral Connectivity 37
2.7.2.1 Lateral Interactions and the "Mexican-Hat" Function 37
2.7.3 Practical Implementation of a Kohonen Feature Map 38
2.7.3.1 The Simplified Self-Organising Algorithm 39
2.7.4 Examples of Topology Preserving Maps 40
2.8 Two Examples of Successful Neural Net Applications. 41

2.8.1 NETtalk 41
2.8.2 The Neocognitron 43
2.9 Summary 44

3. Computational and Perceptual Approaches to Computer Vision 46
3.1 Introduction to Computer Vision 46
3.1.1 Levels of Processing 46
3.2 Some 3D Vision Systems 48
3.2.1 ACRONYM 48
3.2.2 A Three Dimensional Part Orientation System (3DPO) 49
3.2.3 SCERPO 50
3.2.4 Use of Perception in Vision Systems 51
3.3 Biologically and Perceptually Motivated Approaches. 52
3.3.1 Introduction 52
3.3.2 The Influence of Biological Findings on Computer Vision 53
3.3.3 Linsker's Perceptual Network 54
3.3.4 Endstopped Neurons and Curvature 57
3.3.5 The Hierarchical Structure Code 57
3.3.6 Perceptual Grouping 59
3.3.6.1 What are Perceptual Groups? 59
3.3.6.2 Perceptual Groups and SCERPO 61
3.3.6.2.1 Grouping According to Proximity 63
3.3.6.2.2 Grouping According to Parallelism 64
3.3.6.2.3 Grouping According to Collinearity 64
3.3.7 Walter's Computer Vision Model and Perceptual Organisation 65
3.3.7.1 Perceived Brightness 66
3.3.7.2 A Computer Model Based on Psychophysics 67
3.3.7.3 Relation to Perceptual Organisation 68
3.4 Neural Networks for Vision 69
3.4.1 Introduction 69
3.4.2 Face Recognition 70
3.4.3 Methods Using Backpropagation 71
3.4.3.1 Character Recognition 71
3.4.3.2 An Autonomous Guided Vehicle. 72
3.4.3.3 Learning to Perceive Left and Right 73
3.4.3.4 The 'T-C' Problem 75
3.4.4 Structured Neural Networks 76
3.5 Summary 78

4. Connectionist Approach to Processing Perceptual Groups 79
4.1 Introduction 79
4.2 Outline of Proposed Project 80
4.3 Coding the Input Pattern 82
4.3.1 Description of Coding Scheme 83
4.3.2 Feature Detection 85
4.3.2.1 The Detection of Parallel Lines 85
4.3.2.2 The Detection of Collinear Lines 86
4.3.2.3 The Detection of Connectivity 86
4.3.3 Some Examples of Patterns Represented Using this Scheme. 89
4.4 What Does the Network Need to Output and How? 90
4.4.1 Perceptual Significance of Parallel Lines. 91
4.4.2 Perceptual Significance of End-end Connectivity 93
4.4.3 Sequential Processing and SCERPO 93
4.4.4 Example of How a Network can Simulate a Real Function 94
4.4.4.1 The Training Set 95
4.4.4.2 The Network 96
4.4.4.3 The Training Procedure 97
4.5 Joining Subnets to Perform Complex Tasks 98

5. Development and Training Results of the Perceptual Network 100
5.1 Detailed Plan of the Perceptual Network 100
5.1.1 The Line-processing Structure 102
5.1.1.1 The Line-pair Structure 104
5.1.1.2 The 'Filter' Structures 105
5.1.2 The Corner Processing Structure. 106
5.1.2.1 Local Corner Detectors 106
5.1.2.2 Receptive Fields 108
5.1.3 Practical Implementation of the Perceptual Net. 110
5.1.4 Validation of the Backpropagation Simulator 111
5.1.4.1 The XOR Problem 111
5.1.4.2 The Parity Problem 112
5.2 Details of the Main Perceptual Net Components 113
5.2.1 Estimating the Accuracy of a Trained Network 113
5.2.2 The "Distance-Discriminator" 114
5.2.2.1 The Training Set 115
5.2.2.2 The Network 116
5.2.2.3 Results of Training 116

5.2.3 The "Line-Pair Evaluator" 117
5.2.3.1 Simulating a Continuous Function 117
5.2.3.2 Using Inhibition to Handle Special Cases 118
5.2.3.3 Results of Training 119
5.2.4 The "Min-Filter" 122
5.2.4.1 Using a Modified Backpropagation Algorithm 123
5.2.4.2 Alternative Ways of Learning the Same Task 124
5.2.5 The "Max-Filter" 126
5.2.6 The "Corner-Detector" 126
5.3 Analysis of Some Training Results 129
5.3.1 Generalisation Problems of the Distance-Discriminator 129
5.3.1.1 Limited Connectivity 130
5.3.1.2 A Possible Explanation for Generalisation Problems 131
5.3.2 Generalisation Problems of the Corner Detector 132
5.3.3 Generalisation of Min / Max Filters 134
5.3.3.1 Generalisation and the Weights of a Trained Network 135
5.4.4 Accelerated Learning with the Modified Backpropagation 137

6. Testing and Evaluation 139
6.1 Software Evaluation 139
6.1.1 Testing Response to All Primitives 140
6.1.2 Ability to Handle Translation Invariance 140
6.1.3 Generalisation of Line Lengths 141
6.1.4 Testing with Simple Shapes 142
6.2 System Evaluation 143
6.2.1 Validity of Significance Values. 143
6.2.1.1 The Distance-Discriminator 143
6.2.1.2 The Corner-Detector 144
6.2.2 Limitations 144
6.2.2.1 Orientation Planes 144
6.2.2.2 Receptive Fields 145
6.2.2.3 Collinear Lines 146
6.2.3 Perceptual Net Applications. 146

7. Shape Recognition 150
7.1 Using Backpropagation for Recognition 150
7.1.1 Format of Desired Outputs 151
7.1.2 Performance of a Trained Net 152

7.2 Using the Kohonen Feature Map for Recognition 158
7.2.1 Validation of the Kohonen Simulator 159
7.2.2 Properties of a Trained Feature Map 162
7.2.3 Performance of a Trained Feature Map 165
7.3 Comparison of a Kohonen Map with a Backpropagation Net 167
7.3.1 Training Advantages over Backpropagation 172

8. Conclusions 174
8.1 The Perceptual Net and Backpropagation 174
8.2 The Perceptual Net and the Kohonen Feature Map 177
8.3 Comparison of the Backpropagation and Kohonen Algorithms 178
8.4 Future Work 179

References 182

Appendix A 189
Appendix B 192
Appendix C 194
Appendix D 196
Appendix E 198
Appendix F 199
Appendix G 207
Appendix H 213
Appendix I 215
Appendix J 216
Appendix K 217
Appendix L 218
Appendix M 222

List of Figures

Figure 2.1 A three layer perceptron....26
Figure 2.2 The sigmoid activation function....27
Figure 2.3 The travelling salesman problem....30
Figure 2.4 A self-organising feature map....35
Figure 2.5 An 'Activity Bubble'....37
Figure 2.6 The "Mexican-Hat" function....38
Figure 2.7 Schematic diagram of NETtalk....42
Figure 3.1 Response of a vertically trained neuron to bars of varying angle....54
Figure 3.2 Feed forward architecture of Linsker's perceptual network....55
Figure 3.3 An illustration of perceptual grouping....60
Figure 3.4 Spontaneous grouping....62
Figure 3.5 Enhancement matrix....67
Figure 3.6 T and C shapes at all possible orientations....75
Figure 4.1 A simple line coding scheme....83
Figure 4.2 Problems with low resolution coding....85
Figure 4.3 How corners can be detected....87
Figure 4.4 Representation of corners....88
Figure 4.5 Some examples of coded patterns....89
Figure 4.6 Problems with encoding distance....90
Figure 4.7 A multi-layer perceptron for determining significances....96
Figure 4.8 Variation of epochs against hidden units....97
Figure 4.9 Splitting up a task....99
Figure 5.1 Perceptual Network - Outer structure details....101
Figure 5.2 Inside the parallel line processing structure....102
Figure 5.3 Inside the line-pair structure....104
Figure 5.4 Composition of the 'filter' structures....105
Figure 5.5 Inside a local corner detector....107
Figure 5.6 Receptive fields....108
Figure 5.7 An exploded view of the corner processing structure....109
Figure 5.8 Examples of how 1/s values are calculated....115
Figure 5.9 Linear ordering of input planes....116
Figure 5.10 Inhibition of the line-pair structure....118
Figure 5.11 The count detector....119
Figure 5.12 The hard-limiting activation function....123
Figure 5.13 Examples of some corners....127

Figure 5.14 Receptive fields for a diamond 128
Figure 5.15 Similarity of input vectors 131
Figure 6.1 Example of spurious outputs 140
Figure 6.2 Performance with actual shapes 142
Figure 6.3 Invalid pattern - three lines of the same orientation 145
Figure 6.4 Invalid pattern - a triangle with three lines in a single receptive field 146
Figure 6.5 Connections for a vertically collinear line detector 146
Figure 6.6 Total significances calculated by the Perceptual Net 147
Figure 6.7 Ordering of perceptual groups according to total significance 148
Figure 6.8 Illustration of cause of lack of rotation invariance 149
Figure 7.1 Examples of output formats 152
Figure 7.2 Outputs from a Kohonen net 162
Figure 7.3 Irregular shaped activity bubbles 163
Figure 7.4 A feature map of different shapes 164
Figure 7.5 Example of generalisation ability of the shapes feature map 166
Figure 7.6 Feature map generated for training set used previously for back-propagation 168
Figure 7.7 Generalisation with test patterns from Tables 7.1 and 7.2 170
Figure 7.8 Significance values for a square and a zigzag 172

List of Tables

Table 5.1a Performance of a 6 hidden unit line-pair evaluator trained to 0.000001 120
Table 5.1b Performance of a 4 hidden unit line-pair evaluator trained to 0.000002 121
Table 5.1c Performance when trained on a larger set 122
Table 5.2 Performance of the min-filter with untrained exemplars 126
Table 5.3 Generalisation of the corner-detector 132
Table 5.4a Performance of identically trained nets on members of the training set 133
Table 5.4b Performance of identically trained nets on patterns outside the training set 133
Table 5.5 Generalisation of max-filter 134
Table 5.6 Generalisation of identically trained max-filters 135
Table 5.7 Weights reached after training, for five repeated training runs 136
Table 5.8 Evidence for faster learning 138
Table 7.1 Generalisation results when only line lengths are modified 155
Table 7.2 Generalisation results when shapes are modified 157

Chapter 1

Introduction

The idea that one day, machines will be developed that possess intelligence comparable to humans has always been a luring prospect for computer scientists. Not surprisingly, a lot of effort has been directed towards the study of artificial intelligence. Most frequently this has been realised through the use of rule based or expert systems, which apparently have been quite successful in emulating human reasoning. However, when it comes to tackling seemingly simple sensory tasks such as object recognition, relatively little or no success has been achieved. Yet this is something that humans are incredibly adept at and personal experience shows that visual or speech recognition is performed almost instantaneously and with little or no conscious effort.

With traditional AI solutions reaching saturation, attention is now focussing towards artificial neural networks in the hope that by capturing some of the basic properties of biological systems such as the brain, answers can be found to real recognition tasks. The development of learning algorithms capable of training multiple layers of artificial neurons, has encouraged this trend, as well as allaying past criticisms that had stifled early research in the area.

The attraction of using neural networks for solving visual tasks is clear. Fast recognition, tolerance to noise, the ability to learn and generalise are all features that a network could possess. Work in the area has been quite fervent with neural networks used in applications as wide ranging as character recognition to autonomous vehicle guidance. The success of such applications is encouraging; however few examples exist where use has been made of ideas from visual psychology. The most popular of these ideas are the grouping phenomena studied by early Gestalt psychologists which suggest that important image components are integrated together and perceived as a single coherent feature. Often referred to as *perceptual organisation*, the processes responsible for these

phenomena are thought to provide an important intermediate stage which facilitates high level recognition. Thus the implementation of such processes could prove very useful for higher level recognition.

As yet the level of activity in this area has been relatively low, with only a few notable examples published. Instead, most neural network approaches involved with vision have opted to solve the problem directly, paying little attention to the benefits that could be gained by exploiting perception. By incorporating ideas relating to perceptual organisation, neural networks could be developed which possess some of the qualities of human vision systems. Furthermore, the low level nature of the processes involved with perceptual organisation would make them highly suitable for implementation in terms of neural networks. Thus the use of neural networks to solve perceptual problems seems a very promising prospect and provides a interesting and worthwhile area for research.

The three main aims of the work presented in this thesis are as follows :

(1) To identify suitable visual tasks based on perceptual organisation.

(2) To carry out an implementation of a neural network, or networks, to perform these tasks.

(3) To study the problems and practicalities encountered when devising and training neural networks.

Note however that this thesis does not attempt to make any claim regarding the biological accuracy or plausibility of the neural network mentioned in point (2) above.

The thesis is organised as follows :

Chapter 2 presents a general review of the current literature found in the area of neural networks, with chapter 3 following up with an account of work in the field of traditional computer vision and more modern, perceptually based approaches to vision. To complete the review, chapter 3 ends with a description of several examples of neural networks as applied to vision problems.

Chapter 4 puts forward a research proposal based on the task of grouping together perceptually significant lines. The task in itself is simple and involves quantitatively evaluating the prominence or *significance* of various features found in groups of straight lines, such as parallelism and connectivity. The network to achieve this, called the Perceptual Network, consists of many networks, each individually trained using backpropagation to perform a specific subtask.

Implementation details of the Perceptual Network are given in chapter 5, which as well as describing how the individual networks of the Perceptual Network were trained, highlights some of the problems and more important aspects associated with training using backpropagation.

Chapter 6 is concerned with aspects relating to testing and evaluation of the Perceptual Network and describes the steps that were taken to test the completed network.

Chapter 7 illustrates how the outputs from the Perceptual Network could be used for recognition of simple shapes and compares the Kohonen algorithm with backpropagation for achieving this.

A summary of the work completed and a discussion of the major conclusions is given in chapter 8.

Chapter 2

Neural Networks

2.1 Introduction and Brief History

The rapid increase in the level of activity in neural network research suggests that the idea is new and is something that has only recently been discovered. However, artificial neural networks have been in existence since the early 1940s. McCulloch and Pitts (1943) first presented a computational model of a real neuron in which it was suggested that a neuron simply computes a weighted sum of its inputs (dendrites) and fires only if the sum is greater than a preset threshold value. It was thought that learning took place in the brain, via changes in the actual physical connections between neurons. Hebb (1949) however, suggested that learning might take place in biological neural nets not by physical changes, but by chemical changes in the synaptic gap between one neuron and another. Research in the field was stimulated when Rosenblatt (1962) introduced his idea of the *perceptron*. Rosenblatt was a psychologist who was interested in finding out how a biological system senses, stores and uses information from the real world. It was these incentives that lead him to the development of the now famous perceptron.

The perceptron consisted of an array of photoreceptor cells providing a 'retinal' input to a network of *association units*. The connection between photoreceptors and these 'A-units' was randomly assigned, such that each unit received input from a limited number of cells selected randomly from the input array. The output of the A-units was then connected to several output units, each designated the task of responding to a particular input pattern. The machine was not solely electronic, since the output units actually consisted of potentiometers and motors, all linked together so that the output of one could affect all of the others.

Rosenblatt demonstrated that the perceptron was capable of learning, since it could be trained to recognise a small set of different patterns and could also perform limited

recognition of unknown patterns. The perceptron embodied all the features of neural networks seen today; weighted inputs, thresholding, and the ability to learn by modification of weights.

At the time, Rosenblatt's work caused a great deal of excitement and some of the claims made in favour of perceptrons were quite impressive. Perceptrons, did however suffer from severe limitations; for example, even then it had been realised that scaling up tasks in size and complexity from simple toy domains was extremely difficult. Secondly, although it was thought that multiple layers of perceptrons could in principle solve anything, methods for training such complex systems were not known. A thorough analysis of the abilities and limitations of perceptrons was undertaken by Minsky and Papert (1969) who in their famous book proved that perceptrons suffered from some fundamental limitations. In particular, they showed that the perceptron could not determine the parity of an input pattern and that to achieve it would require a massive increase in the size of the network. Such restrictions were indeed severe, especially as the problem was so simple and could instead be very easily solved using conventional computation. The results of Minsky and Paperts' work had an almost devastating effect on perceptron research at the time. Particularly as the now traditional approaches to AI, were just emerging at the time and were obviously seen as the preferred target for further funding.

Although small groups of workers have continued to study neural nets since these early attempts, it is only recently that research has been taken up in earnest. This has been due to the development of new training algorithms and network topologies as well as the general belief that neural net models are somehow better suited to cope with visual and speech recognition tasks than the knowledge based techniques used in current AI approaches. This is the view expressed by Fahlman and Hinton (1987) who believe that massively parallel networks of simple computing elements, may provide a way of capturing some of the essential properties of intelligence that have not been yet been implemented using existing AI technology. Fahlman and Hinton (1987) put forward a

convincing argument in favour of connectionist approaches, particularly as a way of tackling, what for humans, are seemingly simple recognition tasks. A persuasive example is that for humans, recognition of an elephant is apparently effortless, yet to describe this process in words is far from easy. Indeed, recognition abilities in many domains, such as speech, vision as well as for example taste and smell, are executed subconsciously and with great ease. This would suggest that humans do not use symbolic descriptions but instead rely upon some other form of internal representation that cannot consciously be accessed.

Connectionism is still very controversial in the AI community, but much of the current work is motivated by the parallels that can be drawn between connectionist models and biological neural networks. For instance, it is known that neural nets can be made to develop their own internal distributed representations. As a consequence of this *fine grained* coding, the network once trained, will possesses a degree of tolerance to degradation, since no single unit is responsible for encoding any one entity. Thus the elimination of any one unit will not result in a great loss in the overall performance. This is a very important property of neural networks, since in the average adult, many thousands of neurons die every day and yet no noticeable degradation in recall or memory is observed.

One of the other attractive properties of neural networks is their ability to generalise. This was first seen in the perceptron (Rosenblatt, 1962) which had a limited ability to recognise patterns which were similar, but not identical to those presented during the learning phase. Fahlman and Hinton (1987) describe a five layer net which was taught various relationships within family trees. After training, the network had generalised well enough to correctly identify relationships between family members, not presented during the training phase.

Sejnowski and Rosenberg (1986) developed a system called NETtalk which could be taught how to pronounce English text. The training set comprised a series of words

cycling through an input vector and the output of the network was in terms of the phonemes necessary to generate the corresponding pronunciation. After training, NETtalk could adequately handle words that were not part of the training set. Obviously, the net had learned the rules implicit in this data set, enabling it to generalise to other unknown words.

These few examples give a flavour of the properties and the potential that neural networks may hold. Many recent introductory articles have appeared in the literature. Lippmann (1987) presents a taxonomy of neural net topologies and learning algorithms, giving good introductory descriptions of the major neural net techniques that have been tried. Notably, Lippmann distinguishes between nets that can be trained with or without supervision, and between nets which use binary values or continuous values.

The following sections give a fuller description of the major neural net paradigms now in existence starting with one of the most powerful and commonly used, the multi-layer perceptron.

2.2 The Multi-Layer Perceptron

The perceptron due to Rosenblatt was a simple single layer network of units and because of its limited complexity, was incapable of solving what were regarded as computationally easy problems, such as parity. It was also known at the time that, by increasing the complexity of the perceptron, particularly by using multiple layers, it would be possible to handle more difficult tasks. However the lack of algorithms for training multiple layers of perceptrons prevented these ideas from being tested and inevitably lead to their decline.

Only recently have suitable training algorithms become available. Notably Rumelhart et al. (1986) present a definitive description of the *error backpropagation algorithm* which is

capable of adapting not just weights belonging to a single layer of units, but any number of layers.

The error backpropagation algorithm is a supervised learning algorithm which learns to associate given input patterns with their corresponding output patterns. It is easily capable of learning the parity problem and as Rumelhart et al. (1986) show, it can be also used to solve many other problems, such as encoding, addition and negation. They also show how multiple layers of neurons could be arranged to handle more complex tasks and discuss how to train a net to recognise a 'T' or a 'C' independently of position or orientation.

The properties of the multi-layer perceptron are quite intriguing. For example, once trained on a set of patterns, the representation that the net will develop is in a distributed fashion, such that any single unit will not be responsible for encoding any single feature or entity that may exist in the input. As a consequence of this, the trained net is tolerant to degradation. Thus if a single neuron fails, then the effect will not result in a significant decrease in performance. During training, the net is attempting to detect and group together, features that are common in the input patterns. This ability to cluster together common features, results in a net that can generalise, that is, the net can generate the correct output, even if an input pattern is presented that was not a member of the training set. Both these properties are attractive features of neural networks, and are also observed in biological neural networks.

The main drawback with the use of the backpropagation algorithm is that because it is an iterative gradient descent method, it can become cumbersomely slow. Even on relatively simple problems, backpropagation can require the presentation of whole training sets many hundreds or thousands of times. This imposes a restriction on the size of networks that can be investigated. Although networks of a few thousand trainable weights might be capable of handling simple tasks, the kinds of networks necessary for real world recognition tasks, will be much too large to practically train using current computing

techniques. Fahlman (1988) confronts this issue and suggests that the only real solution is to combine advances in hardware technology, that is, use more powerful computers, with faster, more advanced learning algorithms.

Fahlman (1988) presents the results of an empirical study on finding ways of accelerating the speed of the standard backpropagation algorithm and devises an improved version which he calls 'Quickprop'. Essentially, Fahlman recommends making several modifications to the standard algorithm, which he says when tested on standard problems such as the parity and encoder provide a significant increase in performance as well as offering the possibility of scaling up the problem size.

The theory of the backpropagation algorithm will now be given since it is central to this project.

2.2.1 The Error Backpropagation Algorithm

The error backpropagation algorithm is an iterative gradient descent procedure, which attempts to minimise the mean square error between the actual output of a multi-layer feed forward net, and the desired output.

The algorithm is capable of finding an optimised set of weights to accomplish an arbitrary mapping between a set of input patterns and their associated output patterns. This optimisation process can be likened to finding a minimum energy well in an undulating energy landscape.

2.2.1.1 Formal Analysis

Given a desired output pattern t_{pj}, which appears across j output nodes, and the actual output pattern o_{pj} that the network is generating, then the error associated with that pattern is given by ,

$$E_p = \frac{1}{2}\sum_j (t_{pj} - o_{pj})^2 \qquad (1)$$

The global error for all patterns in the training set is thus $E = \sum_p E_p$

Now for a multi-layer feed forward network, the output of any unit j, for any single pattern is given by

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

where f_j is an activation function which maps the weighted sum net_{pj}, of a given neuron onto a range of output values, usually between 0 and 1 and where net_{pj} is calculated from the sum of the products of the inputs and the weights to unit j.

$$net_{pj} = \sum_i w_{ji}\, o_{pi} \qquad (3)$$

It can be shown that the standard delta-rule implements gradient descent[1] (Rumelhart et al. 1986), which expressed mathematically leads to an equality of the form

$$\Delta_p w_{ji} \propto -\frac{\partial E_p}{\partial w_{ji}}$$

where $\Delta_p w_{ji}$ is the change to be made to the weights from node i to node j, after the presentation of a pattern p. This relationship simply sets the required condition for gradient descent. Now the right hand side may be re-expressed as follows

$$\frac{\partial E_p}{\partial w_{ji}} = \frac{\partial E_p}{\partial net_{pj}} \frac{\partial net_{pj}}{\partial w_{ji}} \qquad (4)$$

---

[1] Backpropagation is actually an approximation to gradient descent, since this rule requires that weights be updated after the presentation of a whole training set, not after each exemplar, as in backpropagation.

where $\frac{\partial E_p}{\partial net_{pj}}$ represents the change in the error as a function of the change in the weighted sum of the inputs to the unit, and $\frac{\partial net_{pj}}{\partial w_{ji}}$ represents the effect of changing a weight on the inputs to the unit. From equation (3) we get

$$\frac{\partial net_{pj}}{\partial w_{ji}} = \frac{\partial}{\partial w_{ji}} \sum_{\kappa} w_{jk} o_{pk} = o_{pi}$$

and if we define

$$\delta_{pj} = -\frac{\partial E_p}{\partial net_{pj}}$$

(δ_{pj} is effectively the error signal that should be used to update the weights for unit j) then equation (4) becomes

$$-\frac{\partial E_p}{\partial w_{ji}} = \delta_{pj} o_{pi}$$

and so $\Delta_p w_{ij}$ can be expressed in terms of o_{pi} , the output of a unit which is input i to unit j, δ_{pj} the error for that unit, and a learning factor η, and thus we have derived a way of implementing gradient descent.

$$\Delta_p w_{ji} = \eta \, \delta_{pj} o_{pi}$$

However, we are still left with finding δ_{pj}. This may also be re-expressed in terms of the rate of change in the error with respect to the output of node j, and the rate of change in the output of node j, with respect to the change in the inputs to node j.

$$\delta_{pj} = -\frac{\partial E_p}{\partial net_{pj}} = -\frac{\partial E_p}{\partial o_{pj}} \frac{\partial o_{pj}}{\partial net_{pj}} \qquad (5)$$

o_{pj} is already known from equation (2), thus

$$\frac{\partial o_{pj}}{\partial net_{pj}} = f_j'(net_{pj})$$

where $f_j'(net_{pj})$ is the derivative of the activation function evaluated at the output of the jth node having a weighted sum net_{pj}. From (1),

$$\frac{\partial E_p}{\partial o_{pj}} = -(t_{pj} - o_{pj})$$

Substituting back into equation (5),

$$\delta_{pj} = (t_{pj} - o_{pj})\, f_j'(net_{pj})$$

However, this is true only for an output unit, not a hidden layer unit, since t_{pj} is a known target value. The target value for hidden layer units is not explicitly known. Thus re-expressing $\frac{\partial E_p}{\partial o_{pj}}$ in terms of hidden layer units, net_{pk}

$$\sum_k \frac{\partial E_p}{\partial net_{pk}} \frac{\partial net_{pk}}{\partial o_{pj}} \qquad (6)$$

and since

$$net_{pk} = \sum_i w_{ki}\, o_{pi}$$

substituting into equation (6),

$$\sum_k \frac{\partial E_p}{\partial net_{pk}} \frac{\partial}{\partial o_{pj}} \sum_i w_{ki}\, o_{pi} = \sum_k \frac{\partial E_p}{\partial net_{pk}} w_{kj} = -\sum_k \delta_{pk}\, w_{kj}$$

where net_{pk} represents the output of the k*th* unit in the hidden layer when presented with the pattern p. w_{kj} represents the weights from the kth hidden unit to the *jth* output unit and w_{ki} represents the weights from the *ith* previous layer unit (input layer if there is only one hidden layer) to the k*th* hidden layer unit.

$$\delta_{pj} = f_j'(net_{pj}) \sum_k \delta_{pk} w_{kj}$$

Summarising then, the three equations which define the backpropagation algorithm are,

$$\Delta_p w_{ji} = \eta \, \delta_{pj} o_{pi} \tag{7}$$

This says that the change in the weights from unit i to a unit j for the pattern p is proportional to the product of the output of unit i and the error to be propagated back from unit j, δ_{pj}, where δ_{pj} for an output unit is given by

$$\delta_{pj} = (t_{pj} - o_{pj}) f_j'(net_{pj}) \tag{8}$$

and δ_{pj} for a hidden unit is given by

$$\delta_{pj} = f_j'(net_{pj}) \sum_k \delta_{pk} w_{kj} \tag{9}$$

Note that $f_j'(net_{pj})$ is the derivative of an activation function, (or thresholding function as it sometimes called) which maps the weighted sum to a range of output values. Obviously, a function must be chosen for which a derivative can be found.

The constant of proportionality in equation (7), η is called the learning rate. The larger this value is, the larger the weight changes will be. Thus it is desirable to choose a large value for η, since it will make learning faster. However with very large magnitudes of η, the weights can be changed by considerable amounts which can lead to oscillations in the learning curve. This can be reduced, if a momentum term is included in equation (7). For each weight change, a small proportion of the previous weight change is used and this acts to filter out 'high frequency' weight changes. i.e.

$$\Delta w_{ji}(t+1) = \eta \, \delta_{pj} o_{pi} + \alpha \, \Delta w_{ji}(t) \tag{10}$$

α is a value between 0 and 1 and determines what proportion of the weight change at the last cycle t, is to be used in the next cycle t+1.

2.2.1.2 Applying the Backpropagation Rule

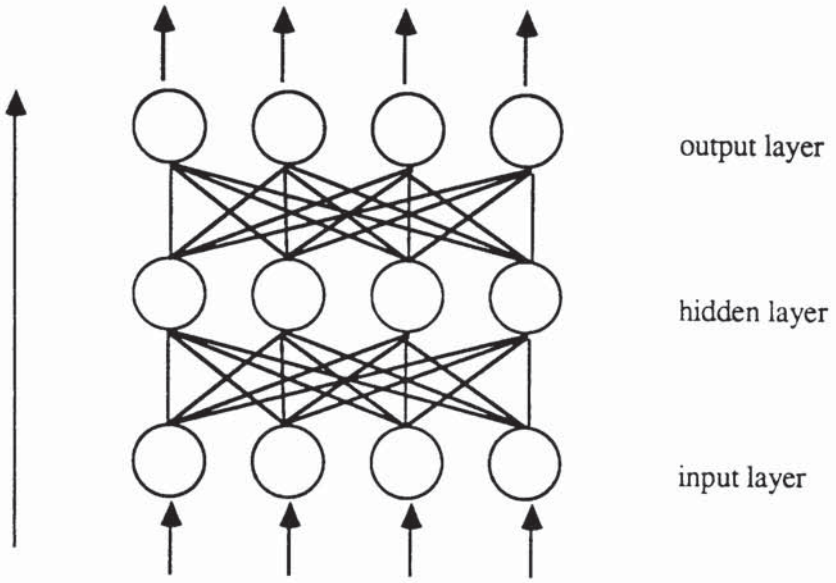


Figure 2.1 A three layer perceptron.

The backpropagation learning procedure is applied in two stages. For each pattern there are two passes through the network. One forward, from inputs to outputs, and the next backwards. It is the backward stage in which the errors are determined and subsequently propagated back down through the net. (Hence the name of the algorithm.) In the forward stage, the pattern is presented and stored in the input layer. A weighted sum is calculated for each hidden layer unit and then the same is done at the next layer, which in this case is the output layer (see figure 2.1). For the backward stage, the computed output values are compared with the desired values for the current input pattern, and an error signal is calculated from equation (8). This error is used to update the weights from the hidden layer to the output layer. Equation (9) enables an error signal to be generated for the hidden layer units, which can be applied to update the weights from the input layer to the hidden layer units. The algorithm works for any number of layers, but usually it is common to use an output layer, a single hidden layer and an input layer which holds the input pattern.

Each forward / backward phase takes place after the presentation of each pattern in the training set, and the presentation of all of the set is usually referred to in the literature as an *epoch.*

2.2.1.3 The Activation Function

The activation function that seems to have been adopted as a standard for back-propagation, is the following exponential:

$$o_{pj} = \frac{1}{1 + e^{-net_{pj}}} \tag{11}$$

where

$$net_{pj} = \left(\sum_i w_{ji} o_{pi} + \theta_j \right)$$

i.e. the weighted sum of the inputs into unit j.

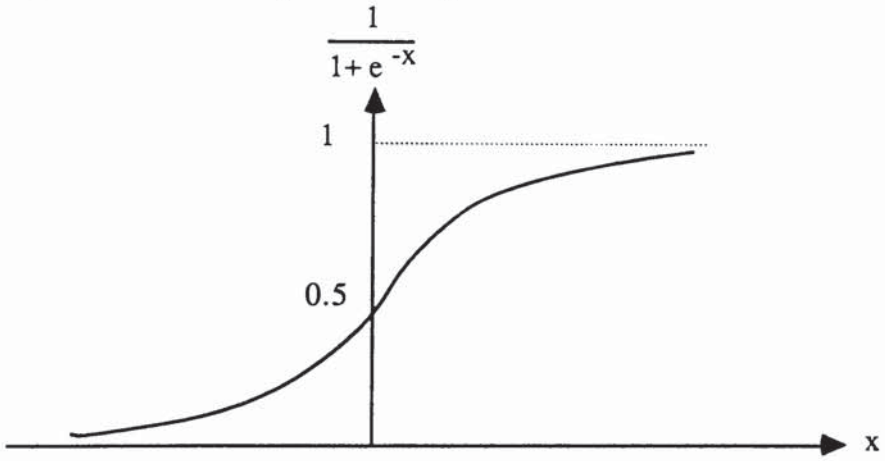


Figure 2.2 The sigmoid activation function

Notice the introduction of the term θ_j. This is a bias term and can be learnt just like any other weight. During learning, it acts to alter the location of the middle of the graph. It is particularly useful when net_{pj}, the weighted sum, has a very high or very low value which puts it on the asymptotes of the sigmoid. If the need to move away from these asymptotes arises, then many weights will have to change to achieve this. By changing

the bias term however, the same result can be achieved by altering, effectively what is just a single weight (see figure 2.2).

The derivative of this function is used by equations (8) and (9). The derivative of this function at an output value of o_{pj} is from equation (11),

$$f_j'(net_{pj}) = \frac{\partial o_{pj}}{\partial net_{pj}} = o_{pj}(1 - o_{pj})$$

this can be substituted back into the equations for δ_{pj}.

$$\delta_{pj} = (t_{pj} - o_{pj})\, o_{pj}(1 - o_{pj}) \qquad (12)$$

$$\delta_{pj} = o_{pj}(1 - o_{pj}) \sum_k \delta_{pk} w_{kj} \qquad (13)$$

2.3 The Hopfield Net

Training of a net can commence in one of two methods; one, where the network is presented with the required input / output vectors to learn. This is known as supervised learning; and two, where the network is presented with only input patterns and then allowed to organise itself into its own optimum state. The Hopfield net can be configured to undergo either supervised or unsupervised learning.

Many versions of the Hopfield net have been developed, but essentially the net can be used in two different ways; either as an content addressable memory (sometimes called an associative memory) (Hopfield 1982) or to solve optimisation problems, such as the classic travelling salesman problem.

2.3.1 Content Addressable Memory

A Hopfield net configured as a content addressable memory consists of a number of units, each taking binary inputs and outputting one binary value. The output of each unit

is fed back into every other unit by a modifiable weight. By careful initialisation of the weights, it is possible for the Hopfield net to have any pattern or patterns stored in its memory. The net can then be used to try to recall unknown patterns from its memory. For example, if the net has been stored with the pattern for a particular letter, then a pattern, say a noisy character can be presented to the net and after a few iterations of a convergence algorithm, the output of the net will 'settle down' into one of the patterns that it has been taught.

Problems with the content addressable memory are that there is a severe limitation on the number of patterns that can be initially stored in a Hopfield net. If too many patterns are used, then the net can easily converge to some spurious output pattern that does not even belong to its set of trained patterns. Also, problems can arise if two or more of the exemplars share many bits in common.

2.3.2 Solving Optimisation Problems

Hopfield and Tank (1986) describe how a net may be used to solve the travelling salesman problem. Briefly, the problem requires a salesman to make a closed tour of a given number of cities in the shortest possible distance, visiting each city only once. An n by n array of neurons can be used to code the solution to the problem, such that columns indicate the order in which that city is to be visited, and the position in the column refers to a particular city. For example, figure 2.3 represents the following visiting order; city b 1st, city c 2nd, etc.

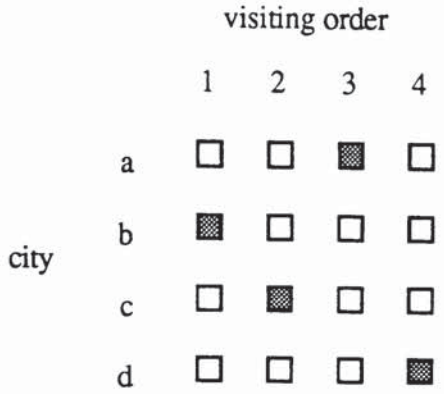


Figure 2.3 The travelling salesman problem

At any one time, only one neuron may be active in a given row or column. This implies the need for some suitable arrangement of inhibitory connections within rows and within columns to ensure that one and only one neuron fires.

Unlike the content addressable memory, the outputs of the neurons are analog, giving real values between 0 and 1. Thus in the travelling salesman problem, the output of a neuron reflects a current hypothesis, that a city is visited in a particular order, and the magnitude of the output represents the strength of that hypothesis. During convergence several conflicting solutions or propositions will be considered simultaneously and this is what gives the net the ability to find optimal solutions with only a few iterations. Hopfield and Tank (1986) point out that, if binary values are adopted instead of continuously variable values, the solutions found are little more than random.

The implications of the net to solve such optimisation problems has some bearing on neurobiological processes. For example, a 900 unit net can solve a thirty city problem, for which there are about 10^{30} possible paths, in as little as only one convergence. As Hopfield and Tank (1986) suggest, this non-linear arrangement of neurons has a natural ability to solve optimisation problems and suggest that this ability may be involved in various biological recognition tasks, such as finding "what three dimensional shape 'best' fits a given pattern in a two dimensional image?" Examples are given of biologically relevant vision problems such as edge detection and stereopsis, which can be formulated in terms of optimisation problems.

2.4 Competitive Learning

Competitive learning is an unsupervised learning paradigm in which training is achieved without any external guidance and the Hopfield net described earlier in section 2.3.2 is an example of this. Although many variations to this type of learning exist, the general rule is that the greater the response of a unit to a particular pattern, the more strongly will it try to shut down other units in the net. The effect of this is that the units are forced to *compete* amongst themselves, so that the unit with the strongest response, will be the best placed to win. Usually, adaptation of weights only takes place for the winning unit. As a result of this, units emerge which learn to respond to particular features found in the input.

Rumelhart and Zipser (1985) present a comprehensive description of the various types of competitive learning methods that exist. Also included, is a short historical survey of competitive learning in past systems. In particular, the perceptrons debate is discussed, but this time more attention is paid to a less well known area of work undertaken by Rosenblatt. "Spontaneous learning" as Rosenblatt called it was a variation on the original perceptron, but this time no external supervision was required. The algorithm that Rosenblatt used was based on updating weights, depending on the response to a particular input pattern. Using this approach, the perceptron had acquired the function of segregating a stream of input patterns into two classes. The patterns from one class had to be similar to each other and different from those in the other class.

The more recent systems that have adopted competitive learning, or their variants, have been due to Grossberg, Kohonen and Fukushima (see sections 2.5, 2.7 and 2.8.2). Rumelhart and Zipser (1985) try to categorise the various types of competitive learning and list four types: the auto associator, pattern associator, classifier and regularity detector. The architecture of these learning systems, is however more or less the same and consists of a hierarchical arrangement of layered units, with connections going from

one layer to the next. Connections within a layer are usually such that there are small clumps or groups of units which are mutually competitive.

Rumelhart and Zipser (1985) report on the findings from some experiments using this learning paradigm. One interesting problem studied is that of learning to develop feature detectors which respond to either a horizontal line or a vertical line in the input space. It turns out that horizontal lines or vertical lines have few pixels in common and because of this, the weights that develop, have lost all information relevant to horizontal or vertical line. The only common pixels available in the input are due to the fact that each horizontal line intersects with every vertical line. It is this similarity that the net learns to respond to. With this in mind, it is shown how competitive learning can be made to discriminate between horizontal and vertical lines. The earlier work used a simple single layer of units, but this proved to be inadequate.

2.5 The Carpenter and Grossberg Net

Carpenter and Grossberg (1988, 1987) propose a self-organising network based on their Adaptive Resonance Theory (ART). The net is designed to cluster and classify its input without the use of any external supervision. The scheme allows for the presentation of various patterns, such as characters and tries to establish whether that pattern already belongs to a known class or not. If so, the pattern is clustered together with that class. If however, the pattern is deemed to belong to a completely new class, then a new cluster is established and the pattern stored as the first member of the class. Theoretically this enables the net to learn new facts very quickly, while at the same time refining its representation of other clusters. For example, if a character, say 'A' is presented to a completely initialised net, then that will immediately form the first cluster. However, an 'A' could be presented again, but this time one that is slightly distorted. The net will consider its existing clusters and try to find a best match. If the match satisfies a threshold, then the new input is included in with the existing cluster. The idea is that

various examples of the same character would be used to improve the existing representation of an 'A'.

Carpenter and Grossberg (1988) compare the ART learning scheme with alternative schemes, and cite several advantages of their approach over others, for example the ability to learn internal top-down expectations, the effective use of all memory capacity and the ability to learn more patterns, without danger of 'washing out' memories of existing ones. In particular, because of the ability to detect a mismatch, the ART architecture is claimed to learn quickly and in real time. Many other learning algorithms need to progress in small steps and are thus more computationally demanding, such as backpropagation.

These claims are quite impressive, but although the ART scheme works quite well for perfect inputs, it runs into problems when subjected to noise. With noisy inputs, it is likely that instead of patterns acting to refine an existing cluster, even a small distortion can lead to a new unit being used to set up a new cluster. A series of noisy inputs can thus cause all available inputs to be rapidly used up.

2.6 The Boltzmann Machine

A potential hazard with any gradient descent method such as backpropagation learning, is the problem of local minima. When converging to a possible solution, the algorithm may not always reach the globally optimum set of weights. In a continually changing landscape of weights, it is not inconceivable that a local minima may be encountered before the global minima is reached. If this could be detected, learning can be restarted from a different set of initial weights, in the hope that starting from a different point on the landscape may avoid this minima. This is not however an ideal solution.

An alternative to backpropagation is the Boltzmann Machine formulation (Ackley et al., 1985). Like backpropagation, Boltzmann learning can be applied to multiple layers of

units. The links between units in the Boltzmann machine are bidirectional and symmetric (i.e. the same value in both directions). The output of the units can only take on binary values of 0 or 1 and is calculated as a probabilistic function of the state of its neighbouring units and the weights to them.

The Boltzmann learning algorithm can develop the same diffuse representations that back-propagation can, but is specifically designed to avoid local minima. The learning algorithm is based on the Boltzmann distribution and uses noise to escape from local minima.

The probability that the output unit is in either the 1 or 0 state depends on a probabilistic function. A parameter of this probability is a term that is analogous to a temperature. Thus the training sets are presented initially with a 'high temperature' and the total error (or energy as it is referred to) is allowed to approach 'thermal equilibrium' before the temperature is lowered. The temperature is then lowered slightly and learning restarted. Ackley et al. (1985) show how this 'simulated annealing' can lead to the global minimum energy being reached in all but a few cases.

The Boltzmann machine is one of a few new learning paradigms capable of choosing internal representations that provide the necessary mapping between input and output patterns. The advantage that Boltzmann learning has over backpropagation is that it is better suited to cope with local minima. This advantage may not however be very significant. Rumelhart et al. (1986) recognise the potential danger of local minima, but report that in their empirical studies, which involved several differing tasks, problems with local minima were never encountered.

Due to the process of simulated annealing, in which the same learning cycle is repeated for gradually decreasing temperatures, Boltzmann learning is even slower than back-propagation. As a result, backpropagation is nearly always the preferred alternative in any practical application.

2.7 The Kohonen Self-Organising Feature Map

Kohonen (1982) developed a self-organising network capable of finding regularities and structure in input data and representing these regularities as orderly arrangements of feature detectors, just like those found in the brain. The self-organising feature map is a competitive network, which can be taught without any guidance from a trainer, that is, it is unsupervised. Figure 2.4 shows a model of a Kohonen net. It consists of an array of output units and some input units. As well as the connections between output units and input units, there also exist a large profusion of connections between output units, providing lateral interactions, causing each unit to compete with every other.

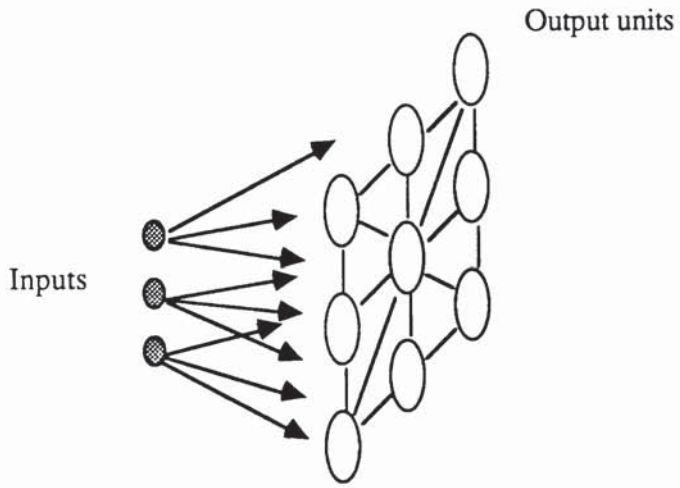


Figure 2.4 A self-organising feature map.

2.7.1 Feature Maps in the Brain

Coding of features in the brain is known to proceed in an orderly fashion, such that neurons anatomically close to each other, respond to features of the input which are physically similar. This 'ordering' results in neuronal maps which are able to describe topological relations of input signals using either a one-dimensional or two-dimensional array of neurons.

2.7.1.1 Evidence for Feature Maps

It has been known for a long time that various areas of the brain are organised according to differing sensory tasks. For example, a one-to-one topographic mapping exists between the retina and the area of the brain known as the primary visual cortex. However, not all sensory maps are due to simple one-to-one mappings. In the auditory cortex, there exists a tonotopic map in which the spatial response of cells, corresponds to an almost logarithmic variation in acoustic frequencies. Kohonen (1988) sees the formation of topology preserving nets such as these, as central to the operation of the brain.

2.7.1.2 How are Feature Maps Formed?

Although many of the basic brain structures are present at birth, the ability to learn from experience indicates the existence of some mechanisms for enabling adaptation. Kohonen (1988) presents a self-organisation algorithm which can be applied to a single layer of neurons and demonstrates how a linear array of neurons, laterally connected, can be made to develop a response to certain characteristic features in the input. Furthermore a two-dimensional array of neurons is set up, with lateral connections. The results from this simple model show how the response of neurons is found to cluster together in 'bubbles of activity', such that the greatest response is from the neuron in the centre of the 'bubble', with the response of neighbouring neurons decreasing with distance.

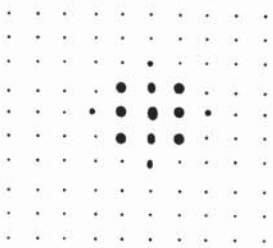


Figure 2.5 An 'Activity Bubble'

It would seem that, the neurons have organised themselves to respond to similar features of the input. It must be noted that lateral feedback, whether inhibitatory or excitatory, plays an important role in self-organisation.

2.7.2 Lateral Connectivity

The cerebral cortex is a densely packed array of neurons. Connections between neurons exist between layers, so that the output from one level of neurons can be passed on to other levels. But connections also exist between neurons in the same layer. These connections are termed *lateral connections* and cause neurons within the same layer to interact or compete with each other. An important question that rises, is to what extent do these intra-layer connections influence the response of individual neurons?

It is suggested in Kohonen (1988) that these lateral interactions are a function of the distance between neurons, so that any one neuron will only influence a limited number of neurons within its vicinity. In particular this interaction follows what is commonly known as a "Mexican-Hat" function.

2.7.2.1 Lateral Interactions and the "Mexican-Hat" Function

Kohonen (1987) presents a good description of biological functions in the brain and points out that it is known that tightly packed cortical neurons have many connections between closely neighbouring neurons, but also have long range connections to other

neurons or groups of neurons. One possible arrangement of lateral connections is that given by a derivative of gaussian function, or "Mexican-Hat" as it is commonly known.

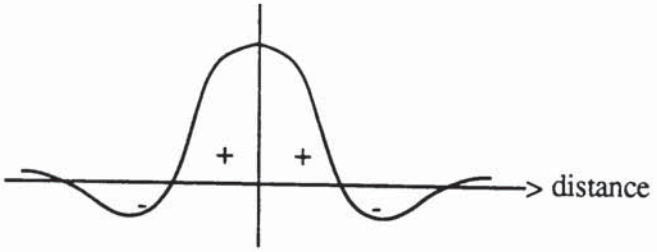


Figure 2.6 The "Mexican-Hat" function

Thus nearby neurons have excitatory interconnections, but connections to more distant neurons are inhibitory. Notice that very long range interactions according to the derivative of gaussian, are only slightly excitatory and so often ignored. The effect of this on the topology will be that small clumps of neurons will emerge that respond strongly to certain patterns whereas more distant neurons, because of inhibitory connections, will be forced to adopt responses to different patterns or features. If during training, the input pattern is varied smoothly, then it turns out that the maximum response in the feature map also moves smoothly around the net.

2.7.3 Practical Implementation of a Kohonen Feature Map

To implement a Kohonen feature map directly would be problematic. It should be recalled that because each neuron in the map is connected to every other, as well as to the input units, the total number of weights would increase in the order of N^2, where N is the number of neurons. Furthermore, complications regarding the order in which laterally connected weights must be updated must also be considered. Kohonen (1988) suggests that it is possible to achieve self-organisation using a simpler technique. According to Kohonen, "without simplifications, the computations become intolerably heavy."

Instead, Kohonen (1988) describes a simplified approach which removes the need for directly implementing lateral connections. The approach relies upon defining a neighbourhood of neurons of a certain size, around each neuron to be updated.

2.7.3.1 The Simplified Self-Organising Algorithm

Consider a two-dimensional grid containing i neurons, each with n input weights. Then the Kohonen self-organising algorithm can be described in two steps:

(1) Locate the best-matching unit in the array of neurons

(2) Increase matching at this unit and its neighbours.

This would be done for each pattern in the training set and the training set would be presented many times.

If an input pattern is defined by the vector

$$x = \left[\xi_1, \xi_2, \ldots . \xi_n\right]$$

and a weight vector associated with neuron i is defined by

$$m_i = \left[\mu_1, \mu_2, \ldots . \mu_n\right]$$

Then the maximum response can be regarded as being the neuron with the best match between these two vectors. Thus step (1) becomes a matter of locating a unit, (let this be unit c), having the minimum Euclidean distance between the vectors x and m_i i.e.

$$\| x - m_c \| = \min_i \| x - m_i \|$$

This allows the neuron having the closest match for a given input vector to be located within the grid. Thus the actual output of a neuron is not required, which is why the

algorithm does not involve a weighted sum calculation. Step (2) then states that the weights of neuron m_c and its neighbours be updated as follows :

$$m_i(t+1) = m_i(t)+\alpha(t)\left[x(t) - m_i(t)\right] \quad \text{for all neurons in the neighbourhood i.e. for all } i \in N_c(t)$$

$$m_i(t+1) = m_i(t) \quad \text{for neurons outside the neighbourhood}$$

Where $N_c(t)$ is a function which determines the size of the neighbourhood and $\alpha(t)$ is effectively a gain term. Both are functions of the discrete time t such that in general they decrease with time, so that the neighbourhood starts large, and eventually shrinks so that no neighbours are updated; the gain term likewise decreases in magnitude according to some function. Kohonen (1988) fails to suggest any actual functions which could be used. Instead, the approach adopted is to experiment and rely upon experience to find suitable functions. However some mention is made of modulating the gain term with a Gaussian function, so that $\alpha(t)$ is excitatory for close neighbours, and inhibitory for distant neighbours.

2.7.4 Examples of Topology Preserving Maps

One of Kohonen's simple tests shows how a square grid of neurons when presented with x,y locations taken randomly from a square, of dimensions 0,0 to 9,9 , organise themselves so that one corner of the map responds to location 0,0 and the opposite corner responds to say 9,9 with the response of intermediate neurons varying smoothly to accommodate all other locations in between. In fact the mapping between locations in the square and the location of neurons responding to a particular location, follow an exactly one-to-one relationship.

An application of this approach has been demonstrated in Kohonen (1988a), in which Kohonen reports on the results of several years work towards a 'phonetic typewriter'.

The aim is to develop a speech recognition system, which can be trained to recognise human speech and output phonemes that comprise each utterance. After training, the net is found to develop a regular ordered structure of neurons, each having a response to a single phoneme, such that neighbouring neurons respond to similar sounding phonemes.

Although the system performs quite well, for example it can generate output in 'near real time', Kohonen (1988a) admits that the system falls short of expectations and that problems regarding recognition accuracy and arbitrary speakers exist. However, the results successfully show how the highly adaptive powers of self-organisation can be effectively used.

2.8 Two Examples of Successful Neural Net Applications.

2.8.1 NETtalk

NETtalk (Sejnowski and Rosenberg, 1986) is a three layer perceptron designed to read given text out aloud. Input to the net is in the form of English words and the outputs generated are the phonemes necessary to correctly pronounce the given words. The output of the net can then be passed directly to a speech synthesiser. Training of the net requires the presentation of text, along with the associated phonemes and back-propagation is used to adjust the weights in order that the correct mapping can be developed.

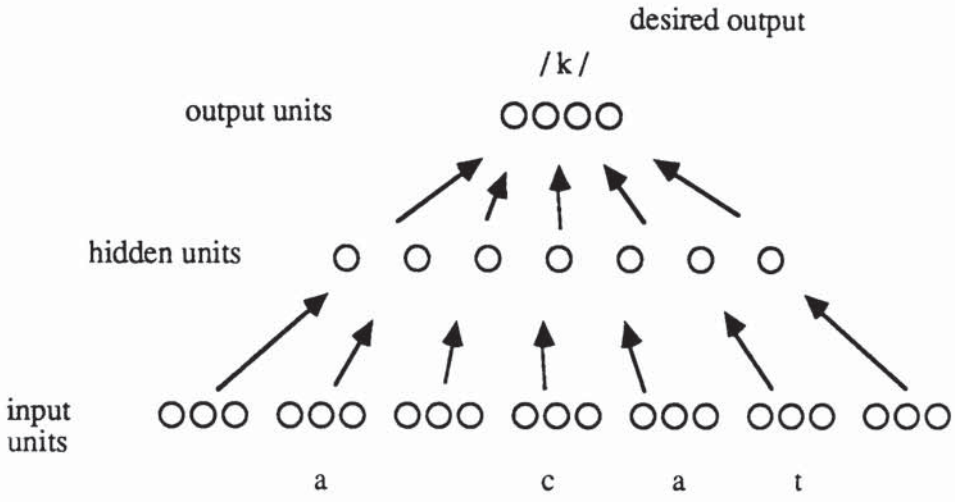


Figure 2.7 Schematic diagram of NETtalk

The net has 203 input units, 80 hidden units and 26 output units. The training set is comprised of a corpus of about 1000 different words. During training, the characters making up the words cycle through a seven letter ascii window and each time, the phoneme associated with the character at the centre of the window is coded as a target value on the output. Note that the surrounding characters are used for context information.

A tape-recording of NETtalk undergoing training has been made and the results are strangely similar to the stages of learning that a child might undergo. For example, initially the net does not know that there should be silence when a space is encountered, but it quickly learns this. As training progresses, the initial babbling becomes quite clear spoken English. The final trained net can not only pronounce the words that it was trained on, but can generalise well to unknown words.

NETtalk is a highly successful application of the multi-layer perceptron and is an example of what can be achieved. It should be noted that although obvious parallels can be drawn with the speech learning process in humans, it should be emphasised that the network, and for that matter backpropagation, is not meant as a biologically plausible explanation of learning in natural systems.

2.8.2 The Neocognitron

Fukushima (1988) proposes a neural net model for recognising handwritten characters and uses the model to gain some insights into how complex mechanisms within the brain may be organised. The model consists of several hierarchically organised layers of neurons in which simpler features are initially extracted from the input, and then integrated into more complex ones higher up. As well as having forward connections, the neocognitron also relies on backward paths. This is not possible for the backpropagation algorithm and the multi-layer perceptron. The forward paths control the bottom-up flow of information whereas the backward paths provide top-down recall of recognised characters.

The neocognitron employs a self-organising training algorithm to learn characters. This is based on a type of competitive learning paradigm in which only the weights for the most responsive neuron within a group are modified, all others being inhibited.

The model is quite complex and has been developed over several years. Originally it started off as a much simpler pattern recognition device; the cognitron (Fukushima, 1975). This early model, like most at the time, was incapable of recognising translated or distorted patterns. In fact, the same character presented at different positions on the input, would be recognised as different patterns.

Currently, the neocognitron is far more complex and impressive than its predecessor, as Fukushima reports (1988). The neocognitron can recognise position shifted, rotated and noisy characters. It can also cope with changes in scale and because of its backward, or top down connections, it can recognise individual patterns presented simultaneously for the same image. Thus the model can segment the image, and has acquired the function of selective attention.

It must be recognised that any abilities possessed by the model are due to the physical architecture of the net and not wholly due to learning. Many details concerning the

connectivity of the net have been 'hand built'. For example physiological findings suggest that neurons in intermediate layers would be expected to have a wider receptive field than the neurons early down the pathway. Hence the connections from one layer to the next have been arranged to make sure that this can take place. Self-organisation just completes the net by determining what features are to be detected in each receptive field.

The 'hand built' connections are designed to cope with only small distortions, thus if a pattern is subjected to large translations or rotations, then a feature that a particular receptive field is expecting, may actually appear within the receptive field of another neuron and hence recognition will fail. As Fukushima (1988) comments, the net is unable to recognise two ones when presented with '11' if the digits are too close together, since then, features from both digits fall into the receptive field of the same cell and discrimination becomes impossible.

2.9 Summary

There exists a general belief that massively parallel networks of simple computing elements are somehow suited to recognition tasks. The rationale underlying such beliefs is the knowledge that biological systems are known to exist, such as the human brain, which consist of large numbers of highly interconnected neurons capable of visual or speech recognition.

This belief, coupled with the recent resurgence in neural network research, has seen a general trend away from traditional or AI type approaches. Instead, neural networks now provide more biologically plausible means of achieving the same task, and it is this attraction that has fuelled the motivation behind much current work.

Early neural systems such as the cognitron (Fukushima, 1975) and work by Kohonen has shown how various neural architectures can be configured to achieve particular recognition tasks. Since then, other neural paradigms have also emerged. Competitive

learning, self-organisation, multi-layer perceptrons etc. The most significant of these new paradigms has been the backpropagation learning algorithm and this is borne out by the frequency with which backpropagation or its variations, appears in the literature.

One of the major attractions of the algorithm, is its ability to find a mapping between a set of arbitrary input patterns and associated output patterns. This flexibility means that back-propagation can be applied to a variety of problems and the previous sections give an idea of its diversity.

If the restriction of using only feed forward architecture can be met, then back-propagation will in most cases be able to provide a solution for mapping any continuous function from inputs to outputs. Backpropagation has been widely applied, but despite this a number of performance issues still need to be addressed. For example, what happens when the learning rate is altered, can changing the momentum factor be used to increase the learning speed and what effect does modifying the network topology have on the ability of the net to converge to a solution? If the answers to these and many other questions can be found, will they hold true for *all* problems, or are they problem dependent? Currently a lot of work is under way to investigate the behaviour and properties of the backpropagation algorithm, but as yet researchers have only been able to come up with simple rules of thumb (Ahmad, 1988). No hard and fast rules have been discovered. The approach to the use of the backpropagation algorithm is very much an empirical approach with the experience gained guiding subseqent work.

Chapter 3

Computational and Perceptual Approaches to Computer Vision

3.1 Introduction to Computer Vision.

Computer vision has been the subject of much research over the past thirty years or so. Consequently a large volume of literature has been published in this area. Fortunately, some good and fairly comprehensive introductory papers and texts have been written. Besl and Jain (1985) present a comprehensive survey of various aspects of three dimensional object recognition, including components of a recognition system and the characteristics of an ideal system. An ideal system, they say, must be able to handle sensory data from arbitrary viewing directions, cope with arbitrarily complex real-world objects and analyse scenes quickly and correctly. Chin and Dyer (1986) present a comparative study of various model based object recognition algorithms, including in this a description of object modelling, feature extraction and matching as these are the major components of any part recognition system. The authors compare and contrast the different approaches that have been taken to these components in several studies. Brady (1982) presents a very comprehensive study of the computational approaches to computer vision. In addition to the problems of feature extraction, object modelling etc, Brady (1982) also looks at the somewhat more low level tasks of segmentation and edge detection and describes some significant approaches.

3.1.1 Levels of Processing

It is generally regarded that the task of visual recognition can be broken down into three distinct levels of processing; low, intermediate and high level.

Computational approaches to low level vision are concerned with the processing of the initial pixel image. The aim is usually to recover relevant intensity change or *edges*. The subject of edge detection has been extensively studied and many different approaches

have been developed. Of the more famous edge detection algorithms are the Canny (1986) edge detector, Marr and Hildreths' (1980) theory of edge detection based on biological findings and the Huekel (1973) approach to edge detection, using local operators to recognise lines and edges.

It is often argued that to interpret a two dimensional image of a three dimensional scene, it is necessary to recover information relevant to three dimensions. Such feature extraction is an example of intermediate level processing and many computational approaches to solving these kinds of problems have been proposed. A general motive in 3D visual systems is to try to extract depth information from the image. Typically, features are not simple 2D lines or circles, but actually represent surfaces or line orientations in 3D (Barrow and Tenenbaum, 1981). An intermediate representation of the image is generated which is not a simple 2D representation, but also contains some 3D data. This is something that Marr refers to as a 2.5D sketch, and it is his belief that similar representations are also used by biological vision systems. Many techniques exist for retrieving shape from intensity images, for example shape from stereo images, shape from optical flow, or shape from shading (Brady, 1982). Each is applicable to images taken under suitable conditions. However, depth can also be obtained directly by obtaining range data, instead of using intensity images (eg. Grimson and Lozano-Perez, 1984).

High level vision is the final stage of processing and involves object modelling and matching. In order to recognise an object, an internal representation of it must first be generated. The representation can be of many types. Typically, object models can either be knowledge based or geometric descriptions similar to those used for computer aided design. Constructive solid geometry, wire frame representations and octree methods are three such CAD modelling techniques which are described in Besl and Jain (1985). Because of their rigidity, geometric models are incapable of efficiently describing general classes of objects. Knowledge based approaches attempt to capture more than just the shape of the object. For example, a semantic net can be used to describe, implicitly the

shape and structure of an object, enabling that description to be valid for a whole range of objects. Connell and Brady (1987) use such an approach and suggest how it could be used to describe general classes of objects. Purely symbolic descriptors however are not efficient in most cases, so often more than one modelling technique is used. ACRONYM (Brooks, 1983) uses generalised cones (primitive geometric solids) in which cone descriptors appear in a frame describing the object at a given hierarchy. This allows explicit geometric descriptions of the object but at varying hierarchies.

3.2 Some 3D Vision Systems

Computer vision comprises three levels of processing. Much of the work presented in this thesis concerns itself with the intermediate or feature extraction stage. The following three vision systems serve to illustrate the differing approaches that have been taken to feature extraction and how such features can be used for higher level recognition.

3.2.1 ACRONYM

Objects in ACRONYM (Brooks, 1983, 1981) are modelled by primitives known as generalised cones. The feature extraction stage tries to find ribbons and ellipses in the image. A ribbon is a section across the length of a cone and an ellipse is a cross or end section of a cone. Higher level 3D reasoning and matching in ACRONYM, is based entirely on matching 2D ribbons and ellipses, that is, although the models in ACRONYM are three dimensional, matching involves comparing the ribbons and ellipses retrieved from the image, with those rendered from the 3D object models. The heart of the matching system, is a constraint manipulation system which propagates constraints throughout the prediction and interpretation stages. The predictor system itself contains some 280 production rules which it uses to generate possible hypotheses.

One vital feature lacking from ACRONYM is a verification stage. At no point are the final hypotheses checked for correctness. This lack of feedback would have a profound effect on the robustness of the system and perhaps this is why ACRONYM has only been tested on aerial images of aircraft on the ground, but never on aircraft images taken at ground level.

3.2.2 A Three Dimensional Part Orientation System (3DPO)

3DPO (Bolles and Hourad, 1987) is an industrial vision system capable of bin picking tasks. The system uses a feature classification network which is associated with each object. The purpose of the network, which is computed off-line, is to act as a decision tree and to guide the feature extraction process. Typically, features would be distinctive patterns or marks on the object, such as holes or slots. If the features comprising the classification network are chosen carefully, it is possible to hypothesise and verify the detection of an object using no more than three or four features from the image. In 3DPO, the recognition strategy is to initially locate a key feature, known as a focus feature and then to add one feature to the tree at a time, until it becomes possible to confidently identify and locate the object. Selection of the focus feature depends upon a number of factors namely its uniqueness, cost of detection, expected contribution to recognition etc. Although these parameters seem intuitively correct, they are not consistent with visual psychology and do not attempt to reflect any obvious characteristics of human perception. For example, problems are encountered if one of the features in the feature extraction network, is occluded. Ideally, the features sought should have geometric properties which are invariant to viewpoint. The next vision system to be described, attempts to solve this problem.

3.2.3 SCERPO

SCERPO, which is an acronym for Spatial Correspondence, Evidential Reasoning and Perceptual Organisation, is a complete 3D vision system devised by Lowe (1985,1987). The system comprises several major components and as its name implies, directly exploits ideas from perceptual psychology. SCERPO is capable of performing bin-picking tasks and like 3DPO can handle multiple objects and occlusion. Lowe (1987) proposes a novel and conceptually very interesting alternative to feature selection, based on perceptual organisation. He argues that because the orientation of objects is not known, then it is not always effective to look for features peculiar to a given object, when such features will obviously be very viewpoint dependent. Instead, the chosen features should remain stable over varying viewpoints and not arise due to coincidental alignment. Lowe recognises that the geometric properties of parallel, collinear and end-proximate lines obey this viewpoint independence constraint and suggests that this is exactly how perceptual organisation would seem to work in humans.

Objects in SCERPO are represented by wire frame models consisting of straight lines only. As well as the model, there is a list of associated features ranked in order of decreasing perceptual significance. Each feature detected must adhere to the viewpoint invariance condition, that is, the lines comprising the feature must be parallel, collinear, or their end points must be in close proximity. This gives rise to a set of primitive perceptual groups consisting of two lines each. However a pattern of only two lines will not provide sufficient constraints to achieve recognition and so further constraints can be imposed by combining groups together to generate more complex groups, for example, two sets of parallel lines could form a trapezoid. However there is still no way of knowing whether one group is more significant than another. Lowe solves this by deriving mathematical grouping operations which can assign a numerical value to the significance of a given perceptual group. Since these grouping operations effectively consider two lines at a time, the complexity of considering all combinations would be somewhat impractical. Lowe realises that as the significance is inversely proportional to

the proximity between the lines, regardless of any other characteristic, then it is possible to limit the search to small regions around each element. The matching process proceeds by attempting to match perceptual groups from the model against those found in the image. Each such prediction is followed by verification, something not present in ACRONYM.

SCERPO is a successful and robust system and this can be attributed in part to its use of perceptual groups. Many of the ideas and approaches adopted in the thesis stem from techniques employed by SCERPO.

3.2.4 Use of Perception in Vision Systems

Perceptual organisation has also been successfully used in other systems, notably by Burns and Kitchens (1987). They show how a prediction hierarchy (i.e. a decision tree) can be automatically pre-compiled in terms of perceptual groups. The careful selection of groups is seen as a way of recognising different objects. This idea is similar to the feature extraction network used in 3DPO, but the features comprising the network or tree are now perceptual groups possessing a certain degree of viewpoint invariance, rather than object based features. Burns and Kitchens support this with an example of a decision tree capable of identifying five different geometric solids.

It is interesting to note that some of the production rules in ACRONYM are also based on perceptual organisation, as are the segmentation rules used by Levine and Nazif (1985). This approach to segmentation, groups together disjoint segments, which are felt to have been a single feature in the real scene. For example, collinear lines whose endpoints are in close proximity are labelled as one. This is very similar to the instances of collinearity found in Lowe's SCERPO. Perceptual organisation has also been considered by the team working on the VISIONS image-understanding system, and their hope is to develop knowledge-directed perceptual organisation mechanisms for complex three dimensional shape representations (Hanson and Riseman, 1988).

An apt use of perceptual grouping is in the field of rock crystal analysis, where the shapes to be studied naturally possess a high degree of symmetry and regular form. Thomson and Claridge (1989) show how the properties of continuity, symmetry, closure and so on can be applied to a computer vision system designed to study the order of crystal growth in an image of a rock slice.

3.3 Biologically and Perceptually Motivated Approaches.

3.3.1 Introduction

Traditionally, approaches to computer vision have been from a computational and mathematical point of view. However, an increasing number of approaches motivated by the study of visual psychology or by biological findings, support the belief that the use or emulation of perceptual functions can provide many advantages over previous computational methods. SCERPO certainly shows this to be the case.

A similar view was held by Marr (1980), who tried to provide computational theories that would be compatible with neurophysiological findings rather than finding ways of mimicking brain functions. He put forward the theory that several levels of processing are required to accomplish the task of visual recognition and suggested two intermediate representations that could help achieve this. The *primal sketch* as he called it, is simply an edge image which is used to determine the *2.5D sketch*. This level of coding is used to describe the properties of visual surfaces, such as their distances, slants, overlap, etc. and is so called because it contains only partial information relating to the three dimensionality of the image. From the 2.5D sketch, which contains limited depth information, it is then possible to describe the shape of any objects present in some primitive form. Marr and Nishihara (1978) discuss some of the issues involved in devising 3D shape representations and proposed an object-centred, hierarchical scheme using generalised cylinders as the primitives. An implementation of a similar scheme can be seen in the ACRONYM program (Brooks, 1981).

Marr suggested that such representations may also be used by the human visual system, but rather than trying to understand how the neural circuitry might implement such transformations, he devised computational methods for generating these representations. A good example of this is Marr's approach to edge detection (Marr and Hildreth, 1980), which is used to determine the primal sketch. Although the theory is quite mathematically involved, it turns out that neurons found in the early stages of biological vision systems have just the receptive field properties that the theory demands.

3.3.2 The Influence of Biological Findings on Computer Vision

Pioneering work by Hubel and Wiesel (1962), would appeared to have had a lot of influence on computer vision. Their work on the study of neurons in the cat's visual cortex revealed that a form of line coding was taking place such that lines of a particular orientation impinging on the retina, excite specific cortical neurons. Furthermore, it was discovered that the architecture of these neurons is arranged in orderly band-shaped regions, called *orientation columns*, where the orientation preference of neighbouring neurons, differs by only a small amount. The response of a neuron to a given line orientation was not discrete, but graded. So the output of a neuron would reach a maximum firing rate, for lines of the preferred orientation and would be a minimum for lines perpendicular to the preferred orientation. Figure 3.1 shows the response of a cortical neuron which has a preferred orientation in the vertical direction.

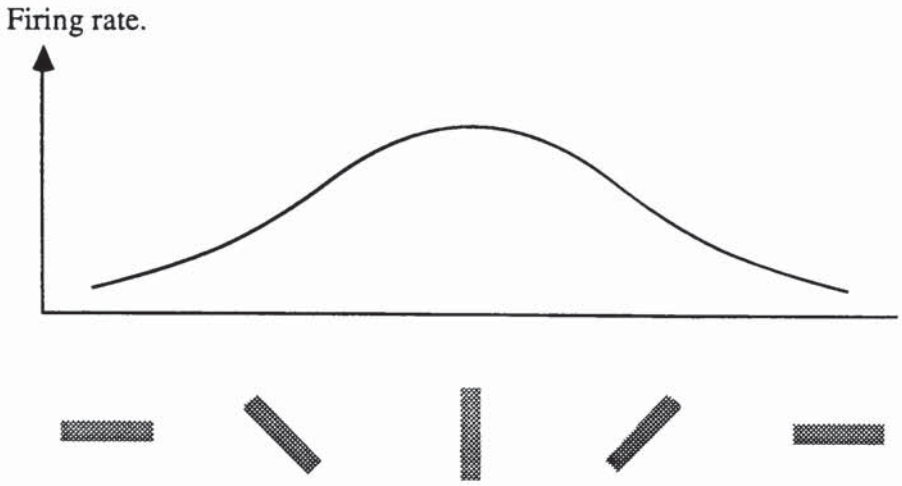


Figure 3.1 Response of a vertically trained neuron to bars of varying angle.

Frisby (1979) reports on work involving kittens which are restricted to seeing only vertical lines for the first few days of their lives. The results suggest that orientation selective neurons in the visual cortex, adapted themselves to their appropriate directions, rather than being fixed rigidly before birth. The point is that there must exist some mechanisms within the brain to allow such adaptations to take place. Chemically, these adaptations manifest themselves as changes in the chemical behaviour of synaptic gaps, linking one neuron to another. It would be interesting to know the factors affecting the choice of synaptic gap and the change of chemical behaviour. Hebb (1949), proposed a simple answer to this. Hebb's idea was that if cell 1 is one of the cells providing input to cell 2, and if cell 1's activity tends to be 'high' whenever cell 2's activity is 'high', then the future contribution that the firing of cell 1 makes to the firing of cell 2 should be increased. This rule is often referred to as Hebb's learning rule.

3.3.3 Linsker's Perceptual Network

The results of all this work were brought together in a simulation set up by Linsker (1988). The question that Linsker asks is, how can a perceptual system develop to recognise specific features of its environment, without actually being told which features it should analyse? The simulation consists of a layered adaptive network, with forward

connections only. The input represents the visual world, but because Linsker is interested particularly in the ability of feature detecting cells to emerge even before birth, this 2D input consists of a random activity of input cells (resembling snow, or noise on a television screen). The cells in the network have a simple linear response; i.e. the output is the linear combination of several inputs, each being weighted by a modifiable connection strength.

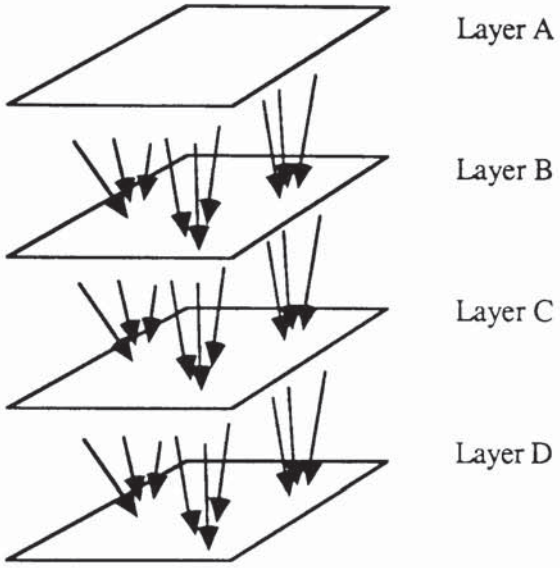


Figure 3.2 Feed forward architecture of Linsker's perceptual network.

The emphasis is on the ability to develop various feature analysing cells and not on modelling a biologically accurate network. Thus many complicating factors have been omitted. For example, feedback connections are known to exist, but have not been included in the model.

Each layer takes input from a small *receptive field* of cells from the previous layer, where layer A is the input layer. The adaptation rule that Linsker employs is a version of the Hebb learning rule described earlier. Although this rule is very simple, it leads to some surprising results. In fact, after the simulation has terminated, and each layer allowed to *mature* (that is to become stable), an examination of the various layers shows that the cells of each layer have developed particular feature analysing properties. Briefly, layer C

cells are found to respond preferentially to bright circular spots centred against a dark background on the cell's receptive field. Other cells develop an opposite response, that is, they respond to a dark disc against a bright background. Cells having these type of properties are known to exist in the mammalian visual system (Hubel and Weisel, 1962).

In layer D, slightly more complex feature analysing cells emerge, which are orientation selective. These cells respond to edges or bars at a particular orientation. Furthermore, cells having the same orientation preferences appear grouped together in small patches or regions in the layer. Linsker comments that the random organisation of the patches is probably due to the lack of lateral connections, which would be present in a natural visual system. As Hubel and Weisel (1962) point out, the organisation of such orientation cells in the cat's visual cortex is very orderly and actually forms columns of equal orientation cells.

The conclusion that can be drawn from this work, is that even using a simple feed forward arrangement of cells and obeying Hebb type adaptation rules, it is possible to simulate quite complex behaviour which is consistent with biological findings. Linsker's explanation of the emergence of feature analysing cells is in terms of the optimisation properties of the Hebb rule. Put simply, each layer is maximising the amount of information preserved from one layer to the next.

This work is one of relatively few which has explicitly attempted to simulate real perceptual networks and although quite successful, still leaves further work to be done. As Linsker says, studies of this nature should not only complement experimental neuroscience, but may also provide the understanding necessary to develop useful artificial perceptual systems.

3.3.4 Endstopped Neurons and Curvature

Orientation columns provide a means of encoding lines and edges within the visual cortex. An interesting question that might be asked is, how does the cortex handle curvature? An answer to this may come from the presence of what are known as *endstopped* neurons. These hypercomplex cells, first observed by Hubel and Weisel (1962) are thought to respond to lines of a specific length.

Dobbins et al. (1987) propose a mathematical model which provides a plausible explanation of how such cells might be involved in encoding curvature. Zucker (1988) takes this approach a step further and discusses how such biological evidence can provide valuable insights into the development of early visual tasks. Zucker illustrates this by showing how the analysis of biological orientation selection processes and the idea of endstopped neurons, can be used to develop an approach to curve detection algorithms.

3.3.5 The Hierarchical Structure Code

Orientation columns within the visual cortex are arranged such that a small section of the cortex (about 1 x 1 millimetre) contains all the neural "circuitry" to encode all possible line orientations that could arise in a small region of the visual input. One might therefore wonder, how such a seemingly unlimited number of differently running contours could be processed by a finite number of cortical neurons? An investigation of this problem was carried out by Hartmann (1987,1985). The results showed that it is possible to encode any line contour or shape using a finite number of differently shaped edge or line detectors. Details of a model, known as a Hierarchical Structure Code (HSC), based on these results, can be found in Hartmann (1987).

The behaviour of the model is very similar to the behaviour of biological visual systems. However, Hartmann (1987,1985) suggests that this does not necessarily imply that the

structure of a biological visual system is similar to the HSC, even though some features of the HSC are biologically plausible. A brief account of the HSC follows.

Encoding of arbitrary lines or shapes is achieved through the use of a number of differently shaped edge and line detectors. The template of each detector is defined in terms of on or off centre-surround cells (Hubel and Wiesel, 1962), and various arrangements of these serve to detect either bright lines, dark lines or regions of edges. Hartmann (1987) describes how such a system can also cope with noise, by encoding the image at varying resolutions. This is achieved by applying detectors of different sizes, and simply eliminating any line segment that is too 'short to fit', i.e. effectively acting as a low pass spatial frequency filter. For example, at the highest resolution each detector takes input from 7 pixels, arranged in a hexagonal fashion. Neighbouring detectors overlap and serve to cover the entire image space. At the next coarser resolution, a detector takes input from 7 smaller detectors, effectively considering them to be pixels. i.e. it does not matter what shape the smaller ones are encoding, only that they are 'on' or 'off'. Each detector regardless of size will only encode the inputs if they are recognised as a continuous contour running through, and this is something that Hartmann (1987) refers to as a linking hierarchy.

The output of Hartmann's system is in terms of trees, in which each node codes the position, shape and size of a detector. Hartmann (1987) also suggests that symbolic shape descriptors can be extracted by traversing the code tree. Examples are given of the kind of descriptors generated, but it is not made clear how these are obtained. An interesting notion that Hartmann (1987) suggests, is that the procedures used to implement the model can be just as effectively implemented as a neural network. The translation of computer operations into neuronal operations is discussed and it is found that the code tree representation is topologically equivalent to the network. Hartmann (1987) attempts to relate components of his hypothetical neural network, to components of biological visual systems.

3.3.6 Perceptual Grouping

Another aspect of modern computer vision has been the study not only of the physical processes but also of the mental processes that take place in cortical systems. Currently, much is known about the physical structure of the brain. However, relatively little is known about the mental processes that take place within it. Consequently many theories and ideas have been put forward, offering explanations of how the visual system may work. Some of the most popular theories amongst psychologists are the so called grouping phenomena, which suggest that important image features are detected in groups. This idea of perceptual grouping has played an increasing role in computer vision systems, notably in the SCERPO vision system.

3.3.6.1 What are Perceptual Groups?

Early visual psychology was dominated by the so called Gestalt laws which are best known for the claim that "the whole is greater than the sum of the parts". Experimental work in visual psychology supports this claim, but what makes perception exhibit these properties is not at all clear. One of the effects now associated with such claims, is the phenomena of perceptual grouping or perceptual organisation.

Palmer (1983) tried to identify the phenomena of perceptual organisation and in doing so identified shape constancy as one of its characteristics. People can perceive the same basic shape at differing orientations, scales, positions and even reflections. This is shape constancy and like most other properties of human vision, it is not easy to see how it is accomplished. One of the theories put forward is *invariant-features*. This hypothesis suggests that to obtain shape constancy, i.e. to be able to recognize the same shape at varying orientations, positions etc., it is necessary to perform recognition using features of the object which are themselves invariant to such transformations. For two

dimensional patterns, angle size, number of angles, relative line lengths, connectivity, continuity etc. will remain invariant. Recognition in three dimensions becomes slightly more complicated since now an additional set of transformations involving the two-dimensional projection of the image must be taken into consideration. Now, only features such as connectedness, number of angles, continuity and not relative line lengths or angle sizes remain invariant. It turns out that these properties are just those that cause the phenomena known as perceptual organisation.

The main effect associated with perceptual organisation is the strong tendency of the visual system to perceive certain collections of features, as single groups. This is the phenomenon known as perceptual grouping and by way of illustration, Figure 3.3 from Roth (1986) shows how a series of unconnected dashes are actually perceived as two smooth intersecting curves.

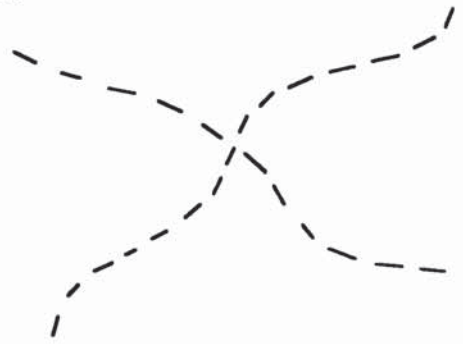


Figure 3.3 An illustration of perceptual grouping

"Perceptual groups" then, is a term used to refer to those image features that collectively are perceived not as disparate fragments, but as a single feature. An explanation in terms of the Gestaltist doctrine would say that the effect observed in figure 3.3, occurs because the image features obey the Gestalt laws of *good continuity*, *proximity* and *similarity*. This is particularly interesting, because continuity, similarity and proximity are analogous to the features associated with the invariant-feature theory described by Palmer (1983). Thus it appears that grouping of features according to the Gestalt laws, may be relevant to shape constancy and invariant object recognition. Lowe (1987) takes up these ideas and

shows how a three-dimensional object recognition system can make use of perceptual organisation (see section 3.2.3).

3.3.6.2 Perceptual Groups and SCERPO

The motivation for Lowe's (1987) approach is the feeling that depth reconstruction is not essential for recognition in three dimensions. Lowe argues that human vision is perfectly adept at recognizing scenes in which there exist very little potential for bottom up depth reconstruction, such as in simple line drawings. This suggests, he says, that although humans do possess capabilities for sensing depth (eg. stereo vision), the need for them is not paramount. Instead it is argued that the preferred approach should rely upon some form of perceptual grouping. The following diagram presents further evidence of the powerful grouping capabilities of human vision. Figure 3.4 (from Lowe, 1987) consists of a random distribution of lines, amongst which are three distinct groups. These groups happen to be instances of parallelism, collinearity and end-point proximity, which are analogous to the Gestalt laws of *similarity*, *continuity* and *proximity*, respectively. Consequently, the components of these features are spontaneously grouped by the human visual system, making them *appear* to stand out above the rest.

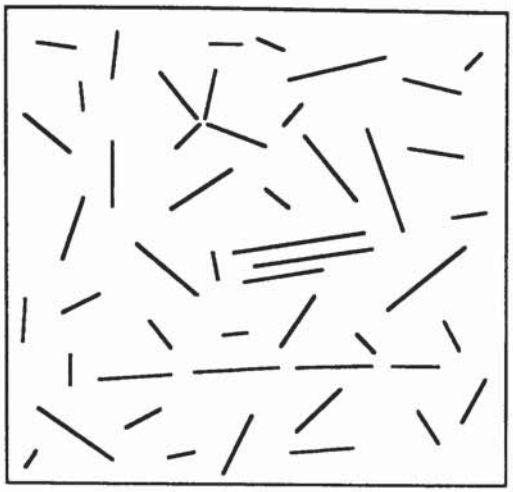


Figure 3.4 Spontaneous grouping

Also the geometric properties of these features are invariant to three-dimensional transformations, making them suitable for object recognition in three-dimensions. Consider for example, a pair of parallel lines in three dimensions. If the viewpoint is varied, then from most angles these lines will still be seen as parallel. The same is true for collinear lines, and lines connected at their end, that is, they will still be seen as such from most viewpoints. This is in fact an example of the *invariant-feature* theory described by Palmer (1983). Only from certain constrained angles will this not hold, for example looking along the plane of two parallel lines.

Thus in SCERPO, Lowe (1987) bases object matching procedures on comparisons between object features which are either parallel, collinear or end-point proximate. The hope is that this will endow SCERPO with some capability to perform recognition of objects in three dimensions.

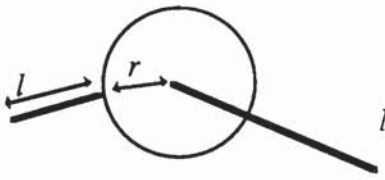
SCERPO starts by searching the image for pairs of lines which contain any of these geometric properties. The problem with this is that, even with a simple image, the number of possible line pairs can become combinatorially high. Fortunately, all grouping phenomena are observed only between lines which are relatively close to each other. This provides a way of limiting the search, since only those lines which are in close proximity are candidates for grouping.

If comparisons were made between just pairs of lines in the object recognition stage, then there would be insufficient constraints to reach a confident decision. For example, a line pair feature may match with a similar feature on more than one different object. So some method is required to reduce the search space. SCERPO attempts to combine together perceptual groups into more complex groups, for example, two sets of parallel lines may be combined together to form a trapezoid; this would obviously generate more constraints than just a pair of lines.

What is particularly interesting about Lowe's (1987) work is that not only is use made of perceptual organisation, but Lowe also tries to quantify how *significant* a particular group may be. This is achieved by basing grouping operations on the geometry of each type of group. The actual derivations are not important, but suffice to say that they are based on calculating the likelihood of each instance occurring accidently.

Each of the following equations relate to two lines at a time.

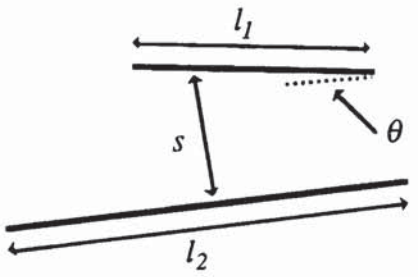
3.3.6.2.1 Grouping According to Proximity.



$$N = \frac{2D\,\pi r^2}{l^2}$$

D is a unitless constant to account for scale independence and N represents the expected number of end points within a radius r from the end of any given line. This makes it approximately equal to the probability of the relation happening accidently and so significance due to proximity is inversely proportional to N.

3.3.6.2.2 Grouping According to Parallelism

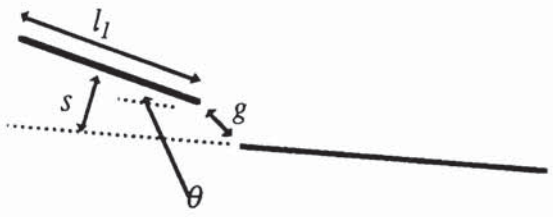


$$E = \frac{4D\theta s l_2}{\pi l_1^2}$$

l_1 is the length of the shorter line, l_2, the length of the longer.

E represents the expected number of lines within the given separation and angular difference and is inversely proportional to the significance.

3.3.6.2.3 Grouping According to Collinearity



$$E = \frac{4D\theta s(g+l_1)}{\pi l_1^2}$$

l_1 is the length of the shorter line.

Here, E represents the possibility of collinearity arising accidently and so the significance of collinearity is inversely proportional to E.

These geometrical formulae enable SCERPO to locate perceptual groups and to estimate the relative significance of each one found. This is important, because to attain the most successful results, matching needs to be performed between the most significant groups detected. These groups will be the least likely to have occurred by accidental alignment and so be more likely to have arisen from the most prominent structures on the object.

3.3.7 Walter's Computer Vision Model and Perceptual Organisation

Much of the work described in earlier sections has tried to make use of biological findings. Experimental neuroscience has been relatively successful when it comes to finding out how the individual components, cortical neurons, operate. It has even been possible to ascertain the overall structure and to some degree, the level of connectivity between neurons. However, even with all this knowledge of the working mechanisms of the system, little has been discovered of the actual processes that are being computed. Hubel and Weisel (1962,1959) have shown that by determining the response of neurons to certain visual patterns, some insight is afforded into how the visual cortex encodes lines at a very low level. The influence that this has had on computer vision has been quite dramatic. Marr and Hildreth (1980), Linsker (1988), Zucker (1988) as well as many others, have all made use of these results in some way or another.

Neurophysiology has been unable to provide answers to higher level processes and because of this, researchers in computer vision still look to the study of visual psychology and psychophysics for possible answers. SCERPO, described earlier in section 3.2.3, took this approach. SCERPO is not alone in this approach and similar ideas of perceptual grouping or perceptual organisation can also be observed in a system devised by Walters (1986). Although not immediately apparent, Walter's "credit assignment" of line components in an image, amounts to a form of grouping based on the role that each line plays in the image. Another interesting feature of the work, is the method of implementation of the model. Rather than adopting traditional programming techniques, Walters has opted to make use of connectionism.

3.3.7.1 Perceived Brightness

Psychophysical experiments (Walters, 1986, 1987) indicate that when observers are presented with simple line patterns, the *perceived brightness* of some line groups is *seen to be* brighter than others. Walters (1987) describes these experiments and what is meant by "perceived brightness". The experiments consisted of patterns of lines presented as low contrast luminance patterns displayed in a dark room. In each experiment, pairs of patterns were displayed, one having a fixed contrast to act as a control and the other having a variable luminance. Each subject is presented with such patterns and is asked to indicate which pattern appears to have the greater luminance. The position of the control pattern alternates and so the subject will never know which is the control pattern. The differences in perceived brightness observed from one subject to the next is found to be small, but quite reliable. Walters (1987) states that all observers tested to date, reported exactly the same difference.

The question that then arises is, what is it about a particular pattern which makes its brightness *appear* brighter or darker than another pattern? Supposedly perceived brightness is related to the shape or the structure of the pattern, but can the particular features causing such effects actually be discovered? Walters (1986) finds two local properties which appear to enhance perceived brightness: line length and local connectedness. Experiments on line length were conducted using a long, fixed length control line and a variable length test line. It was found that as the length of the test line approached the length of the control line, the difference in perceived brightness decreases. This result was interpreted as an increase in perceived brightness for increasing line lengths. Walters and Weissteins (1982) performed a series of experiments which looked at the effect of junction type on the perceived brightness. The results suggested that a strong correlation existed between perceived brightness and the way in which the lines were connected. The results indicate that perceived brightness increases as we move from end-end connected lines, to end-middle connected lines, to middle-middle connected lines. Unconnected lines, subsequently are seen to exhibit the lowest

perceived brightness. Figure 3.5 summarises these findings. The labels A, B and C indicate the level of enhancement that each junction has on the perceived brightness. A is greater that B which is greater that C.

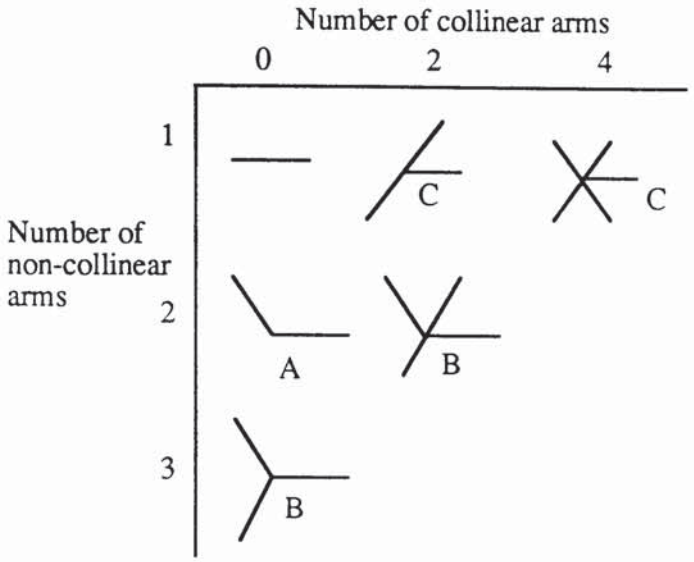


Figure 3.5 Enhancement matrix

3.3.7.2 A Computer Model Based on Psychophysics

Walters's research effort is mainly directed towards the creation of a computer model that will emulate the perceptual functions described above. Although such psychophysical results could be simulated by a computer program, adoption of a connectionist approach was felt to be conceptually more interesting and more relevant to the nature of this task.

Walters describes how the use of 'little calculators' could be used to execute local processes at the pixel level. This involved building a network of connectionist units, each set up to recognise local occurrences of different junctions, as seen in figure 3.5.

Many different types of unit were used, each designed for a particular purpose. One unit that is described in relative detail is the 'U unit'. This operates on each pixel and stores

the perceived brightness of that pixel. The inputs to the U unit are local in nature and include inputs from the line and end connection enhancement networks.

The end result is a network which appears complex, but is doing no more than implementing simple local production rules at each pixel. The input to the network is a line image, and the output is a labelled line image such that pixels belonging to individual lines are labelled with the perceived brightness for that line. The results of the network are found to be in good agreement with the results obtained from actual psychophysical test described in the previous section.

3.3.7.3 Relation to Perceptual Organisation

Walters suggests that the enhancement of perceived brightness of lines depending on their local properties, does not necessarily indicate that perceived brightness plays an important role in human vision. Besides, the differences in perceived brightness are small making it unlikely that the visual system is directly using this property. If different types of connectivity result in different levels of perceived brightness, this indicates the possibility that human vision also treats each type of connectivity differently.

It may be that perceived brightness is a measure of how important a particular feature is to visual processing. In this respect it would be analogous to the calculation of significance values for simple perceptual groups (section 3.3.6.2). If enhancement of perceived brightness is compared with the calculation of significance values in SCERPO, then it becomes apparent that some similarities do exist. In both SCERPO and Walters's model, an increase in line length gives rise to an increase in significance or perceived brightness. Similarities also exist between how connectivity is handled. In SCERPO, connected lines are given a higher perceived brightness than unconnected lines and likewise the perceived brightness of connected lines is also higher than that of unconnected lines.

The implications of this are that if perceived brightness is an indication of how significant various image features are, then it is likely that Walters model is also performing some form of grouping based on these significances. This is where Walters model becomes useful. By simulating the appropriate local operations, the perceived brightness of line patterns can be enhanced. However, whereas human observers were only able to make a judgement on the global perceived brightness of the shape, the computer model is also able to inform about the perceived brightness of any line comprising the object. By doing this, it is found that bounding contours are enhanced more than inner lines. Implying as Walters (1986) says, that the human visual system can also be expected to treat outer contours preferentially over inner contours.

3.4 Neural Networks for Vision

3.4.1 Introduction

The previous section provides an overview of neural nets and describes the properties and methodologies relating to the use of neural networks. The application of neural networks has received increased attention, particularly in the field of vision. The Neocognitron, described earlier (see section 2.8.2) is a real application of neural nets in the visual domain. Qian and Sejnowski (1988) discuss how a net could be used to learn to solve the stereo disparity problem for random-dot stereograms. Hummel et al. (1988) present a connectionist approach to volume recognition, based on recognition by parts or geons (volumetric primitives, not dissimilar to generalised cones, see section 3.2.1). The work addresses the important issue of achieving viewpoint independent recognition. Transform invariant recognition is again addressed in Zemel et al. (1988). A model known as TRAFFIC is presented, which attempts to perform translation and rotation invariant recognition of objects. The model is a hybrid which is built up using ideas from earlier work, such as the generalised Hough transform (Ballard, 1981). Storage and use

of object models is considered by Pawlicki (1988), who proposes a neural network object recognition architecture called NORA.

All these are examples of neural networks applied to visual tasks. The following sections attempt to describe in more detail some of the fundamental work and ideas that have been discovered in the area of neural networks and vision. Of particular relevance are those applications based on backpropagation learning.

3.4.2 Face Recognition

The content addressable memory has already been described in section 2.3.1. Here we look at one realisation of content addressable memories namely the distributed associative memory. Kohonen (1987) supports the view that biological memories operate according to these principles and that the distributed representations involved have previously been encountered only in optical holography. Using adaptable neural networks, Kohonen (1987, 1988) describes how a model of associative memory can be built up using a layer of 'meshed' neurons. With connection strengths being modified by an approximation of the Hebb law.

As a demonstration, Kohonen (1988) carried out some work aimed to show how associative memories could be used for recognition of faces. The network used, was a 'mesh' of interconnecting neurons in which each neuron represented a pixel in the image. The activation of each neuron served to indicate the size or darkness of each pixel. The net was trained with 500 individual images of human faces, using Hebb's law to modify the weights. Thus the information stored was in terms of these 'memory traces' or weights and the retrieval of the original memory could only be achieved associatively, that is by supplying some initial activation or key which could be used to spread activation to other neurons in the net.

The illustrations accompanying the descriptions (Kohonen, 1987,1988) are quite impressive. With just a small portion of the image as a key, such as a horizontal band around the eyes, the net is able to recollect the entire face. Although such abilities appear quite powerful, it must be realised that the net is simply recalling patterns and possesses no abilities to handle translations, rotations etc.

3.4.3 Methods Using Backpropagation

3.4.3.1 Character Recognition

A handwritten character recognition system due to Fukushima (see section 2.8.2) has already been described. The system was complex because it involved many stages or layers of neural nets that were specifically designed to perform particular tasks. eg. the receptive field of neurons at different layers ensured a certain level of tolerance to deformations.

A much simpler network based on multi-layer perceptrons and the backpropagation algorithm is described by Burr (1988). He uses it as a 'test bed' to study the properties of backpropagation. But the application itself is worthy of further attention. Characters are input as a 19 x 19 image. Burr (1988) devises a coding scheme based on a seven segment bar display to code the character. A 13 segment display is used, thus the input to the net consists of only 13 numbers ranging from 0 to 1. Tests were conducted on both handwritten text and handwritten digits. Of 208 handwritten letters (taken from a single writer), 104 were used to train the net and the remaining 104 used to test it. Results showed that recognition rate of untrained characters peaked at a value of 94% when 20 hidden units were used. Similarly in a separate experiment, 100 handwritten digits were collected, of which 50 made up the training set and the other 50, the test set. This time the input was coded like a seven segment display and so there were only seven inputs. The accuracy achieved with untrained digits was 97.5% with 6 hidden units.

The approach shows that the use of a suitable mechanism of encoding the input, can be help to greatly simplify the problem, and learning of the problem.

3.4.3.2 An Autonomous Guided Vehicle.

The flexibility of the multi-layer perceptron and backpropagation is demonstrated by an application presented in Pomerleau (1989). ALVINN is a project at CMU to develop an autonomous guided vehicle using neural networks. The work shows the diversity of the applications to which a standard three layer net can been put, but more seriously is quite a realistic application, since once trained, the nets can operate in real time. The performance of the system is comparable to the best traditional vision systems developed so far, although the speeds reached by the vehicle are only 1/2 a meter a second; which is not much more than a mile an hour! However, of interest is the way in which a three layer net has actually been used to guide provide a mapping between road images, and a guidance mechanism.

The network takes two 'retinal' inputs, one from a video camera and another from a laser range finder. The video camera maps its input onto an 30x32 input in which the activation of each input unit is in proportion to the brightness of the image at that point. The laser finder maps onto an 8x32 grid of input units. These inputs are fed into a hidden layer comprised of 29 units. To tell the vehicle which direction to turn, 45 output units are used to code the turn curvature. The net is fully connected. The coding of this curvature is particularly interesting. Of the 45 units, the middle one indicates drive straight ahead, and the ones to the left and right of this, indicate by how much the vehicle should turn left or right. The desired values of the output units are not binary, but are configured as a 'hump' of activation spread over 9 units, with the preferred output unit at the middle of the 'hump'. Thus, the direction to turn will be the output unit with the greatest curvature. No indication is given why this kind of coding is chosen rather than more precise binary values. However it is suspected that this coding makes the net somewhat easier to train.

Simulated road images are used to train the net, and backpropagation applied after the presentation of each the 1200 images. After only 40 cycles through all 1200 exemplars, training reaches a point where only asymptotic improvements seem likely. The performance of the net appears to be good, with the correct direction, accurate to within two output units, being chosen 90% of the time.

Although it is too early to say at this stage whether neural networks will in the long term be successful in producing 'usable' road following systems, the flexibility of neural networks to adapt themselves to seemingly difficult tasks has been clearly demonstrated. Earlier road followers developed at CMU, based on traditional computer vision approaches, took many months to 'fine tune' and test before they were ready. The most time consuming and difficult section of this work, as Pomerleau (1989) states, was the development of the road image simulator. After that, it took only half an hour or so of backpropagation to produce a road following system as good as any of the earlier ones developed at CMU. This ability of the error backpropagation algorithm to quickly and effectively assimilate large amounts of information is one the hallmarks of back-propagation and what makes it use so attractive for many differing applications.

3.4.3.3 Learning to Perceive Left and Right

The nature of most problems has been towards actually recognising some pattern, or some features. Scalettar and Zee (1988) discuss how neural networks could perform simple perceptual tasks. They present an investigation aimed at teaching a net how to tell whether a given object, is left or right of another object.

They adopt the standard input layer, hidden layer, output layer, fully interconnected feed forward architecture common to backpropagation and devise a scheme allowing them to code two objects which they call a house and a tree. The net has a row of inputs taking on values of -1 or +1. A house is defined to be a string of four +1s surrounded by -1s and a

tree is a single +1, again in between -1s. The output of the net is meant to be 1 if the house is to the left of the tree and 0 otherwise.

The total number of possible exemplars is high, and only a limited number are ever presented for training with the expectation that the net will generalise for those patterns not presented. However, Scalettar and Zee report that such generalisation did not take place and instead suggest that perhaps the net is not actually learning a perceptual task, but simply learning the patterns by rote. They realise that because of the fully connected architecture, the net is not making use of any geometrical information available. However, geometry can be made more meaningful for the net, if the network connectivity is altered to specifically exploit spatial information.

If it is realised that what is needed are some kind of feature detectors, to say detect a tree or a house, then it is possible to try to discover how these detectors may operate. Once this is established, an attempt can be made to force units in the hidden layer to take on these functions.

In terms of a fully interconnected net, this means restricting connections to induce the desired behaviour. In fact, if each hidden unit is limited to three inputs each, a simple arrangement of weights is all that is needed to make a hidden unit act as a tree detector or to act as a house detector. With the appropriate modifications to the connectivity and geometry (lateral connections are included) of the net, it is found that the type of detectors necessary to achieve the task emerge when trained using backpropagation. Because the net has captured the common quantity required to achieve the task, it is also capable of better generalisation.

The method of problem solving adopted by Scalettar and Zee is in itself quite important. Ideally, it would be preferred if backpropagation learning could learn the task with no interference. However, as they point out, this may not always happen even if a possible set of weights does exist to perform the desired mapping. Instead, if it can be determined

how the net will need to operate in order to learn the task, then the net architecture should be modified 'manually' in order that the required behaviour is more readily induced.

3.4.3.4 The 'T-C' Problem

The idea of limiting connectivity is seen again in the 'T-C' problem (Rumelhart et al., 1986). Here, the task is to learn to recognise a letter T or C, in a visual input, independent of translation or orientation. Using the backpropagation paradigm, it might be thought that by simply presenting a multi-layer network with instances from the input domain, the appropriate function could be implemented. However, when the task is as complex as this, such an attitude is a little naive and instead it is more fruitful to adopt the kind of approach taken by Scalettar and Zee (1988).

The problem is simplified somewhat, by using very simple looking letters. Each T and C is comprised of five squares or pixels.

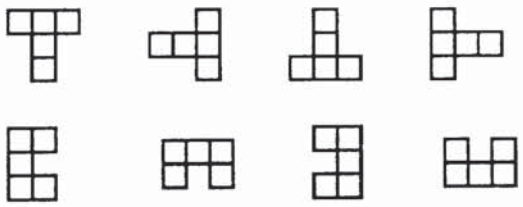


Figure 3.6 T and C shapes at all possible orientations.

Again the network architecture is not fully interconnected, but constrained so as to facilitate learning of the given patterns. The hidden layer is organised as a two dimensional grid in which each hidden unit takes its input from a 3x3 'receptive field' on the input space. From figure 3.6 it can be seen that a 3x3 receptive field is large enough to completely cover each letter. A single unit is used to output a 1 if the input was a T and a zero if it was a C.

It should be realised that to achieve translation invariance, the hidden layer units have to perform the same operation in all places over the input space. This means that the weights for each hidden unit once trained should be identical. As a consequence of this, it is only necessary to train a single hidden neuron to learn to recognise a T or a C independent of its orientation within the receptive field of the neuron. Then the weights learned can be copied to the links for all other units in the hidden layer. Rumelhart et al. (1986) report on results of this work and show that by repeated presentation of the eight possible patterns (see figure 3.6), a suitable set of weights will emerge capable of differentiating between the two characters. Interestingly, different training runs develop different weights and it is found that there are four sets of possible weights that may emerge, each representing four possible templates that a unit could use to discriminate between Ts and Cs.

In both these examples, learning to perceive left and right and the T-C problem, the idea of strategically restricting connections leads to successful and efficient neural net solutions.

3.4.4 Structured Neural Networks

The idea of configuring a net for special tasks can be taken a step further. In the visual domain, we have seen how it might be possible to devise nets to achieve rotational and translation invariance for simple objects. The real world exists in three dimensions, not two and so natural vision systems have to cope with viewpoint independence as well. For normal humans, recognition in three dimensions is achieved subconsciously and without effort. Also, human vision is powerful enough to handle all the consequences that 3D vision brings, such as occlusion. Can it be said then that such vision systems are also based on structured nets specifically configured to handle visual processing in three dimensions?

This debate is taken up in Feldman (1985) and Feldman and Ballard (1983). Of interest is their approach to representing complex concepts. Rather than rely on the ability of nets to learn their own distributed representations, Feldman (1985) is very much in favour of coarse coding as a way of storing and using higher level symbolic data and shows how networks could be structured to store information relevant to whole objects. Essentially, individual features or entities are assigned to single neurons, that is each value is designated a single unit. As an example, Feldman (1985) shows how a representation of a golf ball and a ping-pong can be set up using such a 'unit / value' idea.

Feldman et al. (1988) use these ideas to propose a structured net that is capable of recognising complex objects in a orientation and translation invariant way. In the 'T-C' problem, the answer was to provide T and C detectors for all possible locations on the input space; such an approach is not possible for the recognition of complex objects. The 'T-C' approach would require the same computational mechanism at every available location in the input and in this case, the computational mechanism can be expected to be very complex. A possible way of solving this problem, as Feldman says, is to structure the net and where necessary, to process it sequentially.

The solution (Feldman et al., 1988) consists of several *populations* of units, where each population is assigned a separate task. For example, a 4x4 grid is used to code the location of an object, a set of units indicate the colour and another set indicate the shape. Translation invariance is due to the presence of a set of units that indicate spatial relations between objects, eg. above, left of, right of etc. The information in the net is accessed sequentially. As each unit in the location grid becomes active, it means that a particular location is under attention and all the connections now relate to properties of the object or sub-object at that location. A layer of hidden units, connected to all the above mentioned populations, is used to enable the net to learn to classify objects.

The project admittedly is very ambitious and is fraught with difficulties. However it serves to highlight that the use of structured neural networks is a useful approach.

Feldman et al. (1988) argues that to take advantage of the possibilities offered by massively parallel neural networks, current neural net techniques must be merged with conventional computing.

3.5 Summary

Recently, the amount of research involving non-computational approaches to computer vision has been increasing. It appears as if biology and visual psychology might inspire promising alternatives to the more traditional methods used in the past. Furthermore, the development of powerful learning algorithms has in particular re-kindled interest in neural networks which are now playing a major role in providing non-computational solutions to vision problems.

Chapter 4

Connectionist Approach to Processing Perceptual Groups

4.1 Introduction

Many of the traditional approaches to computer vision have typically relied upon model based recognition. That is, the process of recognition is based upon the ability to match object features obtained from the image, with those held in memory. When recognition is to take place in three dimensions, systems based on these approaches can run into severe problems. ACRONYM (see section 3.2.1) for example, is a complex system employing sophisticated prediction and constraint manipulation techniques. Yet it has failed to demonstrate its capability to perform recognition in three dimensions, even with a restricted domain of objects (aircrafts). Where systems have succeeded is when recognition is limited to just one or two known objects such as in 3DPO or SCERPO.

Human vision seems to solve the vision problem almost effortlessly, yet advanced techniques in computer vision have been unable to even approach this level of performance. It is not surprising then, that neural networks which are claimed to possess properties similar to those of biological networks are creating so much interest. Many feel that neural networks may eventually be more successful in tackling recognition and perception problems than previous approaches.

Recently, a lot of work involving neural networks has tried to tackle those problems which have been so difficult in the past. For example, neural networks have been used for applications such as handwritten character recognition (the Neocognitron, section 2.8.2), for face recognition (section 2.9.2) and even for autonomously guided vehicles (section 2.9.3.2). These are all high level vision problems, which in biological systems would probably involve several levels of processing. However, neural network systems such as these seem to have ignored the need for any form of preliminary low level processing. A few researchers have addressed this issue, Linsker (see section 3.3.3) for

example, concentrates solely on a network that can learn to develop simple feature detecting cells. Another network is Walters computer model reviewed in section 3.3.7. This time, the aim was to use the results of psychophysical experiments to find ways of locating perceptually *bright* lines. Again, the task undertaken is a low level task; no high level recognition takes place.

The relative infrequency of neural network approaches to low or intermediate level vision would appear to suggest that much of the effort is concentrated on solving difficult problems, perhaps too quickly. The opinion held by this thesis is that much could be learned from the application of neural networks to low or intermediate levels of processing as well. Not only would this be a way of studying the applicability of neural net techniques to the chosen problem but would also provide a vehicle for exploring the practicalities of neural networks themselves.

4.2 Outline of Proposed Project

The aim of this project is to process primitive perceptual groups using connectionist techniques. The processing will seek to evaluate the determination of relative significance values for these perceptual groups and this is deemed a worthwhile task since the findings of Lowe and Walters seem to indicate that such significance values can be useful, especially in the lower levels of visual processing.

The differences between Lowe and Walters approaches are quite prominent. The way each has been implemented is particularly worthy of note. SCERPO is implemented in the 'traditional' way, that is, as a set of sequentially run programs. Walters's computer model on the other hand, is built using a connectionist approach. Walters admits that the same local operations could be implemented as a conventional algorithm, but feels that it is conceptually more interesting to incorporate such local operations into a connectionist architecture. This is the view pursued in this project and the intention is to design a connectionist network that when given an input describing a simple perceptual group,

will generate an output (or outputs) which correspond to the perceptual significance of that group. Again, the same could be accomplished through conventional programming techniques. However it is felt that the use of neural network techniques is far better suited to problems of this nature.

Rather than build and attempt to use one large network, the approach taken in this study will be to try to partition the task into smaller, manageable sub-tasks, so that each sub-task can be tackled individually, and a 'sub-network' developed to carry out that task. Once all sub-nets have been built and trained, they can be wired together into a final 'network of networks' which carries out the set task.

Before a significance value for a whole pattern can be evaluated, separate significances must be determined for the various features that comprise that pattern. SCERPO treats each pair of lines as a feature and evaluates their significances as shown in section 3.3.6.2. Thus the task of finding a significance value is broken down to actually finding significances for pairs of lines. Note that Walter's computer model is also built in a similar way. Various types of connectionist unit are employed to perform specific operations. These units are then 'wired' together into small sub-networks which have been designed to perform various tasks. Essentially, the model comprises two basic network types; the length enhancement network and the end-connection enhancement network. Many instances of each sub-network exist in order that the same computation can be performed throughout the entire image.

The basic building block for Walters model is the connectionist unit (such as the 'U' unit mentioned in section 3.3.7.3). These units perform simple arithmetic operations on their inputs. However, the functions necessary to evaluate perceptual significance values are a little more involved. For example, notice that the geometric equations in section 3.3.6.2 appear quite complex and could not be handled by simple arithmetic units. If the proposed network needs to evaluate functions of this kind, then the basic building block will have to be a little more complicated than a simple U-unit, in order to cope. A suitable

candidate for a basic building block would be the multi-layer perceptron (see section 2.2). Networks of this kind can be trained to solve many types of problem using the back-propagation algorithm. In particular they can be made to learn continuous functions (Hoskins, 1989). Although this algorithm is renowned for being computationally intensive and therefore slow, its ability to handle a large variety of problems makes it an attractive method to use.

4.3 Coding the Input Pattern

The input to both SCERPO and the Walters computer model is some form of image. In the case of SCERPO, the initial input is a grey scale image, resulting in the need to perform a significant amount of low level image processing (eg. edge detection, line detection) before the relevant features become accessible. With the Walters model a line image is used for the input. The model requires that a large proportion of the network is dedicated to providing feedback to each and every pixel. This level of processing is not relevant to this project and would be best avoided. This can only be done if a suitable method of image coding can be found, that can adequately describe the required features at an intermediate level.

The coding of inputs (or outputs for that matter) can in itself become quite an important issue. In section 2.9.3.1, Burr's approach to handwritten character recognition uses a special form of encoding based on the seven segment display. If the input had not been encoded in this fashion, then the unprocessed 19 x 19 image would have to be used. This would make learning the task far more difficult since now, relevant features (character strokes) would no longer be explicitly represented.

4.3.1 Description of Coding Scheme

This section describes a simple and effective coding scheme which is used in conjunction with this work. The coding must be able to describe simple line patterns such that each line is coded as a single entity, i.e. a single entity should encode line length, orientation and location.

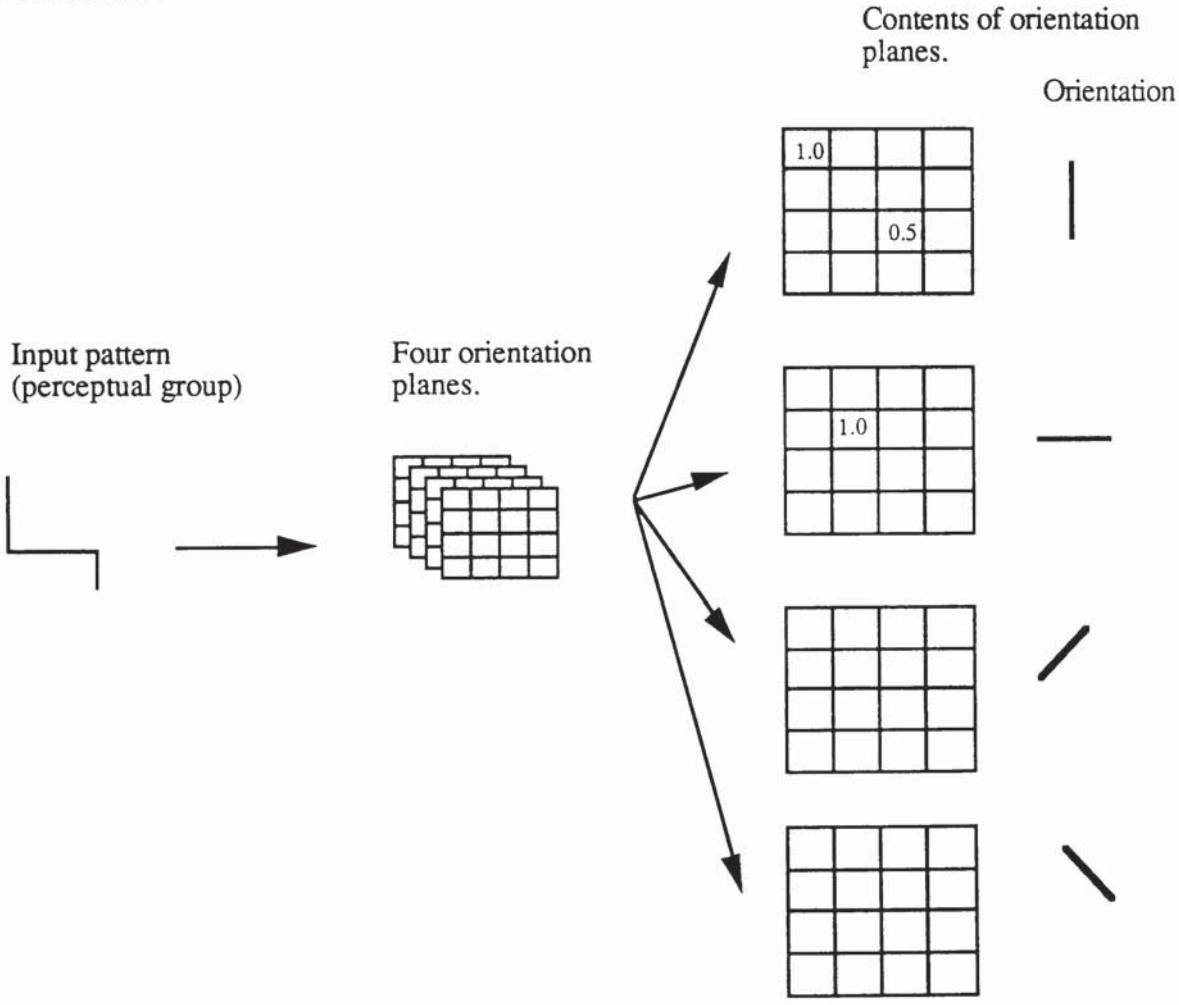


Figure 4.1 A simple line coding scheme.

Figure 4.1 above shows a schematic diagram of a possible coding scheme. The main features of the scheme are listed below:

1. There are 4 planes (4 x 4 in size), one for each of the four orientations catered for.
2. A line will be encoded as a number representing its relative length on the appropriate orientation plane.

3. The location at which the line is coded, represents the location through which the major portion of the line is passing

Only one location can be used to represent a line and the criterion for choosing this location is based on finding the grid location through which the major portion of the line passes. An implication of this is that if the line is equal to or greater than the length of two grid locations, then the line would be split into two, which is undesirable. To prevent this a further restriction can be imposed limiting the line length to two locations or less. In later practical work, this maximum length is normalised to a value of 1.0.

Although this coding scheme is fairly simple, it does offer an attractive method for describing line patterns. Its main asset is that individual lines are represented by a single entity, rather than a series of pixels as they would be in an uncoded image. However, it has several drawbacks, of which the main one is the loss in resolution. Consider say a 256 x 256 line image, which has to be mapped onto the 4x4 scheme described above. If the image contains two lines which are far apart, both lines will be coded. However, if the two lines are parallel and very close together, that is, less than a quarter image width apart, then it is likely that both lines will fall in the same location on the plane and so will be coded as a single line. Fukushima's Neocognitron also suffers from a similar problem. When two digits are close together, they fall within the input area of a single receptive field and are thus not detected (see section 2.8.2).

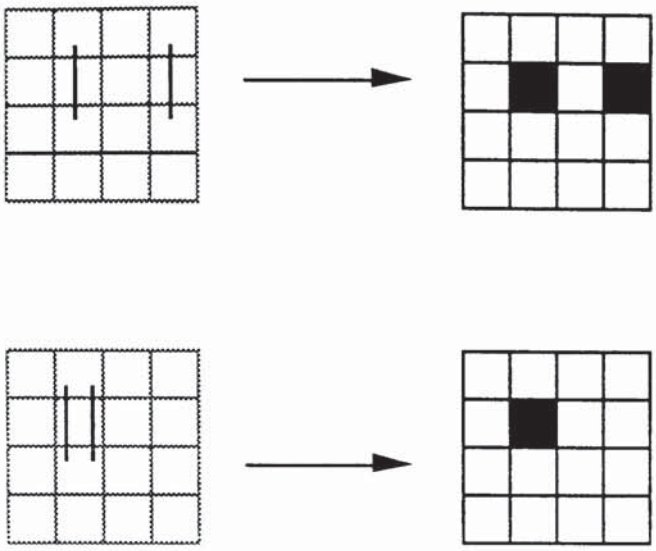


Figure 4.2 Problems with low resolution coding

As mentioned before, the task of calculating a perceptual significance for a particular pattern can be broken down into sub-tasks where each sub-task will be a procedure for evaluating the perceptual significance of just two lines. However, these pairs of lines can appear in any of several configurations, eg. end-end connected, parallel, collinear etc. This means that firstly the configuration present must be found and then the appropriate function used to establish the significance of that pair. In particular, the following cases are considered : the detection of parallel lines, the detection of collinear lines, and the detection of connectivity.

4.3.2 Feature Detection

4.3.2.1 The Detection of Parallel Lines.

Each orientation plane contains only lines of the same orientation. The consequence of this is that the detection of parallel lines will be based on counting the number of lines present in each orientation plane. So an instance of parallelism can only occur if at least two lines are present in a single plane.

The geometric equations used by Lowe to detect parallel lines (see section 3.3.6.2.2) allow for lines of differing orientations to be considered. Because of the coarse coding

scheme used here, the definition of parallel lines is somewhat less flexible and considers only lines of equal orientation. Also the definition used does not discriminate between parallel lines that do or do not overlap along their length.

4.3.2.2 The Detection of Collinear Lines.

Similarly, collinear lines can only exist if an orientation plane is found with at least two or more lines. However this time, a further constraint exists. The condition for two lines to be collinear is that their orientations must be equal and their endpoints must be close together.

The main aim of the coding scheme is to represent each line as a single entity. An ambiguity emerges when we try to cater for collinear lines. In the SCERPO scheme, collinear lines need not be exactly collinear, that is, their orientations might vary slightly. In this coding scheme, collinear lines (and parallel lines) are expected to have exactly the same orientations. This means that two collinear lines effectively become a single broken line and should, in accordance with the afore-mentioned aim, be stored as a single item, not two items. Therefore it may be more sensible to assume that collinear lines have already been dealt with at the pixel level, thus eliminating the need to carry out further processing. Hence the network to be designed will only need to detect parallel lines and corners, not collinear lines.

4.3.2.3 The Detection of Connectivity

Corners are not explicitly coded, instead the location and orientation of lines can be used to determine if a corner exists at a particular point.

Connectivity can be formed in many ways. The simplest is when two lines meet at or near their end points to make up an end-end connection. However connections may be end-end, end-middle or middle-middle.

By way of example figure 4.3 helps to illustrate how an end-end connected corner would be detected. (The shading in each location indicates the orientation of the line coded.)

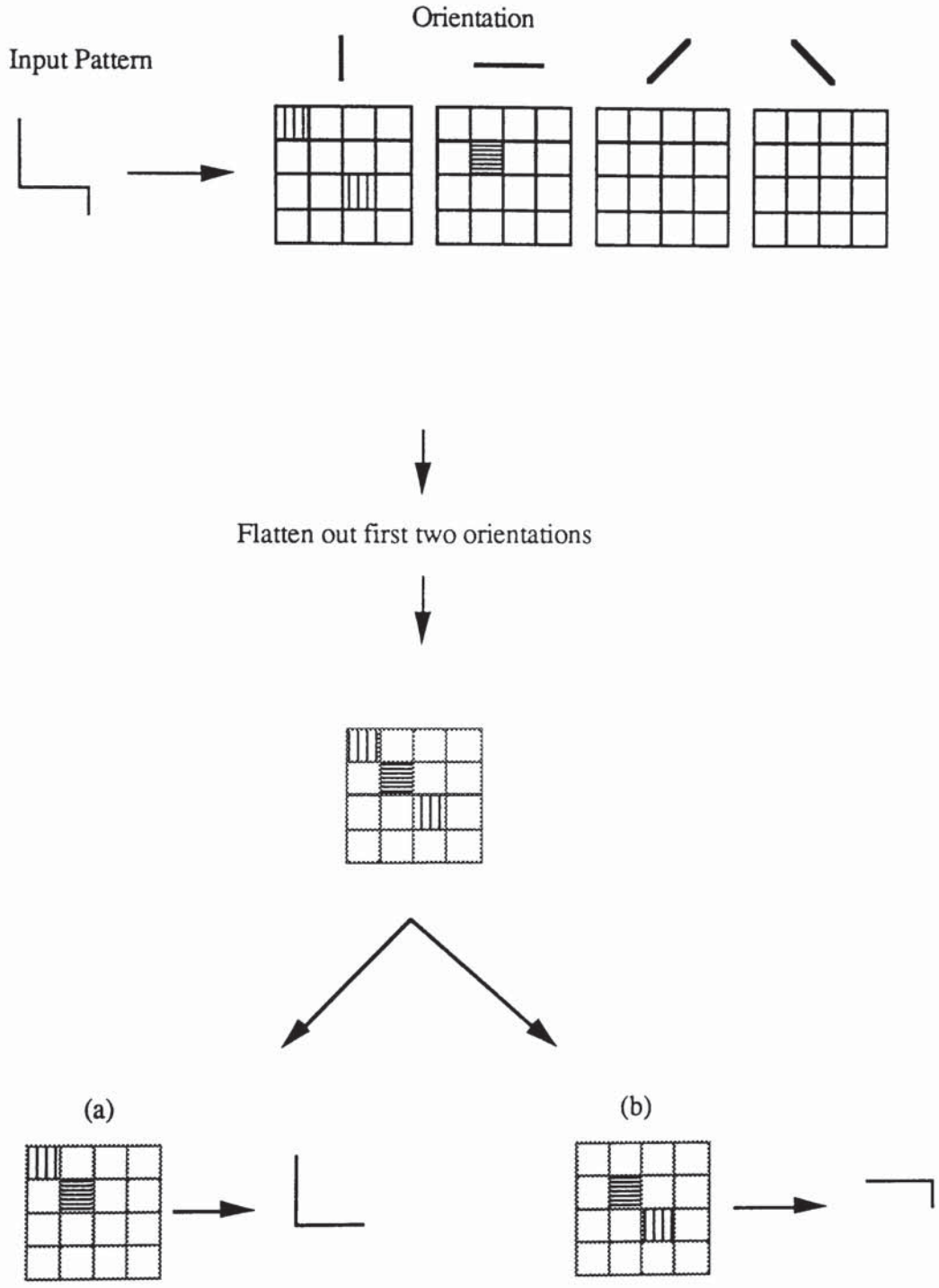


Figure 4.3 How corners can be detected.

Figure 4.3 shows how an input pattern (representing a simple perceptual group) would be described. Since only vertical and horizontal lines are present, the last two orientation planes are empty. 'Flattening' the planes together results in a plane with three lines. This contains two corners, shown as (a) and (b). Thus to detect an 'L' shaped corner, the code found in (a) must be recognized, and to detect an upside down 'L', the code shown in (b) must be detected. This is summarised below:

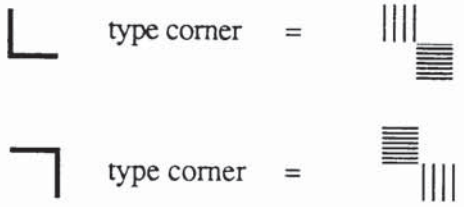


Figure 4.4 Representation of corners

By considering all the combinations of end-end connections that can be generated by pairs of lines at differing orientations, it is possible to find suitable representations for all corners, positioned at any allowable orientation. (See Appendix F for a full list of allowable corners)

The section thus far has dealt with end-end connections only. Matters are complicated when end-middle and middle-middle connections are also considered, since problems such as ambiguity arise, for example a cross could be described by two lines or four lines. To cater for end-middle and middle-middle junctions, the proposed coding scheme would have to be significantly elaborated. This could unduly complicate matters and for this reason no special provisions will be made to handle such junctions. All connections will be expected to be of the end-end type.

4.3.3 Some Examples of Patterns Represented Using this Scheme.

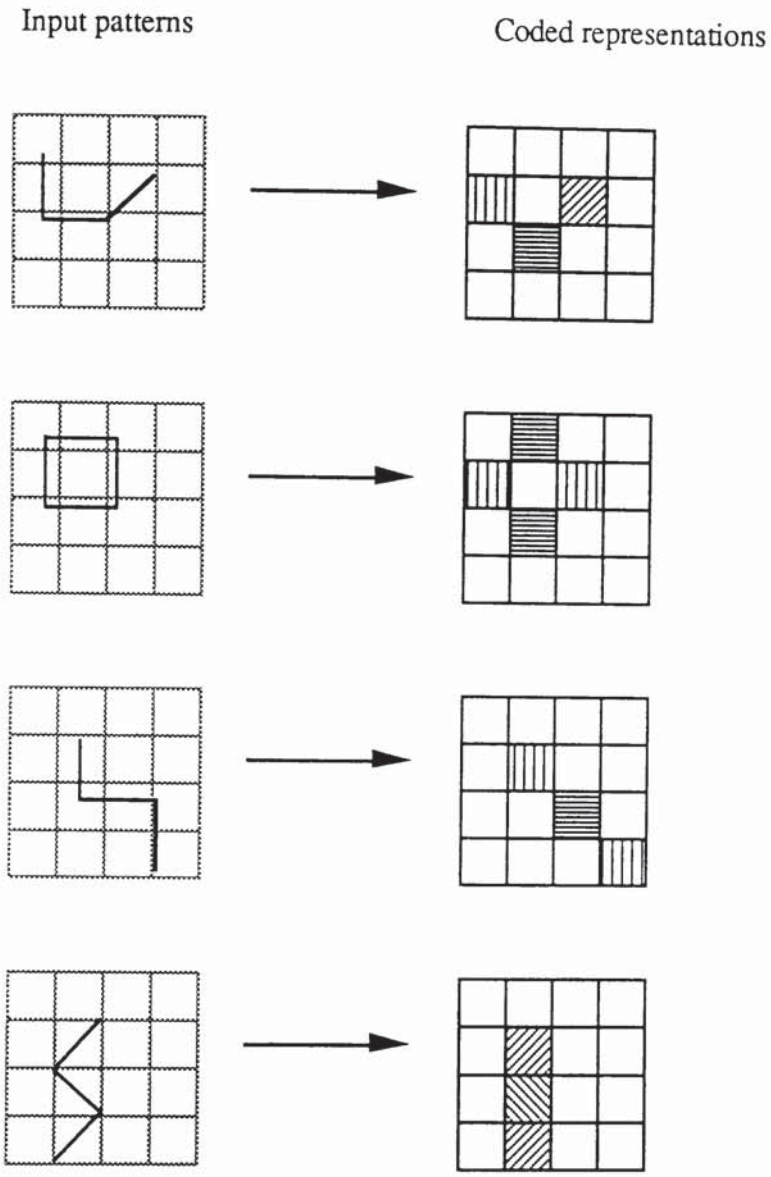


Figure 4.5 Some examples of coded patterns.

Each of the coded representations consists of four orientation planes. Here, all four frames have been superimposed onto one. The direction of the shading indicates to which of the orientation planes each code belongs.

4.4 What Does the Network Need to Output and How?

Lowe's (1987) rigorous treatment of the geometry of line pairs results in the set of equations shown in section 3.3.6.2. The equations are functions of many parameters, such as the distance between endpoints, distance between midpoints, angle between lines and length of lines. Their thoroughness enables imperfect 'features' to be detected. For example, by accounting for the angle between two given lines, SCERPO is able to deal with lines that are not quite parallel. Similarly, collinear lines need not be perfectly collinear. The proposed coding scheme does not have the capacity to handle so many different values. The low resolution of the representation does not permit accurate graduation of such parameters. Figure 4.6 below illustrates one aspect of this problem.

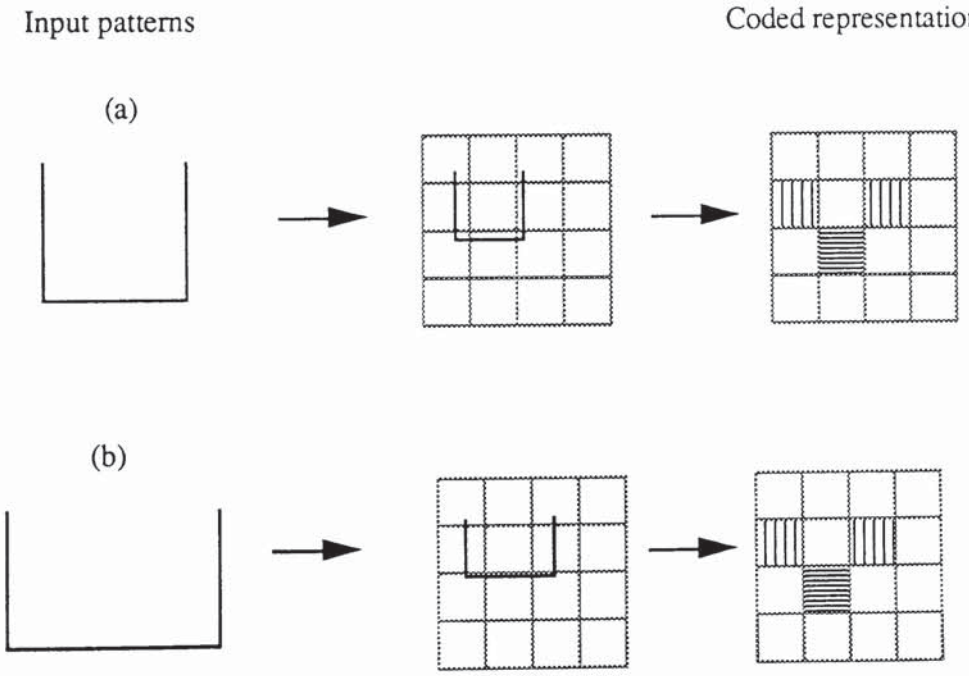


Figure 4.6 Problems with encoding distance

Patterns (a) and (b) are similar except that (b) is wider than (a). i.e. the distance between the vertical lines is different. Since each line is coded by the location on the orientation plane through which the majority of the line passes, then it turns out that the vertical lines are coded in the same place in each case. The distance between these two grid

locations then serves only to give an approximation to the actual distance between these lines.

Line lengths are coded accurately. The presence of a line is indicated by a real number between 0 and 1 at the appropriate location on the plane and this number represents the length of the line.

4.4.1 Perceptual Significance of Parallel Lines.

Lowe's (1987) equation for perceptual significance of parallel lines is the reciprocal of

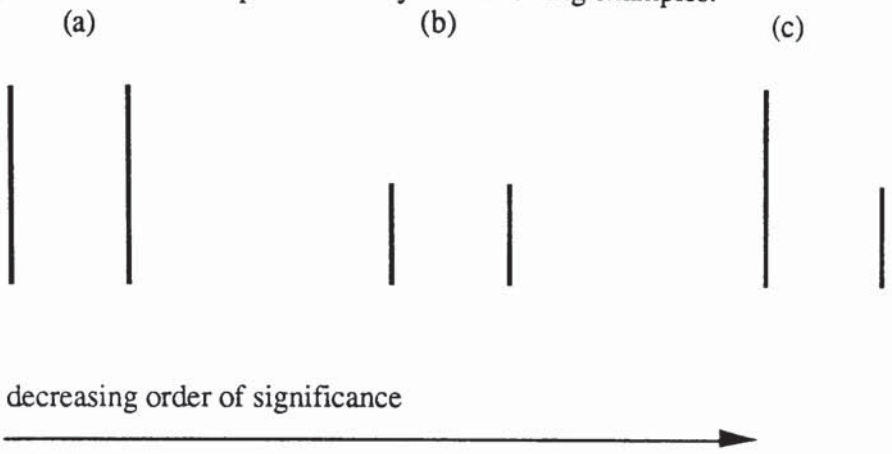
$$E = \frac{4D\theta s l_2}{\pi l_1^2}$$

i.e. significance is proportional to $\frac{l_1^2}{s\, l_2} \frac{\pi}{\theta}$

Both lines are assumed to be always parallel, and so pi/theta is a constant. This leaves :

$$\text{significance of parallel lines} \propto \frac{l_1^2}{s\, l_2} \qquad (1)$$

The effect of this relationship is shown by the following examples:



(a), (b) and (c) are examples of parallel lines. The distance, s, between the lines is equal in all cases. Using the relationship just arrived at, it is found that (a) has a significance higher than (b), which has a significance higher than that of (c).

One of the parameters involved is 's', the distance between the parallel lines. This parameter as used in the coding scheme lacks precision, not only because of the drop in pictorial resolution but also because it does not actually measure the distance as specified in SCERPO. Instead, 's' is taken to be the cartesian distance between the points on the orientation plane encoding the two lines.

4.4.2 Perceptual Significance of End-end Connectivity.

Consider the following corners:

(a) (b) (c)

decreasing order of significance

Corner (a) has a higher significance than (b) which in turn is more significant than (c). If l_1 is the length of the shorter line and l_2 the length of the longer line, then the following, a simplified version of equation (1), could reflect the above behaviour :

$$\text{significance of corners} \propto \frac{l_1^2}{l_2} \qquad (2)$$

4.4.3 Sequential Processing and SCERPO

For any given line pattern, SCERPO will try to locate the most important line pairs. It achieves this by realizing that the most significant line pair is the one where the lines are the closest to each other. Thus given three randomly positioned spaced parallel lines, SCERPO is able to work out which pair forms the most useful primitive by finding the pair having the least distance between them.

Although not immediately obvious, this is a sequential task. Firstly one pair of lines is selected and measured, then another and finally a third combination is tested. Described in terms of a biological visual system, this would be analogous to a focus of attention mechanism. Whether the human visual system processes images in sequence or in

parallel is not clearly understood and is the subject of some debate. However, some general theories do exist. It is believed that higher levels of image understanding and recognition must perform some degree of serial processing. For example, Ullman (1984) refers to such processing in terms of 'visual routines'. The existence of fovealization, suggests that the scene is being broken down and processed in parts and not all at once. Such focus of attention mechanisms which cause the eye to rapidly scan the image for points of image, may be evidence for serial processing. Early processing on the other hand, such as edge detection and line detection is understood to be entirely parallel. Somewhere in between these low and high levels, parallel processing may slowly give way to serial processing.

The detection of perceptual primitives by a network using only feed forward connections can only take place in parallel. This has implications on the determination of distance between parallel lines. In the example above, three parallel lines were detected. To select the most significant pair will however, require a serial search of all possible combinations of pairs that can be generated (three in this case). Given that the number of lines coded is unknown, then the number of comparisons necessary will also be unknown. Thus the task of teaching a multi-layer perceptron this problem would be very difficult. The simplest case would be if it was known that at most two lines per orientation plane were present. With these assumptions there would be no need for performing serial searches. Although this limit of two lines per plane seems restrictive, with four orientation planes it allows a maximum of eight lines to be used which is sufficient to code a rich variety of patterns and shapes.

4.4.4 Example of How a Network can Simulate a Real Function

It has been shown that backpropagation can be used to learn continuous valued functions. Several examples of how this can be achieved are given in Hoskins (1989). This section

describes how a multi-layer perceptron can be used to learn functions useful for calculating perceptual significance.

Equation (2) above is mathematically quite simple and can actually be simulated using a multi-layer perceptron. To illustrate the training process, we can outline the steps necessary to teach a multi-layer perceptron a problem such as this.

4.4.4.1 The Training Set

The function intended for simulation is :

$$\text{significance of corners} \propto \frac{l_1^2}{l_2}$$

The training set will need to consist of a list of vectors which comply with the above equality. Each vector will have two input values (representing l_1 and l_2) and a third output value (representing the calculated significance). The range of values should be over the intended range of line lengths that are to be used (i.e. between 0 and 1). The table below shows a training set that would be used to learn the function

$$\text{significance of corners} = \frac{l_1^2}{2l_2}$$

significance (required output)	l_2 (inputs)	l_1
0.125	0.25	0.25
0.0625	0.5	0.25
0.04167	0.75	0.25
0.03125	1	0.25
0.25	0.5	0.5
0.16667	0.75	0.5
0.125	1	0.5
0.375	0.75	0.75
0.28125	1	0.75
0.5	1	1

If the network is performing some kind of interpolation it would be wise to arrange the data points in a regular distribution. Only a few points have been used in the above training set.

4.4.4.2 The Network

The function to be simulated has two parameters and generates just one output. The network will therefore need two input nodes and one output node and is shown in figure 4.7. The number of hidden units is initially unknown but can be found empirically.

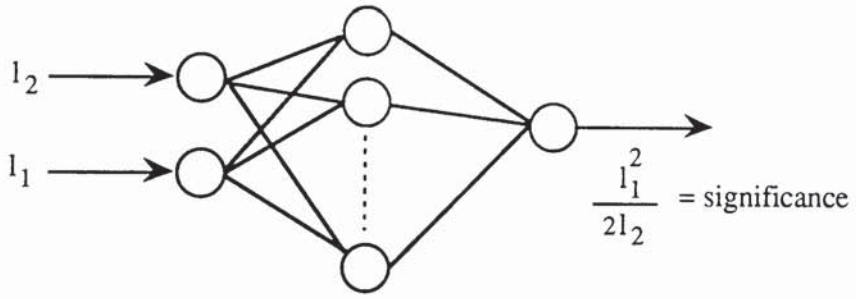


Figure 4.7 A multi-layer perceptron for determining significances

The optimum number of hidden units is regarded to be the smallest number that can learn the given training set. It is generally understood that the smaller the number of hidden units, the better the generalisation is likely to be. However, with too few, convergence to a global minima becomes difficult if not impossible. Consequently knowing the optimum number of hidden units can help to achieve efficient learning.

Preliminary work carried out to investigate how the optimum number of hidden units could be determined involved looking at the variation of number of epochs against

number of hidden units. Several identical learning trials were completed (see Appendix A for basic procedure) to obtain the graph in figure 4.8. In each case, the number of epochs to reach a global error of 0.02 with the given number of hidden neurons was recorded.

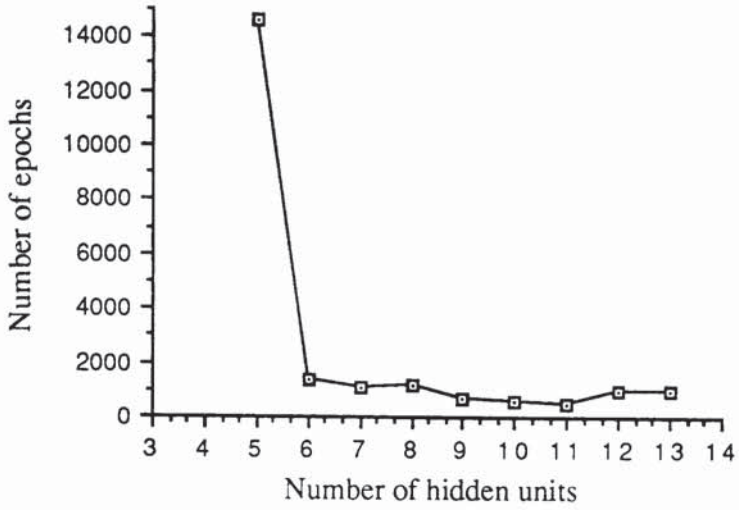


Figure 4.8 Variation of epochs against hidden units.

The above graph is typical of the behaviour expected. There exists a minimum number of units (in this case six) for which the solution can be reached relatively quickly. Increasing the hidden units improves learning speed slightly, but then with too many, the the cost of updating each extra neuron becomes significant and the speed advantage becomes less prominent.

Although this procedure will determine the optimum number of hidden units, it is rather lengthy and time consuming and as such is not always appropriate. If efficiency of the networks is not of paramount importance, then usually it is sufficient to select a reasonable number of hidden neurons to start with and then vary the amount as required during the development stages.

4.4.4.3 The Training Procedure.

Once the training set and the network dimensions have been decided upon, the error backpropagation algorithm can be used to present the training set to the network. The

stopping criterion for the iterative algorithm is based on a global measure of error. For each exemplar in the training set, there will be a difference between the actual output and the required output. This error will contribute to the global error term. When the global error reaches some specified minimum, the algorithm will stop. The network will have learnt the function to a certain accuracy, within the range of input value used. If it turns out that this accuracy is not sufficient, it may be improved by either decreasing the stopping criterion and so extending the learning stage, or by increasing the size of the training set.

4.5 Joining Subnets to Perform Complex Tasks

Section 4.4.4 describes the steps involved in teaching a network to solve a particular problem. In this case the problem is fairly simple and straightforward. For a more complex task it may not be possible to train a single network. A more practical approach is to try to split up the task into sub-tasks and then to deal with them separately.

An example of this is as follows: consider a 4x4 orientation plane encoding two parallel lines. To evaluate the perceptual significance of these lines, a multi-layer perceptron trained to evaluate function (1) (see section 4.4.1) is required. However, because there are 16 inputs coming from a 4x4 code plane, the multi-layer perceptron (mlp) will have to be trained to perform this operation not for 2 inputs, but for 16. This is not an easy task because a large number of inputs can generate impractically large training sets. Training with such large sets will be computationally demanding with the possibility that convergence may not be obtained.

One solution may be to divide the task into sub-tasks, whereby the lines are first detected and then their significance evaluated. Two divisions may be sufficient to accomplish this. One network with 16 inputs and 2 outputs could be used to detect the two lines and then a second net trained to evaluate their significance. These two nets can then be 'wired'

together and made to act as one (see figure 4.9 below). In this way it is possible to design large networks designed to solve possibly complex tasks.

The novel idea of splitting up tasks into more manageable chunks of networks was first used by the author as described in Singh and Claridge (1989) and can also be seen in similar forms in later studies as well, for example Green and Noakes (1989) also use modules of separately trained multi-layer perceptrons to construct a larger, more complex structure.

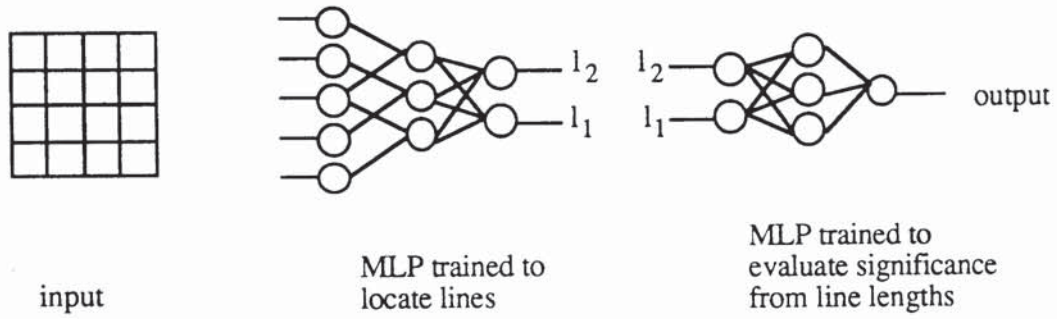


Figure 4.9 Splitting up a task.

Chapter 5

Development and Training Results of the Perceptual Network

The previous chapter proposed a connectionist network for processing perceptual groups. The operations to be computed were also discussed. This chapter describes in detail the development of a large scale network, called a 'Perceptual Network', to perform all of the intended operations.

Design and development of the network has followed a mostly top-down approach where tasks have been identified and subdivided into smaller, simpler subtasks. Thus, beginning with the overall aim of obtaining a single significance value, lower levels were devised to generate the parameters necessary to calculate this single value. The structure of this chapter also roughly follows this top-down approach.

5.1 Detailed Plan of the Perceptual Network

Figure 5.1 shows details of the outer most structure of the Network. The overall significance value is the sum of the significances due to all parallel lines and all corners. Each of these values is determined by a separate structure. Here, the term 'structure' will be used to refer to a network of three layer, perceptron type networks, that is a 'network of networks.'

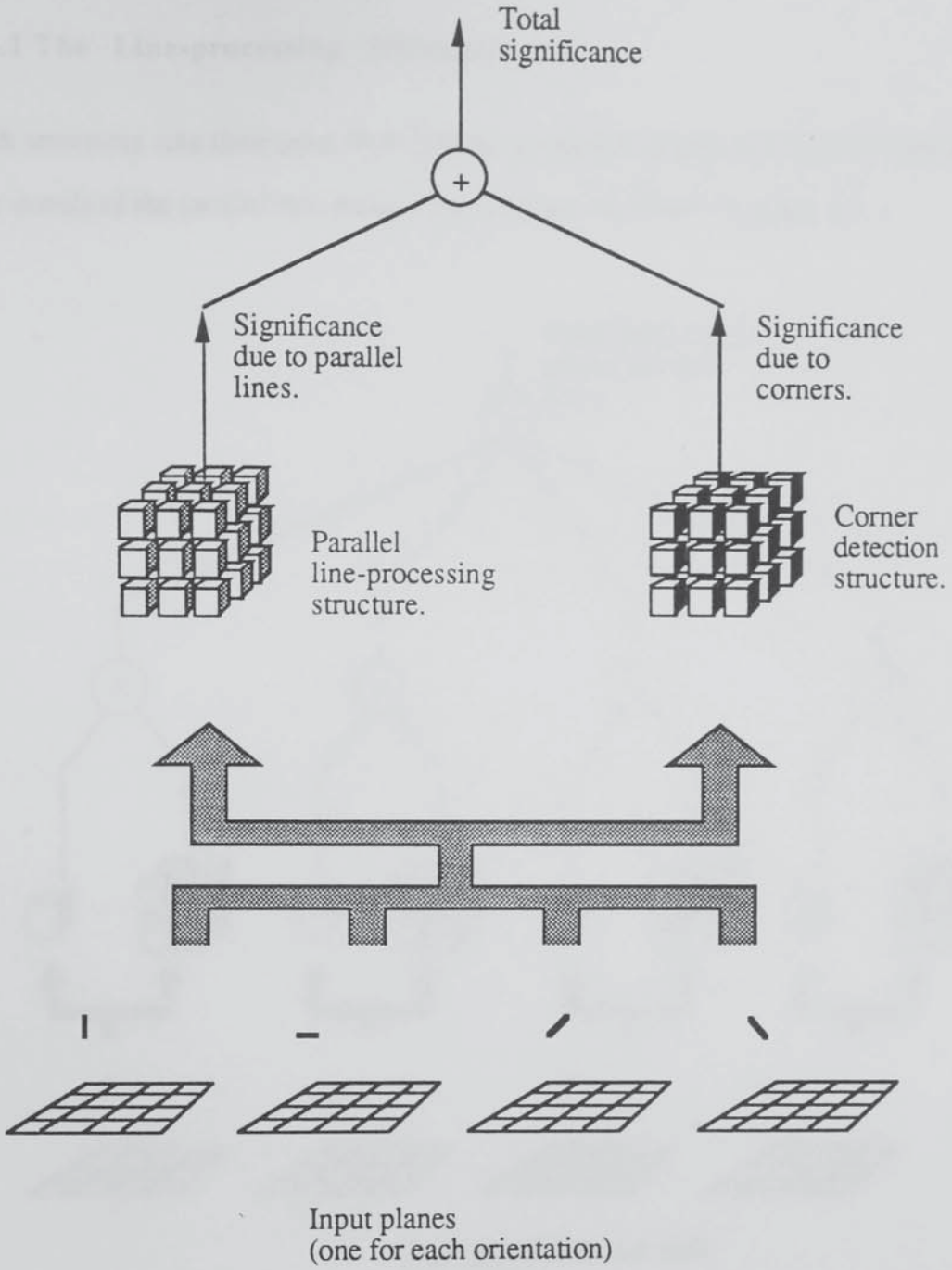


Figure 5.1 Perceptual Network - Outer structure details

5.1.1 The Line-processing Structure

Both structures take their input from all four orientation planes in a feed forward manner.

The details of the parallel line processing structure are shown in figure 5.2.

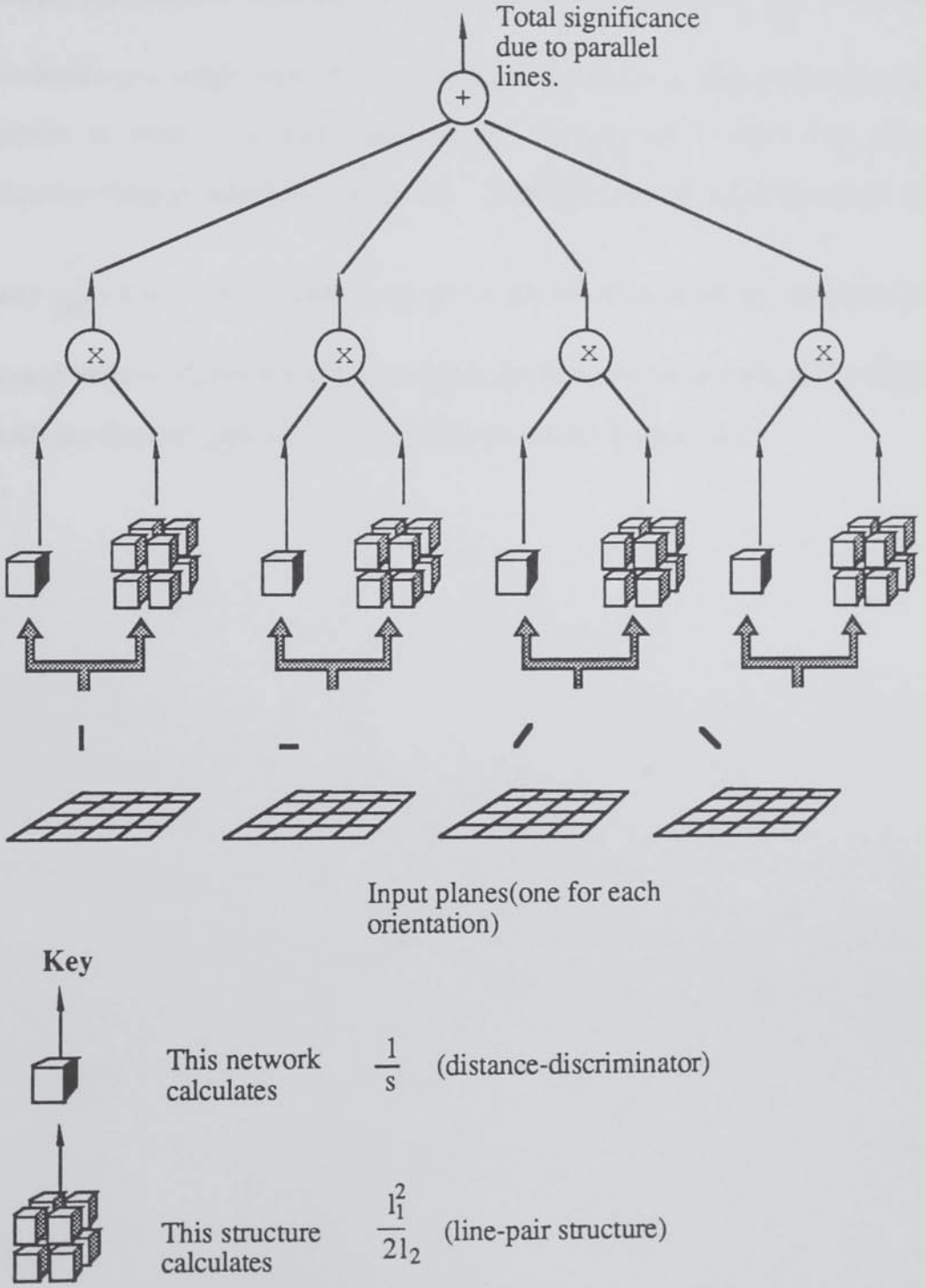


Figure 5.2 Inside the parallel line processing structure

It can be seen that the significance due to parallel lines is calculated as the sum of the four individual significances of each orientation. Each orientation plane is handled by a network (referred to as the distance-discriminator) and a structure (known as the line-pair structure). The network calculates $\frac{1}{s}$ and the structure calculates $\frac{l_1^2}{2l_2}$. The distance discriminator is a single three layer perceptron, but the line-pair structure is again a composite of other multi-layer perceptrons. The product of these two gives the significance value as defined in section 4.4.1. A single structure could have been used to evaluate $\frac{l_1^2}{2sl_2}$, but it was decided that a composite structure would be preferable for the following reasons. Firstly the task would become a lot simpler to train in two stages and secondly the line-pair structure is also used in the corner detector structure.

5.1.1.1 The Line-pair Structure

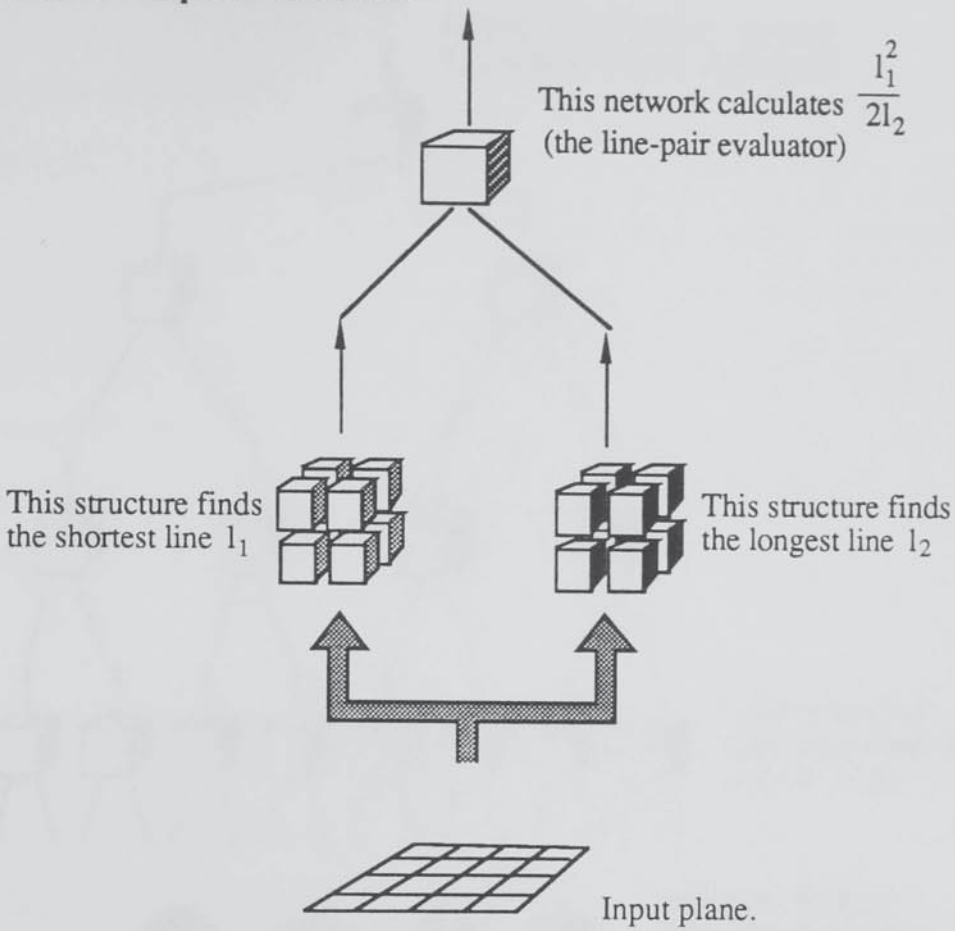


Figure 5.3 Inside the line-pair structure

The input to the line-pair structure is a 4 x 4 orientation plane. On any plane there are at least fourteen zero values and at most two non-zero values representing any lines present.

To facilitate the learning of the $\frac{l_1^2}{2l_2}$ function, it was found necessary to devise a method of 'filtering' up each of the two values present. It was also necessary to know which was the greater and which the smaller of the values. This is the purpose of the two 'filter' structures seen in figure 5.3.

5.1.1.2 The 'Filter' Structures

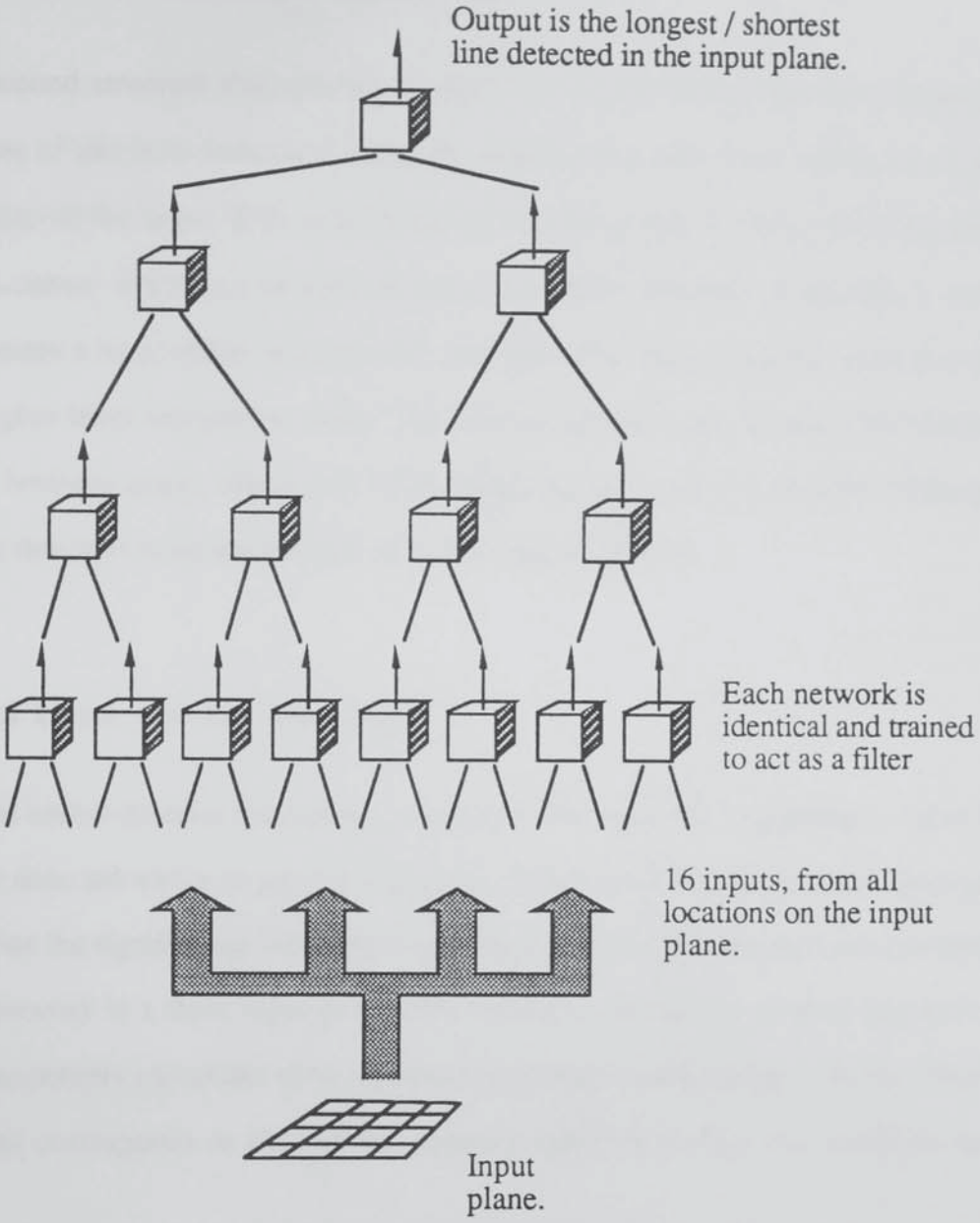


Figure 5.4 Composition of the 'filter' structures

Sixteen inputs from an orientation plane are used, but just a single output is passed on. Figure 5.4 shows how a group of identically trained multi-layer perceptrons can be made to filter up a single value from a group of sixteen. The overall operation of the structure will be determined by what each component is trained to do. The components known as max / min-filters are trained to output the greater or lesser of their two real valued inputs. A pyramid of like trained networks will allow the appropriate value to 'rise' to the top.

5.1.2 The Corner Processing Structure.

The second structure that appears in figure 5.1 is the corner detection structure. The purpose of this is to detect and determine significance values for corners at all possible locations of the input. This is achieved by the use of many corner detecting structures (local corner detectors) situated at each individual location. A secondary task is to implement a basic ability to categorise each type of corner, since this could prove useful for higher level recognition tasks. The coding scheme used, allows a distinction to be made between acute, obtuse and right angled corners, and it is possible to design local corner detectors to recognise each type of corner mentioned.

5.1.2.1 Local Corner Detectors

A local corner detector must detect, categorise and calculate a significance value for any corner detected within its local area of input. This is achieved using a line-pair structure to calculate the significance value and a network to perform the detection and categorisation. The network is a three layer perceptron trained to recognise patterns that give rise to various corners and unlike other networks, has three outputs rather than one. Each of the outputs corresponds to each of the possible types of corners that could be detected. Combining the outputs of the corner-detector network and the line-pair structure, $\frac{l_1^2}{2l_2}$ values for each type of corner can be determined (see figure 5.5).

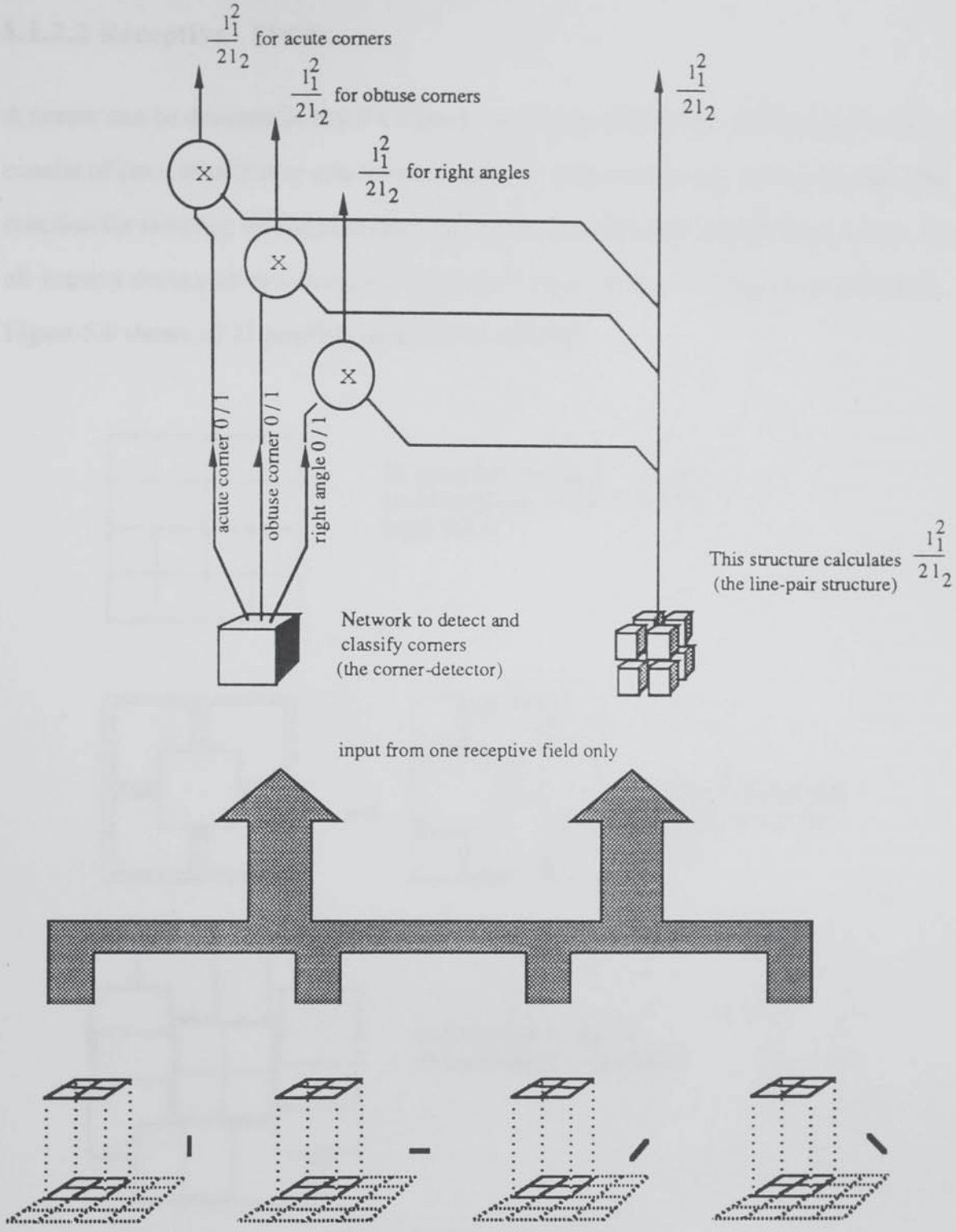


Figure 5.5 Inside a local corner detector

The input to a local corner detector comes from a 2 x 2 array positioned at identical locations on each of the four orientation planes. This input is called here a receptive field and represents those areas of the input where it is possible for corners to be detected.

5.1.2.2 Receptive Fields

A corner can be detected in any 2 x 2 patch, or receptive field (RF) and because corners consist of lines of different orientations, each RF will *run through* all four planes. The criterion for selecting the location of an RF is that it must be able to describe a corner. As all corners consist of two lines, each RF must cover at least two inputs on the plane. Figure 5.6 shows all 21 possible locations of each RF.

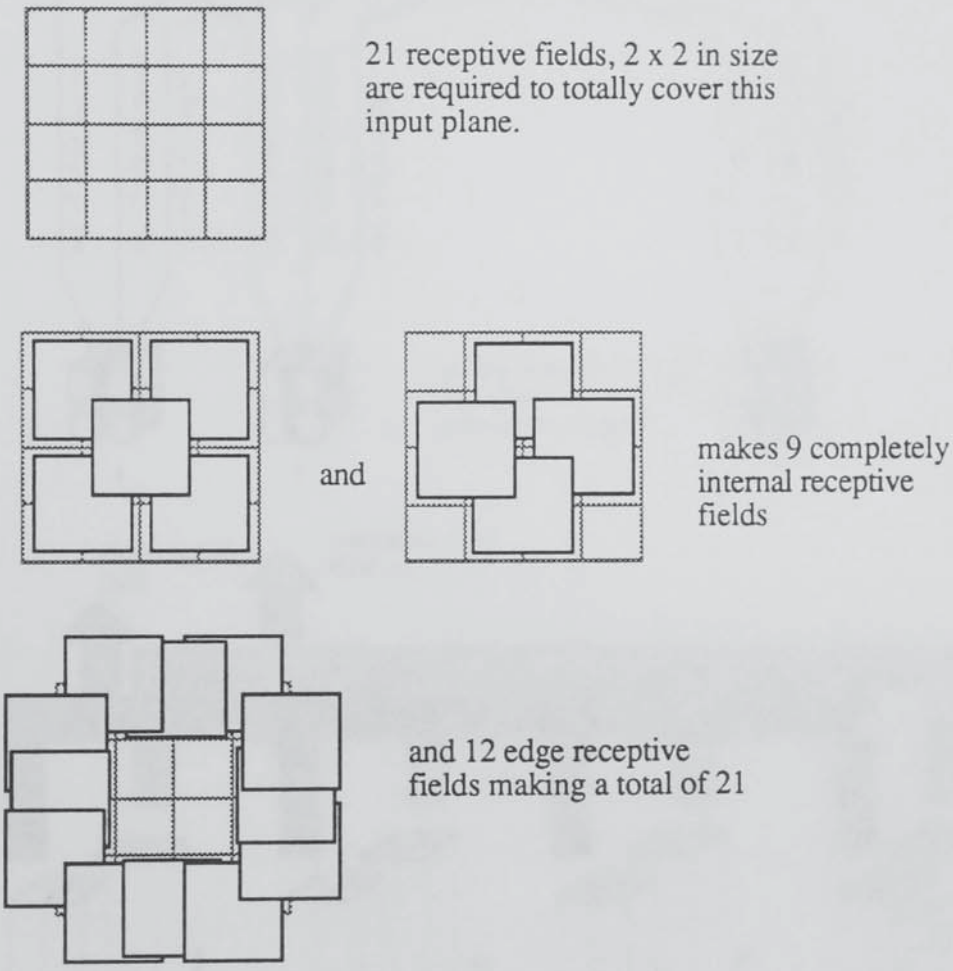


Figure 5.6 Receptive fields.

Thus 21 local corner-detectors are needed to process the output from each of the receptive fields and figure 5.7 shows the resulting structure.

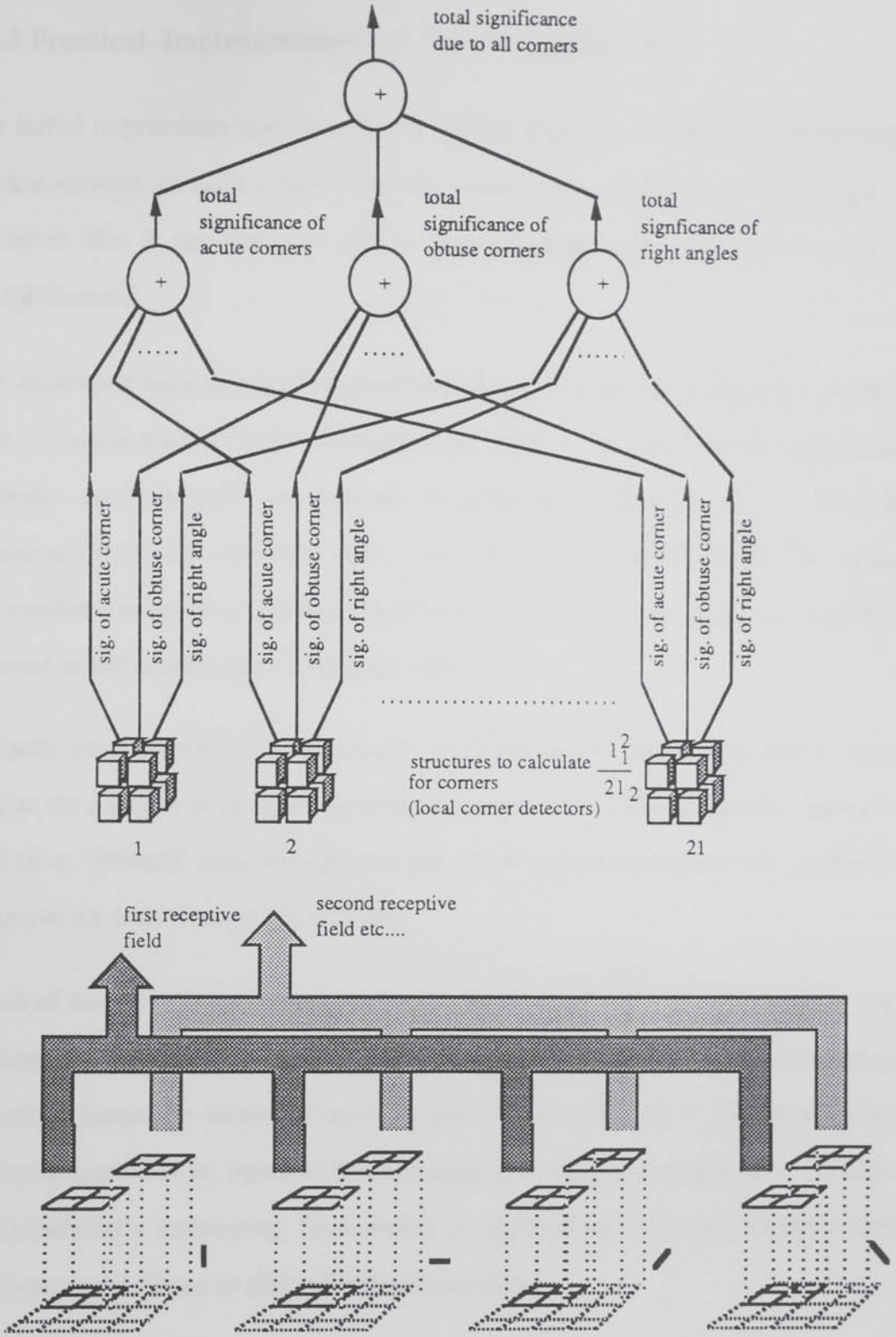


Figure 5.7 An exploded view of the corner processing structure

The significance due to all corners is calculated as the sum of the corner significances from all 21 receptive fields. A local corner-detector is assigned the task of evaluating the significance from a particular RF. Note that because an RF spans across all four orientation planes, sixteen locations are involved.

5.1.3 Practical Implementation of the Perceptual Net.

The initial impressions conveyed by the above plans, may lead one to believing that implementation of such a vast network would prove to be an overwhelming task. However this is not the case and in fact, implementation turns out to be quite straightforward.

The underlying approach to programming and indeed to the whole design of the net, has been to facilitate a high degree of modularity. That is, the intention has always been to consider each network or network of networks (structures) as a 'black-box'. Construction will then involve 'wiring' the various black-boxes together and regarding the resultant network as another black-box. Only at the basic level will connections between actual neurons have to be dealt with.

Actually using or calculating the output of the Perceptual Net will be done in the same way as for a simple three-layer network. In terms of the backpropagation algorithm this will be a 'forward' pass through the net. In programming terms, this amounts to a recursive traversal through the network.

Much of the effort in producing the Perceptual Net will amount to time spent actually training the various subnets. Subsequently, implementation of an error backpropagation simulator forms the basis for much of the work carried out in this thesis. Because backpropagation is an iterative descent algorithm, it can be expected to be slow and computationally demanding. So it would be appropriate to employ the most efficient hardware and software available for implementation.

The programming language chosen is 'C' because it generates fast and efficient code as well as being widely available and the availability of SUN Sparcstations meant that the latest RISC technology could be exploited for its speed and efficiency.

With many of the learning phases likely to take more than a few minutes to complete (a few hours in some cases), the ability to run processes in the background is a necessary

requirement. Again SUN Sparcstations running under the UNIX operating system provide the required facilities as UNIX easily facilitates the entry or removal of background jobs.

5.1.4 Validation of the Backpropagation Simulator

Validation plays an important role in the implementation of any computer simulation and it is crucial that the backpropagation simulator is thoroughly tested before it is used to obtain results. The availability of standards or benchmarks can provide a reliable means of testing a system and comparing its performance. Because the backpropagation algorithm is relatively new, it might be expected that few benchmarks exist for it. However, some problems such as the XOR and Parity, have received considerable attention and so can be used for validation of the simulator.

5.1.4.1 The XOR Problem

Ever since Minsky and Papert (1969) pointed out that single layer perceptrons could not learn to solve simple XOR problems, a lot of effort has been directed towards this particular problem. For example, Rumelhart et al. (1986) state the results of an experiment in which a 2 input, 2 hidden unit layer, 1 output network was trained using standard backpropagation to a global error of 0.01. This was done using a learning rate of 0.25 and a momentum term of about 0.9 and with randomly chosen small initial weights. Rumelhart et al. report that the number of epochs to reach an error of 0.01 was found to be on average about 245.

This experiment can be repeated and used to test our own version of the simulator. Many repeated trials were undertaken so that an average value could be calculated and it was found that for 12 learning trials, each taking less than a thousand epochs, the average was 288.5 and this value is comparable with the result stated.

5.1.4.2 The Parity Problem

The parity problem is often seen as an extension of the XOR, which can be regarded as a two bit parity problem. In this case, no published results were found that were detailed enough to make them suitable for comparisons. Instead, validation was achieved by comparing the simulator in this project with that developed independently by a colleague. The parity problem was set up on both simulators, with exactly the same parameters and exactly the same initial starting weights. Comparing the outputs from both simulators showed that the error value after a given number of epochs was found to be exactly the same (correct to six decimal places). This is a strong indication that both simulators are functioning correctly, especially since both were independently produced and written in different programming languages.

During these tests, two observations were made regarding the precision used in the program and the role of initial weights. In the simulator, all real variables were declared as single precision. However, it was found that by changing to double precision, the number of epochs to reach the same global error decreased. For example, using single precision, a 12 hidden unit net could be trained on the six bit parity problem to an error of 0.32 in 214 epochs whereas, with double precision an identical system would require 190 epochs to reach the same set of weights. In general it was found that using double precision in place of single precision, appeared to reduce the number of epochs required during training. This is something rarely mentioned in the literature.

Regarding initial weights and the parity problem, it was found that learning was very dependent upon a favourable set of initial weights. Some learning trials would converge very rapidly, often within 200 or 300 epochs, whereas others would not seem to converge at all. This was also found in the XOR problem where it was noticed that if a particular learning trial was to converge in a reasonable number of epochs, it would do so in less than a thousand. Thus in the XOR validation test, only those trials that converged in less than a thousand epochs, are considered.

5.2 Details of the Main Perceptual Net Components

The following sections describe the functions and training methods of the five main 'building blocks' of the Perceptual Network. Each component consists of a three layer perceptron trained to process data at various levels of the Perceptual Network. One concern that influenced training was the potential for error propagation. In such a large 'network of networks', it is possible that even small errors in the initial stages could lead to unreasonably large errors in later stages. During training and development, much attention was given to the accuracy of the outputs of each trained network. This was considered to be of great importance and so a general method of estimating the output accuracy of the net was developed. The method and its derivation is described below.

5.2.1 Estimating the Accuracy of a Trained Network

When training a network using backpropagation, the accuracy of the final trained network will depend on two factors; the global error reached and the number of exemplars in the training set. From these parameters it possible to estimate the accuracy of the output, relative to the given training set. The derivation of the relevant formula is as follows :

If x represents the actual output of the net, and T the target or desired output, then the total (global) error over all patterns in the training set is given by :

Total error, $\varepsilon = \frac{1}{2}\sum_{i=1}^{n}(T_i - x_i)^2$ over n patterns.

consider $\Delta x_i = T_i - x_i$ to be the error in x_i, therefore

Total error, $\varepsilon = \frac{1}{2}\sum_{i=1}^{n}\Delta x_i^2$

Now let Δx_{tot} be the total absolute error in the actual values,

i.e. $\Delta x_{tot} = \sum_{i=1}^{n}|\Delta x_i|$ therefore

the average error in actual values is $: \overline{\Delta x} = \frac{\Delta x_{tot}}{n}$

The total *squared* error, call it Δy_{tot}, is $: \Delta y_{tot} = \sum_{i=1}^{n} \Delta x_i^2$

so the average squared error $\overline{\Delta y}$, becomes $: \overline{\Delta y} = \frac{\Delta y_{tot}}{n}$

Expressing ε in terms of Δy_{tot}, $: \varepsilon = \frac{1}{2}\Delta y_{tot}$ (1)

By assuming $\overline{\Delta y} \approx \overline{\Delta x}^2$

and substituting for $\overline{\Delta y}$ in (1), ε becomes $: \varepsilon = \frac{1}{2} n \overline{\Delta x}^2$

giving :

$$\overline{\Delta x} = \sqrt{\frac{2\varepsilon}{n}}$$

Note that $\overline{\Delta x}$ is effectively the root mean square error and is equal to the standard deviation if $\sum_{i=1}^{n} \Delta x = 0$

5.2.2 The "Distance-Discriminator"

In the evaluation of the significance of parallel lines, the term $\frac{1}{s}$ needs to be evaluated. The network that performs this appears in the parallel-line processing structure of figure 5.2 and is called the distance-discriminator network.

The symbol 's' refers to the distance between lines. As the lines are coded on a 4 x 4 orientation plane, then 's' is actually the cartesian distance between points on the grid. For example, two points horizontally adjacent to each other will be just 1 unit apart. Thus s=1, and therefore $\frac{1}{s}$ =1. The furthest distance they can be is when the points are at

diagonally opposite corners, then $s=\sqrt{3^2+3^2}$ giving $\frac{1}{s}$ =0.2357. See figure 5.8 below for more examples.

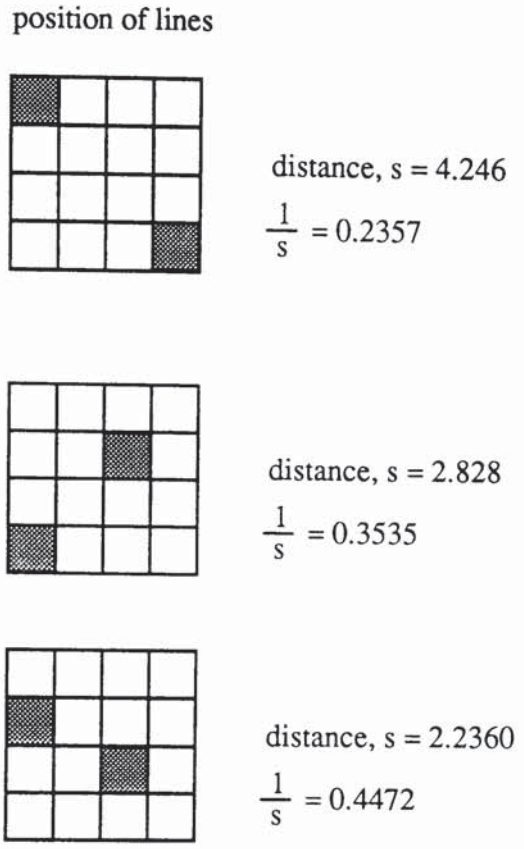


Figure 5.8 Examples of how 1/s values are calculated.

5.2.2.1 The Training Set

The input to the net is a single orientation plane consisting of 16 input values. The output is $\frac{1}{s}$ where s is the distance between two points, if they exist, on the grid. A valid input vector will consist of 14 zero values and 2 non-zero values to code the length and approximate position of two lines. Altogether there are 120 combinations of two points amongst 16 possible locations. A complete training set will consist of all 120 vectors along with their corresponding $\frac{1}{s}$ output values.

The 4 x 4 orientation plane is a two-dimensional input, whereas the input layer of the network is just a linear input. The input plane has therefore to be translated into a linear input. This is achieved by simply numbering the locations from left to right and top to bottom.

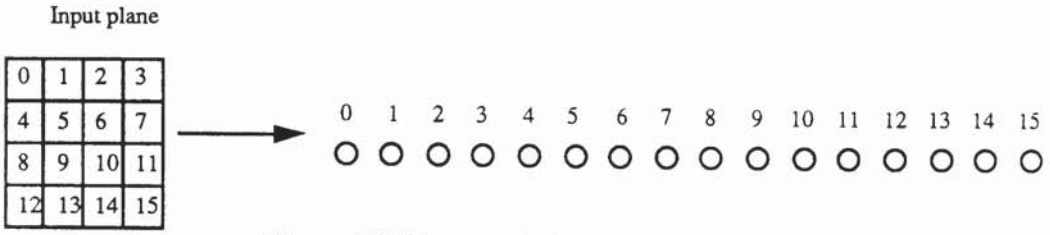


Figure 5.9 Linear ordering of input planes

To simplify the training set, all input values are thresholded so that all non-zero values are treated as one, and zero values as zero. The inputs are then effectively binary valued vectors. The full training set for this problem can be found listed in Appendix B.

5.2.2.2 The Network

A three layer network is used. The input layer contains 16 input units and the output layer, one unit. The number of middle or hidden layer units is initially not known and is found experimentally. Each layer is fully interconnected to the next.

5.2.2.3 Results of Training

Many trials were undertaken using different parameters and it was found that 10 hidden units could adequately learn the task. With a learning rate of η=0.25 and a momentum term of α=0.7, a final global error of ε=0.00005 could be reached in an average of 55000 epochs (calculated over five trials). The global error ε, does not however show how accurate the output of the network will be. This can be determined using the equation derived earlier in section 5.2.1 :

$$\overline{\Delta x} = \sqrt{\frac{2\varepsilon}{n}}$$

Using this relation, it is possible to get an indication of the final accuracy of the output of the network. For example,with 120 exemplars in the training set (n=120) and a final error of ε=0.00005, the average error in the output, x, is $\sqrt{\frac{2 \times 0.00005}{120}}$=0.0009 indicating that on average, the output values have been learned correct to about three decimal places.

5.2.3 The "Line-Pair Evaluator"

The purpose of the line-pair evaluator is to approximate the function; $\frac{l_1^2}{2l_2}$.

5.2.3.1 Simulating a Continuous Function

The network that is taught to achieve this has two inputs and one output (and one hidden layer). According to Hoskins (1989), neural networks are quite capable of simulating elementary continuous functions such as $y=x^2$, $y=e^x$ and y= sin x.

Although the function to be simulated is relatively simple, a few constraints need to be imposed to preserve continuity. In particular, it is necessary to make a distinction between the two physical inputs of the network. One must be designated for l_1 and the other for l_2, so that the largest and smallest values always appear on the same inputs.

Catering for special cases can also destroy continuity. For example, to handle the situations when l_1=0 or l_2=0 or both, additional exemplars representing these cases would have to be incorporated into the training set, possibly resulting in longer, more difficult learning. Instead of doing this, a separate network can be used to recognise these situations and when appropriate, act to suppress or *inhibit* the output of line-pair structure.

5.2.3.2 Using Inhibition to Handle Special Cases

The synaptic inputs to a biological neuron are said to be either excitatory or inhibitatory, that is they either excite or suppress the output of a neuron. Similar terms have been adopted for describing various types of connectivity in artificial neural networks. When a weight is negative, it is said to be inhibitory and when it is positive it is said to be excitatory. In feed forward networks, if one neuron is to be used to inhibit the output of another, then its output must be connected to a large negatively weighted input on the other neuron. This is how inhibition of the line-pair evaluator is achieved. A simple two layer network outputs a binary value 1 when it detects less than two lines and a binary 0 otherwise. The output of this, as described before, is connected to a large negative weight on the output neuron of the line-pair structure and if sufficiently negative, can result in complete inhibition of this network.

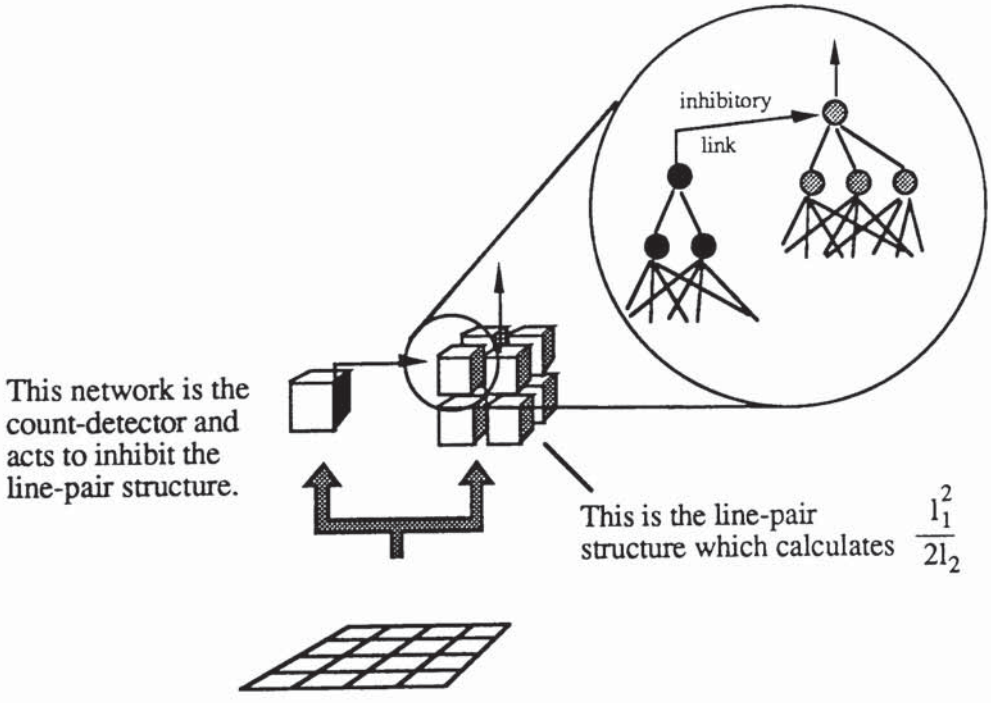


Figure 5.10 Inhibition of the line-pair structure

The count detector is a very simple network and does not require training. It has 16 thresholded inputs and an output neuron using a step activation function with a bias value

set to +1.5. All weights are set to -1. Figure 5.11 shows the step function that is set up by this network.

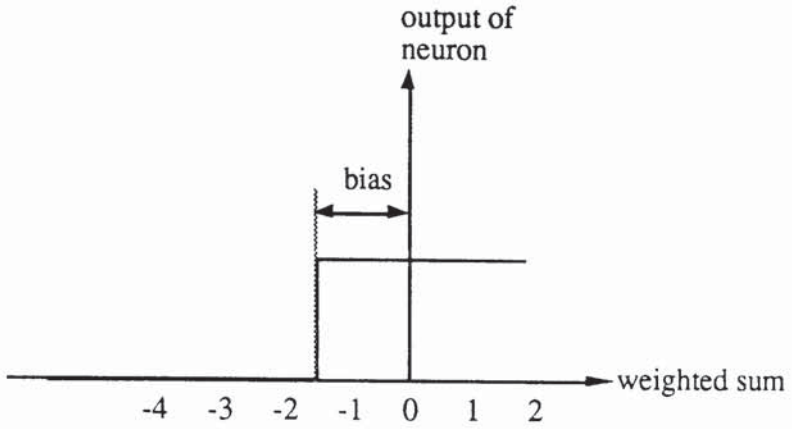


Figure 5.11 The count detector

The only other quantity that needs to be determined is the value of the negative weight on the inhibitory link. The activation function used in the neurons of the line-pair evaluator is the standard sigmoid. Thus the inhibitory weight must be large enough to cause the function to output a near zero value. A weight in the order of -25 was found sufficient to completely dominate the $\frac{1}{1+e^{-x}}$ function by causing an exponential of about e^{25} to be calculated.

5.2.3.3 Results of Training

It is important that the line-pair evaluator reasonably approximates the required function. It is also important that once trained, it can generalise well enough to handle all untrained exemplars. It turns out that this problem can be learned quite readily with small networks, using fairly small training sets. The results from three different tests are shown below, each time the trained network is tested on an appropriate set of trained and untrained exemplars. Note that this problem learns quite slowly, in fact some of the results

obtained, took over 1 million epochs to converge. Thus to keep learning time low, it is important to keep the size of the training set to a minimum.

• Trial 1

Using a small training set having just 10 exemplars and a six unit hidden layer, a final error of 0.000001 was reached. The following table typifies performance of this network.

inputs		actual output	desired output
for untrained exemplars			
0.2	0.1	0.040036	0.025
0.5	0.3	0.090163	0.09
0.8	0.1	0.010277	0.00625
0.97	0.43	0.095292	0.09531
0.47	0.4	0.170887	0.17021
0.7	0.7	0.349657	0.35
0.3	0.1	0.028311	0.01667
0.85	0.23	0.031175	0.03112
0.5	0.45	0.203169	0.2025
		average error = 0.00363	
and for trained exemplars			
0.75	0.5	0.165912	0.16667
1	1	0.499997	0.5
0.5	0.25	0.063423	0.0625
0.75	0.25	0.041357	0.04167
1	0.25	0.031078	0.03125
		average error = 0.00043	

Table 5.1a Performance of a 6 hidden unit line-pair evaluator trained to 0.000001

Using the relation derived in the previous section, 0.000001 represents an average error in the output of 0.00045. This compares very favourably with the value of 0.00043 for the average absolute error calculated between actual and desired outputs of trained exemplars in table 5.1a, indicating that the net has learned the training set relatively well. However, generalisation is not so good, particularly near the extremes of the test set, eg., 0, 0 and is probably due to the lack of exemplars in this region. (See Appendix E for the contents of this training set.)

• Trial 2

The above experiment was repeated for a network of 4 hidden units and the convergence criterion set at a a global error of 0.000002. The intention is that a network with less hidden units might be expected to give better generalisation.

inputs		actual output	desired output
for untrained exemplars			
0.2	0.1	0.031754	0.025
0.5	0.3	0.088076	0.09
0.8	0.1	0.010627	0.00625
0.97	0.43	0.095003	0.09531
0.47	0.4	0.168849	0.17021
0.7	0.7	0.349848	0.35
0.3	0.1	0.024056	0.01667
0.85	0.23	0.031657	0.03112
0.5	0.45	0.202033	0.2025
		average error = 0.002585	
and for trained exemplars			
0.75	0.5	0.168108	0.16667
1	1	0.500108	0.5
0.5	0.25	0.061689	0.0625
0.75	0.25	0.041842	0.04167
1	0.25	0.031035	0.03125
		average error = 0.00055	

Table 5.1b Performance of a 4 hidden unit line-pair evaluator trained to 0.000002

The final error is 0.000002, which relates to an error in the output of 0.00063 and is comparable to the actual accuracy obtained with trained patterns. The average error for untrained patterns is slightly smaller than in trial 1, although the improvement is not significant.

• Trial 3

The current training set is quite small. By including more exemplars in the set, significant improvements can be made. The following table shows the results of an eight unit hidden layer network trained using a larger training set, (this time an additional 8 exemplars have been included; see Appendix E for the full set). Convergence was allowed to reach a final error of 0.000021 (representing an error in the output of about 0.0015).

inputs		actual output	desired output
for untrained exemplars			
0.1	0.05	0.014178	0.012500
0.23	0.12	0.032583	0.031304
0.76	0.32	0.066896	0.067368
0.87	0.56	0.180780	0.180230
0.455	0.28	0.085555	0.086154
0.82	0.76	0.354260	0.352195
0.95	0.46	0.111183	0.111368
0.88	0.87	0.430107	0.430057
0.32	0.21	0.070347	0.068906
0.76	0.67	0.297956	0.295329
0.9	0.28	0.044054	0.043556
0.66	0.55	0.231070	0.229167
0.77	0.22	0.030789	0.031429
0.56	0.11	0.009480	0.010804
		average error = 0.001094	
for trained exemplars			
0	0	0.005451	0.000000
0.12	0.1	0.039260	0.041666
0.09	0.06	0.019508	0.020000
0.25	0.25	0.125289	0.125000
0.3	0.2	0.068388	0.066666
0.5	0.25	0.061620	0.062500
0.75	0.25	0.041100	0.041666
0.6	0.4	0.132751	0.133333
1	0.25	0.032221	0.031250
0.5	0.5	0.249484	0.250000
0.7	0.4	0.113777	0.114286
0.86	0.32	0.059739	0.059535
0.75	0.5	0.167466	0.166666
1	0.5	0.124269	0.125000
0.75	0.75	0.374959	0.375000
1	0.2	0.020175	0.020000
1	0.75	0.281264	0.281250
1	1	0.499820	0.500000
		average error = 0.000918	

Table 5.1c Performance when trained on a larger set

This time an improvement in generalisation is observed. The results indicate that the output of the network should be accurate to about three decimal places.

5.2.4 The "Min-Filter"

The aim of the Min-filter is to output the lowest non-zero value of two inputs. Fifteen of these two input filters can be arranged as in figure 5.4 and because each min-filter will

pass on only the lowest non-zero value, the output at the top of the 'pyramid' will be the actual lowest non-zero value on the 4 x 4 orientation plane. This will represent l_1, the shorter length of the two lines. The max-filter on the other hand, is used to find the largest value on the plane which represents the longest line, l_2.

5.2.4.1 Using a Modified Backpropagation Algorithm

Initial attempts using the standard backpropagation method showed that the output of a trained network can never be zero, even when this is the required value. The reason for this can be seen when the activation function (see figure 2.2, page 27) is examined. The asymptotes of the sigmoid prevent the output of the neurons from ever reaching 0.0 or 1.0. This means that the value delivered by the min-filter will be subjected to some small positive rounding error.

One solution would be to ensure that the network always outputs exactly a zero, instead of a near-zero error value. A modified activation function, see figure 5.12, known as a hard-limiting sigmoid would however accomplish this.

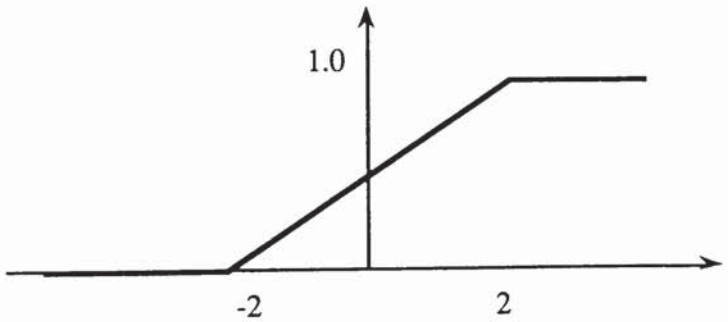


Figure 5.12 The hard-limiting activation function

The slope of the hard-limiter is equal to the gradient of the sigmoid at the point where it crosses the y-axis.

The error that is propagated back in the backpropagation algorithm is a function of the derivative of the activation function[0]. This derivative is zero at the flat regions of the hard-limiter and as a consequence, error values calculated at these points are also zero. When this happens, weight updates for that particular neuron become zero, and convergence ceases. This may explain the practical results which indicate that it is not possible to reach a state in which the output of the min-filter actually becomes 0.0 or 1.0, when all neurons use the hard-limiter. Fahlman (1989) puts forward a very simple solution to this. The derivative of the activation function, the 'sigmoid-prime' as Fahlman refers to it, approaches zero at the asymptotes of the sigmoid activation function and actually reaches zero when a hard-limiter is used. To prevent it from reaching zero, Fahlman's solution is to simply add a small value, such as 0.1 to the sigmoid-prime function. Using this modification, the min-filter can actually be trained to output exactly the required values of 0.0 or 1.0.

5.2.4.2 Alternative Ways of Learning the Same Task

In all the problems discussed so far, the choice of training sets has always been quite straightforward. This does not however mean that other alternative ways of expressing the problem do not exist. In training a network to find the lowest non-zero, the most obvious format for a training set is for example, the following :

inputs		desired output
0.2	0.4	0.2
0.6	0.3	0.3
0.8	0.0	0.8
etc ...		

---

[0] Backpropagation requires a continuous derivative. The derivative of the hard-limiter is not continuous, but since the hard-limiter is an approximation to the sigmoid, the derivative of the sigmoid can be used instead.

However with this type of set and using the modifications discussed in the previous section, training was found to be tortuous with the global error oscillating rather than monotonically decreasing for networks containing a range of hidden units from 10 to 30. This seems to suggest that the problem is a difficult one to learn and that the network is perhaps encountering local minima during learning.

Training of the max-filter (see section 5.2.5) was far easier than the min-filter because the problem is simpler. The max-filter requires the larger of two values to be filtered through, whereas the min-filter requires not simply the lowest, but the lowest non-zero. It appears as if the extra, 'non-zero' constraint makes learning significantly more difficult.

Several alternative input formats were investigated but it was found that one of the most successful ways of re-expressing this problem is to multiply the input values by -1. The effect of this is to change the problem from being a 'find the lowest non-zero' to one of 'finding the biggest non-zero'. The training set now looks like :

inputs		desired output
-0.2	-0.4	0.2
-0.6	-0.3	0.3
-0.8	-0.0	0.8
etc ...		

This may or may not simplify the problem, however re-expression will result in a different weight landscape during learning and it is hoped that this may provide an alternative, possibly quicker route for convergence.

In this case it turns out that using about 18 hidden units, an error of 0.000001 can be reliably reached in approximately 350,000 epochs or less and table 5.2 shows that the network generalises well and provides a good degree of accuracy.

inputs		actual output	desired output
0.85	0.24	0.240968	0.24
0.23	0.12	0.120159	0.12
0.56	0.34	0.340054	0.34
0.414	0.666	0.414041	0.414
0.345	0.123	0.123094	0.123
0	0.543	0.542981	0.543
0	0.01	0.009960	0.01
0.678	0.680	0.678050	0.680
0.24	0.98	0.240062	0.24
0.456	0.567	0.456036	0.456
0.213	0.2	0.200199	0.2
0.2	0.124	0.124177	0.124
0.836	0.478	0.478027	0.478
0	0.1	0.099963	0.1

Table 5.2 Performance of the min-filter with untrained exemplars

5.2.5 The "Max-Filter"

The max-filter is very similar to the min-filter, but is a far simpler task in terms of ease of learning. This time, the network has to output the larger of its two inputs. The problem can be solved using standard backpropagation, but convergence is faster if the modified algorithm and hard-limiter are used.

The training set for this problem is quite straightforward and can be found in Appendix C. It contains just 59 exemplars, with values in the range 0 to 1. Low global errors can be reached quite readily, for example, with 6 hidden units an error of 0.000001 can be reached in between 7000 and 25000 presentations of the training set. The problem can also be solved using 4 hidden units, but convergence then becomes more difficult. The trained network also generalises well, see section 5.3.3.

5.2.6 The "Corner-Detector"

The objective of this task is to train a network to detect and categorise corners. Figure 5.5 shows that this particular network is different from the others used so far, since it uses

three outputs, not one. Each is a binary output that is used to indicate the presence or absence of acute, obtuse or right angled corners found in a given receptive field.

In section 4.3.2.3, it has already been suggested how a corner may be detected. On this basis, it is possible to define a set of patterns which would describe various kinds of corner. Figure 5.13 shows how some corners would appear in a receptive field.

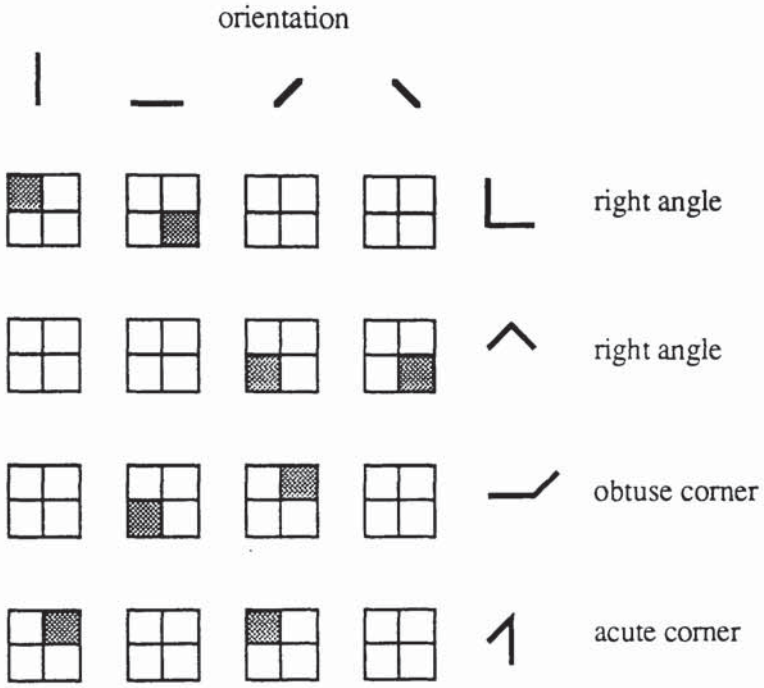


Figure 5.13 Examples of some corners

The training set will therefore consist of a list of patterns representing every instance of a corner in a receptive field, along with its appropriate output. The set will also include exemplars representing line occurrences which do not constitute corners. Appendix F lists all the corners that are accepted and the resulting training set.

As well as the usual corners, three further additions were made to the basic set of corners.

1) An alternative way of encoding acute corners was adopted. The reason for this was to permit a larger variety of shapes to be coded.

2) Since the detection of corners is expected to occur for two lines only, the training set has been designed accordingly. But problems occur with simple diamond shapes. Squares are conveniently coded in four separate receptive fields, one for each corner and each with two lines. Diamonds are coded slightly differently and because of this five receptive fields are generated. Four for each of the corners and a fifth containing all four lines (see figure 5.14). To handle this shape, extra exemplars must be included in the training set to explicitly indicate that a receptive field with more than two lines, is not a corner.

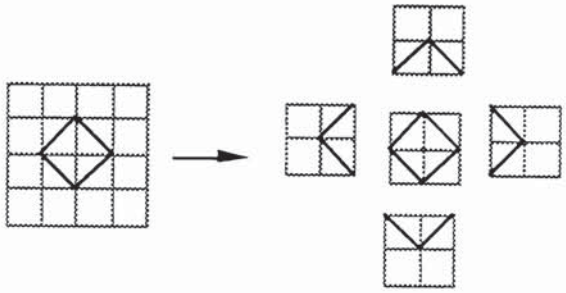


Figure 5.14 Receptive fields for a diamond

Thus the pattern for the middle receptive field in figure 5.14 must be included in the training set and be trained a non-corner.

3) Some receptive fields will contain no lines and others only one line. It is necessary to include these instances in the training set too.

Results found that by using a network with 7 or 8 hidden layer units and the modified learning algorithm, a final global error of 0.000001 can be attained quite readily with the given training set, often taking less than a thousand epochs. However, it must be remembered that this global error is calculated over three outputs, not one as usual. Thus the level of accuracy is actually a third of this value and turns out to be very satisfactory, since in this problem, the outputs are binary values and with the hard-limiting activation function, these can be readily arrived at.

5.3 Analysis of Some Training Results

The building blocks of the Perceptual Network are the individual three layer networks that have been trained to perform each set task. Because of the role they play in the network, much attention has been directed towards producing accurate and reliable networks. Efforts have even been made to establish the reproducibility of the results. In particular it has been verified that the expected minima can be reached in repeated training runs of the subnets. During this development stage, a great deal has been learned about the characteristics, properties and the problems associated with the learning process itself. The following sections aim to describe some of the problems and properties that have been encountered during the development stage.

5.3.1 Generalisation Problems of the Distance-Discriminator

One attractive feature of neural networks is their ability to generalise. That is, once trained on a set of exemplars, the network should be able to correctly categorise other, untrained exemplars. The underlying theory is that the network should be able to find and then use the essential qualities that describe the members of the training set.

This is not however the case with the distance-discriminator. All experiments indicated that if the existing training set of 120 exemplars was reduced, then training would not compensate for any missing patterns. For example, in one test, a single pattern was removed from the full set. After training, this pattern was then presented to the net to see if the correct value could be obtained. Three trials were ran and in each a different randomly selected pattern was removed. In all trials, a 10 hidden unit network was trained to a final error of 0.00005, taking between 45000 and 70000 epochs.

Results

trial	desired output	input pattern removed	actual output
1	0.447214	0000000100000010	0.992962
2	0.707107	0100001000000000	0.985740
3	0.316228	0010000000000001	0.032894

The outputs generated are not even close to the required value. Such results strongly suggest that in this case, generalisation is not possible. This has implications for the size of the code planes, since without the ability to generalise, training must use all possible combinations that could be encountered. With a 4 x 4 plane, the number of possible combinations of the location of two lines is 120. With larger code planes, this number increases dramatically. A 5 x 5 plane for example generates a total of 300 combinations and a 6 x 6 would generate 630. Without generalisation, it becomes necessary to train using the full set and this will greatly increase training times.

5.3.1.1 Limited Connectivity

Preliminary experiments were undertaken to investigate whether or not the distance-discriminator had learned to exploit the two-dimensional (2D) nature of the input. This was achieved by examining how a trained version of the net generalised to symmetrical input patterns.

The results confirmed what was suspected, that the net showed significant differences in the outputs generated for patterns which are identical, but rotated. This implies that the network is treating the input vector as linear, not as 2D. If instead the network can be made to learn to use the two-dimensionality of the problem, then the prospects for correct generalisation are greatly improved. In an attempt to do this, trials were undertaken in which the connectivity of the hidden layers was limited in a way that was thought would facilitate learning in 2D. This involved not a fully connected hidden layer, but 8 hidden layer neurons of 4 inputs each, such that half of them took inputs from each of the 4 rows

and the remaining half took inputs from the 4 columns. However it appears as if restricting the connectivity makes learning difficult, as all attempts were unsuccessful and a global minima could not be reached.

5.3.1.2 A Possible Explanation for Generalisation Problems

The parity problem is another for which generalisation is difficult. The reason for this is that changing just one bit, alters the output.

eg. for even 3 bit parity

input	parity bit (output)
0 0 0	0
0 0 1	1
0 1 0	1
0 1 1	0
1 0 0	1

etc...

It is well known that the parity problem is difficult to learn, especially for large input bit patterns. The difficulty arises because each patterns gives rise to its own disjoint point and training is unable to cluster these points in any efficient way.

Having established that the distance problem is linear and not two dimensional, it can be compared to the parity problem. Patterns exist which differ only in one bit, yet generate widely different outputs, see figure 5.15.

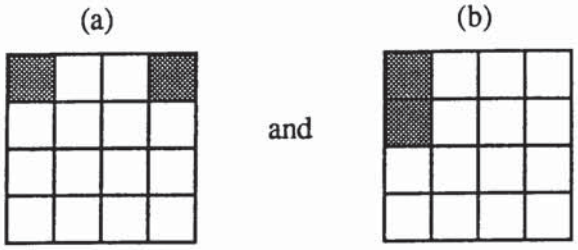


Figure 5.15 Similarity of input vectors

Pattern (a) leads to the output / input vector :

```
0.33333      1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
```

and (b) leads to the output / input vector :

```
1.00000      1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
```

Although the input patterns differ by only one bit position, the actual distances represented by the patterns, are significantly different. This is similar to the parity problem in which similar patterns also give rise to contrasting outputs. If this is so, then the distance problem can also be expected to experience generalisation problems.

5.3.2 Generalisation Problems of the Corner Detector

Unlike the distance-discriminator, the corner-detector does show some potential for generalisation but somewhat unpredictably. The results in table 5.3 have been obtained from an 8 hidden unit, single output network trained to an error of 0.000001 on a training set of 142 exemplars (see Appendix F) and show that the net generalised correctly 4 out of 7 times. In each case, to test the ability to generalise, a randomly selected exemplar was removed from the set, the network trained on the remaining set and then tested with the missing exemplar to see if it could be correctly classified.

trial	desired output	input pattern removed	actual output
1	1.000000	0000100000001000	0.246393
2	0.000000	1000000000010000	1.000000
3	1.000000	0000000000011000	1.000000
4	1.000000	0000000000100010	1.000000
5	0.000000	0000100000010000	1.000000
6	1.000000	0000000001000000	1.000000
7	1.000000	0000000000010110	1.000000

Table 5.3 Generalisation of the corner-detector

Further investigations show that the network can also generalise when more than one pattern is removed. For example, in one test, 10 patterns were removed and when tested after training, the net correctly classified 7 of these. Another test showed that a trained net

could correctly classify 13 out of 20 missing patterns. Although these results are encouraging, they are unpredictable. Thus it is not possible to determine for which patterns generalisation works and for which it does not. The following tests highlight this problem. When training is repeated, with all parameters kept identical (other than random initial weights), the final performance may be expected to be identical. However, this is not the case. For example, two identical training runs were undertaken for an 8 hidden unit network and then each was tested, first on members within the training set, and then members outside the training set.

input patterns	Trial 1 output	Trial 2 output
0000000000100001	0.000000	0.000000
0000000000000011	1.000000	1.000000
0000000100001000	0.000000	0.000000
0000000000110000	1.000000	1.000000
0000000100000000	1.000000	1.000000
0000000010010100	1.000000	1.000000

Table 5.4a Performance of identically trained nets on members of the training set

For members of the training set, performance is seen to be identical. This may suggest that the network has reached the same solution for both trials. That is, in both cases, the same global minima and the same set of weights have been reached.

input patterns	Trial 1 output	Trial 2 output
0111000000000000	1.000000	0.611521
0000110001100000	1.000000	0.296605
0010101010000000	1.000000	0.000000
0000000111111111	1.000000	1.000000
1111111111111111	0.000137	0.000000
1111000011110000	0.581642	0.000000
0000100010001000	1.000000	1.000000
1110001010000001	0.830373	0.000000

Table 5.4b Performance of identically trained nets on patterns outside the training set

However, when the same two networks are presented with patterns outside the training set, performance is far from identical, as table 5.4b shows. This would suggest that in

each case, training leads to a different set of weights. It just so happens that the minima reached in both cases can adequately describe the same training set, but responds differently to patterns outside this training set.

The following section re-addresses this problem but in terms of generalisation of the max-filter.

5.3.3 Generalisation of Min / Max Filters

The table 5.5 shows how well the max-filter can handle new, untrained exemplars. The filter in this case was trained to a final global error of 0.000001 on a 6 hidden unit network trained on a set of 59 exemplars.

inputs		actual output	desired output
0.85	0.24	0.849971	0.85
0.23	0.12	0.230058	0.23
0.56	0.34	0.560051	0.56
0.414	0.666	0.665891	0.666
0.345	0.123	0.345068	0.345
0	0.543	0.542944	0.543
0	0.01	0.000000	0.01
0.678	0.680	0.679896	0.680
0.24	0.98	0.979913	0.98
0.456	0.567	0.566953	0.567
0.213	0.2	0.213101	0.213
0.2	0.124	0.200058	0.2
0.836	0.478	0.835968	0.836

Table 5.5 Generalisation of max-filter.

Compared to the distance-discriminator and the corner-detector, the max-filter network has very good generalisation properties, as table 5.5 shows. Furthermore, this ability to generalise remains constant and can be reliably reproduced in repeated training sessions.

5.3.3.1 Generalisation and the Weights of a Trained Network

Five separate trials were set up, using a network with 4 hidden units and the same 59 member training set and all training parameters were kept identical, except the random initial starting weights. Table 5.6 shows the response of the network to untrained exemplars, after each of the five trials. Unlike the corner-detector, table 5.6 shows that untrained exemplars when tested, all generate the same value (to within the accuracy reached). This would seem to indicate that the weights and the global minima that are being reached are also the same.

inputs		Trial 1 output	Trial 2 output	Trial 3 output	Trial 4 output	Trial 5 output
0.85	0.24	0.850030	0.849986	0.850034	0.850057	0.849983
0.23	0.12	0.230149	0.230109	0.230156	0.230192	0.229999
0.56	0.34	0.559952	0.560046	0.559962	0.560171	0.559989
0.414	0.666	0.666025	0.666110	0.666030	0.666157	0.666059
0.345	0.123	0.345054	0.345086	0.345058	0.345161	0.344997
0	0.543	0.543039	0.542884	0.543042	0.542602	0.542844

Table 5.6 Generalisation of identically trained max-filters.

Examining the weights of a network is often not very fruitful, especially if the purpose is to establish how the network is working. The sheer volume of connectivity as well as the likelihood of distributed representations makes such attempts unviable. For small networks such as this, however, it is not too difficult to inspect and compare the weights of different trials, particularly if the aim is to look for evidence suggesting that the weights reached are the same. Table 5.7 below shows the weights from the same five trials as above.

An analysis of table 5.7 shows a close correlation, particularly in the hidden layer, between weights of corresponding neurons in different trials. There are some differences between the signs of weights in the output unit, but these correspond to differences in the sign of the bias of the appropriately connected hidden unit. In terms of weight space, these results suggest that the same solution is reached every time.

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5
Output neuron					
bias	-0.010379	-1.155083	-0.723123	0.386089	-1.916570
weight 1	2.086542	2.900048	1.376331	3.846242	2.465938
weight 2	-4.716921	-4.589755	4.632215	-4.525097	4.274971
weight 3	3.380001	3.311513	-2.851769	-3.015640	-3.143729
weight 4	3.243570	3.387650	2.825906	3.521638	3.193904
Hidden layer neurons					
neuron 1					
bias	1.725603	1.681682	1.530013	1.613849	1.426296
weight 1	4.165188	5.430847	5.220040	6.371010	6.053165
weight 2	2.640293	3.105887	4.545412	4.323667	5.639293
neuron 2					
bias	2.000550	2.000344	-2.000582	2.001078	-2.001673
weight 1	-3.390236	-3.484351	-3.451015	-3.530763	-3.738562
weight 2	3.389151	3.483621	3.452044	3.528694	3.741653
neuron 3					
bias	-5.078247	-5.069464	5.682373	4.933338	5.305197
weight 1	-0.000209	-0.000086	0.000119	0.000537	0.000884
weight 2	4.731595	4.829363	-5.607465	-5.303084	-5.086904
neuron 4					
bias	-1.208340	-1.001311	-1.717215	-0.511490	-1.253359
weight 1	4.928723	4.719830	5.657011	4.535630	5.003923
weight 2	0.001746	0.001320	0.002342	0.005241	0.001299

Table 5.7 Weights reached after training, for five repeated training runs.

Why should this task reach the same point in weight space each time and the corner-detector not? One explanation for this is related to the dimensionality of the input. In the corner problem, 16 binary inputs are used leading to a universe of 2^{16} possible input patterns. However, the training set contains only 142 of these, and so is just a subset of the universal set. Under these circumstances it is more than likely that there are several minima in weight space which satisfy the requirements of this subset. Had the training set contained all 2^{16} possible input patterns, then only a unique point could satisfy this set. For the max-filter, the universe of inputs is the range of real values between 0.0 and 1.0. Under this hypothesis, it would appear that since the same solution in weight space is reached every time, the training set must define the entire domain of inputs, that is, the universal set. This is an important result as it has implications for the ability to generalise. If for example, the training set for a given problem is known to be representative of the entire domain of inputs, then it can be expected to generalise for all values within that universe. This is the case for the max-filter. If however the training set describes only a

subset of the universal set of possible inputs, eg. corner-detector and the distance-discriminator, then that training set cannot be expected to generalise well for all patterns.

5.4.4 Accelerated Learning with the Modified Backpropagation

Preliminary work with the backpropagation simulator investigated the possibility of speeding up learning and one of the techniques tested was based on Scalettar and Zee (1988) method known as 'drilling'. This made the assumption that in every training set there exist exemplars that are more difficult to train than others, and to accelerate learning, greater effort should be placed in training those exemplars.

Ideally such techniques should be automatic and problem independent. In practice this is rarely the case and the same is true for this method which required a lot of ad-hoc choices for the various parameters used. This method was therefore abandoned.

The modified backpropagation algorithm (see section 5.2.4.1) on the other hand was found to have one further advantage not yet mentioned, in that it results in significantly faster learning. To investigate this, the max-filter was trained using standard backpropagation and the modified algorithm. Ten trials were completed in each case so that an average could be calculated for the number of epochs to reach a given global error. All parameters such as learning rate, momentum term, number of hidden units etc. were kept identical. Using standard backpropagation, the average number of epochs to reach an error of 0.01 was 16899, whereas with the modified algorithm, the average was 3242. The result supports the view that accelerated learning takes place and furthermore, appears to be problem independent. When the modified and standard algorithms were tested and compared with other problems, noticeable differences were recorded. Table 5.8 exemplifies the level of improvements obtained with other types of problems.

	corner evaluator	distance-discriminator	line-pair evaluator	parity 6
no. of hidden units	10	10	8	12
average epochs with hard-limiter	209	876	1151	486
average epochs with sigmoid	>10000	1147	3127	1612

Table 5.8 Evidence for faster learning

The above results show a consistent reduction in the learning phase, when compared with trials done with standard backpropagation. The reason for this speed up is the use of the modified 'sigmoid-prime' function. Near the flat portions of the activation function, the magnitude of the sigmoid-prime diminishes resulting in the propagation of relatively small errors. Thus learning becomes very slow. If the value of the sigmoid-prime is artificially 'propped up' by adding a small constant value (0.1 was used), then this will ensure that a certain level of error propagation will still take place, even near or on the flattest regions of the activation function, where convergence would be expected to be slow. The result will be an increase in the learning speed.

Chapter 6

Testing and Evaluation

When evaluating a system, it is often useful to make comparisons with similar systems if they exist; since a quantitative assessment can then be made. If similar systems do not exist then clearly a comparative evaluation is not possible and the system must be judged against the original specification.

6.1 Software Evaluation

Chapter 5 has already dealt with the training and testing of the various sub-nets used by the Perceptual Network. It now remains to test the network as a whole, and also to see how it responds to simple shapes.

The overall significance value calculated by the Perceptual Network is the sum of seven different components; four parallel-line significances and three corner significances. To test the net, different input patterns were presented and the resulting outputs considered against expected values and required precision.

The input patterns used for testing were carefully chosen so as to test as much of the network as possible. Four sets of test data were devised to accomplish this, each one testing a different aspect of the net :

1. All primitives
2. The same primitive at all locations
3. The same shape with varying line lengths
4. Simple shapes

The results of these tests are shown in the following sections :

6.1.1 Testing Response to All Primitives

The purpose of this test set is to establish that the Perceptual Net can actually recognise all the possible primitive corners and parallel lines. The set is fairly large and contains 48 patterns, each one representing a corner or a parallel line pair. The length of the lines in each primitive is kept fixed to the maximum value of 1.0. See Appendix G for the full test set and the results.

Comparing the actual values with the desired outputs, it can be seen that nearly all of the outputs are found to be correct to three decimal places. This indicates that the Perceptual Net is functioning correctly and that at least in isolation, all of the taught features can be satisfactorily detected. The absence of features is also detected well, although there are a few instances when instead of 0.000000, a very small spurious value such as 0.000040 or 0.000051 is output, see for example figure 6.1 which shows an occurrence of this.

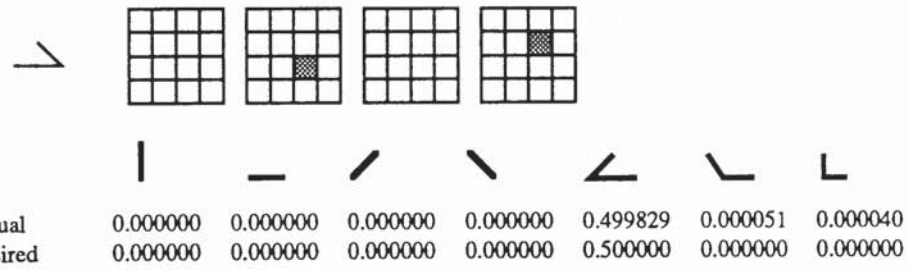


Figure 6.1 Example of spurious outputs

These errors are small and insignificant and seem to occur only for corners, indicating that they have arisen in the corner-detector network.

6.1.2 Ability to Handle Translation Invariance

The Perceptual Net is designed to process features independently of their location in the input. Some features are naturally translation invariant such as the detection of parallel lines. However, the detection of corners relies upon the use of many identical 'local

corner detectors' (see figures 5.5 & 5.6) operating at every receptive field. To test these detectors and the ability to handle translation invariance as a whole requires that the same pattern be tested at all locations.

The input set for this test consists of patterns of right angles at all possible locations; this accounts for the nine internal receptive fields. An orthogonal right angle cannot be coded on the edges and so to test some of the edge receptive fields, a rotated right angle at a few locations is also included. Again, all lines involved are 1.0 long. The test set and the results for 16 patterns is shown in Appendix H.

Again the results are very good, with all values comfortably accurate to three decimal places. The net seems to handle translation invariance very well. This is the case for simple corners, and it would be expected that other more complex patterns would also be handled in the same way since all component features of a pattern are handled separately as simple corners.

6.1.3 Generalisation of Line Lengths

The components of the Perceptual Network that deal with line lengths are designed to generalise for all permitted line lengths (i.e. between 0.0 and 1.0). The networks in question are the line-pair evaluator, the min-filters and the max-filters. The intention of this test set is to determine how these networks when incorporated into the Perceptual Net handle variations in line length. The test set consisted of eight versions of a square, such that each occurred in the same location but had lines of different lengths. See Appendix I for the test data and the results.

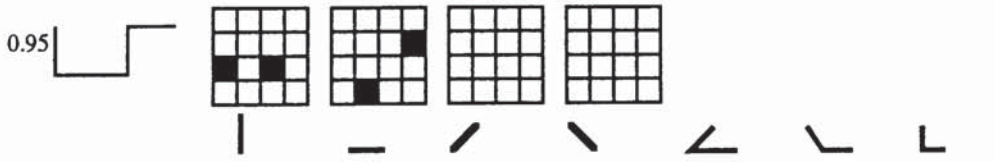
The first four test patterns contained combinations of line lengths which appeared in the training sets of all the above mentioned networks. For these line lengths, the actual values are accurate to three decimal places. However, it appears as if the accuracy of the outputs for untrained exemplars is appreciably less, with some outputs accurate to only two

decimal places. This accuracy is slightly worse than when each of the networks mentioned was tested individually on untrained inputs and a likely cause of this is the possible escalation of errors as a value passes through the various filter networks (figure 5.4). In absolute terms the error is within 10% and does not significantly affect performance.

6.1.4 Testing with Simple Shapes

The aim of this test is to establish how well the Perceptual Net handles combinations of features, as they would appear in simple shapes. The test set consisted of eight randomly selected shapes and patterns with various line lengths. See Appendix J for the test set and the results obtained.

The presence or absence of corners and parallel lines appears to be correctly detected in all cases, however the error in the significance values are sometimes higher than might be expected. For example, in one case (see figure 6.2), a value of 1.564517 for the significance of right angled corners is generated, instead of the expected value of 1.425; this is an error of about 0.15 (10%).



actual	0.260705	0.167717	0.000000	0.000000	0.000000	0.000000	1.564517
desired	0.237500	0.167900	0.000000	0.000000	0.000000	0.000000	1.425000

Figure 6.2 Performance with actual shapes

As well as the likelihood of errors propagating within the network, as in the filter structures, errors can also be generated due to compounding. For example, in the case that the input pattern contains 3 right angles, each of which gives a small error of the same sign, then the final error being the sum of the individual errors would be

unexpectedly large. This happens to be the severest case observed and apart from a few results showing errors of about 0.05, most other values are correct to about 2 decimal places.

6.2 System Evaluation

An assessment was made of the scope of performance of the Perceptual Net. To aid in this assessment, it is necessary to devise criteria suitable for the examination. Three areas have been chosen for evaluation; the validity of significance values, the limitations of the Perceptual Net and its applications.

6.2.1 Validity of Significance Values.

Section 6.1 has already shown that the significance values generated by the Perceptual Net net can be quite accurate. However in a few cases, some parameters used to evaluate the significance values have already lost some of their accuracy, because of a lack of precision in coding. There are two instances in which this happens.

6.2.1.1 The Distance-Discriminator

The distance between parallel lines is defined as the distance perpendicularly from the longest line, to the midpoint of the shortest line (see section 3.3.6.2). Since end-point information is not stored, this midpoint information is also lost when the lines are represented using the coding scheme. Furthermore, with each orientation grid being 4 x 4 in size, the position of a line can only ever be accurate to a quarter of the width of the original image. Whereas the accuracy to which the distance-discriminator has been trained is far higher than that actually needed. This does not necessarily indicate that this network is working inefficiently, as the time taken to evaluate an output is the same irrespective of the accuracy to which the net is trained. The inefficiency arises in the training stage, since it would have been easier and faster to train the same network to a coarser accuracy.

6.2.1.2 The Corner-Detector

The Perceptual Net responds to three different types of corners, acute, obtuse and right angles. When a corner is detected, the appropriate local corner detector outputs a significance value for that type of corner. The significance is based only on line lengths, not on the angles between the lines. Intuitively, this does not seem to be correct, since the significance of a corner is related to the angle between the corresponding lines. For example, as the angle increases, the significance of a corner decreases, until eventually the lines become collinear and the significance is then zero. This suggests that the significance of a right angled corner should be greater than the significance of obtuse and acute corners having the same line length. This is not taken into account by the Perceptual Net.

6.2.2 Limitations

The major limitations of the Perceptual Net and the coding scheme are discussed below :

6.2.2.1 Orientation Planes

At most, only two lines per orientation plane can be coded simultaneously. This is because the distance-discriminator can only find the distance between two coded lines at any one time. With less than two lines, distance is not significant and any values generated by the net are ignored. When three or more lines are coded, the question of finding the distance becomes problematic since more than one distance value could be generated. For example, with three lines, three possible combinations of line pairs may be defined, leading to three different distance values. Figure 6.3 shows the output of the net when three vertical lines 0.8 long are input. The net correctly indicates that vertically parallel lines are present, but the output significance is actually meaningless. The solution

adopted by Lowe is to locate line pairs only, and then to group together these pairs with others at a later stage.

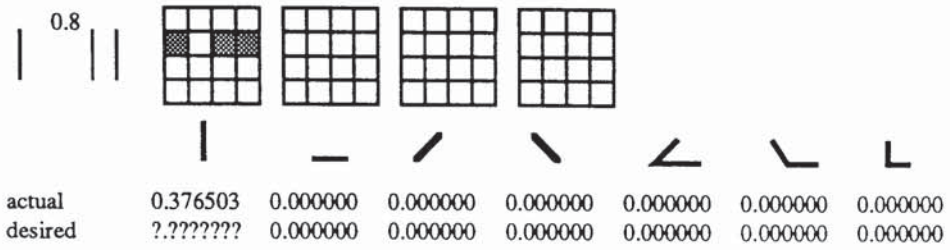


Figure 6.3 Invalid pattern - three lines of the same orientation

6.2.2.2 Receptive Fields

Similarly for the detection of corners, each receptive field is limited to at most two lines (except for the special cases referred to in figure 5.14). The reason for this is that the corner-detector is taught to respond to simple corners consisting of line pairs. Even this leads to a training set of 144 exemplars, comprised mainly of permutations of 2 binary 'ones' amongst 14 binary 'zeros' (120 combinations + 24 special cases). This restriction means that shapes such as a triangle shown in figure 6.4, cannot be correctly handled. To cater for three lines per receptive field, more exemplars covering all the possibilities of 3 'ones' amongst 13 'zeros' would also have to be included. This would increase the training set by another 1120 exemplars, with a consequent increase in the computational resources required.

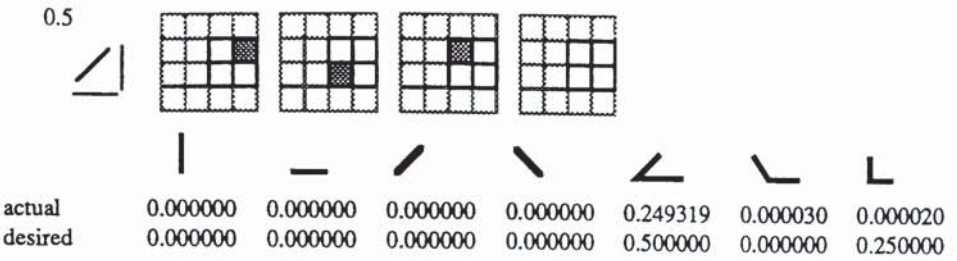


Figure 6.4 Invalid pattern - a triangle with three lines in a single receptive field

6.2.2.3 Collinear Lines

Any pair of lines detected on an orientation plane is automatically assumed to be parallel, although it is possible that the lines are actually collinear. To distinguish between parallel and collinear lines requires retrieval of extra information. Lowe (1987) presumes that parallel lines overlap and that collinear lines are separated along the direction of their length. However, it would be impossible to use this criteria for the detection of collinear lines in the system developed here, since once coded all end-point information is lost.

The detection of collinear lines could be achieved through the use of detectors, specially arranged to recognise collinear configurations. Figure 6.5 shows how this may be done for vertical lines. Appropriately 'wired' detectors would be needed to handle occurrences of collinear lines at all other orientations as well.

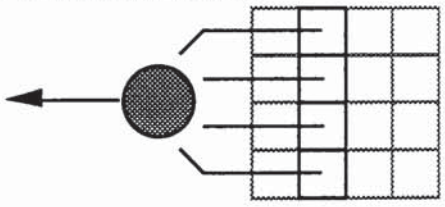


Figure 6.5 Connections for a vertically collinear line detector

6.2.3 Perceptual Net Applications.

As mentioned in chapter 3, SCERPO (Lowe, 1987) identifies perceptual groups and attempts to rank them in order of perceptual significance. A similar task can be performed by the Perceptual Net. Figure 6.6 shows a series of 16 different perceptual groups

together with their total significance values. All line lengths are shown to scale, and the grid location through which the major portion of the line passes is used to code that line. The line lengths in each pattern are indicated alongside the patterns and if only one value is shown, then all the lines in the pattern have this length.

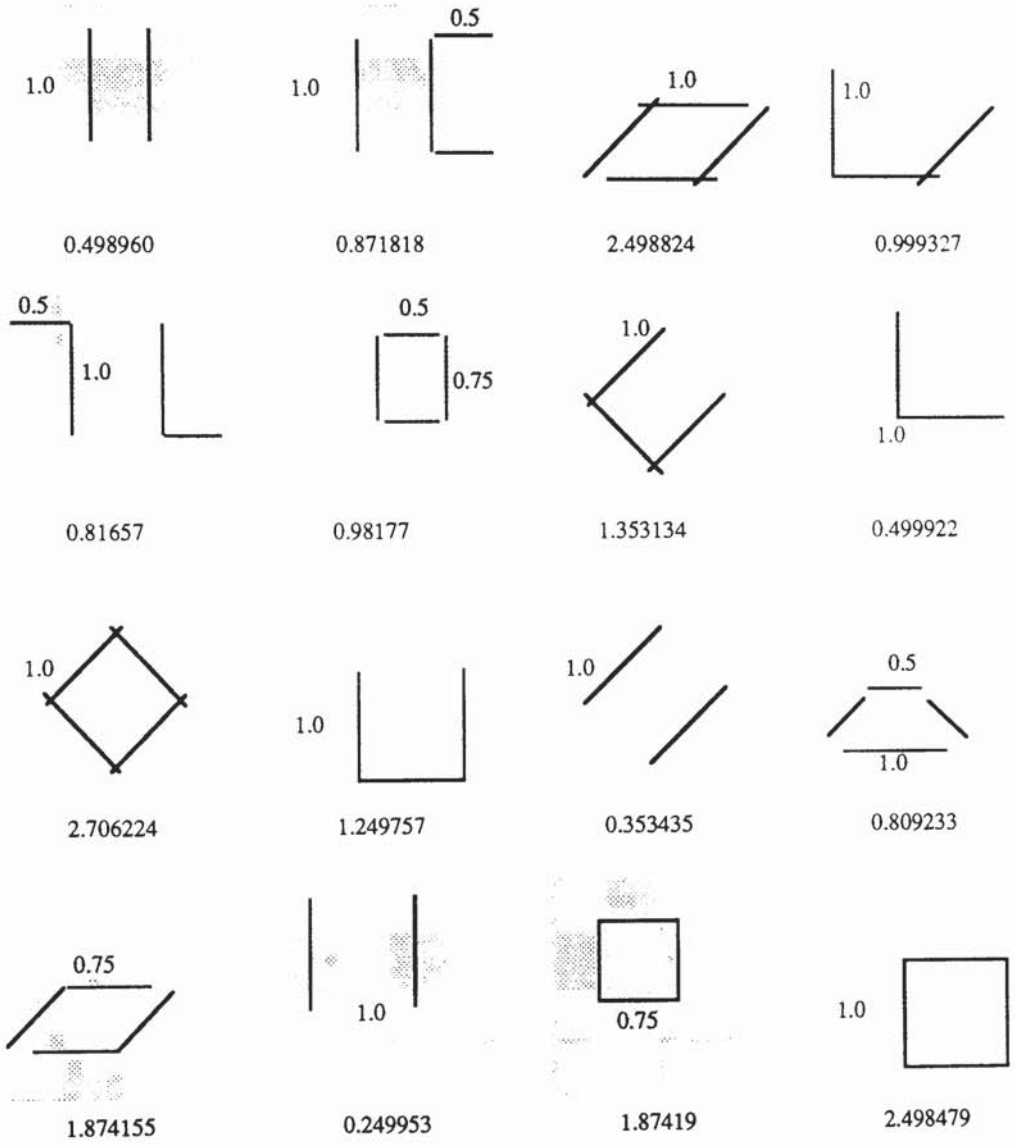
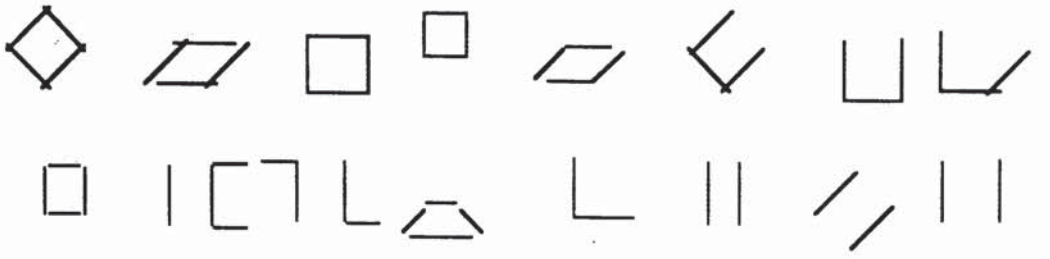


Figure 6.6 Total significances calculated by the Perceptual Net for some perceptual groups

Figure 6.7 shows the same patterns but placed in their correct order of significance.

Most significant



Least significant

Figure 6.7 Ordering of perceptual groups according to total significance.

Main observations

- Significance values are not rotationally invariant (see below).
- Equally sized parallelograms and squares are effectively equal in significance because identical significances are generated for all types of corners.
- It is possible for very similar shapes to generate different significance values, for example the distance between the vertically parallel lines is similar in both cases, but because they are coded discretely, a large difference is recorded.

An inspection of figure 6.7 shows that the ordering of the perceptual groups is similar to what might have been expected and appears to be intuitively correct. For example, the closed shapes, such as the squares and parallelograms are found to have the highest significance values and in terms of Lowe's geometrical equations, the least likely to have occurred by some accidental alignment. Whereas the simplest shapes, such as the corner or parallel lines have the least perceptual significance.

Thus the Perceptual Net seems to provide a means of grading the importance of a perceptual group. One good way to test the validity of these results would be to devise a test which could obtain similar results from human observers.

The output of the Perceptual Net has already been shown to be invariant to translation. However, the total significance values are not quite invariant to rotation. Consider the square of line length 1.0 and the diamond of line length 1.0. Ideally these should have the

same perceptual significance since they are the same shape at differing orientations. Practically however, the significance values are found to be slightly different. The reason for this is coarse coding which arises because the distance between diagonally coded lines and identical vertical or horizontal coded lines is different. This is illustrated in figure 6.8 which shows an identical pair of lines at different orientations and it can be seem that the distances **a** and **b** are geometrically different.

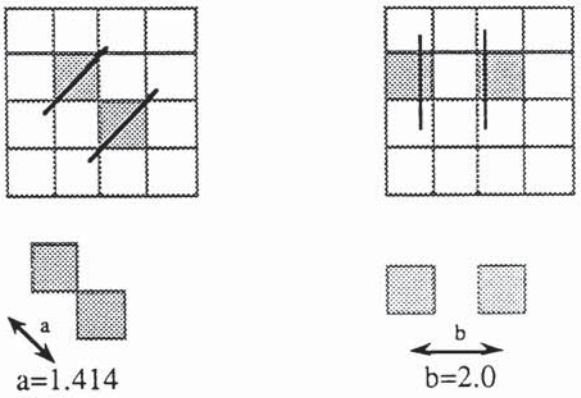


Figure 6.8 Illustration of cause of lack of rotation invariance

Chapter 7

Shape Recognition

Section 6.2.3 described a possible use for the Perceptual Network. Ultimately, such a network could be used not only to order perceptual groups, but could also provide some recognition of actual objects. This would naturally require a much more complex system necessitating the integration of higher level knowledge with lower level processes. At a more basic level though, it would be useful if the net could recognise not only whole objects, but portions or segments of objects. This is exemplified by the method proposed by Burns and Kitchens (1987) in which a pre-compiled decision tree is used to guide the matching process (see section 3.2.4). The basic steps involve recognising simple perceptual groups and pre-compilation is used to find a suitable group or groups that could uniquely identify each object.

This chapter aims to show how the outputs from the Perceptual Network could be used to recognise perceptual groups and other simple line patterns which could be useful in the recognition of whole objects. Two approaches to recognition will be presented, one based on backpropagation and another based on the Kohonen feature map. Until now, backpropagation has always provided the best obtainable solution, however, for our study the Kohonen algorithm was found to have several useful advantages.

7.1 Using Backpropagation for Recognition

Even with simple coding methods, it is possible to generate many thousands of different shapes. So one of the desired features of a network for recognising shapes, is that it should be able to generalise, otherwise the training set must include every conceivable shape. The training set used here contains examples of squares, parallelograms, pentagonal shapes and hexagons along with the variations that can be generated by

selectively removing one or more sides from each of these basic shapes. Each shape is present in three sizes, with line lengths that are long, medium and short and all the lines in each shape having the same length. The last requirement is introduced to retain simplicity. Each shape is described using the seven significance values that are generated by the Perceptual Network.

7.1.1 Format of Desired Outputs

Devising a format for the target outputs is not a trivial problem since it is not clear how each shape in the training set should be classified. The basic shapes could be used to define four categories referring to squares, parallelograms, pentagons and hexagons so that there would be four binary outputs to indicate which type of input pattern had been presented. However, simple binary coding of a restricted size is limited and is not flexible enough to permit much variation of the basic shape.

Instead, a format is adopted which uses real values in place of binary outputs. The hope is that this will facilitate generalisation by using an output representation which can indicate the shape and size of the input pattern. Figure 7.1 serves to illustrate this. For the largest square, the desired outputs indicate the presence of a square and the magnitude of the value has been set arbitrarily to 1.0 to indicate that the shape is in fact the 'best' square. The second square is half the size and its desired value is set to 0.5. Continuing this scheme further, the smallest square (coarse coding restrictions force the sides of the square to be separated), is set to 0.1 and a large square with a side missing is set to indicate 0.75 of a square. These output values are chosen arbitrarily but are adequate for indicating the basic shape of a particular input pattern.

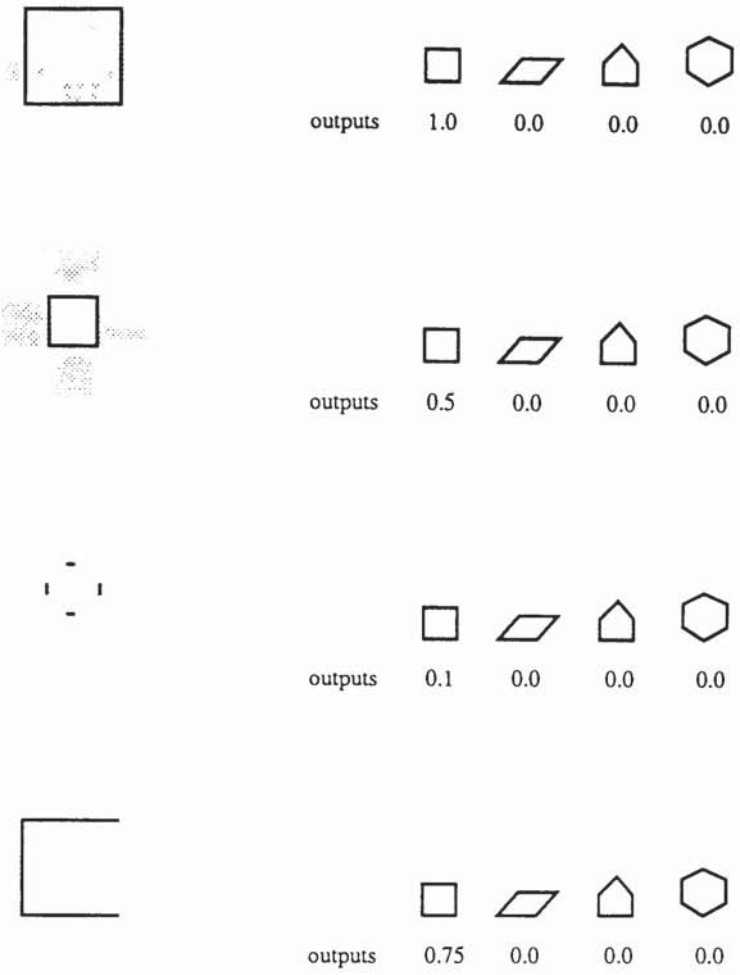


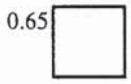
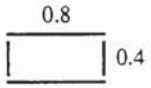
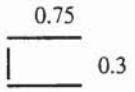
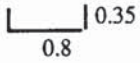
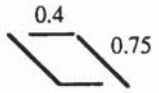
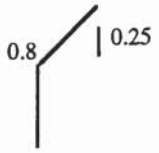
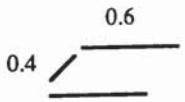
Figure 7.1 Examples of output formats

Figure 7.1 is just a sample from the actual training set used. Appendix L lists the entire contents of the training set which contains a full complement of squares, parallelograms, pentagons and hexagons at all allowable angles and at three different 'sizes'. Also included are descriptions of the shapes obtained when lines are selectively removed from each of the basic shapes. An attempt has been made to avoid including patterns which give rise to the same set of seven parameters.

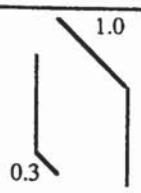
7.1.2 Performance of a Trained Net

A network with 7 inputs, 4 outputs and 14 hidden units was trained on the set in Appendix L to a final error of 0.1 usually in less than 40000 epochs and often in less than

10000 epochs. This represents an error in each output of about 0.015 which is deemed to be sufficiently accurate. In order to test the ability of the net to generalise, two test sets have been devised. The first consists of 21 shapes taken from the training set but with modified line lengths and the second set contains 16 completely different patterns. Each shape in the test set, like in the training set is represented using the seven significance parameters obtainable from the Perceptual Network. The following tables represent the outputs generated when the trained net is presented with the test shapes. The four outputs represent how close the net places the pattern to each of the four basic shapes. Note that the shapes are not drawn to scale. The comments under each set of outputs are the suggested interpretations of these outputs.

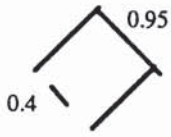
		Closeness to a : square	parallelogram	pentagon	hexagon
1		0.9834	0.0000	0.0017	0.0015
		strong indication of a square			
2		0.0972	0.0000	0.0001	0.0021
		very little suggestion of a square			
3		0.2437	0.0000	0.0000	0.0035
		either a small square or a medium square with a side missing			
4		0.0943	0.0000	0.0001	0.0023
		very little suggestion of a square			
5		0.0003	0.0267	0.0000	0.0056
		very little suggestion of a small parallelogram			
6		0.0000	0.6071	0.0099	0.0471
		strong presence of a medium to large parallelogram			
7		0.0020	0.0160	0.0000	0.0116
		ambiguous			

8



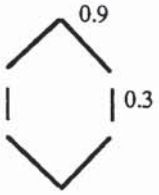
0.0000 0.5874 0.0000 0.0000
medium parallelogram or a 3 sided large parallelogram

9



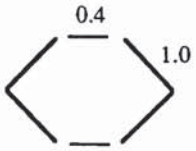
0.9626 0.0000 0.0005 0.0011
large square

10



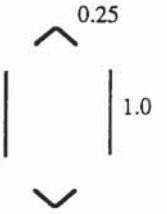
0.0155 0.0000 0.0059 0.0051
ambiguous

11



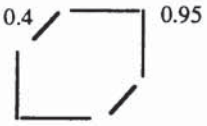
0.0000 0.0000 0.9022 0.0102
strongly pentagonal

12



0.0111 0.0000 0.0000 0.0679
slight indication of a hexagon

13

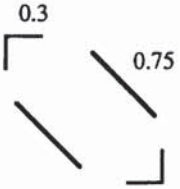


0.0000 0.0000 0.9640 0.0130
strongly pentagonal

14

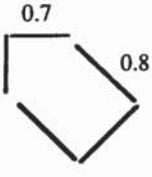
0.0727 0.0000 0.0027 00054
slight suggestion of a square

15



0.0002 0.0000 0.0002 0.3430
medium hexagonal with a side missing

16



0.0000 0.0000 0.9509 0.0013
large pentagon

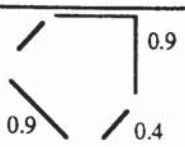
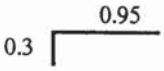
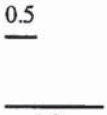
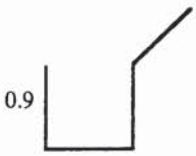
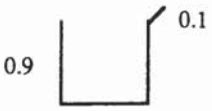
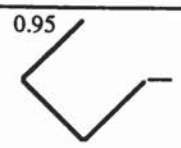
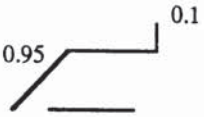
17		0.0015	0.0000	0.0767	0.0063
		slight indication of a pentagonal shape			
18		0.0091	0.0046	0.0001	0.0048
		ambiguous			
19		0.0076	0.0231	0.0000	0.0051
		slight indication of a parallelogram			
20		0.0173	0.0121	0.0000	0.0122
		ambiguous			
21		0.02217	0.0073	0.0000	0.0062
		slight indication of a square			

Table 7.1 Generalisation results when only line lengths are modified

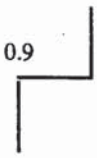
		Closeness to a :			
		square	parallelogram	pentagon	hexagon
22		0.0001	0.0000	0.7732	0.0005
		strong indication of a pentagonal			
23		0.58623	0.0000	0.0002	0.0011
		medium square or large square with a side missing			
24		0.0000	0.0000	0.9059	0.0056
		strong indication of a pentagonal			
25		0.0000	0.0000	0.0009	0.0025
		ambiguous			



26 0.7149 0.0000 0.0006 0.0029
large square with a side missing

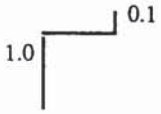


27 0.0000 0.2126 0.0000 0.0001
small parallelogram

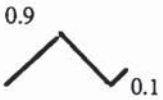


28 0.7278 0.0000 0.0002 0.0008
large square with a side missing

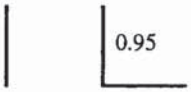
29 0.9971 0.0000 0.0007 0.0007
large square



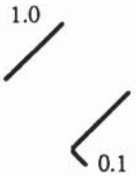
30 0.4946 0.0000 0.0005 0.0024
medium square



31 0.4271 0.0000 0.0005 0.0019
medium square



32 0.1178 0.0000 0.0003 0.0076
small square or square with a side missing



33 0.1396 0.0000 0.0001 0.0467
small square or square with a side missing

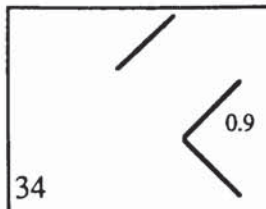
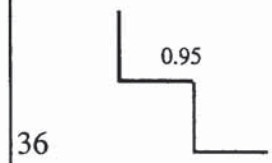
34		0.5460	0.0002	0.0000	0.0196
		medium square or large with a side missing			
35		0.0000	0.0000	0.0766	0.0048
		ambiguous			
36		0.9948	0.0000	0.0042	0.0025
		large square			
37		0.9915	0.0000	0.0011	0.0010
		large square			

Table 7.2 Generalisation results when shapes are modified

Difficulties are encountered when attempting to interpret the results. Only in a few cases are the outputs found to be clear and unambiguous. For example, patterns 1, 8 and 13 all give rise to a large activation value on just one of the outputs and so it is obvious which of the categories has been selected. Elsewhere, other outputs have much lower values making discrimination quite ambiguous. For example, in pattern 5, all of the outputs are less than the expected minimum value of 0.1 (which is the smallest desired value that appears in the training set) and thus it is not possible to confidently determine which of the classes is meant to have been selected.

Of the two sets of test data, generalisation seems to be more successful with the first set in which several patterns are correctly classified and some even appear to have activation levels that bear some relationship to the 'closeness' of match. The second test set is less successful and this time only three patterns are categorised correctly with eight grossly misclassified, eg. pattern 24 which is classified as strongly pentagonal whereas it is actually a modified square. On the whole it appears that the network is able to generalise better to changes in line length, than to changes in shape.

The practical usefulness of the network is further hindered by the inability to generalise consistently. This unpredictable nature means that similar patterns can frequently be classified quite differently. For example, consider patterns 13 and 15. Both are hexagonal, yet the outputs in each case are significantly different; 13 is classified as strongly pentagonal, and 15 as hexagonal.

Although backpropagation is capable of learning a set of patterns described in terms of significance values, generalisation capabilities seem to be limited here. This is probably due to the relatively small size of the training set. Generalisation would be expected to improve with increased size of training set. The choice of output formats is also expected to have some significant bearing on the results. Here, the output formats were not aimed at classifying just the shape, but also at being able to estimate how 'close' the match was. In order to do this, the desired outputs were not simply binary, but real values arbitrarily selected to form a simple grading mechanism. Other variations are clearly possible however, the use of such output formats is felt to be somewhat artificial and unnatural since it involves decisions made external to the system. Instead, a far more natural approach would involve an unsupervised learning algorithm which could automatically devise a suitable way of describing the outputs.

7.2 Using the Kohonen Feature Map for Recognition

The need for inventing suitable output formats for the recognition and interpretation of simple shapes can be quite a hindrance when setting up a recognition network. A self-organising learning algorithm would solve this as the need for target values would be removed. This is the attraction of using the Kohonen algorithm which can map a set of vectors onto a two-dimensional array of neurons without the need for external targets. This makes the algorithm a lot easier to use than backpropagation because now the onus of organising the outputs shifts from the network designer to the algorithm itself.

The Kohonen algorithm is capable of developing feature maps corresponding to the distribution of shapes in the input set and is capable of organising such maps in a topologically coherent manner. Thus given a set of patterns, the Kohonen algorithm would be expected to organise the output layer into a map on which similar shapes are detected by clusters of neurons close to each other. The generalisation properties of topologically consistent maps are quite attractive, since the expectations are that those patterns which are not members of the training set, should nevertheless be detected by those neurons responding to similar patterns if they existed in the training set.

To determine how close two shapes are to each other, the Kohonen algorithm calculates the euclidean distance between the vectors describing each shape. Each vector consists of seven significance values representing the constituent features of the shape.

Training a Kohonen network, like backpropagation involves a number of presentations of the whole training set of vectors. Unlike backpropagation, it is not known what each output is categorising, therefore before use the output layer must be analysed to ascertain which neurons respond to which shapes. This can be achieved by presentation of each of the training patterns one by one and noting which neurons respond. Generally, a 'bubble' of activation will be observed for each pattern and the neuron at the centre of this bubble is the neuron that shows the best match. In this way a map can be built up which shows where in the output layer, each shape is best recognised.

7.2.1 Validation of the Kohonen Simulator

Feedback in supervised learning algorithms such as backpropagation provides a natural mechanism for evaluating how well a given training set has been learned. With unsupervised learning algorithms such as the Kohonen algorithm, such feedback is not available and so the degree of learning cannot be measured so effectively. Difficulties arise in validating the Kohonen algorithm (1988) since Kohonen fails to describe in detail

some of the functions used, opting instead to refer to them as "slowly decreasing functions of time". This makes it impossible to replicate the algorithm. Secondly without any quantitative methods for evaluating performance, qualitative methods must be used. Instead, Kohonen describes how the algorithm learns some simple distributions of input data and the kind of network activity that can result. Such examples can be useful for making comparisons and for judging the performance of other implementations.

Lippmann (1987) presents an example in which a 2 input, 100 output node Kohonen net is presented with inputs representing random points uniformly distributed over a square shaped area. The resulting feature map is found to arrange itself to reflect the distribution of the input points. An orderly grid emerges in which each output neuron responds to points at a particular co-ordinate and the neighbouring neurons actually respond to neighbouring points in the input. This problem provided a suitable validation test for our system.

The experiment consisted of setting up the same 2 input / 100 output network and using a training set of 500 points taken randomly from a uniform distribution of points in a square (see Appendix K for the contents of this training set). After the training set had been presented to the net for 500 times, the results shown in the figure below were obtained :

Parameters used
Tmax (no. of iterations) : 500
Initial gain : 0.01
Excitation factor : 0.8 (this determines shape of neighbourhood function) (see section 2.7.3)

input 0 : 0
input 1 : 0

75	73	78	84	95	106	111	124	143	156
50	53	57	63	72	82	93	107	126	138
35	42	46	52	60	67	83	98	114	121
25	31	36	42	51	61	74	89	103	110
16	22	27	32	42	52	65	80	93	98
9	15	21	25	35	44	57	68	82	87
3	8	15	21	32	40	53	63	80	84
2	6	12	18	28	36	50	60	77	81
0	2	9	14	24	31	43	53	75	80
0	1	5	9	20	31	43	50	72	79

(a)

input 0 : 9
input 1 : 9

72	47	35	24	15	9	5	2	0	0
73	52	39	28	20	13	9	6	3	1
81	55	43	33	25	20	15	11	8	5
78	61	50	40	31	26	20	18	16	11
90	70	56	49	39	34	30	27	25	21
104	80	65	60	53	48	43	40	43	39
123	94	79	69	63	58	54	53	53	50
127	105	89	83	75	70	64	63	62	62
145	127	105	95	86	86	83	78	73	72
153	140	124	112	96	90	86	83	79	78

(b)

input 0 : 0
input 1 : 9

0	5	10	17	26	36	49	57	70	79
4	8	13	21	28	40	55	67	79	80
9	12	18	25	35	50	65	75	87	84
18	19	23	30	45	59	68	84	95	89
26	27	32	39	51	63	77	92	103	100
37	37	45	53	69	78	90	100	119	119
49	51	55	63	78	86	101	112	130	130
63	63	66	76	87	96	109	121	138	142
71	76	77	82	93	105	120	129	148	152
80	81	84	86	97	110	123	130	151	158

(c)

input 0 : 9
input 1 : 0

147	115	103	91	85	79	67	70	73	76
120	97	82	69	64	55	47	46	50	59
106	85	71	60	50	37	33	33	36	43
85	73	63	51	36	28	26	23	24	32
80	64	52	42	30	23	17	15	15	18
75	58	41	32	19	14	10	8	5	7
77	51	38	28	17	12	7	4	2	3
66	48	35	25	16	10	5	2	1	1
74	53	37	28	17	11	5	2	0	0
74	60	45	35	20	11	5	3	0	0

(d)

input 0 : 5
input 1 : 5

33	18	14	11	11	12	12	16	24	29
23	12	7	3	3	4	6	11	18	22
20	9	4	1	0	1	5	9	15	17
14	8	4	1	0	1	4	10	15	15
17	8	4	1	1	2	5	10	15	15
22	11	5	5	5	6	9	13	20	20
30	16	10	8	9	10	14	17	25	25
31	21	15	14	14	15	18	22	29	30
41	32	22	19	18	21	25	27	34	36
45	38	31	26	22	24	26	28	36	39

(e)

input 0 : 8
input 1 : 8

56	34	23	15	8	4	1	0	0	1
55	36	25	16	10	5	2	1	1	0
60	37	27	19	13	9	6	5	4	2
56	42	32	24	17	14	10	10	10	6
66	49	37	31	24	20	18	17	16	14
77	57	44	40	35	32	29	27	31	29
93	69	55	48	43	40	38	38	40	37
97	78	64	59	54	50	46	47	47	48
113	97	79	70	63	64	62	59	57	57
120	108	95	85	72	68	65	63	62	62

(f)

input 0 : 0
input 1 : 5

13	15	20	27	37	47	57	67	83	94
4	8	13	20	27	39	52	65	80	86
1	5	10	17	26	38	53	65	79	81
1	4	8	15	28	40	50	66	79	79
1	5	10	16	27	38	52	66	79	79
5	7	15	20	34	43	56	66	82	85
9	12	17	24	37	46	60	70	88	90
16	17	22	30	41	50	62	74	91	95
19	23	27	32	43	52	66	75	96	100
24	25	29	32	43	55	67	74	96	103

(g)

input 0 : 5
input 1 : 0

95	76	72	68	70	71	67	74	85	92
69	58	51	46	47	47	48	53	64	74
54	46	40	37	34	30	35	42	50	58
38	34	31	27	23	23	27	32	39	47
32	25	21	18	16	16	18	24	30	34
26	19	12	9	6	8	11	15	19	22
24	12	8	5	3	4	7	10	17	19
17	9	5	2	1	2	5	8	15	17
21	11	4	2	0	0	2	5	13	16
21	14	7	3	0	0	2	4	12	15

(h)

Figure 7.2 Outputs from a Kohonen net

Each grid in figure 7.2 represents the output of the 100 neuron Kohonen layer when presented with the given input and the number at each location in the grid represents how closely that neuron matches the given input pattern. The smaller the number, the closer the match. In figure 7.2a for example, neurons near the bottom left hand corner of the network are found to closely approximate the input pattern 0,0. An input of 9,9, figure 7.2b, however is matched by neurons near the top right of the network and figure 7.2e shows that neurons near the centre of the network are the closest to the input pattern 5,5. In fact all of the outputs shown in figures 7.2a to 7.2g indicate that the net has tried to arrange itself so that the location of a neuron actually reflects the input pattern it matches. For example, the neuron at location 0,0 (bottom left hand corner of the grid) best matches the input 0,0. The neuron at 9,9 (top right hand corner of the grid) best matches the input 9,9. Similarly the neuron at 0,9 matches the input 0,9 and neuron 9,0 matches the input 9,0. Thus outputs from the Kohonen net reflect the mutual distribution of inputs.

7.2.2 Properties of a Trained Feature Map

To investigate the possibilities of using a Kohonen feature map for shape recognition, a network with 7 real valued inputs, and a 12 by 12 output layer was set up. The training set consisted of 79 different shapes similar to those in figures 6.1 and 6.2 and used the seven significance values obtained from the Perceptual Network. Each shape occurs three

times, once with all long line lengths, typically 0.9 or 1.0, secondly with all medium line lengths between 0.4 and 0.6 and then finally with very short line lengths such as between 0.2 and 0.05. (See Appendix L).

To establish how the map developed after training, the members of the training set can be presented to the net individually and the ensuing activity bubble examined to locate the best matching neuron. Unlike the circular activity bubbles observed in section 7.2.1, some bubbles are found to be rather irregular in shape (eg. see figure 7.3) and because of this, the location of the best matching neuron was found using a simple iterative software procedure and not by simple inspection.

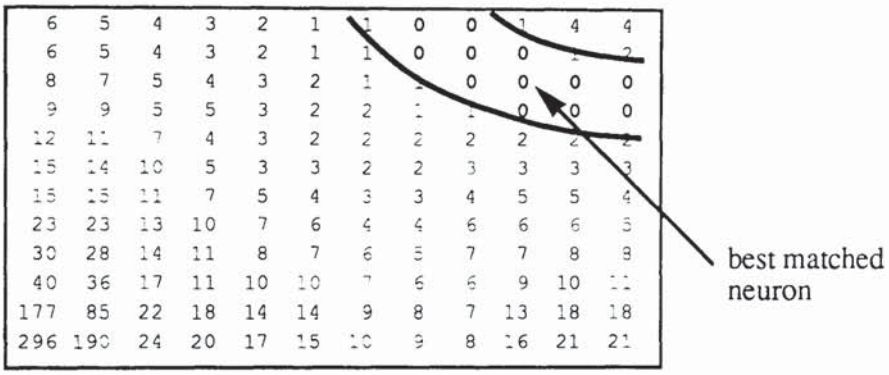


Figure 7.3 Irregular shaped activity bubbles

One possible explanation for these 'odd' shaped activity bubbles may be attributed to the fact that the algorithm is attempting to map a seven dimensional input onto a two dimensional array. In section 7.2.1 in contrast, two dimensional inputs were mapped onto a two dimensional array. This one-to-one correspondence between input and output dimensions leads to regular circular shaped activity bubbles. With greater numbers of inputs the extra dimensions cannot be incorporated ideally into a flat, 2-D array and hence irregularities and folds would be expected to arise as learning attempts to accommodate these vectors.

Figure 7.4 shows the feature map that develops. The shapes are drawn on the map using three different print styles, so that similar shapes with different sizes can be discriminated.

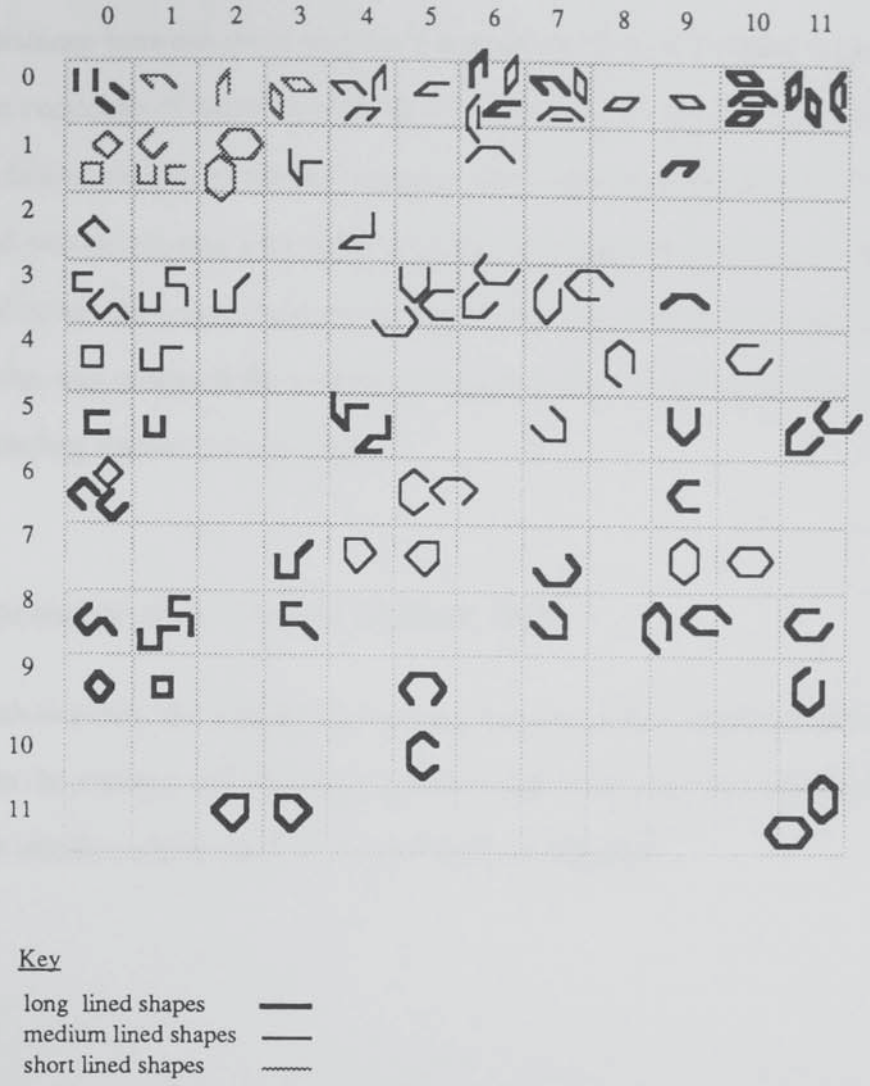


Figure 7.4 A feature map of different shapes

At a first glance, all small shapes appear clustered near the top left corner and the larger, more complex shapes such as hexagons are detected in the opposite corner, with medium sized shapes found in between. However a relationship also exists between the perceptual significance of a patterns and its position in the matrix. Thus all low significance shapes are detected near the top left hand corner and high significance shapes are detected in the bottom right hand corner.

Further examination suggests that some anomalies exist. In particular, '□' and '⊏' (medium squares and large 'C' shapes) are detected by neighbouring neurons implying that both shapes are topologically very similar. Visually this may not be the case, however in terms of the seven significance parameters used to describe these shapes, the

euclidean distance between them may be relatively small. The Kohonen algorithm relies on using the euclidean distance to evaluate how close each neuron is to a particular input pattern and this also explains another anomaly that is observed; equal sized '⊏' and '⊏' (squares and parallelograms with sides missing), are detected relatively far apart. Again this is not what would be expected, since visually these shapes are very similar. However because of the way in which the corners are represented, the euclidean distance between the corresponding vectors is quite large.

7.2.3 Performance of a Trained Feature Map

Figure 7.5 shows how the trained feature map behaves when presented with shapes not belonging to the training set. The map is shown again, but with four additional numbers marking the location where each of the test shapes is detected.

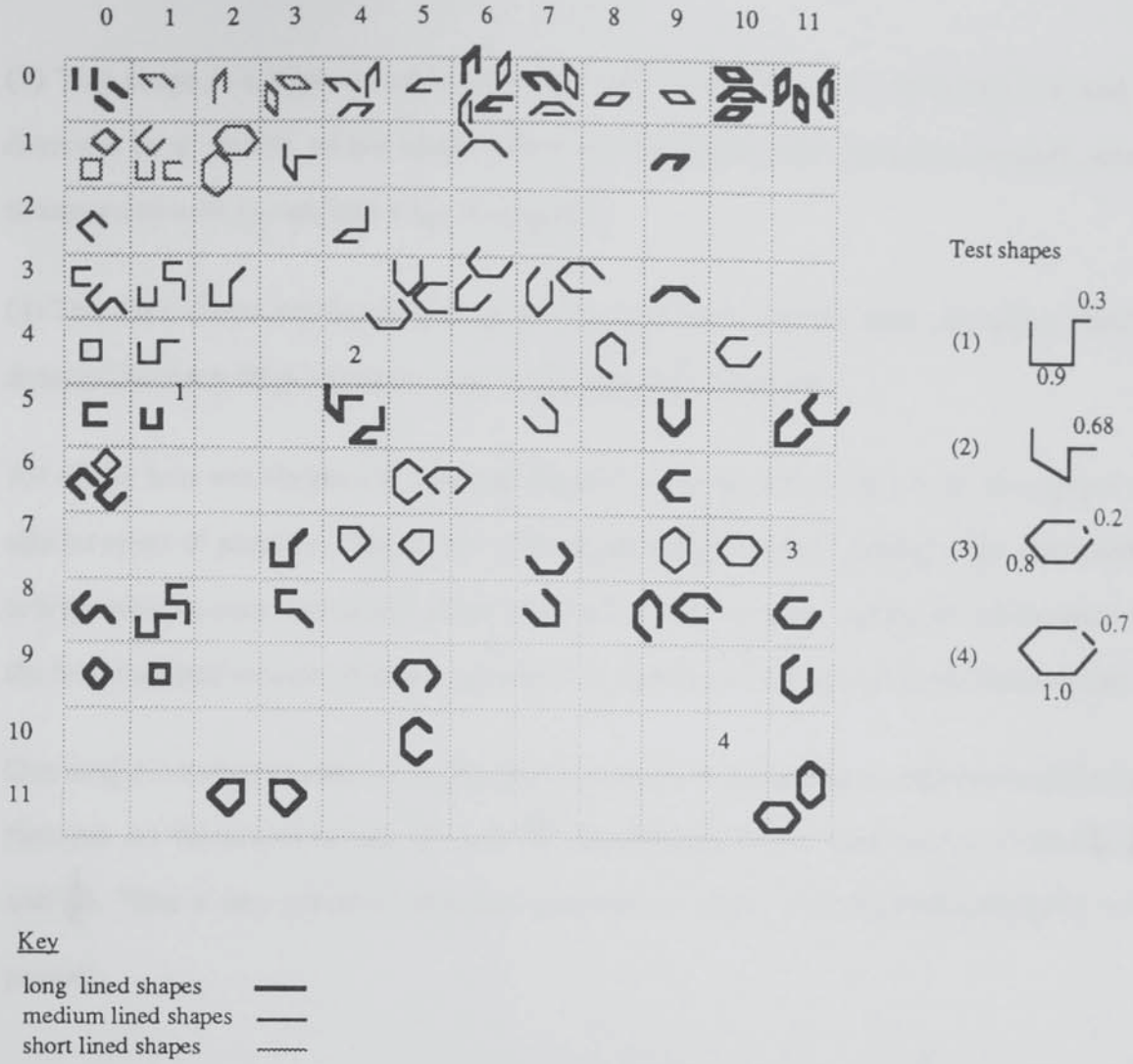


Figure 7.5 Example of generalisation ability of the shapes feature map

Considering each of the test patterns in turn :

(1) This shape is essentially a large square with a missing side. The additional line is relatively short and is meant to test the nets ability to handle distortions. The shape is detected by the same neuron that detects large open squares indicating that it is very close to this kind of shape. As expected, the small extra line (0.3 long) has little influence on final classification.

(2) This shape is similar to (1) but this time the additional line is comparably large. The extra line length is found to be influential and results in detection being distant from simple '⊏' type shapes but close to the correct '⌐' type of shape.

(3) This shape is a large hexagon with one very short (effectively missing) side and is detected by a neuron which could either be associated with complete medium sized hexagons or with incomplete large hexagons.

(4) This final shape is basically a large hexagon with one slightly short side and is ideally detected between large hexagons and incomplete large hexagons.

All of the four test shapes are detected close to neurons which are known to respond to similar types of patterns. This proves to be a successful method of recognition as not only is it possible to ascertain which shape has been been presented, but the actual location of the best matched neuron provides a qualitative measure of the *type* of shape encountered.

One major drawback however, is that some neurons show the same response to differing patterns. As mentioned before '□' and '⊏' are detected by the same neuron, as are ( ,  and ). Thus if any unknown shape is detected by these neurons, then ambiguity will prevail.

7.3 Comparison of a Kohonen Map with a Backpropagation Net

Recognition based on backpropagation has already been examined. As a way of comparison, the same training set used for backpropagation will be used to train a Kohonen map (with the desired values removed). This will be useful as it will permit a shape-for-shape comparison of the ability to generalise.

To help avoid too much clustering, as seen in figures 7.4 and 7.5, especially near the top right hand corners, the size of the Kohonen map will be increased to an array of 20 x 20 neurons.

Figure 7.6 shows the feature map that developed after 500 presentations of the training set in Appendix L.

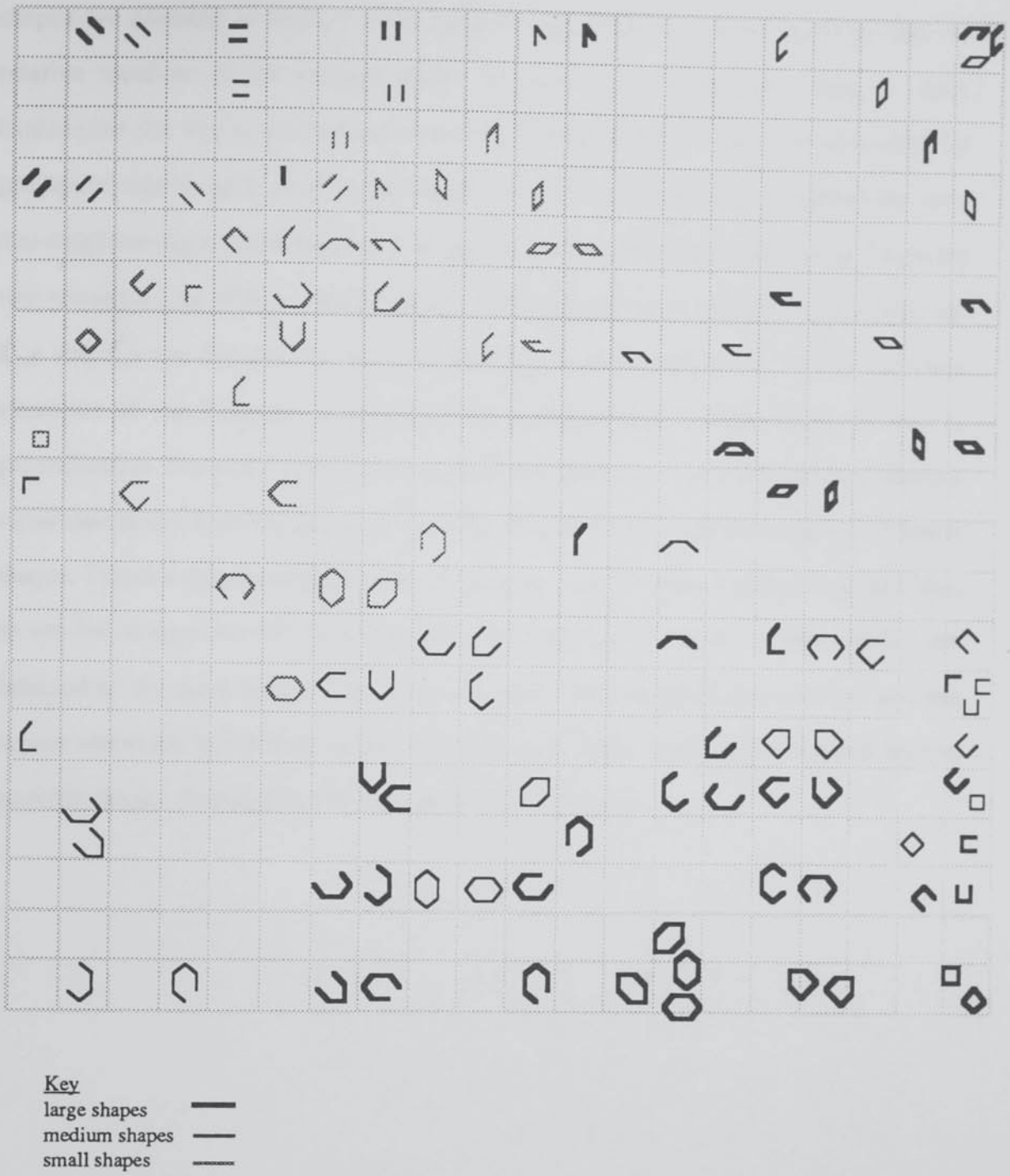
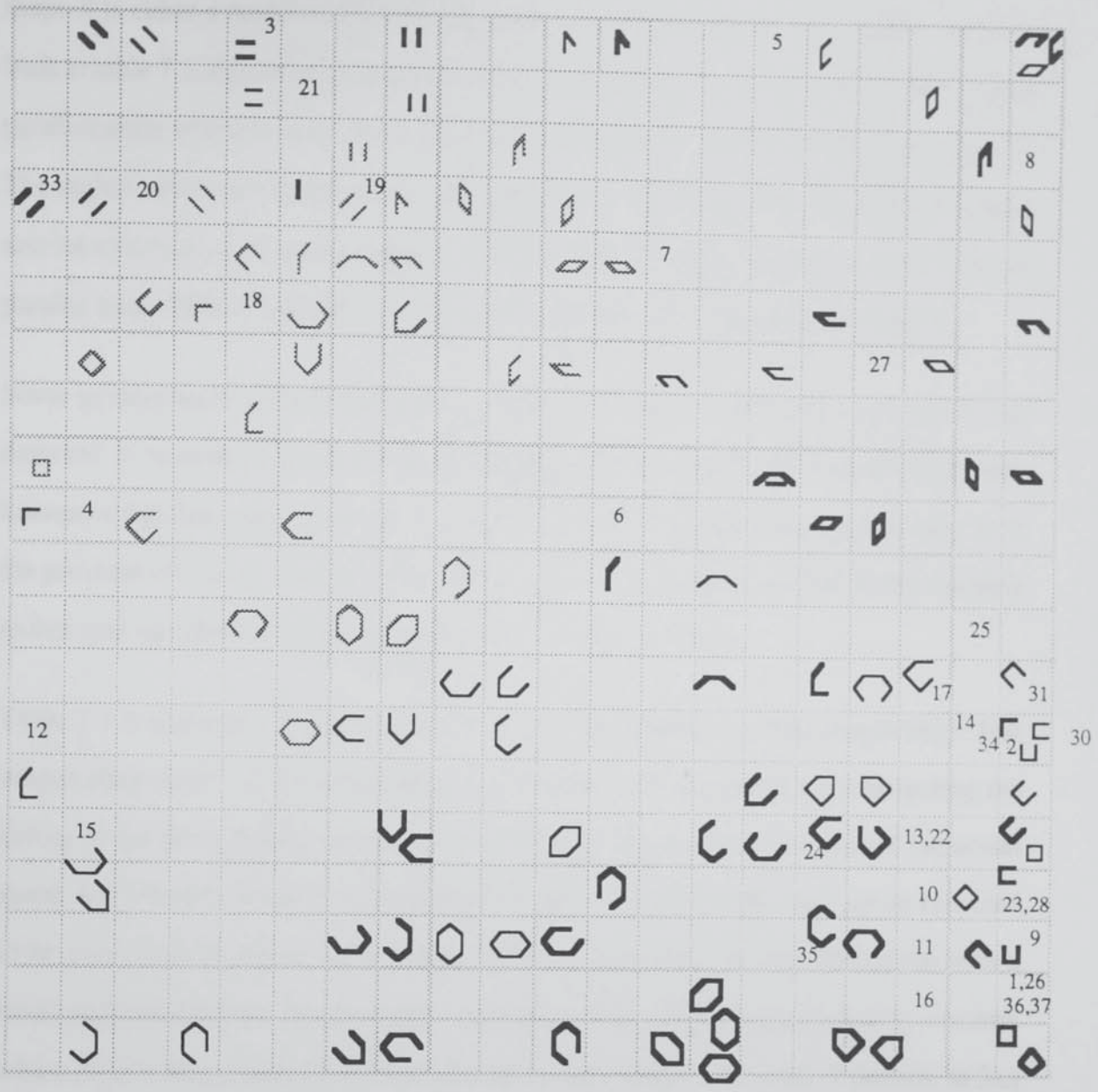


Figure 7.6 Feature map generated for training set used previously for backpropagation

It is interesting to note that several learning trials using the same training set and training patterns (except random initial starting weights), appear to develop into completely differently organised feature maps. For example, this map looks quite different from the map represented by the 12 x 12 array shown in figure 7.4 even though many of the

shapes are common to both training sets. Closer inspection shows however that the relative locations of the various shapes remains approximately the same. A likely explanation for this is that the organisation of the constituent shapes is influenced by perceptual significance. Both maps, figure 7.4 and 7.6 would seem to support the view that neighbouring neurons respond to shapes of similar perceptual significance. There are two consequences of this : firstly shapes which actually look similar to each other eg. and  are detected by neurons that are close to each other. This is the main attraction of the Kohonen map and is the reason why it is expected to do well in generalisation. Secondly, however because of differences in size, the separation between significances and hence locations on the map, may be small even for completely different shapes. This is a disadvantage because if a neuron does not show a unique response, then its use for recognition will be ambiguous. For example, in figure 7.6, ( and □) are detected by the same neuron. Thus if an unknown shape is presented to the net and this neuron shows the best match, then it will not be possible to discriminate between the two possible shapes. Fortunately, this occurs in only few cases.



Key
large shapes
medium shapes
small shapes

Figure 7.7 Generalisation with test patterns from Tables 7.1 and 7.2

Figure 7.7 shows how the feature map handles generalisation. The numbers marked on the map refer to the test shapes listed in tables 7.1 and 7.2 and the location of the number can be used to find out what pattern has been input. Although it is not always possible to say accurately what shape has been detected, it is certainly possible to suggest or hypothesize what *kind* of shape may have been presented. For example, shape 8 appears amidst a group of large '  ' and '  ' type shapes. This would seem to indicate that

shape 8 is either a medium sized parallelogram, or a large open parallelogram. Looking back at table 7.1, this analysis is found to be quite accurate as the shape is actually a large parallelogram with one very short, effectively missing side. Another example is shape 19. The region where this shape is detected contains mostly small, low significance shapes and its exact location would suggest something like a small corner or a pair of short parallel lines. Table 7.1 shows that shape 19 is actually a low significance corner.

Some generalisations can be misleading at times. For example, shapes 10, 11 and 13 are detected in between an area of square shapes on the right and an area of large open hexagons and large open pentagonal shapes on the left. Recognition is not as easy as in the previous two cases because the locations make interpretation ambiguous, for example in this case the shape could be hexagonal, pentagonal or a square.

Table 7.1 is made up of shapes which test for generalisations in line length only. Test shapes from table 7.2 are completely new patterns and are aimed at investigating the ability of the net to handle generalisations not only of line lengths but also of the actual shape. For example, shape 27 is detected between a '⊐' and a '⊏' and can be expected to be very close to one of these shapes. In fact it is a large open parallelogram with a small additional corner. Another good example is shape 33. This is detected by a neuron which detects long parallel lines and so must be very close to this kind of pattern. In fact it is a pair of long parallel lines, again with an additional small corner. In both cases, the net has performed well to minor distortions. Major distortions are somewhat more influential and can also make interpretation difficult. For example, shape 34 is the same as 33 but with a large, not small additional corner and is detected next to the neuron associated with '⊔' and a '⌜'. However this output is not enough to deduce its real shape. The same is true of other test shapes detected in this region, eg. 29, 36 and 37 which would presumably be deduced to be some close variations on large squares, but are actually 'zigzag' shapes.

Just as with backpropagation, generalisation is better for test shapes that show changes in

line length only and both methods are relatively poor at handling major distortions to shape. Interestingly, patterns such as 29,36 and 37 are recognised as being close to large squares by both algorithms, suggesting that this is more likely to be an artefact of the training set or the representation used rather than the learning algorithms applied. The reason for this turns out to be an inability of the representation to show a strong distinction between squares and 'zigzags' because it cannot clearly represent closure.

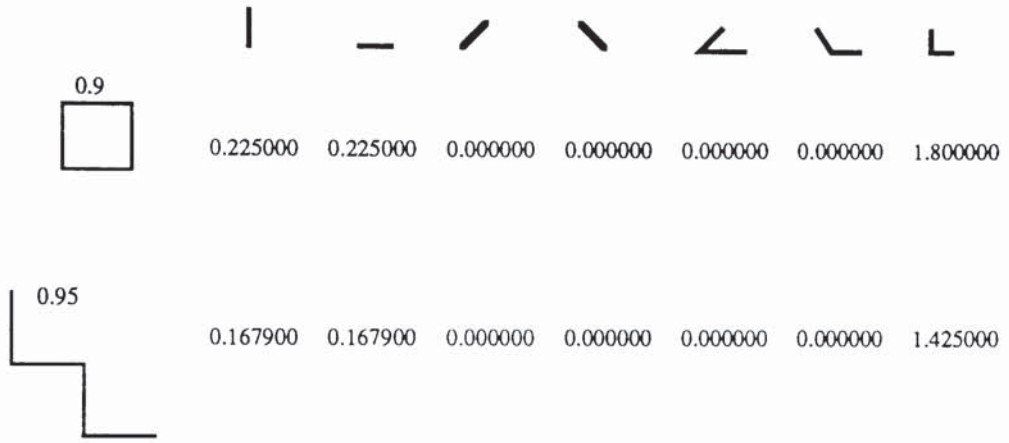


Figure 7.8 Significance values for a square and a zigzag

Figure 7.8 shows these two shapes along with their corresponding significance values and it can be seen that the same parameters are active for both shapes, but at slightly different levels. In fact the significance values for the zigzag may equally well represent a medium to large square.

7.3.1 Training Advantages over Backpropagation

Both the backpropagation and Kohonen methods can be used for recognition of trained shapes and it seems that the Kohonen algorithm is not significantly better at generalisation than backpropagation. In fact both methods are poor at handling completely new shapes.

Yet it is felt that the Kohonen algorithm is far better suited to this kind of recognition task than backpropagation. The most important advantage can be attributed to the fact that

learning can take place without the need for any external teacher and that the outputs are organised automatically. A important consequence of this, is that the location of detection provides a qualitative measure of the type of shape present.

Speed of learning is always an issue when backpropagation is used and can often restrict the size of training set used or the complexity of the problem being tackled. The Kohonen algorithm has no hidden layers and requires only one forward pass through the net during learning making it significantly faster at learning than backpropagation. This can be quite advantageous when large training sets are involved.

Post-training analysis of the Kohonen net is possible, which is in contrast to the backpropagation method. Although the weights for an individual neuron can be retrieved, it is usually not possible to discover what function each part of the net is performing. An in depth study of the multi-layer perceptron revealed that although it was possible to establish whether a patterns excited or inhibited a particular neuron, it was not possible to say how much influence that neuron would have on the final output of the net. Since the same pattern also affects the other neurons to varying degrees. This is a characteristic feature of distributed representations.

The function of each neuron in the Kohonen net is to match a particular input pattern. Since the weights actually represent actual descriptions of various input patterns, simple inspection is all that is necessary to establish what type of pattern the neuron will respond best to.

Chapter 8

Conclusions

The work presented here has shown how it is possible to construct a network that will perform simple perceptual grouping functions on straight lines. The network developed, referred to as the Perceptual Network takes as an input a representation of a simple shape and outputs a value relating to its perceptual significance. The perceptual grouping functions that are implemented by the net involve locating instances of parallel lines and end connected lines (corners).

Essentially there have been two facets to the approach used; (i) devising a coding scheme suitable for use as inputs to a neural network and (ii) using a novel "divide and conquer" strategy to split up the grouping operations into simpler, smaller tasks.

A coarse coding scheme was devised in which each line was represented by a single line length value. The position of these values on an orientation plane approximately indicates the lines' actual location and their orientation determines which plane they appear in.

8.1 The Perceptual Net and Backpropagation

Backpropagation was found to be very flexible and was used to train all of the sub-networks used in the Perceptual Network. Furthermore, it was found relatively easy to 'customise' the algorithm to suit specific needs, for example, by replacing the sigmoid activation function with a hard limiter and artificially 'propping' up error values to ensure they never reach zero, backpropagation was made to generate output values of exactly 1.0 or 0.0; something not possible with the asymptotic sigmoid function.

The operations implemented by the networks fall roughly into two categories: filters and significance evaluators. The perceptual significance is always calculated for pairs of lines and the task of the filters is to pass up the appropriate pair of values from the input code planes. Once isolated, networks trained on a simple relationship based on line lengths,

generate the appropriate perceptual significance values.

Once the individual networks had been completed, bringing them together was straightforward. This involved generating and 'wiring' together instances of each required network into what was effectively a large tree structure known as the Perceptual Network. No backward training pass was necessary and the forward traversal to calculate the outputs was achieved using a simple recursive procedure. The Perceptual Network can be arranged to output a single perceptual significance value for the whole shape, or significance values for each feature detected in the shape. Altogether, seven different features were used, parallel lines in four orientations and three different types of end connected lines (acute, obtuse and right angled corners). When summed together these parameters represent the total significance of the shape.

The completed Perceptual Network was rigorously tested for correctness and the ability to generalise. In these tests the network performed very well, with many outputs near to expected values. Tests for generalisation showed the network to hold up better to changes in line length rather than changes in actual shape (eg. changes in the number of corners or parallel lines). The final test involved subjecting the network to simple shapes; shapes that could be regarded as useful perceptual groups. Again in most cases the network behaved satisfactorily, with outputs on average correct to 2 decimal places of the expected values. The worst case observed showed an error just in excess of 1 decimal place. The Perceptual Net was found to cope with translation invariance very well and theoretically should also be able to handle rotational invariance since perceptual significances are invariant to both translation and rotation. However, identical significance values are not output for identical shapes at different orientations. The reason for this is due to the coding scheme in which the distance between the diagonal grid locations is greater than adjacent locations.

The Perceptual Net can be arranged to output seven significance values. Since these parameters describe the shape or pattern in terms of the perceptually most significant

features in the pattern, it should be possible to perform recognition of the shapes in terms of these parameters. This is possible using backpropagation, but unlike in all other uses of the algorithm, devising a useful set of outputs is not a trivial problem. The onus is on the network designer to create an efficient output coding scheme, especially when there are many different shapes in the training set.

By exploiting the ability of neural networks to generalise, it was often possible to confidently use small training sets whilst at the same time learn a large range of patterns. However this was not always the case. For example with the corner-detector, it was found necessary to use the full training set, that is, all the exemplars that the network was ever required to handle. This was one reason why the coding scheme was limited to code planes 4 x 4 in size. With larger sizes the full training sets generated would have been much larger and a lot slower to train, which would have been somewhat of a hindrance.

The reason for this was found to be related to the dimensionality (the number of inputs) and whether the training set employed was representative of the universal set of all possible inputs or a subset of this universe. In the case of say the max-filter, the training set included exemplars which covered the whole range of possible inputs at regular intervals. Thus it represented the entire domain and would be expected to generalise well. The corner-detector used a 16 dimensional, binary valued input. The universal set defined for this domain contains 2^{16} (65536) different patterns. The training set was however limited to exemplars with at most 2 binary ones and 14 binary zeros, and in total contained only 142 exemplars. This effectively formed a subset within the universal set.

An examination of the weights after training provided more evidence supporting this view. Whereas the max-filter was found to reach effectively the same set of weights for each identical learning trial and gave identical performance in each case for all patterns, the corner-detector reached a different set of weights leading to identical performance for trained exemplars, but different for untrained patterns. A satisfactory explanation of this can be offered in terms of weight space. If as is the case with the corner-detector,

different points in space (minima) can adequately learn the same training set, then performance for the corner-detector will be as observed, only identical for trained exemplars. To achieve true generalisation the training set should not be ambiguous; there should be only one global minima in weights space that can satisfactorily represent the solution. If this is so, all trials will reach the same single solution, resulting in the same performance in each case.

Accuracy of trained networks became a major issue when building a large scale tiered network, as the opportunity for errors compounding presented a significant problem. Thus the ability to estimate the expected accuracy of a network was considered an important facility. To meet this need a simple relationship was derived which could be used to determine the error in the output of a trained exemplar, given the number of exemplars in the training set and the final global error to which the network is to be trained. Thus, if the size of the training set is known, then the global error (which was used as a stopping criteria in the simulation) can be estimated for any required accuracy.

A criticism often made against backpropagation is its slow learning speed. Such criticisms are not without foundation since training times can often lead into many hours or even days, even when using fast, modern workstations. The motivation for attempting to speed up algorithms is quite strong and many researchers have taken up the challenge of trying to accelerate learning. Although originally not the purpose for adopting this modification, Fahlman's suggestion of adding a small value such as 0.1 to the 'sigmoid-prime' function was found to significantly improve learning time while at the same time being very simple to implement.

8.2 The Perceptual Net and the Kohonen Feature Map.

As an example of the practical use of the Perceptual Network, this thesis demonstrates how the outputs may be used to train 'higher level' nets to perform shape recognition, based on the parameters delivered. The motivation for favouring a self-organising

learning algorithm has already been touched upon in the previous section and that is that when the outputs of an MLP have to be used to represent many differing categories, it is not always a trivial step to devise a simple and efficient coding format. This is the case when training an MLP for shape recognition using the outputs of the Perceptual Network.

A self-organising algorithm can learn without the aid of external targets and thus shifts the need of developing an output coding format from the net programmer, to the net itself. The Kohonen algorithm in particular is seen as the most appropriate method as it is capable of generating spatially-coherent feature maps whose topology is believed to be not unlike that found in biological neural systems. The results in this thesis show how shapes represented using seven parameters can be mapped onto a two dimensional feature map of neurons via the Kohonen algorithm. The resulting topology of the map is such that shapes that are similar are detected by neurons that are in close proximity to one another. It is this property that allows the net to be used for the identification and categorisation of different shapes.

8.3 Comparison of the Backpropagation and Kohonen Algorithms

In tests, comparison between a trained MLP and a trained feature map, did not show any significant difference in performance. Both were capable of handling generalisation when only lengths of lines in the shapes were altered, and both performed equally well when asked to generalise to new shapes. Although in the latter case problems were encountered, especially with respect to ambiguities arising due to the absence of any explicit coding of closure in the inputs. However because no effort was required to devise suitable output coding formats, the ease of training experienced with the Kohonen algorithm, made it the preferred alternative over backpropagation in this case.

The Kohonen algorithm also has several other, particularly speed, advantages over backpropagation. Backpropagation relies on gradient descent to slowly converge to a possible solution and is thus slow and computationally intensive, especially when hidden

layers are involved. The practical implementation of the Kohonen algorithm is based on measuring the closeness of match between the input feature and the preferred response of individual neurons. Furthermore, no hidden layers or error propagation is involved. This means that in general, training with the Kohonen algorithm is substantially faster than with backpropagation.

The mechanisms by which a Kohonen feature map and an MLP perform generalisation are quite different. It should be understood that the Kohonen algorithm actually has no intrinsic ability to perform generalisation. It cannot discover the underlying trend in a given set of data. Instead, it is its ability to arrange input vectors in a topologically consistent manner that makes the resulting feature maps useful. Particularly in problems dealing with arrangement of patterns or vectors in some logical and consistent manner. Backpropagation, on the other hand, relies on the ability of hidden units to learn to adequately represent the important features in the input data, thus enabling the network to perform categorisation and generalisation in many different situations.

8.4 Future Work

The Perceptual Network as it stands, serves to illustrate how perceptual organisation can be implemented in terms of neural networks and the range of problems presented in this work are considered to represent as yet one of the most comprehensive studies in the area of neural networks and perceptual organisation. However from a practical viewpoint, the network cannot be applied to real line images, only to appropriately coded representations and needs to be 'scaled-up' if it is to be used to cope with real world problems.

There are several aspects, mostly associated with the coding mechanism, that could be 'scaled-up'. In particular, the resolution could be improved so that the position of lines and orientation of lines could be recovered more accurately. For example, instead of using a 4 x 4 code plane, larger sizes, eg. 8 x 8, 20 x 20 or even 100 x 100 could be set-up. This would increase the spatial resolution; to increase the number of allowable

orientations, the number of orientation planes could be increased to any desired amount. However, the increased number of inputs would become difficult and unmanageable for any significant increase. Some of the sub-tasks trained on inputs 4 x 4 in size would take substantially longer to train, even for an increase to 5 x 5 or 6 x 6. To overcome this, and make efficient use of the greater number of inputs, a hierarchical approach could be adopted based on over-lapping receptive fields and multi-level representations. An example of such an approach is the Hierarchical Structure Code suggested by Hartmann (1987) (described in section 3.3.5), which it is claimed can efficiently represent images at varying resolutions.

Processing a complex hierarchically coded image presents a problem in itself, and it is not at all clear what the most effective method for handling data at different resolutions is. However, it is felt that such a structure is especially suited for use with neural networks; a view supported by Hartmann.

Another area for improvement is related to the restriction which permits only two lines to be coded per orientation plane and only two lines per receptive field. Whereas the resolution of the planes could be increased by increasing their size and the resolution of the allowable orientations by increasing the number of orientation planes. To modify the scheme to process more than one pair of lines at a time is a little more subtle. It appears that some form of sequential processing should be introduced to allow inspection if several groups are detected in a single plane, a view supported by Feldman (1988). In terms of the human visual system, sequential processing appears as a focus of attention mechanism. This would be an attractive method to adopt, but as yet such a mechanism is not fully understood and how a sequential mechanism could be embodied into a neural architecture is not clear.

Recent work has attempted to address this issue and increasingly the feeling is that serial processing is necessary at some point in the recognition process. Ullman (1984) for example, suggests that high level processes may be implemented as a sequence of 'visual

routines'. A good account of the mechanisms that may be involved is presented in Ahmad and Omohundro (1990) who also describe how neural networks could be devised to achieve selective attention. It appears that by using focus of attention mechanisms, it also becomes possible to process images in their pre-processed pixel state, without the need for employing restrictive coding methods.

In general, implementation of sequential processes is seen as the most promising way of endowing neural networks with high level recognition capabilities and one popular theory proposed, suggests that a means of achieving this lies in the integration of neural nets with traditional AI approaches such as rule based systems. Whatever solution maybe found, the ability to detect or perceive strong visual cues such as perceptual groups will provide an essential foundation for higher level recognition.

References

Ackley D.H., Hinton G.E. and Sejnowski T.J., 1985, "A Learning Algorithm for Boltzmann Machines.", *Cognitive Science*, vol. **9**(1), pp. 147-169

Ahmad S., 1988, "A Study of Scaling and Generalization in Neural Networks.", Technical Report No. UIUCDCS-R-88-1454, Department of Computer Science, University of Illinois at Urbana-Champaign.

Ahmad S. and Omohundro S., 1990, "A Network for Extracting the Locations of Point Clusters Using Selective Attention.", International Computer Science Institute, Berkeley, Technical Report #90-011.

Ballard D.H., 1981, "Generalizing the Hough Transform to Detect Arbitrary Shapes.", *Pattern Recognition*, vol. **13**(2), pp. 111-122

Barrow H. G. and Tenenbaum J. M., 1981, "Interpreting Line Drawings as 3-D Surfaces.", *Artificial Intelligence*, vol. **17**, pp. 75-116

Besl P. J. and Jain R. C., 1985, "Three Dimensional Object Recognition.", *ACM Computing Surveys.*, vol. **17**(1), pages 76-145

Bolles R.C. and Hourad P., 1987, "3DPO: A Three Dimensional Part Orientation System.", in Three Dimensional Machine Vision. (ed. Takeo Kanade), Kluwer Academic Publisher, pp. 399-450

Brady M. 1982, "Computational Approaches to Image Understanding." vol. **14**(1), *ACM.*, pp. 3-71

Brooks R. A., 1981, "Symbolic Reasoning among 3-D Models and 2-D Images.", *Artificial Intelligence.*, vol. **17**, pp. 285-348

Brooks R. A., 1983, "Model-Based 3-D Interpretations of 2-D Images.", *IEEE Transactions on Pattern Analysis and Machine Intelligence.*, **PAMI-5**(2), pp. 140-150

Burns J.B. and Kitchens L.J., 1987, "Recognition in 2D Images of 3D Objects from Large Model Bases using Prediction Hierarchies.", *Proceedings of the Tenth IJCAI.*, vol. **2**, Milan, Italy, pp. 763-766

Burr D.J., 1988, "Experiments on Neural Net Recognition of Spoken and Written Texts." , *IEEE Transactions on Acoustics, Speech and Signal Processing.*, vol. **36**(7), pp. 1162-1168

Canny J., 1986, "A Computational Approach to Edge Detection.", *IEEE Transactions on Pattern Analysis and Machine Intelligence.*, **PAMI-8**(6), pp. 679-698

Carpenter G.A. and Grossberg S., 1987, "A Massively Parallel Architecture for a Self-Organising Neural Pattern Recognition Machine.", *Computer Vision, Graphics and Image Processing.*, vol. **37**(Jan), pp. 54-115

Carpenter G.A. and Grossberg S.,1988, "The ART of Adaptive Pattern Recognition by a Self Organising Neural Network.", *Computer*, vol. **21**(3), pp. 77-88

Chin R. T. and Dyer C. R., 1986, "Model Based Recognition in Robot Vision.", *ACM Computing Surveys.*, vol. **18**(1), pp. 67-110

Connell J.H and Brady M., 1987, "Generating and Generalizing Models of Visual Objects.", *Artificial Intelligence.*, vol. **31**, pp. 159-183

Dobbins A., Zucker S. and Cynader M.S., 1987, "Endstopped Neurons in the Visual Cortex as a Substrate for Calculating Curvature.", *Nature*, vol. **329**, pp. 438-441

Fahlman S. E. and Hinton G. E., 1987, "Connectionist Architectures for Artificial Intelligence.", *Computer.*,vol. **20**(1), pp. 100-109

Fahlman S., 1988, "Faster Learning Variations on Back-Propagation : An Empirical Study.", *Proceedings of the 1988 Connectionist Models Summer School*, Carnegie Mellon University, pp. 38-51

Feldman J.A., 1985, "Connectionist Models and Parallelism in High Level Vision.", *Computer Vision, Graphics and Image Processing.*, vol. **31**(2), pp. 178-200

Feldman J.A., Fanty M.and Goddard N.H., 1988, "Computing with Structured Neural Networks.", *Computer*, vol. **21**(3), pp. 91-103

Feldman J.A. and Ballard D.H., 1983, "Computing with Connections.", in Human and Machine Vision. (eds. Beck, Hope & Rosenfeld), Academic Press, pp. 107-155

Frisby J.P., 1979, "Seeing: Illusion, Brain and Mind.", Oxford University Press

Fukushima K., 1988, "A Neural Network for Visual Pattern Recognition." *Computer*, vol. **21**(3), pp. 65-75

Fukushima K., 1975, "Cognitron: A Self-Organising Multilayered Neural Network." *Biological Cybernetics*, vol. **20**(3/4), pp. 121-136

Green A.D.P. and Noakes P.D., 1989, "Linked Assembly of Neural Networks to Solve the Interconnection Problem.", *Proceedings of the First IEE International Conference on Artificial Neural Networks*. pp. 216-220

Grimson W. E. L. and Lozano-Perez T., 1984, "Model Based Recognition and Localization from Sparse Range or Tactile Data.", *The International Journal of Robotics Research*, vol.**3**(3), pp. 382-414

Hanson A. and Riseman E., 1988, "The VISIONS Image Understanding System.", in Advances in Computer Vision. (ed. C. Brown), pp. 1-114

Hartmann G., 1985, "Hierarchical Contour Coding by the Visual Cortex.", in Models of the Visual Cortex. (eds. D. Rose & V.G. Dobson), John Wiley and Son Ltd., pp. 137-145

Hartmann G., 1987, "Recognition of Hierarchically Encoded Images by Technical and Biological Systems.", *Biological Cybernetics*., vol. **57**(1/2), pp. 73-84

Hebb D.O., 1949, "The Organization of Behaviour.", John Wiley.

Hopfield J.J. and Tank D.W., 1986, "Computing with Neural Circuits : A Model.", *Science*, vol. **Aug**, pp. 625-633

Hopfield J.J., 1982, "Neural Networks and Physical Systems with Emergent Collective Computational Abilities.", *Proceedings of the National Academy of Sciences*, vol. **79**, pp. 2554-2558

Hoskins J.C., 1989, "Speeding up Artificial Neural Networks in the 'Real' World." MCC Technical Report Number STP-049-89

Hubel D.H. and Wiesel T.N., 1962, "Receptive Fields, Binocular Interactions and Functional Architecture in the Cat's Visual Cortex.", *Journal of Physiology*, vol. **160**, pp. 106-154

Hubel D.H. and Wiesel T.N., 1959, "Receptive Fields of Single Neurons in the Cat's Striate Cortex.", *Journal of Physiology*, vol. **149**, pp. 574-591

Huekel M. H., 1973, "A Local Visual Operator which Recognises Edges and Lines.", *Journal of the ACM*, vol. **20**(4), pp. 634-647

Hummel J., Biederman I., Gerhardstein P. & Hilton H., 1989, "From Image Edges to Geons : A Connectionist Approach.", *Proceedings of the 1988 Connectionist Models Summer School*, Carnegie Mellon University, pp. 462-471

Kohonen T., 1988, "Self-Organizing Feature Maps." in Self-Organisation and Associative Memory, 2nd edition. Springer-Verlag, pp. 119-157.

Kohonen T., 1988a, "The 'Neural' Phonetic Typewriter.", *Computer*, vol. **21**(3), pp. 11-22

Kohonen T., 1987, "Adaptive, Associative and Self Organising functions on Neural Computing.", *Applied Optics*, vol. **26**(23), pp. 4910-4918

Kohonen T., 1982, "Self-Organised Formation of Topologically Correct Feature Maps.", *Biological Cybernetics*, vol **43**, pp. 59-69

Levine M.D. and Nazif A.M., 1985, "Rule-Based Image Segmentation: A Dynamic Control Strategy Approach.", *Computer Vision, Graphics and Image Processing.*, vol. **32**, pp. 104-126

Linsker R., 1988, "Self Organisation in a Perceptual Network." , *Computer*, vol. **21**(3), pp. 105-117

Lippmann R.P., 1987, "An Introduction to Computing with Neural Nets.", *IEEE Acoustics, Speech & Signal Processing Magazine*, vol. **4**(2), pp. 4-22

Lowe D. G., 1985, "Perceptual Organisation and Visual Recognition.", Kluwer Academic Publishers.

Lowe D. G., 1987, "Three Dimensional Object Recognition from Single Two Dimensional Images.", *Artificial Intelligence.*, vol. **31**, pp. 355-395

Marr D.C., 1980, "Visual Information Processing: The Structure and Creation of Visual Representations.", *Phil. Trans. R. Soc. Lond,* vol. **290** (series B), pp. 199-218

Marr D.C. and Hildreth E., 1980, "Theory of Edge Detection.", *Proceedings Royal Society London*, vol. **207**(Series B), pp. 187-217

Marr D.C. and Nishihara H.K., 1978, "Representation and Recognition of the Spatial Organisation of Three-Dimensional Shapes.", *Proceedings of the Royal Society of London*, vol. **200**(Series B), pp. 269-294

McCulloch W.S. and Pitts W., 1943, "A Logical Calculus of the Ideas Imminent in Nervous Activity.", Bulletin of Mathematical Biophysics, vol. **5**, pp. 115-133

Minsky M. and Papert S., 1969, "Perceptrons: An Introduction to Computational Geometry.", The MIT Press.

Palmer S., 1983, "The Psychology of Perceptual Organisation: A Transforational Approach ." in Human and Machine Vision (eds. Beck, Hope and Rosenfeld). Academic Press, pp. 269-339

Pawlicki T., 1988, "NORA : Neural-network Object Recognition Architecture.", *Proceedings of the 1988 Connectionist Models Summer School*, Carnegie Mellon University, pp. 444-451

Pomerleau D.A., 1989, "ALVINN: An Autonomous Land Vehicle in a Neural Network.", Tech Report: CMU-CS-89-107 (Computer Science Dept. Carnegie Mellon.)

Qian N. and Sejnowski T., 1988, "Learning to Solve Random dot Stereograms of Dense and Transparent Surfaces with Recurrent Backpropagation.", *Proceedings of the 1988 Connectionist Models Summer School*, Carnegie Mellon University, pp. 434-443

Rosenblatt F., 1962, "Principles of Neurodynamics.", Spartan Books

Roth I., 1986, "Part II. An Introduction to Object Perception." in Perception and Representation. A Cognitive Approach. (eds. Roth I. & Frisby J.P.) The Open University Press. pp. 79-131

Rumelhart D.E. and Zipser D., 1985, "Feature Discovery by Competitive Learning.", *Cognitive Science*, vol. **9**, pp. 75-112

Rumelhart D.E., Hinton G.E., and Williams R.J., 1986, "Learning Internal Representations by Error Propagation.", in Parallel Distributed Processing. Exploration in the Microstructure of Cognition. (eds. Rumelhart D.E. & McClelland J.L.), vol. **1**, The MIT Press, pp. 319-362

Scalettar R. and Zee A., 1988, "Perception of Left and Right by a Feed Forward Net.", *Biological Cybernetics*., vol. **58**(3), pp. 193-201

Sejnowski T. and Rosenberg C.R., 1986, "NETtalk: A Parallel Network that Learns to Read Aloud.", Technical Report JHU/EECS-86/01, John Hopkins University

Singh S. and Claridge E., 1989, "A Connectionist Approach to Three Dimensional Perception.", *Proceedings of the Sixth Scandinavian Conference on Image Analysis*, vol. **1**, Oulu, Finland, pp. 341-348.

Thomson R.C, and Claridge E., 1989, "A 'Computer Vision' Approach to the Analysis of Crystal Profiles in Rock Sections.", *Proceedings of the Sixth Scandinavian Conference on Image Analysis*, vol. **2**, Oulu, Finland, pp. 1208-1215.

Ullman S., 1984, "Visual Routines: Where Bottom-up and Top-down Processing Meets.", *Cognition*, **18**, pp. 97-159

Walter D.K.W., 1987, "Selection of Image Primitives for General-Purpose Visual Processing.", *Computer Vision, Graphics and Image Processing*, vol.**37** pp. 261-298

Walters D.K.W., 1986, "A Computer Model Based on Psychophysical Experiments.", in Pattern Recognition by Humans and Machines. (ed. Schwab and Nusbaum.), vol. **2**, Academic Press, pp. 87-120

Walters, D.K.W. and Weisstein, N., 1982, "Perceived Brightness is a Function of Line Length and Perceived Connectivity.", *Bulletin of the Psychonomic Society*, **Sept**

Zemel R., Mozer M. and Hinton G., 1988, "TRAFFIC: A MOdel of Object Recognition Based on Transformations of Feature Instances." *Proceedings of the 1988 Connectionist Models Summer School*, Carnegie Mellon University, pp. 452-461

Zucker S.W., 1988, "A Biologically Motivated Approach to Early Visual Computations: Orientation selection, texture and optical flow.", *Proceedings of the 4th International Conference of the BPRA.*, Cambridge, UK, pp. 417-428

Appendix A

A Step by Step Guide to Using the Backpropagation Algorithm

Step 1.

Devise a network for the problem. The number of input and output units will depend on the problem itself. But the number of hidden layers, and the number of units in a hidden layer is a variable quantity. Some experimentation may be necessary to determine the optimum number of hidden units.

Set all weights and biases to small random values, say between 0 and 1. This is necessary since if training commences with all weights initialised to equal values and if the solution requires that the weights be unequal, then because the error propagated back at each stage is in proportion to the weights, it is apparent that the error propagated back will also be the same for each unit. Consequently the net will not be able find a local minima, as the weights will be the same all the time.

Step 2

Present an exemplar to the net. This will consist of an input pattern, and the desired output pattern associated with this input. The patterns may be binary 1's or 0's or real values.

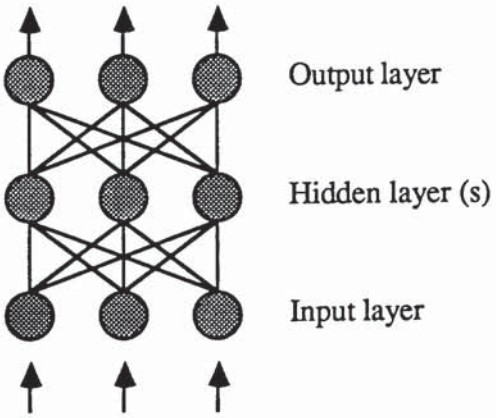


Figure A.1

Step 3

Determine the actual outputs of the net by performing a weighted sum at the hidden layer, and then at the next layer, and so on until the output layer is reached. Use the sigmoid activation function, equation (A.1) to threshold the output of each unit.

$$o_{pj} = \frac{1}{1 + e^{-net_{pj}}} \tag{A.1}$$

Step 4

Determine the error signal, and update the weights for each unit. This is a recursive procedure starting at the output layer, and working back down the net until the first hidden layer is reached. The weights should be updated according to equation (A.2) or equation (A.3) if a momentum is to be used.

$$\Delta_p w_{ji} = \eta \, \delta_{pj} o_{pi} \tag{A.2}$$

or

$$\Delta w_{ji}(t+1) = \eta \, \delta_{pj} o_{pi} + \alpha \, \Delta w_{ji}(t) \tag{A.3}$$

For the output layer, the error for each unit j, δ_{pj}, is calculated using,

$$\delta_{pj} = (t_{pj} - o_{pj}) \, o_{pj} (1 - o_{pj}) \tag{A.4}$$

and for a hidden layer unit, the error for each unit must be calculated in a recursive fashion, by propagating the errors down from the layer above

$$\delta_{pj} = o_{pj} (1 - o_{pj}) \sum_k \delta_{pk} w_{kj} \tag{A.5}$$

where the k subscripts denote a unit in the layer above.

Step 5

If there are further exemplars in the training set, then repeat the procedure by going to

step 2, and presenting the next input / output pattern set.

Step 6

After each *epoch* (presentation of a whole training set), evaluate the global error using equations (A.6). Continue learning until the global error reaches the desired tolerance.

$$E_p=\frac{1}{2}\sum_j (t_{pj}-o_{pj})^2 \qquad (A.6)$$

Appendix B

Training set used for the distance-discriminator

desired output	input pattern
1.000000	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
0.500000	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1
1.000000	0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0
0.333333	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1
0.500000	0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0
1.000000	0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0
1.000000	0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1
0.707107	0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0
0.447214	0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0
0.316228	0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0
0.707107	0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1
1.000000	0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0
0.707107	0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0
0.447214	0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0
1.000000	0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0
0.447214	0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1
0.707107	0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0
1.000000	0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0
0.707107	0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0
0.500000	0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0
1.000000	0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0
0.316228	0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1
0.447214	0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0
0.707107	0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0
1.000000	0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0
0.333333	0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0
0.500000	0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0
1.000000	0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0
0.500000	0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1
0.447214	0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0
0.353553	0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0
0.277350	0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0
1.000000	0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0
0.707107	0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0
0.447214	0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0
0.316228	0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0
0.447214	0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1
0.500000	0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0
0.447214	0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0
0.353553	0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0
0.707107	0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0
1.000000	0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0
0.707107	0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0
0.447214	0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0
1.000000	0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0
0.353553	0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1
0.447214	0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0
0.500000	0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0
0.447214	0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0
0.447214	0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0
0.707107	0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0
1.000000	0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0
0.707107	0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0
0.500000	0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0
1.000000	0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0
0.277350	0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1

0.353553	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0
0.447214	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0
0.500000	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0
0.316228	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0
0.447214	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0
0.707107	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0
1.000000	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0
0.333333	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0
0.500000	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0
1.000000	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
0.333333	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1
0.316228	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0
0.277350	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0
0.235702	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0
0.500000	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0
0.447214	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0
0.353553	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0
0.277350	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0
1.000000	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0
0.707107	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0
0.447214	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0
0.316228	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
0.316228	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1
0.333333	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0
0.316228	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0
0.277350	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0
0.447214	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0
0.500000	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0
0.447214	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0
0.353553	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0
0.707107	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0
1.000000	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0
0.707107	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0
0.447214	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0
1.000000	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
0.277350	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0.316228	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0.333333	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0.316228	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0
0.353553	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0
0.447214	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0
0.500000	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0
0.447214	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0
0.447214	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0.707107	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0
1.000000	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0
0.707107	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0
0.500000	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
1.000000	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0.235702	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0.277350	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0.316228	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0.333333	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
0.277350	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
0.353553	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
0.447214	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
0.500000	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
0.316228	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0.447214	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0.707107	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
1.000000	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
0.333333	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0.500000	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
1.000000	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Appendix C

Training set used for max-filter

desired output	inputs	
0.5	0.5	0.4
0.5	0.4	0.5
0.7	0.7	0.1
0.7	0.1	0.7
0.2	0.1	0.2
0.2	0.2	0.1
0.9	0.9	0.3
0.9	0.3	0.9
0.8	0.8	0.4
0.8	0.4	0.8
0.6	0.3	0.6
0.6	0.6	0.3
0.8	0.7	0.8
0.8	0.8	0.7
0.7	0.7	0.3
0.7	0.3	0.7
0.3	0.3	0.2
0.3	0.2	0.3
0.4	0.1	0.4
0.4	0.4	0.1
0.9	0.2	0.9
0.9	0.9	0.2
1.0	0.8	1.0
1.0	1.0	0.8
0.5	0.5	0.3
0.5	0.3	0.5
0.7	0.7	0.5
0.7	0.5	0.7
0.6	0.6	0.6
0.2	0.2	0.2
0.4	0.4	0.4
0.1	0.1	0.1
0.8	0.8	0.8
0.9	0.9	0.9
0.3	0.3	0.3
1.0	1.0	1.0
0.5	0.5	0.5
0.7	0.7	0.7
0.1	0.1	0
0.1	0	0.1
0.2	0.2	0
0.2	0	0.2
0.3	0.3	0
0.3	0	0.3
0.4	0.4	0
0.4	0	0.4
0.5	0.5	0
0.5	0	0.5
0.6	0.6	0
0.6	0	0.6
0.7	0.7	0
0.7	0	0.7
0.8	0.8	0
0.8	0	0.8
0.9	0.9	0
0.9	0	0.9
1.0	1.0	0

```
1.0             0       1.0
0               0       0
```

Appendix D

Training set used for min-filter

desired output	inputs	
0.1	1.0	0.1
0.1	0.1	1.0
0.1	0.9	0.1
0.1	0.1	0.9
0.1	0.8	0.1
0.1	0.1	0.8
0.1	0.1	0.7
0.1	0.7	0.1
0.1	0.6	0.1
0.1	0.1	0.6
0.1	0.5	0.1
0.1	0.1	0.5
0.1	0.4	0.1
0.1	0.1	0.4
0.1	0.3	0.1
0.1	0.1	0.3
0.1	0.2	0.1
0.1	0.1	0.2
0.2	0.2	1.0
0.2	1.0	0.2
0.2	0.9	0.2
0.2	0.2	0.9
0.2	0.8	0.2
0.2	0.2	0.8
0.2	0.7	0.2
0.2	0.2	0.7
0.2	0.6	0.2
0.2	0.2	0.6
0.2	0.5	0.2
0.2	0.2	0.5
0.2	0.4	0.2
0.2	0.2	0.4
0.2	0.3	0.2
0.2	0.2	0.3
0.3	1.0	0.3
0.3	0.3	1.0
0.3	0.9	0.3
0.3	0.3	0.9
0.3	0.8	0.3
0.3	0.3	0.8
0.3	0.7	0.3
0.3	0.3	0.7
0.3	0.6	0.3
0.3	0.3	0.6
0.3	0.5	0.3
0.3	0.3	0.5
0.3	0.4	0.3
0.3	0.3	0.4
0.4	1.0	0.4
0.4	0.4	1.0
0.4	0.9	0.4
0.4	0.4	0.9
0.4	0.8	0.4
0.4	0.4	0.8
0.4	0.7	0.4
0.4	0.4	0.7
0.4	0.6	0.4

0.4	0.4	0.6
0.4	0.5	0.4
0.4	0.4	0.5
0.5	1.0	0.5
0.5	0.5	1.0
0.5	0.9	0.5
0.5	0.5	0.9
0.5	0.8	0.5
0.5	0.5	0.8
0.5	0.7	0.5
0.5	0.5	0.7
0.5	0.6	0.5
0.5	0.5	0.6
0.6	1.0	0.6
0.6	0.6	1.0
0.6	0.9	0.6
0.6	0.6	0.9
0.6	0.8	0.6
0.6	0.6	0.8
0.6	0.7	0.6
0.6	0.6	0.7
0.7	1.0	0.7
0.7	0.7	1.0
0.7	0.9	0.7
0.7	0.7	0.9
0.7	0.8	0.7
0.7	0.7	0.8
0.8	1.0	0.8
0.8	0.8	1.0
0.8	0.9	0.8
0.8	0.8	0.9
0.9	1.0	0.9
0.9	0.9	1.0
0.1	0.1	0.1
0.2	0.2	0.2
0.3	0.3	0.3
0.4	0.4	0.4
0.5	0.5	0.5
0.6	0.6	0.6
0.7	0.7	0.7
0.8	0.8	0.8
0.9	0.9	0.9
1.0	1.0	1.0
0.0	0.0	0.0
0.1	0.1	0
0.1	0	0.1
0.2	0.2	0
0.2	0	0.2
0.3	0.3	0
0.3	0	0.3
0.4	0.4	0
0.4	0	0.4
0.5	0.5	0
0.5	0	0.5
0.6	0.6	0
0.6	0	0.6
0.7	0.7	0
0.7	0	0.7
0.8	0.8	0
0.8	0	0.8
0.9	0.9	0
0.9	0	0.9
1.0	1.0	0
1.0	0	1.0

Appendix E

Training set initially used for the line-pair evaluator

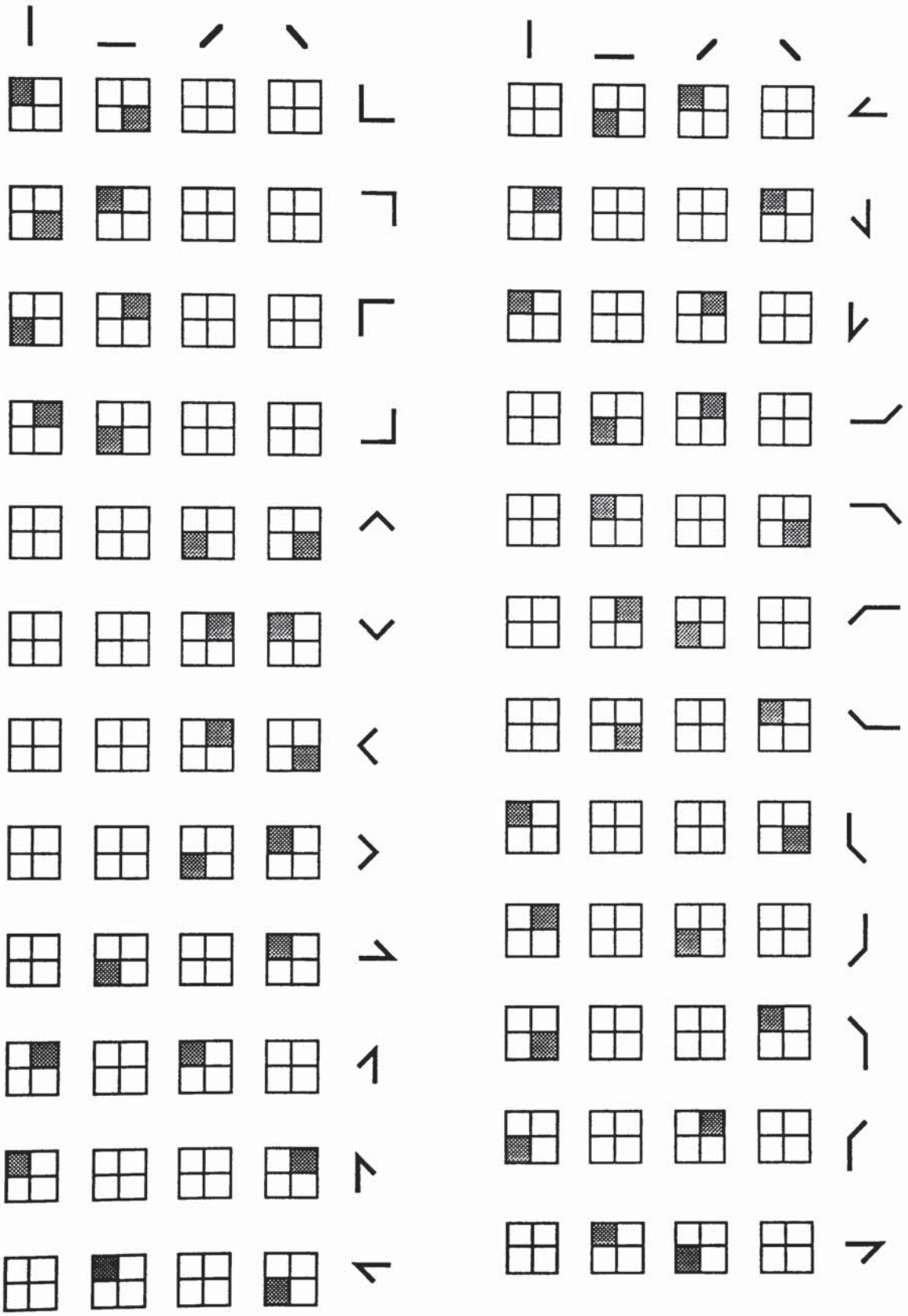
desired output	inputs	
0.125	0.25	0.25
0.0625	0.5	0.25
0.04167	0.75	0.25
0.03125	1	0.25
0.25	0.5	0.5
0.16667	0.75	0.5
0.125	1	0.5
0.375	0.75	0.75
0.28125	1	0.75
0.5	1	1

Training set finally used to train the line-pair evaluator

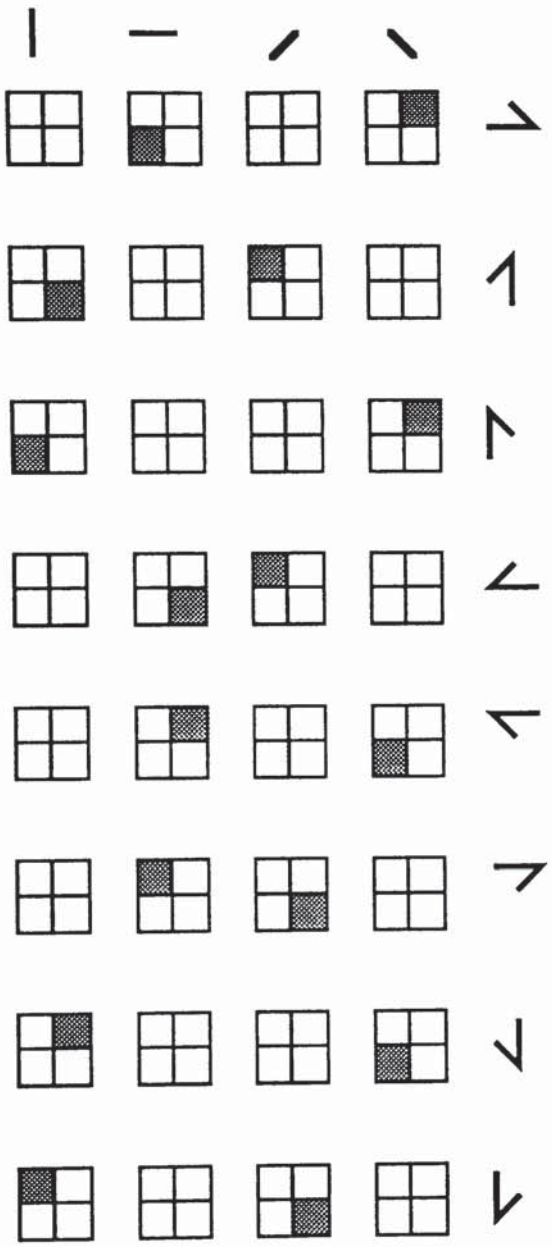
desired outputs		inputs
0.0	0	0
0.04167	0.12	0.1
0.02	0.09	0.06
0.125	0.25	0.25
0.06667	0.3	0.2
0.0625	0.5	0.25
0.04167	0.75	0.25
0.13333	0.6	0.4
0.03125	1 0	25
0.25	0.5	0.5
0.11429	0.7	0.4
0.05953	0.86	0.32
0.16667	0.75	0.5
0.125	1	0.5
0.375	0.75	0.75
0.02	1	0.2
0.28125	1	0.75
0.5	1	1

Appendix F

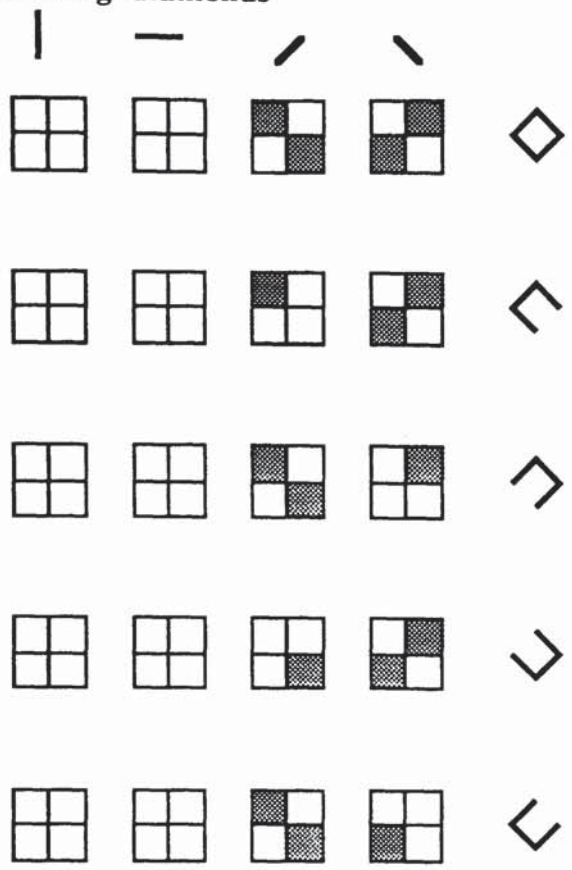
List of allowable corners



Alternative way of coding acute corners



Coding diamonds



Training set used to test for reproducibility of untrained exemplars

desired output	input patterns
1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1
1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0
1	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1
1	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0
1	0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0
1	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1
1	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0
1	0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0
1	0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0
0	0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1
1	0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0
1	0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0
0	0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0
1	0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0
0	0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1
1	0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0
1	0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0
0	0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0
1	0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0
1	0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0
1	0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1
1	0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0
1	0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0
1	0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0

1	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0
1	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0
1	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0
1	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0
1	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0
1	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0
1	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0
1	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0
1	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0
1	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0
1	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0
1	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0
1	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0
1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0
1	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0
1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0
1	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0
1	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0
1	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0
1	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0
1	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0
1	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0
1	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0
1	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0

1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0
1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

1	0	0	0	0	0	0	0	0	1	0	0	1	0	1	1	0
1	0	0	0	0	0	0	0	0	1	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	0	1	0	0	1	0	1	0	0
1	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0
1	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0

Training set used for corner-detector

desired outputs	input patterns
0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1
0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0
0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1

0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0
0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0
0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1
0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0
0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0
0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0
0 0 1	0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1
0 0 0	0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0
0 0 0	0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0
0 0 1	0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0
0 0 0	0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0
0 0 1	0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1
0 0 0	0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0
0 0 0	0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0
0 0 1	0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0
0 0 0	0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0
0 0 0	0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0
0 0 0	0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1
0 0 0	0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0
0 0 0	0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0
0 1 0	0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0
0 0 0	0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0
0 0 0	0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0
0 0 0	0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0
1 0 0	0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0
0 0 0	0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1
0 0 0	0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0
1 0 0	0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0
1 0 0	0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0
0 0 0	0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0
0 0 0	0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0
0 1 0	0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0
1 0 0	0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0
0 0 0	0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0
0 0 0	0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1
1 0 0	0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0
0 0 0	0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0
0 0 0	0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0
0 1 0	0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0
0 0 0	0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0
0 0 0	0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0
0 0 0	0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0
0 0 0	0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0
0 1 0	0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0	0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0
0 0 0	0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0	0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0
1 0 0	0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0
1 0 0	0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0	0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0
0 0 0	0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0	0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0	0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0	0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0	0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0	0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0	0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0
0 0 0	0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0	0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0
0 0 0	0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0
0 0 0	0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0	0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 0	0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0	0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0	0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0	0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0	0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0	0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0	0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0

```
0 1 0
0 0 0
0 0 0
0 0 0
1 0 0
0 0 0
0 0 0
0 0 0
0 0 1
0 0 0
0 0 0
1 0 0
0 0 0
0 0 0
0 0 0
0 1 0
0 0 0
0 0 0
0 0 0
0 0 1
0 0 0
0 0 0
0 0 0
0 0 0
1 0 0
0 0 0
1 0 0
0 0 0
0 1 0
0 0 0
1 0 0
0 0 0
0 0 1
0 0 0
0 0 0
0 0 0
0 0 0
0 1 0
0 0 0
1 0 0
0 0 0
1 0 0
0 0 0
1 0 0
0 0 0
0 0 1
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
```

```
0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0
0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0
0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0
0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0
0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0
0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0
0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0
0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0
0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

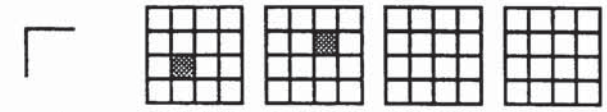
```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
```

```
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
```

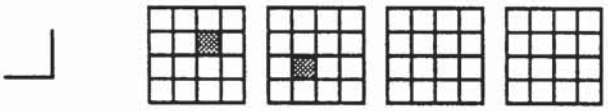
```
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 1 0 0 1 0 1 1 0
0 0 0 0 0 0 0 0 1 0 0 0 0 1 1 0
0 0 0 0 0 0 0 0 1 0 0 1 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0
0 0 0 0 0 0 0 0 1 0 0 1 0 0 1 0
```

Appendix G

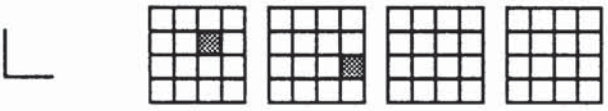
test data 1-8



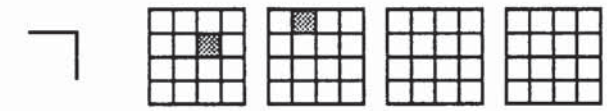
actual	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.499953
desired	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.500000



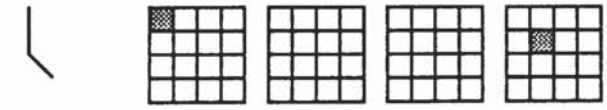
actual	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.499948
desired	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.500000



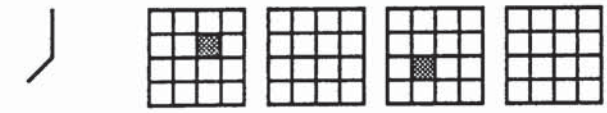
actual	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.499922
desired	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.500000



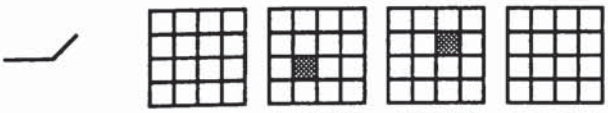
actual	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.499819
desired	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.500000



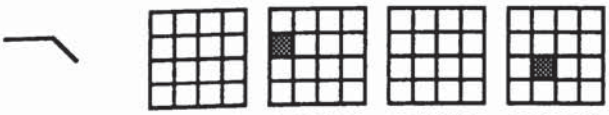
actual	0.000000	0.000000	0.000000	0.000000	0.000000	0.499763	0.000000
desired	0.000000	0.000000	0.000000	0.000000	0.000000	0.500000	0.000000



actual	0.000000	0.000000	0.000000	0.000000	0.000000	0.499752	0.000000
desired	0.000000	0.000000	0.000000	0.000000	0.000000	0.500000	0.000000



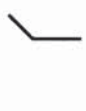
actual	0.000000	0.000000	0.000000	0.000000	0.000102	0.499303	0.000000
desired	0.000000	0.000000	0.000000	0.000000	0.000000	0.500000	0.000000

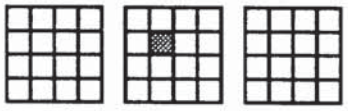


actual	0.000000	0.000000	0.000000	0.000000	0.000000	0.499669	0.000000
desired	0.000000	0.000000	0.000000	0.000000	0.000000	0.500000	0.000000

test data 9-16

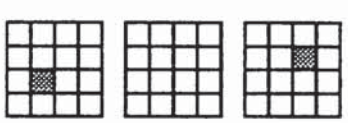
actual	0.000000	0.000000	0.000000	0.000000	0.000000	0.499880	0.000040
desired	0.000000	0.000000	0.000000	0.000000	0.000000	0.500000	0.000000



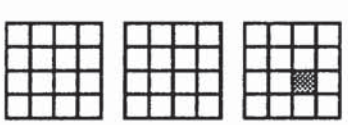


actual	0.000000	0.000000	0.000000	0.000000	0.000000	0.499730	0.000040
desired	0.000000	0.000000	0.000000	0.000000	0.000000	0.500000	0.000000

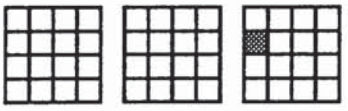
actual	0.000000	0.000000	0.000000	0.000000	0.000000	0.499827	0.000000
desired	0.000000	0.000000	0.000000	0.000000	0.000000	0.500000	0.000000



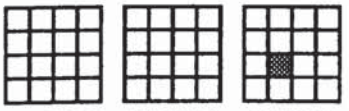
actual	0.000000	0.000000	0.000000	0.000000	0.000000	0.499912	0.000000
desired	0.000000	0.000000	0.000000	0.000000	0.000000	0.500000	0.000000



actual	0.000000	0.000000	0.000000	0.000000	0.000000	0.000055	0.499837
desired	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.500000



actual	0.000000	0.000000	0.000000	0.000000	0.000022	0.000000	0.499796
desired	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.500000

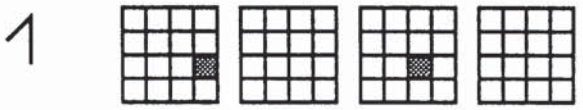


actual	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.499829
desired	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.500000

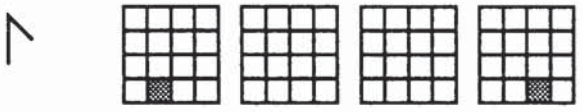
actual	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.499796
desired	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.500000

test data 17-24

actual	0.000000	0.000000	0.000000	0.000000	0.499829	0.000051	0.000040
desired	0.000000	0.000000	0.000000	0.000000	0.500000	0.000000	0.000000



actual	0.000000	0.000000	0.000000	0.000000	0.499913	0.000000	0.000000
desired	0.000000	0.000000	0.000000	0.000000	0.500000	0.000000	0.000000



actual	0.000000	0.000000	0.000000	0.000000	0.499865	0.000000	0.000000
desired	0.000000	0.000000	0.000000	0.000000	0.500000	0.000000	0.000000

actual	0.000000	0.000000	0.000000	0.000000	0.499831	0.000000	0.000000
desired	0.000000	0.000000	0.000000	0.000000	0.500000	0.000000	0.000000

actual	0.000000	0.000000	0.000000	0.000000	0.499863	0.000000	0.000000
desired	0.000000	0.000000	0.000000	0.000000	0.500000	0.000000	0.000000

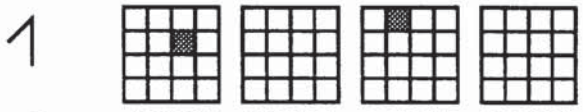
actual	0.000000	0.000000	0.000000	0.000000	0.499863	0.000149	0.000040
desired	0.000000	0.000000	0.000000	0.000000	0.500000	0.000000	0.000000

actual	0.000000	0.000000	0.000000	0.000000	0.499913	0.000000	0.000000
desired	0.000000	0.000000	0.000000	0.000000	0.500000	0.000000	0.000000

actual	0.000000	0.000000	0.000000	0.000000	0.499885	0.000000	0.000000
desired	0.000000	0.000000	0.000000	0.000000	0.500000	0.000000	0.000000

test data 25-32

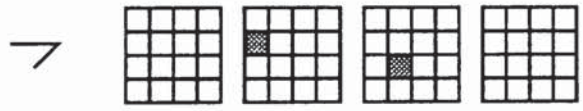
actual	0.000000	0.000000	0.000000	0.000000	0.499829	0.000000	0.000040
desired	0.000000	0.000000	0.000000	0.000000	0.500000	0.000000	0.000000



actual	0.000000	0.000000	0.000000	0.000000	0.499830	0.000000	0.000000
desired	0.000000	0.000000	0.000000	0.000000	0.500000	0.000000	0.000000

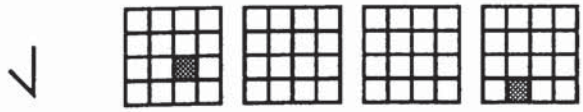
actual	0.000000	0.000000	0.000000	0.000000	0.499882	0.000000	0.000000
desired	0.000000	0.000000	0.000000	0.000000	0.500000	0.000000	0.000000

actual	0.000000	0.000000	0.000000	0.000000	0.499807	0.000000	0.000040
desired	0.000000	0.000000	0.000000	0.000000	0.500000	0.000000	0.000000



actual	0.000000	0.000000	0.000000	0.000000	0.499863	0.000000	0.000000
desired	0.000000	0.000000	0.000000	0.000000	0.500000	0.000000	0.000000

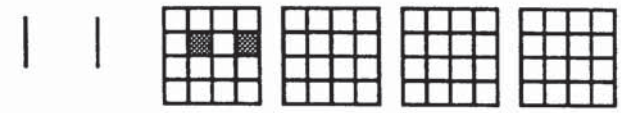
actual	0.000000	0.000000	0.000000	0.000000	0.499794	0.000000	0.000040
desired	0.000000	0.000000	0.000000	0.000000	0.500000	0.000000	0.000000



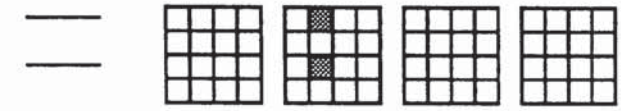
actual	0.000000	0.000000	0.000000	0.000000	0.499790	0.000000	0.000000
desired	0.000000	0.000000	0.000000	0.000000	0.500000	0.000000	0.000000

actual	0.000000	0.000000	0.000000	0.000000	0.499896	0.000000	0.000000
desired	0.000000	0.000000	0.000000	0.000000	0.500000	0.000000	0.000000

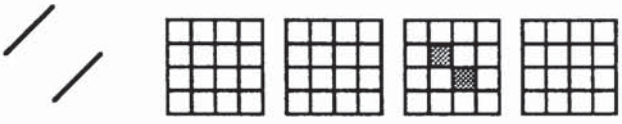
test data 33-40



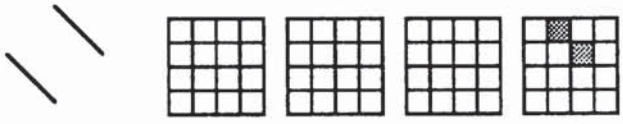
actual	0.249963	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
desired	0.250000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000



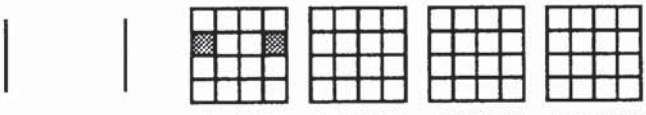
actual	0.000000	0.249963	0.000000	0.000000	0.000000	0.000000	0.000081
desired	0.000000	0.250000	0.000000	0.000000	0.000000	0.000000	0.000000



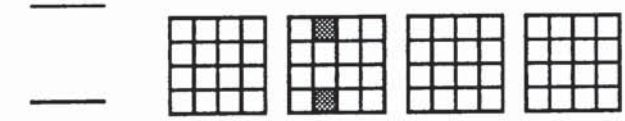
actual	0.000000	0.000000	0.353466	0.000000	0.000000	0.000000	0.000000
desired	0.000000	0.000000	0.353600	0.000000	0.000000	0.000000	0.000000



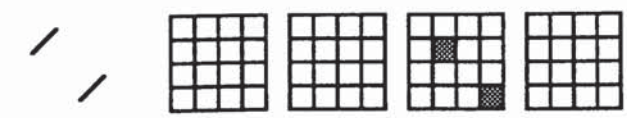
actual	0.000000	0.000000	0.000000	0.035348	0.000000	0.000000	0.000000
desired	0.000000	0.000000	0.000000	0.035360	0.000000	0.000000	0.000000



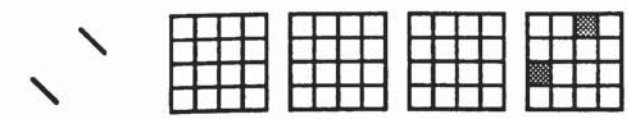
actual	0.166621	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
desired	0.166700	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000



actual	0.000000	0.166621	0.000000	0.000000	0.000000	0.000000	0.000081
desired	0.000000	0.166700	0.000000	0.000000	0.000000	0.000000	0.000000

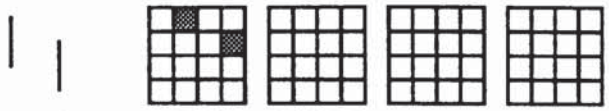


actual	0.000000	0.000000	0.176688	0.000000	0.000000	0.000000	0.000000
desired	0.000000	0.000000	0.176800	0.000000	0.000000	0.000000	0.000000

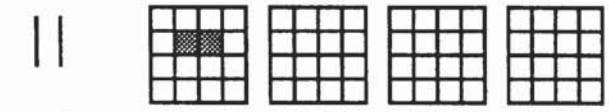


actual	0.000000	0.000000	0.000000	0.176725	0.000000	0.000000	0.000000
desired	0.000000	0.000000	0.000000	0.176800	0.000000	0.000000	0.000000

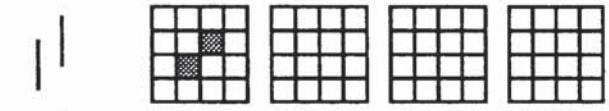
test data 41-48



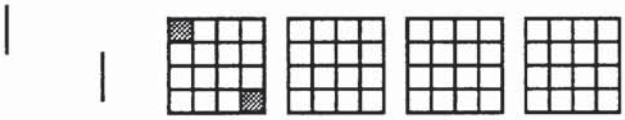
actual	0.223556	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
desired	0.223600	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000



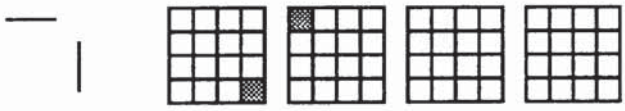
actual	0.498960	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
desired	0.500000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000



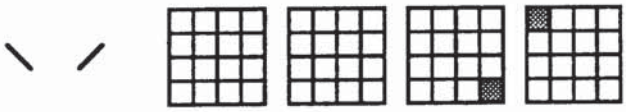
actual	0.353421	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
desired	0.353600	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000



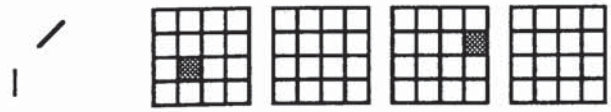
actual	0.117798	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
desired	0.117900	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000



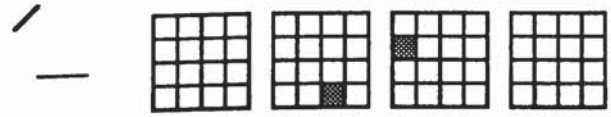
actual	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000040
desired	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000



actual	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
desired	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000



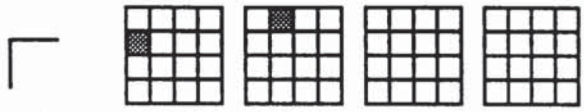
actual	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000040
desired	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000



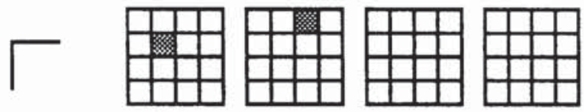
actual	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000040
desired	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000

Appendix H

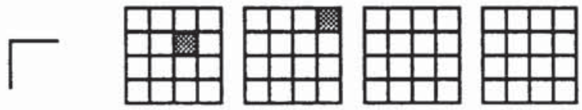
test data 49-56



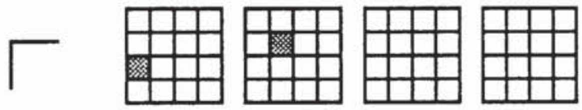
actual	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.499953
desired	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.500000



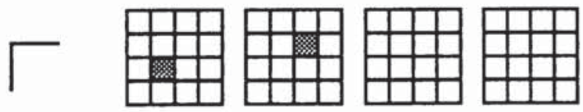
actual	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.499953
desired	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.500000



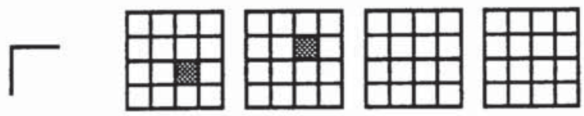
actual	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.499953
desired	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.500000



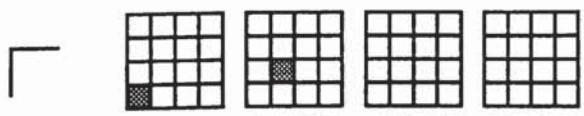
actual	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.499953
desired	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.500000



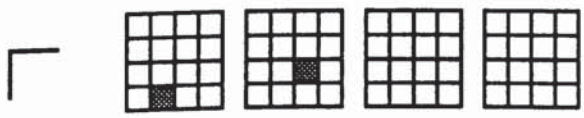
actual	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.499953
desired	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.500000



actual	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.499953
desired	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.500000

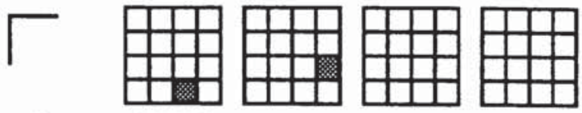


actual	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.499953
desired	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.500000

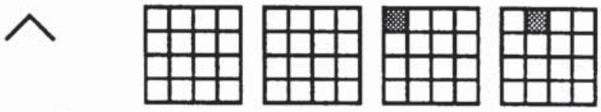


actual	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.499953
desired	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.500000

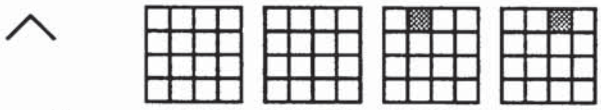
test data 57-64



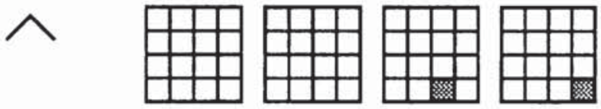
actual	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.499953
desired	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.500000



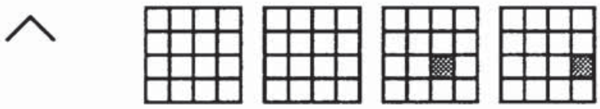
actual	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.499796
desired	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.500000



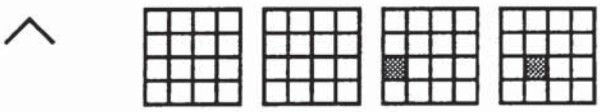
actual	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.499796
desired	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.500000



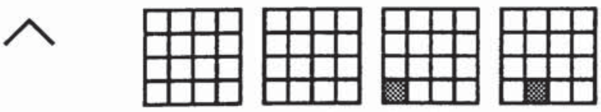
actual	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.499796
desired	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.500000



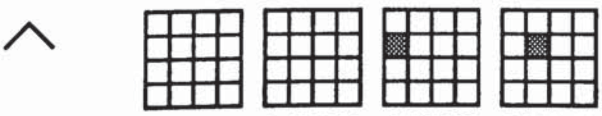
actual	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.499796
desired	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.500000



actual	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.499796
desired	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.500000



actual	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.499796
desired	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.500000

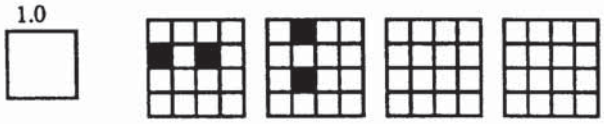


actual	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.499796
desired	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.500000

Appendix I

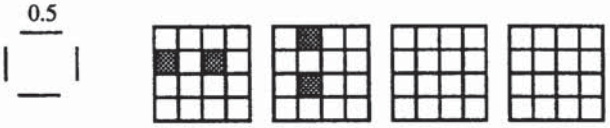
test data 65-72

1.0



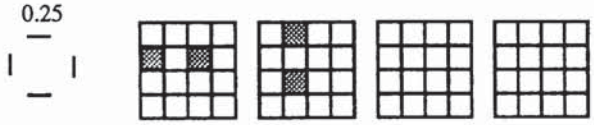
actual	0.249937	0.249932	0.000000	0.000000	0.000000	0.000000	1.999561
desired	0.250000	0.250000	0.000000	0.000000	0.000000	0.000000	2.000000

0.5



actual	0.124657	0.124654	0.000000	0.000000	0.000000	0.000000	0.997823
desired	0.125000	0.125000	0.000000	0.000000	0.000000	0.000000	1.000000

0.25



actual	0.062541	0.062540	0.000000	0.000000	0.000000	0.000000	0.500320
desired	0.062500	0.062500	0.000000	0.000000	0.000000	0.000000	0.500000

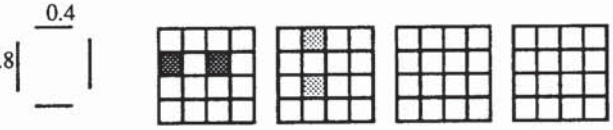
0.5

0.25

actual	0.062541	0.124654	0.000000	0.000000	0.000000	0.000000	0.246221
desired	0.062500	0.125000	0.000000	0.000000	0.000000	0.000000	0.250000

0.4

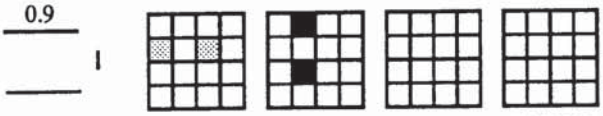
0.8



actual	0.199528	0.099170	0.000000	0.000000	0.000000	0.000000	0.398626
desired	0.200000	0.100000	0.000000	0.000000	0.000000	0.000000	0.400000

0.9

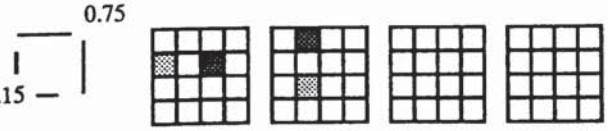
0.1



actual	0.022113	0.217654	0.000000	0.000000	0.000000	0.000000	0.017982
desired	0.025000	0.225000	0.000000	0.000000	0.000000	0.000000	0.022200

0.75

0.15



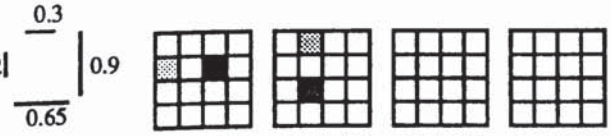
actual	0.006824	0.006818	0.000000	0.000000	0.000000	0.000000	0.474545
desired	0.007500	0.007500	0.000000	0.000000	0.000000	0.000000	0.480000

0.3

0.2

0.9

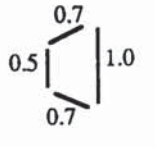
0.65

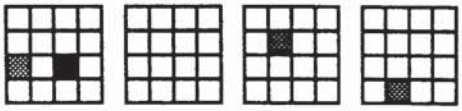


actual	0.010959	0.034081	0.000000	0.000000	0.000000	0.000000	0.384329
desired	0.011100	0.034600	0.000000	0.000000	0.000000	0.000000	0.382200

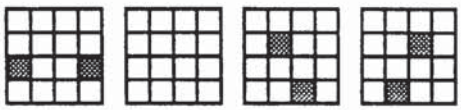
Appendix J

test data 73-80

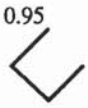


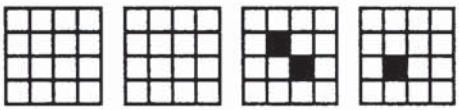


actual	0.062127	0.000000	0.000000	0.000000	0.488533	0.359239	0.000000
desired	0.062500	0.000000	0.000000	0.000000	0.490000	0.357100	0.000000

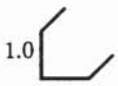


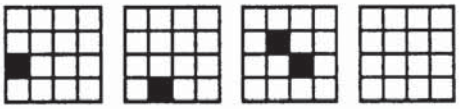
actual	0.100062	0.000000	0.134258	0.134257	0.000000	1.200689	0.600422
desired	0.100000	0.000000	0.134200	0.134200	0.000000	1.200000	0.600000



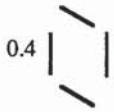


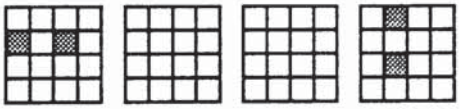
actual	0.000000	0.000000	0.368697	0.000000	0.000000	0.000058	0.995816
desired	0.000000	0.000000	0.335900	0.000000	0.000000	0.000000	0.950000



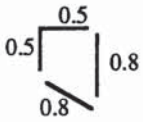


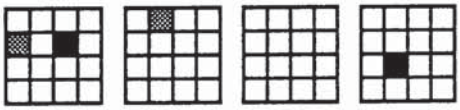
actual	0.000000	0.000000	0.353466	0.000000	0.000102	0.999216	0.499922
desired	0.000000	0.000000	0.353600	0.000000	0.000000	1.000000	0.500000



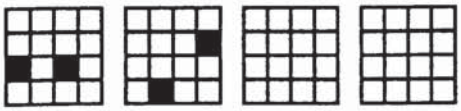


actual	0.099172	0.000000	0.000000	0.099170	0.396704	0.396673	0.000000
desired	0.100000	0.000000	0.000000	0.100000	0.400000	0.400000	0.000000

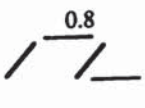




actual	0.078299	0.000000	0.000000	0.000000	0.399018	0.156543	0.405930
desired	0.078100	0.000000	0.000000	0.000000	0.400000	0.156300	0.406300



actual	0.260705	0.167717	0.000000	0.000000	0.000000	0.000000	1.564517
desired	0.237500	0.167900	0.000000	0.000000	0.000000	0.000000	1.425000



actual	0.000000	0.178461	0.199528	0.000000	0.797967	0.399143	0.000032
desired	0.000000	0.178900	0.200000	0.000000	0.800000	0.400000	0.000000

Appendix K

The following is a table of 500 random coordinates used to train a Kohonen network.

3,0	7,4	0,7	3,4	4,3	3,0	5,1	6,0	6,3	9,0	8,2	9,2
4,2	8,9	8,6	1,3	5,1	2,7	1,7	9,7	6,0	7,0	6,2	2,2
4,7	6,8	3,0	2,6	5,0	1,4	7,9	6,4	6,4	9,1	5,4	3,7
0,9	7,7	8,7	8,8	9,0	8,3	9,1	7,3	4,6	7,2	3,8	9,6
6,2	7,8	2,4	1,6	5,6	2,7	0,0	2,7	0,1	2,6	3,6	4,6
4,5	2,8	1,3	8,3	3,5	8,5	8,9	4,9	4,4	6,9	8,4	2,7
7,6	6,4	4,7	0,5	2,5	7,0	6,8	1,7	4,7	7,7	3,1	4,6
9,8	1,6	8,6	0,6	7,1	4,2	8,2	9,5	1,5	9,9	9,2	3,7
3,7	6,6	4,0	4,6	9,4	8,7	4,1	6,9	1,7	6,4	6,9	5,7
0,4	4,3	5,7	2,5	2,5	4,1	0,4	4,6	2,1	6,5	6,4	7,7
9,4	1,9	9,2	2,0	1,9	6,2	3,1	1,6	1,1	3,2	7,4	6,1
7,5	4,3	4,7	1,9	9,9	8,0	0,6	6,2	8,1	9,9	9,6	9,2
0,9	9,3	5,3	7,2	0,7	4,2	5,8	1,7	1,7	8,1	0,2	1,2
8,2	3,8	3,7	2,7	1,0	2,4	8,7	4,9	0,1	2,7	2,5	0,0
6,8	1,1	4,1	5,9	2,2	7,8	5,4	0,9	6,8	2,6	3,6	2,4
1,7	5,3	1,6	6,8	6,7	2,4	5,7	2,7	9,0	5,9	0,3	2,9
2,4	1,6	5,7	2,1	6,9	2,9	7,5	9,4	4,2	7,8	2,5	4,8
0,2	1,7	0,7	1,4	4,2	5,8	7,1	8,6	3,6	7,5	3,5	6,3
1,1	6,3	8,9	9,1	3,7	4,8	8,5	3,5	5,4	2,3	1,8	5,2
9,8	1,3	9,7	2,0	8,0	3,2	7,5	5,9	0,9	6,2	0,9	7,6
6,2	8,1	6,2	9,6	3,7	4,3	6,5	4,9	5,9	4,5	7,7	1,6
2,4	8,5	1,2	5,7	9,2	1,1	4,6	7,4	7,5	0,5	9,9	2,8
7,8	0,2	1,6	9,6	3,9	2,3	7,2	3,8	9,8	2,9	2,4	7,8
0,2	2,5	7,2	1,8	5,0	4,4	8,2	1,6	2,2	5,5	0,6	5,6
4,7	5,2	0,5	6,4	1,8	7,1	3,7	2,9	8,4	8,1	1,9	9,0
9,4	0,2	3,1	7,1	0,7	0,3	1,0	1,9	8,6	4,3	3,4	1,8
6,2	1,8	7,3	5,6	1,7	8,4	5,1	7,9	8,7	9,9	3,9	0,8
6,2	6,2	9,5	0,6	8,1	2,2	2,3	1,2	9,4	6,4	1,1	6,1
9,2	6,1	7,2	9,9	1,1	2,1	6,1	1,4	5,4	5,2	1,1	2,7
3,2	6,6	4,8	1,6	6,5	3,5	9,6	8,3	3,6	3,6	9,3	0,1
8,1	4,9	8,2	7,0	9,0	4,2	7,7	7,0	2,7	3,7	3,9	2,5
6,2	4,3	5,6	0,7	2,3	3,9	5,5	8,5	6,0	1,0	3,5	0,2
5,8	9,8	8,4	8,6	5,2	4,5	6,5	7,2	4,7	3,2	9,4	7,5
3,4	3,4	7,9	6,0	7,4	5,6	1,4	5,9	8,9	0,2	0,0	3,1
2,0	0,0	0,8	4,6	7,3	8,7	2,5	3,7	7,2	9,7	5,6	9,1
9,2	9,5	6,8	5,0	2,4	7,0	8,3	1,0	7,5	4,2	4,7	5,1
7,6	5,8	6,9	2,7	8,6	7,0	4,6	9,5	7,4	8,3	4,1	0,3
1,4	0,2	2,0	1,1	5,8	1,9	9,1	9,2	1,0	8,8	8,5	4,6
6,5	0,6	7,6	3,7	3,6	5,7	0,4	5,7	3,2	0,1	9,2	4,8
3,8	9,7	2,6	6,7	1,6	3,9	5,2	9,1	4,8	1,3	0,6	4,8
8,5	4,6	0,8	4,6	3,0	3,5	2,7	0,2	9,5	6,1	5,6	6,1
1,7	7,8	9,1	0,7	7,5	9,1	6,3	6,6				

Appendix L

Training set used to train a multi-layer perceptron to recognise shapes.

Desired outputs	Inputs						
.1 0 0 0	0.025	0.025	0	0	0	0	0.2
.5 0 0 0	0.1	0.1	0	0	0	0	0.8
1 0 0 0	0.225	0.225	0	0	0	0	1.8
.05 0 0 0	0.0125	0	0	0	0	0	0.05
.25 0 0 0	0.125	0	0	0	0	0	0.5
.75 0 0 0	0.25	0	0	0	0	0	1.0
.05 0 0 0	0	0.0375	0	0	0	0	0.15
.25 0 0 0	0	0.125	0	0	0	0	0.5
.75 0 0 0	0	0.2375	0	0	0	0	0.95
0 .1 0 0	0	0.025	0	0.025	0.1	0.1	0
0 .5 0 0	0	0.15	0	0.15	0.6	0.6	0
0 1 0 0	0	0.2375	0	0.2375	0.95	0.95	0
0 .05 0 0	0	0	0	0.0125	0.025	0.025	0
0 .25 0 0	0	0	0	0.125	0.25	0.25	0
0 .75 0 0	0	0	0	0.25	0.5	0.5	0
0 .05 0 0	0	0.025	0	0	0.05	0.05	0
0 .25 0 0	0	0.15	0	0	0.3	0.3	0
0 .75 0 0	0	0.2	0	0	0.4	0.4	0
0 .1 0 0	0	0.025	0.025	0	0.1	0.1	0
0 .5 0 0	0	0.125	0.125	0	0.5	0.5	0
0 1 0 0	0	0.2375	0.2375	0	0.95	0.95	0
0 .05 0 0	0	0	0.025	0	0.005	0.005	0
0 .25 0 0	0	0	0.125	0	0.25	0.25	0
0 .75 0 0	0	0	0.25	0	0.5	0.5	0
0 .1 0 0	0.025	0	0.025	0	0.1	0.1	0
0 .5 0 0	0.1	0	0.1	0	0.4	0.4	0
0 1 0 0	0.25	0	0.25	0	1.0	1.0	0
0 .05 0 0	0.0375	0	0	0	0.075	0.075	0
0 .25 0 0	0.125	0	0	0	0.25	0.25	0
0 .75 0 0	0.225	0	0	0	0.45	0.45	0
0 .05 0 0	0	0	0.025	0	0.05	0.05	0
0 .25 0 0	0	0	0.15	0	0.3	0.3	0
0 .75 0 0	0	0	0.25	0	0.5	0.5	0
0 .1 0 0	0.0125	0	0	0.0125	0.05	0.05	0
0 .5 0 0	0.125	0	0	0.125	0.5	0.5	0
0 1 0 0	0.25	0	0	0.25	1.0	1.0	0
0 .05 0 0	0	0	0	0.0025	0.005	0.005	0
0 .25 0 0	0	0	0	0.125	0.25	0.25	0
0 .75 0 0	0	0	0	0.25	0.5	0.5	0
0 .1 0 0	0	0.025	0	0	0.1	0.1	0
0 .5 0 0	0	0.125	0	0	0.5	0.5	0
0 1 0 0	0	0.25	0	0	1.0	1.0	0
0 .05 0 0	0	0	0	0	0	0.05	0

0 .25 0 0	0	0	0	0	0	0.6	0
0 .75 0 0	0	0	0	0	0	0.9	0
0 .1 0 0	0.0375	0	0	0	0.15	0.15	0
0 .5 0 0	0.1	0	0	0	0.4	0.4	0
0 1 0 0	0.25	0	0	0	1.0	1.0	0
.1 0 0 0	0	0	0.0177	0.0177	0	0	0.1
.5 0 0 0	0	0	0.1591	0.1591	0	0	0.9
1 0 0 0	0	0	0.3182	0.3182	0	0	1.8
.05 0 0 0	0	0	0	0.0071	0	0	0.02
.25 0 0 0	0	0	0	0.1768	0	0	0.5
.75 0 0 0	0	0	0	0.3536	0	0	1.0
.05 0 0 0	0	0	0.0354	0	0	0	0.1
.25 0 0 0	0	0	0.2121	0	0	0	0.6
.75 0 0 0	0	0	0.3359	0	0	0	0.95
0 0 0 .1	0.0167	0	0.0224	0.0224	0	0.2	0.1
0 0 0 .5	0.1	0	0.1342	0.1342	0	1.2	0.6
0 0 0 1	0.158	0	0.2124	0.2124	0	1.9	0.95
0 0 0 .05	0	0	0.0112	0.0112	0	0.05	0.05
0 0 0 .25	0	0	0.1118	0.1118	0	0.5	0.5
0 0 0 .75	0	0	0.2236	0.2236	0	1.0	1.0
0 0 0 .05	0.0167	0	0	0.0224	0	0.15	0.05
0 0 0 .25	0.0833	0	0	0.1118	0	0.75	0.25
0 0 0 .75	0.15	0	0	0.2014	0	1.35	0.45
0 0 0 .05	0.0167	0	0.0224	0	0	0.15	0.05
0 0 0 .25	0.0833	0	0.1118	0	0	0.75	0.25
0 0 0 .75	0.15	0	0.2014	0	0	1.35	0.45
0 0 0 .05	0.0083	0	0	0	0	0.05	0.05
0 0 0 .25	0.0833	0	0	0	0	0.5	0.25
0 0 0 .75	0.15	0	0	0	0	0.9	0.45
0 0 0 .05	0	0	0.0224	0	0	0.1	0.05
0 0 0 .25	0	0	0.1342	0	0	0.6	0.3
0 0 0 .75	0	0	0.2236	0	0	1.0	0.5
0 0 0 .05	0	0	0	0.0224	0	0.1	0.05
0 0 0 .25	0	0	0	0.1342	0	0.6	0.3
0 0 0 .75	0	0	0	0.2236	0	1.0	0.5
0 0 0 .1	0	0.0167	0.0224	0.0224	0	0.2	0.1
0 0 0 .5	0	0.1	0.1342	0.1342	0	1.2	0.6
0 0 0 1	0	0.1667	0.2236	0.2236	0	2.0	1.0
0 0 0 .05	0	0	0.0224	0.0224	0	0.1	0.1
0 0 0 .25	0	0	0.1118	0.1118	0	0.5	0.5
0 0 0 .75	0	0	0.2124	0.2124	0	0.95	0.95
0 0 0 .05	0	0.0083	0	0.0112	0	0.075	0.025
0 0 0 .25	0	0.0667	0	0.089	0	0.6	0.2
0 0 0 .75	0	0.15	0	0.2012	0	1.35	0.45
0 0 0 .05	0	0.0083	0.0112	0	0	0.075	0.025
0 0 0 .25	0	0.0667	0.089	0	0	0.6	0.2
0 0 0 .75	0	0.15	0.2012	0	0	1.35	0.45
0 0 0 .05	0	0.0167	0	0	0	0.1	0.05
0 0 0 .25	0	0.0833	0	0	0	0.5	0.25

0 0 0 .75	0	0.1667	0	0	0	1.0	0.5
0 0 0 .05	0	0	0	0.0112	0	0.05	0.025
0 0 0 .25	0	0	0	0.1006	0	0.45	0.225
0 0 0 .75	0	0	0	0.2124	0	0.95	0.475
0 0 0 .05	0	0	0.0224	0	0	0.1	0.05
0 0 0 .25	0	0	0.1118	0	0	0.5	0.25
0 0 0 .75	0	0	0.2236	0	0	1.0	0.5
0 0 0 .1	0.0224	0.0224	0.0354	0	0	0.2	0.1
0 0 0 .5	0.1118	0.1118	0.1768	0	0	1.0	0.5
0 0 0 1	0.2124	0.2124	0.3356	0	0	1.8	0.9
0 0 0 .05	0.0112	0	0.0177	0	0	0.075	0.025
0 0 0 .25	0.1006	0	0.1591	0	0	0.675	0.225
0 0 0 .75	0.2236	0	0.3536	0	0	1.5	0.5
0 0 0 .05	0	0.0224	0.0354	0	0	0.15	0.5
0 0 0 .25	0	0.1342	0.2121	0	0	0.9	0.3
0 0 0 .75	0	0.2236	0.3536	0	0	1.5	0.5
0 0 0 .05	0	0	0.0177	0	0	0.05	0.025
0 0 0 .25	0	0	0.1767	0	0	0.5	0.25
0 0 0 .75	0	0	0.3536	0	0	1.0	0.5
0 0 .05 0	0	0	0	0	0	0.05	0.05
0 0 .25 0	0	0	0	0	0	0.25	0.25
0 0 .75 0	0	0	0	0	0	0.45	0.45
0 0 0 .1	0.0112	0.0112	0	0.0177	0	0.1	0.05
0 0 0 .5	0.1006	0.1006	0	0.1591	0	0.9	0.45
0 0 0 1	0.2012	0.2012	0	0.3182	0	1.8	0.9
0 0 0 .05	0	0.0224	0	0.0354	0	0.15	0.05
0 0 0 .25	0	0.1342	0	0.2121	0	0.9	0.3
0 0 0 .75	0	0.2012	0	0.3182	0	1.35	0.45
0 0 0 .05	0.0224	0	0	0.0354	0	0.15	0.05
0 0 0 .25	0.1118	0	0	0.1768	0	0.75	0.25
0 0 0 .75	0.2236	0	0	0.3536	0	1.5	0.5
0 0 0 .05	0	0	0	0.0177	0	0.05	0.025
0 0 0 .25	0	0	0	0.1591	0	0.45	0.225
0 0 0 .75	0	0	0	0.3182	0	0.9	0.45
0 0 .1 0	0	0	0	0.0354	0	0.1	0.15
0 0 .5 0	0	0	0	0.1768	0	0.5	0.75
0 0 1 0	0	0	0	0.3182	0	0.9	1.35
0 0 .1 0	0	0	0.0354	0	0	0.1	0.15
0 0 .5 0	0	0	0.1768	0	0	0.5	0.75
0 0 1 0	0	0	0.3536	0	0	1.0	1.5
0 0 .05 0	0	0	0	0.0354	0	0.05	0.1
0 0 .25 0	0	0	0	0.1768	0	0.24	0.5
0 0 .75 0	0	0	0	0.3568	0	0.45	0.9
0 0 .05 0	0	0	0.0354	0	0	0.05	0.1
0 0 .25 0	0	0	0.1768	0	0	0.25	0.5
0 0 .75 0	0	0	0.3182	0	0	0.45	0.9
0 0 0 0	0	0	0	0	0	0	0
.01 0 0 0	0	0	0	0	0	0	0.05

```
.1 0 0 0      0         0       0       0       0       0       0.25
.5 0 0 0      0         0       0       0       0       0       0.5

0 .01 0 0     0         0       0       0       0       0.025   0
0 .1 0 0      0         0       0       0       0       0.2     0
0 .5 0 0      0         0       0       0       0       0.475   0

0 .01 0 0     0         0       0       0       0.05    0       0
0 .1 0 0      0         0       0       0       0.25    0       0
0 .5 0 0      0         0       0       0       0.475   0       0

.01 0 0 0     0.025     0       0       0       0       0       0
.1 0 0 0      0.125     0       0       0       0       0       0
.5 0 0 0      0.25      0       0       0       0       0       0

.01 0 0 0     0         0       0.0177  0       0       0       0
.1 0 0 0      0         0       0.1591  0       0       0       0
.5 0 0 0      0         0       0.3359  0       0       0       0

.01 0 0 0     0         0.005   0       0       0       0       0
.1 0 0 0      0         0.125   0       0       0       0       0
.5 0 0 0      0         0.25    0       0       0       0       0

.01 0 0 0     0         0       0       0.0354  0       0       0
.1 0 0 0      0         0       0       0.2121  0       0       0
.5 0 0 0      0         0       0       0.3182  0       0       0
```

The training set used to train a Kohonen net to recognise is identical to this, but contains no desired values.

Appendix M

Listing of the Backpropagation simulator code

```
/*******************************************/
/* BP Network Simulator 20th Nov 1989      */
/*                                         */
/* Features include                        */
/*          •uses a hard limiter           */
/*          •uses Fahlman's mod.           */
/*          •allows specification of a test */
/*           set different to training set */
/*          •global error output at regular */
/*           intervals                     */
/*          •aborting execution via ctrl-c */
/*           causes weights to be saved    */
/*          •training can be commenced from */
/*           a given set of weights        */
/*******************************************/

#include <stdio.h>
#include <math.h>

/* following "includes" needed to obtain system time */
/* which is needed to obtain random numbers          */

#include <sys/time.h>
#include <sys/types.h>
#include <sys/timeb.h>

/* following include needed for auto saving of weights on ctrl-c */
#include <signal.h>

/* max no. dendrites on any neuron */
#define max_links 30
int hidden;


#define eof -99
/* no. of input and output units */
#define num_input_units  2
#define num_output_units 1


/* declaration of structure of a neuron */
struct neuron { float output,error,bias;
                int num_ins, num_outs;
                /* synaptics weights */
                float weights[max_links];
                float last_change[max_links];
                float last_bias;
                /* dendrites */
                struct neuron *dendrites[max_links];
                /* axons */
                struct neuron *axons[max_links];
              };

/* declaration of the linked list into which the training */
/* set and test sets are loaded. */
struct vector_set { float value;
                    struct vector_set *ptr_next;
                  };
```

```
struct vector_set *train_head=NULL,
                  *test_head =NULL;


float eta, alpha;
char filename[30], filename2[30];

/* declare floating point functions */
float get_vectors();
float test_testset();

/* declare function called on ctrl-c signal */
save_and_exit();

/* declare input and output units */
struct neuron *ptr_output[num_output_units],
              *ptr_inputs[num_input_units];
/* declare target vector units */
struct neuron *ptr_desired[num_output_units];
/* declare 'corner' of square sigmoid */
float corner;


FILE *fp, *fopen(); /*file usage */

/*******************************/

main ()
    { int i,j,num_loops=0,num_patterns,plot_point=1,step_size=1,reply;
      float outbit,stop_value,orig_alpha,last_tot_error,tot_error=999;

      /* initialise random num generator */
      unsigned int seed;
      unsigned long t,time2;

      time2=time(&t); /* get elapsed system time in seconds */
      seed=(unsigned int)time2; /* set new seed */
      srand(seed);

      /* call save_and_exit if prog is terminated */
      signal(SIGINT,save_and_exit);

      /* input value for 'corner' of square sigmoid */
      printf ("Enter sigmoid approximation parameter. ");
      scanf  ("%f",&corner);

      /* input number of hidden units */
      printf ("Enter number of units in hidden layer. ");
      scanf  ("%d",&hidden);

      create_network();
      printf ("Restore weights (1/0)?\n");
      scanf  ("%d",&reply);
      if (reply==1) restore_all_weights();

      /* train network */
        printf ("Enter filename containing training set. ");
        scanf  ("%s",filename);
        printf ("Enter filename of test set. ");
        scanf  ("%s",filename2);
        printf ("Enter stopping criteria.  ");
        scanf  ("%f",&stop_value);
        printf ("Enter eta.   ");
        scanf  ("%f",&eta);
```

```
        printf ("Enter alpha.  ");
        scanf  ("%f",&orig_alpha);
        alpha=orig_alpha;
        printf ("\nTraining in progress...\n");

        load_train_set ();
        load_test_set  ();

        while (tot_error>stop_value) {
          last_tot_error=tot_error;
          tot_error=get_vectors();

          num_loops++;

          /* print out errors */
          if (num_loops==plot_point) {
            printf("After %d loops, error=%f,
eta=%f\n",num_loops,tot_error,eta);
            plot_point=num_loops+step_size;
            if ((float)(plot_point/10)==step_size) step_size *= 10;
          }/* endif */
        }/* end while */
      /* end train network */

      /* use net */
      /* first clear inputs */
      for (i=0; i<num_input_units; i++) ptr_inputs[i]->output=0;

      test_testset ();
      printf ("\nnumber of units in hidden layer = %d",hidden);
      printf ("\neta = %f\nalpha = %f",eta,alpha);
      printf ("\nstopping tolerance was = %f",stop_value);
      printf ("\nno. of training loops = %d",num_loops);
      printf ("\nfinal error is = %f\n",test_testset());

      /* save weights and exit */
      save_and_exit ();
    }/* end main */

/*********************************/

/*********************************/
/* Routines to save and restore */
/* weights.                     */
/*                              */
/*********************************/

save_node (ptr_node)
    struct neuron *ptr_node;
    { int i;

      /* first save bias */
      fprintf(fp,"%f\n",ptr_node->bias);

      for (i=0; i<ptr_node->num_ins; i++)
        fprintf(fp,"%f\n",ptr_node->weights[i]);

      /* output a blank line to sep weights of */
      /* next node */
      fprintf(fp,"\n");
    }/* end */

/*********************************/

save_layer (ptr_node)
```

```
    struct neuron *ptr_node;
    { int i;

      /* ptr_node must point to any node above layer to be saved */

      for (i=0; i<ptr_node->num_ins; i++)
        save_node(ptr_node->dendrites[i]);
    }/* end */

/*********************************/

save_all_weights()
    { int i;
      struct neuron *ptr_node;

      fp=fopen("weights4","w");

      /* save output units */
      for (i=0; i<num_output_units; i++)
        save_node(ptr_output[i]);

      /* pick a node from the output layer */
      ptr_node=ptr_output[0];

      /* while not input layer */
      while (ptr_node->dendrites[0]->num_ins !=0) {
        save_layer(ptr_node);
        ptr_node=ptr_node->dendrites[0]; /* move down a layer */
      }/* end while */
      fclose(fp);
    }/* end */

/*********************************/

restore_node (ptr_node)
    struct neuron *ptr_node;
    { int i;

      /* first restore bias */
      fscanf(fp,"%f\n",&(ptr_node)->bias);

      for (i=0; i<ptr_node->num_ins; i++)
        fscanf(fp,"%f\n",&(ptr_node)->weights[i]);
    }/* end */

/*********************************/

restore_layer (ptr_node)
    struct neuron *ptr_node;
    { int i;

      /* ptr_node must point to any node above layer to be saved */

      for (i=0; i<ptr_node->num_ins; i++)
        restore_node(ptr_node->dendrites[i]);
    }/* end */

/*********************************/

restore_all_weights()
    { int i;
      struct neuron *ptr_node;

      fp=fopen("weights4","r");
```

```
      for (i=0; i<num_output_units; i++)
        restore_node(ptr_output[i]);

      /* pick any output units */
      ptr_node=ptr_output[0];

      /* while not input layer */
      while (ptr_node->dendrites[0]->num_ins !=0) {
        restore_layer(ptr_node);
        ptr_node=ptr_node->dendrites[0]; /* move down a layer */
      }/* end while */
      fclose(fp);
    }/* end */

/*********************************/

float sigmoid(x)
    float x;
    {
      return (1/(1+exp(-x)));
    }/* end */

/*********************************/
float square_sigmoid(x)
    double x;
    { float a;                    /* -a & +a are the corners of the */
      a=corner;                   /* square threshold function.     */
      if (x>= a) return(1);              /* threshold at 1 */
      else if (x<=-a) return(0);         /* threshold at 0 */
           else return(0.5*(1+x/a));     /* linear func in between */
    }/* end */


/*********************************/

calculate_outputs (ptr_node)
/* recursively goes down network */

    struct neuron *ptr_node;
    { double sum=0;
      int i;

      for (i=0; i<ptr_node->num_ins; i++) {
        if (ptr_node->dendrites[i]->num_ins != 0)
          calculate_outputs(ptr_node->dendrites[i]);
        sum += ptr_node->dendrites[i]->output * ptr_node->weights[i];
      }/* end for */

      ptr_node->output = square_sigmoid (sum + ptr_node->bias);

    } /* end calculate_outputs. */

/*********************************/

float find_weight (ptr_node,axon)
    struct neuron *ptr_node,*axon;
/* finds the weight associated with a given axon */
    { int i;
      for (i=0; i<axon->num_ins; i++)
        if (ptr_node==axon->dendrites[i]) return (axon->weights[i]);
        /* nb: assumes that a match WILL be found. */
    }


/*********************************/
```

```
update_node (ptr_node)
/* back prop learning for weights associated with one node */
    struct neuron *ptr_node;
    { float delta,desired,x,y,weight_change,bias_change,sum=0;
      int i,k;
      float sigmoid_prime;  /* Fahlmans' modification   */

      /* special case. an output unit */
      if (ptr_node->num_outs==0)
        { y=ptr_node->output;
          desired=ptr_node->axons[0]->output;
          sigmoid_prime = y*(1-y);              /* Fahlman's mod    */
          sigmoid_prime += 0.1;                 /* add 0.1          */
          delta=sigmoid_prime*(desired-y);      /* Fahlman's mod    */
        }/* end if */


      /* hidden layer units */
      else { x=ptr_node->output;
             for (k=0; k<ptr_node->num_outs; k++)
               sum += ptr_node->axons[k]->error
                      * find_weight(ptr_node,ptr_node->axons[k]);
              sigmoid_prime = x*(1-x);          /* Fahlman's mod    */
              sigmoid_prime += 0.1;             /* add 0.1          */
              delta=sigmoid_prime*sum;          /* Fahlman's mod    */
            }/* end if */
      ptr_node->error=delta;

    /* now have worked out delta. need to adjust weights now */
      for (i=0; i<ptr_node->num_ins; i++) {
        weight_change = eta * delta * ptr_node->dendrites[i]->output +
                        alpha * ptr_node->last_change[i];
        /* update weights */
        ptr_node->weights[i] += weight_change;
        /* update last weight change */
        ptr_node->last_change[i] = weight_change;

      }/* end for */

      /* update bias for each node */
      bias_change = eta * delta * 1 + alpha * ptr_node->last_bias;
      ptr_node->bias += bias_change;
      ptr_node->last_bias = bias_change;

    }/* end */

/*********************************/

update_layer (ptr_node)
/* ptr_node points to node above layer of nodes to be updated */
    struct neuron *ptr_node;
    { int i;

    /* now  update weights on layer below. */
      for (i=0; i<ptr_node->num_ins; i++) {
        /* check if node below isn't an input node */
        if (ptr_node->dendrites[i]->num_ins !=0)
          /* now do each node below */
          update_node(ptr_node->dendrites[i]);
      }/* end for */
    }/* end /

/*********************************/
```

```
update_weights ()
/* should do back_prop on all nodes in correct order as well! */

    { struct neuron *ptr_node;
      int i;

      /* do output nodes first */
      for (i=0; i<num_output_units; i++)
        update_node(ptr_output[i]);

      /* pick any output node */
      ptr_node=ptr_output[0];

      /* now update layers below if not input layers */
      while (ptr_node->dendrites[0]->num_ins !=0) {
        update_layer(ptr_node);
        /* need any node above the next layer now */
        ptr_node=ptr_node->dendrites[0];
      }/* end while */
    }/* end */

/*********************************/

initialise_node (ptr_node)
    struct neuron *ptr_node;
    { int i;

      ptr_node->output=0;
      ptr_node->error=0;
      ptr_node->num_ins=0;
      ptr_node->num_outs=0;

      /* randomise weights */
      for (i=0; i<max_links; i++) {
        ptr_node->weights[i]=(float) rand() / 10000000000;
        ptr_node->bias      =(float) rand() / 10000000000;
        ptr_node->last_change[i]=0;
      }/* end for */
    }/* end */

/*********************************/

struct neuron *gen_new_node ()
    { struct neuron *ptr_node;

      ptr_node=(struct neuron *)malloc(sizeof(struct neuron));
      initialise_node (ptr_node);
      return (ptr_node);
    }

/*********************************/

create_network()
    { int i,j;
      struct neuron *ptr_hidden;

      /* Hand build network */

      /* output and desired vector nodes */
      for (i=0; i<num_output_units; i++) {
        ptr_desired[i]=gen_new_node();
        ptr_output[i]=gen_new_node();
        ptr_output[i]->num_outs=0;
        ptr_output[i]->num_ins=hidden;
        /* connect output to desired */
```

```
        ptr_output[i]->axons[0]=ptr_desired[i];
      }/* end for */

      /* input nodes */
      for (i=0; i<num_input_units; i++) {
        ptr_inputs[i]=gen_new_node();
        ptr_inputs[i]->num_outs=hidden;
        ptr_inputs[i]->num_ins=0;
      }/* end for */

      /* hidden layer */
      for (i=0; i<hidden; i++) {
        ptr_hidden=gen_new_node();
        ptr_hidden->num_ins=num_input_units;
        ptr_hidden->num_outs=num_output_units;

        /* connect dendrites to input units */
        for (j=0; j<num_input_units; j++)
          ptr_hidden->dendrites[j]=ptr_inputs[j];


        /* connect to output units */
        for (j=0; j<num_output_units; j++) {
          ptr_hidden->axons[j]=ptr_output[j];
          ptr_output[j]->dendrites[i]=ptr_hidden;
        }/* end for */

      }/* end for */

    }/* end */


/*********************************/

load_train_set ()
    { struct vector_set *ptr;
      float value;

      fp=fopen(filename,"r");
      ptr=train_head;

      /* get all data values from file */
      fscanf(fp,"%f",&value);
      while (value != eof) {
        if (train_head==NULL) {
        /* make first node */
          ptr=(struct vector_set *)(malloc(sizeof(struct
vector_set)));
          ptr->value=value;
          ptr->ptr_next=NULL;
          train_head=ptr;
        }
        else {
        /* make next node and move on pointer */
          ptr=ptr->ptr_next=(struct vector_set *)(malloc(sizeof(struct
vector_set)));
          ptr->ptr_next=NULL;
          ptr->value=value;
        }/* end if */
      fscanf(fp,"%f",&value);
      }/* end while */
    }/* end */

/*********************************/
load_test_set ()
```

```
    { struct vector_set *ptr;
      float value;

      fp=fopen(filename2,"r");
      ptr=test_head;

      /* get all data values from file */
      fscanf(fp,"%f",&value);
      while (value != eof) {
        if (test_head==NULL) {
        /* make first node */
          ptr=(struct vector_set *)(malloc(sizeof(struct
vector_set)));
          ptr->value=value;
          ptr->ptr_next=NULL;
          test_head=ptr;
        }
        else {
        /* make next node and move on pointer */
          ptr=ptr->ptr_next=(struct vector_set *)(malloc(sizeof(struct
vector_set)));
          ptr->ptr_next=NULL;
          ptr->value=value;
        }/* end if */
      fscanf(fp,"%f",&value);
      }/* end while */
    }/* end */

/*********************************/

ptr_scanf (ptr, data) /* pointer analogue of fscanf */
    struct vector_set **ptr;
    float *data;
    {
#ifdef DEBUG
printf("In ptr_scanf()\n");
#endif

      if (*ptr != NULL) {
      *data=(*ptr)->value;       /* get value */
      *ptr=(*ptr)->ptr_next;     /* advance pointer */
    }

#ifdef DEBUG
printf("Out off ptr_scanf()\n");
#endif

    }/* end */

/*********************************/

float test_testset () /* this routine is nearly identical to
                         get_vectors */
    { int i,j,count=0;
      float total=0,diff;
      struct vector_set *fptr;

      fptr=test_head;

      /* get desired values */
      for (j=0; j<num_output_units; j++)
        ptr_scanf(&fptr,&(ptr_desired[j])->output);

      /* stop loop if first desired value is eof */
      while (fptr != NULL) {
```

```
        for (i=0; i<num_input_units; i++)
          ptr_scanf(&fptr,&(ptr_inputs[i])->output);

        /* execute one cycle */
        for (i=0; i<num_output_units; i++) {
          calculate_outputs(ptr_output[i]);
          diff=ptr_desired[i]->output - ptr_output[i]->output;
          total += diff*diff;
        }/* end for */

        /* get desired values again */
        /* but also check for eof   */
        j=0;
        while (j<num_output_units) {
          ptr_scanf(&fptr,&(ptr_desired[j])->output);
          /* if first desired value = eof */
          if ((j==0) && (fptr == NULL))
            j=num_output_units; /* force loop to stop */
            else j++;
        }/* end while */

      }/* end while */
      total /= 2.0;
      return (total);
    }/* end */


/*********************************/

float get_vectors ()
    { int i,j;
      float diff,total;
      struct vector_set *fptr;

      fptr=train_head;

      /* get desired values */
      for (j=0; j<num_output_units; j++)
        ptr_scanf(&fptr,&(ptr_desired[j])->output);

      while (fptr != NULL) {
        for (i=0; i<num_input_units; i++)
          ptr_scanf(&fptr,&(ptr_inputs[i])->output);

        /* execute one cycle */
        /* get next set of outputs */
        for (i=0; i<num_output_units; i++) {
          calculate_outputs(ptr_output[i]);
          diff=ptr_desired[i]->output - ptr_output[i]->output;
          total += diff*diff;
        }/* end for */
        update_weights ();

        /* get desired values again */
        /* but also check for eof   */
        j=0;
        while (j<num_output_units) {
          ptr_scanf(&fptr,&(ptr_desired[j])->output);
          if ((j==0) && (fptr == NULL))
            j=num_output_units; /* force loop to stop */
          else j++;
        }/* end while */

      }/* end while */
```

```
      return(test_testset());
    }/* end */

/*********************************/

use_network ()
    { int i,reply=1;     .

      while (reply) {
        printf("Try net again? (1/0) ");
        scanf("%d",&reply);

        /* interactively enter inputs. */

        if (reply) {
          for (i=0; i<num_input_units; i++) {
            printf("Enter input %d ",i);
            scanf("%f",&(ptr_inputs[i])->output);
          }/* end for */
          for (i=0; i<num_output_units; i++) {
            calculate_outputs(ptr_output[i]);
            printf("\nOutput from the network is %f\n",ptr_output[i]->
output);
          }/* end for */

        }/* end if */
      }/* end while */
    }/* end */



/*********************************/

save_and_exit()
    {
      save_all_weights();
      exit(0);
    }

/*********************************/
```

Listing of the Kohonen simulator code

```
/*******************************************/
/* Kohonen Simulator    17th Jan 1990      */
/*                                         */
/*******************************************/

/* Kohonen (1988) section 5.4.1 says that   */
/* the demonstrations presented, have a     *
/* gain term that is "also a function of    */
/* bubble radius, a Gaussian function with  */
/* a width that was decreasing in time.     */
/* This version is similar to kohonen.c, but*/
/* this time the magnitude of the gain term */
/* decreases with time.                     */

#include <stdio.h>
#include <math.h>

/* following "includes" needed to obtain system time */
/* which is needed to obtain random numbers          */

#include <sys/time.h>
#include <sys/types.h>
#include <sys/timeb.h>

/* following include needed for auto saving of weights on ctrl-c */
#include <signal.h>

#define eof -99
#define num_input_units  7
#define net_size 12

/* declaration of structure of a neuron */
struct neuron { float output;
                float weights[num_input_units];
              };

/*declaration of the linked list into which the training */
/* set is loaded */
struct vector_set { float value;
                    struct vector_set *ptr_next;
                  };

struct vector_set *train_head=NULL;

/* declare function called on ctrl-c signal */
save_and_exit ();

/* declare functions used by kohonen algorithm */
float gain_term ();
float neighbourhood ();

/* declare input and output units */
struct neuron *ptr_outputs[net_size][net_size],
              *ptr_inputs[num_input_units];

FILE *fp, *fopen(); /*file usage */
char filename[30];

int     tmax;      /* t is effectively the */
                   /* no. of epochs.       */
```

```
float   b;          /* b the gaussian height, i.e. the gain   */
float   excitation_factor;
                    /* width of gaussian equal to the width   */
                    /* of the neighbourhood. excitation_factor*/
                    /* is proportion of width in which        */
                    /* excitation takes place                 */


/***********************************************/

/* The MAIN routine */

main ()
    { int i,x,y,t,norm,restore;
      struct vector_set *fptr;

      /* seed random number generator */
      unsigned int seed;
      unsigned long t1,time2;

      time2=time(&t1);
      seed=(unsigned int)time2;
      srand (seed);

      /* call save_and_exit if prog is terminated */
      signal(SIGINT,save_and_exit);

      printf ("Kohonen_3.exe\n\n");

      printf ("Enter filename         : ");
      scanf  ("%s", filename);

      printf ("Enter Tmax             : ");
      scanf  ("%d", &tmax);

      printf ("Enter initial gain     : ");
      scanf  ("%f", &b);

      printf ("Enter excitation factor\n");
      printf ("ranging between 0-1    : ");
      scanf  ("%f",&excitation_factor);

      printf ("Normalize inputs (0/1) : ");
      scanf  ("%d", &norm);

      printf ("Restore weights  (0/1) : ");
      scanf  ("%d", &restore);

      create_network ();   /* this needs to normalize the weights */
                           /* as each neuron is created.          */

      load_train_set ();
      if (restore) restore_layer ();
      if (norm) normalize_training_set ();

      for (t=0; t<tmax; t++) {

        fptr=train_head;
        while (fptr != NULL) {
          for (i=0; i<num_input_units; i++)
            ptr_scanf(&fptr, &(ptr_inputs[i])->output);
          find_winning_neuron (&x,&y);
          update_weights (x,y,t);
        }/* end while */
```

```
      }/* end for */

     save_and_exit ();

    }/* end */

/*********************************/
/* Routines to save and restore */
/* weights.                     */
/*                              */
/* start    03rd Jan 1989       */
/* updated  04th Jan 1990       */
/*          for K-net simulator */
/*********************************/

save_node (ptr_node)
    struct neuron *ptr_node;
    { int i;

      for (i=0; i<num_input_units; i++)
        fprintf(fp,"%f\n",ptr_node->weights[i]);

      /* output a blank line to sep weights of */
     /* next node */
      fprintf(fp,"\n");
    }/* end */

/*********************************/

/* saves weights of all nodes from left to right */
/* top to bottom                                 */

save_layer ()
    { int i,j;

      fp=fopen("k_weights","w");

      for (j=0; j<net_size; j++)
        for (i=0; i<net_size; i++)
          save_node (ptr_outputs[i][j]);

      fclose (fp);

    }/* end */

/*********************************/

restore_node (ptr_node)
    struct neuron *ptr_node;
    { int i;

      for (i=0; i<num_input_units; i++)
        fscanf(fp,"%f\n",&(ptr_node)->weights[i]);

    }/* end */

/*********************************/

restore_layer ()
    { int i,j;

      fp=fopen("k_weights","r");

      for (j=0; j<net_size; j++)
        for (i=0; i<net_size; i++)
```

```
          restore_node (ptr_outputs[i][j]);

      fclose (fp);

    }/* end */

/*********************************/

float sigmoid(x)
    float x;
    {
      return (1/(1+exp(-x)));
    }/* end */

/*********************************/

calculate_outputs ()
    { double sum=0;
      int i,j,k;

      /* replaces the output of each neuron, with the */
      /* thresholded weighted sum at each node.       */
      for (j=0; j<net_size; j++)
        for (i=0; i<net_size; i++) {

          for (k=0; k<num_input_units; k++)
            sum += ptr_outputs[i][j]->weights[k] * ptr_inputs[k]-
>output;
          ptr_outputs[i][j]->output = sigmoid (sum);
          sum=0;

       }/* end fors */

    }/* end calculate_outputs. */

/*********************************/

initialise_node (ptr_node)
    struct neuron *ptr_node;
    { int i;

      ptr_node->output=0;

      /* randomise weights */
      for (i=0; i<num_input_units; i++)
        ptr_node->weights[i]=(float) rand() / 10000000000;

    }/* end */

/*********************************/

struct neuron *gen_new_node ()
    { struct neuron *ptr_node;

      ptr_node=(struct neuron *)malloc(sizeof(struct neuron));
      initialise_node (ptr_node);
      return (ptr_node);
    }

/*********************************/

create_network ()
    { int i,j,k;

      /* first build input layer */
```

```
      for (i=0; i<num_input_units; i++)
        ptr_inputs[i]=gen_new_node ();

      /* now build square output layer */
      for (j=0; j<net_size; j++)
        for (i=0; i<net_size; i++) {
          ptr_outputs[i][j]=gen_new_node ();

          normalize_weights (ptr_outputs[i][j]);

        }/* end fors */

    }/* end */

/*********************************/

load_train_set ()
    { struct vector_set *ptr;
      float value;

      fp=fopen(filename,"r");
      ptr=train_head;

      /* get all data values from file */
      fscanf(fp,"%f",&value);
      while (value != eof) {
        if (train_head==NULL) {
        /* make first node */
          ptr=(struct vector_set *)(malloc(sizeof(struct
vector_set)));
          ptr->value=value;
          ptr->ptr_next=NULL;
          train_head=ptr;
        }
        else {
        /* make next node and move on pointer */
          ptr=ptr->ptr_next=(struct vector_set *)(malloc(sizeof(struct
vector_set)));
          ptr->ptr_next=NULL;
          ptr->value=value;
        }/* end if */
      fscanf(fp,"%f",&value);
      }/* end while */

   }/* end */

/*********************************/

ptr_scanf (ptr, data) /* pointer analogue of fscanf */
    struct vector_set **ptr;
    float *data;


    { if (*ptr != NULL) {
      *data=(*ptr)->value;      /* get value */
      *ptr=(*ptr)->ptr_next;    /* advance pointer */
      }/* end if */

    }/* end */


/*********************************/

save_and_exit ()
    {
```

```
      save_layer ();
      exit(0);
    }

/************************************/

/* Normalization functions            */
/* Euclidean distances are calculated */

normalize_weights (ptr_node)
    struct neuron *ptr_node;
    { int i;
      float root;
      double sum=0;

      /* calculate denominator */
      for (i=0; i<num_input_units; i++)
        sum += ptr_node->weights[i] * ptr_node->weights[i];

      root = (float) sqrt (sum);

     /* now divide each weight by the root */
     for (i=0; i<num_input_units; i++)
       ptr_node->weights[i] /= root;

    }/* end */

/***************************************/

normalize_training_set ()
    { int i;
      float root, input;
      double sum;
      struct vector_set *fptr, *ptr_temp[num_input_units];

      fptr=train_head;

      while (fptr != NULL) {

      /* Calculate Denominator                        */
      /* an array of pointers keeps track of where    */
      /* the values to be changed are.                */

      sum=0;
      for (i=0; i<num_input_units; i++) {
        ptr_temp[i]=fptr;
        ptr_scanf (&fptr, &input);
        sum += input*input;
      }/* end for */
      root = (float) sqrt (sum);

      /* Now Divide each input by this Root           */
      /* the 'values' to be changed are accessed      */
      /* by array of pointers.                        */

      for (i=0; i<num_input_units; i++)
        if (root>0.0) ptr_temp[i]->value /= root;

      }/* end while */
    }/* end */

/***************************************/

normalize_inputs ()
    { int i;
```

```
      float root;
      double sum=0;

      /* calculate denominator */
      for (i=0; i<num_input_units; i++)
        sum += ptr_inputs[i]->output * ptr_inputs[i]->output;

      root = (float) sqrt(sum);

      /* now divide each input by the root */
      for (i=0; i<num_input_units; i++)
        if (root>0.0) ptr_inputs[i]->output /= root;

    }/* end */

/*************************************/

/* routine to find the winning neuron */
/* in the net. */

find_winning_neuron (x,y)
    int *x, *y;

    { float difference, distance, min_distance=9999999;
      int i,j,k;

      for (j=0; j<net_size; j++)
        for (i=0; i<net_size; i++) {

          /* calculate distance to i, jth neuron */
          distance=0;
          for (k=0; k<num_input_units; k++) {
            difference = ptr_inputs[k]->output - ptr_outputs[i][j]-
>weights[k];
            distance += difference * difference;
          } /* end for */

          /* update if this is the closest */
          if (distance < min_distance) {
            min_distance = distance;
            *x = i; /* pass location back */
            *y = j;
          }/* end if */

        }/* end fors */
    }/* end */


/*****************************************/

/* routine to update the weights of the winning neuron */
/* as well as those in the current neighbourhood.      */

update_weights (x,y,t)
    int x,y,t;

    { int i,j,k,left,right,top,bottom,neighbourhood_radius,gap;

      /* work out the max neighbourhood size at time t */
      neighbourhood_radius = (int) neighbourhood (t);

      /* work out the edges of the neighbourhood */
      left   = x - neighbourhood_radius;
      right  = x + neighbourhood_radius;
      top    = y - neighbourhood_radius;
```

```
      bottom = y + neighbourhood_radius;

      /* check for boundary conditions */
      if (left < 0)           left   = 0;
      if (right >= net_size)  right = net_size;
      if (top < 0)            top    = 0;
      if (bottom >= net_size) bottom= net_size;

      /* update weights */

      for (j=top; j<bottom; j++)
        for (i=left; i<right; i++) {

          /* distance between chosen neuron x,y and any other neuron
i,j */
          /* is used to get correct gain value at that point. */
          /* find gap between neuron i,j and neuron x,y */
          gap = abs (x-i); /* gap across columns */
          /* neuron may be in the same column, so check gap across
rows */
          if (abs(y-j) > gap) gap = abs (y-j);

            /* now update each weight on this neuron. */
            /* delta_weight=(input-weight)*gain       */
           for (k=0; k<num_input_units; k++)
              ptr_outputs[i][j]->weights[k] +=
                (ptr_inputs[k]->output - ptr_outputs[i][j]-
>weights[k])
                * gain_term (gap,t);
        }/* end for i,j */

    }/* end */


/************************************************/
/* this is the approximated gaussian            */

float gain_term (distance,t)
    int distance,t;

    { int excite;
      float inhibit, gauss_height;

      gauss_height = b*(1 - (float)t/(float)tmax);
      inhibit      = neighbourhood (t);
      excite       = inhibit * excitation_factor;

      /* up to distance=excite, gain is excitatory, then  */
      /* between excite and inhibit, it is inhibitory.    */
      if (distance <= excite)
           return (gauss_height);    /* excitatory */
      else return (-gauss_height/3); /* inhibitory */

    }/* end */

/************************************************/

float neighbourhood (time)
    int time;

    { /* neighbourhood decreases linearly with time. */
      /* and becomes zero, when t=tmax.              */

      /* need to 'float' times to avoid int division */
```

```
      return ( (1 - (float)time/(float)tmax) * net_size / 2 );
    } /* end */

/*************************************************/

use_network ()
    { int i,j, reply=1,norm;
      float value;

      printf ("Normalize inputs (0/1) : ");
      scanf  ("%d",&norm);

      while (reply) {

        /* get pattern to be tested */
        for (i=0; i<num_input_units; i++) {
          printf ("Enter input %d : ",i);
          scanf  ("%f",&value);
          ptr_inputs[i]->output = value;
        }/* end */
        if (norm) normalize_inputs ();

        /* calculate and print activation of each neuron */
        /* in the output layer. */
        calculate_outputs ();

       for (j=0; j<net_size; j++) {
         for (i=0; i<net_size; i++)
            printf ("%f  ",ptr_outputs[i][j]->output );
         printf ("\n");  /* get ready to print next row. */
       }/* end for */


       printf ("\nTry net again (1/0)? ");
       scanf  ("%d", &reply);

      }/* end while */

    }/* end */
```

Listing of the Perceptual Network Simulator

```
/*****************************************/
/* Perceptual Network Simulator          */
/*                    05th Feb  1990     */
/*****************************************/

#include <stdio.h>
#include <math.h>

FILE *fp, *fopen();             /* file usage                          */
                                /* files of weights to be used         */

/* when using different weight files...remember to change network    */
/* parametres (inputs, hidden_units etc) at the point of the         */
/* function call.                                                    */

#define distance_weights        "weights_10hid_0.00005"
#define find_bigger_weights     "weights_6hid_0.000001_a2"
#define find_lower_weights      "weights_low2b_18hid_0.000000_a2"
#define corner_detect_weights   "weights_7hid_corners5_a1_0.000001"
#define l1_l2_weights           "weights_8hid_l1_l2_bigger_0-
1_0.000035"

#define max_links 21            /* max no. dendrites on any neuron   */
#define inhibitory_weight -25.0 /* weight for inhibitory links       */

typedef enum {weighted_sum, weighted_product} neuron_type;
typedef enum {calculated, not_calculated} neuron_processed;
typedef enum {no_effect, add_1, negate, threshold}
pre_processing_class;
typedef enum {no_activation, smooth_sigmoid, step, linear_1, linear_2}
sigmoid_class;

typedef struct neuron
        { float output, bias;
          int   num_inputs;         /* variable ins, but only 1 out */
          neuron_processed status;  /* flag whether a node has been */
          float weights [max_links];/* calculated or not.           */

          struct neuron *dendrites[max_links];
          neuron_type   neuron_class; /* i.e. weighted sum,  or
product */
          sigmoid_class sigmoid;                /* used to decided
which activation */
          pre_processing_class pre_processing;  /* func, to use and
what to do to   */
        } neuron;                               /* the inputs before
processing     */

typedef struct network
        { int num_inputs, num_outputs;
          neuron *ptr_outputs[max_links], *ptr_inputs [max_links];
        } network;

typedef char filename[40];

/* function declarations */
network *create_parallel_net (), *create_corner_net ();



/********** MAIN ****************************/
```

```
main ()
    { network *ptr_corner_net, *ptr_parallel_net;
      filename input_file;
      int i;

      printf ("Creating network....\n");
      ptr_parallel_net  = create_parallel_net ();
      ptr_corner_net    = create_corner_net (ptr_parallel_net);

      printf ("-------------------------------------------------------
--------------------\n");
      printf ("      |          --         /         \\        acute
obtuse     right\n");
      printf ("-------------------------------------------------------
--------------------\n");


      printf ("Enter filename or exit.... ");
      scanf  ("%s", input_file);

      while (strcmp(input_file, "exit"))
      {
        load_inputs_from_file (ptr_parallel_net, input_file);

        use_network (ptr_parallel_net);

        use_network (ptr_corner_net);
        printf("\n");

        printf ("Enter filename or exit.... ");
        scanf  ("%s", input_file);

      }/* end while */

    }/* end */

/*********** END MAIN **********************/

restore_node (ptr_node)
    neuron *ptr_node;
    { int i;

      /* first restore bias */
      fscanf(fp,"%f\n",&(ptr_node)->bias);

      for (i=0; i<ptr_node->num_inputs; i++)
        fscanf(fp,"%f\n",&(ptr_node)->weights[i]);

    }/* end */

/*****************************************/

restore_network_weights (ptr_network, weights_file)
    network *ptr_network;
    filename weights_file;

    { neuron *ptr_node;
      int i;

      /* have this as a parameter? */
      fp=fopen (weights_file,"r");

      /* get output unit weights */
      for (i=0; i<ptr_network->num_outputs; i++)
        restore_node(ptr_network->ptr_outputs[i]);
```

```
      /* get hidden unit weights */
      for (i=0; i<ptr_network->ptr_outputs[0]->num_inputs; i++)
        restore_node(ptr_network->ptr_outputs[0]->dendrites[i]);

      fclose(fp);

    }/* end */

/*****************************************/

network *gen_network_unit (inputs, outputs)
    int inputs, outputs;

    { network *ptr_node;

#ifdef DEBUG
printf ("In gen_network_unit\n");
#endif

      ptr_node=(network *)malloc (sizeof(network));

      /* initialise network */
      ptr_node->num_inputs =inputs;
      ptr_node->num_outputs=outputs;

#ifdef DEBUG
printf ("Out of gen_network_unit\n");
#endif

      return (ptr_node);
    }/* end */

/*****************************************/

neuron *gen_neuron_unit (inputs, sig_class, pre_proc_class,
neuron_proc_class)
    int                     inputs;
    sigmoid_class           sig_class;
    pre_processing_class    pre_proc_class;
    neuron_type             neuron_proc_class;

    { neuron *ptr_node;
      int i;

#ifdef DEBUG
printf ("In gen_neuron_unit\n");
#endif

      ptr_node=(neuron *)malloc (sizeof(neuron));

      /*initialise neuron */
      ptr_node->num_inputs          =inputs;
      ptr_node->sigmoid             =sig_class;
      ptr_node->pre_processing      =pre_proc_class;
      ptr_node->status              =not_calculated;
      ptr_node->output              =0.0;
      ptr_node->neuron_class        =neuron_proc_class;

      /* default value for weights is 1 */
      for (i=0; i<max_links; i++)
        ptr_node->weights[i]=1;

      /* default value for bias is 0 */
      ptr_node->bias=0;
```

```
#ifdef DEBUG
printf ("Out of gen_neuron_unit\n");
#endif

      return(ptr_node);
     }/* end */

/*****************************************/

/* the following network generates a 16 input, 1 output */
/* 0 hidden layer net, with fixed weights -1, and bias  */
/* +1.5. Its output will be used to inhibit 1/s and     */
/* 11_12 net outputs when no. of inputs is 0 or 1       */

/* nb. to use the output of this neuron as an excitatory*/
/* link, change the weights to +1, and the bais to -1.5 */

network *create_count_detector ()
    { int i;
      float weight= -1, bias= 1.5;
      network *ptr_network;

      ptr_network=gen_network_unit(16,1);

      /* inputs */
      for (i=0; i<16; i++)
        ptr_network-
>ptr_inputs[i]=gen_neuron_unit(1,no_activation,threshold,weighted_sum)
;

      /* output and bias    */
      ptr_network-
>ptr_outputs[0]=gen_neuron_unit(16,step,no_effect,weighted_sum);
      ptr_network->ptr_outputs[0]->bias=bias;

      /* wire up inputs to output   */
      /* and set weights            */
      for (i=0; i<16; i++)
      {
        ptr_network->ptr_outputs[0]->dendrites[i]=ptr_network-
>ptr_inputs[i];
        ptr_network->ptr_outputs[0]->weights[i] = weight;
      }/* end for */

      return (ptr_network);
    }/* end */

/*************************************************************/
/* To add up the sig. of acute, obtuse and right angled      */
/* corners, a 21 input 1 output net, having no hidden units */
/* is needed. The weights are all +1, there is no bias, and */
/* no activation function.                                   */
/*************************************************************/

network *create_adder ()
    { network *ptr_network;
      int i, num_inputs=21;

      ptr_network=gen_network_unit(num_inputs,1);
      /* inputs */
      for (i=0; i<num_inputs; i++)
        ptr_network-
>ptr_inputs[i]=gen_neuron_unit(1,no_activation,no_effect,weighted_sum)
;
```

```
      /* output */
      ptr_network-
>ptr_outputs[0]=gen_neuron_unit(num_inputs,no_activation,no_effect,wei
ghted_sum);

      /* wire up dendrites of output neuron to inputs */
      for (i=0; i<num_inputs; i++)
        ptr_network->ptr_outputs[0]->dendrites[i] =ptr_network-
>ptr_inputs[i];

      return(ptr_network);
    }/* create_adder */



/*****************************************/

network *create_network_from_neurons
    (inputs, hidden, outputs, weights_file, sig_class, pre_proc_class)

    int inputs, hidden, outputs;
    sigmoid_class sig_class;
    pre_processing_class pre_proc_class;
    filename weights_file;

    { network *ptr_network;
      neuron  *ptr_hidden;
      int i,j;


#ifdef DEBUG
printf ("In create_network_from_neurons\n");
#endif


      ptr_network=gen_network_unit (inputs, outputs);

      /* wire up output units */
      for (i=0; i<outputs; i++)
        ptr_network->ptr_outputs[i]=gen_neuron_unit (hidden,
sig_class, no_effect, weighted_sum);

      /* wire up input units */
      /* inputs have only 1 dendrite,    */
      /* and require no thresholding.    */
      for (i=0; i<inputs; i++)
        ptr_network->ptr_inputs[i]=gen_neuron_unit
(1,no_activation,pre_proc_class,weighted_sum);


      /* wire up hidden units */
      for (i=0; i<hidden; i++) {
        ptr_hidden=gen_neuron_unit (inputs,sig_class,
no_effect,weighted_sum);

        /* connect dendrites to input units */
        for (j=0; j<inputs; j++)
          ptr_hidden->dendrites[j]=ptr_network->ptr_inputs[j];

        /* connect to output units */
        for (j=0; j<outputs; j++)
          ptr_network->ptr_outputs[j]->dendrites[i]=ptr_hidden;

      }/* end for */
```

```
      restore_network_weights (ptr_network,weights_file);

#ifdef DEBUG
printf ("Out of create_network_from_neurons\n");
#endif

      return (ptr_network);
    }/* end */

/*****************************************/

create_part_filter_net
    (ptr_output_neuron, ptr_input_neurons, inputs, hidden, outputs,
weights_file, sig_class, pre_proc_class)

    neuron *ptr_output_neuron,
           *ptr_input_neurons[4];

    int inputs, hidden, outputs;
    sigmoid_class sig_class;
    pre_processing_class pre_proc_class;
    filename weights_file;

    { network *ptr_root, *ptr_left, *ptr_right;

#ifdef DEBUG
printf ("In create_part_filter_net\n");
#endif

      /* generate 3 network units */
      ptr_root =create_network_from_neurons (inputs, hidden, outputs,
weights_file, sig_class, pre_proc_class);
      ptr_left =create_network_from_neurons (inputs, hidden, outputs,
weights_file, sig_class, pre_proc_class);
      ptr_right=create_network_from_neurons (inputs, hidden, outputs,
weights_file, sig_class, pre_proc_class);

      /* top neuron must be connected to some other neuron above */
      /* generally output layers have only 1 neuron, hence [0]   */
      ptr_output_neuron->dendrites[0]=ptr_root->ptr_outputs[0];

      /* likewise for left descendent */
      ptr_root->ptr_inputs[0]->dendrites[0]=ptr_left->ptr_outputs[0];

      /* and right descendent */
      ptr_root->ptr_inputs[1]->dendrites[0]=ptr_right->ptr_outputs[0];

      /* store pointers to the 4 available input neurons */
      ptr_input_neurons[0]=ptr_left->ptr_inputs[0];
      ptr_input_neurons[1]=ptr_left->ptr_inputs[1];
      ptr_input_neurons[2]=ptr_right->ptr_inputs[0];
      ptr_input_neurons[3]=ptr_right->ptr_inputs[1];

#ifdef DEBUG
printf ("Out of create_part_filter_net\n");
#endif

    }/* end */

/*****************************************/

network *create_filter_net
    (inputs, hidden, outputs, weights_file, sig_class, pre_proc_class)
```

```
    int inputs, hidden, outputs;
    sigmoid_class sig_class;
    pre_processing_class pre_proc_class;
    filename weights_file;

    { network *ptr_network;
      neuron *ptr_neurons[4],
             *ptr_input_neurons[4],
             *dummy;
      int num_input_units=16,
          num_output_units=1,
          i,j;

#ifdef DEBUG
printf ("In create_filter_net\n");
#endif

      dummy=gen_neuron_unit (1,0,0,weighted_sum);
      ptr_network=gen_network_unit (num_input_units,
num_output_units);

      /* create top two levels */
      create_part_filter_net

(dummy,ptr_neurons,inputs,hidden,outputs,weights_file,sig_class,pre_pr
oc_class);
      ptr_network->ptr_outputs[0]=dummy->dendrites[0];

      /* create bottom two levels */
      for (i=0; i<4; i++) {
        create_part_filter_net

(ptr_neurons[i],ptr_input_neurons,inputs,hidden,outputs,weights_file,s
ig_class,pre_proc_class);
        for (j=0; j<4; j++)
          ptr_network->ptr_inputs[i*4+j]=ptr_input_neurons[j];
      }/* end for */

      /* number of dendrites on the input neurons must be zero */
      for (i=0; i<num_input_units; i++)
        ptr_network->ptr_inputs[i]->num_inputs=0;

#ifdef DEBUG
printf ("Out of create_filter_net\n");
#endif

      return (ptr_network);
    }/* end */

/*****************************************/

network *create_l1_l2_net ()
    { network *ptr_network, *ptr_bigger_than,
              *ptr_less_than, *ptr_l1_l2;
      int i;

#ifdef DEBUG
printf ("In create_l1_l2_net\n");
#endif

      ptr_network=gen_network_unit(16,1);

      ptr_less_than   =create_filter_net
(2,18,1,find_lower_weights,linear_2,negate);
      ptr_bigger_than =create_filter_net
```

```
(2,6,1,find_bigger_weights,linear_2,no_effect);
      ptr_l1_l2       =create_network_from_neurons
(2,8,1,l1_l2_weights,smooth_sigmoid,no_effect);

      /* overall inputs will come from ptr_bigger_than  */
      /* wire up input units from ptr_less_than         */
      /* to point to input units in ptr_bigger_than     */

      for (i=0; i<ptr_bigger_than->num_inputs; i++) {
        ptr_less_than->ptr_inputs[i]->dendrites[0]=ptr_bigger_than-
>ptr_inputs[i];
        ptr_less_than->ptr_inputs[i]->num_inputs=1;
      }/* end for */

      /* wire up the l1_l2 net  */
      /* l1_l2 inputs           */
      /* remember, 1st input=   */
      /* biggest value          */

      ptr_l1_l2->ptr_inputs[0]->dendrites[0]=ptr_bigger_than-
>ptr_outputs[0];
      ptr_l1_l2->ptr_inputs[1]->dendrites[0]=ptr_less_than-
>ptr_outputs[0];

      /* insert this whole network into ptr_network */

      /* inputs */
      for (i=0; i<ptr_network->num_inputs; i++)
        ptr_network->ptr_inputs[i]=ptr_bigger_than->ptr_inputs[i];

      /* output */
      ptr_network->ptr_outputs[0]=ptr_l1_l2->ptr_outputs[0];

#ifdef DEBUG
printf ("Out of create_l1_l2_net\n");
#endif

      return(ptr_network);

    }/* end */

/*****************************************/

network *create_single_orientation_net (num_inputs, num_outputs)
    int num_inputs, num_outputs;

    { network *ptr_network, *ptr_l1_l2, *ptr_distance, *ptr_counter;
      int i;

      ptr_network   =gen_network_unit (num_inputs, num_outputs);

      ptr_counter   =create_count_detector ();
      ptr_l1_l2     =create_l1_l2_net ();
      ptr_distance  =create_network_from_neurons

(16,10,1,distance_weights,smooth_sigmoid,threshold);

      /* create inhibitory link into ptr_l1_l2  */
      ptr_l1_l2->ptr_outputs[0]->num_inputs++;
      i= ptr_l1_l2->ptr_outputs[0]->num_inputs-1;
      ptr_l1_l2->ptr_outputs[0]->weights[i]= -20.0;
      ptr_l1_l2->ptr_outputs[0]->dendrites[i]=ptr_counter-
>ptr_outputs[0];

      /* create inhibitory link into ptr_distance   */
```

```
      ptr_distance->ptr_outputs[0]->num_inputs++;
      i= ptr_distance->ptr_outputs[0]->num_inputs-1;
      ptr_distance->ptr_outputs[0]->weights[i]= inhibitory_weight;
      ptr_distance->ptr_outputs[0]->dendrites[i]=ptr_counter-
>ptr_outputs[0];

      /* inputs to ptr_distance come from ptr_l1_l2 inputs */
      for (i=0; i<ptr_l1_l2->num_inputs; i++)
        ptr_distance->ptr_inputs[i]->dendrites[0]=ptr_l1_l2-
>ptr_inputs[i];

      /* inputs to ptr_counter also come from ptr_l1_l2 inputs */
      for (i=0; i<ptr_l1_l2->num_inputs; i++)
        ptr_counter->ptr_inputs[i]->dendrites[0]=ptr_l1_l2-
>ptr_inputs[i];

      /* insert these three nets into ptr_network       */
      /* inputs */
      for (i=0; i<ptr_network->num_inputs; i++)
        ptr_network->ptr_inputs[i]=ptr_l1_l2->ptr_inputs[i];

      /* outputs */
      ptr_network->ptr_outputs[0]=ptr_distance->ptr_outputs[0];
      ptr_network->ptr_outputs[1]=ptr_l1_l2->ptr_outputs[0];

      return (ptr_network);

    }/* end */

/*****************************************/

network *create_parallel_net ()
    { network *ptr_network, *ptr_1_orient;
      int i,j, num_orients=4, input_size=16, inputs_per_mltplr=2;
      neuron  *ptr_multipliers[4];  /* used to multiply together 1/s
and l1_l2 */


      ptr_network=gen_network_unit (input_size*num_orients,
num_orients);


      for (i=0; i<num_orients; i++) {
        ptr_1_orient=create_single_orientation_net (input_size,
inputs_per_mltplr);

        /* generate multipliers. note: weights and bais set to  */
        /* 1.0 and 0.0 resp. by default.                        */
        ptr_multipliers[i]=gen_neuron_unit
(inputs_per_mltplr,no_activation,no_effect,weighted_product);

        /* wire up inputs       */
        for (j=0; j<ptr_1_orient->num_inputs; j++)
          ptr_network->ptr_inputs[ptr_1_orient->num_inputs*i +
j]=ptr_1_orient->ptr_inputs[j];

        /* wire up multipliers. */
        for (j=0; j<2; j++) ptr_multipliers[i]-
>dendrites[j]=ptr_1_orient->ptr_outputs[j];

        /* wire up output       */
        ptr_network->ptr_outputs[i]=ptr_multipliers[i];

      }/* end for */
      return (ptr_network);
```

```
    }/* end */

/*****************************************/

network *create_single_rf_net ()
    { network *ptr_network, *ptr_single_rf_net, *ptr_corner_detector;
      neuron  *ptr_acute, *ptr_obtuse, *ptr_right;
      int i, num_inputs=16, num_outputs=3;

      ptr_network=gen_network_unit (num_inputs, num_outputs);

      ptr_single_rf_net=create_l1_l2_net ();
      ptr_corner_detector=create_network_from_neurons

(16,7,3,corner_detect_weights,linear_1,threshold);

      /* create three 'AND' gates, one each for                    */
      /* acute, obtuse and right angled corners.                   */
      /* each has 2 inputs, no thresholding, performs              */
      /* weighted_product and doesn't pre process inputs.          */
      /* by default, weights are=1, and bias=0.                    */

      ptr_acute =gen_neuron_unit
(2,no_activation,no_effect,weighted_product);
      ptr_obtuse=gen_neuron_unit
(2,no_activation,no_effect,weighted_product);
      ptr_right =gen_neuron_unit
(2,no_activation,no_effect,weighted_product);

      /* wire up the inputs on these, one goes to corner_detector  */
      /* and the other to the single_rf_net.                       */

      ptr_acute ->dendrites[0]=ptr_corner_detector->ptr_outputs[0];
      ptr_obtuse->dendrites[0]=ptr_corner_detector->ptr_outputs[1];
      ptr_right ->dendrites[0]=ptr_corner_detector->ptr_outputs[2];
      ptr_acute ->dendrites[1]=ptr_single_rf_net  ->ptr_outputs[0];
      ptr_obtuse->dendrites[1]=ptr_single_rf_net  ->ptr_outputs[0];
      ptr_right ->dendrites[1]=ptr_single_rf_net  ->ptr_outputs[0];

      /* inputs to corner_detector come from inputs to single_rf   */

      for (i=0; i<ptr_single_rf_net->num_inputs; i++)
        ptr_corner_detector->ptr_inputs[i]-
>dendrites[0]=ptr_single_rf_net->ptr_inputs[i];

      /* because single_rf_net is constructed from filter_net      */
      /* input units in ptr_single_rf_net have num_inputs=0        */
      /* this will cause problems when connecting to               */
      /* the parallel net. this value must be set to 1.            */

      for (i=0; i<ptr_single_rf_net->num_inputs; i++)
        ptr_single_rf_net->ptr_inputs[i]->num_inputs=1;

      /* insert this net into ptr_network       */
      /* inputs */
      for (i=0; i<ptr_network->num_inputs; i++)
        ptr_network->ptr_inputs[i]=ptr_single_rf_net->ptr_inputs[i];

      /* outputs    */
      ptr_network->ptr_outputs[0]=ptr_acute;
      ptr_network->ptr_outputs[1]=ptr_obtuse;
      ptr_network->ptr_outputs[2]=ptr_right;

      return (ptr_network);
```

```
    }/* end */

/*******************************************/

int in_range (i,j)
    int i,j;

    { if ((i>-1) && (i<4) && (j>-1) && (j<4))
      return (1);
      else return (0);

    }/* end */

/*******************************************/

wire_up_rf (ptr_parallel, ptr_single_rf, x,y)
    network *ptr_parallel, *ptr_single_rf;
    int     x,y;    /* location of top_left on 4*4 grid */

    { int i, top_left, top_right, bottom_left, bottom_right;

      /* need to convert ptr_single_rf inputs to parallel_net
locations */
      /* eg. given top_left of rf, find which input this is actually
*/
      /* referring to.
*/

      top_left      =x+4*y;
      top_right     =x+1+4*y;
      bottom_left   =x+4*(y+1);
      bottom_right  =x+1+4*(y+1);

      /* wire up inputs only if in range    */

      for (i=0; i<4; i++)   /* four orientation planes  */
      {
        if (in_range (x,y))     /* x,y is top_left      */
        ptr_single_rf->ptr_inputs[i*4+0]->dendrites[0]=ptr_parallel-
>ptr_inputs[i*16+top_left];
        else
        ptr_single_rf->ptr_inputs[i*4+0]->num_inputs=0;

        if (in_range (x+1,y))   /* x+1,y is top_right   */
        ptr_single_rf->ptr_inputs[i*4+1]->dendrites[0]=ptr_parallel-
>ptr_inputs[i*16+top_right];
        else
        ptr_single_rf->ptr_inputs[i*4+1]->num_inputs=0;

        if (in_range (x,y+1))   /* x,y+1 is bottom_left */
        ptr_single_rf->ptr_inputs[i*4+2]->dendrites[0]=ptr_parallel-
>ptr_inputs[i*16+bottom_left];
        else
        ptr_single_rf->ptr_inputs[i*4+2]->num_inputs=0;

        if (in_range (x+1,y+1)) /* x+1,y+1=bottom_right */
        ptr_single_rf->ptr_inputs[i*4+3]->dendrites[0]=ptr_parallel-
>ptr_inputs[i*16+bottom_right];
        else
        ptr_single_rf->ptr_inputs[i*4+3]->num_inputs=0;

      }/* end for */

    }/* end */
```

```
/*******************************************/

int rf_required (x,y)
    int x,y;

    { /* rf not required at corner of a square going from   */
      /* (-1,-1) to (3,3)                                   */

      if ( ((x == -1) && (y == -1)) ||
           ((x == -1) && (y ==  3)) ||
           ((x ==  3) && (y == -1)) ||
           ((x ==  3) && (y ==  3))
         ) return (0);
      else return (1);

    }/* end */

/*******************************************/
/* Corner net has three outputs.           */
/* 1) tot. sig. of acute corners           */
/* 2) tot. sig. of obtuse corners          */
/* 3) tot. sig. of right angled corners    */
/*******************************************/


network *create_corner_net (ptr_parallel)
    network *ptr_parallel;

    { int     i,j,k=0,l;
      int     num_outputs=3;
      network *ptr_network, *ptr_single_rf, *ptr_adder[3];

      /* no inputs; they all come from ptr_parallel net        */
      /* 3 outputs: see above comment in box.                  */

      ptr_network=gen_network_unit (0,num_outputs);

      /* create adders and wire up their outputs to ptr_network */
      for(i=0; i<num_outputs; i++) {
        ptr_adder[i]=create_adder();
        ptr_network->ptr_outputs[i]=ptr_adder[i]->ptr_outputs[0];
      }/* end for */

      /* loop for 25 times  */
      for (j= -1; j<4; j++)
        for (i= -1; i<4; i++) {
          if (rf_required(i,j)) {   /* miss out 4       */
            ptr_single_rf=create_single_rf_net ();
            wire_up_rf (ptr_parallel, ptr_single_rf, i,j);

            /* now wire up outputs from this rf (3 outputs) */
            /* to the corresponding inputs on each adder.   */
            for (l=0; l<num_outputs; l++)
              ptr_adder[l]->ptr_inputs[k]->dendrites[0]=ptr_single_rf-
>ptr_outputs[l];

            /* k is a 'valid rf' count  */
            k++;
          }/* end if */
        }/* end for */

      return (ptr_network);
    }/* end create_corner_net */
```

```
/*****************************************/

float square_sigmoid(x,a)
    double x;                   /* -a & +a are the corners of the   */
    float  a;                   /* square threshold function.       */

    { if (x>= a) return(1);             /* threshold at 1           */
      else if (x<=-a) return(0);        /* threshold at 0           */
           else return(0.5*(1+x/a));    /* linear func in between   */
    }/* end */

/*****************************************/

float sigmoid (x)
    double x;

    { return 1/(1+exp(-x));
    }

/*****************************************/

calculate_outputs (ptr_node)    /* recursively goes down network    */
    neuron *ptr_node;

    { double total;
      int i;
      float input;

#ifdef DEBUG2
printf("In calculate_outputs\n");
#endif

      /* correctly set initial value of total */
      if (ptr_node->neuron_class == weighted_sum) total =0.0; else
total =1.0;

      for (i=0; i<ptr_node->num_inputs; i++)
      {
        if ((ptr_node->dendrites[i]->num_inputs != 0)
            &&(ptr_node->dendrites[i]->status == not_calculated))
        calculate_outputs(ptr_node->dendrites[i]);

        input= ptr_node->dendrites[i]->output;
        switch (ptr_node->dendrites[i]->pre_processing)
        {
          case no_effect    :                           break;
          case add_1        :input += 1.0;              break;
          case negate       :input *= -1.0;             break;
          case threshold    :input = input>0.0 ? 1:0;   break;
        }

        if (ptr_node->neuron_class == weighted_sum)
             total += input * ptr_node->weights[i];
        else total *= input * ptr_node->weights[i];

      }/* end for */

      /* activation function */
      switch (ptr_node->sigmoid)
      {
        case no_activation  : ptr_node->output = total; break;
        case smooth_sigmoid : ptr_node->output = sigmoid (total +
ptr_node->bias); break;
        case step           : ptr_node->output = square_sigmoid (total
```

```
+ ptr_node->bias, 0.0); break;
        case linear_1       : ptr_node->output = square_sigmoid (total
+ ptr_node->bias, 1.0); break;
        case linear_2       : ptr_node->output = square_sigmoid (total
+ ptr_node->bias, 2.0); break;
      }

      ptr_node->status=calculated;

#ifdef DEBUG2
printf("Out calculate_outputs\n");
#endif

    } /* end calculate_outputs. */


/*****************************************/

reset_network_status (ptr_node)
    neuron *ptr_node;

    { int i;

#ifdef DEBUG2
printf("In reset_network_status\n");
#endif

      for (i=0; i<ptr_node->num_inputs; i++)
      {
        if ((ptr_node->dendrites[i]->num_inputs != 0)
            &&(ptr_node->dendrites[i]->status == calculated))
        reset_network_status (ptr_node->dendrites[i]);

      }/* end for */

      ptr_node->status=not_calculated;

#ifdef DEBUG2
printf("Out of reset_network_status\n");
#endif

    }/* end */

/*****************************************/

load_inputs_from_file (ptr_network, input_file)
    network *ptr_network;
    filename input_file;

    { int i;
      float value;

      fp=fopen(input_file,"r");

      for (i=0; i<ptr_network->num_inputs; i++)
      {
        fscanf (fp,"%f",&value);
        ptr_network->ptr_inputs[i]->output=value;
        ptr_network->ptr_inputs[i]->status=calculated;
      }/* end for */

      fclose (fp);

    }/* end */
```

```
/*******************************************/

use_network (ptr_network)
    network *ptr_network;

    { int i;
      float value;
      neuron *ptr_neuron;

      /* calculate outputs      */
      for (i=0; i<ptr_network->num_outputs; i++)
      {
        ptr_neuron=ptr_network->ptr_outputs[i];
        calculate_outputs(ptr_neuron);
        printf("%11f",ptr_network->ptr_outputs[i]->output);
      }/* end for */


      /* reset network status   */
      for (i=0; i<ptr_network->num_outputs; i++)
      {
        ptr_neuron=ptr_network->ptr_outputs[i];
        reset_network_status (ptr_neuron);
      }/* end for */
    }/* end */

/*******************************************/
```