Copyright is owned by the Author of the thesis. Permission is given for a copy to be downloaded by an individual for the purpose of research and private study only. The thesis may not be reproduced elsewhere without the permission of the Author.

The Effects of Using Problem Knowledge in a Neural Network for Image Processing Tasks

A thesis presented in partial fulfilment of the requirements for the degree of Doctor of Philosophy in Computer Science at Massey University, Palmerston North, New Zealand.

Kapila Sanjeeva Gunetileke

2001

To the memory of my father Dr. K. G. Gunetileke

ABSTRACT

This thesis is concerned with aspects of computational intelligence. Computational intelligence is a new paradigm of artificial intelligence based on biological intelligence. Computational Intelligence explores the potential for biologically inspired intelligent adaptive machines and behavior. A relatively new and important sub discipline within the field of computational intelligence, is that of neural networks. Neural networks are networks of artificial neurons with a high degree of interconnectivity. The networks capture and accumulate knowledge as the pattern of weights in the interconnections between neurons. Neural networks are iteratively trained to perform tasks by "learning" from examples. These networks are often slow to train. One of the reasons for this is that the starting weights of the network are conventionally not related to the problem being solved. In this thesis a methodology is explored that maps problem knowledge to the starting weights of a fuzzy neural network window filter (FuNNWF). The FuNNWF architecture was developed from the combination of fuzzy logic and artificial neural networks for image processing tasks. The effects of the use of problem knowledge on FuNNWF training are investigated. The problem knowledge mapping procedure is extended from boolean rule mapping to conditional rule mapping, which allows better representation of the problems. Four real world image processing problems are investigated using the new weight initialization methodology. The experiments reported in this thesis demonstrate that the use of problem knowledge improves the robustness and convergence of the neural network. It is also shown that the use of the methodology is most effective on network training when the training data is noisy, unreliable and ambiguous.

ACKNOWLEDGEMENTS

I would like to thank my two supervisors Prof. Bob Hodgson and Dr. Bob Chaplin for their help, encouragement, understanding and patience. I have been very fortunate to have them as my supervisors, who have encouraged high standards in research and writing at the same time as providing considerable insight into the fascinating area we have been studying. I would also like to thank them for their friendship and sense of humour.

I would like to thank my mother and grandfather who always encourage me to pursue further studies. Finally I would like to thank my wife Samani for her support, encouragement and the considerable amount of "family time" that she sacrificed to enable me to complete this thesis.

CONTENTS

Abstract i

Acknowledgements iii

Contents v

List of Figures xiii

List of Publications xvii

CHAPTER 1 1

1 INTRODUCTION 1

1.1 Scope of research 3

1.2 Thesis overview 4

1.3 Contents by chapter 5

CHAPTER 2 7

2 BACKGROUND 7

2.1 Fuzzy logic 7
- 2.1.1 Example of a fuzzy membership function 8
- 2.1.2 Example of a rule 9

2.2 Artificial Neural Networks 10
- 2.2.1 Neural network weight initialization 10
- 2.2.2 Architecture of a Fuzzy Neural Network 12
- 2.2.3 FuNN training methods 13
- 2.2.4 The training algorithm 14

2.3 Image processing 18
- 2.3.1 The window filter 18

2.4 The fuzzy neural network window filter 20
- 2.4.1 Structure of the FuNNWF 21

2.5 Putting it all together 21
- 2.5.1 Problem knowledge 21

2.6 Environment 22
- 2.6.1 Software 22

2.6.2 Hardware 23
2.6.3 Simulations 23

2.7 Summary 24

CHAPTER 3 25

3 RULE MAPPING PROCEDURE 25

3.1 Problem knowledge 26

3.2 Mapping to FuNN architecture 28

3.3 Rule insertion procedure 29
3.3.1 Boolean logic rule mapping 30
3.3.2 Conditional rule mapping 33

3.4 General derivation of rules 41
3.4.1 Rule: Input > Constant 41
3.4.2 Rule: Input = Constant 44
3.4.3 Multiple inputs to a Rule node 45
3.4.4 K out of M expressions 45
3.4.5 Rule: Input i > Input j 46
3.4.6 Non conclusive logic 47

3.5 The quality factor 47

3.6 Discussion 47

CHAPTER 4 49

4 USE OF A FUNN TO SOLVE FULLY DETERMINED PROBLEMS 49

4.1 The XOR Problem 49
4.1.1 Training and test data generation 50
4.1.2 Rules representing the XOR problem 52
4.1.3 The error function 52
4.1.4 The MSE error 53
4.1.5 Fuzzy neural network structure used for XOR problem 53
4.1.6 Experiment to determine if a FuNN could emulate the XOR problem without training using boolean logic rules. 53
4.1.7 Experiment to determine if a FuNN could emulate the XOR problem without training using conditional rules. 55
4.1.8 Experiment - Operational performance test 1 for a range of parameters on the XOR problem using boolean logic rules 56
4.1.9 Experiment – Operational performance test 2 for a the same parameters but a smaller range on the XOR problem using boolean logic rules 58
4.1.10 Experiment – Operational performance test 3 for a range of parameters on the XOR problem using conditional rules 59
4.1.11 Experiment – Operational performance test 4 for the same parameters on a smaller range on the XOR problem using conditional rules 60

4.2 Partial Rule Sets 60
4.2.1 Training and test data 61
4.2.2 Fuzzy neural network structure used for this problem 61
4.2.3 Experiment 1 - Problem A = B & C & (not D) – complete rule set 62

4.2.4 Experiment 1a - Problem A = B & C & (not D) – incomplete rule set 63
4.2.5 Experiment 2 – Problem A = B & (not C) & D – complete rule set 65
4.2.6 Experiment 2a – Problem A = B & (not C) & D – incomplete rule set 66
4.2.7 Experiment 3 – Problem A = (not B) & C & D – complete rule set 68
4.2.8 Experiment 3a – Problem A = (not B) & C & D – incomplete rule set 69
4.2.9 Experiment 4 – Problem A = B&C&(~D) using only a partial rule set 71

4.3 Discussion and Conclusions 72

CHAPTER 5 75

5 FULLY UNDERSTOOD NON DETERMINED PROBLEM 75

5.1 Overview of the toy problem 75
5.1.1 A detailed description of the problem 75
5.1.2 Description of a toy 76
5.1.3 The training images 77

5.2 Network training method 80
5.2.1 The training data 80
5.2.2 Rule development 80
5.2.3 The output image 81

5.3 Toy type classification 81
5.3.1 Membership functions for fuzzification process 82
5.3.2 Membership functions for defuzzification process 82
5.3.3 Error measure 83
5.3.4 Experiment to determine how a network initialized with rules performs compared to one initialized without rules 83
5.3.5 Experiment to measure the performance of a network over 100 runs when initialized with rules 85
5.3.6 Experiment to measure the performance of a network initialized with random numbers over 100 runs. 86
5.3.7 Experiment to determine if adding rules developed to remove false positives would improve the performance of the network. 86
5.3.8 Experiment to determine if changing the quality factor for individual rules improved the classification performance of the network. 87
5.3.9 Experiment to determine the effects of adding small random numbers to a network initialized with rules over 100 runs. 88
5.3.10 This experiment is a continuation of the previous experiment. A network with the same structure was initialized with random number and trained. 89
5.3.11 Experiment is to determine the network with the best performance from several networks initialized with rules. 89
5.3.12 This is a continuation of the previous experiment. The quality factor for the rules was changed and the experiment was repeated 91
5.3.13 This is a continuation of the previous experiment to measure the performance of the network without rules. 92
5.3.14 Experiment to determine how a bad rule changes with network training 92
5.3.15 This Experiment is a continuation of the previous experiment to determine how a bad rule changes with network training 95
5.3.16 Experiment to measure the operational performance of a network initialized with rules using a larger number of membership functions. 96

5.4 Toy orientation classification 97
5.4.1 Membership functions for fuzzification process 98
5.4.2 Defuzzification membership functions 98
5.4.3 Experiment to compare the performance of a network initialized with rules and random numbers 99

5.4.4 Experiment to determine network performance with a new set of fuzzification membership functions and extended rules. 100
5.4.5 A statistical experiment to determine the performance of a network..... 102
5.4.6 Experiment to determine the operational performance of a population of networks tested on several new images. 103
5.4.7 Experiment to determine the incremental and cumulative weight change during training 103
5.4.8 This is a continuation of the previous experiment to determine the effects of adding free nodes to the network 106
5.4.9 Experiment to determine the effects of adding a bad rule to the network..... 107
5.4.10 Experiment to determine the range of operational performances shown by a population of networks when initialized with rules 109

5.5 Classification of Rules 110
5.5.1 A strong rule 110
5.5.2 A weak rule 110
5.5.3 A weak incorrect rule 110
5.5.4 A strong incorrect rule 110
5.5.5 A bad rule 110

5.6 Conclusions 111
5.6.1 Specific Conclusions 111
5.6.2 General Conclusions 114

CHAPTER 6 115

6 WANE EDGE DETECTION 115

6.1 The wane edge detection problem 115

6.2 Problem knowledge for wane edge detection 118
6.2.1 Problem knowledge expressed in high level terms 118
6.2.2 The size and shape of the window filter 119

6.3 Training data 119

6.4 The membership functions 120

6.5 The error measure 121
6.5.1 The error measure function 121

6.6 The experiments 123
6.6.1 Experiment to detect the edges using a 3x3 window filter 124
6.6.2 Experiments to compare the effectiveness of wane edge detection for a range of window sizes 127
6.6.3 Experiment to determine if the network could detect one wane edge only using a 9x23 window filter 133
6.6.4 Use of a 9x9 window filter to detect the left wane edge. 137
6.6.5 Network training experiment that involves changing the number of rules and training times 139
6.6.6 Experiment to determine if a FuNN using a 9x9 window filter with a smaller rule set could detect the left wane edge. 142
6.6.7 Experiment to detect the left wane edge using conditional rules. 144
6.6.8 Experiment to determine the performance of a FuNN seeded with different random numbers. 100 runs were made 147
6.6.9 Experiment to detect the right wane edge 149
6.6.10 Generating the final image containing both left and right wane edges 150
6.6.11 Using the trained network to detect the wane edges on other planks of wood 151

6.6.12 Experiment to test alternative training algorithms 154
6.6.13 Experiment to determine the quality factor for the rules 158

6.7 Conclusions 161

CHAPTER 7 163

7 TREE CROWN DETECTION 163

7.1 Introduction 163
7.1.1 The training images 165
7.1.2 Window filter size 166
7.1.3 The membership functions 167
7.1.4 The rules 167

7.2 Experiments 169
7.2.1 Experiment to develop a neural network to detect tree crowns 169
7.2.2 Experiment to investigate the used of a hierarchical network structure to separate blobs of clustered trees into blobs corresponding to individual trees 173
7.2.3 Experiment to determine the tree counts for multiple networks initialized with random numbers 176
7.2.4 The new training images 177
7.2.5 Experiment to determine the tree count when the network was trained using the supplied ground truth based training images 179
7.2.6 Experiment to determine the tree count made by the neural network using modified ground truth data. 182
7.2.7 Experiment to determine the tree counts for 100 networks initialized with random numbers 184
7.2.8 Experiment to investigate the hypothesis that the rules compensate for inaccuracies in the training data 186

7.3 Conclusions 190

CHAPTER 8 191

8 IRIS EDGE DETECTION 191

8.1 Introduction 191
8.1.1 Iris isolation 192
8.1.2 Use of a standard filter to detect the iris edge 194

8.2 The training images 196

8.3 Development of a figure of merit 197

8.4 The experiments 198
8.4.1 Experiment to determine if a FuNN could detect the iris edge 198

8.5 The new training images 203

8.6 Generation of the new membership functions 204
8.6.1 Membership functions for fuzzification process 205
8.6.2 Membership functions for defuzzification process 206

8.7 The experiments 206
8.7.1 Experiment to find the iris edge using a FuNN initialized with rules. 206

8.7.2 Experiment to find the iris edge using a FuNN using a larger window than in experiment 8.7.1 ... 209
8.7.3 Experiment to find the iris edge using a FuNN using a greater number of free nodes compared to experiment 8.7.2 ... 212
8.7.4 Experiment to train a network to detect the iris edge near the eyelashes ... 215
8.7.5 Use of two neural networks to detect the iris edge ... 217
8.7.6 Operational test for iris detection problem using two neural networks ... 222
8.7.7 Experiment to determine if the least cost path algorithm in the post process procedure could be replaced by a neural network ... 225
8.7.8 Experiment to determine the weight change of the rules during training ... 228
8.7.9 Experiment to determine if removing rule 2 and 7 from the rule set from experiment 8.7.8 would improve the network performance ... 230

8.8 Conclusions ... 233

CHAPTER 9 ... 235

9 CALCIFICATION DETECTION IN MAMMOGRAMS ... 235

9.1 Introduction ... 235

9.2 The training data ... 236
9.2.1 Window filter size ... 239

9.3 Rule development ... 240
9.3.1 The texture based method ... 240
9.3.2 Spatial Gray Level Dependence Matrix (SGLDM) ... 241
9.3.3 Neural network training method ... 244
9.3.4 The values for the texture measures ... 244

9.4 The experiments ... 246
9.4.1 Experiment to determine if the FuNN could detect microcalcifications in a mammogram ... 247
9.4.2 Experiment to compare network performance ... 249
9.4.3 Experiment to determine the network performance of a large population of networks 251
9.4.4 Experiment to determine if the calcification detection rate of the network could be improved ... 252

9.5 Discussion and conclusions ... 254

CHAPTER 10 ... 257

10 REDUCING THE FUNN TO A CONVENTIONAL MLP NETWORK ... 257

10.1 Introduction ... 257

10.2 Funn to MLP mapping ... 258
10.2.1 Fuzzification layer equivalence ... 259
10.2.2 Defuzzification layer equivalence ... 261
10.2.3 Yam and Chow's weight initialization method ... 263

10.3 The experiments ... 265
10.3.1 Experiment to determine the network performance of a MLP network initialized with rules ... 265
10.3.2 Experiment to determine the effects on MLP training when a modified version of Yam and Chow's initialization method was used ... 272

10.4 Conclusions 273

CHAPTER 11 275

11 SUMMARY AND CONCLUSIONS 275

11.1 Summary 275

11.2 Conclusion 277

11.3 Future work 279

APPENDIX A 283

REFERENCES 309

BIBLIOGRAPHY 315

LIST OF FIGURES

Figure 2.1 A membership function for young. 8
Figure 2.2 Membership function for an example rule. 9
Figure 2.3 Fuzzy neural network architecture. 14
Figure 2.4 Operation of a window filter. 19
Figure 2.5 Example of filter operation. Image (a) is the original image. Image (b) was generated by passing a common edge detection filter over image (a). Image (c) was generated by passing an embossing filter over image (a). 19
Figure 2.6 Neural network window filter. 20
Figure 2.7 Fuzzy neural network window filter structure. 21
Figure 3.1 (a) Image containing circular objects and other geometric shapes, all the objects are outlined for clarity. (b) – (e) clusters of objects extracted from (a) 26
Figure 3.2 Fuzzy neural network section. 34
Figure 3.3 Rule node output function. 35
Figure 3.4 Fuzzy function for Input>Constant. 41
Figure 3.5 Fuzzy function for Input = Constant. 44
Figure 4.1 XOR training data – 200 pairs. 51
Figure 4.2 XOR test data – 600 pairs. 51
Figure 4.3 FuNN 2-4-4-2-1 architecture for XOR problem. 53
Figure 4.4 Operational error graph for XOR problem. 57
Figure 4.5 Operational errors for different saturation values. 59
Figure 4.6 Operational error graph for A=B&C&(~D). 64
Figure 4.7 Operational error graph for A=B&(~C)&D. 67
Figure 4.8 Operational error graph for A=(~B)&C&D. 70
Figure 4.9 Operational performance of A=B&C&(~D) for a network using a partial rule set with high certainty. 72
Figure 5.1 Toys that can be classified by type and orientation. 76
Figure 5.2 Image containing 25 individual toys of different type and orientation. 77
Figure 5.3 Training images. 78
Figure 5.4 Window filter structure. 81
Figure 5.5 Incremental weight change graph for experiment 5.3.14. 93
Figure 5.6 Cumulative weight change for experiment 5.3.14. 94
Figure 5.7 Incremental weight change graph for experiment 5.3.15. 95
Figure 5.8 Toy orientation problem defined by the placement of the 2x2 square around the 3x3 larger square. 97
Figure 5.9 Incremental weight change of the network (Q=0.5) over 175 epochs for experiment 5.4.7. (A) is the incremental weight change for the rules and (B) is the incremental weight change for the free nodes in the network. 104
Figure 5.10 Cumulative weight change for the network (Q=0.5) over 175 epochs for experiment 5.4.7. (A) is the cumulative weight change for the rule nodes and (B) is the cumulative weight change for the free nodes. 105
Figure 5.11 Incremental weight change of the network (Q=0.5) over 175 epochs for experiment 5.4.8. (A) is the incremental weight change for the rules and (B) is the incremental weight change for the free nodes in the network. 106
Figure 5.12 Cumulative weight change for the network (Q=0.5) over 175 epochs for experiment 5.4.8. (A) is the cumulative weight change for the rule nodes and (B) is the cumulative weight change for the free nodes. 107
Figure 5.13 Incremental weight change when (Q=0.5) for experiment 5.4.9. 108
Figure 5.14 Cumulative weight change for the network (Q=0.5) over 175 epochs for experiment 5.4.9. (A) is the cumulative weight change for the rule nodes and (B) is the cumulative weight change for the free nodes. 108
Figure 6.1 Image of a plank with manually marked defects. 116

Figure 6.2 The images show the output generated by processing the original image with a vertical edge detector and thresholding. Image (a) is the output generated by processing figure 6.1 with a common edge detection filter. Image (b) was generated by thresholding image (a). . 117
Figure 6.3 Training images for wane problem. 119
Figure 6.4 The training data was generated using the rectangular sections in the image. 120
Figure 6.5 Image showing the wane edge, bark and useable area on the plank of wood. 122
Figure 6.6 Neural network output for experiment 0. Image (a) is the output generated by the network initialized with rules and image (b) is the output generated by the network initialized with random numbers. 126
Figure 6.7 Representation of If I1=HI Then O1=LO. 131
Figure 6.8 Network output for experiment 6.6.3 135
Figure 6.9 The post process procedure, image (a) generated by thresholding the network output, image (b) generated by thinning the thresholded output and image (c) generated by scanning the thinned image from left to right to detect the first occurrence of a high intensity pixel... 137
Figure 6.10 Results of training the network for 200 epochs for experiment 6.6.4 (a) network output (b) network output thresholded (c) thresholded image thinned (d) thinned image scanned for left wane edge. 138
Figure 6.11 Results after training t network for 800 epochs for experiment 6.6.4 (a) output image (b) output image thresholded (c) thresholded image thinned (d) thinned image scanned for left wane edge. 139
Figure 6.12 Output images for experiment 6.6.5 141
Figure 6.13 Output images for experiment 6.6.6 143
Figure 6.14 Training performance graph for experiment 6.6.7. 146
Figure 6.15 Output images for experiment 6.6.7. Image (a-1) output image from network with rules. Image (a-2) generated by post processing (a-1). Image (b-1) output image from network without rules and image (b-2) generated by post processing (b-1). 147
Figure 6.16 The output images for experiment 6.6.9. (a) the output image generated by the network, (b) the output image thresholded, (c) thresholded image thinned, (d) thinned image scanned for right wane edge. 150
Figure 6.17 (a) shows the left and right wane combined, (b) shows the combined wanes superimposed on the input image. 151
Figure 6.18 Plank 1 used to test the trained FuNN. (a) input image, (b) left and right wane detected using trained FuNN, (c) detected left and right wane superimposed on the input image. 152
Figure 6.19 Plank 2 used to test the trained FuNN. (a) input image, (b) left and right wane detected using trained FuNN, (c) detected left and right wane superimposed on the input image. 153
Figure 6.20 Plank 3 used to test the trained FuNN. (a) input image, (b) left and right wane detected using trained FuNN, (c) detected left and right wane superimposed on the input image. 153
Figure 6.21 Graph showing the error between the output and target image during training for the network initialized with rules. 157
Figure 6.22 Graph showing the error between the output and target image during training for the network initialized with random numbers. 158
Figure 6.23 Network performance for experiment 6.6.13 when 6 rules and 1 free node was used. 159
Figure 6.24 Network performance for experiment 6.6.13 when 6 rules and 2 free nodes were used. 160
Figure 6.25 Network performance for experiment 6.6.13 when 6 rules and 4 free nodes were used. 160
Figure 7.1 Aerial photograph of a commercial Pinus Radiata plantation. 164
Figure 7.2 The training images. Image (a) is a sub image from the original aerial photograph. Image (b) is the pre-processed image and image (c) is the target image. 166
Figure 7.3 Image of a tree superimposed on the 11x11 window. 166
Figure 7.4 Image of two trees close to each other. Rule 3 was generated from this image 168
Figure 7.5 Image of two trees close to each other. Rule 4 was generated from this image. 168
Figure 7.6 Image of two trees horizontally close to each other. Rule 5 was generated from this image. 169
Figure 7.7 Image of two trees vertically close to each other. Rule 6 was generated from this image. 169
Figure 7.8 Output images generated by the trained networks for experiment 7.2.1. Image (a) is the output image generated by the network initialized with rules and image (b) is the output generated by the network initialized with random numbers. 170

Figure 7.9 Training performance graph for network initialized with and without rules for experiment 7.2.1 ... 171
Figure 7.10 Hierarchical neural network structure ... 173
Figure 7.11 Output images generated by the trained networks for experiment 7.2.2. Image (a) is the input image, image (b) is the output image generated by the network initialized with rules and image (c) is the output generated by the network initialized with random numbers. ... 175
Figure 7.12 Training performance graph for networks initialized with and without rules for experiment 7.2.2 ... 175
Figure 7.13 Tree count distribution for network initialized with random numbers for experiment 7.2.3 ... 177
Figure 7.14 The commercial plot. ... 178
Figure 7.15 The ground truth data for the commercial plot. ... 178
Figure 7.16 The new training images for the tree crown detection problem. Image (a) is the input image and image (b) is the target image ... 179
Figure 7.17 Image (a) is the output image generated by the hierarchical neural networks for experiment 7.2.5. Image (b) is the target image. ... 180
Figure 7.18 The image shows a sub-image from the commercial plot. The boxes and circles mark the tree positions given by the ground truth data ... 181
Figure 7.19 New training image for tree crown detection problem. (a) input image and (b) target image. ... 183
Figure 7.20 Output image generated by hierarchical neural network for experiment 7.2.6. ... 184
Figure 7.21 Tree count distribution for experiment 7.2.7. ... 185
Figure 8.1 Image of an eye used for the iris edge detection problem ... 192
Figure 8.2 Image (a) is the linearised eye image using a polar to rectangular transform. Image (b) shows the pupil and iris edges of the eye. ... 194
Figure 8.3 Processing the input image using a vertical edge detection filter. Image (a) is the input image and image (b) is the processed image. ... 195
Figure 8.4 Training images for iris detection problem. Image (a) is the input image and image (b) is the target image. ... 196
Figure 8.5 Image to demonstrate the error calculation. ... 197
Figure 8.6 Input image showing areas of interest ... 199
Figure 8.7 Shows the window filter placed over a section of the eye. ... 200
Figure 8.8 Output image generated by processing the input image with the trained neural network. ... 202
Figure 8.9 Training images for iris detection problem. Image (a) is the original eye image, image (b) was generated by passing an illumination correction process on (a) and is used as the input image and image (b) is the target image ... 204
Figure 8.10 Input image displaying areas of interest to generate fuzzification membership functions. ... 205
Figure 8.11 Output generated from the trained network for experiment 8.7.1 ... 208
Figure 8.12 Output image generated by the trained network initialized with rules for experiment 8.7.2 ... 211
Figure 8.13 Network output for experiment 8.7.3. Image (a-1) was generated by the network initialized with rules, image (a-2) was generated by thresholding and thinning image (a-1). Image (b-1) was generated by the network initialized with no rules and image (b-2) was generated by thresholding and thinning image (b-1) ... 214
Figure 8.14 Profile plots of different sections along the iris edge ... 215
Figure 8.15 The training images and network output for experiment 8.7.4. Images (a) and (b) were the training images and image (c) was the network output. ... 217
Figure 8.16 Segmented input image showing section 1 and section 2. ... 218
Figure 8.17 Image (a) is section 1 from the input image and image (b) is the output generated by the network ... 218
Figure 8.18 Image (a) is the input image and image (b) is the output generated for section 2 ... 219
Figure 8.19 Image (a) is the combined output. Image (b) was generated by post processing image (a). Image (b) was superimposed on the input image to generate image (c). ... 220
Figure 8.20 Image (a) is the combined output from the two networks. Image (a) was post processed to generate image (b). Image (b) was superimposed on the input image to generate image (c). ... 221
Figure 8.21 New eye images for operational test. ... 223
Figure 8.22 Eye images in figure 8.21 post processed. ... 223

Figure 8.23 Eye images in figure 8.22 processed using network one and network two................ 224
Figure 8.24 Images in figure 8.23 post processed.. 224
Figure 8.25 Post processed network output overlaid over input images. 225
Figure 8.26 Input image showing areas of interest.. 226
Figure 8.27 Image (a) is the input image and image (b) is output image generated by processing image (a) using the trained network. 228
Figure 8.28 The cumulative weight change graph for the network when Q=0.5. The weight change (rules) graph shows the cumulative weight change for the rule nodes and the weight change (free) graph shows the cumulative weight change for the free nodes. 230
Figure 8.29 The results of the cumulative weight change for experiment 8.7.9 using a quality factor of 0.5. The weight change (rules) graph shows the cumulative weight change for the rule nodes and the weight change (free) graph shows the cumulative weight change for the free nodes. 231
Figure 9.1 The two mammographic views taken for each breast.. 237
Figure 9.2 CC mammographic view of the right breast from case 1220 from the DDSM............. 237
Figure 9.3 MLO mammographic view of the right breast from case 1220 from the DDSM. 238
Figure 9.4 An image of a clustered microcalcification. 238
Figure 9.5 Training images for microcalcification detection problem. Image (a) is the input image and image (b) is the target image. 239
Figure 9.6 Input generation for FuNN for mammogram problem.. 244
Figure 10.1 The proposed FuNN to MLP mapping. 259
Figure 10.2 The proposed equivalent MLP network for the FuNN. (a) shows a section from the FuNN and (b) shows the equivalent MLP section.. 260
Figure 10.3 Proposed mapping for defuzzification section. (a) is the output layer of the FuNN and (b) is the equivalent output layer for the MLP network. 261
Figure 10.4 Training performance of MLP network for the Sobel filter emulation problem......... 269
Figure 10.5 Training performance of MLP network for wane edge detection problem................ 270
Figure 10.6 Training performance of MLP network for the tree crown detection problem........... 271
Figure 11.1 Extended input layer of a FuNNWF. 279

PUBLICATIONS

Publications prepared during the course of the research for this thesis are listed below. Refereed publications are highlighted:

1. Gunetileke S., Chaplin R. I., Siroki M., Hodgson R. M., The Development of Neural Network Technologies for Image Processing, Proceedings of the 5th Annual New Zealand Engineering and Technology Postgraduate Conference, Palmerston North, New Zealand, November 1998, pp 103-104.

2. **Chaplin R. I., Siroki M., Hodgson R. M. & Gunetileke S., The Development of Mapping Techniques to Incorporate Image Processing Problem Specific Rules into Neural Networks, Proceedings of the Image & Vision Computing New Zealand International Conference (IVCNZ '98), Auckland, New Zealand, November 1998, pp 357-362 (Refereed publication).**

3. **Chaplin R. I., Hodgson R. M. & Gunetileke S., Towards the Use of Problem Knowledge in Training Neural Networks for Image Processing Tasks, Proceedings of IEE Seventh International Conference on Image Processing and its Applications, Manchester, UK, July 1999, pp 62-66 (Refereed publication).**

4. **Chaplin R. I., Hodgson R. M. & Gunetileke S., Automatic Wane Detection in the Images of Planks using a Neural Network, Proceedings of the 5th International Symposium on Signal Processing and its Applications (ISSPA '99), Brisbane, Australia, August 1999, pp 657-660 (Refereed publication).**

5. **Chaplin R. I., Hodgson R. M. & Gunetileke S., Seeding an ANN From Problem Knowledge for Image Processing Tasks, Proceedings of the Image & Vision Computing New Zealand International Conference (IVCNZ '99), Christchurch, New Zealand, August 1999, pp 277-282 (Refereed publication).**

6. Chaplin R. I., Hodgson R. M. & Gunetileke S., Towards the Use of Problem Knowledge in Training Neural Networks for Image Processing Tasks, 6th National New Zealand Postgraduate Conference, Christchurch, New Zealand, November 1999.

7. **Chaplin R. I., Hodgson R. M. & Gunetileke S., Seeding an ANN from Problem Knowledge for Image Processing Tasks, *in* Emerging Knowledge Engineering and Connectionist-Based Information Systems, Proceedings of the ICONIP/ANZIIS/ANNES '99 International Workshop, Dunedin, New Zealand, November 1999, pp 83-87 (Refereed publication).**

8. Gunetileke S., Chaplin R. I. & Hodgson R. M., A Tutorial on how to Insert Problem Knowledge into a Fuzzy Neural Network, Proceedings of the 7th Annual New Zealand Engineering and Technology Postgraduate Conference, Palmerston North, New Zealand, November 2000, pp 411-416.

9. **Gunetileke S., Chaplin R. I. & Hodgson R. M., The Use of Problem Knowledge to Improve the Robustness of a Fuzzy Neural Network, *in* Neural Networks for Signal Processing X, Proceedings of the 2000 IEEE Signal Processing Society Workshop, Sydney, Australia, December 2000, pp 682-691 (Refereed publication).**

10. **Chaplin R. I., Gunetileke S. & Hodgson R. M., Initializing Neural Networks with A-Priori Problem Knowledge, *in* Neural Networks for Signal Processing X, Proceedings of the 2000 IEEE Signal Processing Society Workshop, Sydney Australia, December 2000, pp 165-174 (Refereed publication).**

For example, let us assume that our problem is to assemble a dismantled table. The table consists of four legs, one tabletop, two sideboards, a single middle board, one drawer and sufficient screws. A diagram of the assembled table is also provided. The problem is to assemble the table to match the one in the diagram. Instructions on how to assemble the table are not provided. A human may or may not be able to assemble the table using the diagram of the assembled table. The ability to assemble the table depends on the intelligence of the human, his or her past experience and the amount of information provided to the human on table assembly. Intelligence can probably be thought of as a means of how effectively connections can be made between experience and knowledge stored in the brain to solve a particular problem. The more intelligent the human is, the easier it is to make connections between experience and knowledge to solve a problem. The level of instructions required for assembling the table increases as intelligence reduces. This can be thought of as providing knowledge to assist the human to solve the problem.

An artificial neural network (ANN) is a biologically inspired computational model, which consists of processing elements called neurons and connections between them. The coefficients or weights of these connections can change during the learning or training process. ANNs are trained through and iterative process. The training process captures and accumulates knowledge in the weights of the interconnections between the neurons. Prior to training, the ANN does not have any knowledge regarding the problem to be solved. This loosely is analogous to the situation where the human with no relevant past experience has to assemble the table without any instructions. The term intelligence does not have a proper definition. The ANN has no intelligence when compared to the intelligence of a human. Usually the weights of the ANNs are initialized with random numbers prior to training. This form of initialization has no relationship to the problem to be solved. Similar to providing a human with table assembly instructions, encapsulating problem knowledge in the initial weights of the ANN could bias the system closer to a solution.

The scope and logical structure of the research work performed is outlined in this chapter. The key ideas to be presented in the body of the thesis are introduced and the content of each chapter is briefly outlined.

1.1 Scope of research

The key area of research investigated in this thesis is neural network initialization prior to training. Neural network training involves the network learning the relationships between inputs and outputs. Usually this training process is started by initializing the weights of the network with random numbers. This form of initialization has no relationship to the problem to be solved.

The thesis shows how knowledge of a problem can be represented using rules. These rules are then mapped to the initial weights of a neural network. As research progressed the mapping procedure was further developed from previous work carried out by a postdoctoral researcher and the related Massey team. This allowed better use of problem knowledge.

The main focus of this research is an investigation of the effects of using problem knowledge in neural network training. The methods developed allowed more effective representation of problem knowledge and mapping.

The experiments performed in this thesis are based on image processing problems because,

1. The group and supervisors are experts in conventional image processing.
2. Large data sets are available.
3. Suitable problems had been identified.
4. Images readily give insight into the processes.
5. The effects of the use of problem knowledge were apparent in training.
6. The problems can be represented in fuzzy terms.

1.2 Thesis overview

The key achievements of this work are:

- The previous research involved mapping boolean logic rules extracted from problem knowledge to the initial weights of a fuzzy neural network. The research carried out in this thesis extends the mapping procedure to include conditional rule mapping.

- For some image processing problems an algorithmic process cannot be defined. It is demonstrated that for some of these problems when standard image processing tasks fails the fuzzy neural network succeeds.

- It has been demonstrated that the use of problem knowledge to initialize a fuzzy neural network has a beneficial effect on network training. In some cases network training time was reduced and a better solution was achieved. In all the problems the use of rules caused the network to converge to a useful solution.

- A program was developed to map symbolic rules to a fuzzy neural network. The program allows the user to represent knowledge symbolically without having to worry about the actual mapping process. This program was used to initialize all of the network weights prior to training for all of the experiments reported here.
- A confidence factor or a quality factor was used to qualify rules. It is demonstrated how placing the optimum quality factor for rules affects network training. The quality factor led to the development of a procedure to refine rules. This iterative procedure can be used to improve network "performance". The term performance is used to describe the operational and learning performance of a network.

The experiments described in this thesis show how the use of rules and the rule refinement procedure could be used to improve neural network training.

1.3 Contents by chapter

Chapter 2:

The work reported in this thesis is a result of study and integration of three areas of knowledge; fuzzy logic, artificial neural networks and image processing. These three areas are explained in this chapter. The chapter also shows how fuzzy logic and neural networks can be combined to generate a fuzzy neural network (FuNN) and how the FuNN can be used for image processing tasks [Siroki 1998, Pugmire 1997].

Chapter 3:

This chapter describes how rules can be generated from problem knowledge, and how they can be mapped to a FuNN. It describes boolean rule mapping and how it has been extended to conditional rule mapping. The conditional rule mapping procedure was developed during work on the tree crown detection problem (chapter 7).

Chapter 4:

A complete rule set can be generated for problems that are fully determined. In this chapter the weights of a FuNN are initialized using complete and partial rule sets describing the problems. The effects on network training are investigated when using the complete and partial rule sets.

Chapter 5:

In this chapter an artificial problem was created that was both fully understood and not fully determined. The problem involves finding and classifying artificial "toys" in an image using a FuNN. The effects of using a partial rule set to initialize the weights of a FuNN are investigated in this chapter.

Chapter 6:

This chapter shows how a FuNNWF can be used to detect the wane edge in a plank of wood. The chapter shows how standard edge detection techniques do not perform adequately in finding the wane edge. It also shows how the network has trouble coping with tasks that are opposite to each other at the same time.

Chapter 7:

This chapter deals with the problem of detecting tree crowns from an aerial photograph. Two neural networks operate sequentially in this problem. The chapter shows how different processes can be divided among neural networks.

Chapter 8:

This chapter deals with another natural problem, to detect the outer iris edge in an eye. This problem has some similarities to the wane edge detection problem. The problem required two FuNNs to detect the iris edge

Chapter 9:

Another natural problem is investigated in this chapter. The problem involves using a FuNN to detect microcalcifications in screening mammograms. For this problem, no benefit was gained from using problem knowledge to initialize the weights of the FuNN, although it did provide useful insight into the initialization methodology.

Chapter 10:

Chapter 3 shows how problem knowledge can be mapped to the weights of a FuNN architecture. This chapter shows how the weights in a FuNN can be mapped to a standard multilayer perceptron (MLP) network.

Chapter 2

2 Background

This thesis is a result of the study and integration of material from three areas of knowledge; fuzzy logic, artificial neural networks and image processing. In this chapter these areas are introduced as the background to the explanation of the operation of a fuzzy neural network based window filter. The work reported is concerned with the development of a fuzzy neural network initialization scheme to improve network performance.

2.1 Fuzzy logic

Capturing human expertise is often a useful means for devising computer programs to solve complicated real world problems [Fogel 1999]. A difficulty arises when this expertise is expressed using natural language to impart knowledge and information as there is a great deal of imprecision and vagueness. People do not often rely on hard and fast rules but instead base their actions on imprecise criteria. Nyugen *et al* in [Nguyen 1997] gives the following examples,

1. A rule for driving such as "if an obstacle is *close*, then brake *immediately*".
2. The classification of people by age such as *old*.
3. The classification of a certain object as *large*.

Terms such as *close*, *old* and *large* are fuzzy in the sense that they cannot be sharply defined [Nguyen 1997]. Concepts like "about 10 a.m." are imprecise and subject to interpretation. Standard logical operations cannot be used on these concepts, as they are fuzzy in nature so a mathematical foundation was developed where words are used for computing rather than numbers. This mathematical foundation known as fuzzy logic was developed by Zadeh in 1965 to handle terms that are not boolean in character but rather fuzzy and indistinct. This can be viewed as an extension of classic set theory. Instead of an element being in or out of a set, each element can be assigned a degree of membership in the set, often scaled in the range {0,1}. If an element has membership zero it is considered not to be a member of the fuzzy set and a membership of one indicates that it is definitely a member of the set.

2.1.1 Example of a fuzzy membership function

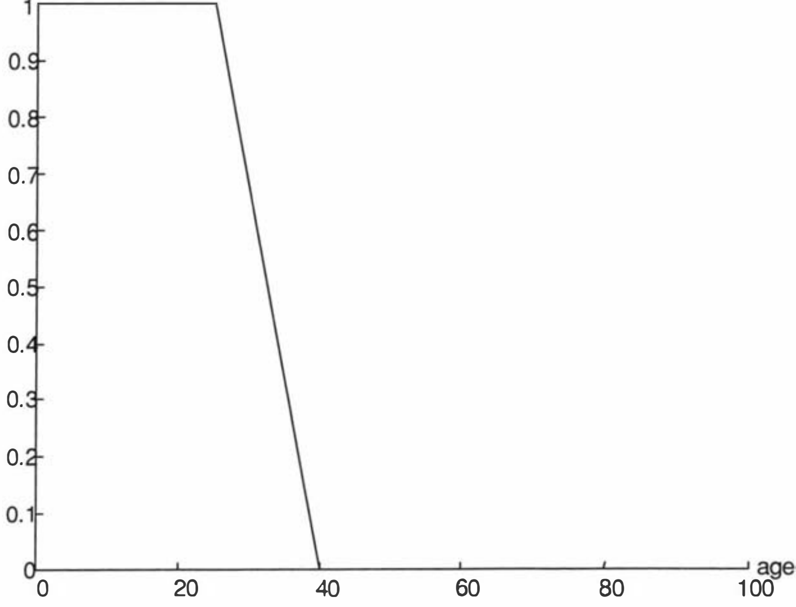


Figure 2.1 A membership function for young.

Figure 2.1 shows a membership function for "young" which is modelled using,

$$Y(x) = \begin{cases} 1 & if \quad x < 25 \\ \left(\dfrac{40 - x}{15} \right) & if \ 25 \leq x \leq 40 \\ 0 & if \quad 40 < x \end{cases}$$

2.1.2 Example of a rule

A rule has two parts, and antecedent and a consequent. Membership functions can be developed for each part of the rule. Figure 2.2 shows membership functions for the antecedent and consequent of a rule. The membership functions for the antecedent represents the intoxication level of a person while the consequent represents the likeliness of that person meeting with an accident.

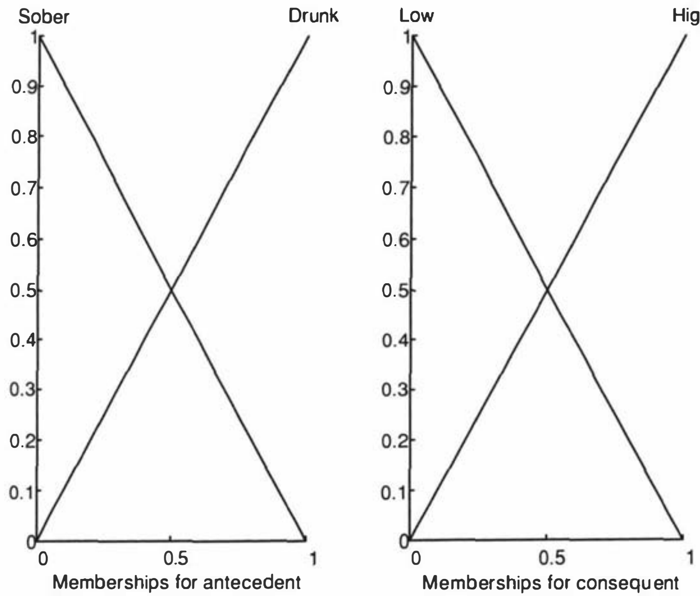


Figure 2.2 Membership function for an example rule.

A rule can be constructed using these membership functions,

- If the person is *drunk* then the likeliness of an accident is *high.*

Expertise to solve a problem or "problem knowledge" can be expressed similarly.

2.2 Artificial Neural Networks

The field of artificial neural networks (ANN), involves the use and study of a range of models and techniques, which are either based on or inspired by the knowledge of biological neural systems. The brain, which is a biological neural system, uses highly interconnected networks of relatively simple parallel processing elements to control interaction with the environment producing "intelligent" behavior. The brain is an immensely complex network of neurons, synapses, axons and dendrites. It comprises at least $2x10^{10}$ neurons, each possessing about 10,000 synapses distributed over each dendrite tree with an average number of synapses on the axon of one neuron again being about 10,000 [Fogel 1999]. The brain has the capability of organizing neurons to perform certain computations (e.g. pattern recognition, perception, and motor control) many times faster than the fastest digital computer in existence today [Haykin 1994]. In its most general form a neural network can be loosely thought of as a machine that is designed to model the way in which the brain can perform a particular task or function. As the human brain learns by experience so does the neural network. One classification system for ANNs is based on the way in which the network learns or trains. This results in three categories; supervised learning, reinforcement learning and self organized or unsupervised learning. As its name implies, supervised learning is performed under the supervision of an external "teacher". Reinforcement learning involves the use of a "critic" that evolves through a trial-and-error process. Unsupervised learning is performed in a self-organized manner in that no external teacher or critic is required to instruct synaptic weight changes in the network [Haykin 1994]. The weights change as a result of the nature of the data present. Kohonens self-organizing maps are an example of this style of network [Kohonen 1989]. The experiments investigated in this thesis used a supervised training algorithm.

2.2.1 Neural network weight initialization

Generally before training the weights of a neural network are initialized with small random numbers that are uniformly distributed. This form of initialization can lead to long training times and convergence to a non optimal solution as the random

numbers have no relationship to the problem to be solved. Proper initialization is one of the most important prerequisites for fast convergence of feedforward neural networks [Thimm 1997]. Several schemes have been developed to initialize the weights of a MLP network.

Yam and Chow proposed a MLP weight initialization scheme using the Cauchy Inequality and linear algebra. The proposed algorithm ensures that the output from the hidden units are in the active region where the derivative of the activation function has a large value. Using the outputs from the last hidden layer and the given output patterns, the optimal values of the last layer of weights are evaluated by a least squares method. This method reduces the initial network error while preventing the network from getting "stuck" with the initial weights [Yam 2000]. Thimm and Fiesler investigate various weight initialization methods by examining their performance on eight real-world benchmark problems in the form of an online back propagation [Thimm 1997]. Nguyen and Widrow speed up the training process by setting the initial weights of the hidden layer so that each hidden node is assigned to approximate a portion of the range of the desired function at the start of training [Nguyen 1990]. The weights are evaluated by approximating the activation function with piece-wise linear segments. Then the thresholds of neurons are selected by assuming the input variables to be between –1 and +1. Other researchers have developed several other weight initialization schemes. Some of them can be found in [Wessels 1992, Thimm 1995, Kolen 1991, Denoeux 1993, Weymaere 1994].

A "problem knowledge" based weight initialization scheme is developed in this thesis. Rules representing problem knowledge are generated and used to initialize the weights of a neural network. These rules are in a "If *antecedent* then *consequent*" form as described in section 2.1.2. The rules are implemented using a fuzzy neural network architecture. A complete description of the network initialization method is described in chapter 3.

2.2.2 Architecture of a Fuzzy Neural Network

A fuzzy neural network (FuNN) is a connectionist model for fuzzy rules implementation and inference [Kasabov 1996]. The fuzzy neural network model is designed to be used in distributed and eventually agent based environments. The architecture facilitates learning from data and approximate reasoning, as well as fuzzy rule insertion and extraction. It allows for the combination of both data and rules into one system, thus producing the synergistic benefits of the two sources [Kasabov 1997]. Figure 2.3 shows the FuNN architecture. As the figure shows the FuNN architecture consists of five layers of neurons with partial feedforward connections. The structure is basically a MLP network with two additional layers, the fuzzification and defuzzification layer.

The input variables used for training are "crisp" or ordinary values. The input layer of neurons represents these input variables. These values are fed to the condition layer which performs fuzzification by triangular membership functions whose centers (fuzzy labels) are the weights into the condition layer. The triangles are completed with the minimum and maximum points attached to adjacent centers, or shouldered in the case of the first and the last membership functions. The weights from the input to the nodes in the condition layer are in the range {0,1}. Initially the membership functions are spaced equally over the weight space. If available, expert knowledge can be used to initialize the spacing of the membership functions. An important aspect of this layer is that different inputs can have a different number of membership functions. As a simple example the height of a person can be divided into five different membership functions representing the range from short to tall, while the status of a light switch can have two membership functions representing on or off.

In the rule layer each node represents a single fuzzy rule. The rule layer is identical in structure and operation to a hidden layer of a conventional MLP network. The activation of a rule layer node is the degree to which input data matches the antecedent component of an associated rule. A bias connection to the rule nodes is optional.

Outputs from the rule layer are fed to the action layer. In this layer each node represents a fuzzy label from the fuzzy quantisation space of an output variable, for example "small", "medium" or "large". The activation of the node represents the degree to which this membership function is supported by the current data.

The output layer performs a modified center of gravity defuzzification. Singletons representing centers of triangular membership functions, as it was in the case for the input variables, are attached to the connections from the action layer to the output layer. Linear activation functions are used here.

2.2.3 FuNN training methods

There are three methods to update the weights of a FuNN.

1. A partially adaptive version where the membership functions of the input and the output variables do not change during training. The MLP section is trained using a modified back propagation algorithm.
2. A fully adaptive version where changes can be made to both rules and membership functions.
3. A partially adaptive version where the membership functions are adapted using genetic algorithms.

The FuNN used for the experiments was coded in MATLAB. MATLAB has a slow code execution time, as it is an interpreter, so the FuNN was coded according to the adaptation method of point 1.

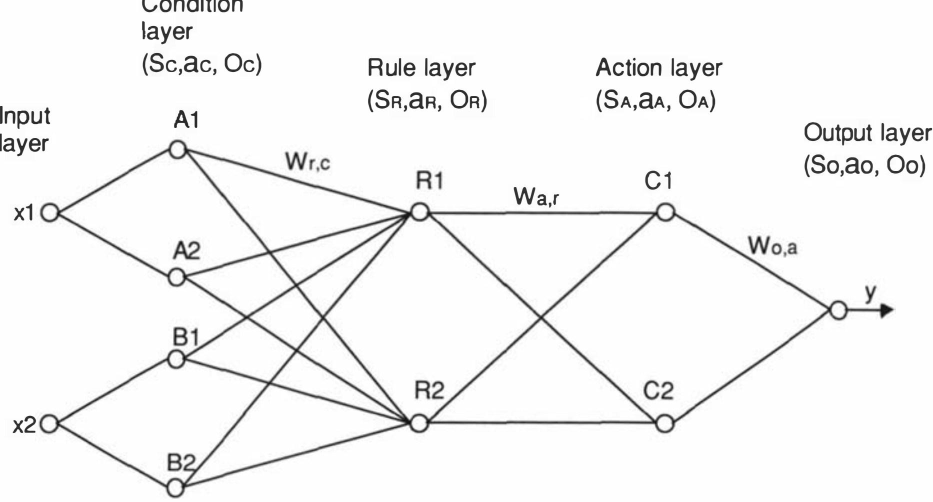


Figure 2.3 Fuzzy neural network architecture.

2.2.4 The training algorithm

This section describes the algorithm used for FuNN training. Kasabov's notation is used for the equations describing FuNN training [Kasabov 1997]. The equations are numbered from k*1* to k*n*. The algorithm is discussed in terms of layers of neurons as they are the focus of the fuzzy architecture.

2.2.4.1 Forward pass

This phase computes the activation values of all the nodes in the network from the first to the fifth layer. In this section a superscript indicates the layer and a subscript indicates a connection between layers.

Layer 1 (Input layer)

The input layer of neurons represents the input variables as crisp values. These values are fed to the condition layer which performs fuzzification.

Layer 2 (Condition layer)

The output from this node is the degree to which the input belongs to the given membership function. The input weight to the condition node represents the center for that particular membership function with the minimum and maximum determined using the center of the adjacent membership functions. In the case of the first and last membership function for a variable a shoulder is used instead. Thus this layer acts as a fuzzifier. Each membership function is triangular and an input signal activates only two neighbouring membership functions simultaneously. The sum of the grades of these two neighbouring membership functions for any given input is always 1. For a triangular membership function the activation functions (Act) for a node i are:

If $a_i < x < a_{i+1}$ then $Act_i^c = 1 - \frac{x - a_i}{a_{i+1} - a_i}$ (k1)

If $a_{i-1} < x < a_i$ then $Act_i^c = 1 - \frac{a_i - x}{a_i - a_{i-1}}$ (k2)

If $x = a_i$ then $Act_i^c = 1$, where a_i is the center of the triangular membership function.

Layer 3 (Rule layer)

The connections from the condition layer to the rule layer are used to perform pre-condition matching of fuzzy rules. The connection weights in this layer may be set either randomly or according to a set of rules. The net input and activation are respectively:

$Net^r = \sum w_{rc} \times Act^c$ (k3)

$Act^r = \frac{1}{1 + e^{(-g \times Net^r)}}$, where g is the gain factor. (k4)

Layer 4 (Action layer)

The nodes in this layer and the connection weights function as those in layer 3 for net input and activation:

$$Net^a = \sum w_{ar} \times Act^r \qquad \text{(k5)}$$

$$Act^a = \frac{1}{1+e^{(-g \times Net^a)}} \qquad \text{(k6)}$$

Layer 5 (Output layer)

This layer performs defuzzification to produce a crisp output value. The Center of Gravity (COG) defuzzification is used to convert from fuzzy to crisp.

$$Net^o = \sum w_{oa} \times Act^a \qquad \text{(k7)}$$

$$Act^o = \frac{Net^o}{\sum Act^a} \qquad \text{(k8)}$$

2.2.4.2 Backward pass

The goal for this phase is to minimise the error function *E*:

$$E = \frac{1}{2} \sum (y^d - y^a)^2 \qquad \text{(k9)}$$

where y^d is the desired output and y^a is the current output.

Hence the general learning rule (gradient descent) used is,

$$\Delta w \approx -\frac{\partial E}{\partial w}$$

$$w_{t+1} = w_t + \eta(-\frac{\partial E}{\partial w}) + \alpha(\Delta w_t) \qquad \text{(k10)}$$

where η is the learning rate and α is the momentum coefficient, and

$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial Net} \times \frac{\partial Net}{\partial w} = -\delta \times Act$$

Hence the weight update rule is:

$$\Delta w_{t+1} = \eta \delta \times Act + \alpha \Delta w_t \qquad \text{(k11)}$$

Layer 5 (Output layer)

$$\delta^o = -\frac{\partial E}{\partial Net^o} = -\frac{\partial E}{\partial Act^o} \times \frac{\partial Act^o}{\partial Net^o} = y^d - y^a \qquad \text{(k12)}$$

When the weights are adapted the constraining rule is taken into account which imposes restrictions to the change of the centers of the membership functions.

Layer 4 (Action layer)

The error for each node in the action layer is calculated individually based on the output error and on the activation of this node having in mind the type of membership functions (triangular) used in the defuzzification layer as well as the type of defuzzification.

If $a_i < y < a_{i+1}$ then $d^a = 1 - \frac{y - a_i}{a_{i+1} - a_i}$ (k13)

If $a_{i-1} < y < a_i$ then $d^a = 1 - \frac{a_i - y}{a_i - a_{i-1}}$ (k14)

If $y = a_i$ then $d^a = 1$

Hence,

$$\delta^a = f(Net^a) \times E^a = Act^a(1 - Act^a) \times (d^a - Act^a) \qquad \text{(k15)}$$

Layer 3 (Rule layer)

$$\delta^r = Act^r(1 - Act^r) \times \sum (w_{ar} \times \delta^a) \qquad \text{(k16)}$$

Layer 2 (Condition layer)

The weight w_{ic} is assigned as follows. If x^i lies in the fuzzy segment, then the corresponding weight should be increased directly proportionally to the propagated error from the previous layer as the error is caused by the weight. This proposition can be represented by the following equation:

$$\delta^c = Act^c \times \sum (w_{rc} \times \delta^r) \qquad \text{(k17)}$$

Thus the weight updating rule for this layer is:

$$w_{ic_{t+1}} = \eta\delta^c + \alpha\Delta w_{ic_t} \qquad \text{(k18)}$$

The new centers of the input triangular membership functions are also adjusted according to a partition range as for the output layer.

2.3 Image processing

The term image processing refers to the process involved in the modification of images for two somewhat different purposes [Russ 1995]:

1. improving the visual appearance of images to a human viewer (visual enhancement).
2. using images for measurement of the features and structures present (information extraction).

There is considerable overlap between the techniques that are appropriate for each task. For visual enhancement it is helpful to have a familiarity about the human visual system, and an appreciation of what cues the viewer responds to in an image [Pugmire 1995]. This may include both cosmetic and informational changes. When the image is to be processed for information extraction, the type of measurement depends on whether it is global or local; global information extraction will be performed on the entire image while local information extraction will be performed on individual features. Statistical analysis, use of transformation operators and various filters are some of the operations used for image enhancement and pre-processing. A window filter is a type of local operator used in the early stages of many image processing algorithms. They are extensively used for reducing noise, enhancing contrast, edge detection and for producing cosmetic effects such as embossing.

2.3.1 The window filter

The term "window filter" refers to the class of local operators in which the output at each position in an output image is defined as a function of the pixel in a window surrounding the equivalent position in the input image. The window filter

can be thought of as a unit that has multiple inputs and a single output and maybe any shape or size. This window filter is scanned across the input image to build an output image (figure 2.4). The filtering operation can be based on simple linear functions, non-linear functions or rule based functions.

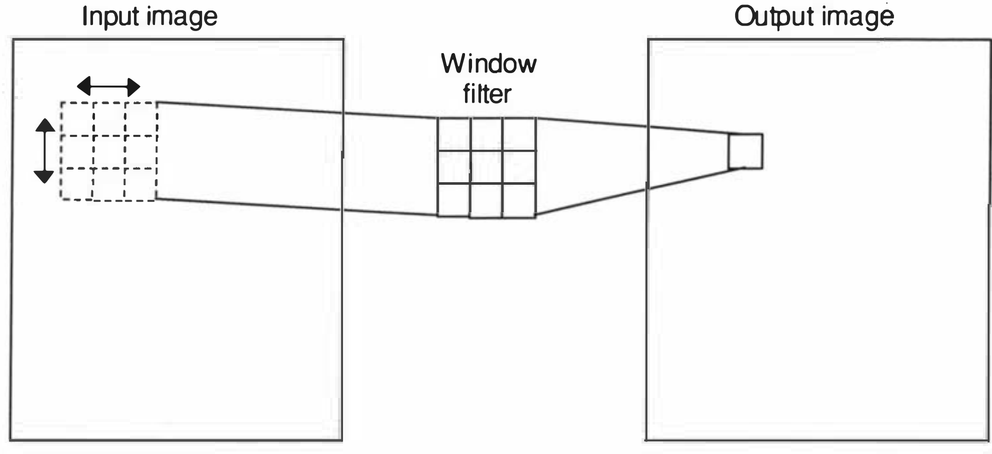


Figure 2.4 Operation of a window filter.

Figure 2.5 shows the output generated by using a common edge detector and an embossing filter [Emboss 5].

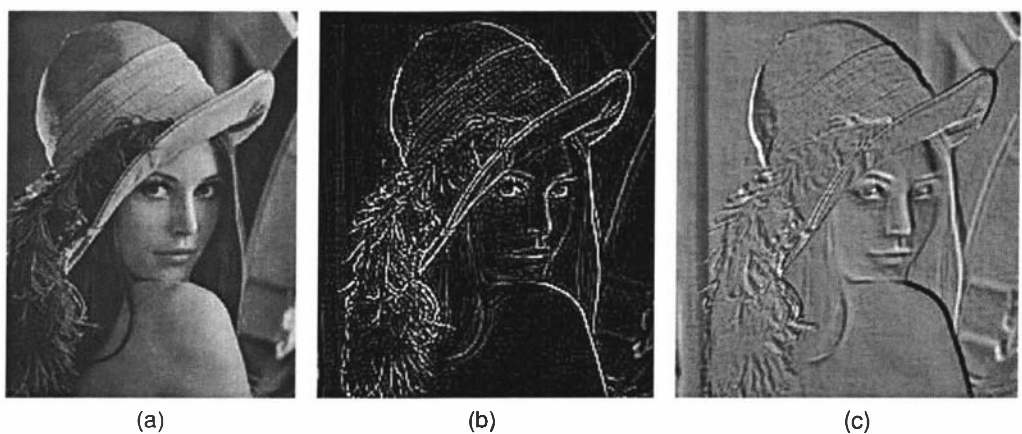


Figure 2.5 Example of filter operation. Image (a) is the original image. Image (b) was generated by passing a common edge detection filter over image (a). Image (c) was generated by passing an embossing filter over image (a).

2.4 The fuzzy neural network window filter

The selection of an appropriate window filter for image processing is usually an empirical process, which relies on skill and experience [Siroki 1998]. In many practical applications the design of an appropriate window filter is a difficult task. Sometimes the application of conventional image processing filters does not produce satisfactory results. When dealing with many practical problems it is difficult to formulate an algorithmic process that can be used to solve an image processing problem. In such a situation it may be easier to train a neural network to perform the image processing task. Here, a neural network can be used to perform the filter operation. The filter operates in one of two modes; the training mode and the run mode. In addition to the normal input and output images associated with a window filter, the neural network window filter (NNWF) [Pugmire 1995] requires a target image that is only used during training. Once trained, the NNWF can be used to "filter" other images. The target image is usually generated by a domain expert or an image processing expert. Figure 2.6 shows the operation of a NNWF. A fuzzy neural network window filter (FuNNWF) is used for the image processing problems described in this thesis. This structure is used as problem knowledge can be used to initialize the weights of the FuNNWF before training.

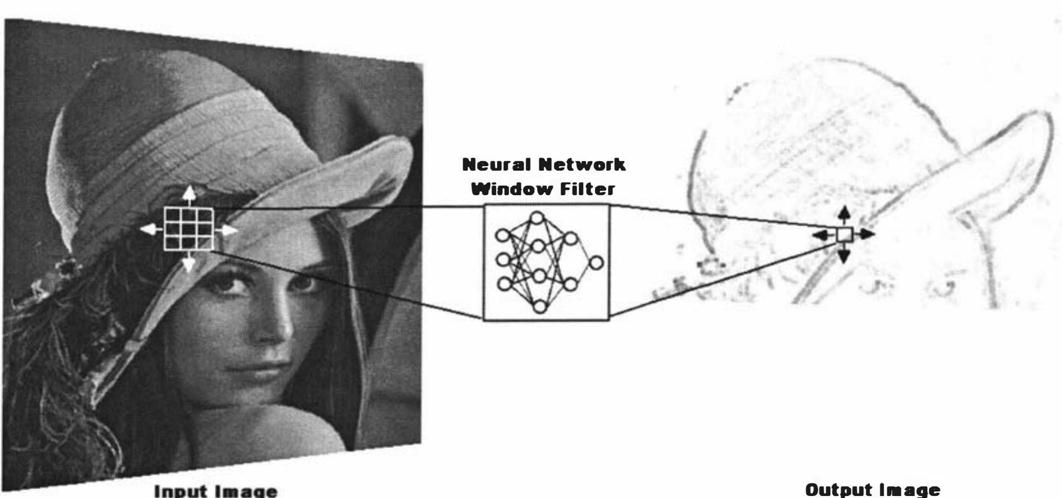


Figure 2.6 Neural network window filter.

2.4.1 Structure of the FuNNWF

Figure 2.7 shows the structure of the FuNNWF. Each element in the window provides an input to the input layer. The number of input nodes in the input layer is equal to the number of pixels in the window. The number of nodes in the condition and action layer depends on the available problem knowledge. The number of nodes in the rule layer is arbitrary and should be set by the user.

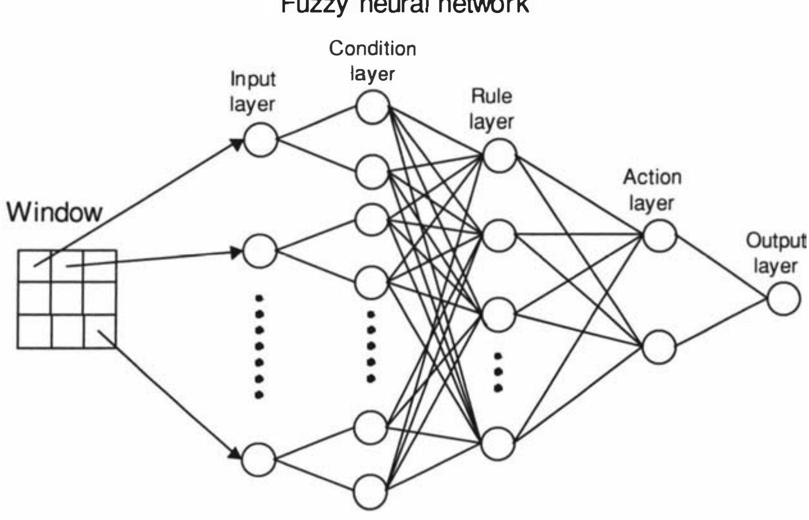


Figure 2.7 Fuzzy neural network window filter structure.

2.5 Putting it all together

The previous sections in this chapter described fuzzy logic, artificial neural networks and image processing. These three areas were combined to produce a FuNNWF. The thesis investigates the effects on training when problem knowledge is used to initialize the weights of a FuNNWF.

2.5.1 Problem knowledge

Knowledge is the information which represents long term relationships, that is ways of doing things, common sense, ideas, methods, skills and so forth. In the context of this thesis problem knowledge can be defined as the steps involved in solving a problem. A complete description of problem knowledge (PK) is described in chapter 3.

2.5.1.1 An example of PK

In an image, a vertical edge is detected when there is a transition in intensity in a window in the horizontal direction from low to high or vise versa. This is the PK defining vertical edge detection. The PK can be represented by the two rules,

1. If the intensity of the pixels on the left side of the window is low and the intensity of the pixels on the right side of the window is high then a vertical edge has been detected.
2. If the intensity of the pixels on the left side of the window is high and the intensity of the pixels on the right side of the window is low then a vertical edge has been detected.

These rules are then used to initialize the weights of the FuNNWF.

2.6 Environment

The software and hardware used for the experiments in this thesis is described in this section.

2.6.1 Software

2.6.1.1 FuNN software

As explained in section 2.2.2 the FuNN architecture is a MLP network with two additional layers, the fuzzification and defuzzification layers. Initially the fuzzification and defuzzification process was coded in Matlab. Then to improve the speed of the FuNN, the fuzzification and defuzzification modules were written in standard C and then compiled as a dynamic link library (DLL) so they could be called from within Matlab. This greatly improved execution time.

Matlab is an interpreter, so the operation of the MLP section in the FuNN was written in standard C and compiled using Microsoft Visual C++ to determine if an improvement in speed could be achieved during training. Subsequently it was discovered that the MLP network in the Matlab Neural Network Toolbox (NNT) [Demuth 1998] performed as well as the network written in C. So the fuzzification and defuzzification DLLs and the MLP network in the NNT were combined to

generate the FuNN, extending the NNT. The code for the FuNN can be found in Appendix A.

2.6.1.2 Parser software

An application was developed to generate the rule and action layer matrices using rules representing PK. This application parses each rule in the rules file to generate a matrix representing PK. The complete code for the rule parser is available in Appendix A.

2.6.1.3 Image processing software

The following software was used for image processing.

- Scion Image from Scion Corporation (http://www.scioncorp.com).
- Painshop Pro from Jasc Software (http://www.jasc.com).
- Matlab Image Processing Toolbox.

2.6.2 Hardware

A PC hardware platform was used for all the experiments. A wide range of machines was used for the simulations. Details of the machines,

1. A Pentium running at 200Mhz was used to write the code for each simulation.
2. The simulation code was then deployed on neural network farms comprising of,
   - Pentium II running at 400MHz (6 machines).
   - Pentium III running at 550MHz (8 machines).
- Pentium III running at 850MHz (6 machines).

2.6.3 Simulations

Several simulations were performed during the course of this research.

- A maximum of 20 machines were used. For some simulations all 20 machines were utilized.

- Over 5000 simulations were performed.
- The simulations used close to 5,000,000 epochs.

2.7 Summary

- This chapter has briefly introduced fuzzy logic, neural networks and window filters and how they were combined to form the fuzzy neural network window filter. The fuzzy neural network architecture allows the insertion of problem knowledge in the form of fuzzy "If ... then ..." rules.
- If a casual relationship exists between the input and target image a FuNNWF can be trained to simulate that relationship so it can be used as a filter to process other images.
- An advantage of using a neural network for image processing is that the required task can be defined by example, this eliminates the need to generate an explicit algorithm for each new task. Examples can be produced by hand by a specialist in that particular application area or by an image processing expert.

Chapter 3

3 Rule mapping procedure

Neural network training is an optimization task [Rumelhart 1986, Lodewky 1992]. The initial state of the network is important to this task [Nguyen 1990, Thimm 1995, Weymaere 1994, Yam 2000, Denoux 1993]. The initial state of the network will govern the number of iterations needed and whether the solution is a global or local optimum. In general practice the weights of a neural network are initialized with small random numbers [Nguyen 1990]. The convention in the field has been to choose initial weights with a uniform distribution between plus and minus ρ, usually set to 0.5 or less [Kolen 1991]. The random numbers used to initialize the network have no relationship to the problem to be solved. Once a network is trained the network weights can be used to extract rules that represent the problem. If these rules could be used to initialize the weights of a neural network it would be expected that the network would train more quickly and possibly to a better solution. A method that makes it possible to map rules into the weights of a fuzzy neural network (FuNN)[Kasabov 1996, Kasabov 1997] has been developed as a key element of the research reported in this thesis. This chapter describes the rule development process.

3.1 Problem knowledge

Knowledge is the information which represents long term relationships, that is ways of doing things, common sense, ideas, methods, skills and so forth. Knowledge is "condensed" information, "squashed" information, an extraction of the essence of things [Kasabov 1996]. Over the years of a lifetime humans have accumulated vast amounts of knowledge that enables them to perform both special and day to day tasks. Given a problem a human can usually lay out a step by step approach to solve it. For example if a human is given the task of finding and counting the circles in an image (Figure 3.1(a)) containing circles (all the circles are approximately the same size) and other geometric shapes, the task would be defined in high level or human level terms as,

- Scan the image from left to right, top to bottom.
- Look for shapes that are circular.

The high level algorithm described above can be considered to be an illustration of problem knowledge.

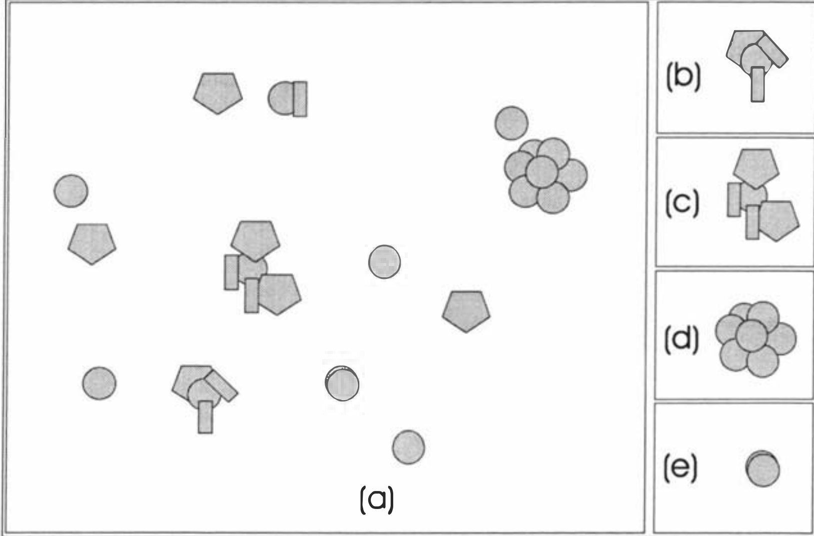


Figure 3.1 (a) Image containing circular objects and other geometric shapes, all the objects are outlined for clarity. (b) – (e) clusters of objects extracted from (a).

Before a problem can be solved the tasks required to solve the problem should be identified. Often the first step is to extract "obvious" problem knowledge from

the task to be solved. Considering the previous example the obvious approach would be,

- Look for shapes that are circular.

A high confidence can be placed on this approach since it is very obvious at least to a human. Problems arise when looking for circular objects in Figure 3.1(b)-(e). Here circular objects overlap with other circular objects and different geometric shapes.

If the objects in Figure 3.1 (a) were not outlined the following errors may occur when counting circular objects:

- Figure 3.1 (c) is a circular object surrounded by non-circular shapes. It could be mistaken for a polygon.
- Figure 3.1 (d) shows a circular object surrounded by other circular objects. The center object may be missed due to overlapping.
- Figure 3.1 (e) shows two circular objects that are nearly coalesced. This could be mistaken as one object.

The previous points show the problems associated in trying to count overlapping circular objects. Although a human may be able to identify the circular objects many ambiguities arise when attempting to develop an algorithm or descriptive knowledge that will enable a computer to perform the task. If it gets progressively more complicated to describe the problem then that knowledge can be called "the not so obvious problem knowledge".

As described above, problem knowledge may be extracted in the following order,

1. extract "obvious" problem knowledge first. High confidence can be placed in this, as it is less ambiguous.
2. next extract "not so obvious" problem knowledge. High confidence cannot be placed on this, as it may or may not be ambiguous.

The problem knowledge or high level tasks that define the problem are then converted to low level rules. These rules are then mapped to the weights of a fuzzy

neural network. The rule mapping procedure is described in the following sections.

3.2 Mapping to FuNN architecture

As described in the previous chapter the FuNN architecture can be seen as an extension of the standard MLP network with two hidden layers made by adding two layers, namely the fuzzification and defuzzification layers [Kasabov 1997]. The FuNN architecture allows:

- insertion of expert knowledge about the particular problem as rules in IF THEN form.
- semantic interpretation of the network operation through a rule extraction process.

The thesis deals only with rule insertion. Through the rule insertion process, expert knowledge about the problem can be used to assist network training. The rule insertion involves presetting some or all of the connection weights to and from the rule layer of the fuzzy neural network. Depending on the problem, preset weights can be kept static or allowed to change during training. The aims of the rule insertion process are three fold:

- to start training from a position in weight space which is closer to the final solution.
- to cover those regions of the input-target space for which no training data are available. For such training data, the preset weights can be kept fixed during training.
- to increase the robustness of the network when the training data has a high degree of uncertainty.

The set of rules used need not be complete in a logical sense as long as they are an approximate description of the problem. When a set of rules can be derived that represent the problem knowledge completely, the FuNN is set to have the number of rule nodes equal to the number of rules and the action nodes equal to the number of actions. In such a situation all the weights in the network are preset (including

the centers of the membership functions). Network training in this case can be seen as a fine tuning process using input-target correspondence data. When a complete rule set cannot be derived from the available problem knowledge the number of rule nodes is made larger than the available number of rules. This is done to ensure that the network has enough freedom to adapt. In this case the weights associated with the rules are preset and the remaining weights are initialized with random numbers.

3.3 Rule insertion procedure

The basis of this procedure is that rules in most problems are expressed as crisp rules even when known to be fuzzy. In crisp rules the consequence is considered to be the outcome of the rule when the inputs have fuzzification layer symbol values that satisfy the rule antecedents. These crisp values determine the initial weights and network training is performed to render them fuzzy. One method of ensuring that the rules are crisp at the symbol values is to make the appropriate nodes saturate with the correct outcome when the inputs have the symbol values. This is achieved if the nodes for the rules are driven close to saturation. Currently allowed rules have product (AND) terms or sum (OR) terms only. The procedure described here is applicable to FuNNs with an unlimited number of symbols (membership functions) for each input. Here we will consider only inputs having two labels. These are called Low and High and are determined by membership functions 0 and 1 respectively. It is presumed that all inputs are normalized to be in the range [0,1]. Weights connected to a rule node are labeled with the symbol **ON** or **OFF**. The labels indicate an antecedent condition to be met in the rule associated with the node. **ON** weights should drive the node to positive saturation (node output $\rightarrow$ 1) and **OFF** weight to negative saturation (node output $\rightarrow$ 0).

The antecedent part of the IF THEN rules are mapped to the rule layer weights while the consequent parts are mapped to the action layer weights. The mapping procedure is slightly different for the rule and action layers. If the weights are to be trained from these initial values then in order that there be sufficiently enough variation in the initial weights, small random values should be added to those

determined by the rule mapping. If all the weights start out with equal values and if the solution requires that unequal weights be developed, the system can never learn. This is because the error is propagated back through the weights in proportion to the values of the weights. This means that all hidden units (rule layer neurons) connected to the output units (action layer neurons) will receive identical error signals, and, since the weight changes depend on the error signal, the weights from those units to the output units must always be the same. The system is starting out at a kind of *local maximum*, which keeps the weights equal, but it is a maximum of the error function, so once it escapes it will never return [Rumelhart 1986].

Two methods of mapping have been developed. They are:
1. Boolean logic rule mapping
2. Conditional rule mapping.

The Boolean logic mapping was developed by a postdoctoral fellow Dr. Mladen Siroki and the related Massey University group. The mapping procedure was extended to a conditional mapping in the subsequent research work carried out by the author. The following notation for the equations is the one used by the Massey group. The Boolean logic rule mapping procedure is also presented in [Kim 2000].

3.3.1 Boolean logic rule mapping

3.3.1.1 Weight mapping for the rule layer

AND rules

For AND rules the associated rule layer node will be driven to positive saturation (rule node output is close to one) when ALL the antecedent terms are true and driven to negative saturation (rule node output is close to zero) otherwise.

Positive saturation:

$$N \times ON \geq Sat \tag{1}$$

Negative saturation:

$$OFF + (N-1) \times ON \leq -Sat \tag{2}$$

Note: This is the worst case where with one **OFF** and N-1 **ON**.

Where N is the number of inputs used in the rule and Sat is a saturation value, which depends on the gain factor of the activation function used. This set of linear inequalities leads to the following equations for calculating **ON** and **OFF** rule layer weights,

$$ON = \frac{Sat}{N} \tag{3}$$

$$OFF = -Sat\left(1 + \frac{N-1}{N}\right) \tag{4}$$

OR rules

For OR rules, the rule layer node should be positively saturated if ANY of its terms is true, and negatively saturated if ALL terms are false. Thus:

Positive saturation:

$$ON + (N-1) \times OFF \geq Sat \tag{5}$$

Negative saturation:

$$N \times OFF \leq -Sat \tag{6}$$

This set of linear inequalities lead to the following equations for calculating ON and OFF rule layer weights:

$$ON = Sat\left(1 + \frac{N-1}{N}\right) \tag{7}$$

$$OFF = -\frac{Sat}{N} \tag{8}$$

3.3.1.2 Weight mapping for action layer

Dependent on the problem, rules can be exclusive or inclusive. In exclusive rules, only one rule can be activated at a time, for inclusive rules more than one rule can be activated simultaneously. For a problem with exclusive rules, action layer weights are set as follows:

$$ON = Sat \tag{9}$$

$$OFF = -Sat \tag{10}$$

For a problem with inclusive rules, action layer weights are set by the same procedure as rule layer weights (equations 1-8) with the difference, that N is the number of possible simultaneous rule layer units [Siroki 1998].

3.3.1.3 Example using Boolean logic mapping

Rule no.	Description
1	If I1=LO and I2=LO then Output=LO
2	If I1=HI and I2=LO then Output=HI
3	If I1=LO and I2=HI then Output=HI
4	If I1=HI and I2=HI then Output=LO

Table 3.1 Rules for Boolean logic mapping example.

Table 3.1 shows four rules. Two membership functions LO and HI are used. The rules are AND rules and each rule uses two nodes (N=2). If a saturation value (Sat) of three was used, using equation (3) and (4),

$$ON = \frac{3}{2} = 1.5$$

$$OFF = -3(1 + \frac{2-1}{2}) = -4.5$$

The weights in the rule layer can be set using the value for *ON* and *OFF*.

As the rules are exclusive, using equations (9) and (10),

$$ON = 3$$

$$OFF = -3$$

The weights in the action layer can be set using these values of *ON* and *OFF*.

Table 3.2 shows the final weight matrices for the rule and action layer.

Rule no	Input 1		Input 2		Output	
	LO	HI	LO	HI	LO	HI
1	1.5	-4.5	1.5	-4.5	3	-3
2	-4.5	1.5	1.5	-4.5	-3	3
3	1.5	-4.5	-4.5	1.5	-3	3
4	-4.5	1.5	-4.5	1.5	3	-3

Table 3.2 Weight matrices for Boolean logic rule mapping example.

3.3.2 Conditional rule mapping

3.3.2.1 Derivation of relational operator mapping

Figure 3.2 is a section of a FuNN network showing a single input A, its associated symbol layer nodes and a rule layer node.

C_0 to C_n in figure 3.1 are the center points of the linearly spread encoded symbols S_0 to S_n.

n+1 is the number of such symbols for each input.

W_0 to W_n are the weights between the rule layer node and each symbol.

The value of each S_i (i = 0..n) is the degree to which A matches C_i and has a range [0,1].

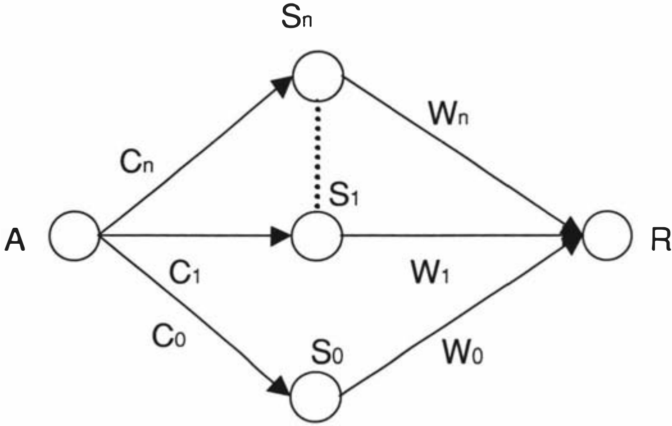


Figure 3.2 Fuzzy neural network section.

The output of the rule node R is:

$$R = f(\phi) \qquad [0,1]$$

$$\phi = \sum_{i=0}^{n} S_i W_i = S_0(A) \times W_0 + S_1(A) \times W_1 + \ldots + S_n(A) \times W_n$$

and f is the logistic function.

R is said to be saturated if it is greater than PS (positive saturation) or less than NS (negative saturation) (Figure 3.3). The corresponding values of ϕ are $\pm Sat$ (where Sat is the saturation value).

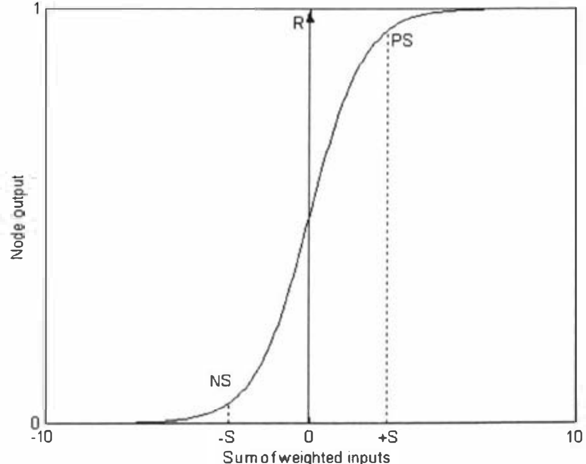


Figure 3.3 Rule node output function.

3.3.2.2 Derivation of rule layer mapping

Rule: If (A > (Contsant = C_i)) then R ≥ PS else R ≤ NS

R should reach PS when A = C_i and remain at or above this if A > C_i.

R should reach NS when A = C_{i-1} and remain at or below this if A < C_{i-1}.

Consider,

1) A = C_{i-1} then $S_{i-1}(A) = 1$ and $S_i(A) = 0$

$\therefore\ \phi = W_{i-1} \le -Sat$ to ensure $R \le NS$.

2) A = C_i

$\therefore\ \phi = W_i \ge +Sat$ to ensure $R \ge PS$.

3) A = C_{i+1}

$\phi = W_{i+1} \ge +Sat$.

4) A = C_{i-2}

$\phi = W_{i-2} \le -Sat$.

This set of inequalities will be satisfied if,

$W_k = -Sat \qquad k = 0, 1, \dots, i\text{-}1.$

$W_k = +Sat \qquad k = i, i+1, \dots, n.$

When A lies between the symbol values, the value of ϕ is the result of the linear addition of the values of the symbol and their weights and will be bounded by the value of ϕ when A is equal to the symbol value.

Rule: if (A < (Constant=C_i)) then R ≥ PS else R ≤ NS

The output R should be ≥ PS if $A \leq C_i$ and should be ≤ NS if $A \geq C_{i+1}$.

By similar reasoning to that in the previous rule we get,

$W_k \geq +Sat \qquad k = 0, 1, \dots, i.$

$W_k \leq -Sat \qquad k = i+1, \dots, n.$

Rule: if ($C_i < A < C_j$) then R ≥ PS else R≤ NS when j≥i

Combining the two previous rules gives,

$W_k \geq +Sat \qquad k = i, i+1, \dots, j.$

$W_k \leq -Sat \qquad k = 0, 1, i\text{-}1, \dots, j+1, j+2, \dots, n.$

Note: when i=j the rule is equivalent to the rule ($A = C_i$) then R≥PS.

Combined inputs,

Conjunctive AND

Rule: if (($A_1 > C_{1,i}$) AND ($A_2 > C_{2,j}$) ... AND ($A_m > C_{m,l}$)) then R ≥ PS

Where $C_{i,j}$ is the center for the jth symbol for input i and $W_{i,j}$ is the weight from the jth symbol of input i to the rule node R.

When $A_1 = C_{1,i}$, $A_2 = C_{2,j}$, ... , $A_m = C_{m,p}$ then

$$\phi = W_{i,1} + W_{2,j} + \ldots + W_{m,p} \geq +Sat .$$

On the ground that there is no evidence that any input is more important than any other, we may choose all the above weights to be equal.

$$\therefore W_{1,i} = W_{2,j} = \cdots = W_{m,p} \geq \frac{Sat}{m}$$

Note that the above equation for ϕ is $\geq$ S for,

A_1 is $\{C_{1,i}, C_{1,i+1}, \ldots , C_{1,n}\}$

A_2 is $\{C_{2,i}, C_{2,i+1}, \ldots , C_{2,n}\}$

$\vdots \qquad \vdots$

A_m is $\{C_{m,i}, C_{m,i+1}, \ldots , C_{m,n}\}$

and we may choose all the weights at and above the respective input breakpoints to be equal to $\frac{Sat}{m}$.

If one term of the Rule is false then $\phi \leq -Sat$.

Let $A_1 = C_{1,i-1}$ and $A_2 = C_{2,j} \ldots A_m = C_{m,p}$ then

$$\phi = W_{1,i-1} + W_{2,j} + \ldots + W_{m,p} \leq -Sat$$

$$\therefore W_{1,i-1} \leq -Sat - \frac{(m-1)}{m} Sat$$

$$\therefore W_{1,i-1} \leq -\frac{(2m-1)}{m} S$$

Note: If more than one term is false the output R will be driven beyond –NS.

Once again taking all inputs identically, we get that all the weights below the respective input break point are equal to $\frac{(2m-1)}{m}Sat$.

Formally ,

$$W_{l,k} \leq -\frac{(2m-1)}{m}Sat \qquad (11)$$

for k less than the break point index,

and

$$W_{l,k} \geq \frac{Sat}{m} \qquad (12)$$

for k greater than or equal to the break point index,

for l=1,2,...,m.

Rule: if ($A_1 < C_{1,i}$) and ($A_2 < C_{2,i}$) and ($A_m < C_{m,p}$) then $R \geq PS$

By similar reasoning to the combined > rule above we get,

$$W_{l,k} \geq \frac{Sat}{m} \qquad (13)$$

for k less than or equal to the break point index.

and

$$W_{l,k} \leq -\frac{(2m-1)}{m}Sat \qquad (14)$$

for k greater than the break point index,

for l=1,2,...,m.

Note than > and < terms may be mixed in any one rule. The results will be set accordingly.

Conjugate OR

Rule: if ($A_1 > C_{1,i}$) OR ($A_2 > C_{2,j}$) OR … OR ($A_m > C_{m,p}$) then R ≥ PS

Let $A_1 = C_{1,i-1}$, $A_2 = C_{2,j-1}$, …, $A_m = C_{m,p-1}$ then

$$\phi = W_{1,i-1} + W_{2,j-1} + \ldots + W_{m,p-1} \leq -Sat$$

Choose

$$W_{1,i-1} = W_{2,j-1} = \ldots = W_{m,p} \leq -\frac{Sat}{m}$$

If any one term is true the Rule is true.

Let $A_1 = C_{1,i}$, $A_2 = C_{2,i-1}$, …, $A_m = C_{m,p}$ then

$$\phi = W_{1,i} + W_{2,j-1} + \ldots + W_{m,p-1} \geq Sat$$

$$\therefore W_{1,i} \geq \frac{(2m-1)}{m} Sat$$

Thus

$$W_{l,k} \leq -\frac{Sat}{m} \tag{15}$$

for k less than the break point index

and

$$W_{l,k} \geq \frac{(2m-1)}{m} Sat \tag{16}$$

for k greater than or equal to the break point index,

for l=1,2,…,m

Rule: if ($A_1 < C_{1,i}$) OR ($A_2 < C_{2,j}$) OR … OR ($A_m < C_{m,p}$) then R ≥ PS

By similar argument to the rule above we get

$$W_{l,k} \geq \frac{(2m-1)}{m} Sat \qquad (17)$$

for k less than or equal to the break point index

and

$$W_{l,k} \leq -\frac{Sat}{m} \qquad (18)$$

for k greater than the break point index,

for $l=1,2,\ldots,m$.

Again the > and < terms may be mixed in one OR rule and the weights set accordingly.

Note!: Rules with mixed AND and OR terms cannot be satisfied within a single rule. The logic operations AND and OR are performed by the single operation of addition. These two logic operators (AND, OR) do not have the same precedence, for example $A.B + C \neq A.(B+C)$. This precedence cannot be modeled with the single addition operator thus disallowing mixed expressions.

Relational operators between two or more inputs

Rule: if $(A_1 > A_2)$ then $R \geq PS$ else $R \leq NS$

For inputs having identical symbols the rule can be expanded as,

$$\begin{aligned}(A_1 > A_2) = \quad & (A_1 > C_1) \text{ and } (A_2 < C_1) \text{ or} \\ & (A_1 > C_2) \text{ and } (A_2 < C_2) \text{ or} \\ & \vdots \\ & (A_1 > C_n) \text{ and } (A_2 < C_n)\end{aligned}$$

This can be mapped using the previous scheme only if there are only two symbols in the linear spread encoding of the inputs, the OR terms do not exist. This will allow for multiple > terms in the expression.

3.4 General derivation of rules

The general derivation of rules is described below.

3.4.1 Rule: Input > Constant

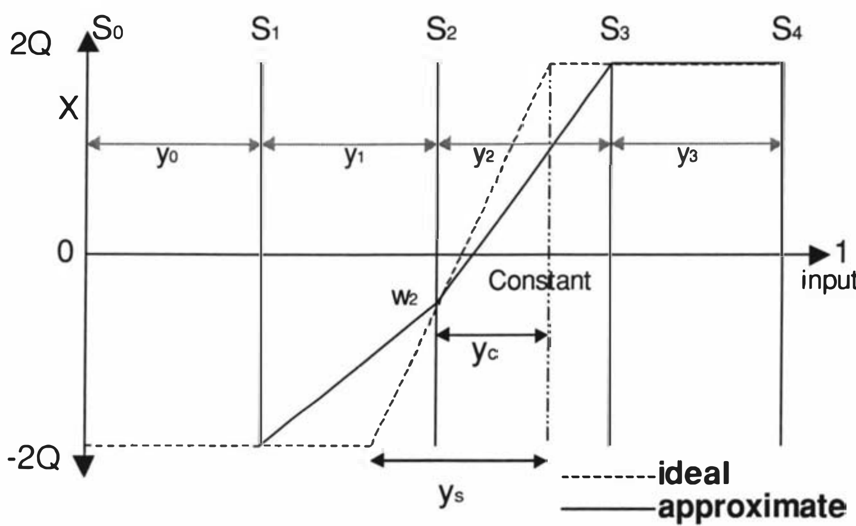


Figure 3.4 Fuzzy function for Input>Constant.

The desired fuzzy function for the internal variable X of the rule node is shown in Figure 3.4 along with a possible approximation to it. S_0, S_1, S_2, S_3 and S_4 are the symbols and 2Q is the saturation as ideal function reaches 2Q when input = constant.

The functions that can be achieved are the composition of the weighted fuzzy functions of the symbols as these functions are linear between the symbol values the composite function can only change the slope at the symbol values. One approximation to the desired function would be to conserve the area under the function. By noting that the internal variable of the rule node when the input is at symbol value is equal to the weight connecting that symbol node to the rule node,

then the expression for the areas are easily obtained from figure 3.4. In order for the two functions to be identical when the constant is at a symbol value (constant = S_j or S_{j-1}) y_s is defined as;

$$y_s = \frac{y_c y_2 + y_2 y_1 - y_c y_1}{y_2}$$

Note $y_s = y_1$ if $y_c = 0$ and $y_s = y_2$ if $y_c = y_2$.

Derivation of y_s:

Assuming that y_s is a linear function of y_c,

$$y_s = Ay_c + B$$

And that y_s=y_1 if y_c=0,

$$\therefore B = y_1$$

And that y_s=y_2 if y_c=y_2

$$A = \frac{y_2 - y_1}{y_2}$$

Substituting for A and B gives,

$$y_s = \frac{y_c y_2 + y_2 y_1 - y_c y_1}{y_2}$$

From Figure 3.4:

Area of ideal = $2Qy_s + (y_2 - y_c)4Q$ (19)

between S_1 & S_3

Area of approximate = $\frac{(2Q + W_2)}{2} y_1 + \frac{(6Q + W_2)}{2} y_2$ (20)

Substituting for y_s in (19),

$$Area = 2Q\left[\frac{y_c y_2 + y_2 y_1 - y_c y_1}{y_2} + 2(y_2 - y_c)\right]$$

$$Area = 2Q\left[\frac{2y_2^2 + y_2 y_1 - y_c(y_1 + y_2)}{y_2}\right] \quad (21)$$

Simplify (20),

$$Area = Q(y_1 + 3y_2) + \frac{(y_1 + y_2)}{2} W_2 \qquad (22)$$

Equating (21) & (22)

$$\frac{(y_1 + y_2)}{2} W_2 = Q\left[\frac{2(2y_2^2 + y_2 y_1 - y_c(y_1 + y_2))}{y_2} - y_1 - 3y_2\right]$$

$$W_2 = 2Q\left[\frac{y_2^2 + y_2 y_1 - 2y_c(y_1 + y_2)}{y_2(y_1 + y_2)}\right]$$

$$W_2 = 2Q\left[1 - \frac{2y_c}{y_2}\right]$$

For inputs with n+1 equidistant symbols the weights can be calculated using:

$$W_i = -2Q \qquad i = 0,1,2,\ldots, j-1 \qquad (23)$$

$$W_j = 2Q(1 - 2ny_c) \qquad (24)$$

$$W_i = 2Q \qquad i = j+1, j+2, \ldots, n \qquad (25)$$

where y_c = Constant -S_j

There is no problem as Constant approaches the upper extremum.

Thus $input > S_n$ is a valid and useful antecedent.

As Constant approaches the lower extremum the function becomes $2Q$ for $input > S_0$.

Thus $input > S_0$ is not a useful antecedent.

The ideal and approximate functions are identical if the constant is equal to a symbol value, when $y_c = \frac{1}{n}$.

The less than relationship can be developed as weights defined similarly.

3.4.2 Rule: Input = Constant

The desired fuzzy function for the equals clause is shown in Figure 3.5. Once again it can be achieved exactly only if the constant equals a symbol value. An approximate function is also shown in Figure 3.5, where the centroid of the approximate and ideal function coincide at the constant. The ideal function has a peak value of 2Q at a point y_c to the right of S_2. At that point both the functions have the value 2Q.

The left and right half areas below the functions and above the –2Q line are equal. The approximate function should pass through the peak of the ideal function and the left and right hand areas should be the same. The expressions for the left and right areas of the function can be derived from Figure 3.5. When equated these can be solved for the weights associated with the symbols on either side of the constant.

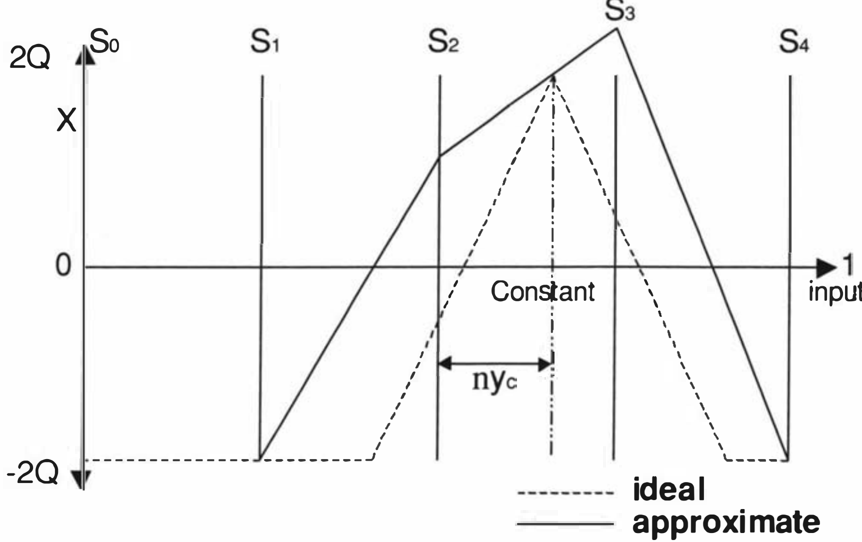


Figure 3.5 Fuzzy function for Input = Constant.

In Figure 3.5 the approximate function intersects S_2 at W_j and S_3 at W_{j+1}.

By similar triangles,

$$\frac{2Q - W_j}{ny_c} = \frac{W_{j+1} - 2Q}{1 - ny_c}$$

The equation can be solved to gives equation (28).

Assuming that the distance between S_i and S_{i+1} = 1 and equating the left and right hand side of the function above the –2Q line gives:

$$\frac{(W_j+2Q)}{2}\times 1+\frac{(W_j+2Q+4Q)}{2}\times ny_c=\left[\frac{4Q+(W_{j+1}+2Q)}{2}\right]\times(1-ny_c)+\frac{(W_{j+1}+2Q)}{2}\times 1$$

Solving for W_{j+1} gives equation (27).

For inputs with n+1 equidistant symbols the weights can be assigned as:

$$W_i=-2Q \qquad i=0,1,\ldots,j-1 \tag{26}$$

$$W_j=2Q\frac{(ny_c(1-3ny_c)+1)}{1-ny_c(1-ny_c)} \tag{27}$$

$$W_{j+1}=\frac{((ny_c-1)W_j+2Q)}{ny_c} \tag{28}$$

$$W_i=-2Q \qquad i=j+2,j+3,\ldots,n \tag{29}$$

Where y_c = Constant - S_j

3.4.3 Multiple inputs to a Rule node

The rule nodes are perceptrons that generate their internal variables using summation. As mentioned previously disjunctive boolean expressions cannot be implemented as the summation operator is unable to simulate the precedence needed to implement such expressions. The combining of multiple inputs is restricted to "K out of M" expressions.

3.4.4 K out of M expressions

The weighting given to the true conditions of the expression (K of them) should drive the rule node towards positive saturation, only if all K of them are true. Multiplying these weights by an "On factor" of 1/K will achieve this. The positive

right hand side terms in (23)...(29) which are the truth conditions are multiplied by this "On factor".

Any (M-K+1) group of false conditions should drive the rule node towards negative saturation. Thus the summation of this group should be 2Q more negative than the sum of the maximums of the remaining node inputs. That is,
$2Q(M\text{-}K+1)(Off\ factor) + 2Q((K\text{-}1)/K) \leq \text{-}2Q$

Taking the case of inequality we have: $Off\ factor = \dfrac{2K-1}{K(M-K+1)}$

The right hand side of (23)...(29) which are negative will be multiplied by this "Off factor".

3.4.5 Rule: Input i > Input j

Linearly increasing the weights connecting input **i** symbol nodes to the rule node in the order of ascending symbol values and decreasing the weights associated with the symbol nodes of input **j** in the order of ascending symbol values, generates a suitable fuzzy function for this antecedent.

We have:

$$W_{0,i}=0, \quad W_{1,i}=2Q, \quad W_{2,i}=4Q \quad \dots \quad W_{n,i}=2nQ$$
$$W_{0,j}=0, \quad W_{1,j}=\text{-}2Q, \quad W_{2,j}=\text{-}4Q \quad \dots \quad W_{n,j}=\text{-}2nQ$$

Where $W_{k,i}$ is the weight connecting symbol **k** for input **i** to the rule node.
Only if input **i** > input **j** will the node internal variable x≥2Q. X will be zero if they are equal and –2Q otherwise.

For the inputs at the symbol values we have:

$$input\ i > input\ j = (input\ i \geq symbol\ 1) \wedge (input\ 2 \leq symbol\ 0)$$
$$\vee (input\ i \geq symbol\ 2) \wedge (input\ 2 \leq symbol\ 1) \text{ etc..}$$

If there are only two symbols in the input range this can be represented as a K out of M expression otherwise multiple inputs cannot be allowed.

3.4.6 Non conclusive logic

There are situations where the presence of a positive result implies that the consequent is true but the presence of a negative result does not imply that the consequent is false. This can be easily implemented by assigning a small value or zero to the "Off factor" when calculating the weights.

3.5 The quality factor

This chapter explained how rules could be generated that represents problem knowledge. For real world problems, it is sometimes difficult to generate rules that are a complete representation of the problem. So the rules generated may not be completely accurate. So a quality factor was developed. The quality factor is a measure of confidence that can be placed on a rule and can be varied between zero and one. When a high confidence level can be placed on a rule a quality factor closer to one can be used. If a high confidence level cannot be placed on the rule then a quality factor closer to zero can be used. The quality factor enabled the development of an iterative method to fine-tune the rules.

3.6 Discussion

1. This chapter discussed the rule mapping procedure. Section 3.3 showed the boolean logic and conditional rule mapping procedure. Section 3.4 showed the general derivation of the rule mapping process.
2. Equations (1) to (10) in section 3.3.1 and equations (11) to (18) in section 3.3.2 are used to calculate the initial weights of the FuNN in the subsequent chapters.
3. The FuNN is trained using input-target pairs. For image processing applications, an input and target image will be used to train the neural network. Depending on the application a domain expert may/may not be used to create the target image.
   - A non image processing expert approach:
     Sometime a clear algorithmic process using standard image processing tasks is not available to process an image to achieve the desired results. A

neural network can be used in such a situation. A non image processing expert can generate the required target image by hand or by some other method. Then rules can be generated that act like ''hints'' helping the neural network train. The trained neural network can then be used as the image processor to process other images.

- A domain expert approach:

  Depending on the application (eg: medical) a domain expert can be used to assist in generating the target image. Domain expertise can be used to generate rules representing problem knowledge. Here, the rules also assist network training.

In both cases the target image may not be 100% accurate. Thus introducing uncertainty in the training data (or image).

4. The thesis investigates the effects of using rules in network training. It is shown how the use of rules compensates for uncertainty in the training data.

Chapter 4

4 Use of a FuNN to solve fully determined problems

This chapter investigates how the use of problem knowledge can effect the operational performance of a fuzzy neural network. Two problems are used to demonstrate the advantages of using problem knowledge to initialize a network. The first is the XOR problem. The second is a similar combinatorial problem. Both problems are fully determined, thus allowing generation of rule sets that describe them completely. This chapter shows how a complete rule set can be used to calculate the weights of a FuNN to give an optimal solution. It also shows the effects of using partial rule sets. The chapter also develops a concept of uncertainty for rules. This can be used for problems with incomplete rule sets where lower certainty factors are needed to allow for the network to train efficiently.

4.1 The XOR Problem

The exclusive OR (XOR) is a classic, linearly unseparable problem. This problem was used by Minski and Papert [Minsky 1969] to illustrate the limitations of a single layer perceptron network. The problem is illustrated in Table 4.1. It is well

known that an MLP network with at least one non-linear hidden layer can solve the XOR problem.

Input 1	Input 2	Output
0	0	0
1	0	1
0	1	1
1	1	0

Table 4.1 The XOR Problem.

The particular variant of the XOR problem used was:

Given

$a,b = \{x \in \Re_{\geq 0} : (x = \cup(0 \leq x \leq 0.3, 0.7 \leq x \leq 1))\}$

$c = \{0,1\}$

Then $c = [a \oplus b]$

Where a is input 1 and b is input 2 and c is the output.

The operation of the XOR problem described above is analogous to a CMOS XOR. The input is defined for values less than 30% and greater than 70%. Operations for inputs between 30% and 70% are not defined.

4.1.1 Training and test data generation

A training set was generated consisting of 200 input/output pairs. Each input class combination had 50 training pairs. These were generated using MATLAB. Inputs [0 0], [0 1], [1 0] and [1 1] were included in the training set. Figure 4.1 shows the distribution of the training data. The trained network was tested using 600 inputs/output pairs that were generated by MATLAB. The same method used to generate the training set was used, but each input class combination had 150 training pairs. Inputs [0 0], [0 1], [1 0] and [1 1] were also included in the test set. Figure 4.2 shows the distribution of the test data.

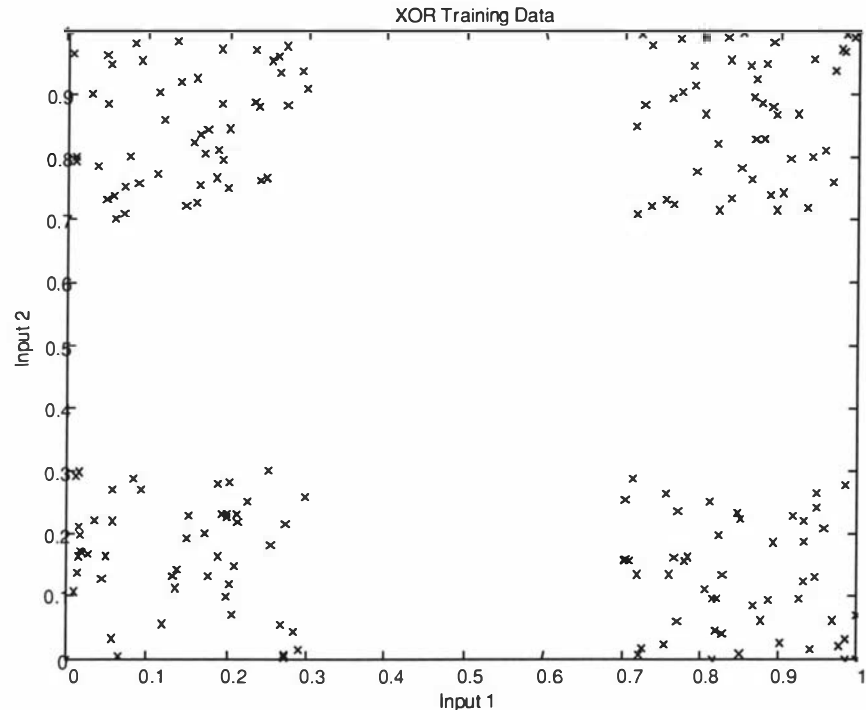


Figure 4.1 XOR training data – 200 pairs.

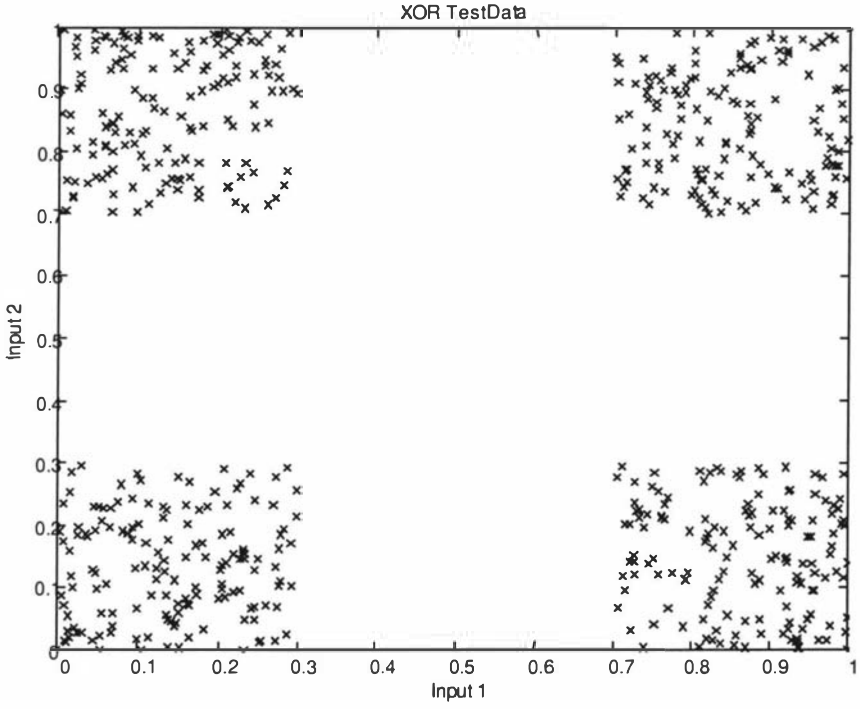


Figure 4.2 XOR test data – 600 pairs.

4.1.2 Rules representing the XOR problem

The following rules represent the operation of the XOR problem (*input 1* and *input 2* are the inputs to the neural network and *output* is the network output). Two membership functions LO and HI were used to represent the low and high state of the inputs and output.

Rule no.	Rule
1	If input 1 is LO and input 2 is LO then output is LO.
2	If input 1 is HI and input 2 is LO then output is HI.
3	If input 1 is LO and input 2 is HI then output is HI.
4	If input 1 is HI and input 2 is HI then output is LO.

Table 4.2 Rule representing the XOR problem.

Since this rule set is a complete representation of the problem an experiment was conducted to determine if a network could simulate the behavior of the XOR problem without training.

4.1.3 The error function

An error function was used to calculate the operational performance of the network as explained below:

If the required output was HI and the network generated a value greater than 0.7 (HI) the error is zero. If not the error for that testing pair is 0.5. This error is added to the final error.

If the required output was LO and the network generated a value less than 0.3 then the error is zero. If not the error for that testing pair is 0.5. This error is added to the final error.

4.1.4 The MSE error

The MSE error is the mean squared error between the network output and the target. In the experiments in this chapter the MSE error is displayed to show the starting point of the network prior to training. The MSE error is representative of where the network is in the solution space. A smaller MSE error denotes that the network is close to a solution and vice versa.

4.1.5 Fuzzy neural network structure used for XOR problem

Figure 4.3 shows the fuzzy neural network structure used for the XOR problem.

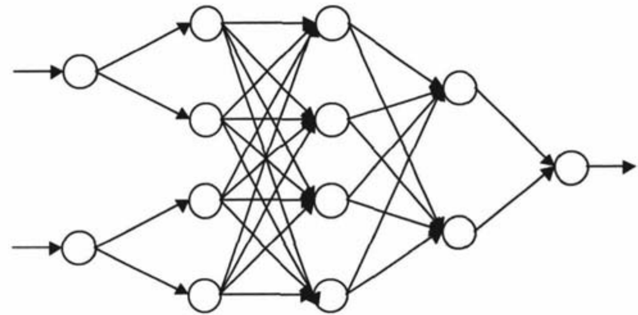


Figure 4.3 FuNN 2-4-4-2-1 architecture for XOR problem.

A 2-4-4-2-1 (2 input nodes, 4 conditional node, 4 rule nodes, 2 action nodes and 1 output node) was used for the XOR problem.

4.1.6 Experiment to determine if a FuNN could emulate the XOR problem without training using boolean logic rules.

The XOR problem has two states, ON and OFF. These states can be represented by two membership functions LO and HI. If I1 and I2 are the two inputs to the network then the boolean logic rules for this problem can be represented as,

Rule no.	Description
1	If I1=LO and I2=LO Then Output = LO
2	If I1=HI and I2=LO Then Output = HI
3	If I1=LO and I2=HI Then Output = HI
4	If I1=HI and I2=HI Then Output = LO

Table 4.3 Rule for experiment 4.1.6.

These rules can be used to generate the antecedent and consequent weight matrices. Table 4.4 shows the weight matrices generated from the rules. A saturation value of three was used.

Rule No	Input 1		Input 2		Output	
	LO	HI	LO	HI	LO	HI
1	1.5	-4.5	1.5	-4.5	3	-3
2	-4.5	1.5	1.5	-4.5	-3	3
3	1.5	-4.5	-4.5	1.5	-3	3
4	-4.5	1.5	-4.5	1.5	3	-3

Table 4.4 Weight matrices generated from rules.

The weight matrices are used to initialize the antecedent and consequent weights of the fuzzy neural network prior to training. Figure 4.3 shows the 2-4-4-2-1 structure used for this problem. The network was tested using the training and test data (total of 800 training pairs). Table 4.5 shows the results of the experiment.

Network	Error as a percentage(%)	starting MSE (training)
4 rules	0	0.0212236
no rules	99.25	0.10404

Table 4.5 Results for experiment 4.1.6.

As the table shows the network performed with no error when rules were used to initialize the FuNN. The starting MSE (training) of the network initialized with

rules is lower than the one without rules. This shows that the rules have placed the network near (or at) an optimal solution without training.

4.1.7 Experiment to determine if a FuNN could emulate the XOR problem without training using conditional rules.

The next experiment tests if the XOR problem can be solved using conditional rules. Table 4.6 shows the four membership functions used.

Membership function	Value
M1	0
M2	0.3
M3	0.7
M4	1

Table 4.6 Membership function for conditional rules.

The following conditional rules describe the operation of the XOR problem.

Rule no.	Description
1	If I1<M2 and I2<M2 Then Output = LO
2	If I1<M2 and I2>M3 Then Output = HI
3	If I1>M3 and I2<M2 Then Output = HI
4	If I1>M3 and I2>M3 Then Output = LO

Table 4.7 Rule for experiment 4.1.7.

The network has two inputs, four rules, two actions and one output. Table 4.6 shows the four membership functions used for this experiment. Each input node will have four membership functions resulting in eight nodes for the condition layer. Therefore a 2-8-4-2-1 FuNN architecture can be used for this problem. Table 4.8 shows the weight mapping for the above four rules. A saturation value of three was used for the rules.

Rule No.	Input 1				Input 2				Output	
									LO	HI
1	1.5	1.5	-4.5	-4.5	1.5	1.5	-4.5	-4.5	3	-3
2	1.5	1.5	-4.5	-4.5	-4.5	-4.5	1.5	1.5	-3	3
3	-4.5	-4.5	1.5	1.5	1.5	1.5	-4.5	-4.5	-3	3
4	-4.5	-4.5	1.5	1.5	-4.5	-4.5	1.5	1.5	3	-3

Table 4.8 Conditional rule mapping for XOR problem.

The weight matrices in Table 4.8 were used to initialize the FuNN. The network was then tested using the training and testing set (800 pairs). Table 4.9 shows the results.

Network	Error as a percentage(%)	starting MSE (training)
4 rules	0	0.0063779
No rules	95.25	0.13315

Table 4.9 Results for experiment 4.1.7 using conditional rules.

The XOR problem can be represented using boolean logic or conditional rules. The problem is fully deterministic. Therefore rules were generated that represent the complete operation of the problem. With a high saturation value (in this case three) the network performs with zero error without training. Table 4.9 also shows that the network with rules has started from a lower MSE (training) than the network without rules.

4.1.8 Experiment - Operational performance test 1 for a range of parameters on the XOR problem using boolean logic rules

This experiment was performed to determine the network operational performance for different saturation values and different training times. The boolean logic rules used in experiment 4.1.6 were used for this experiment. The saturation value was varied between 0.5 and 6 (0.5, 1, 2, 3, 4, 5, 6). From this point onwards Q will be used to represent the quality factor. The quality factor was one. The training data (Figure 4.1) was used to train the network. The network was trained for 25 epochs. After every epoch the operational performance was measured using the test data

(Figure 4.2). A network initialized with random numbers in the range {-1,+1} was used as a benchmark. Random numbers in the range {-0.1,+0.1} were added to the weights of the FuNN initialized with rules to remove network symmetry at the start of training [Rumelhart 1986]. Figure 4.4 shows the operational error for different training times and different saturation values.

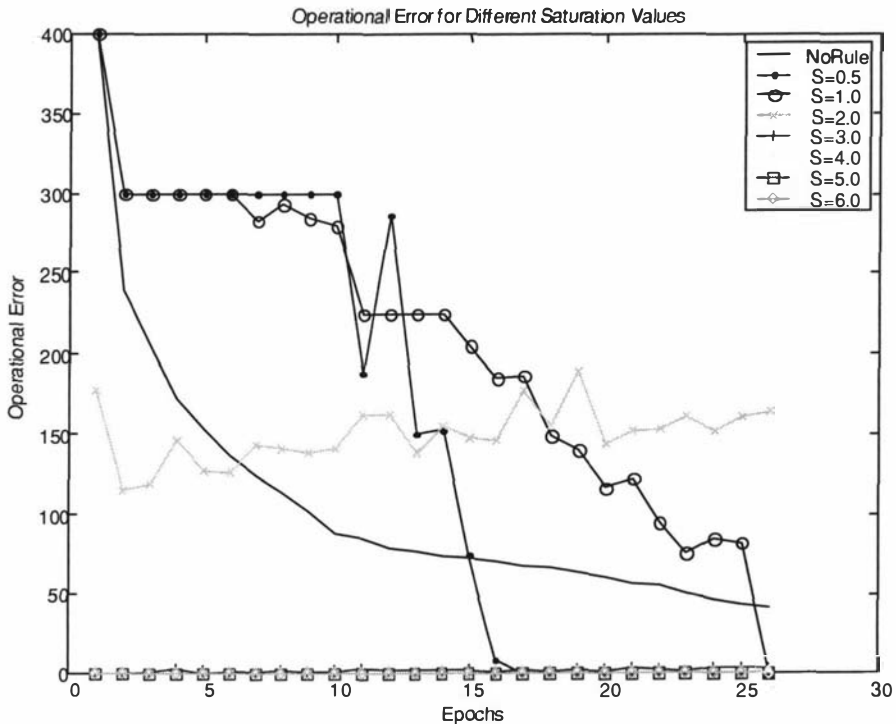


Figure 4.4 Operational error graph for XOR problem.

When the saturation value was greater than or equal to four the network performed with zero error before and after training. When the saturation value was equal to three there is a very small error in the operational performance of the network. This is because the network is having difficulty changing the weights due to saturation. The effectiveness of the rules reduces as the saturation value drops (0.5, 1), in effect the rules are being declared as less certain and the training becomes more effective.

4.1.9 Experiment – Operational performance test 2 for a the same parameters but a smaller range on the XOR problem using boolean logic rules

This is a continuation from experiment 4.1.8. It was determined that the network performed with zero error after using a saturation value of three. The saturation value was varied between three and four (3.0, 3.2, 3.4, 3.5, 3.6, 3.8, 4.0) to determine the point at which the operational error was zero. Table 4.10 shows the operational error of the network for a range of saturation values and epochs.

Epoch	Saturations						
	3	3.2	3.4	3.5	3.6	3.8	4
0	177.5	0	0	0	0	0	0
1	115	0	0	0	0	0	0
2	119.5	0.5	0.5	0	0	0	0
3	146	2.5	0	0	0	0	0
4	127	0	0	0	0	0	0
5	126.5	0.5	0.5	0	0	0	0
6	142.5	1	0	0.5	0	0	0
7	140	2	0	0	0	0	0
8	138	0.5	0	0	0	0	0
9	141	0.5	0	0	0.5	0	0
10	161.5	2.5	1	0	0	0	0
11	162	1.5	0.5	0	0	0	0
12	137.5	1.5	1	0	0	0	0
13	155	1.5	0	0	0	0	0
14	147.5	1.5	0	0	0	0	0
15	145.5	0.5	0	0	0	0	0
16	177.5	1.5	0	0	0	0	0
17	155.5	1	0	0	0	0	0
18	189.5	2	0	0	0	0	0
19	144	0.5	0	0	0	0	0
20	151	3	0.5	0	0	0	0
21	153	2	0.5	0	0	0	0
22	160.5	0.5	1	0	0	0	0
23	151	2.5	0.5	0	0	0	0
24	161	3	1.5	0	0	0	0
25	162.5	2.5	0	0	0	0	0

Table 4.10 Operational error for different saturation values.

As the table shows, the network operational error is zero before and after training when the saturation was higher than 3.4.

4.1.10 Experiment – Operational performance test 3 for a range of parameters on the XOR problem using conditional rules

This experiment was performed to determine the operational performance of the network for different saturation values and epochs. The same network structure and conditional rules that were used in experiment 4.1.7 were used. The saturation value was varied between 0.5 and 6 (0.5, 1, 2, 3, 4, 5, 6). The network was trained for 25 epochs and the operational error was measured for each epoch. Figure 4.5 shows the operational error of the network for different saturation values for 25 epochs.

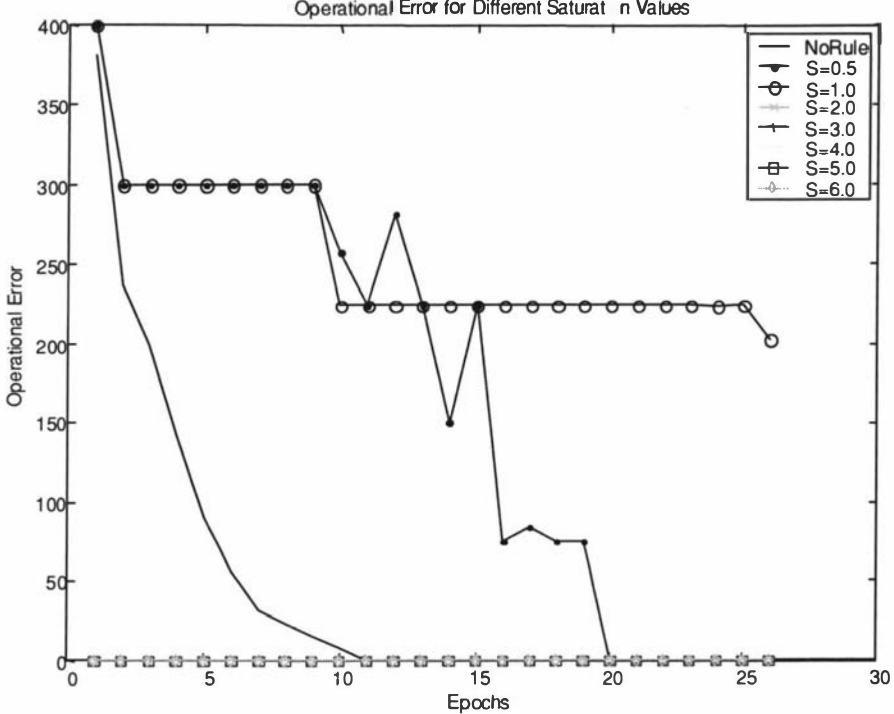


Figure 4.5 Operational errors for different saturation values.

The network performed with zero error when the saturation value was greater than two.

4.1.11 Experiment – Operational performance test 4 for the same parameters on a smaller range on the XOR problem using conditional rules

This is a continuation from the previous experiment to determine at which point the operational error was zero. The previous experiment showed that when the saturation value was between two and three the operational performance was zero before and after training. The saturation value was varied between two and three (2, 2.2, 2.4, 2.5, 2.6, 2.8, 3). The network was trained for 25 epochs and the operational performance was measured. Table 4.11 shows the operational errors for different saturation values and epochs. The operational error for five epochs is shown in the table as the error remains zero for further training when the saturation value is 2.2. The operational error for saturation values greater than 2.2 is not shown as they are zero.

Epochs	Error when saturation =2.0	Error when saturation= 2.2
0	400	0
1	300	0
2	300	0
3	300	0
4	300	0
5	300	0

Table 4.11 Operational errors for different saturation values.

4.2 Partial Rule Sets

The previous section showed the use of rules to solve a fully determined problem. The rules used in that section represented the complete operation of the XOR problem. A fully determined problem is also investigated in this section. The operation of the problem can be completely represented by rules, but only a partial rule set is used. This is to investigate the effectiveness of using partial problem knowledge when available.

The problem involves three inputs B, C and D and one output A.

Three problems were generated from these inputs. They are,

- A = B & C & (not D)
- A = B & (not C) & D
- A = (not B) & C & D

Inputs B, C and D can have values 0 or 1. For each experiment only two inputs were used to define the rules to generate a partial rule set.

4.2.1 Training and test data

The training data was generated using the MATLAB random number generator function *rand*. This function generates random numbers from a uniform distribution in the interval (0.0,1.0). Input B, C and D were each assigned 800 numbers from the random number generator. Out of this 400 were in the interval (0.0, 0.1) and the other 400 were in the interval (0.9, 1.0). A 3x800 matrix containing the input data was generated. The first 200 columns were used as training data. The next 600 columns were used as test data for the trained network. The target data for each case of A was generated using the input data.

4.2.2 Fuzzy neural network structure used for this problem

Layer	No. of nodes used
Input layer	3 inputs
Condition layer	2 conditions per node = 6 condition nodes
Rule layer	8 rule nodes
Action layer	2 nodes
Output layer	1 node

Table 4.12 FuNN structure used for partial rule sets.

The 3-6-8-2-1 fuzzy neural network structure was used for the remaining experiments in this chapter.

4.2.3 Experiment 1 - Problem A = B & C & (not D) – complete rule set

This experiment was performed to confirm the hypothesis that the problem could be solved using a FuNN without training. Table 4.13 shows the operation of A = B & C & (not D).

B	C	D	A=B&C&(~D)
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Table 4.13 Operation of A=B&C&(~D).

Two membership functions LO and HI were used to represent the states 0 and 1. I1, I2 and I3 were used to represent inputs B, C and D. Output was used to represent output A. Eight rules were generated to represent the problem.

Rule no.	Description
1	If I1=LO and I2=LO and I3=LO Then Output = LO
2	If I1=LO and I2=LO and I3=HI Then Output = LO
3	If I1=LO and I2=HI and I3=LO Then Output = LO
4	If I1=LO and I2=HI and I3=HI Then Output = LO
5	If I1=HI and I2=LO and I3=LO Then Output = LO
6	If I1=HI and I2=LO and I3=HI Then Output = LO
7	If I1=HI and I2=HI and I3=LO Then Output = HI
8	If I1=HI and I2=HI and I3=HI Then Output = LO

Table 4.14 Rules for experiment 4.2.3.

The saturation value for the rules was varied between 0.5 and 6 (0.5, 1, 2, 3, 4, 5, 6). A FuNN was initialized using the rules and tested (without training) using the training and test data (total of 800 columns). Table 4.15 shows the operational error for the network.

When no rules were used	Saturation value for used for rules						
	0.5	1	2	3	4	5	6
300	200	50	0	0	0	0	0

Table 4.15 Operational error for A=B&C&(~D) using complete rule set.

When the saturation value was between two and six the operational error of the network was zero. The network performed as expected, the original hypothesis was confirmed.

4.2.4 Experiment 1a - Problem A = B & C & (not D) – incomplete rule set

Table 4.13 shows the operation of A = B & C & (not D). This experiment was performed to determine the operational performance of a FuNN initialized with a partial rule set. Only inputs B and C were used in the rules. I1 and I2 represent input B and C. Output represents the output A.

Rule no.	Description
1	If I1=LO and I2=LO Then Output = LO
2	If I1=LO and I2=LO Then Output = LO
3	If I1=LO and I2=HI Then Output = LO
4	If I1=LO and I2=HI Then Output = LO
5	If I1=HI and I2=LO Then Output = LO
6	If I1=HI and I2=LO Then Output = LO
7	If I1=HI and I2=HI Then Output = HI
8	If I1=HI and I2=HI Then Output = LO

Table 4.16 Rules for experiment 4.2.4.

The rules were used to initialize a FuNN. Random numbers in the interval –0.1 to +0.1 were added to the weights of the network to remove the problem of symmetry. The saturation value was varied between 0.5 and six (0.5, 1, 2, 3, 4, 5, 6). The network was trained for 25 epochs using the training data. The operational performance was measured after every epoch. Table 4.17 shows the operational error value for each epoch of the network for a range of saturation values.

Epoch	No rules	Saturation value for rules						
		0.5	1	2	3	4	5	6
0	300	0	0	0	100	100	100	100
1	188	0	0	0	75	75	75	75
2	124.5	0	0	0	0	75	75	75
3	75	0	0	0	0	73	75	75
4	75	0	0	0	0	0	75	75
5	73.5	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0

Table 4.17 Operational error for A=B&C&(~D).

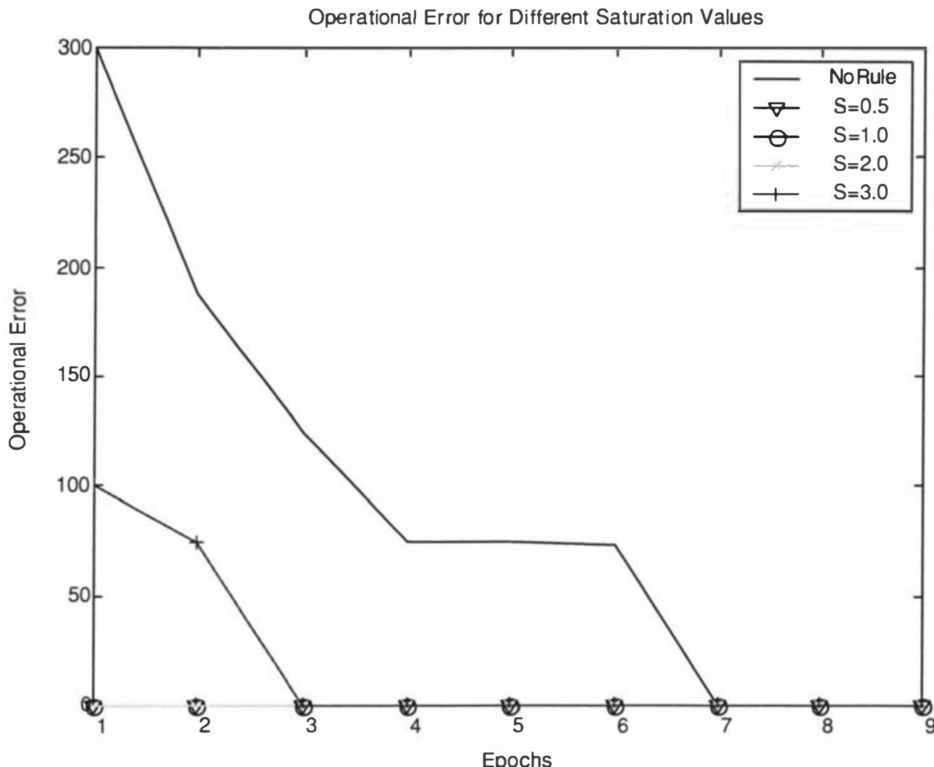


Figure 4.6 Operational error graph for A=B&C&(~D).

Figure 4.6 shows the operational error graph for different saturation values for nine epochs. The error remains zero thereafter. The operational performance error is zero before and after training when the saturation value is between 0.5 and 2. As the saturation value increases beyond three the network has to train for a longer time for zero operational error (only S=3 is shown in the graph for clarity).

The above combinatorial problem is fully determined. The experiment demonstrates that when a partial rule set is used the network will converge to an optimal solution faster than when no rules are used.

4.2.5 Experiment 2 – Problem A = B & (not C) & D – complete rule set

This experiment was performed to show that the problem could be solved without training a FuNN. Table 4.18 shows the operation of A = B & (not C) & D.

B	C	D	A=B&(~C)&D
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

Table 4.18 Operation of A=B&(~C)&D.

Two membership functions LO and HI were used to represent 0 and 1. I1, I2 and I3 were used to represent inputs B, C and D. Output was used to represent output A. Eight rules were generated to represent this problem.

Rule no.	Description
1	If I1=LO and I2=LO and I3=LO Then Output = LO
2	If I1=LO and I2=LO and I3=HI Then Output = LO
3	If I1=LO and I2=HI and I3=LO Then Output = LO
4	If I1=LO and I2=HI and I3=HI Then Output = LO
5	If I1=HI and I2=LO and I3=LO Then Output = LO
6	If I1=HI and I2=LO and I3=HI Then Output = HI
7	If I1=HI and I2=HI and I3=LO Then Output = LO
8	If I1=HI and I2=HI and I3=HI Then Output = LO

Table 4.19 Rules for experiment 4.2.5.

The saturation value for the rules was varied between 0.5 and 6 (0.5, 1, 2, 3, 4, 5, 6). A FuNN was initialized using the rules and tested (without training) using the training and test data (total of 800 columns). Table 4.20 shows the operational error of the network.

When no rules were used	Saturation value used for rules						
	0.5	1	2	3	4	5	6
300	200	50	0	0	0	0	0

Table 4.20 Operational error for A=B&(~C)&D using complete rule set.

When the saturation value was between two and six the operational error of the network was zero without training.

4.2.6 Experiment 2a – Problem A = B & (not C) & D – incomplete rule set

Table 4.18 shows the operation of A = B & (not C) & D. This experiment was performed to determine the operational performance of a FuNN with only a partial rule set. Eight rules were generated to represent the problem. Only inputs B and C were used in the rules.

Rule no.	Description
1	If I1=LO and I2=LO Then Output = LO
2	If I1=LO and I2=LO Then Output = LO
3	If I1=LO and I2=HI Then Output = LO
4	If I1=LO and I2=HI Then Output = LO
5	If I1=HI and I2=LO Then Output = LO
6	If I1=HI and I2=LO Then Output = HI
7	If I1=HI and I2=HI Then Output = LO
8	If I1=HI and I2=HI Then Output = LO

Table 4.21 Rules for experiment 4.2.6.

The rules were used to initialize a FuNN. Random numbers in the interval –0.1 to +0.1 were added to the weights of the network to remove the problem of symmetry. The saturation value was varied between 0.5 and 6 (0.5, 1, 2, 3, 4, 5, 6). The network was trained for 25 epochs using the training data. The operational performance was measured during each epoch. Table 4.22 shows the operational error value for each epoch for different saturation values.

Epoch	No Rules	Saturation						
		0.5	1	2	3	4	5	6
0	300	0	0	0	100	100	100	100
1	188	0	0	0	75	75	75	75
2	124.5	0	0	0	75	75	75	75
3	75	0	0	0	75	75	75	75
4	75	0	0	0	75	75	75	75
5	73.5	0	0	0	75	75	75	75
6	0	0	0	0	68	75	75	75
7	0	0	0	0	75	75	75	75
8	0	0	0	0	75	75	75	75
9	0	0	0	0	75	75	75	75
10	0	0	0	0	75	75	75	75
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
25	0	0	0	0	75	75	75	75

Table 4.22 Operational errors for A = B&(~C)&D.

Figure 4.7 shows the operational error of the network for a range of saturation values.

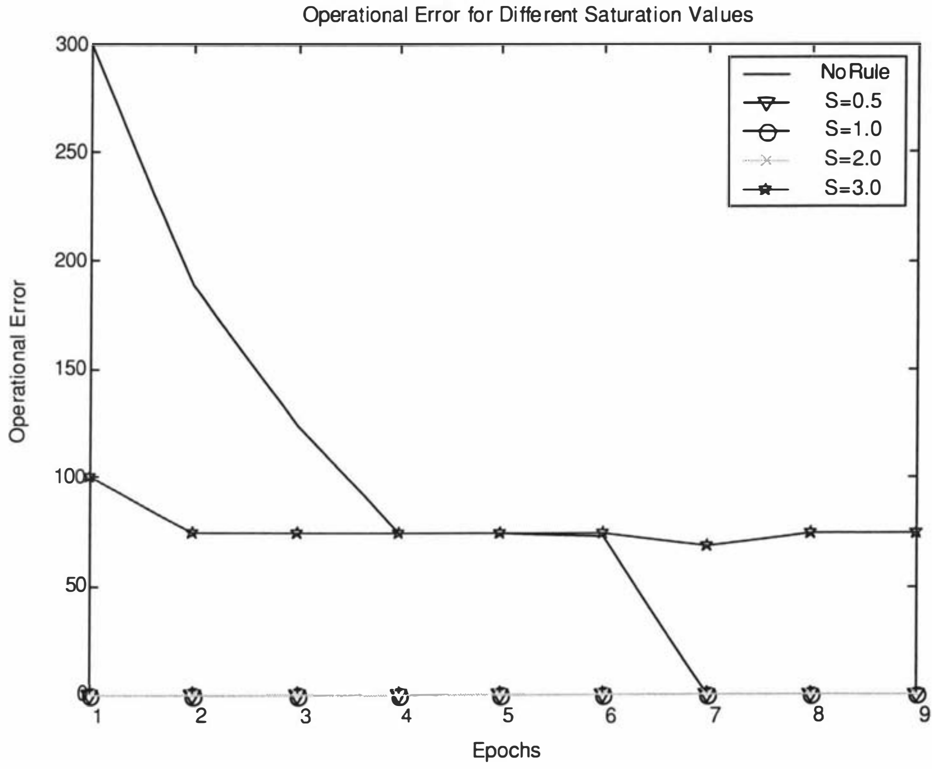


Figure 4.7 Operational error graph for A=B&(~C)&D.

Table 4.22 shows that the network has performed with zero error when the saturation value was 0.5, 1 and 2. When the saturation value was between three

and six the network converged to a non-optimal solution. This is due to the high saturation values causing the weights to change very slowly.

4.2.7 Experiment 3 – Problem A = (not B) & C & D – complete rule set

Table 4.23 shows the operation of A = (not B) & C & D.

B	C	D	A=(~B)&C&D
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Table 4.23 Operation of A=(~B)&C&D.

Two membership functions LO and HI were used to represent the states 0 and 1. I1, I2 and I3 were used to represent inputs B, C and D. Output was used to represent output A. Eight rules were generated to represent the problem.

Rule no.	Description
1	If I1=LO and I2=LO and I3=LO Then Output = LO
2	If I1=LO and I2=LO and I3=HI Then Output = LO
3	If I1=LO and I2=HI and I3=LO Then Output = LO
4	If I1=LO and I2=HI and I3=HI Then Output = HI
5	If I1=HI and I2=LO and I3=LO Then Output = LO
6	If I1=HI and I2=LO and I3=HI Then Output = LO
7	If I1=HI and I2=HI and I3=LO Then Output = LO
8	If I1=HI and I2=HI and I3=HI Then Output = LO

Table 4.24 Rules for experiment 4.2.7.

The saturation value for the rules was varied between 0.5 and 6 (0.5, 1, 2, 3, 4, 5, 6). A FuNN was initialized using the rules and tested (without training) using the

training and test data (total of 800 columns). Table 4.25 shows the operational error of the network.

When no rules were used	Saturation value used for the rules						
	0.5	1	2	3	4	5	6
300	200	50	0	0	0	0	0

Table 4.25 Operational error for A=(~B)&C&D using complete rule set.

When the saturation value was between two and six the operational error of the network was zero without training.

4.2.8 Experiment 3a – Problem A = (not B) & C & D – incomplete rule set

Table 4.23 shows the operation of A = (not B) & C & D. This experiment was also performed to determine the operational performance of a FuNN initialized with a partial rule set. Only inputs B and C are used to generate the rules.

Rule no.	Description
1	If I1=LO and I2=LO Then Output = LO
2	If I1=LO and I2=LO Then Output = LO
3	If I1=LO and I2=HI Then Output = LO
4	If I1=LO and I2=HI Then Output = HI
5	If I1=HI and I2=LO Then Output = LO
6	If I1=HI and I2=LO Then Output = LO
7	If I1=HI and I2=HI Then Output = LO
8	If I1=HI and I2=HI Then Output = LO

Table 4.26 Rules for experiment 4.2.8.

The rules were used to initialize a FuNN. Random numbers in the interval –0.1 to +0.1 were added to the weights of the network to remove the problem of symmetry. The saturation value was varied between 0.5 and 6 (0.5, 1, 2, 3, 4, 5, 6). The network was trained for 25 epochs using the training data. The operational performance was measured after every epoch. Table 4.27 shows the operational

error for each epoch for different saturation values. Figure 4.8 shows the operational error of the network for a range of saturation values.

Epoch	No Rule	Saturation						
		0.5	1	2	3	4	5	6
0	300	0	0	0	100	100	100	100
1	188	0	0	0	75	75	75	75
2	124.5	0	0	0	75	75	75	75
3	75	0	0	75	72	75	75	75
4	75	0	0	0	75	75	75	75
5	73.5	0	0	5.	75	75	75	75
6	0	0	0	0	75	75	75	75
7	0	0	0	0	75	75	75	75
8	0	0	0	0	75	75	75	75
9	0	0	0	0	75	75	75	75
10	0	0	0	0	75	75	75	75
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
25	0	0	0	1	75	75	75	75

Table 4.27 Operation errors for A=(~B)&C&D.

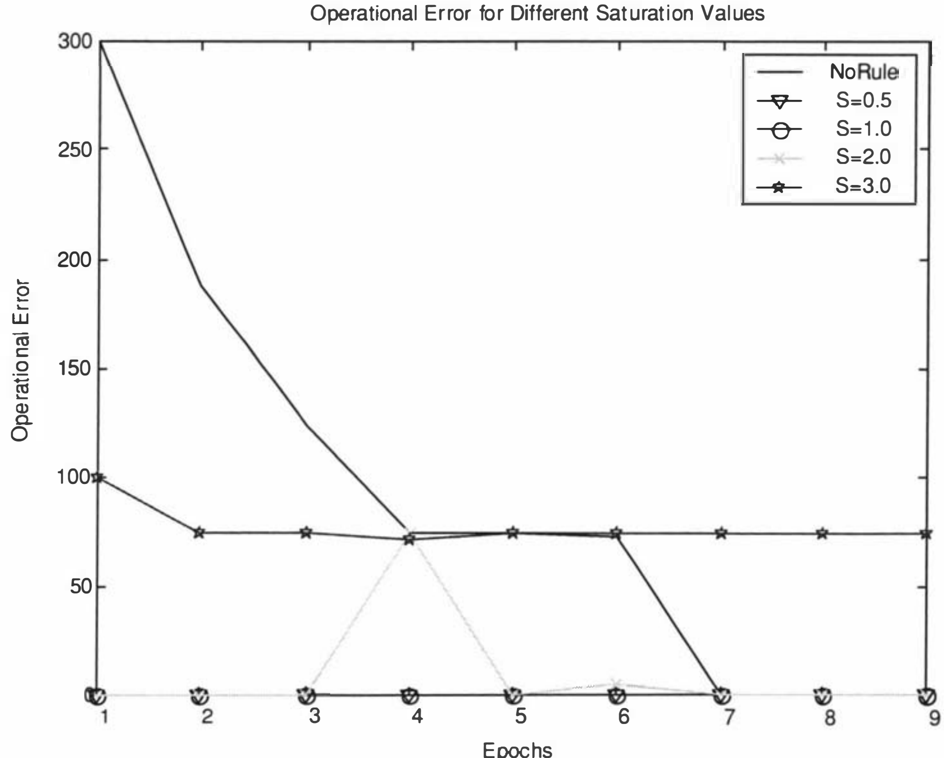


Figure 4.8 Operational error graph for A=(~B)&C&D.

When the saturation value was between 0.5 and 2 the network had zero operational error before training.

4.2.9 Experiment 4 – Problem A = B&C&(~D) using only a partial rule set

All the rules for A=B&C&(~D) are known. The previous experiments dealt with incomplete rule sets. This experiment will determine how a network will perform when only a partial rule set is used. The rules used have a high degree of certainty and are correct but only a few rules are used in the rule set. Inputs B, C and D are used to generate the rules. The rules used are;

Rule no.	Description
1	If I1=LO and I2=LO and I3=LO Then Output = LO
2	If I1=LO and I2=LO and I3=HI Then Output = LO
3	If I1=LO and I2=HI and I3=LO Then Output = LO
4	If I1=M1 and I2=HI and I3=HI Then Output = LO
5	If I1=HI and I2=LO and I3=LO Then Output = LO

Table 4.28 Rule for experiment 4.2.9.

The rules were used to initialize a FuNN. Random numbers in the interval –0.1 to +0.1 were added to the weights of the network to remove the problem of symmetry. The saturation value was varied between 0.5 and 6 (0.5, 1, 2, 3, 4, 5, 6). The network was trained for 25 epochs using the training data. The operational performance was measured after every epoch.

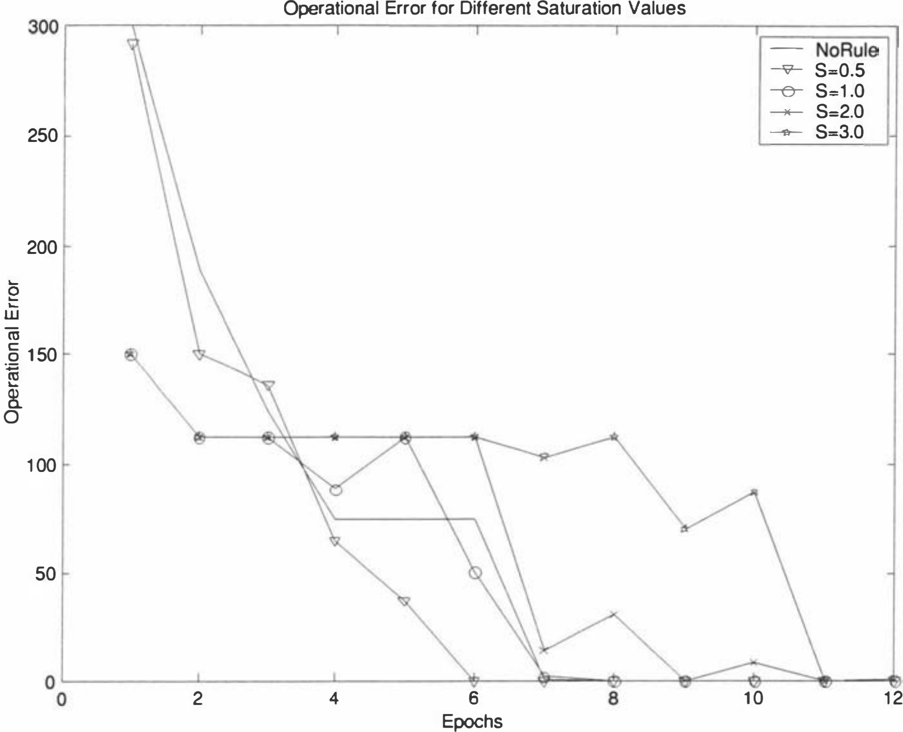


Figure 4.9 Operational performance of A=B&C&(~D) for a network using a partial rule set with high certainty.

For clarity Figure 4.9 only shows operational errors for saturation value less than or equal to three. When the saturation value increases beyond three the network required a larger number of epochs to train for zero operational error. As the graph shows as the saturation value increases from 0.5 to 3 the network took more epochs to achieve zero operational error. Although the rules have a high degree of certainty, the rule set is incomplete. Then the saturation for the existing rules should remain between 0.5 and 1.

4.3 Discussion and Conclusions

The chapter shows how problem knowledge can be incorporated into a fuzzy neural network to solve fully determined problems.

- The operation of the XOR problem can be completely described using rules. The experiments show that when rules were used the FuNN can emulate the XOR operation without any training (exp: 4.1.6). This shows that a complete

rule set with a sufficiently high saturation value develops a correct weight set for the problem.

- When boolean logic rules were used a saturation value of 3.8 was required for the network to perform with zero error without training (exp: 4.1.9).
- When conditional rules were used a saturation value of 2.2 was required for the same operation (exp: 4.1.11). This shows that conditional rules have a greater influence than boolean logic rules.
- Conditional rules can be said to be more ''powerful'' than boolean logic rules.

The second fully determined problem (involving A,B,C & D) was also solved by a FuNN without training when rules were used.

- This problem showed the need for saturation values between 0.5 and one when a partial rule set was used to enable the network to learn.
- Rule sets with partial problem knowledge still assist the network to train.
- Higher saturation values reduced network adaptability when partial rule sets were used (exp: 4.2.6, 4.2.8).
- Higher saturation values constrict the adaptability of the rules. If only partial problem knowledge is know the saturation value should be set low enough to allow the rules to adapt.
- The saturation value can be likened to quantified certainty values for the rules. High saturation values effectively stops the training from changing the weights and represents high certainty in the correctness and completeness of each rule. Low saturation values mean the rules are incomplete and/or uncertain and allows the training to alter the weight values relatively easily.

Chapter 5

5 Fully Understood Non Determined Problem

The previous chapter investigated the effects of using problem knowledge on two fully determined problems. An artificial problem is investigated in this chapter. The problem is fully known and non-determined. The problem is called the ''Toy Problem'' as it involves using a fuzzy neural network to count and classify artificial ''toys'' or artificial objects in an image.

5.1 Overview of the toy problem

The toy problem is an artificial problem designed to test the use of rules in a FuNN. The problem is fully known and non-determined. The toy problem consists of an image containing toys (artificial objects). A fuzzy neural network was used to detect the presence of toys in the image and then to classify them.

5.1.1 A detailed description of the problem

A 256x256 (pixel) grayscale image was generated. The image consists of,

- noise.
- 50 toys.

The MATLAB *rand* function generates random numbers from a uniform distribution on the interval {0.0, 1.0}. The *rand* function was used to generate a 256x256 matrix of random numbers. The random numbers in the matrix were then normalized in the range {3, 16} to generate the noisy image. The toys were then placed on this noisy image. Each toy can be classified by toy type and toy orientation.

5.1.2 Description of a toy

A toy fits into a square area of 7x7 pixels. Figure 5.1 shows one type of toy and its possible orientations.

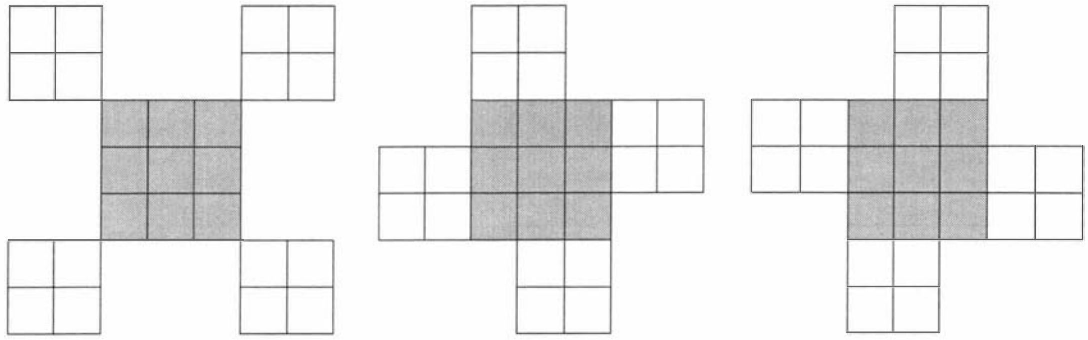


Figure 5.1 Toys that can be classified by type and orientation.

As the figure shows a toy consists of a central 3x3 square surrounded by four 2x2 squares. The toy "type" is classified by the pixel intensity of the central 3x3 square. Four toy types where used in this problem. Toy orientation is determined by the placement of the small 2x2 square blocks around the central 3x3 square. Figure 5.1 shows the three possible orientations of a toy. Figure 5.2 shows an image containing 25 individual toys. The image shows some of the possible types and orientations a toy can have.

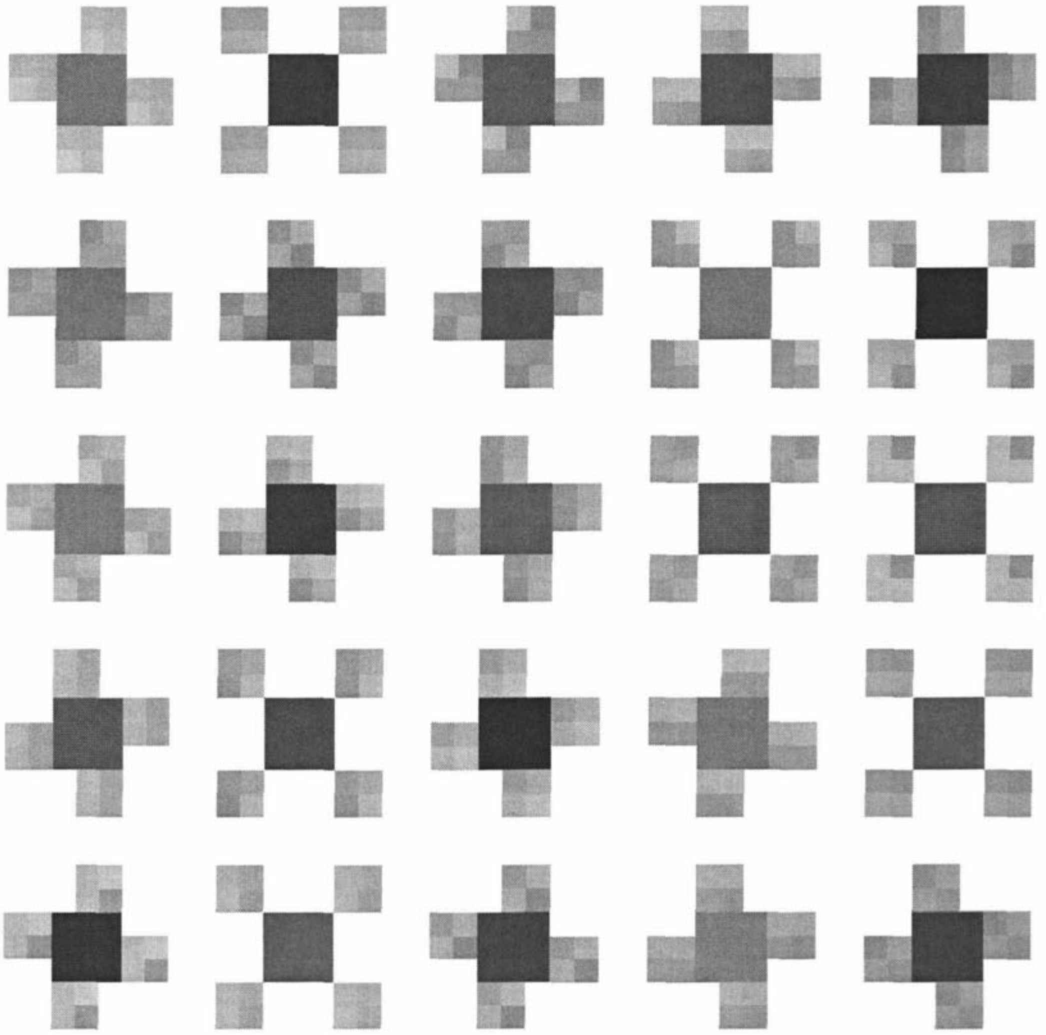
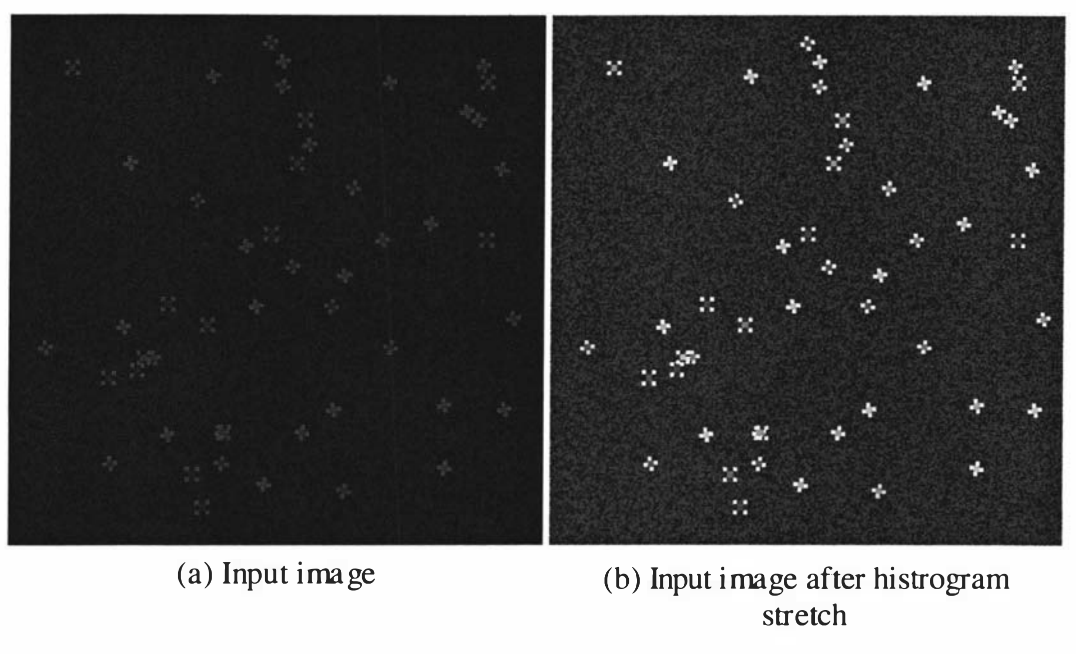


Figure 5.2 Image containing 25 individual toys of different type and orientation.

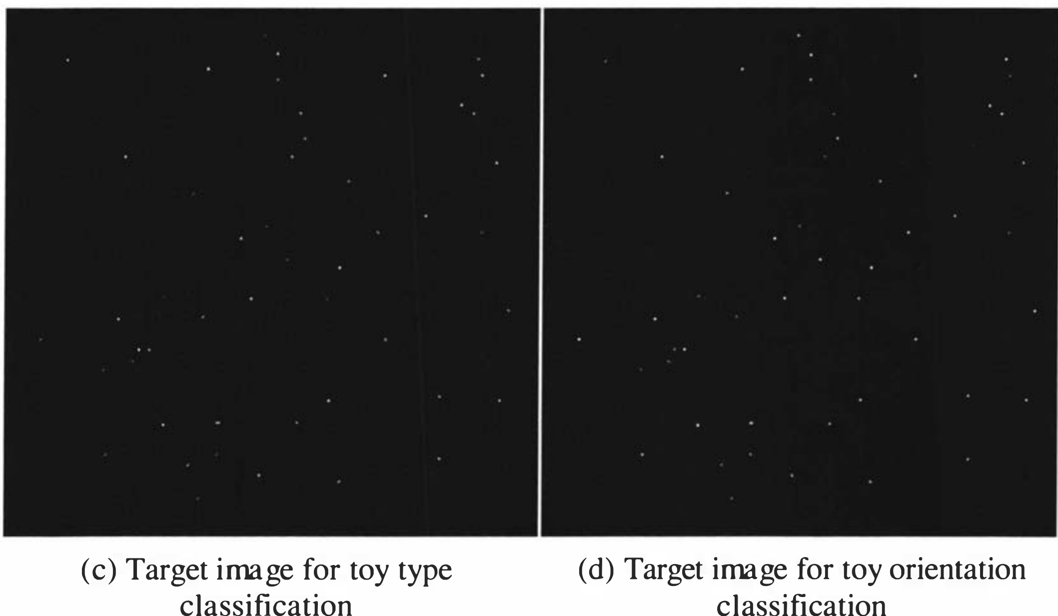
5.1.3 The training images

Before the neural network could be trained input and target images needed to be generated. Figure 5.3 shows the images used to train the network. Figure 5.3 (a) is the input image. A histogram stretch function was performed on Figure 5.3 (a) to generate Figure 5.3 (b). This figure is included to better illustrate the nature of the input image, it is not used in the training process.



(a) Input image

(b) Input image after histrogram stretch



(c) Target image for toy type classification

(d) Target image for toy orientation classification

Figure 5.3 Training images.

Figure 5.3 (c) is the target image for toy type classification. Figure 5.3 (d) is the target image for toy orientation classification.

5.1.3.1 Toy type target image generation

Four toy types were used for this problem. Each type was classified as 1, 2, 3 and 4. Gray scale intensity values were assigned to each type, Table 5.1 shows the values assigned.

Toy type	Gray scale intensity value
1	63
2	127
3	191
4	255

Table 5.1 Gray scale intensity values for each toy type.

A blank target image of intensity 0 with dimensions 256x256 was generated. A pixel was placed in the target image that corresponded to the location of a toy in the input image. The toy type was indicated by setting the pixel to the appropriate intensity as per Table 5.1.

5.1.3.2 Toy orientation target image generation

Three toy orientations were used for this problem. Each orientation was classified as 1, 2 and 3. Gray scale intensity values were assigned to each orientation, Table 5.2 shows the values assigned.

Toy orientation	Gray scale intensity value
1	85
2	170
3	255

Table 5.2 Gray scale intensity values for each toy orientation.

Again a blank target image of intensity 0 with dimensions 256x256 was generated. A pixel was placed in the target image that corresponded to the location of a toy in the input image. The toy orientation was indicated by setting the pixel to the appropriate intensity as per Table 5.2.

5.2 Network training method

5.2.1 The training data

A toy fits into a 7x7-pixel area. Therefore a 7x7-pixel window filter [Pugmire 1995] was used for training.

A single input-target training pair is defined as,
Input - a 7x7-pixel area from the input image converted to a 1x49 input vector.
Target – a 1x1-pixel on the target image that corresponds to the location of the center of the 7x7 window on the input image was converted to a 1x1 target vector.

A single scan area is defined as a rectangular area that is larger than 7x7-pixels. The scan area is extracted from the input and target image. The window filter passes over this scan area to generate several training pairs.

Several scan areas were combined to generate the training data. The input matrix will have a size of Cx49 and the target matrix will have a size of Cx1 where C is the number of training vectors.

5.2.2 Rule development

Figure 5.4 shows the structure of the window filter used for the toy experiments. Each element of the window filter is numbered as I1, I2, etc. Rules were developed using this window filter. If M1, M2 and M3 are membership functions, an example of a rule is,

R 1 If I1=M1 and I7=M1 and I43=M2 and I49=M2 and I25=M3 Then O1=M1

This rule is used to generate a matrix that will be used to initialize the weights of a fuzzy neural network. The rules were developed using problem knowledge.

I1	I2	I3	I4	I5	I6	I7
I8	I9	I10	I11	I12	I13	I14
I15	I16	I17	I18	I19	I20	I21
I22	I23	I24	I25	I26	I27	I28
I29	I30	I31	I32	I33	I34	I35
I36	I37	I38	I39	I40	I41	I42
I43	I44	I45	I46	I47	I48	I49

Figure 5.4 Window filter structure.

5.2.3 The output image

The trained network is used as a fuzzy neural network window filter (FuNNWF) to process the input image to generate the output image.

5.3 Toy type classification

These experiments were performed to determine the effects of using rules to train a FuNN to detect and classify a toy by type. Four types of toys were used. The type of the toy is determined by the pixel intensity of the central 3x3 square section of the toy. The intensity in the centers of the toys ranges from 22 to 47. Table 5.3 shows the gray level value for each toy type.

Toy type	Center gray level value
1	24
2	31
3	38
4	45

Table 5.3 Toy type classification table.

These gray scale values were used to generate membership functions that were used in the rule development process.

5.3.1 Membership functions for fuzzification process

Table 5.4 shows the membership functions that were used for the fuzzification process for this problem. The membership functions were derived from the values in Table 5.3. Membership functions M1 and M6 were used for pixels that do not belong to a toy. According to the definition of a toy, pixels with intensities close to 0 and 255 cannot belong to a toy.

Membership function	Value	Classification
M1	0	No object
M2	24/255 = 0.0941	Type 1
M3	31/255 = 0.123	Type 2
M4	38/255 = 0.149	Type 3
M5	45/255 = 0.177	Type 4
M6	1	No object

Table 5.4 Fuzzification membership functions used in toy type classification problem

5.3.2 Membership functions for defuzzification process

Table 5.5 shows the membership functions used in the defuzzification process of this problem. The membership functions were generated using the values from Table 5.1. Membership function M1 was used when no objects were found (conventional methods would use 5 nodes to separate the classes).

Membership function	Value	Classification
M1	0	No object
M2	63/255 = 0.25	Type 1
M3	127/255 = 0.5	Type 2
M4	191/255 = 0.75	Type 3
M5	1	Type 4

Table 5.5 Defuzzification membership functions used in toy type classification problem

5.3.3 Error measure

Four error measures were calculated.

Name	Description
False positives (fp)	The number of points that should not be on the output image.
False negatives (fn)	The number of points that should be on the output image.
Misclassifications (mc)	The number of points with a correct position but misclassified.
Correct classifications (cc)	The number of points with a correct position and correct classification.

Table 5.6 Error measures table.

5.3.4 Experiment to determine how a network initialized with rules performs compared to one initialized without rules.

This was the first experiment performed on the toy problem to determine the performance of a network initialized with rules. The membership functions in 5.3.1 and 5.3.2 were used in the rule generation process.

Rule no.	Description
1	If I17=M2 and I18=M2 and I19=M2 and I24=M2 and I25=M2 and I26=M2 and I31=M2 and I32=M2 and I33=M2 Then O1=M2 (If the center 3x3 square of the toy has intensity M2 then the toy type is M2 or toy type 1)
2	If I17=M3 and I18=M3 and I19=M3 and I24=M3 and I25=M3 and I26=M3 and I31=M3 and I32=M3 and I33=M3 Then O1=M3(If the center 3x3 square of the toy has intensity M3 then the toy type is M3 or toy type 2)
3	If I17=M4 and I18=M4 and I19=M4 and I24=M4 and I25=M4 and I26=M4 and I31=M4 and I32=M4 and I33=M4 Then O1=M4

	(If the center 3x3 square of the toy has intensity M4 then the toy type is M4 or toy type 3)
4	If I17=M5 and I18=M5 and I19=M5 and I24=M5 and I25=M5 and I26=M5 and I31=M5 and I32=M5 and I33=M5 Then O1=M5(If the center 3x3 square of the toy has intensity M5 then the toy type is M5 or toy type 4)

Table 5.7 Rule used for experiment 5.3.4.

5.3.4.1 The network structure used for experiment 5.3.4

Layer	No. of nodes
Input layer	49 input nodes (7x7 window)
Condition layer	6 membership functions for each node = 294 nodes
Rule layer	4 rules + 4 free = 8 nodes
Action layer	5 nodes
Output layer	1 node

Table 5.8 Network structure for experiment 5.3.4.

Another network with the same structure was initialized with random numbers in the range {-0.1, +0.1} and trained. The performance of the network initialized with rules was compared to that of the network initialized with random numbers. The network initialized with rules was trained for 239 epochs (the network was setup to train for 300 epochs but stopped training after 239 epochs as the minimum gradient was reached in the neural network toolbox in MATLAB, this occurred in some of the following experiments). A saturation value of 1 and a quality factor of 0.25 were used as they were found to be appropriate. The network initialized with random numbers was also trained for 239 epochs.

Error type	Network with rules	Network without rules
False positives	6	4
False negatives	0	3
Misclassifications	0	13
Correct classifications	50 (100%)	34 (68%)

Table 5.9 Performance measure for experiment 5.3.4.

Table 5.9 shows the performance measure for the network trained with and without rules. The network initialized with rules has detected and classified all 50 toys correctly. Overlapping toys generated the six false positives. The network initialized with random numbers only detected 47 toys, out of which 13 were misclassifications. The network without rules also had a higher number of false negatives. From the experiments performed it was observed that the number of free nodes could be set to a value close to the number of rule nodes.

5.3.5 Experiment to measure the performance of a network over 100 runs when initialized with rules

A network initialized with rules was trained 100 times. Each time different random numbers in the range {-0.1, +0.1} were added to the network to remove the problem of symmetry [Rumelhart 1986]. The training images were the same for every run. The network structure was the same as for experiment 5.3.4.

The network initialized with rules was trained for 239 epochs (the network was setup to train for 300 epochs but stopped at 239 epochs). A saturation value of 1 and a quality factor of 0.5 were used for the rules.

Number of toys to classify = 5000 (100 images each having 50 toys)

Error type	Performance (%)
False positives	10.94
False negatives	0.18
Misclassifications	1.76
Correct classifications	98.06

Table 5.10 Performance measures for experiment 5.3.5.

The network initialized with rules had a 98% classification rate. The error increased due to overlapping toys.

5.3.6 Experiment to measure the performance of a network initialized with random numbers over 100 runs.

A network initialized with random numbers was trained 100 times. Each time the network was initialized with different random numbers in the range {-1, +1}. The training images were the same for every run. The network structure in experiment 5.3.4 was used for this run. The network was trained for 300 epochs with a saturation value of 1.

Number of toys to classify = 5000 that is 100 images each having 50 toys.

Error type	Performance (%)
False positives	14.02
False negatives	1.68
Misclassifications	14
Correct classifications	84.32

Table 5.11 Performance measures for experiment 5.3.6.

The network initialized with random numbers did not perform as well as the network initialized with rules when compared to the results in experiment 5.3.5.

5.3.7 Experiment to determine if adding rules developed to remove false positives would improve the performance of the network.

A new membership function was added to the system to generate a new rule. This new rule was added to reduce the number of false positives.

The network initialized with rules was trained for 236 epochs (the network was setup to train for 300 epochs but only trained for 236). A saturation value of one and a quality factor of 0.25 were used.

5.3.7.1 The network structure used for experiment 5.3.7

Layer	No. of nodes
Input layer	49 input nodes (7x7 window)
Condition layer	7 membership functions for each input = 343 nodes
Rule layer	5 rules + 5 free = 10 nodes
Action layer	5 nodes
Output layer	1 node

Table 5.12 Network structure for experiment 5.3.7.

Error type	Performance
False positives	2
False negatives	0
Misclassifications	1 (2%)
Correct classifications	49 (98%)

Table 5.13 Performance measures for experiment 5.3.7.

By adding a new rule the number of false positives reduced but the number of misclassifications increased to 1.

5.3.8 Experiment to determine if changing the quality factor for individual rules improved the classification performance of the network.

In this experiment the quality factor for each rule was changed.
Quality factor for rules 1 to 4 = 0.5
Quality factor for rule 5 = 0.25

The network had the same structure as in 5.3.7.

The network initialized with rules was trained for 238 epochs (the network was setup to train for 300 epochs but only trained for 238). A saturation value of one was used.

Error type	Performance
False positives	0
False negatives	0
Misclassifications	0
Correct classifications	50 (100%)

Table 5.14 Performance measure for experiment 5.3.8.

By changing the quality factor for each individual rule the classification performance of the network improved. The network achieved a 100% classification rate.

5.3.9 Experiment to determine the effects of adding small random numbers to a network initialized with rules over 100 runs.

This is a continuation of the previous experiment. A network initialized with rules was trained 100 times. Each time different random numbers in the range {-0.1, +0.1} were added to the network to remove the problem of symmetry. The training images were the same for every run. The network had the same structure as in 5.3.7.

Quality factor for rules 1 to 4 = 0.5
Quality factor for rule 5 = 0.25

The network initialized with rules was trained for 236 epochs (the network was setup to train for 300 epochs but only trained for 236). A saturation value of one was used. There were 5000 toys to classify (100 images each having 50 toys).

Error type	Mean of percentages	Variance
False positives	0.58%	0.2080
False negatives	0%	0
Misclassifications	0%	0
Correct classifications	100%	0

Table 5.15 Performance measures for experiment 5.3.9.

The network had a 100% classification rate.

5.3.10 This experiment is a continuation of the previous experiment. A network with the same structure was initialized with random number and trained.

A network was initialized with different random numbers {-0.1, +0.1} and trained 100 times. The network had the same structure as in experiment 5.3.7. The network was trained for 300 epochs with a saturation value of 1.
Number of toys = 5000 toys (100 images each having 50 toys).

Error measure	Mean of percentages	Variance
false positives	2.52%	4.0933
false negatives	1.9%	4.4722
misclassifications	14.78%	38.1797
Correct classifications	83.32%	58.0044

Table 5.16 Performance measures for experiment 5.3.10.

The network without rules did not performance as well as the network initialized with rules in experiment 5.3.9.

5.3.11 Experiment is to determine the network with the best performance from several networks initialized with rules.

30 networks were initialized with rules and trained for 200 epochs. The operational performance of each network was measured. Each network was used to classify toys in 10 images. For this experiment 20000 new toys were created and the mean intensity of the centers of these toys were used to produce new membership functions. Table 5.17 shows the new membership functions.

New Membership function	Value
Type 1	0.1043
Type 2	0.1301
Type 3	0.1564
Type 4	0.1823

Table 5.17 New membership function values for experiment 5.3.11.

These new membership functions were used to remake the rules. The basic functionality of the rules was the same as for the previous experiments. The only difference was that the centers of the membership functions were shifted slightly.

5.3.11.1 The network structure used for experiment 5.3.11

Layer	No. of nodes
Input layer	49 nodes (7x7 window)
Condition layer	8 membership functions for each input nodes = 392 nodes
Rule layer	8 rules + 7 free = 15 nodes
Action layer	5 nodes
Output layer	1 node

Table 5.18 Network structure for experiment 5.3.11.

Training time = 200 epochs

Saturation = 1

Number of networks trained = 30

Number of images for each network = 10

Number of toys in each image = 50 (Total of 500 toys)

Table 5.19 shows the mean performance measures for the 30 trained networks and the performance measures for the network with the highest classification rate.

Error	Mean percentage of errors (%)	Error of best network (%)
False positives	4.093	4.2
False negatives	0.3876	0.4
Misclassifications	2.6133	1.6
Correct classifications	97	98

Table 5.19 Error measures for experiment 5.3.11.

The average classification rate of the networks initialized with rules was 97%.

5.3.12 This is a continuation of the previous experiment. The quality factor for the rules was changed and the experiment was repeated

100 networks were initialized with rules and trained for 200 epochs in order to investigate the effects of fine-tuning the rules. The operational performance of each network was measured. Each network was used to classify toys in 30 images. The network had the same structure as in experiment 5.3.11.

Training time = 200 epochs

Saturation = 1

The quality factor for rules 1-4 was 0.75.

The quality factor for rules 5-8 was 0.2.

100 networks were trained and each network was tested on 30 test images. Each image had 50 toys.

Error type	Errors for network with rules (%)
Mean of false positives	0.5252
Mean of false negatives	0.5605
Mean of misclassifications	0.5986
Mean of correct classifications	98.7741

Table 5.20 Error measures for experiment 5.3.12.

By reducing the quality factor of rules 5-8, and increasing the quality factor of rules 1-4 the network performed better than the previous experiment.

5.3.13 This is a continuation of the previous experiment to measure the performance of the network without rules.

50 networks were initialized with random numbers in the range {-1,+1} and trained for 200 epochs. The operational performance of each network was measured by using the trained network to classify toys in 30 test images, each test image having 50 toys. The network had the same structure as in experiment 5.3.11.

Training time = 200 epochs

Saturation = 1

Error type	Errors for network without rules (%)
Mean of false positives	7.1440
Mean of false negatives	27.2147
Mean of misclassifications	26.0120
Mean of correct classifications	46.7067

Table 5.21 Error measures for experiment 5.3.13.

The network initialized with no rules (random numbers) has performed poorly when compared to the network initialized with rules in experiment 5.3.12.

5.3.14 Experiment to determine how a bad rule changes with network training

A rule is termed bad if it does not assist or accelerate network training. A bad rule was added to the system of good rules to determine how the network weights changed with network training.

Two measures were used in recording the weight change over time.

a) Incremental weight change

   This records the weight change of a rule node between epochs during training.

b) Cumulative weight change

   This records the weight change of a rule node between the first epoch and the current epoch during training.

The incremental and cumulative weight change was recorded over 175 epochs. The saturation value was one during training. The quality factor was varied between 0.5 and 1 (0.5, 0.75 and 1). The network used four good rules, two bad rules and nine free nodes. Rules 1-4 were good rules and rules five and six were bad rules.

Figure 5.5 shows the incremental weight change over 175 epochs with a quality factor of 0.5 for the rules. Figure 5.5 (A) shows the incremental weight change for the rules. Figure 5.5 (B) shows the incremental weight change for the free nodes.

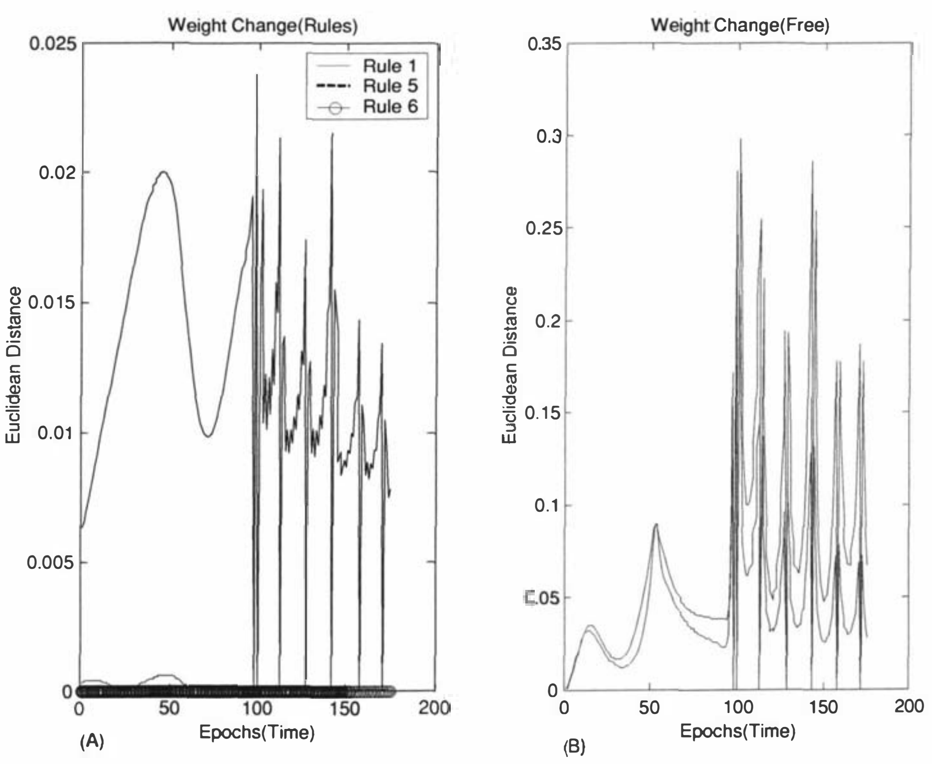


Figure 5.5 Incremental weight change graph for experiment 5.3.14.

The incremental weight change lines for rules 5 and 6 are close to the horizontal axis. The incremental weight change lines for rules 2,3 and 4 are not displayed, as they are similar to the weight change for rule 1. Figure 5.6 shows the cumulative weight change for the rules and the free nodes for a quality factor of 0.5 over 175 epochs. Table 5.22 shows the error measures for a range of quality factors.

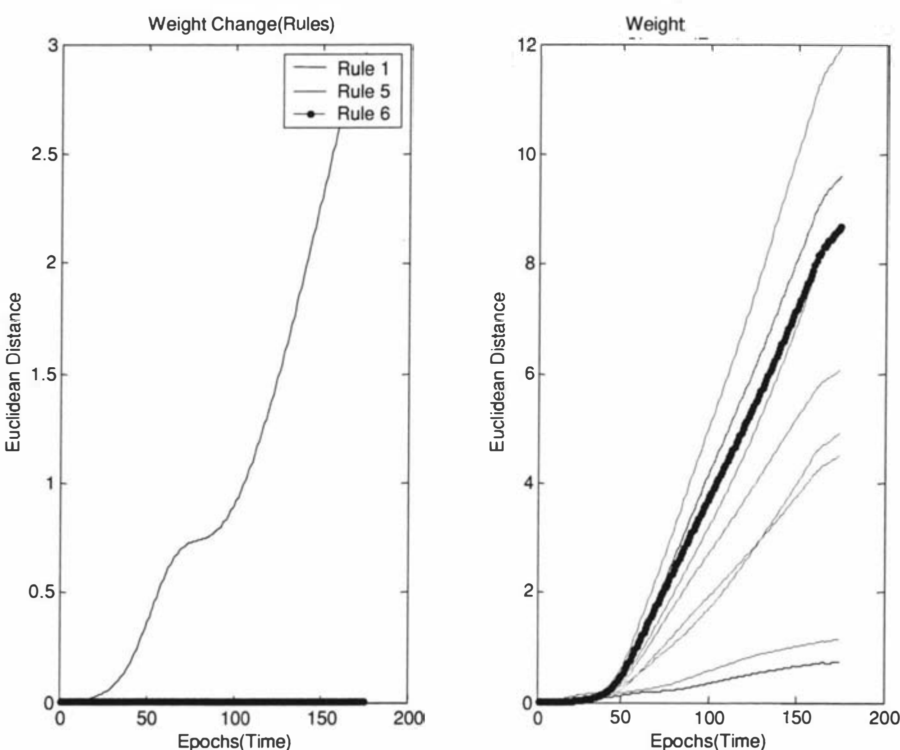


Figure 5.6 Cumulative weight change for experiment 5.3.14.

Quality Factor	False positives	False Negatives	Mis-classifications	Correct Classifications
0.5	2	0	0	50
0.75	2	0	3	47
1	3	0	4	46

Table 5.22 Error measure for a range of quality factors for experiment 5.3.14.

Rule 5 and 6, which are considered bad rules have a quality factor of 0.5. Figure 5.5 (A) shows the weights for the good rules change more than the weights for the bad rule. This is also visible in Figure 5.6. A quality factor of 0.5 may be too high

for the bad rules, forcing them to change less. Therefore the quality factor or the confidence factor for the bad rules must be lowered. Table 5.22 shows how the bad rules effect the performance of the network. As the quality factor for the good and bad rules increased the correct classification rate dropped. This also showed that the quality factor for the bad rules should be lower.

5.3.15 This Experiment is a continuation of the previous experiment to determine how a bad rule changes with network training

A bad rule was added to the system of good rules to determine how the network weights change during training. The incremental and cumulative weight change for each node in the rule layer was recorded. The incremental and cumulative weight change was recorded over 175 epochs. The saturation value was one during training. The quality factor was varied between 0.5, 0.75 and 1. The network used four good rules, two bad rules and nine free nodes. Rules 1-4 were good rules, and rules five and six were bad rules. The quality factor for the bad rules was reduced to 0.01.

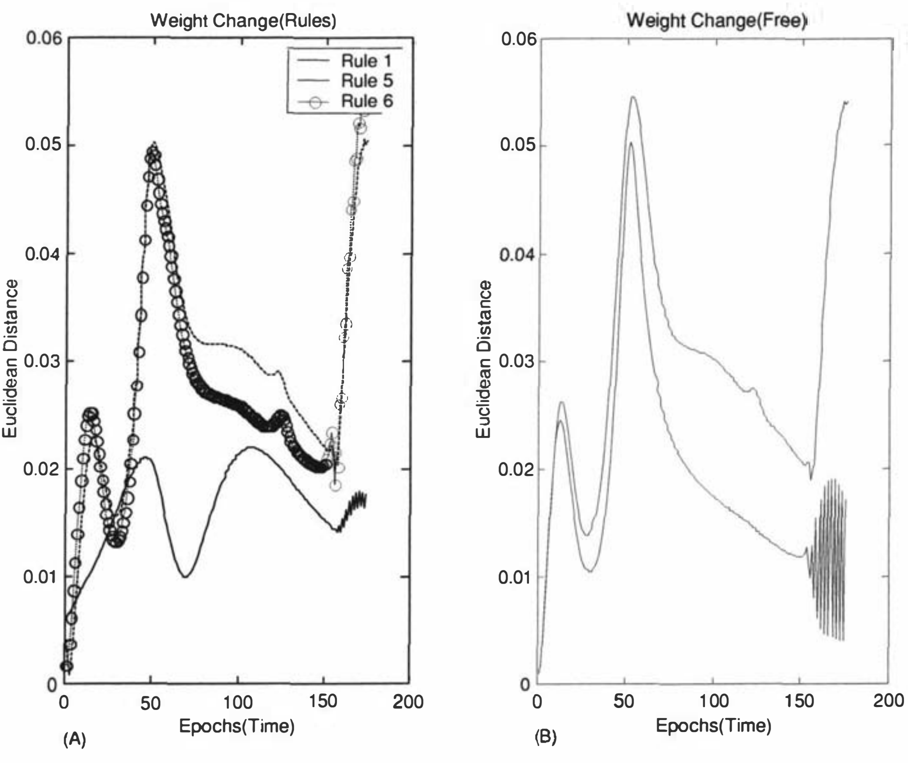


Figure 5.7 Incremental weight change graph for experiment 5.3.15.

Figure 5.7 (A) shows the incremental weight change for rules one, five and six over 175 epochs. Rules 2-4 are not shown in the diagram for clarity. They follow a similar weight change to rule one. Figure 5.7 (B) shows the incremental weight change of the free nodes. Figure 5.7 (A) shows that rules five and six (bad rules) have changed more than rules one, two, three and four (good rules).

5.3.16 Experiment to measure the operational performance of a network initialized with rules using a larger number of membership functions.

This experiment was performed to determine the effects of increasing the number of membership functions used in a FuNN. The number of membership functions was increased from 8 to 18. Sixteen rules were used in this experiment.

New membership function details			
Pixel value	Normalized value	MF#	Action
0	0	1	Noise
10	0.0392	2	
15	0.0588	3	
19	0.0745	4	
24	0.0941	5	type 1
26.7008	0.1043	6	
27.5	0.1078	7	
32	0.1255	8	type 2
33.1755	0.1301	9	
34.5	0.1353	10	
39	0.1529	11	type 3
40.0384	0.1564	12	
41	0.1608	13	
45	0.1765	14	type 4
46.6688	0.1823	15	
47.5	0.1863	16	
60	0.235	17	noise
255	1	18	

Table 5.23 New membership function details for experiment 5.3.16.

20 Networks were generated and each network was trained for 200 epochs. Each network performed 10 operational runs (10 images, with 50 toys each).

Error type	Errors for network with rules (%)
Mean of false positives	0.55
Mean of false negatives	0.6
Mean of misclassifications	0.42
Mean of correct classifications	98.987

Table 5.24 Error measures for experiment 5.3.16.

Table 5.24 shows the operational performance errors. The classification rate was higher (0.2%) than when eight membership functions were used (experiment 5.3.12). Although the network was trained for the 200 epochs the training time was longer as the network was larger due to the increased number of membership functions.

5.4 Toy orientation classification

The previous section investigated how a FuNN was used to classify toys by type. The experiments in this section were performed to determine the effects of using rules to train a FuNN to detect and classify toys by orientation.

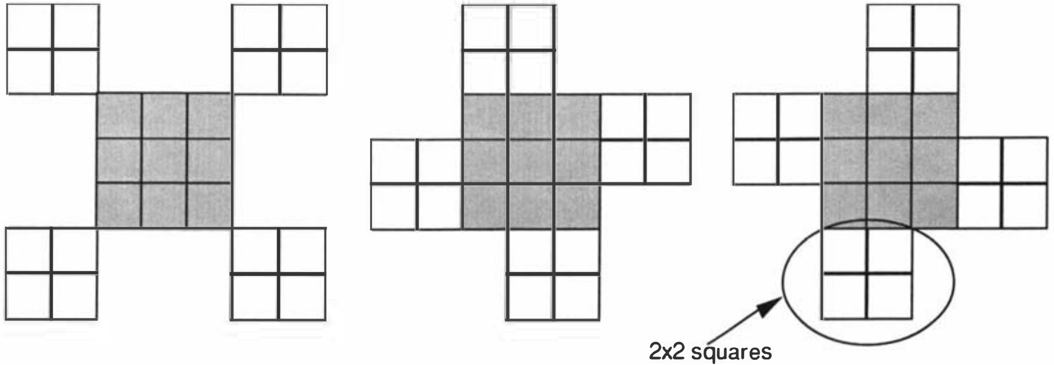


Figure 5.8 Toy orientation problem defined by the placement of the 2x2 square around the 3x3 larger square.

Figure 5.8 shows the three possible orientations of a toy. The orientation is defined by the placement of the four 2x2 squares around the central 3x3 square. The central 3x3 square can be of any toy type.

5.4.1 Membership functions for fuzzification process

Table 5.25 shows the membership functions used in the fuzzification process for this problem. Membership functions M1 and M7 are used for pixels that do not belong to a toy. Membership functions M2, M3, M4 and M5 are used to identify the toy by type. Membership function M6 is used to classify the toys by orientation.

Membership function	Value
M1	0
M2	0.0941
M3	0.123
M4	0.149
M5	0.177
M6	0.235 - used for the orientation of the toy
M7	1

Table 5.25 Fuzzification membership functions for toy orientation.

5.4.2 Defuzzification membership functions

Table 5.26 shows the membership functions used in the defuzzification process. Membership function M1 is used when no objects are found.

Membership function	Value	Classification
M1	0	No object
M2	84/255 = 0.3294	Type 1
M3	169/255 = 0.6627	Type 2
M4	1	Type 3

Table 5.26 Defuzzification membership functions for toy orientation.

5.4.3 Experiment to compare the performance of a network initialized with rules and random numbers.

The following experiment was performed to compare the performance of a network initialized with and without rules. The membership functions in 5.4.1 and 5.4.2 were used in the rule generation process. The three rules in the following table were used to initialize the network.

Rule no.	Description
1	If I1=M6 and I2=M6 and I8=M6 and I9=M6 and I6=M6 and I7=M6 and I13=M6 and I14=M6 and I36=M6 and I37=M6 and I43=M6 and I44=M6 and I41=M6 and I42=M6 and I48=M6 and I49=M6 Then O1=M2 (Rule developed to classify a toy that has orientation 1)
2	If I4=M6 and I5=M6 and I11=M6 and I12=M6 and I15=M6 and I16=M6 and I22=M6 and I23=M6 and I27=M6 and I28=M6 and I34=M6 and I35=M6 and I38=M6 and I39=M6 and I45=M6 and I46=M6 Then O1=M3 (Rule developed to classify a toy that has orientation 2)
3	If I3=M6 and I4=M6 and I10=M6 and I11=M6 and I22=M6 and I23=M6 and I29=M6 and I30=M6 and I20=M6 and I21=M6 and I27=M6 and I28=M6 and I39=M6 and I40=M6 and I46=M6 and I47=M6 Then O1=M4 (Rule developed to classify a toy that has orientation 3)

Table 5.27 Rules for experiment 5.4.3.

5.4.3.1 The network structure used for experiment 5.4.3

Layer	No. of nodes
Input layer	49 nodes (7x7 window)
Condition layer	7 membership functions for each node = 343 nodes
Rule layer	3 rule + 3 free = 6 nodes
Action layer	4 nodes
Output layer	1 node

Table 5.28 Network structure for experiment 5.4.3.

Another network with the same structure was initialized with random numbers in the range {-0.1,+0.1} and trained. The performance of the network initialized with rules was compared to that of the network initialized with random numbers. A quality factor of 0.5 and a saturation value of 1 were used. Training time was 175 epochs. Table 5.29 shows the performance of the network with and without rules. The network initialized with rules has detected all the toys in the input image but misclassified two of them. The network initialized with rules has performed better than the one initialized with random numbers.

Error type	Network with rules	Network without rules
False positives	0	0
False negatives	0	12
Misclassifications	2	1
Correct classifications	48 (96%)	37 (74%)

Table 5.29 Performance measures for experiment 5.4.3.

5.4.4 Experiment to determine network performance with a new set of fuzzification membership functions and extended rules.

In the previous experiment three rules were used to initialize the network. A new set of fuzzy membership functions was generated. These new fuzzy membership functions were used to develop rules that were used to initialize the network. This experiment was performed to determine if the new membership functions and rules would improve network performance. Table 5.30 shows the new membership functions. The rules in experiment 5.4.3 were extended using the new fuzzy labels.

Membership function	New value
M1	0
M2	0.0748
M3	0.2275
M4	1

Table 5.30 New membership functions for toy orientation problem.

Membership functions M1 and M2 are used to detect the artificial noise in the input image. Membership function M3 is used to detect the 2x2 squares surrounding the center 3x3 square of a toy. More input nodes were added to the existing rules to generate the new rules. In the previous experiment, 16 input nodes were used to generate each rule. Each rule in this experiment was generated from 40 input nodes. This can be thought of as adding more problem knowledge to the system.

5.4.4.1 The network structure used for experiment 5.4.4

Layer	No. of nodes
Input layer	49 nodes (7x7 window)
Condition layer	4 membership functions for each node = 196 nodes
Rule layer	3 rule + 3 free = 6 nodes
Action layer	4 nodes
Output layer	1 node

Table 5.31 Network structure for experiment 5.4.4.

Another network with the same structure was initialized with random numbers in the range {-1, +1} and trained. The performance of the network initialized with rules was compared to that of the network initialized with random numbers. A quality factor of 0.5 and a saturation value of one were used. Training time was 175 epochs. Table 5.32 shows the performance of the network with and without rules.

Error type	Network with rules	Network without rules
False positives	0	0
False negatives	0	0
Misclassifications	0	16
Correct classifications	50 (100%)	34 (68%)

Table 5.32 Performance measures for experiment 5.4.4.

The new network structure is much smaller than the one used in experiment 5.4.3 resulting in less training time. The extended rules improved the performance of the network.

5.4.5 A statistical experiment to determine the performance of a network.

This experiment investigates the effect of adding different random numbers to a network initialized with rules. A network was trained 100 times. Each time a different set of random numbers in the range {-0.1, +0.1} was added to the network initialized with rules and trained. The performance was measured after each training session. An additional network with the same structure was generated. This network was also trained 100 times. Each time this network was initialized with a different set of random numbers in the range {-1, +1}.

A quality factor of 0.5 and a saturation value of 1 were used for the network initialized with rules. Each network was trained for 175 epochs and the performance was measured. The network structure was the same as in experiment 5.4.4.

Error type	Network with rules	Network without rules
False positives	0%	0%
False negatives	0.24%	16.22%
Misclassifications	0.34%	8.42%
Correct classifications	98.34%	74.96%

Table 5.33 Performance measures for experiment 5.4.5.

Table 5.33 shows the performance measures. The mean of the errors from the 100 runs was converted to a percentage and displayed in the table. As the table shows the misclassifications and false negatives represent the overlapping toys which the network could not classify. The sum of the columns in Table 5.33 is not 100%. This is because the networks have not detected all of the toys. Overlapping toys can cause this error.

5.4.6 Experiment to determine the operational performance of a population of networks tested on several new images.

This experiment was performed to determine the operational performance of a population of networks. A population of 50 FuNNs were created. Each network was initialized with the same rules. A different set of random numbers were added to each network and trained for 175 epochs. A quality factor of 0.5 and a saturation value of 1 were used. The network structure was the same as in experiment 5.4.4. Each trained network was tested using 30 test images.

Error type	Network with rules
False positives	5.86%
False negatives	0.0693%
Misclassifications	0.232%
Correct classifications	99.632%

Table 5.34 Performance measures for experiment 5.4.6.

Table 5.34 shows the mean performance measures of the networks. The mean of the errors is represented as percentages. As the table shows the false positives have increased. This could be due to overlapping toys in the test images.

5.4.7 Experiment to determine the incremental and cumulative weight change during training

This experiment was performed to investigate the incremental and cumulative weight change for the nodes in the rule layer of the network. The hypothesis is that the rule node weights should change less than the free nodes as the rule nodes embody problem knowledge. A network initialized with rules was trained for 175 epochs. A saturation value of one was used. The quality factor (Q) for the rules was varied between 0.5 and 1 (0.5, 0.74 and 1). The incremental and cumulative weight change was recorded.

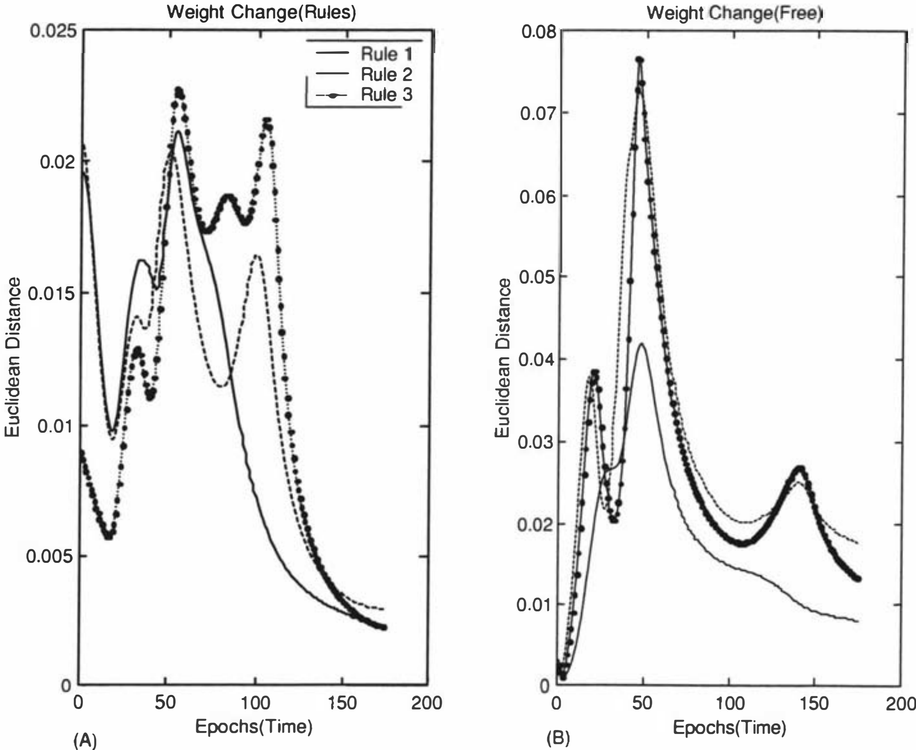


Figure 5.9 Incremental weight change of the network (Q=0.5) over 175 epochs for experiment 5.4.7. (A) is the incremental weight change for the rules and (B) is the incremental weight change for the free nodes in the network.

Figure 5.9 shows the incremental weight change for the network over 175 epochs. The incremental weight change graph for the network when Q was 0.75 and 1 was similar to graph in Figure 5.9. As expected the free nodes changed more than the rule nodes.

Figure 5.10 shows the cumulative weight change for the rule and free nodes over 175 epochs. The cumulative weight change graph for the network when Q was 0.75 and 1 was similar to the graph in Figure 5.10. Figure 5.10 shows that the cumulative weight change for each rule node is less than the cumulative weight change of the free nodes.

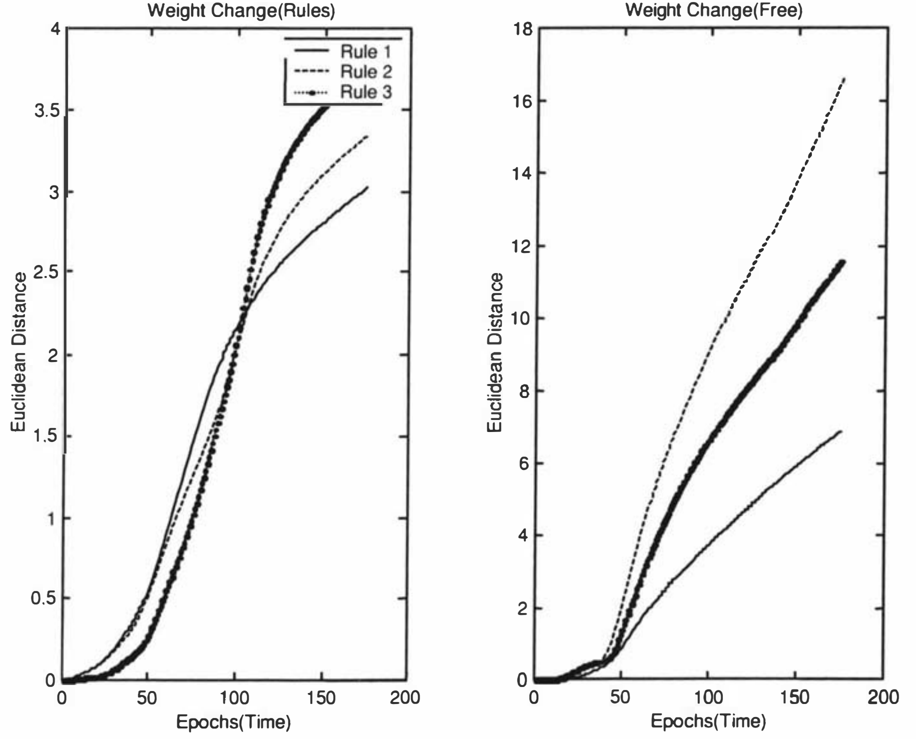


Figure 5.10 Cumulative weight change for the network (Q=0.5) over 175 epochs for experiment 5.4.7. (A) is the cumulative weight change for the rule nodes and (B) is the cumulative weight change for the free nodes.

Table 5.35 shows the error measure for a range of quality factors. The increase in the quality factor has not affected the correct classification rate.

Quality Factor	False Positives	False Negatives	Mis-Classification	Correct Classification as a percentage
0.5	0	0	0	100%
0.75	0	0	0	100%
1	0	0	0	100%

Table 5.35 Error measure for the network when Q=0.5, 0.75 and 1 for experiment 5.4.7.

5.4.8 This is a continuation of the previous experiment to determine the effects of adding free nodes to the network

This is a continuation from the previous experiment. The number of free nodes was increased from three to nine and the incremental and cumulative weight change was recorded.

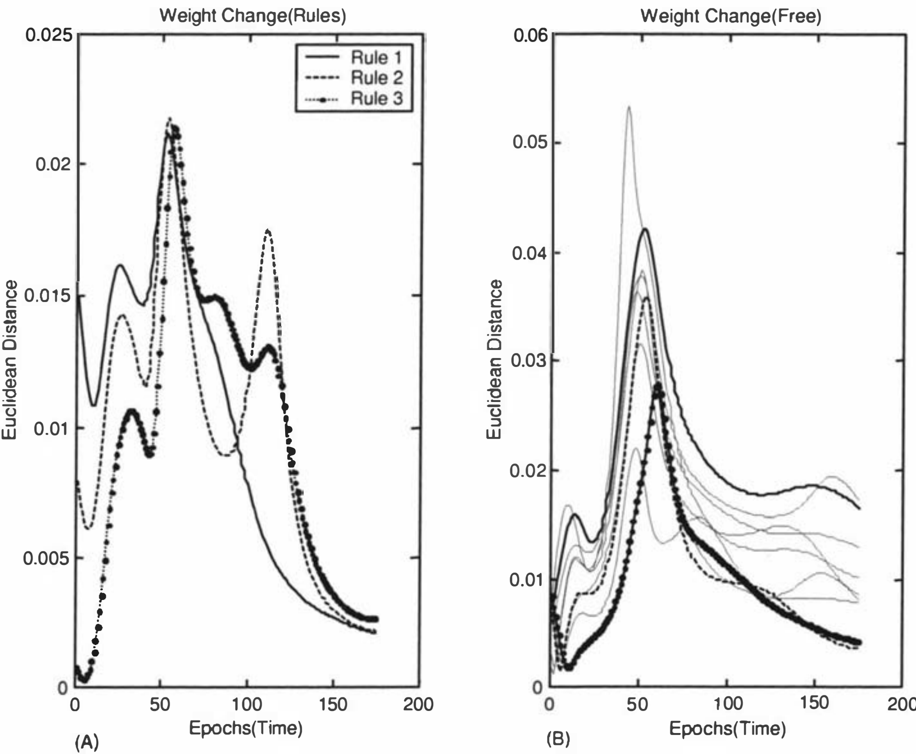


Figure 5.11 Incremental weight change of the network (Q=0.5) over 175 epochs for experiment 5.4.8. (A) is the incremental weight change for the rules and (B) is the incremental weight change for the free nodes in the network.

Figure 5.11 shows the incremental weight change during training. Both the rule and free nodes change less over time compared to the incremental weight change in experiment 5.4.7. This would be expected, as the network has a greater degree of freedom. Figure 5.12 displays the same behavior.

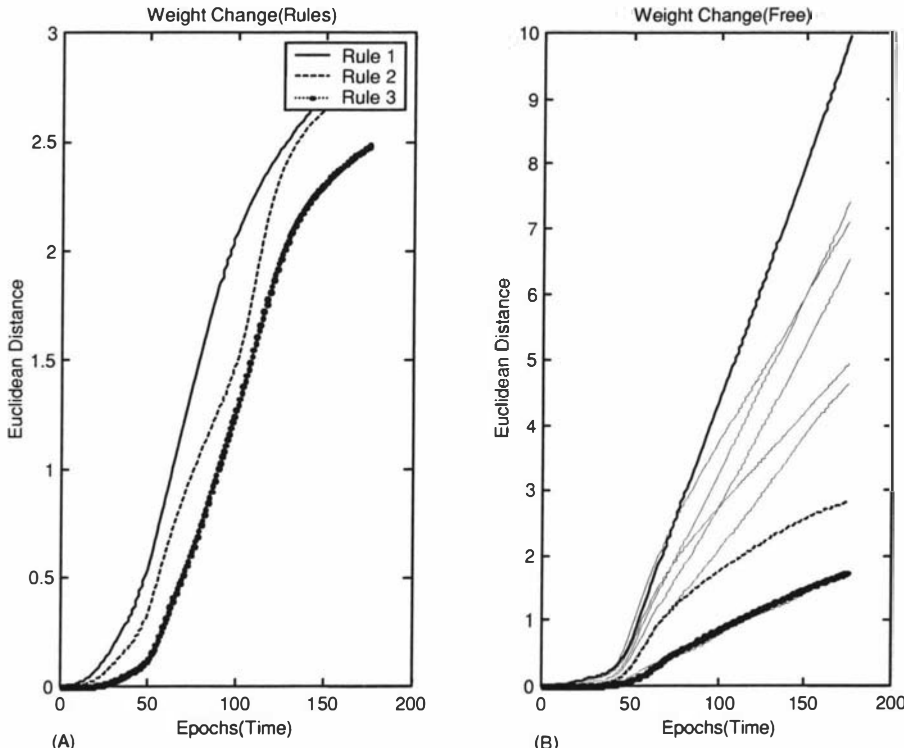


Figure 5.12 Cumulative weight change for the network (Q=0.5) over 175 epochs for experiment 5.4.8. (A) is the cumulative weight change for the rule nodes and (B) is the cumulative weight change for the free nodes.

The error measures are the same as in Table 5.35.

5.4.9 Experiment to determine the effects of adding a bad rule to the network

This experiment was performed to determine the effects of adding a bad rule to the network. The network was then trained for 175 epochs and the incremental and cumulative weight change was measured. Rules one, two and three were good rules. The quality factor for these rules was varied between 0.5 and 1 (0.5, 0.75 and 1). Rule four was a bad rule. The quality factor for this rule was 0.01. The network structure was the same as in experiment 5.4.4. The only change was that the number of free nodes was reduced to two, as there were four rule nodes. Figure 5.13 shows the incremental weight change during training for the rule and free nodes. Figure 5.14 shows the cumulative weight change during training for the rule and free nodes.

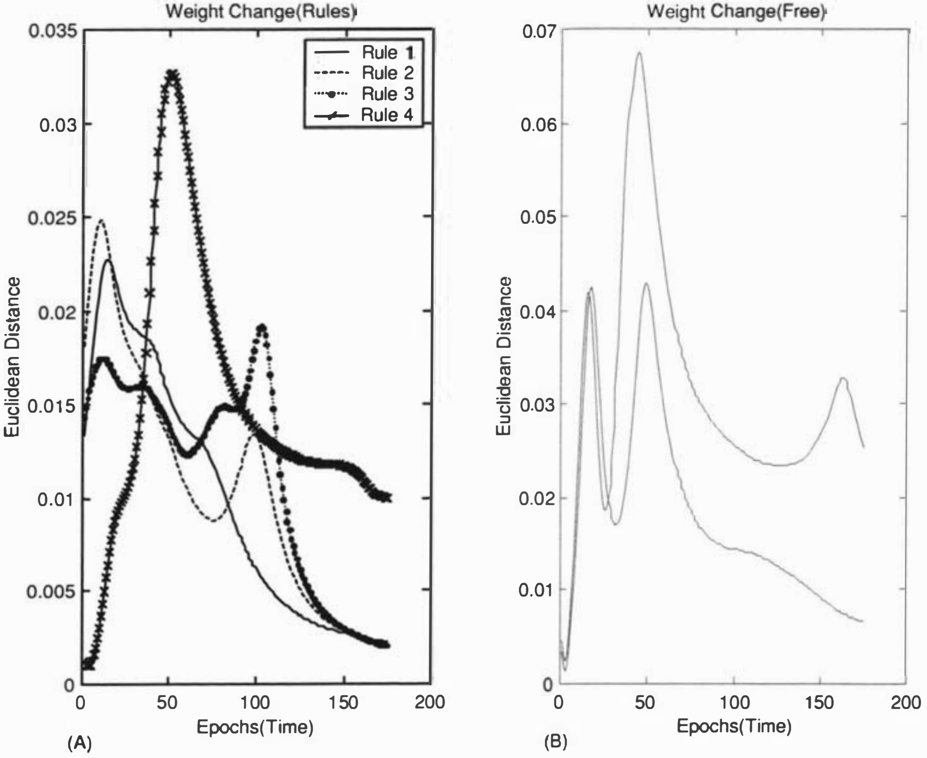


Figure 5.13 Incremental weight change when (Q=0.5) for experiment 5.4.9.

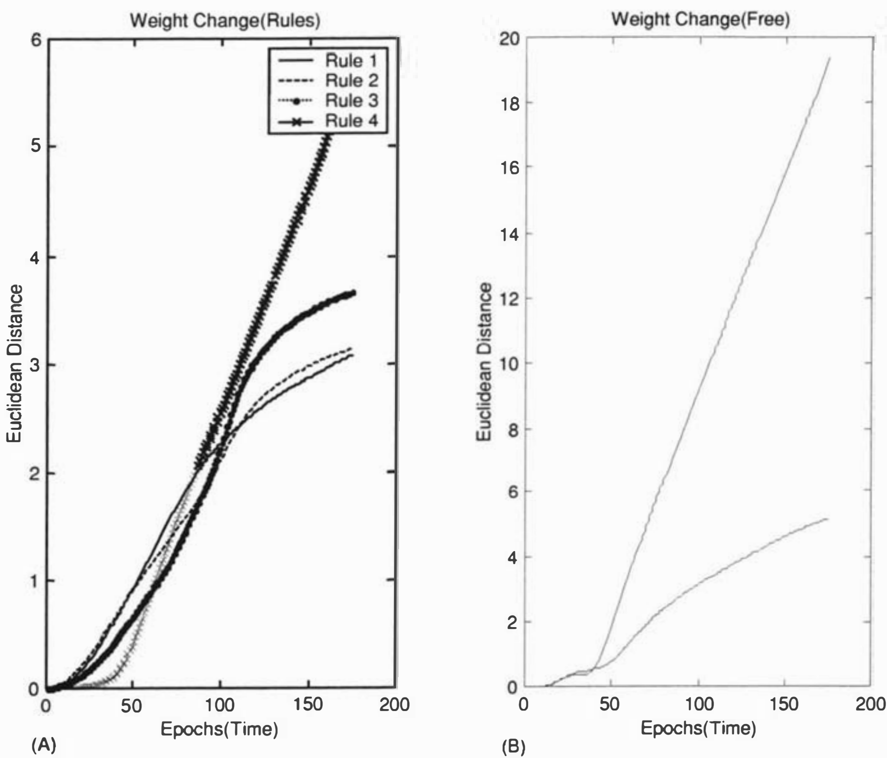


Figure 5.14 Cumulative weight change for the network (Q=0.5) over 175 epochs for experiment 5.4.9. (A) is the cumulative weight change for the rule nodes and (B) is the cumulative weight change for the free nodes.

As expected Figure 5.13 and Figure 5.14 show that rule four, which is the bad rule, has changed more than the good rules.

5.4.10 Experiment to determine the range of operational performances shown by a population of networks when initialized with rules

This experiment was performed to determine the operational performance of a population of networks. 50 networks were generated and each network was initialized with three rules. Random numbers in the range {-0.1, +0.1} were added to each network initialized with rules to remove symmetry. The network structure was the same as in experiment 5.4.4.

Training time for each network = 175 epochs.
Saturation value = 1.
Quality factor for rules = 0.5.
Number of test images for each network = 30.

Error name	Value
False positives	5.68%
False negatives	0.0693%
Misclassifications	0.232%
Correct classifications	99.632%

Table 5.36 Error measures for experiment 5.4.10.

Table 5.36 shows the error measures for this experiment. The mean of the errors is represented as a percentage. The false positives in the table are a result of overlapping toys.

5.5 Classification of Rules

A classification scheme that rates the relationship between a rule and problem knowledge is developed below.

5.5.1 A strong rule

A rule is said to be strong if it has a high confidence level. A rule derived from the most obvious problem knowledge can be considered a strong rule. A high ''Quality'' factor (Q=0.5-1) can be placed on these rules.

5.5.2 A weak rule

A rule is said to be weak if a high confidence level cannot be placed on that rule. A rule derived from less obvious problem knowledge can be considered a weak rule. When deriving this type of rule, only a small portion of the rule is extracted from problem knowledge. The rest is ''guessed'' by intuition (by the human). A low ''Quality'' factor (Q=0.01-0.2) can be placed on these rules.

5.5.3 A weak incorrect rule

If the relationship between a rule and the problem is very weak and incorrect then a rule is said to be a weak incorrect rule.

5.5.4 A strong incorrect rule

If the relationship between a rule and the problem is non-existent then the rule is a strong incorrect rule.

5.5.5 A bad rule

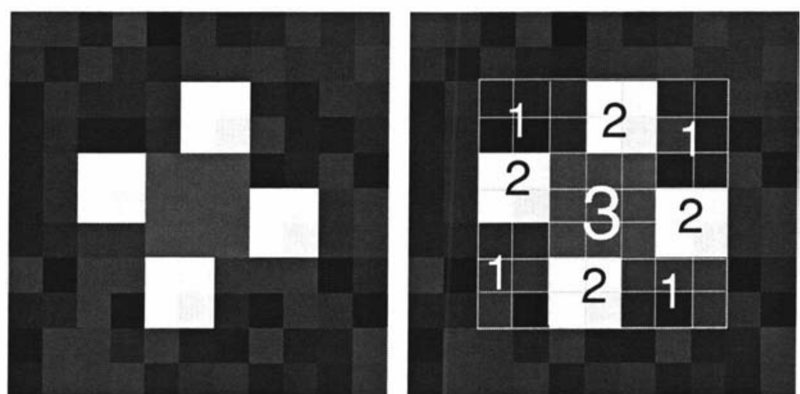
A bad rule can be classified as a combination of a weak rule and a weak incorrect rule. It is difficult to justify whether a rule is bad as a non-linear relationship exists

between the rules and the problem. Including a bad rule may have an indirect effect on network performance.

5.6 Conclusions

5.6.1 Specific Conclusions

5.6.1.1 Definition of obvious information



The image on the left is an image of a toy. Our problem is to detect the presence of a toy. The image on the right is an image of a 7x7 window filter overlaid on the toy. The problem can be defined by using the pixel intensities outlined by the filter grid. In the above image (right) the obvious information would be:

> IF,
> the pixels of the central 3x3 area (denoted by **3**) have medium intensity and the pixels of the surrounding 2x2 areas (denoted by **2**) have high intensity and the pixels in the rest of the image (denoted by **1**) have low intensity
> THEN,
> a toy is present at that location.

The above problem knowledge is very obvious from the window. This type of knowledge can be called ''obvious knowledge''. This knowledge can be used to generate a ''strong rule''.

5.6.1.2 Use of obvious knowledge about the problem

All obvious knowledge about the problem should be used when generating rules. In the initial experiments for the toy orientation problem a correct classification rate of 96% and misclassification rate of 4% was achieved by using rules (experiment 5.4.3). However the rules used did not incorporate all the obvious knowledge. A 100% classification rate (0% misclassification) was achieved when more of the obvious problem knowledge was incorporated into the rules (experiment 5.4.4).

5.6.1.3 Use of proper membership functions

The membership functions for the initial experiments for the toy problem were determined by inspection. The results of the experiments performed using these membership functions had a few misclassifications. There were two types of misclassifications.

1. Misclassification due to overlapping toys.
2. Misclassification due to toys lying close to the midpoint of two membership functions.

20000 new toys were generated to recalculate the membership functions. After the new membership functions were generated, the problem in point 2 did not arise.

5.6.1.4 The number of membership functions

The membership functions used for the toy problem were derived from the obvious problem knowledge. Eight membership functions were used for toy type classification and four membership functions were used for toy orientation classification. The membership functions were derived from the most obvious

visible problem knowledge. When the number of membership functions used in the toy type classification problem was increased to 18, the network had a classification rate of 98.98% (experiment 5.3.16), a slight increase from 98.77% (experiment 5.3.12) from using 8 membership functions. The only drawback from using a higher number of membership functions was the training time. As the network size increased the training time also increased.

5.6.1.5 Optimal quality factor

A high quality factor should be used for strong rules and a low quality factor should be used for weak rules. This is demonstrated in experiment 5.3.8. When a quality factor of 0.25 was given to all the rules the network had a 98% correct classification rate and a 4% false positive generation rate. After the good rules were given a higher quality factor (0.5) the network gave a 100% correct classification rate and a 0% false positive generation rate. An incorrect rule with a high quality factor can have a degrading effect on network performance as the correct rules and free nodes will try to compensate for them. The graphs in experiments 5.3.14 and 5.3.15 show how the quality factor for the incorrect rules affects the weight change of the nodes in the rules layer. As the quality factor drops the weights associated with the incorrect rules can change more easily and the weights associated with the rule nodes change less. This can also be observed in the natural problems that will be discussed in the following chapters. By using an optimal set of quality factors for the rules the classification rate of the network increased. The value of the quality factor is not always apparent. If in doubt it is always better to use a lower value for the quality factor for an uncertain rule.

5.6.1.6 Influence of incorrect rules on the network

Use of an incorrect rule can impair a network's classification rate. Experiment 5.3.14 shows how incorrect rules affect network performance in toy classification. The correct rules compensate for the effects of the incorrect rules that have a high quality factor. The use of incorrect rules can result in poorer network performance. As the quality factor for the incorrect rules increased the correct classification rate of the network reduced.

5.6.2 General Conclusions

5.6.2.1 The weight change

Rules that have a higher association (strong rule) with the problem tend to change less than the weights associated with free nodes. Stronger correct rules change less than weaker correct rules. Incorrect rules change more than correct rules. It is difficult to associate a relationship between rules as a weak rule may sometimes have a small but beneficial influence on network performance.

5.6.2.2 Random numbers

Small random numbers are added to the weights in the network in order to remove the problem of symmetry. Experiments 5.3.9 and 5.4.5 show that adding different sets of random numbers in the range of {-0.1, +0.1} to a network initialized with rules had a minor effect on the classification rate of the network.

Chapter 6

6 Wane edge detection

This chapter describes how a fuzzy neural network window filter (FuNNWF) was used to solve an industrial problem, that is the detection of the wane edge on a plank of sawn timber. The chapter investigates the effects of rule selection and tuning on the performance of a fuzzy neural network applied to the task. These experiments were one of the first practical problems and were exploratory in nature. Later problems were investigated in a more systematic manner using the information derived from the wane edge detection application which had been originally worked on by Siroki [Siroki 1998].

6.1 The wane edge detection problem

The timber components industry could make better utilization of its timber if sawn planks could be characterized in terms of the number of defects, their type and size. Knots, pith, check and split are some of the typical defects in wood. In order to study this problem a database has been assembled containing defect size, position and type for a range of representative planks. Manual measurement of defects is expensive and time consuming so a semi-automated process has been investigated in which all the defects except wane are marked and classified manually. The measurements of the defect (position, size and type) are then

determined by a computer visual based inspection system as the planks are carried through the inspection system on a conveyor belt [Siroki 1998].

Figure 6.1 shows a typical plank of wood with the defects manually marked. The research work described in this chapter is focused on wane edge detection. Wane is that area on a plank whose surface is the original de-barked surface of the log and is not co-planar with the plank surface. It is delineated by the plank edge and the edge formed where its surface meets the plank surface. The wane edge must be detected in order to optimize the subsequent cutting of the plank into components. The wane edge must be differentiated from grain lines, surface intensity changes, knots and other defects [Chaplin 2000].

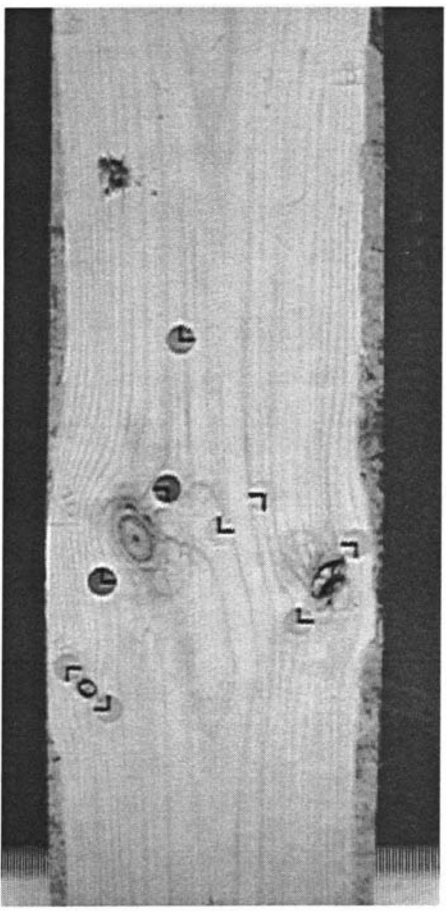


Figure 6.1 Image of a plank with manually marked defects.

The wane edge usually appears on one or both edges of the plank. Conventional image processing techniques do not perform adequately in detecting these edges. Figure 6.2 shows the output generated by processing the original image with a

simple edge detection filter and thresholding. The outer wane edge is visible in image Figure 6.2 (b) but what is required is the inner wane edge. The required wane edges cannot be detected using a simple vertical edge detector.

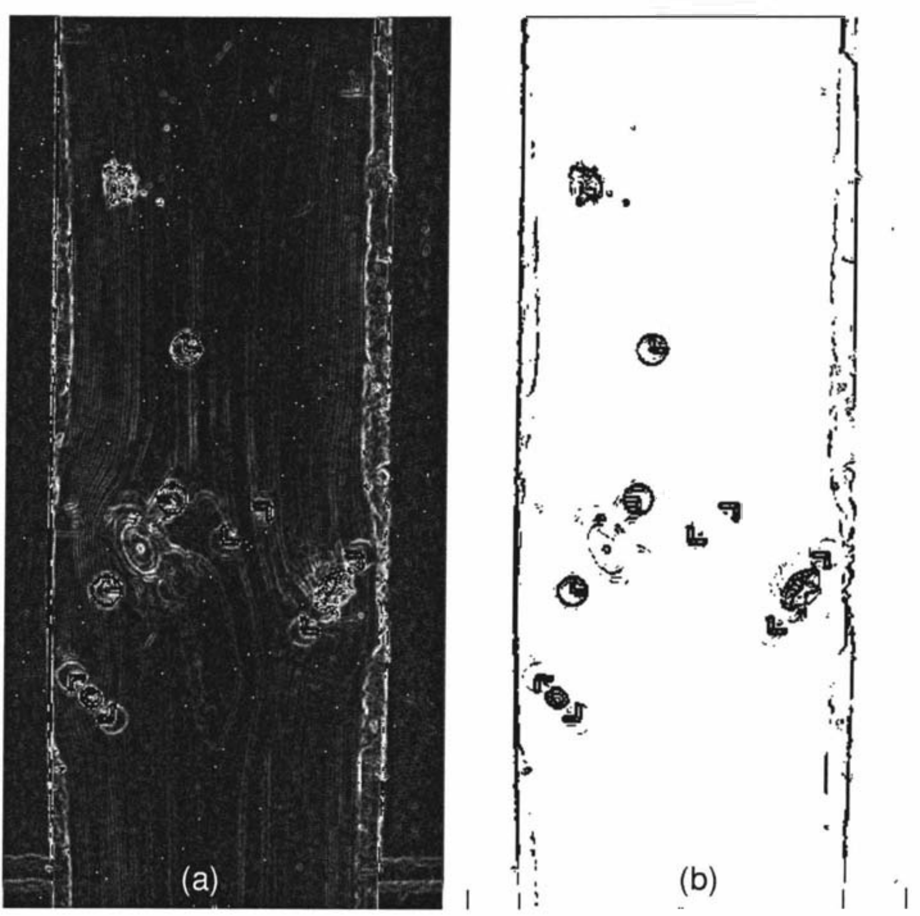


Figure 6.2 The images show the output generated by processing the original image with a vertical edge detector and thresholding. Image (a) is the output generated by processing figure 6.1 with a common edge detection filter. Image (b) was generated by thresholding image (a).

The wane edge can be poorly defined, discontinuous or obscured by defects such as knots. Though there is some subjectivity in marking its position humans can detect the wane edge with little difficulty. Manually wane edge marked input-target image pairs can be used to train a neural network to detect wane edges automatically.

For the following reasons wane edge detection is both a difficult image processing problem and a difficult neural network training task.

- The process of creating a target image by hand is subjective and therefore may include imprecision and errors.
- When creating a target image, a person can see the whole image and so use global information, however NNWF are trained and tested by presenting only a small window that is only using local information. The network has less information available than does the human generating the target training pairs.
- The required size and shape of the optimum window filter is not known in advance as planks may have other defects such as knots that could appear to be a wane edge when viewed through the window filter.

6.2 Problem knowledge for wane edge detection

Before training can begin, problem knowledge has to be extracted and converted into rules. These rules can then be used to initialize the weights of the fuzzy neural network prior to training.

6.2.1 Problem knowledge expressed in high level terms

1. The wane edge is continuous and runs along the length of the plank.
2. It is predominantly vertical relative to the surface of the plank under investigation.
3. A left wane edge exists when there is a horizontal transition in the intensity from dark to light and a right wane edge exists when there is a horizontal transition in intensity from light to dark.
4. A wane edge is generally found close to the left or right edge of the plank.

Only points one, two and three are used to generate rules. Point four is defined using global information; therefore it is not used to generate rules as the window filter used for the wane edge detection experiments, only uses the local information available in the window.

6.2.2 The size and shape of the window filter

A 3x3 pixel window filter was used for the initial experiments. The most suitable size and shape of the window filter was not known prior to the experiments. As the experiments progressed the size and shape of the window filter was varied and refined.

6.3 Training data

Figure 6.3 shows typical training images used for the wane edge detection problem. The images are 256 gray scale images, image (a) is the input image and image (b) is the target image. The target image was generated by hand. The wane edges were drawn using a 1-pixel wide pen. The positions of the wane edges were not perfect as they are drawn by a human and are therefore subjective, for this reason a Gaussian blur filter with a radius of 1.4 was passed over the wane edge image to produce the target image (b).

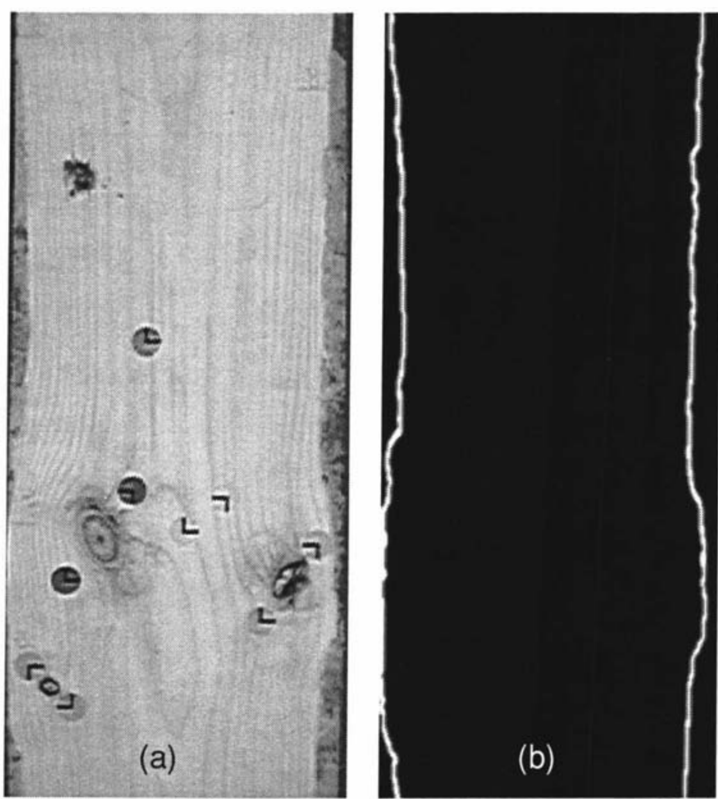


Figure 6.3 Training images for wane problem.

Figure 6.4 shows the input image and the sections of a plank that were used to generate the training data. Each rectangular section brackets a feature of the plank that was used for training. The window filter was passed over each rectangular section to generate a set of training data. These were then concatenated to generate the complete training data set. The training set had over 6000 input-target training pairs. When the window filter size was changed the training data was regenerated. The training data was normalized between zero and one.

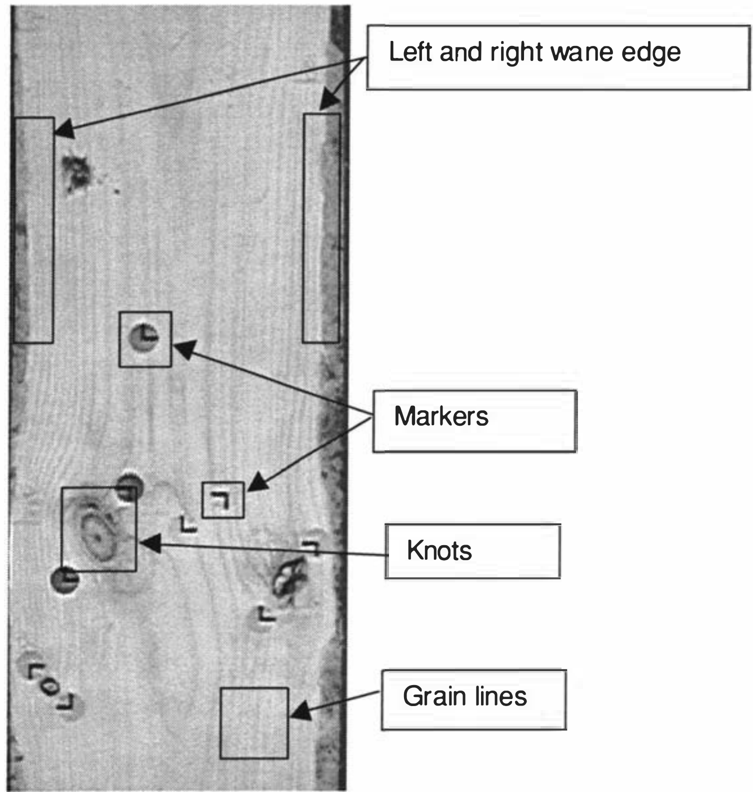


Figure 6.4 The training data was generated using the rectangular sections in the image.

6.4 The membership functions

The membership functions are generated from the available problem knowledge. Point three of section 6.2.1 states that a wane edge exists when there is a horizontal transition in intensity from dark to light or light to dark in the window filter. The problem knowledge is defined in terms of low and high intensities, so two states

representing low and high intensity can be used for the membership functions. Table 6.1 shows the two membership functions that were used in the experiments.

Membership function	Description	Value
LO	This membership function represents the dark areas on the plank covered with bark.	0
HI	This membership function represents the lighter usable areas on the plank.	1

Table 6.1 Membership functions used for wane edge detection.

6.5 The error measure

An error measure was developed to determine the error between the target and output image where the output image is the image generated by processing the input image with the FuNN after training. The measure may in fact not be perfect or at least satisfactory but the target image may not be perfect as it is drawn by hand and estimated.

6.5.1 The error measure function

Figure 6.5 shows a section of the plank with the superimposed wane edge.

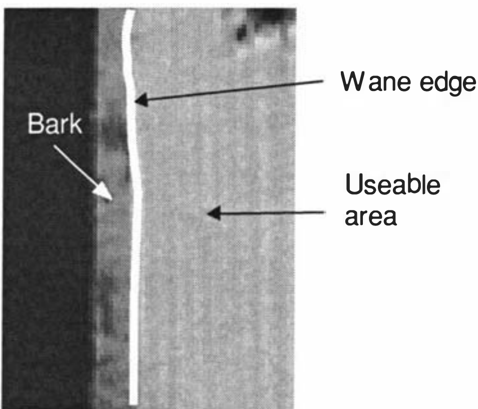


Figure 6.5 Image showing the wane edge, bark and useable area on the plank of wood.

The error calculation function uses the target and output image as a basis to calculate the error. Prior to calculating the error, both images are processed using a line thinning routine and converted to binary form. The error measures are then calculated from the two resulting binary images.

6.5.1.1 False positives

The output image consists of the wane edge and other pixels that do not belong to the wane edge. These other pixels are false positives. A false positive is also generated if the position of a pixel in the wane edge has a displacement error when compared to the corresponding pixel in the target image. The mean of these displacement errors is calculated in the error function.

6.5.1.2 False negatives

If a pixel should be but is not present on the wane edge then a false negative is generated.

The *calcerror* function uses the target and output image as input parameters. The function and the weightings for the *errorValue* were generated through an iterative process.

[m,s,fn,fp, errorValue]=calcerror(Target, Output)

where,

m - mean horizontal distance measure

s - variance of m

fn – false negative

fp – false positives

errorValue = 2s + 0.1fn + 0.2fp

This error measure will be used to calculate the error between the target and output images. The RMS (root mean square) error will be used to calculate the training error.

6.6 The experiments

The best size for the window filter to be used was not known in advance. Previous work done by Siroki [Siroki 1998] involved using a 3x3 fuzzy neural network window filter (FuNNWF) to emulate the operation of the Sobel filter. So the 3x3 neural network window filter (NNWF) was used as a basis for the experiments described in this chapter. Several experiments were performed varying the size and orientation of the window filter, varying the number of rules and training time. Table 6.2 shows the variation in the size of the window filter used for the experiments in this chapter. The first experiments started with a 3x3 window filter. As the experiments progressed the size of the window filter was varied as shown in Table 6.2. All of the experiments are not discussed as they all produce similar results.

3x3	3x5	3x7	3x9
5x5	5x7	5x9	
7x7	7x9		
9x9	9x23		

Table 6.2 Table showing the variation in the size of the window filter used for the wane edge detection problem.

6.6.1 Experiment to detect the edges using a 3x3 window filter

Previous experiments performed by Siroki [Siroki 1998] involved using a 3x3 neural network window filter to emulate the Sobel filter. The rules used for the Sobel filter were modified to represent the wane edge detection problem.

6.6.1.1 Rules for experiment 6.6.1

Rule	Description
1	If I4=LO and I6=HI Then Output = HI
2	If I4=HI and I6=LO Then Output = HI
3	If I4=LO and I6=LO Then Output = LO
4	If I4=HI and I6=HI Then Output = LO
5	If I1=LO and I7=LO and I3=HI and I9=HI Then Output = HI
6	If I1=HI and I7=HI and I3=LO and I9=LO Then Output = HI
7	If I1=LO and I7=LO and I3=LO and I9=LO Then Output = LO
8	If I1=HI and I7=HI and I3=HI and I9=HI Then Output = LO
9	If I1=LO and I3=LO and I7=HI and I9=HI Then Output = LO
10	If I1=HI and I3=HI and I7=LO and I9=LO Then Output = LO
11	If I2=LO and I8=HI Then Output = LO
12	If I2=HI and I8=LO Then Output = LO
13	If I2=LO and I8=LO Then Output = LO
14	If I2=HI and I8=HI Then Output = LO

Table 6.3 Rules for experiment 6.6.1.

The rules were developed as follows. The wane edge is predominantly vertical, therefore the operation of the rules is analogous to a vertical edge detector. Rules one, two, five and six are used for left and right wane detection. Rules nine to fourteen are used to ignore horizontal lines. Rule seven is used to ignore dark areas (bark) that fill the window and rule eight is used to ignore light areas that fill the window (grain lines).

6.6.1.2 The network structure used for experiment 6.6.1

Layer	No. of nodes
Input layer	9 nodes (3x3 window)
Condition layer	2 membership functions for each input = 18 nodes
Rule layer	14 rules + 11 free = 25 nodes
Action layer	2 nodes
Output layer	1 node

Table 6.4 Network structure for experiment 6.6.1.

Two networks with the above structure was generated and one network was initialized with the above rules and the other network was initialized with random numbers in the range {-1,+1|}. Each network was then trained for 400 epochs. Figure 6.6 shows the images that were generated by processing the input image using the trained FuNNWF. The images are inverted for clarity. Figure 6.6 (a) is the output image generated by the network initialized with rules and image (b) is the output generated by the network initialized with random numbers.

Figure 6.6 Neural network output for experiment 0. Image (a) is the output generated by the network initialized with rules and image (b) is the output generated by the network initialized with random numbers.

The left and right wane edges can be seen in both images. The output image generated by the network initialized with rules is similar to the output image generated by the network initialized with random numbers. Table 6.5 shows the training RMS error for both networks.

	Training RMS error before training	Training RMS error after 400 epochs
Network initialized with rules	0.1138	0.0480
Network initialized with random numbers	0.4621	0.0517

Table 6.5 Training RMS error of FuNN initialized with and without rules before and after training for experiment 0.

Table 6.5 shows that the network initialized with rules has started from a lower RMS error than the network initialized without rules. It has also reached a lower RMS error after 400 epochs. Although the network initialized with rules seems to have performed better in training, the output is almost the same as the output from the network initialized without rules. The rules did not have a noticeable effect on network training.

Several other experiments were performed using the same network initialized with different rules representing the wane edge detection problem. The resulting output image from all those experiments were similar to image (a) in Figure 6.6. Several runs where the network was initialized with random numbers resulted in a blank output image. The network failed to detect the wane edges using a 3x3 window, so several new experiments were performed using a larger window.

6.6.2 Experiments to compare the effectiveness of wane edge detection for a range of window sizes

The previous experiment investigated the performance of a 3x3 NNWF on the wane edge detection problem. The 3x3 window failed to detect the wane edge effectively. The experiments described in this section investigate the effects of using larger window filters to detect the wane edges. Rectangular windows of dimension 3x5, 3x7 and 3x9 were used. In the previous experiment the operation of the rules was analogous to a vertical edge detector. The rules used in this experiment are an extension to those used in the previous experiment. These rules are described in high level terms and are independent of window filter size.

6.6.2.1 The verbal rules

Rule	High level description
1	If the intensity of the left side of the window is low and the intensity of the right side of the window is high then output is high (rule designed to detect left wane edge).
2	If the intensity of the left side of the window is high and the intensity of the right side of the window is low the output is high (rule designed to detect

	right wane edge).
3	If the intensity of the whole window is low then output is low (rule designed to detect bark).
4	If the intensity of the whole window is high the output is low (rule designed to detect useable area of wood).
5	If intensity of the top of the window is low and the intensity of the bottom of the window is high then the output is low (rule designed to detect horizontal edges).
6	If the intensity of the top of the window is high and the intensity of the bottom of the window is low then the output is low (rule designed to detect horizontal edges).
7	If the intensity of the left and right side of the window is high and the intensity of the center of the window is low then the output is low (rule designed to detect grain lines).
8	If the intensity of the left and right side of the window is low and the intensity of the center of the window is high then the output is low (rule designed to detect grain lines).

Table 6.6 High level rules for experiment 6.6.2.

The above rules are described in high level terms. These high level rules were used to generate low-level rules that were used to initialize the FuNN. Three networks were generated for the 3x5, 3x7 and 3x9 window sizes. Each network was initialized with the rules and trained for 400 epochs. The training performance of each network was compared to the training performance of a network initialized with random numbers in the range {-1, +1}.

Table 6.7 shows a summary of the results from these experiments. Table 6.7 shows that the network starts training from a lower RMS error when the window sizes are 3x5 and 3x7 when compared to a network initialized with random numbers. The table also shows that the RMS error at epoch 400 is lower for window sizes 3x7 and 3x9 for the network initialized with rules.

Several other runs were performed to extend the experiments reported in Table 6.2; the number of free nodes was varied in these runs and the results were similar to those in Table 6.5.

Window filter size	Rule layer structure	Training RMS error at 0 epochs	Training RMS error after 400 epochs	Comments on output image generated by trained network
3x5	9 rules + 16 free nodes (total of 25 nodes)	0.541	0.033	The output image was similar to (a) in Figure 6.6. The left and right wane edge was not detected.
	25 free nodes (0 rules)	0.699	0.032	The output image was similar to (b) in Figure 6.6.
3x7	10 rules + 30 free nodes (Total of 40 nodes)	0.474	0.027	The output image was similar to (a) in Figure 6.6. The left and right wane edge was not detected.
	40 free nodes (0 rules)	0.859	0.033	The output image was similar to (b) in Figure 6.6.
3x9	8 rules + 46 free nodes (Total of 54 nodes)	0.289	0.039	The output image was similar to (a) in Figure 6.6; the right wane edge was defined more clearly.
	54 free nodes (0 rules)	0.284	0.051	The network generated a blank image

Table 6.7 Results of experiments when using window filters with different sizes.

For some runs the network detected only one wane edge. Initially the detection of one wane edge was considered to be a poor result as the defined task was to detect both wane edges at the same time. The experiments performed later in this chapter describe how this information was useful in solving the wane edge detection problem. The poor performance of the NNWF in the experiments so far described, led to the speculation that this was a result of the network having insufficient capability to discriminate between plank features.

6.6.2.2 Rules for different features

Sub Image	Name
	Marker
	Grain lines
	Knot
	Knot with markers
	Left and right wane and marker
	Left wane
	Marker
	Right wane

Table 6.8 The different feature of the plank that were used to generate rules for the experiments of the wane edge detection problem.

Table 6.8 shows the features on a plank of wood. A new set of rules was generated describing each feature. Some of the sub images in table 6.5 were described using several rules. The experiment 6.6.2.1 was rerun with the new set of rules. For some runs as many as 65 rules describing the plank features were used. The inclusion of more rules did not improve network performance.

6.6.2.3 Rule verification experiment

As the network performed poorly on the wane edge detection problem a rule verification experiment was conducted to determine if the rules that were generated from problem knowledge were still present after training. A rule extraction process was developed to extract rules from the trained network that was similar to the inverse of the rule insertion process.

6.6.2.4 Rule extraction process

The rule extraction process is similar to the inverse of the boolean logic rule insertion process explained in chapter 3. So far only two membership functions have been used in the rule generation process, LO and HI. A brief overview of the rule insertion process is described. When only two membership functions LO and HI are defined,

If I1=HI Then O1=LO can be represented by Figure 6.7.

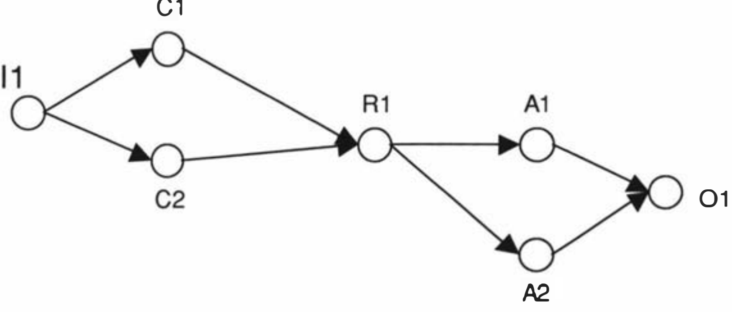


Figure 6.7 Representation of If I1=HI Then O1=LO.

If I1C1 represents LO and I1C2 represents HI then for I1 to be HI, connection C1R1 should be negative and connection C2R1 should be positive. A similar

principle can be applied to rule extraction. For rule extraction the weights associated with the node should have opposite signs. If they have the same sign a rule cannot be extracted. If C1R1 is negative and C2R1 is positive then I1=HI can be extracted. The same principle can be applied to the action layer. This method of rule extracted is not perfect but it was sufficient for the rule verification experiment.

Table 6.9 shows the sub-images used to train the network to determine if the rules changed significantly after training. A FuNN was initialized using the rules associated with the input-target sub-images. After training, the network weights were used to extract rules.

Input sub-image	Target sub-image	Associated rule
		Left wane edge detection rules
		Right wane edge detection rules
		Grain line detection rules
		Left wane + knot detection rules
		Knot + marker detection rules

Table 6.9 Sub images used to train the network to determine if the rules were present in the weights after training.

The extracted rules were compared with the original rules. The extracted rules were the same as the original rules except for the last pair of sub-images in the table. The rules for the knot section were slightly modified. As expected the rules did not change significantly confirming that the rules represented problem knowledge.

6.6.2.5 Rethinking the problem

The network failed in the task of detecting the left and right wane edge at the same time. In some experiments the network detected the left wane edge and did not detect the right wane edge and vise versa. When a large number of rules were used (number of rules > 35) the training time increased exponentially without a significant improvement in the network performance. Since the network failed to detect both wane edges at the same time, an experiment was performed with the aim of detecting only the left or right wane edge.

6.6.2.6 The window size

A program was developed that extracted a sub image of a predetermined size from the original wane image. A user who has not seen the plank image was asked to determine if the sub-image was a wane edge. Global information was not available, as the plank image was not shown to the user in advance. If the user can determine if the sub-image contains a wane edge, then the size of the sub-image is sufficient for a machine to perform the task. The user could not identify the wane edge when smaller window sizes were used. More accurate identification of the wane edge was achieved as the window size was increased. When a 9x9 window size was used the user was able to identify sub-images containing wane edges but misclassified sub-images containing knot edges as wane edges. The average width of a knot is 23 pixels. When the window size was increased to 9x23, misclassification of knot edges reduced. From this test it was observed that a 9x23 and 9x9 window may be used to detect the left or right wane edge. The experiments described below were performed using a 9x23 and a 9x9 NNWF.

6.6.3 Experiment to determine if the network could detect one wane edge only using a 9x23 window filter.

This experiment was performed to determine if a 9x23 window filter could detect one wane edge. As described in section 6.6.2.5 the problem was divided into two parts to detect the left and right wane edges individually. A new target image

containing only the left wane edge was generated. The input and new target images were used to generate a new training set to train the network. The membership functions in Table 6.1 were used to generate the rules.

6.6.3.1 The rules expressed in high-level terms for experiment 6.6.3

Rule	Description
1	If the intensity of the left side of the window is low and the intensity of the right side of the window is high then output is high (rule designed for left wane edge detection).
2	If the intensity of the top half of the window is low and the intensity of the bottom half of the window is high then the output is low (rule designed to detect horizontal lines).
3	If the intensity of the top half of the window is high and the intensity of the bottom half of the window is low then the output is low (rule designed to detect horizontal lines).
4	If the intensity of the entire window is low then the output is low (rule designed to detect bark area of plank).
5	If the intensity of the entire window is high then the output is low (rule designed to detect useable area of plank).
6	If the intensity of most of the window is low then output is low (another rule designed to detect bark areas of plank).
7	If the intensity of most of the window is high then output is low (another rule designed to detect useable area of plank).

Table 6.10 Rules for experiment 6.6.3.

Each high-level rule expressed above represents a feature on the plank. Several low-level rules can be generated from each high level rule. For this experiment 29 low-level rules were generated. A saturation level of three was used for the rules. The FuNN was initialized using these rules prior to training.

6.6.3.2 The network structure used for experiment 6.6.3

Layer	No. of nodes
Input layer	207 nodes (9x23 window)
Condition layer	2 membership functions for each input = 414 nodes
Rule layer	29 rules + 35 free = 65 nodes
Action layer	2 nodes
Output layer	1 node

Table 6.11 Network structure for experiment 6.6.3.

Small random numbers in the range {-0.1, +0.1} were added to the weights of the network prior to training to overcome symmetry problems. The network was then trained for 200 epochs. Figure 6.8 shows the output image generated by processing the input image with the trained network.

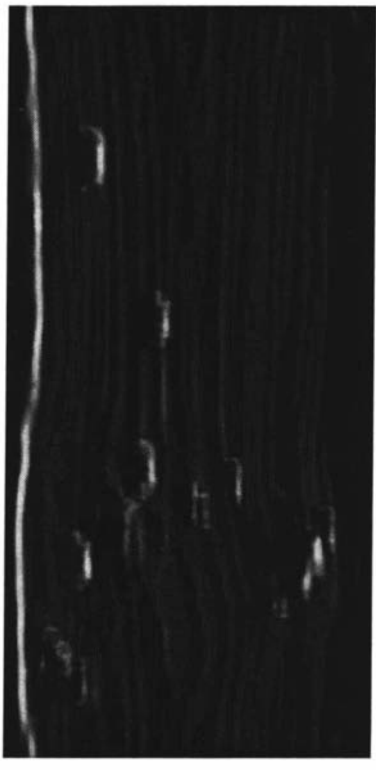


Figure 6.8 Network output for experiment 6.6.3

The network has detected the left wane edge and the right edges of knots and other defects on the plank. Table 6.12 shows the error between the output and target image.

Standard deviation	1.1629
False positives	220
False negative	0
Error	46.33

Table 6.12 Error figure for experiment 6.6.3.

The network trained without rules generated a blank image. The experiment shows that the network initialized with problem knowledge was able to detect the left wane edge after the problem was broken down into two components.

6.6.3.3 Post-processing the network output

The post process routine outlined below was designed through an iterative process. The output image generated by the trained FuNN was post processed to generate the final output image. Figure 6.9 (a), (b) and (c) show the post process steps.

1. Figure 6.9 (a) was generated by thresholding the output image generated by the trained FuNN.

   The threshold value t was calculated as follows,

   $t = m + (255-m)*15/100$

   where,

   $$m = \frac{\sum_{1}^{I}\sum_{1}^{J} S(i,j)}{I \times J}$$

   and S(i,j) is the intensity value of the pixel at (i,j) in image S (output image). The thresholded image is in binary format. This formula was derived through an iterative process.
2. Figure 6.9 (b) was generated as follows,

   Figure 6.9 (a) was processed using a thinning procedure. The center pixel for each high intensity section was used for this process.
3. Figure 6.9 (c) was generated as follows,

   Figure 6.9 (b) was scanned from left to right. The first high intensity pixel encountered is considered to belong to the left wane edge. The entire image is scanned to find the complete left wane edge.

❖ The right wane edge can be found by modifying step 3 to scan from right to left and detect the first high intensity pixel from the right.

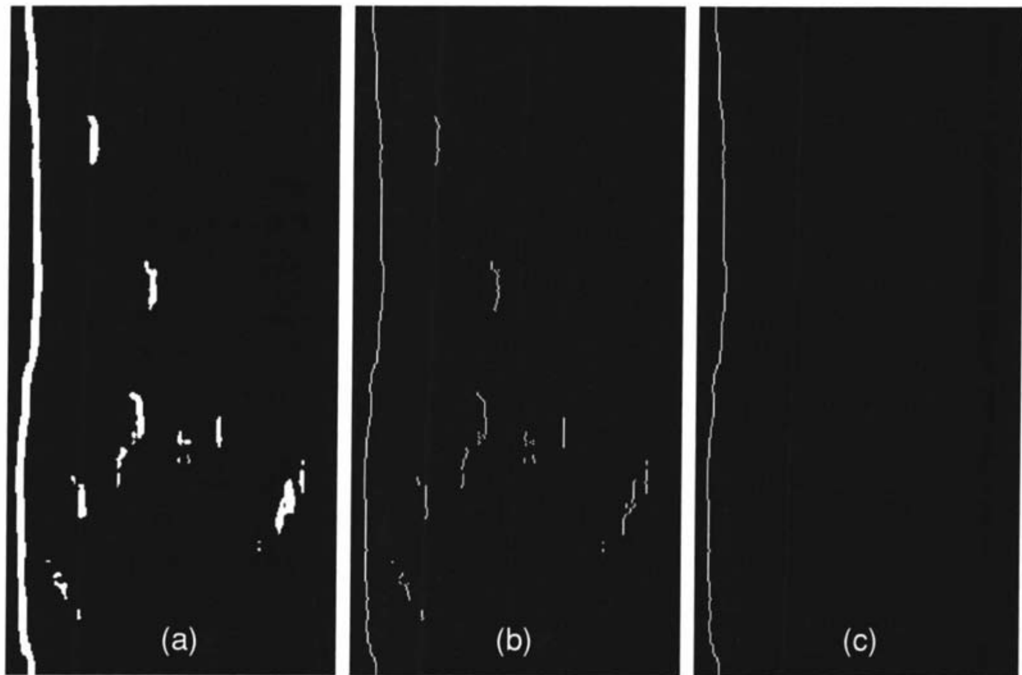


Figure 6.9 The post process procedure, image (a) generated by thresholding the network output, image (b) generated by thinning the thresholded output and image (c) generated by scanning the thinned image from left to right to detect the first occurrence of a high intensity pixel.

6.6.4 Use of a 9x9 window filter to detect the left wane edge.

Experiment 6.6.3 used a 9x23 window filter to successfully detect the left wane edge. Due to the size of the network training time was extensive. This experiment was performed to determine if a 9x9 window filter could detect the left wane edge, thus reducing training time. High level rules 1-7 in section 6.6.3 were used to generate 34 low-level rules.

6.6.4.1 The network structure used for experiment 6.6.4

Layer	Description
Input layer	81 nodes (9x9 window)
Condition layer	2 membership functions for each node = 162 nodes
Rule layer	34 rules + 1 free = 35 nodes

Action layer	2 nodes
Output layer	1 node

Table 6.13 Network structure for experiment 6.6.4.

The network initialized with rules was then trained for 200 epochs. The trained network was used to process the input image.

Figure 6.10 shows the network output after training for 200 epochs. The error between the target and the output image was 96.3.

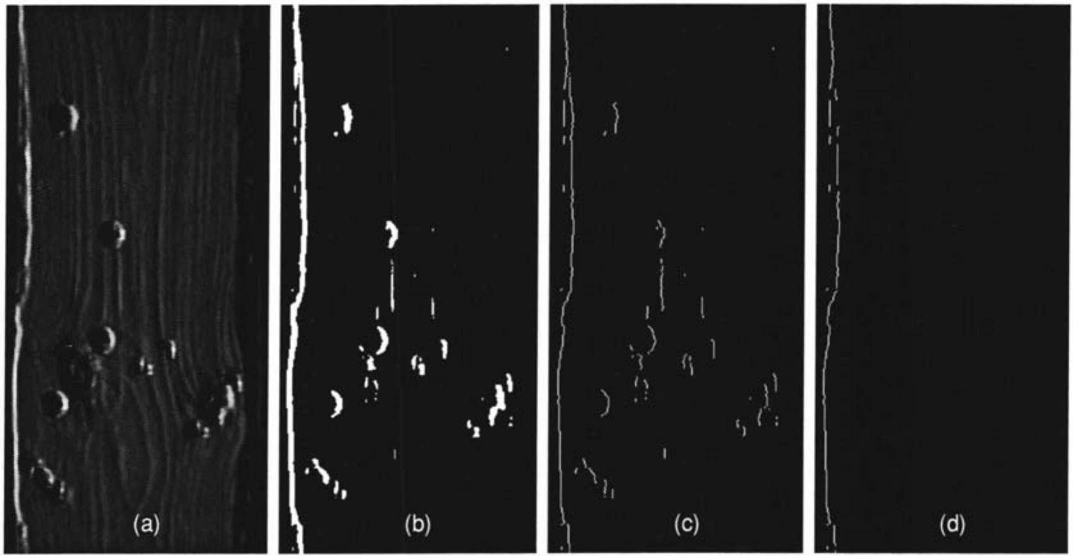


Figure 6.10 Results of training the network for 200 epochs for experiment 6.6.4 (a) network output (b) network output thresholded (c) thresholded image thinned (d) thinned image scanned for left wane edge.

The output image generated has a high number of false positives. The network was trained for an additional 600 epochs to determine if the error could be reduced. Figure 6.11 shows the output of the network.

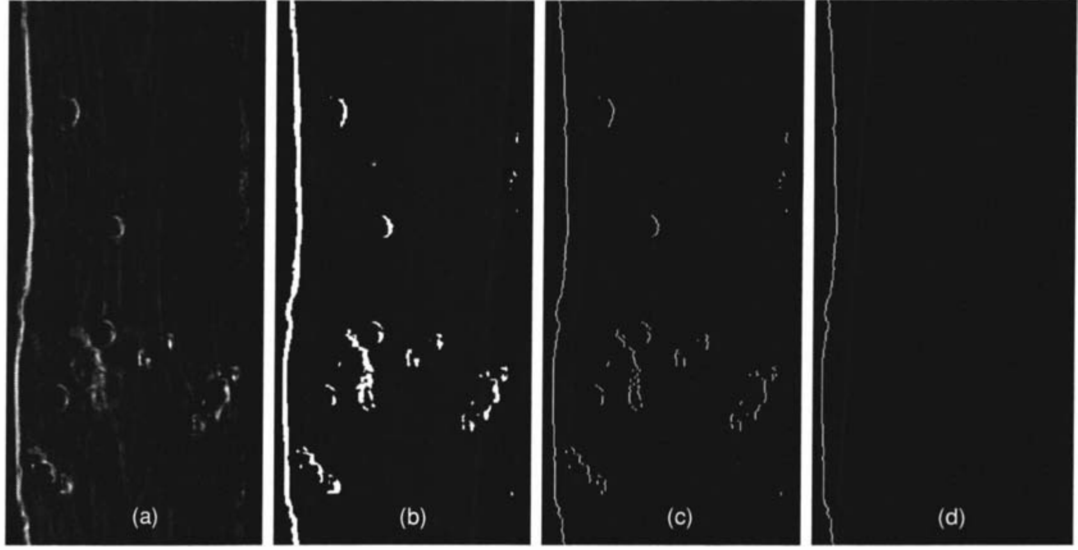


Figure 6.11 Results after training t network for 800 epochs for experiment 6.6.4 (a) output image (b) output image thresholded (c) thresholded image thinned (d) thinned image scanned for left wane edge.

Additional training reduced the error between the target and output image to 74.7. The number of false positives also reduced to 363. For comparison purposes the same network was initialized with random numbers in the range {-1, +1} and trained for the same number of epochs. The network initialized with random numbers generated a blank image. As the experiment shows a 9x9 FuNN initialized with rules detected the left wane edge.

6.6.5 Network training experiment that involves changing the number of rules and training times.

This experiment was performed to determine how the error between the target and output image changed as the number of rules and training time was varied. The number of rules was varied between 0 and 30 (0, 5, 10, 15, 20, 25, 30). The training time was varied between 0 and 400 (0, 50, 100, 200, 400) epochs.

6.6.5.1 The network structure used for experiment 6.6.5

Layer	Description
Input layer	81 nodes (9x9 window)
Condition layer	2 membership functions for each node = 162 nodes
Rule layer	35 nodes. As the number of rules increased the number of free nodes decreased maintaining a total of 35 nodes for the rule layer. If five rules were used then the number of free nodes was 30. If 10 rules were used then the number of free nodes was 20, etc.
Action Layer	2 nodes
Output layer	1 node

Table 6.14 Network structure for experiment 6.6.5.

As described above a 81-162-35-2-1 network structure was used for this experiment. A saturation value of 1.5 was used for the rules. The network was trained and the error between the target and output image was recorded. Table 6.15 shows these errors. The table shows that the error was smallest when the network was trained for 100 epochs using 15 rules. When a blank image is generated the error between the target and the output image is over $2x10^4$. So the number 999 was used instead of values over $2x10^4$ to represent blank images.

Epochs	Number of rules used						
	0 rules	5 rules	10 rules	15 rules	20 rules	25 rules	30 rules
0	178	999	999	999	999	999	999
50	999	999	999	999	999	999	999
100	549	159	110	48	54	50	999
200	999	78	82	82	83	77	74
400	999	154	127	156	112	72	69

Table 6.15 Error between target and output image for experiment 6.6.5.

Figure 6.12 shows the output images generated by each network.

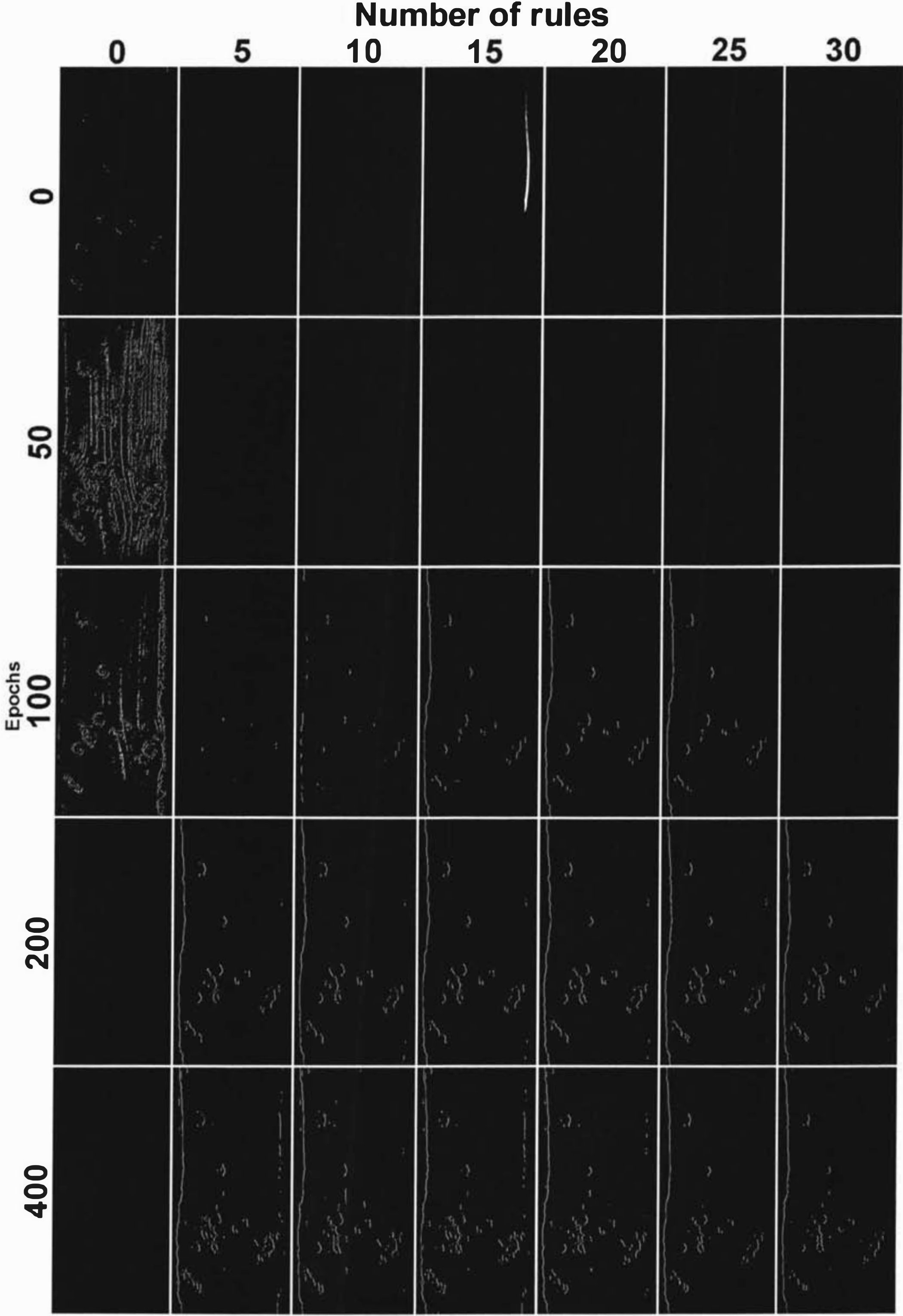


Figure 6.12 Output images for experiment 6.6.5.

As training progresses beyond 200 epochs the error between the output and target image increased. As Table 6.15 shows it is sufficient to train the network between 100 and 200 epochs. Figure 6.12 shows that a network initialized with a minimum of five rules and trained for 200 epochs could detect the wane edge. This led to the

speculation that a network with a smaller number of rules and free nodes could detect the wane edge.

6.6.6 Experiment to determine if a FuNN using a 9x9 window filter with a smaller rule set could detect the left wane edge.

As described in experiment 6.6.5 the network detected the wane edge with a minimum of five rules. This experiment was performed to determine if the left wane edge could be detected using a smaller rule set. Three runs were performed with smaller rule sets. The high level rules in section 6.6.3.1 were used to generate the rules that were used to initialize the network. Table 6.16 shows the structure, the number of rules and free nodes used for each network. A saturation value of 1.5 was used for the rules and each network was trained for 200 epochs. The trained network was then used to process the input image to generate the output image. The error between the output and target image is shown in the table below.

Network Structure	Number of rules used	Error
81 rule nodes 162 condition nodes **10 rule nodes** 2 action nodes 1 output node	7 rules + 3 free nodes	Error = 89.54 False positives = 436
81 rule nodes 162 condition nodes **10 rule nodes** 2 action nodes 1 output node	8 rules + 2 free nodes	Error = 83.03 False positives = 404
81 rule nodes 162 condition nodes **12 rule nodes** 2 action nodes 1 output node	8 rules + 4 free nodes	Error = 75.28 False positives = 365

Table 6.16 Network structure, number of rules and free nodes and the error between the target and output image for experiment 6.6.6.

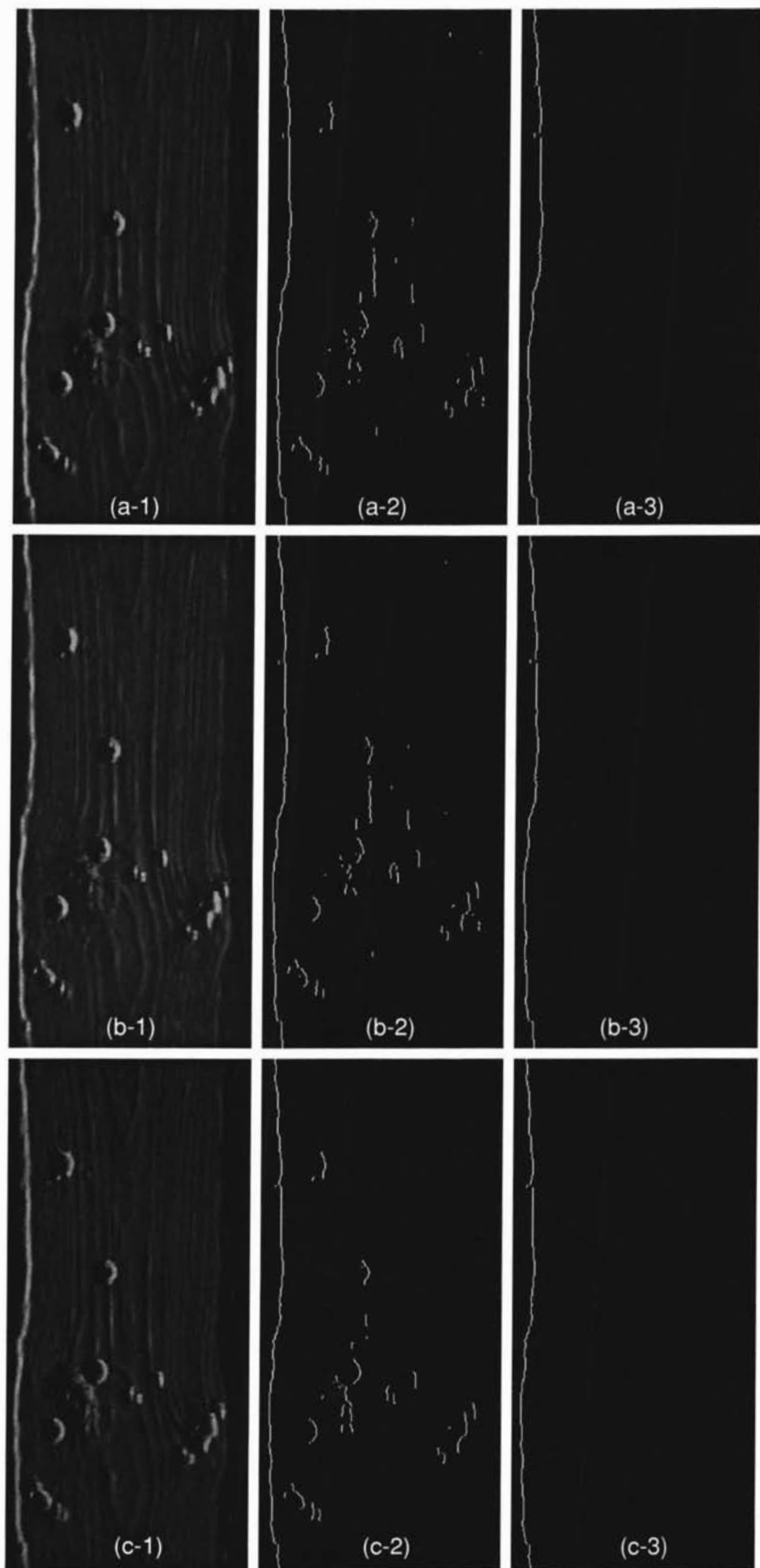


Figure 6.13 Output images for experiment 6.6.6.

Figure 6.13 shows the output images generated by the networks in Table 6.16. Image (a-1) is the output image for network one, image (a-2) is the output image thresholded and thinned and image (a-3) is the thinned image scanned for left wane. Images (b-1), (b-2) and (b-3) are the output images for network two and images (c-1), (c-2) and (c-3) are the output images for network three. When compared with the experiments performed in 6.6.5, the experiments in this section show that the network has detected the left wane edge by using a smaller rule set.

6.6.7 Experiment to detect the left wane edge using conditional rules.

Boolean logic rules were used in the experiments performed so far. This experiment was performed to determine the network performance when conditional rules were used to initialize the network. The high level rules in section 6.6.3.1 were converted to conditional and boolean logic rules.

6.6.7.1 The conditional rules for experiment 6.6.7

Rule	Description
1	If intensity of the left side of the window < intensity of the right side of the window then the output is high.
2	If intensity of the left side of the window > intensity of the right side of the window then the output is low.
3	If intensity of the top half of the window < intensity of the bottom half of the window then the output is low.
4	If intensity of the top half of the window > intensity of the bottom half of the window then the output is low.

Table 6.17 Conditional rules for experiment 6.6.7.

6.6.7.2 The boolean logic rules for experiment 6.6.7

Rule	Description
5	If the intensity of the entire window is low then the output is low.
6	If the intensity of the entire window is high then the output is low.

Table 6.18 Boolean logic rules for experiment 6.6.7.

6.6.7.3 The network structure used for experiment 6.6.7

Layer	No. of nodes
Input layer	81 nodes (9x9 window)
Conditional layer	2 membership functions for each input = 162 nodes
Rule layer	6 rules + 2 free = 8 nodes
Action layer	2 nodes
Output layer	1 node

Table 6.19 Network structure for experiment 6.6.7.

The network was initialized with rules and trained for 100 epochs. Another network with the same structure was initialized with random numbers and trained for 100 epochs. Table 6.20 shows the difference in training performance between the network initialized with and without rules. Table 6.20 also shows the error between the target and output image generated by each network.

	Starting RMS error	RMS error after 100 epochs	Error between target and output image
Network initialized with six rules	0.0850	0.0363	65.45
Network initialized with random numbers	0.2965	0.0321	129.84

Table 6.20 Training errors for experiment 6.6.7.

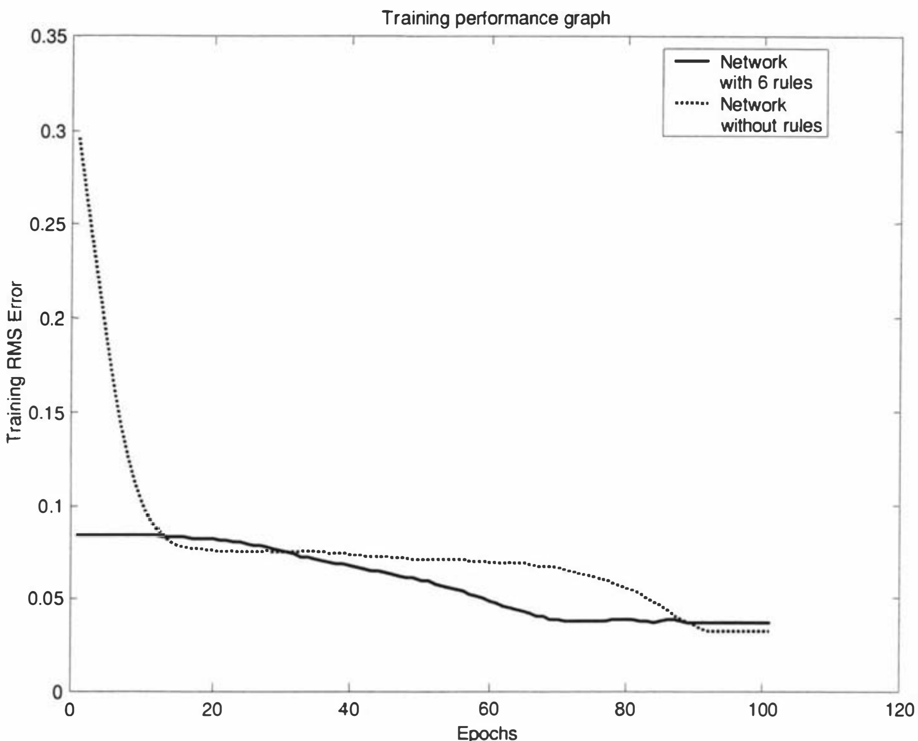


Figure 6.14 Training performance graph for experiment 6.6.7.

Figure 6.14 shows that the network initialized with random numbers has a lower training RMS error after 100 epochs compared to the network initialized with rules. The network initialized with random numbers appears to have converged to a better solution due to the lower training RMS error. In fact the error between the target and output image for the network initialized with rules is lower compared to the network initialized with random numbers. Figure 6.15 shows the output images.

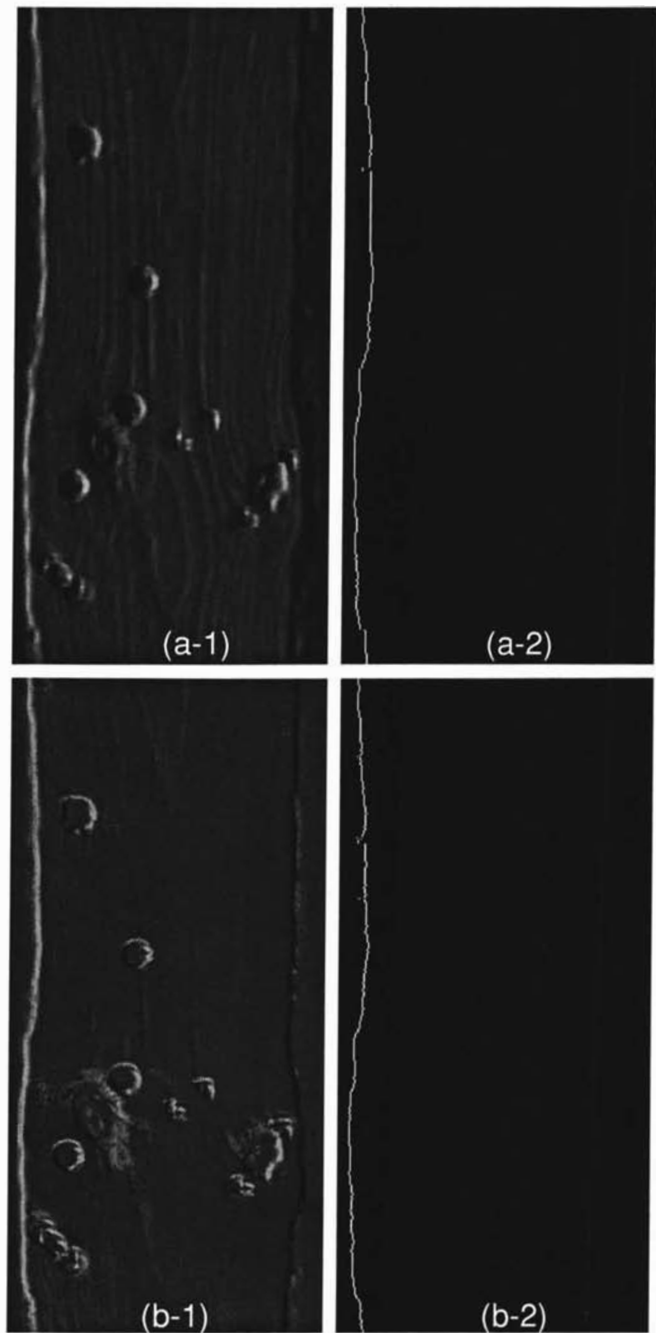


Figure 6.15 Output images for experiment 6.6.7. Image (a-1) output image from network with rules. Image (a-2) generated by post processing (a-1). Image (b-1) output image from network without rules and image (b-2) generated by post processing (b-1).

6.6.8 Experiment to determine the performance of a FuNN seeded with different random numbers. 100 runs were made.

The previous experiment compared the performance between a network initialized with rules and without rules. This experiment was performed to determine the performance of a network trained with different random seedings.

6.6.8.1 Network structure used for experiment 6.6.8

Layer	No. of nodes
Input layer	81 nodes (9x9 window)
Conditional layer	2 membership functions for each input = 162 nodes
Rule layer	15 nodes
Action layer	2 nodes
Output layer	1 node

Table 6.21 Network structure for experiment 6.6.7.

6.6.8.2 Method

The following steps describe how the experiment was performed.

1. Generate network structure.
2. Initialize random number generator so the experiment can be repeated to get the same results if necessary.
3. Obtain a set of random numbers to initialize the weights of the neural network.
4. Train network for 200 epochs.
5. Generate output image by processing the input image with the trained network.
6. Measure error between output and target image.
7. Go back to step 3 until 100 training sessions have been completed.

Another network with the same structure was initialized with rules for comparison purposes. Out of the 100 runs, 20% of the networks generated blank images. The remaining 80% generated an image containing a wane edge.

	Network initialized with rules	Average error of networks initialized with random numbers
False positives	282	375
Error	58.6	77.7

Table 6.22 Comparison of errors between network initialized with and without rules.

Table 6.22 compares the average error of the network initialized with rules against the error of the network initialized without rules. The error between the blank

images and the target image was over $2x10^4$, so those errors were not used in the average error calculation. The number of nodes in the rule layer was increased and the experiment was repeated. It was observed that as the number of nodes in the rule layer increased the percentage of blank images generated by the neural network initialized with random numbers increased. However the network initialized with rules consistently detected the wane edge. Table 6.22 shows that the error of the image generated by the network with rules is lower than the average error of the images generated by the network initialized with random numbers.

6.6.9 Experiment to detect the right wane edge

The wane edge detection problem was separated into two individual parts; one to detect the left wane and the other to detect the right wane. The left wane edge was detected in the experiments performed so far. This experiment was performed to determine if the FuNN could detect the right wane edge.

6.6.9.1 High level rules used for experiment 6.6.9

Rule	Description
1	If the intensity of the left side of the window is light and the intensity of the right side of the window is dark then the output is high (right wane detected).
2	If the intensity of the left side of the window is dark and the intensity of the right side of the window is light then the output is low.
3	If the intensity of the top half of the window is light and the intensity of the bottom half of the window is dark then the output is low (horizontal edges).
4	If the intensity of the top half of the window is dark and the intensity of the bottom half of the window is light then the output is low (horizontal edges).
5	If the intensity of the entire window is low then the output is low(bark).
6	If the intensity of the entire window is high then the output is low (useable area of wood detected).

Table 6.23 Rule for experiment 6.6.9.

The high level rules shown in the table are the opposite or mirror of the rules to detect the left wane edge. Rules 1-4 were converted to low-level conditional rules and rules 5 & 6 were converted to low-level boolean logic rules. A saturation level of one was used for the rules as it was found to be an appropriate value when starting a new experiment. The network was trained for 150 epochs and the input image was processed using the trained network.

Figure 6.16 shows the output image generated by processing the input image using the trained FuNN. The error between the target and output image was 49.06.

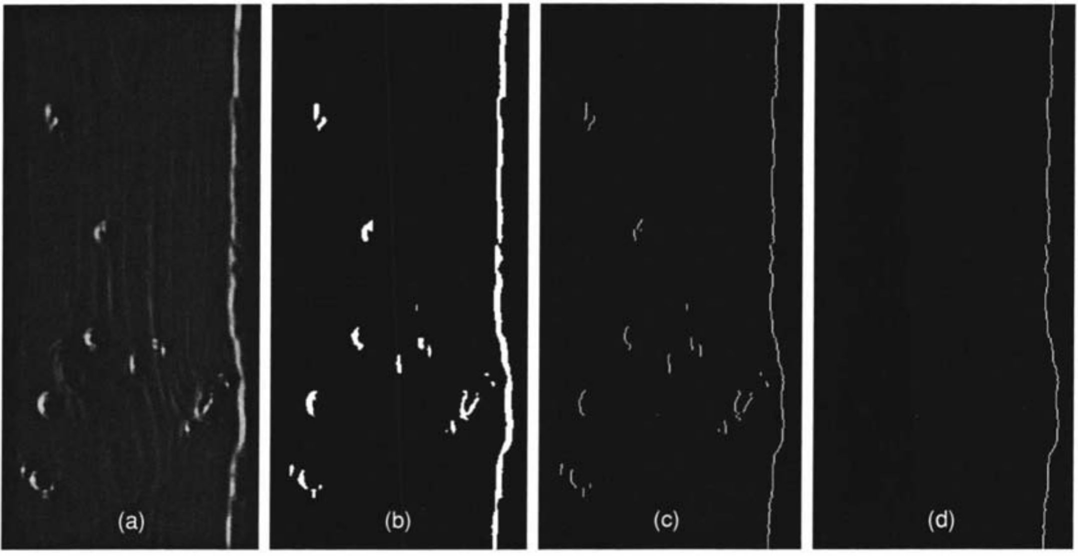


Figure 6.16 The output images for experiment 6.6.9. (a) the output image generated by the network, (b) the output image thresholded, (c) thresholded image thinned, (d) thinned image scanned for right wane edge.

6.6.10 Generating the final image containing both left and right wane edges

The results of the previous experiments showed how one network could be trained to detect the left wane edge and another could be trained to detect the right wane edge. The output images generated from these two networks could then be combined to generate an image containing both wane edges. Figure 6.17 (a) shows the result of combining the output images from the two networks. The thresholded and thinned output images from each network were combined using an OR operation to generate the final image containing both the left and right wane edge. Figure 6.17 (b) shows an image containing the left and right wane edge superimposed on the original plank image.

In a commercial environment one network could be trained to detect the left wane edge and another could be trained to detect the right wane edge. Both networks could run in parallel to detect the left and right wane edges simultaneously. A machine could then use the detected wane edges to remove the bark from the plank.

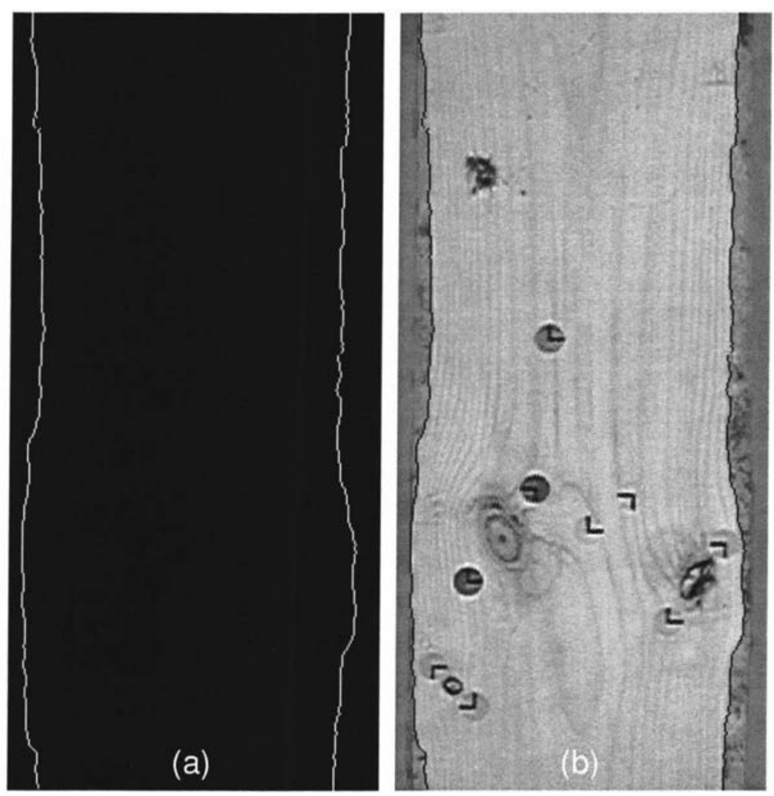


Figure 6.17 (a) shows the left and right wane combined, (b) shows the combined wanes superimposed on the input image.

6.6.11 Using the trained network to detect the wane edges on other planks of wood

In this experiment the algorithm was evaluated on images not used in training. The networks trained to detect the left and right wane edge were used to detect the wane edges of planks that were not used in the training set. Each image generated by each network was processed using the post process routine described in section 6.6.3.3 and then the images were combined using the process described in section 6.6.10.

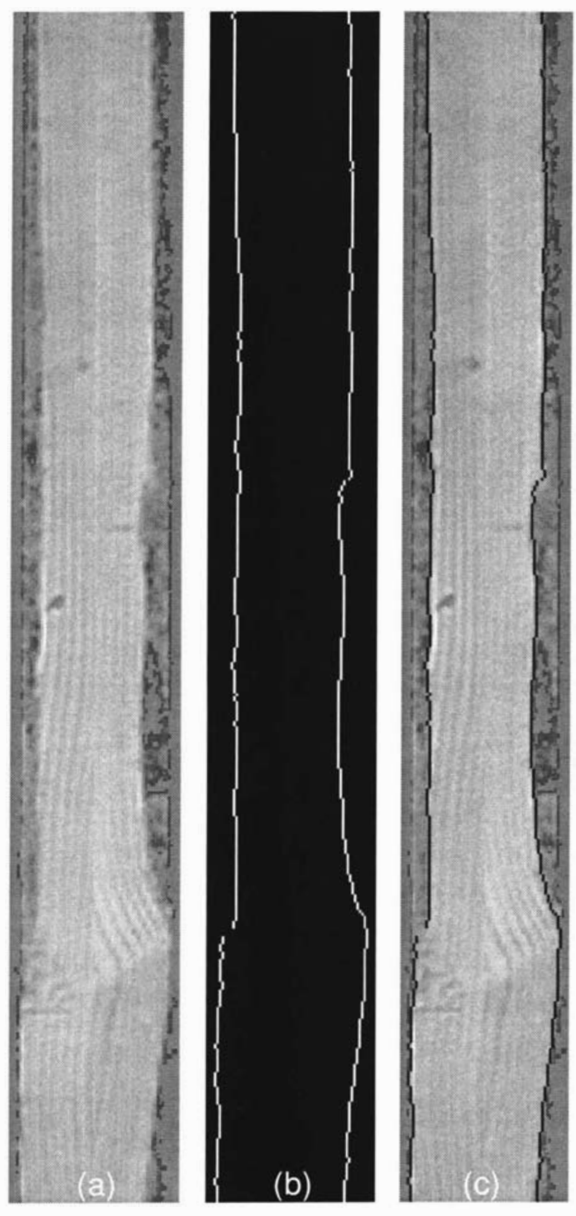


Figure 6.18 Plank 1 used to test the trained FuNN. (a) input image, (b) left and right wane detected using trained FuNN, (c) detected left and right wane superimposed on the input image.

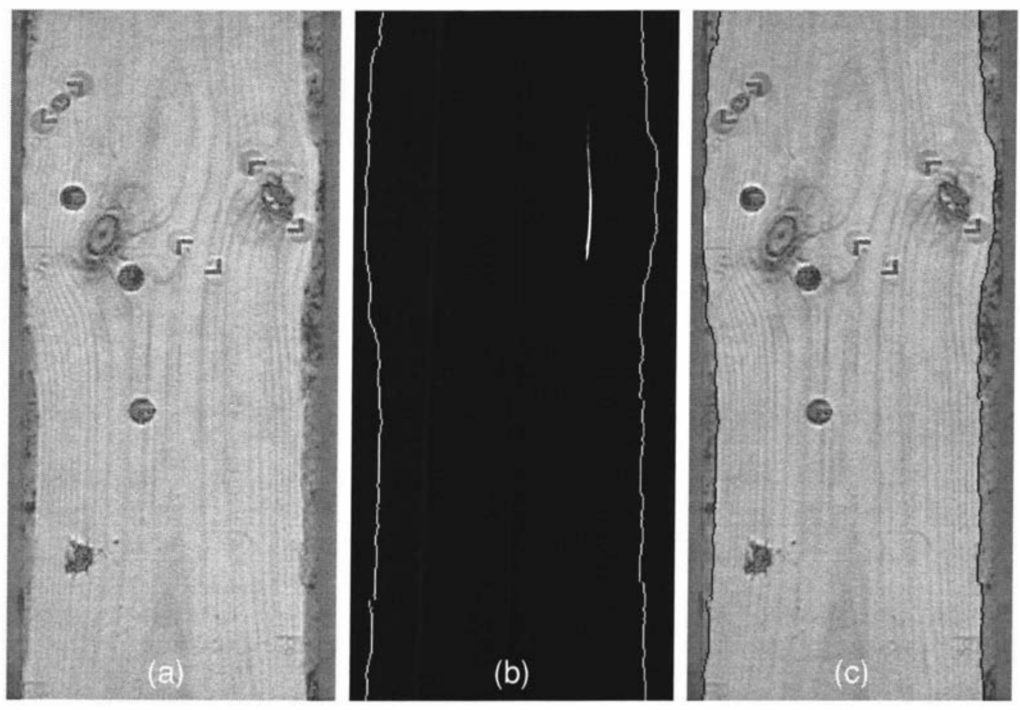


Figure 6.19 Plank 2 used to test the trained FuNN. (a) input image, (b) left and right wane detected using trained FuNN, (c) detected left and right wane superimposed on the input image.

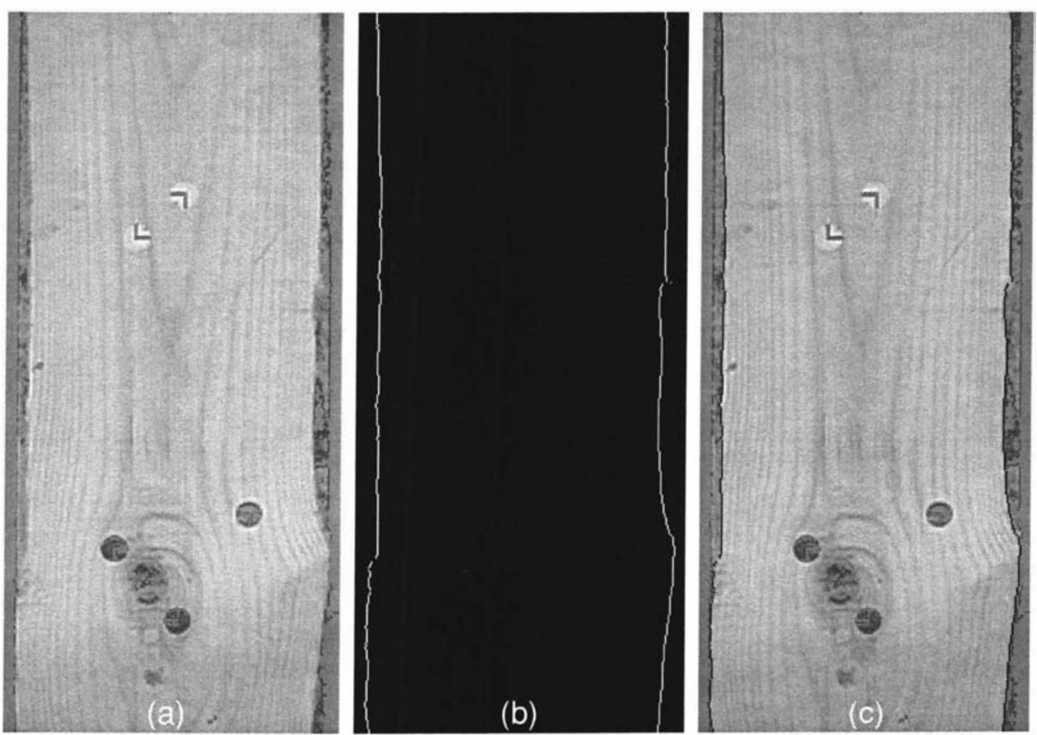


Figure 6.20 Plank 3 used to test the trained FuNN. (a) input image, (b) left and right wane detected using trained FuNN, (c) detected left and right wane superimposed on the input image.

The above figures show the left and right wane detected by the FuNN.

6.6.12 Experiment to test alternative training algorithms

The steepest descent algorithm with momentum and adaptive learning rate was used for neural network training for all of the experiments performed so far. This experiment was performed to determine the training performance of the network for alternative training algorithms. Table 6.24 shows the different training algorithms used.

Training algorithm	Description
Traingdx	Gradient descent with momentum and adaptive training rate†.
Trainrp	Resilient backpropagation. In this algorithm only the sign of the derivative is used to determine the direction of the weight update, the magnitude of the derivative has no effect on the weight update. The size of the update value is determined by a separate update value†.
Traincgf	Fletcher-Reeves conjugate gradient algorithm. All of the conjugate gradient algorithms start out by searching in the steepest descent direction (negative of the gradient) on the first iteration. $p_0 = -g_0$ A line search is then performed to determine the optimal distance to move along the current search direction. $x_{k+1} = x_k + \alpha_k p_k$ Then the next search direction is determined so that it is conjugate to previous search directions. The general procedure for determining the new search direction is to combine the new steepest descent direction with the previous search direction. $p_k = -g_k + \beta_k p_{k-1}$ The various versions of conjugate gradient are distinguished by the manner in which the constant is computed. For the Fletcher-Reeves update the procedure is $\beta_k = \frac{g_k^T g_k}{g_{k-1}^T g_{k-1}}$ This is the ratio of the norm squared of the current gradient to the norm squared of the previous gradient†.
Traincgp	Another version of the conjugate gradient algorithm was proposed by Polak and Ribiére. As with the Fletcher-Reeves algorithm, the search direction at each iteration is determined by

† Description of training algorithms from Matlab Neural Network Toolbox version 3 help file.

	$p_k = -g_k + \beta_k p_{k-1}$ For the Polak-Ribiére update †, the constant β_k is computed by $\beta_k = \frac{\Delta g_{k-1}^T g_k}{g_{k-1}^T g_{k-1}}$
Traincgb	Powell-Beale conjugate gradient algorithm. For all conjugate gradient algorithms, the search direction will be periodically reset to the negative of the gradient. The standard reset point occurs when the number of iterations is equal to the number of network parameters (weights and biases), but there are other reset methods which can improve the efficiency of training. For this technique the search direction will restart if there is very little orthogonality left between the current gradient and the previous gradient. This is tested with the following inequality: $\left\|g_{k-1}^T g_k\right\| \geq 0.2\left\|g_k\right\|^2$ If this condition is satisfied, the search direction is reset to the negative of the gradient †.
Trainscg	Scaled conjugate gradient algorithm. Each of the conjugate gradient algorithms described previously require a line search at each iteration. This line search is computationally expensive, since it requires that the network response to all training inputs be computed several times for each search. The scaled conjugate gradient algorithm was designed to avoid the time consuming line search. This algorithm is too complex to explain in a few lines, but the basic idea is to combine the model-trust region approach, which is used in the Levenberg-Marquardt algorithm †.

Table 6.24 The different training algorithms used for experiment 6.6.12.

Further information on the training algorithms can be found in [Demuth 1998].

6.6.12.1 Network structure used for experiment 6.6.8

Layer	No. of nodes
Input layer	81 nodes (9x9 window)
Conditional layer	2 membership functions for each input = 162 nodes
Rule layer	4 nodes
Action layer	2 nodes
Output layer	1 node

Table 6.25 Network structure for experiment 6.6.8.

As described above an 81-162-4-2-1 network structure was used for this experiment. Two networks having the above structure were used for this experiment. One network was initialized with 3 rules and the other was initialized with random numbers. Each network was trained for 150 epochs using the training algorithms described in Table 6.24. Every ten epochs during training the network was stopped to generate an output image by processing the input image. The error between the output and target image was recorded for each stop.

Epoch	Traingdx	Trainrp	Traincgf	Traincgp	Traincgb	Trainscg
5	999	999	167	60	148	999
10	999	999	69	87	112	64
20	999	999	76	76	72	74
30	999	70	71	86	77	107
40	145	112	70	89	70	119
50	165	57	75	74	75	85
60	160	60	78	84	82	90
70	172	891	75	94	85	74
80	168	771	75	89	89	78
90	130	435	80	94	90	80
100	107	435	81	100	92	85
110	82	435	91	105	93	88
120	67	435	100	109	96	84
130	58	435	105	111	98	104
140	58	435	104	114	101	99
150	58	435	105	114	103	99

Table 6.26 Error between output and target image generated by network initialized with rules and trained with a range of training algorithms.

Epoch	Traingdx	Trainrp	Traincgf	Traincgp	Traincgb	Trainscg
5	999	999	638	338	719	999
10	999	155	109	79	65	999
20	999	60	74	108	73	64
30	999	58	70	66	72	128
40	999	73	79	104	76	83
50	999	69	83	97	85	76
60	999	74	89	96	88	76
70	999	74	98	98	141	74
80	999	74	102	124	146	74
90	999	74	114	139	160	87
100	999	74	102	138	161	89
110	999	74	117	133	162	95
120	999	74	108	136	176	93
130	999	74	113	152	173	91
140	999	74	117	151	177	91
150	999	74	107	154	177	91

Table 6.27 Error between output and target image generated by network initialized with random numbers and trained with a range of training algorithms.

Table 6.26 shows the error between the target and output image generated by the network initialized with rules and train using the training algorithms described in Table 6.24. Table 6.27 shows the error between the target and output image generated by the network initialized with random numbers and trained using the training algorithms described in Table 6.24. Table 6.26 shows that error between the output and target image was the lowest (58) when the network was trained for 130 epochs using the traingdx algorithm. Table 6.27 shows that the same network initialized with random numbers and trained with the same algorithm generated a blank output image resulting in an error of 999. Table 6.26 shows that the network detected the left wane edge after training for only five epochs when the traincgp algorithm was used. Figure 6.21 shows the error between the output and target image during training for a network initialized with rules. The error for training algorithms traincgp, traincgb and trainscg is similar to traincgf so for clarity they are not shown on the graph. All of the training algorithms except trainrp converged to a useable solution in a lower number of epochs compared to traingdxs.

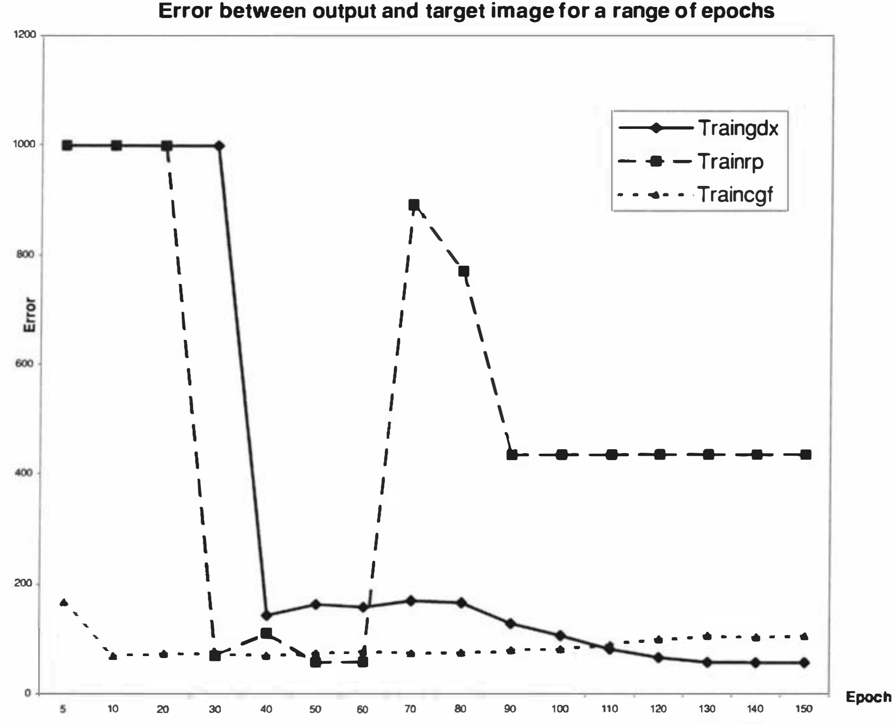


Figure 6.21 Graph showing the error between the output and target image during training for the network initialized with rules.

Figure 6.22 shows the error between the output and target image during training for a network initialized with random numbers. The error for training algorithms traincgp, traincgb and trainscg is similar to traincgf so for clarity they are not shown on the graph.

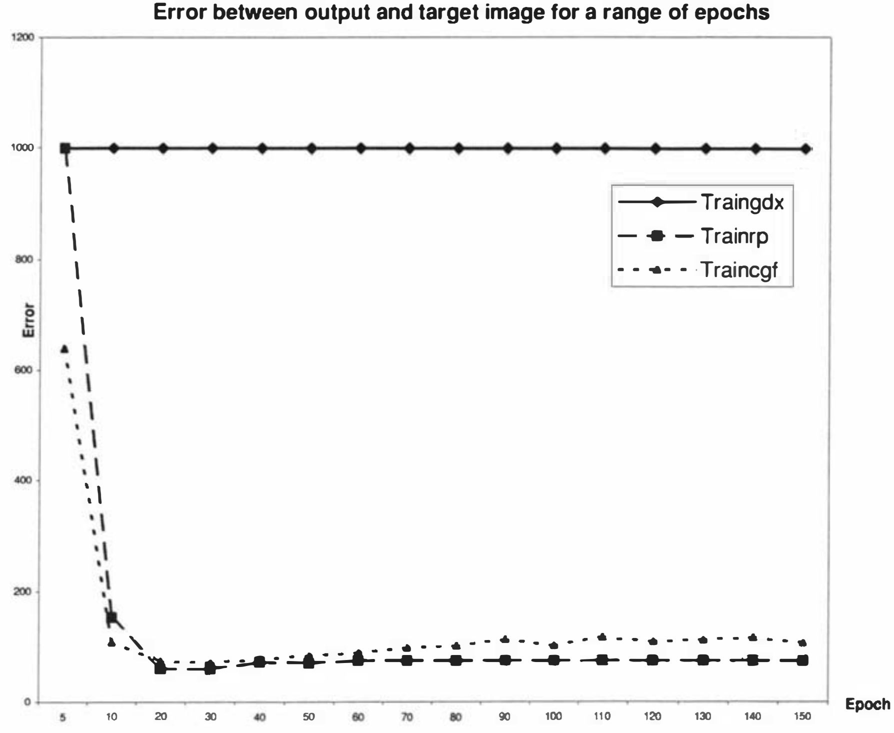


Figure 6.22 Graph showing the error between the output and target image during training for the network initialized with random numbers.

6.6.13 Experiment to determine the quality factor for the rules

This experiment was performed to determine the training performance of the network when the quality factor, saturation value and the number of free nodes was varied. The quality factor was varied between 0.25 and 1 (0.25, 0.5, 0.75, 1). The saturation value was varied between 0.001 and 4 (0.001, 0.0025, 0.005, 0.0075, 0.01, 0.025, 0.05, 0.075, 0.1, 0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75, 2, 2.25, 2.5, 2.75, 3, 4). The number of rules used in this experiment was kept constant and the number of free nodes was varied between 1 and 4 (1, 2, 4). A total of six rules were used for this experiment. A total of 198 runs were performed by varying the saturation value, quality factor and the number of free nodes. Each network was trained until

a training RMS error of 0.035 was achieved and the number of epochs required was recorded.

Figure 6.23 shows the network performance when the network was trained using six rules and one free node; the quality factor for the rules was varied between 0.25 and 0.75 (0.25, 0.5, 0.75) and the saturation value was varied between 0.001 and 4. Figure 6.24 shows the network performance when the network was trained using six rules and two free nodes; the quality factor for the rules was varied between 0.25 and 0.75 (0.25, 0.5, 0.75) and the saturation value was varied between 0.001 and 4. Figure 6.25 shows the network performance when the network was trained using six rules and four free nodes; the quality factor for the rules was varied between 0.25 and 0.75 (0.25, 0.5, 0.75) and the saturation value was varied between 0.001 and 4. In the figures, 6R + 1F denotes the use of 6 rules and 1 free node where R is the number of rules and F is the number of free nodes.

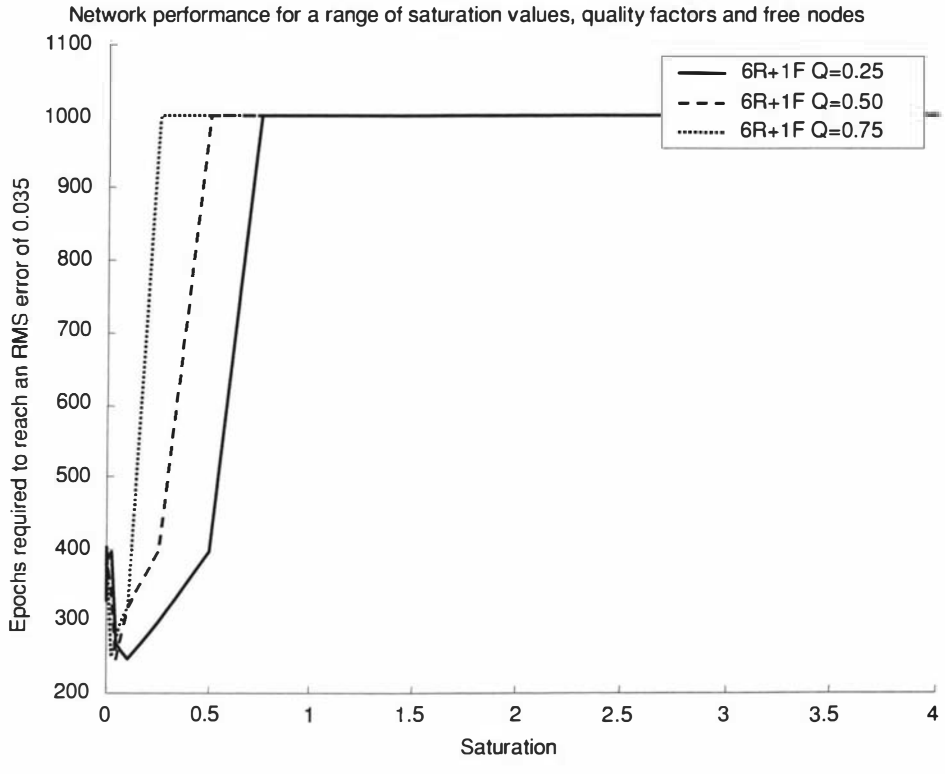


Figure 6.23 Network performance for experiment 6.6.13 when 6 rules and 1 free node was used.

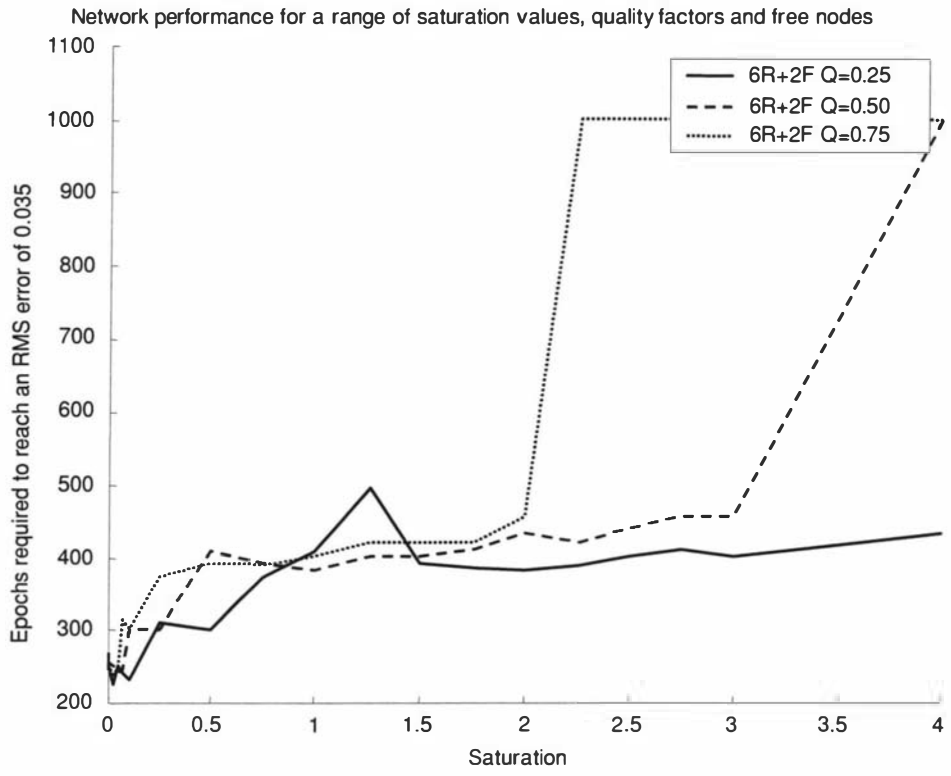


Figure 6.24 Network performance for experiment 6.6.13 when 6 rules and 2 free nodes were used.

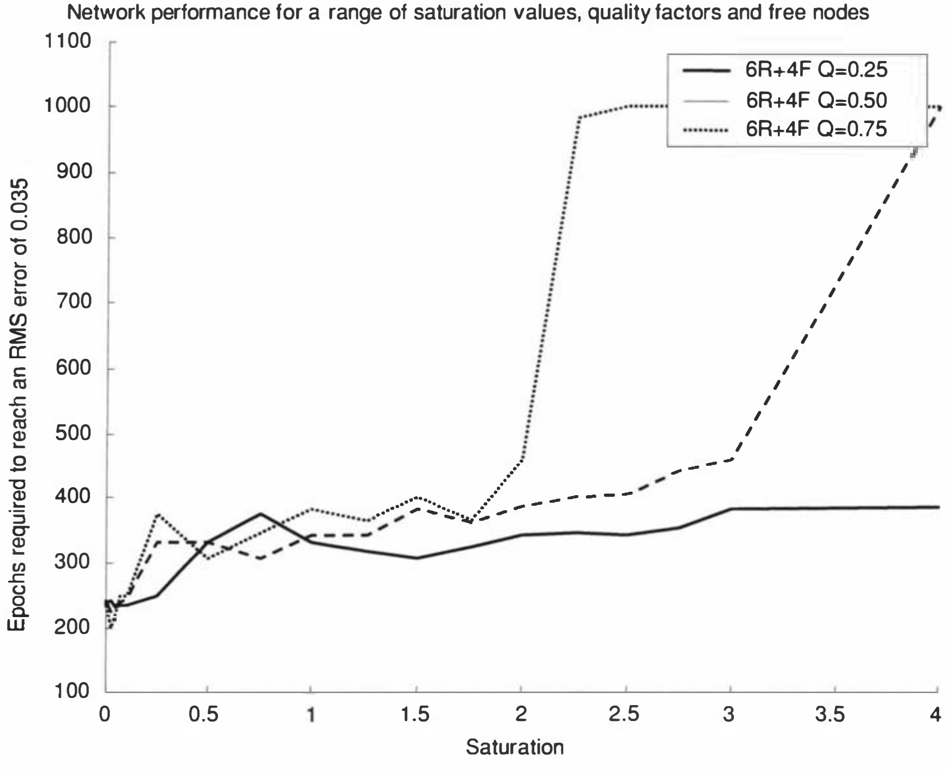


Figure 6.25 Network performance for experiment 6.6.13 when 6 rules and 4 free nodes were used.

Figure 6.23 shows the performance of the network when six rules and one free node was used. As the saturation value and the quality factor increased the network required a longer training time to achieve training RMS error of 0.035. This showed that one free node was insufficient for network training. Figure 6.24 and 6.25 shows that when a saturation value between 0 and 0.5 was used the network trained for fewer epochs to achieve training RMS error of 0.035. Figure 6.25 shows that when the quality factor was 0.25 network training time remained between 300 and 400 epochs to achieve training RMS error of 0.035. But as the quality factor increased the training time of the network to achieve training RMS error of 0.035 also increased. This shows that the rules had a greater influence on network training as the quality factor increased.

6.7 Conclusions

- The neural network initialized with problem knowledge consistently detected the wane edges whereas the network initialized with random numbers did not.

- In the experiments a comprehensive set of 65 rules representing problem knowledge was used to initialize the neural network. In experiment 6.6.6 the number of rules used to initialize the network reduced to eight. This showed that a large number of rules are not required to represent problem knowledge. A few rules that describe problem knowledge is sufficient to initialize the network to improve training performance.

- Experiment 6.6.13 showed how a ‘good’ rule changed during training. If a high confidence can be placed on a rule then a quality factor greater that 0.75 can be used for that rule.

Chapter 7

7 Tree crown detection

This chapter investigates the use of a fuzzy neural network window filter for the reliable detection of tree crowns in an aerial photograph of a forest plantation consisting solely of Pinus Radiata. The detected tree crowns are then used to count the number of trees for inventory purposes. Aerial photograph and ground truth data containing the position of the trees used to train the network. A considerable time interval exists between the capture of aerial images of the forest plantation and the collection of the data (8 years). Due to this time interval, a high confidence level could not be placed on the ground truth. This problem was therefore an ideal test case for experiments designed to determine the effects of using rules to initialize a FuNN when unreliable and noisy data is available. The experiments show that the use of rules improves the robustness of a FuNN in such a situation.

7.1 Introduction

There are large numbers of commercial Pinus Radiata plantations in New Zealand containing in aggregate hundreds of millions of trees. Methods are being developed to automatically count the number of trees from aerial photographs so that an inventory can be made. The plantations that are currently under scrutiny consist solely of Pinus Radiata. Standard procedures for tree count estimation

involves using aerial photographs and manual photo interpretation techniques. Experts use the aerial photographs to generate pseudo three-dimensional images by which the number of trees in that plot can be counted. Figure 7.1 shows an aerial photograph of a commercial plantation consisting of Pinus Radiata. The aerial photographs were taken at an elevation of 5000 feet with a sun elevation of approximately 30^0 and 60^0.

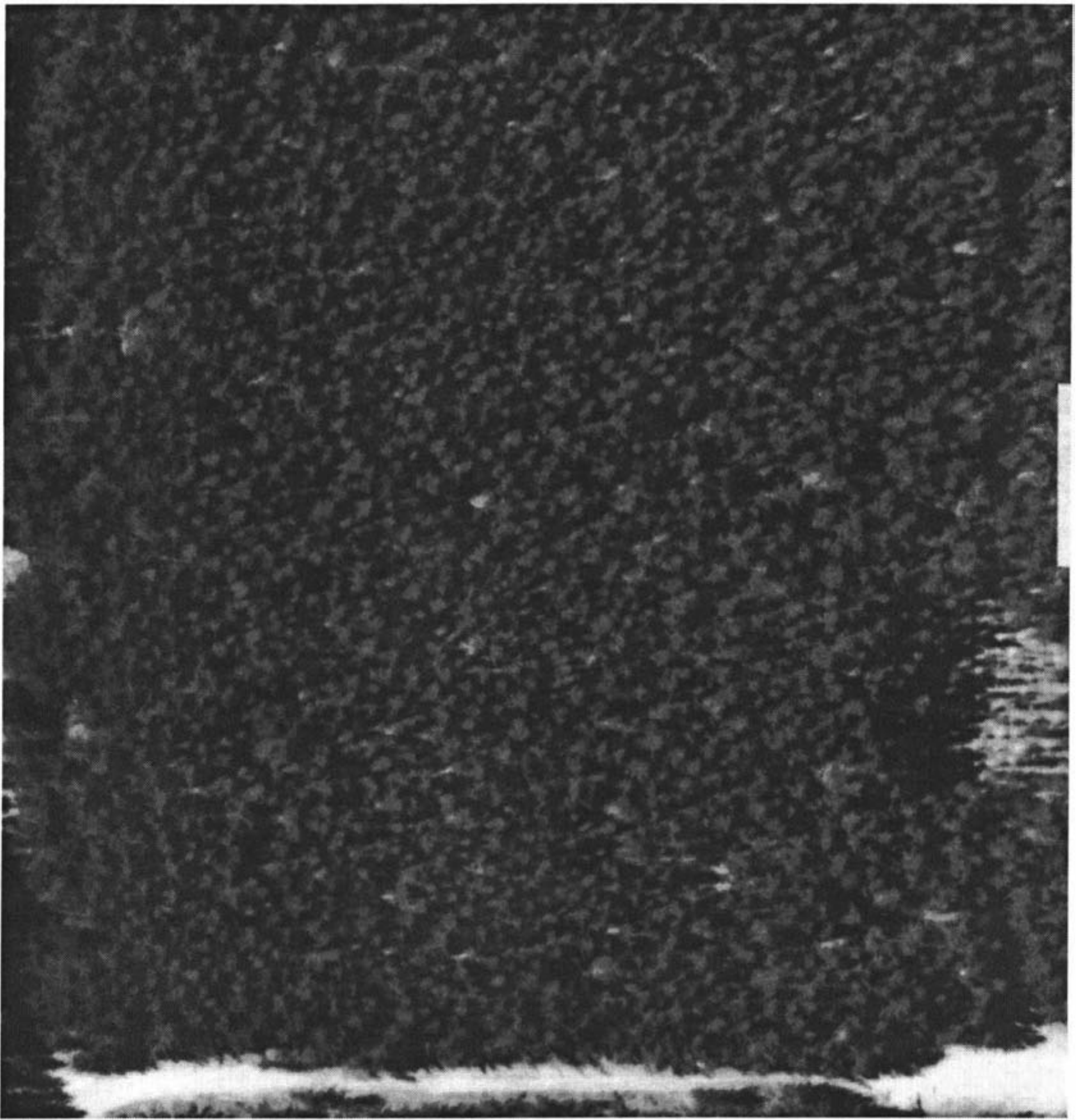


Figure 7.1 Aerial photograph of a commercial Pinus Radiata plantation.

The images used in the experiments were obtained from Landcare Research in Palmerston North. Initially the ground truth data detailing the positions of the trees in the aerial photograph was not available. What is described as ground truth is expert interpretation. Therefore the experiments described in this chapter were

performed in two series. The first series describes the experiments that were performed using ground truth data manually generated by the author. The second series describes the experiments that were performed using the ground truth data that was subsequently obtained from Landcare Research. A considerable time interval exists between aerial image capture and ground truth generation by the 3D photo interpreters. The images were captured in April 1991 and the ground truth data was collected in early 1999. During this time some trees have died and other have been wind swept thus introducing error in the ground truth data.

7.1.1 The training images

The aerial image was pre-processed to remove roads and clearings before extracting a sub image to generate the training images. The aerial photograph was a colour infrared image. The pre-processing steps were,

1. Split the original colour infrared image into its RGB components.
2. Roads and clearings were visible in the blue component. The blue component is then thresholded.
3. The resulting thresholded blue component and red component are ANDed to remove unwanted roads and clearings from the red component image.
4. The resultant image from step 3 was then converted to a 256-grey scale image.
5. On closer inspection of the aerial photograph it was observed that the left side of the image had a lower illumination when compared to the right side of the image. This was corrected by performing an illumination correction process [Matlab IPT] on the image from step 4. A histogram equalisation process [Russ 1995] was applied to the resultant image to generate the final image that was used as the input image for the network.

The pre-process procedure was developed through an iterative process. Figure 7.2 shows the training images, image (a) is a sub image form the aerial photograph. Image (b) was extracted from the pre-processed aerial photograph. Image (c) was the target image that was generated by hand. A point that corresponded to a tree position in image (b) was placed on the target image. Then the target image was

processed using a Gaussian filter to generate the final target image (c). Errors maybe present in the position of the trees in the target image as they were marked by hand. Images (b) and (c) were used for training.

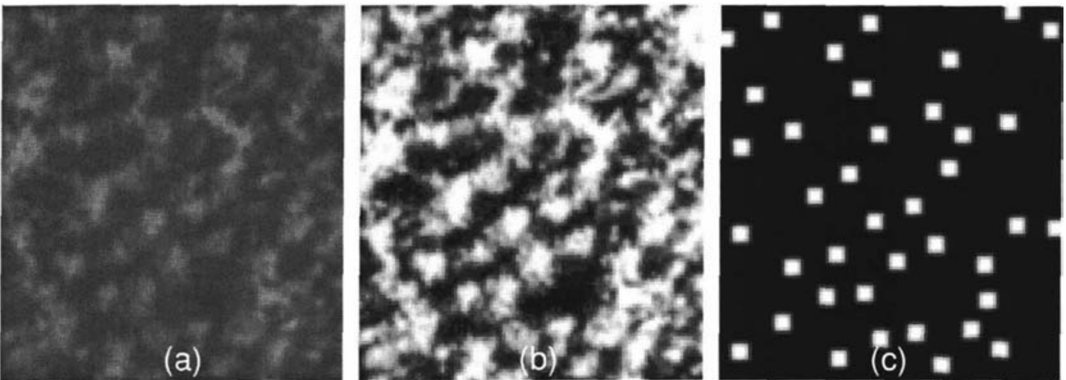


Figure 7.2 The training images. Image (a) is a sub image from the original aerial photograph. Image (b) is the pre-processed image and image (c) is the target image.

7.1.2 Window filter size

The experiments reported in the previous chapters confirmed that the size of the window filter should be matched to the size of the feature under scrutiny. For this problem the feature was a tree. After inspecting many trees in the aerial photograph it was determined that a typical tree had an average diameter of 11 pixel, so an 11x11 window filter was used for the experiments. Figure 7.3 shows an image of a tree superimposed on the 11x11 window filter.

Figure 7.3 Image of a tree superimposed on the 11x11 window.

7.1.3 The membership functions

Figure 7.3 shows an image of a tree superimposed on the 11x11 window. The window shows that when a tree lies within the window the centre pixels have a high intensity and the boundary pixels have a low intensity. Two membership functions can be derived from this information.

Membership function	Verbal representation	Value
M1	Low	0
M2	High	1

Table 7.1 Membership functions for tree crown detection.

M1 and M2 were used as the fuzzification membership functions. M1 and M2 were also used as the defuzzification membership functions representing *tree not found* and *tree found* respectively.

7.1.4 The rules

Rule 1

If the intensity of the pixels in the middle of the window is greater than the intensity of pixels near the boundary of the window and the high intensity area is circular then the output is high (tree found).

Rule 2

This rule represents the case when the window passes over small tree crowns that may represent secondary branches that have grown to the top of the canopy. This type of crown should not be labeled as a tree crown. The verbal representation is as follows:
If the intensity of the middle is high and the high intensity area is small then output is low.

Rule 3

Figure 7.4 shows two trees that are close to each other. The window represents how the two trees can be separated. The verbal representation is as follows:
If the intensity of the top right and bottom left of the window is high and the intensity of the top left and bottom right of the window is low then output is low.

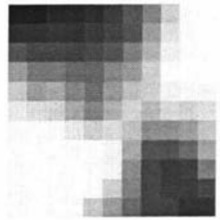


Figure 7.4 Image of two trees close to each other. Rule 3 was generated from this image

Rule 4

Figure 7.5 also represents two trees close to each other. The window represents how the two trees can be separated. This can be verbally represented as:
If the intensity of the top left and the bottom right of the window is high and the intensity of the top right and bottom left is low then output is low.

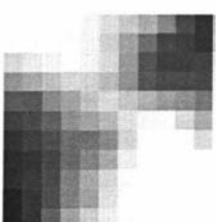


Figure 7.5 Image of two trees close to each other. Rule 4 was generated from this image.

Rule 5

Figure 7.6 also shows two trees close to each other. The window represents how the trees can be separated. This can be verbally represented as:
If the intensity of the top and bottom of the window is low and the intensity of the left middle and right middle of the window is high then the output is low.

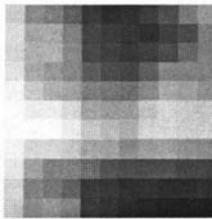


Figure 7.6 Image of two trees horizontally close to each other. Rule 5 was generated from this image.

Rule 6

Figure 7.7 also shows two trees close to each other. The window represents how the trees can be separated. This can be verbally represented as:
If the intensity of the left and right side of the window is low and the intensity of the top middle and bottom middle is high then the output is low.

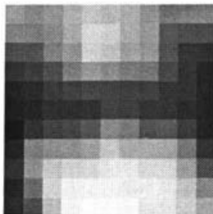


Figure 7.7 Image of two trees vertically close to each other. Rule 6 was generated from this image.

7.2 Experiments

Initially ground truth data was unavailable so the experiments described in this section were performed using the target images generated by hand.

7.2.1 Experiment to develop a neural network to detect tree crowns

This experiment was performed to develop a network to detect tree crowns. The rules described in section 7.1.4 were used to generate a rule set. A saturation value of one was used for the rules as it was observed to be an appropriate value to use when starting new experiments.

7.2.1.1 Network structure used

Layer	No. of nodes
Input layer	121 nodes (11x11 window)
Condition layer	2 membership functions for each input = 242 nodes.
Rule layer	4 rules + 11 free = 15 nodes
Action layer	2 nodes
Output layer	1 output

Table 7.2 Network structure for experiment 7.2.1.

Initially the number of epochs needed to train the network was unknown, so the network initialized with rules was trained for 500 epochs. Another network with the same structure was initialized with random numbers in the range {-1, +1} and trained for 500 epochs for comparison purposes. Figure 7.8 shows the output images generated by the trained networks. Image (a) was generated by the network initialized with rules and image (b) was generated by the network initialized with random numbers.

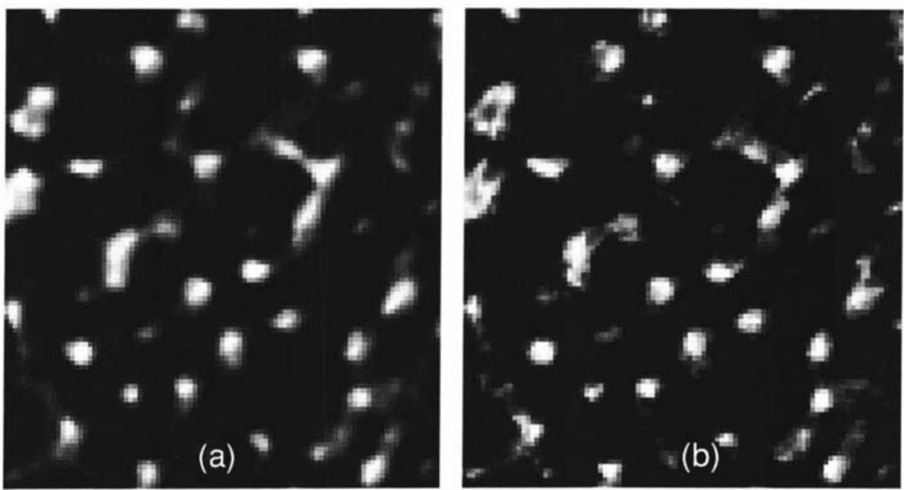


Figure 7.8 Output images generated by the trained networks for experiment 7.2.1. Image (a) is the output image generated by the network initialized with rules and image (b) is the output generated by the network initialized with random numbers.

Figure 7.9 shows the training performance graph for the network initialized with rules and random numbers. The graph shows that the network initialized with random numbers had a better training performance than the network initialized

with rules. The output images were processed using a routine to count the number of detected trees.

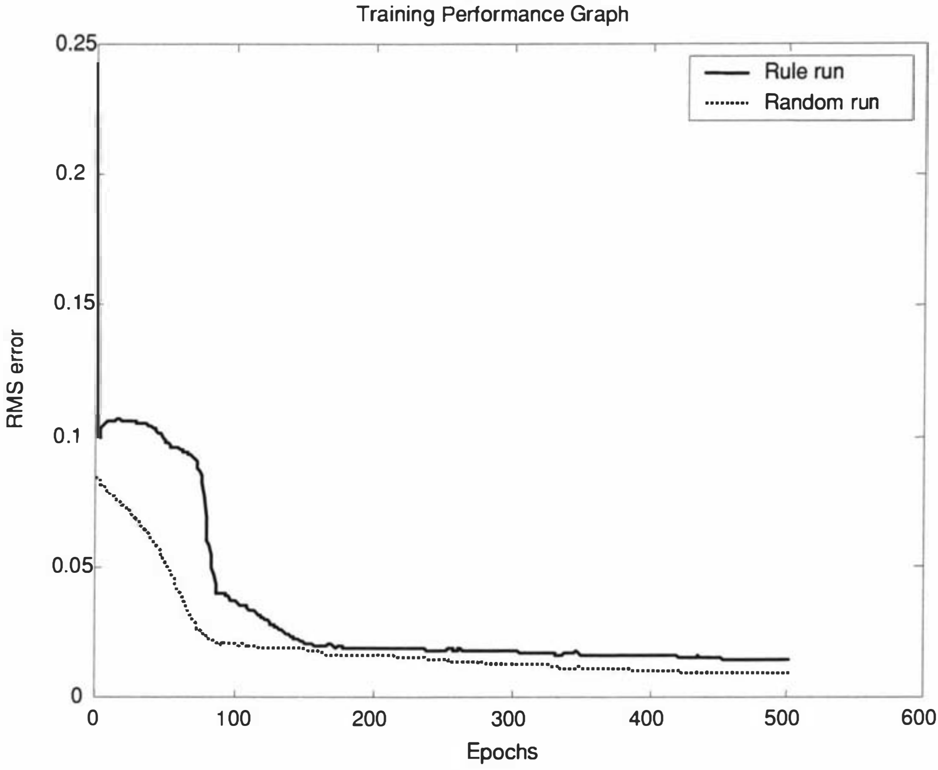


Figure 7.9 Training performance graph for network initialized with and without rules for experiment 7.2.1.

7.2.1.2 The counting routine

The counting routine is as follows,

1. The output image was thresholded using a value of 90.
2. The items in the thresholded image were labelled and counted using the label command in Matlab.

The value used for thresholding was found through an iterative process.

For tree count comparison, the target image was cropped to the same size as the output image as the border of the input image is lost during window processing. The target image contained 31 trees. The output image generated by the network initialized with rules indicated 38 trees while the output image generated by the

network initialized with random numbers indicated 43 trees. The network initialized with rules gave a tree count closer to the correct tree count.

Several similar experiments were performed to detect and count the tree crowns in the input image. In all of the experiments the number of trees in the output image was higher than the expected tree count. When the output image was superimposed on the input image the following was observed,

- The network detected pixels in the input image that belong to trees.
- Trees that are clumped together create a blob of clustered pixels on the output image. A blob of clustered pixels is counted as one tree.
- As there are many blobs of clustered pixels in the output image the tree count should be lower than the expected tree count. The noise in the output image causes the tree count to rise.

7.2.1.3 Post processing the output image

The trained network detected pixels that belong to a tree in the input image. One problem with this approach was that trees that were close together appeared as blobs on the output image. For example, three trees that were close to each other would appear as one large blob on the output image. A post process routine was used on the output image to determine if the blobs could be divided to represent the individual trees. The post process routine involved passing an erosion filter and then a dilation filter over the output image. When the erosion filter was used, noise and small points representing trees were removed. As this reduced the tree count this post processing routine was abandoned.

7.2.1.4 A different approach to solving the problem

The previous experiments showed that a neural network could detect pixels that belong to a tree in the input image. Trees that were close together were detected and displayed as a blob of pixels on the output image. The trees that were close together could not be represented as individual trees on the output image by the network. It was concluded that a different approach should be investigated.

7.2.1.5 Hierarchical neural network

In a modular systems neural networks can be arranged sequentially or in this case hierarchically to solve subtasks of the global task [Kasabov 1996]. Figure 7.10 shows the hierarchical network structure that is expected to detect the tree crowns. Neural network 1 (NN1) will detect the pixels that belong to a tree and neural network 2 (NN2) will post process the output image generated by NN1. It is expected that NN2 will divide the larger blobs representing clusters of trees into smaller blobs representing individual trees and remove unwanted noise from the output image generated from NN1.

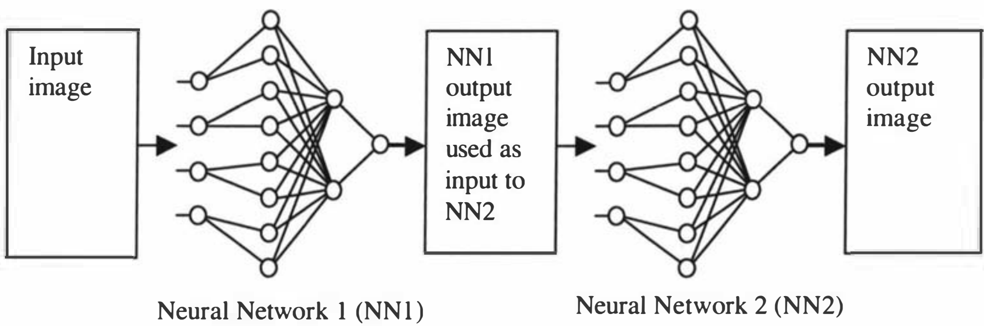


Figure 7.10 Hierarchical neural network structure.

7.2.2 Experiment to investigate the used of a hierarchical network structure to separate blobs of clustered trees into blobs corresponding to individual trees

Neural network 1 is not trained in this experiment as it has already been done in the previous sections. The output image generated by processing the original input image using NN1 is used as the input image for this experiment. As window processing an image removes an outer border, the resulting image is smaller than the original. So the input image must be cropped to the size of the output image when performing tree counts. This experiment was performed to determine if neural network 2 described in section 7.2.1.5 could divide the larger blobs representing clusters of trees into individual smaller blobs representing individual trees. The average size of a blob representing a single detected tree in the output image generated by neural network 1, had a diameter of 7 pixels, so a 7x7 window filter was used for network 2.

7.2.2.1 Network structure used 7.2.2

Layer	No. of nodes
Input layer	49 nodes (7x7 window)
Condition layer	2 membership functions for each input = 98 nodes
Rule layer	3 rules + 12 free nodes = 15 nodes
Action layer	2 nodes
Output layer	1 node

Table 7.3 Network structure for experiment 7.2.2.

7.2.2.2 The rules used for experiment 7.2.2

Rule	Description
1	If the intensity of the middle section of the window is less than the intensity of the side sections of the window then output is low (rule designed to separate trees that are close together in the Y axis).
2	If the intensity of the middle section of the window is less than the intensity of the top and bottom sections of the window then output is low (rule designed to separate trees that are close together in the X axis).
3	If the intensity of the middle of the window is high and the intensity near the border of the window is low and the high intensity area is circular then the output is high (rule designed to retain single trees).

Table 7.4 Rules for experiment 7.2.2.

A FuNN initialized with the above rules was trained for 1000 epochs. A network with the same structure was initialized with random numbers {-1, +1} and trained for 1000 epochs for comparison purposes. Figure 7.11 shows the output images generated by the network initialized with and without rules. Image (a) is the input image, image (b) is the output image generated by the network initialized with rules and image (c) is the output image generated by the network initialized with random numbers. The input image has 28 trees as this was the number of trees remaining from the original image after cropping due to window filter processing. NN2 initialized with rules gave a tree count of 34 trees and NN2 initialized with random numbers gave a tree count of 33.

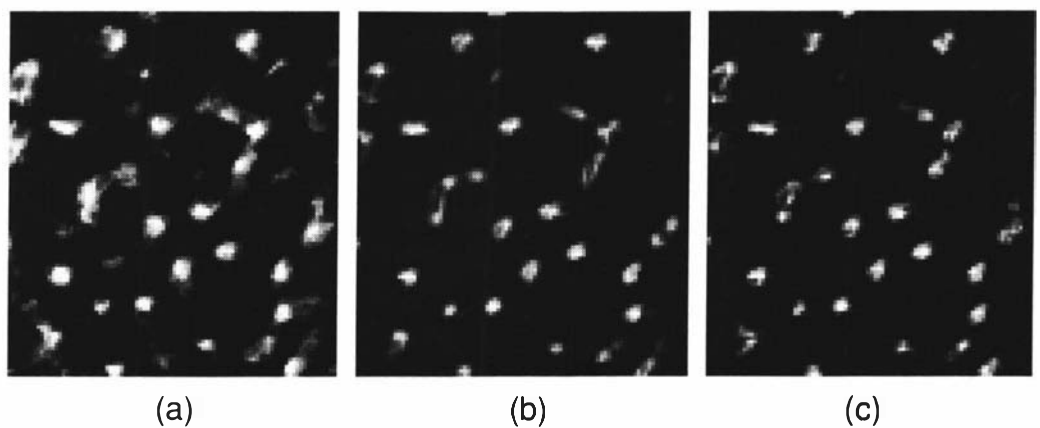


Figure 7.11 Output images generated by the trained networks for experiment 7.2.2. Image (a) is the input image, image (b) is the output image generated by the network initialized with rules and image (c) is the output generated by the network initialized with random numbers.

Figure 7.12 shows the training performance graph for the networks initialized with and without rules.

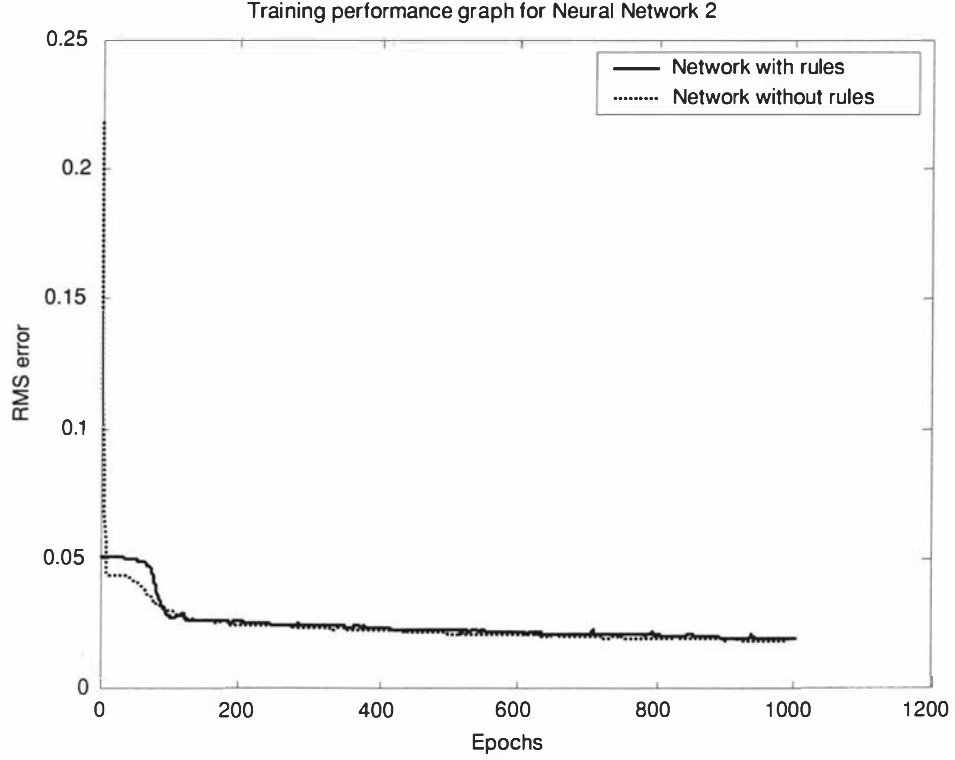


Figure 7.12 Training performance graph for networks initialized with and without rules for experiment 7.2.2.

Table 7.5 shows the training RMS error for both networks. The network initialized with rules has started from a lower RMS error compared to the network initialized without rules. Although the network initialized with rules has started from lower training error, after 1000 epochs both networks have the same training RMS error.

	Starting RMS error	RMS error after 1000 epochs
Network initialized with rules	0.051	0.02
Network initialized without rules	0.22	0.02

Table 7.5 Training RMS error for networks before and after training for experiment 7.2.2.

As expected NN2 divided the larger blobs into smaller blobs representing individual trees. For this experiment the use of rules has only improved the starting position of the network for training. It is important to note that a reliable error cannot be derived from the tree count for the output images, as the ground truth was not available.

7.2.3 Experiment to determine the tree counts for multiple networks initialized with random numbers.

The previous experiment showed how NN2 from the hierarchical network structure was used to separate blobs representing trees close together into individual trees. This experiment was performed to determine the tree crown detection performance of several hierarchical neural networks. The training images were generated from a sub image from a commercial plot that was available. The sub image was chosen from an area on the original image where the trees were spread out evenly. The ground truth for the training images was not available, although the tree count for the commercial plot was available. 300 hierarchical neural networks were initialized with random numbers in the range {-1, +1}; NN1 was trained for 3000 epochs and NN2 was trained for 1000 epochs. NN1 used an 11x11 window filter and NN2 used a 7x7 window filter. The training images were used to train the network and after training, each hierarchical network was used to process the image of the commercial plot and the tree count was recorded. A second

hierarchical network was initialized with rules, was trained and used to process the image of the commercial plot to obtain a tree count for comparison purposes.

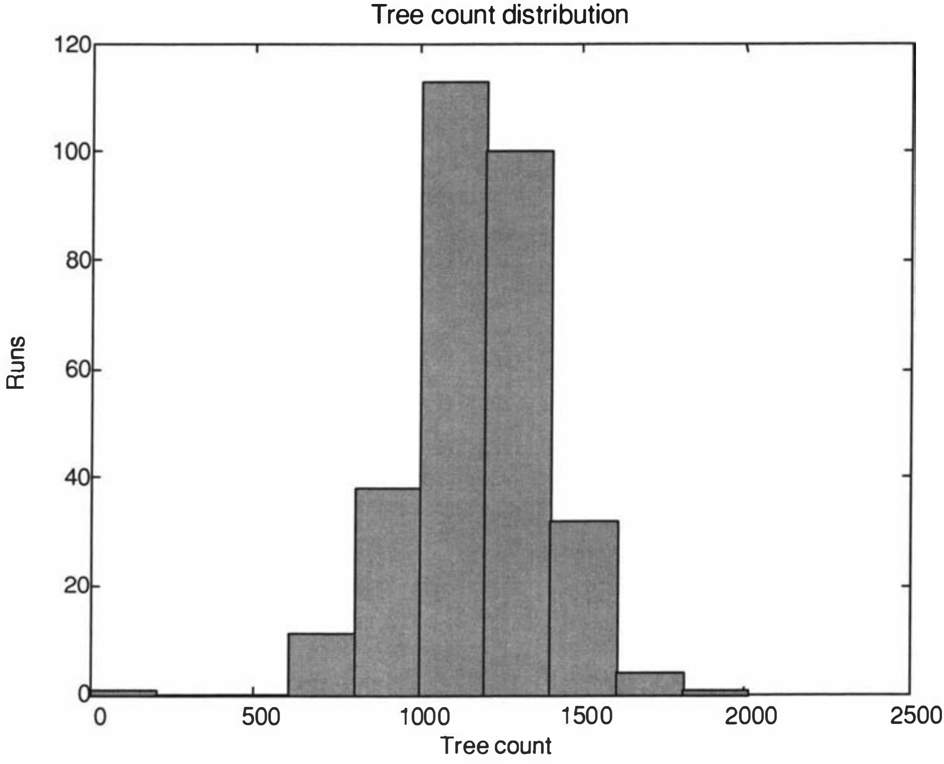


Figure 7.13 Tree count distribution for network initialized with random numbers for experiment 7.2.3.

The commercial plot contained 2370 trees. The mean of the tree count generated by the hierarchical networks without rules was 1184 (an error of 50%) with a standard deviation of 206 with one run generating a blank image (Figure 7.13). The network initialized with rules gave a tree count of 2100 (an error of 11.4%). Even though ground truth data was unavailable the network initialized with rules has given a tree count closer to the actual value. It appears that the rules have compensated for the uncertainty in the ground truth. This hypothesis is investigated in section 7.2.8 in this chapter. After these experiments were completed, ground truth data became available from Landcare Research in Palmerston North.

7.2.4 The new training images

Figure 7.14 shows a commercial plot cropped to its boundary.

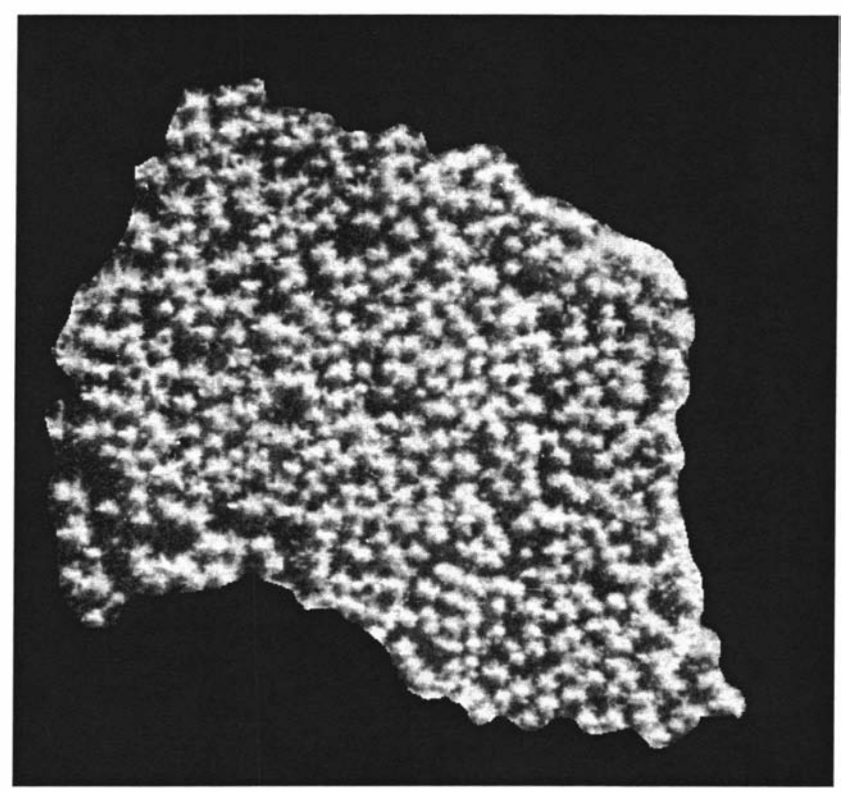


Figure 7.14 The commercial plot.

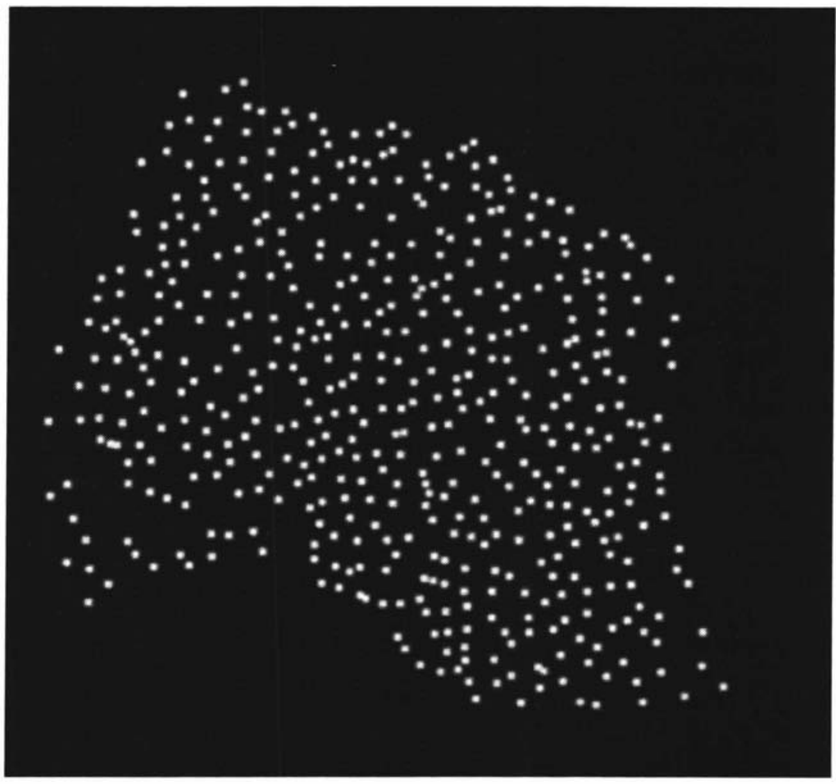


Figure 7.15 The ground truth data for the commercial plot.

Figure 7.15 shows the positions of the trees in the commercial plot. A sub image from the commercial plot and the ground truth image were used to generate

training images. Figure 7.16 shows the new training images that were used for the remaining experiments described in this chapter. To generate the target image white dots were placed at the locations specified by the ground truth data. A Gaussian filter with a radius of 1.4 was then passed over the target image to create the final target image used for training. The Gaussian filter was used to generate a range of possible locations around the point where a tree is known to exists, this was done to facilitate network training [Pugmire 1995].

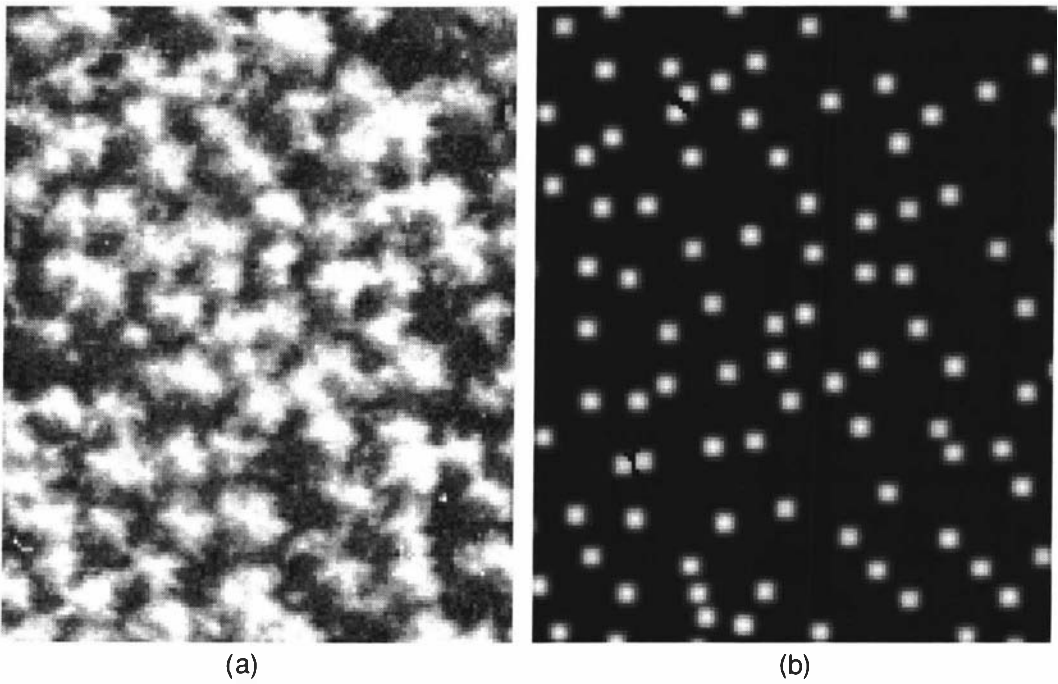


Figure 7.16 The new training images for the tree crown detection problem. Image (a) is the input image and image (b) is the target image.

7.2.5 Experiment to determine the tree count when the network was trained using the supplied ground truth based training images

The previous experiments were performed using unreliable training images. This experiment was performed to determine the network performance achieved with reliable training data. The hierarchical network structure was used for the experiment. NN1 and NN2 were initialized with rules. NN1 used an 11x11 window filter and NN2 used a 7x7 window filter. NN1 was trained for 3000 epochs to achieve an acceptable final training RMS error and NN2 was trained for 1000 epochs. The rules in section 7.1.4 were used to initialize NN1 and the rules

specified in section 7.2.2.2 were used to initialize NN2. Figure 7.17 shows the output image generated by the trained hierarchical neural networks. The target image was cropped to match the size of the output image, as the border of the input image is lost due to window processing. The target image had 71 trees. The trained hierarchical networks were used to process the input image to generate the output image which contained 64 trees.

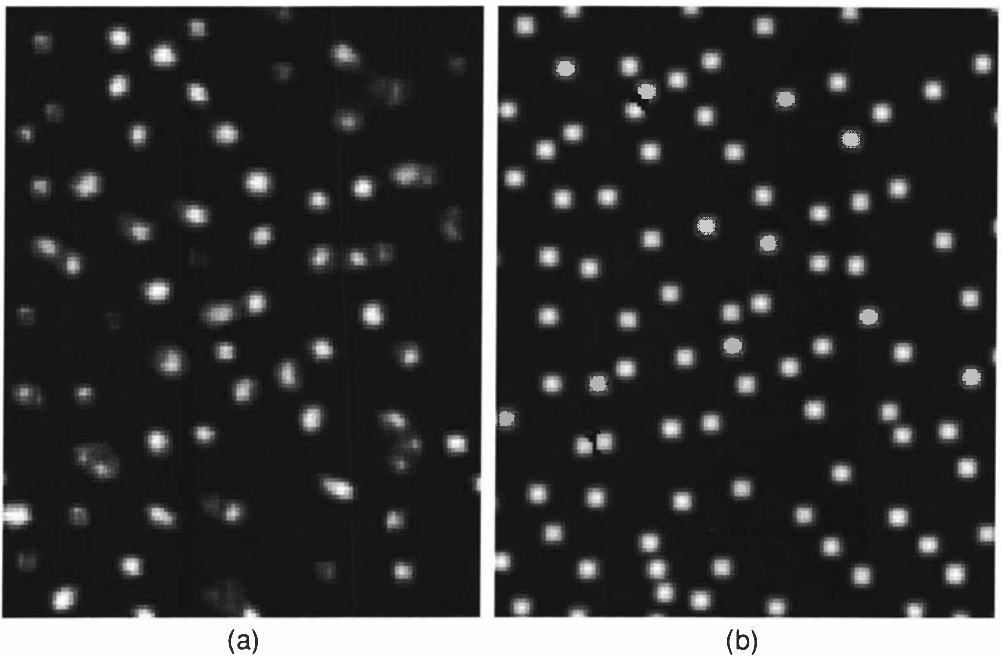


Figure 7.17 Image (a) is the output image generated by the hierarchical neural networks for experiment 7.2.5. Image (b) is the target image.

The error between the numbers of trees detected in the output and the target image was 9.8%. When the network was trained using the target image generated by the author the error between the number of trees detected in the output and target image was 11%. A smaller error was expected when the ground truth data was used for training. This gave rise to the speculation that the ground truth data was not 100% accurate so the target image was superimposed on the input image to scrutinize the marked tree positions. Figure 7.18 shows the target image superimposed on the input image. The rectangular boxes and circles show the tree positions that correspond to the positions given by the ground truth data. There is no significance between the use of boxes and circles. These were generated by the superimposing process.

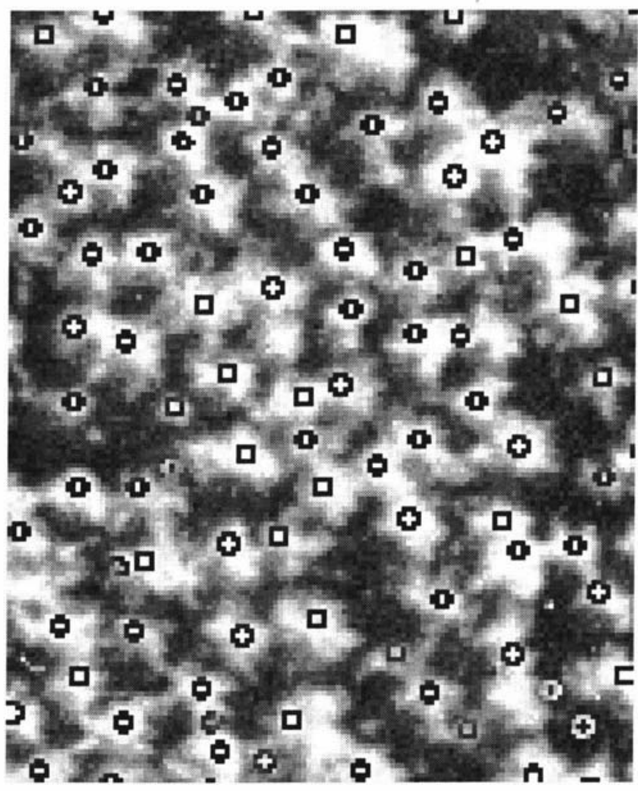


Figure 7.18 The image shows a sub-image from the commercial plot. The boxes and circles mark the tree positions given by the ground truth data.

It is apparent that the ground truth data is not 100% accurate. Careful study of the ground truth data revealed the following points,

1. Some positions in the ground truth data do not coincide with the positions of trees on the input image - the data has transposition errors.
2. In some positions, where a tree is supposed to exist, a tree is not present. Wind blown trees or trees that have died in the plot could explain this as there was a significant time difference between the image capture and the ground truth data collection. This was termed as a false positive.
3. In some positions a tree apparently exists but according to the ground truth data a tree cannot exist in that position. This was termed as a false negative.

The ground truth data was confirmed to be less than perfect, placing a limit on the performance expected of the neural network.

7.2.6 Experiment to determine the tree count made by the neural network using modified ground truth data.

The previous experiment investigated the performance of the neural network using the original ground truth data. It was observed that a high confidence level could not be placed on the ground truth data. For this experiment the tree positions in the ground truth data were modified to better coincide with the actual tree positions in the input image. The modifications were done manually. Details of the modifications are given below.

7.2.6.1 Adjustments to target image

Table 7.6 shows the points that were added to the target image to create the new target image.

Points added (x,y) format
(78,11)
(111, 47)
(56, 68)
(4,119)
(66,151)

Table 7.6 Points added to the target image.

Table 7.7 shows the points that were repositioned in the target image to create the new target image.

Points moved (x,y) format)
(39,38)→(42,40)
(28,113) →(33,116)
(23,115)→(25,114)
(45,110)→(47,113)
(105,112)→(106,116)
(112,141)→(106.142)

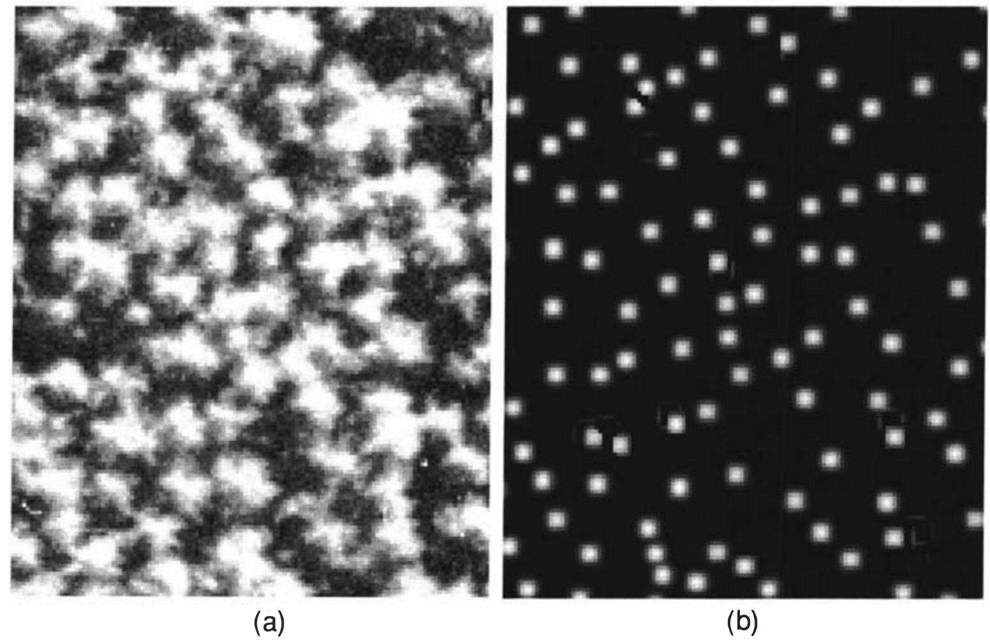
Table 7.7 Points moved in the target image.

Table 7.8 shows the point removed from the target image to create the new target image.

Points removed (x,y) format)
(119,148)

Table 7.8 Points removed from the target image.

Figure 7.19 shows the new target image used to train the hierarchical network.



(a) (b)

Figure 7.19 New training image for tree crown detection problem. (a) input image and (b) target image.

Table 7.9 shows the training performance of the networks.

Neural network	Starting RMS Error (training)	Final RMS Error (training)
NN1 (trained for 3000 epochs)	0.0417	0.0140
NN2 (trained for 1000 epochs)	0.0138	0.0109

Table 7.9 Training performance for NN1 and NN2 for experiment 7.2.6.

Figure 7.20 shows the network output generated by processing the input image with the trained hierarchical neural networks. The image has a tree count of 72. The cropped target image had a tree count of 77. The error between the number of trees in the output and target image was 6.5%. The tree count in the output image increased after the network was trained on the new target image. 100 training runs were performed by adding different random numbers in the range {-0.1, +0,1} to the network initialized with rules to remove symmetry problems. The output images generated by these networks gave a consistent tree count of 72. The small random numbers added to the network initialized with rules had a minimum effect on network training.

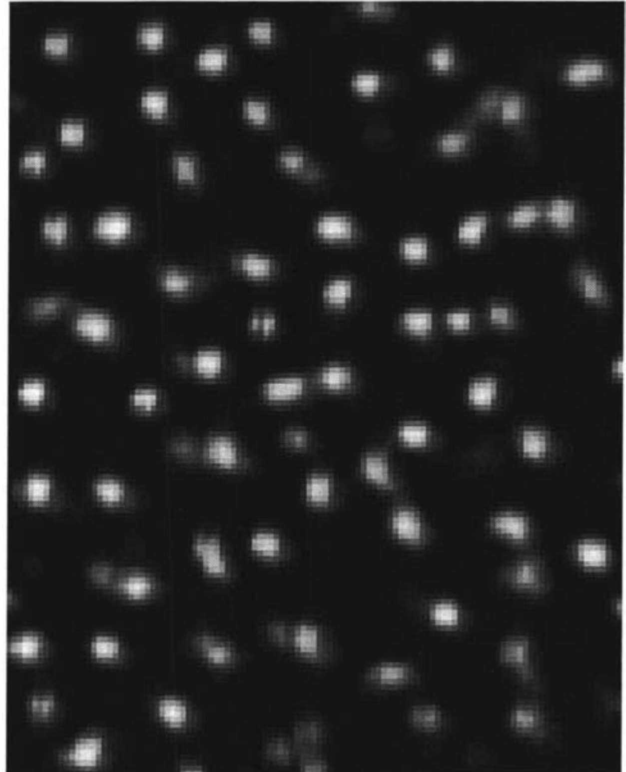


Figure 7.20 Output image generated by hierarchical neural network for experiment 7.2.6.

7.2.7 Experiment to determine the tree counts for 100 networks initialized with random numbers.

The network structured was the same as that used in the previous experiment. The network weights of NN1 and NN2 were initialized with random numbers. NN1 was trained for 3000 epochs and NN2 was trained for 1000 epochs. The network

was trained using the new target image generated in section 7.2.6.1. Figure 7.21 shows the tree count for the 100 runs made from different random weight seedings.

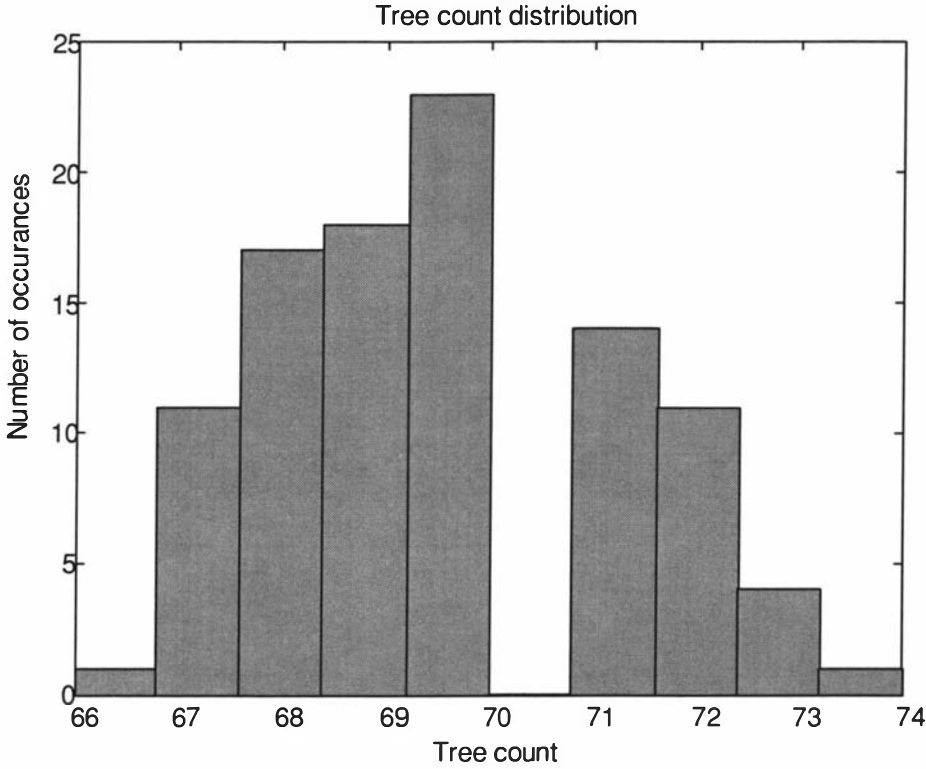


Figure 7.21 Tree count distribution for experiment 7.2.7.

The number of trees detected by the hierarchical networks had an average tree count of 69 with a variance of 3 and a median of 70. The network initialized with rules in experiment 7.2.6 gave a tree count closer to the actual when compared to the networks initialized with random numbers.

7.2.8 Experiment to investigate the hypothesis that the rules compensate for inaccuracies in the training data

PROPOSAL:

Initializing a neural network with rules can compensate for inconsistencies in the training data.

This was developed after observing the behavior of the neural network recorded in section 7.2.3, where several neural network runs were performed using random initialization. In the initial experiments (section 7.2.3) the confidence level of the training data was very low, as ground truth data was not available at the time. When the network was initialized with rules the network produced a more accurate tree count than when initialized from a random start. In a practical environment a 100% accuracy of training data cannot be achieved. The task of counting and recording the position of trees is very expensive, performing such a task every six months of every year would not be practical. Inconsistencies in the training data can be compensated for by using rules prior to training the network. The following experiments investigate the proposal that rules can be used to compensate for the effects of inconsistencies in the training data.

7.2.8.1 Experiments to determine if the accuracy of the training data complements neural network training.

The network performance is known when the modified ground truth data was used to train the network. For this experiment a new set of target images were created in which the tree positions were shifted randomly in the new target images creating uncertainty. The network was trained on these new target images to determine network performance. Five new target images were generated for the experiment. The images were generated as follows:

- The known individual tree positions were randomly shifted by a distance D (pixels) creating uncertainty.

  Position of noisy point on new target image =

  Position of point on (original) target image + D

- The average amount of uncertainty added was determined by the standard deviation σ and the displacement D was derived by a random selection from a normal distribution curve.

σ was varied between 1 and 5 (1,2,3,4,5) to generate 5 new target images.
The pixel displacement was varied according to the value of D.
If D>=0 and D<0.5 then pixel displacement = 0
If D>0.5 and D<1.5 then pixel displacement = 1
If D>1.5 and D<2.5 then pixel displacement = 2
If D>2.5 then pixel displacement =3
*each tree point in the target image was displaced in both x and y directions (positively and negatively). The new target images were used to train the network.

For each new target image six training runs were performed. Of the six runs, one run was a rule initialized run and the other five were performed by initializing the weights of the network with random numbers in the range {-1, +1}. The target images had a tree count of 72. As several runs were to be performed, the training time for NN1 and NN2 was reduced to 200 and 100 epochs respectively. These training times were found to be sufficient to arrive at a final tree count for this experiment. The output image generated by the network initialized with rules when σ=0 (no uncertainty) gave a tree count of 69. Table 7.10 shows the tree count for the output images generated by the trained networks.

σ	Rule run	Five runs initialized with random numbers				
1	59	59	61	42	58	60
2	31	3	11	6	1	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0

Table 7.10 Tree count for experiment 7.2.8.1.

As the uncertainty increases (σ) the randomly initialized neural network fails to find a reasonable solution. The network initialized with rules performed better

than the randomly initialized network when the uncertainty in the data was increased. The rules have compensated for the inaccuracy of the training data.

7.2.8.2 Experiments to determine if the accuracy of the training data complements neural network training continued.

This is a continuation of the previous experiment. In the previous experiment σ was varied between one and five pixels. Table 7.10 shows that the tree count dropped to zero when σ was greater than three. This experiment was performed to determine the performance of the network for value of σ between 1 and 2.5. The value of σ was varied between 1 and 2.5 (σ = 1, 1.25, 1.50, 1.75, 2.0, 2.25, 2.50) to create 7 new target images.

Target image set generation method

The positions of the trees were shifted as follows,

tree position in new target image = tree position in original target image + D

Table 7.11 shows the tree count for the target image set.

σ	Rule run	Five runs initialized with random numbers				
1	59	59	61	42	58	60
1.25	40	3	18	0	0	45
1.50	19	1	16	0	15	15
1.75	19	0	2	0	0	31
2.0	29	0	1	0	2	1
2.25	4	0	0	0	0	0
2.50	25	0	7	0	0	0

Table 7.11 Tree count for target image set 1 for experiment 7.2.8.2.

It can be seen that when the uncertainty of the training data increased the network initialized with rules performed better than when the network weights were initialized with random values.

7.2.8.3 Experiment to determine the network performance for several rule runs

In the previous experiment the network performance was measured for one run where the network was initialized with rules and several runs where the network was initialized with random numbers. This experiment was performed to determine the network performance for several runs where the network was initialized with rules using a range of values for σ.

Target image set generation method

The target images set was generated from the formula

tree position in new target image = tree position in original target image + D

where D is the uncertainty generated by varying σ (0, 1, 1.25, 1.50, 1.75, 2, 2.25, 2.5) from a normal distribution. Ten rule runs were performed on each set of images. Small random values in the range {-0.1, +0.1} where added to each network initialized with rules. Table 7.12 shows the tree count for the output images generated by the networks. Table 7.12 shows that adding small random values to the weights initialized with rules only had a minor effect on the performance of the network. It can be seen that the rules "bias" the network to one particular solution.

Noise levels	Run number									
σ	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
0.00	61	60	60	63	61	63	61	64	61	63
1.00	62	59	59	59	59	61	59	59	59	59
1.25	25	36	25	24	28	36	29	33	35	41
1.50	28	34	34	26	32	15	26	30	25	28
1.75	39	12	22	26	27	19	30	34	2	22
2.00	20	11	11	18	20	20	18	19	10	19
2.25	4	1	2	3	3	3	3	0	2	3
2.50	28	8	27	31	14	21	24	20	22	20

Table 7.12 Tree count generated by network trained on target set 2 for experiment 7.2.8.3.

7.3 Conclusions

- This chapter investigated the effects of rules when the available data is noisy or unreliable. The initial experiments were performed with unreliable data, as ground truth data was unavailable. Experiment 7.2.3 was performed without ground truth data. The output image generated by the network initialized with rules gave an error of 11% while the mean error in the output images generated by the network initialized with random numbers was 50%. This suggested that rules could compensate for the uncertainty in training data. The proposal was demonstrated in section 7.2.8.

- The experiments performed in section 7.2.8 also showed that in this application, rules improve the robustness of a fuzzy neural network.

- The experiments performed in section 7.2.6 showed that a network initialized with rules will always generated a useable solution, unlike when random numbers are used to initialize the weights, where the network may converge to an unsatisfactory solution.

- The experiments reported in this chapter also demonstrated a two stage sequential network where each individual network performed a different task.

Chapter 8

8 Iris edge detection

This chapter investigates the use of a fuzzy neural network window filter for the reliable detection of the edges of the human iris as required for a biometrics verification system based on the use of iris patterns. This problem has been chosen as a test problem because conventional approaches had been found, at best, to give barely satisfactory results and it was suspected that the effective use of problem knowledge could be the key to a more satisfactory solution to the problem. An iris pattern identity verification system was studied as a final year honours project within the image processing research group, and so became a candidate for this study.

8.1 Introduction

In building security systems a method of assessing authorization is required to ensure appropriate access or denial of access is given. One method of assessing authorization is the use of biometrics. Biometrics involves the use of personal physical characteristics to aid verification. These include finger-print, voice, facial and iris recognition [Bishop 2000]. The iris is composed of elastic connective tissue, the trabecular meshwork, whose prenatal morphogenesis is completed during the eighth month of gestation. It consists of pectinate ligaments adhering to a tangled mesh revealing striations, ciliary processes, crypts, rings, furrows, a

corona, sometime freckles, vasculature and other features [Daugman 1999]. The pattern that these features form is unique to every individual [Iridian T].

Iris recognition can be used as a passive identification technique requiring minimal client action. Iris identification also has the following advantages over other biometrics, as the iris:

- Has a structure with low genetic influence,
- Is protected from the environment,
- Changes little over time,
- Changes little with environment conditions,
- Changes little with state of health,
- Is available to most members of society,
- Does not need normal eye function for recognition to work, it works for most blind people.

8.1.1 Iris isolation

Figure 8.1 shows an image of an eye.

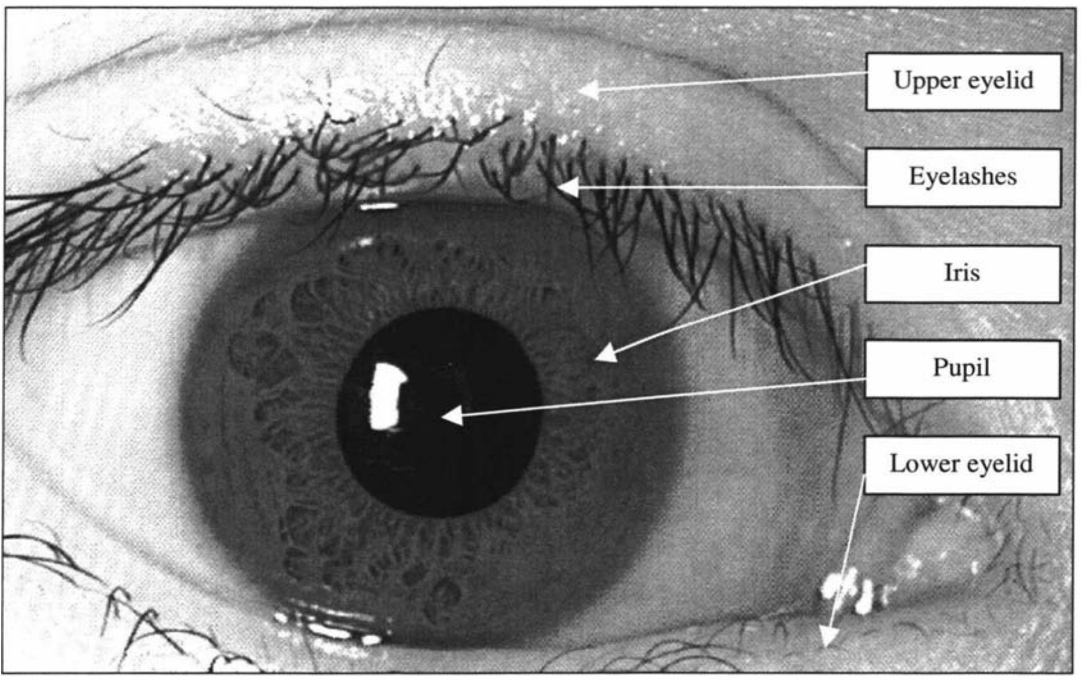


Figure 8.1 Image of an eye used for the iris edge detection problem.

Before an authorization process can be used the iris must be isolated from the rest of the eye. A number of steps are involved in this process. The first step is to apply a linear transformation to the eye image in Figure 8.1 transforming the iris from a donut to a rectangular shape, allowing easier boundary detection methods to be used. First the center of the pupil was found using a center determination process [Bishop 2000]. The center of the pupil was used in the linear transformation process of the eye image.

8.1.1.1 Center determination process

An important pre-processing step in the iris detection process is the determination of the center or origin of the coordinates of the eye. The algorithm for this process is,

1. Expand the lower half of the gray scale of the eye image to remove darker features such as eyelashes.
2. Threshold the image to allow for the detection of the object with the largest area. This assumes that the pupil is the largest dark connected object in the image.

The center pupil is defined as the center of its bounding rectangle, as opposed to its chain coded center of gravity. This removes bias introduced by specular highlights from the illumination source. These highlights cause 'holes' in the pupil after thresholding. If the 'holes' were off-center, this would introduce a weighting bias. There is also the possibility of the 'holes' creating a 'U'-shaped region, in which case a bounding rectangle is affected less than a weighted center [Bishop 2000].

8.1.1.2 Polar conversion

Using the center of the pupil, the next stage in the algorithm is to apply a polar transformation to the image of the eye. This transforms the iris from a donut shape to a rectangular shape, allowing boundary detection methods to be readily used [Bishop 2000].

Figure 8.2 (a) shows the result of using a polar to rectangular transform on the eye image in Figure 8.1. Figure 8.2 (b) shows the pupil and iris edges on the linearised eye image. The next step is to isolate the iris from the rest of the eye. The area between the pupil and white region of the eye is what is required for iris identification.

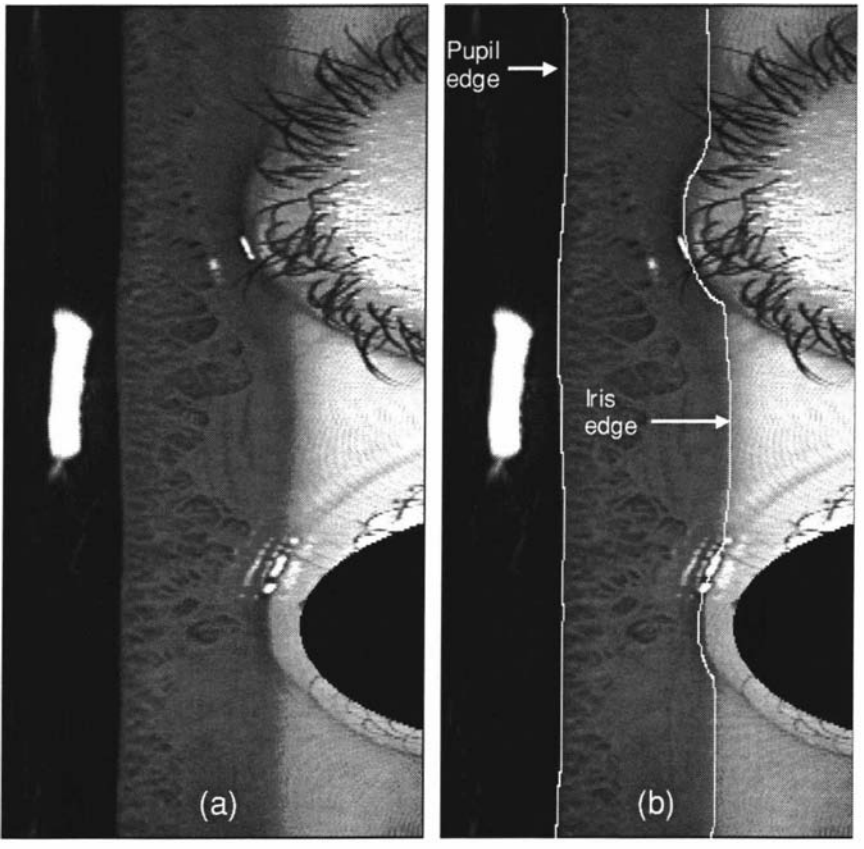


Figure 8.2 Image (a) is the linearised eye image using a polar to rectangular transform. Image (b) shows the pupil and iris edges of the eye.

8.1.2 Use of a standard filter to detect the iris edge

The iris edge is predominantly vertical, so a common vertical edge detector was used to detect the iris edge. A 3x3 window with coefficients

$$\begin{matrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{matrix}$$

was used to detect the vertical edges in the transformed eye image.

Figure 8.3(a) shows the input image and image (b) is the result of processing the input image using the specified vertical edge detection filter. The pupil edge has been detected but the iris edge has not been detected. Here a standard image processing approach using a simple window filter has failed to detect the iris edge. Other more sophisticated approaches were tried by A. W. Bishop [Bishop 2000] and were also found to be unsuccessful.

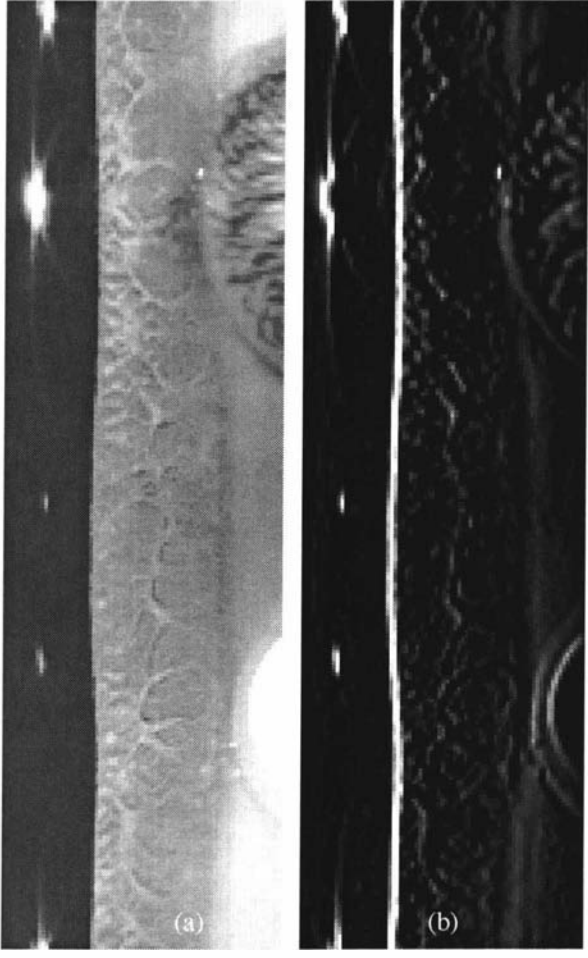


Figure 8.3 Processing the input image using a vertical edge detection filter. Image (a) is the input image and image (b) is the processed image.

This chapter investigates the use of a fuzzy neural network window filter for the detection of the iris edge.

8.2 The training images

Figure 8.4 shows the training images used. Image (a) is the transformed eye image and image (b) is the target image. The iris edge in the target image was drawn by hand. As a first estimate an iris edge with a width of one pixel was first drawn and then the image was blurred using a Gaussian filter. The positions of the pixels in the iris edge drawn by hand had inherent displacement errors, by blurring the image a range of possible iris edge positions was generated.

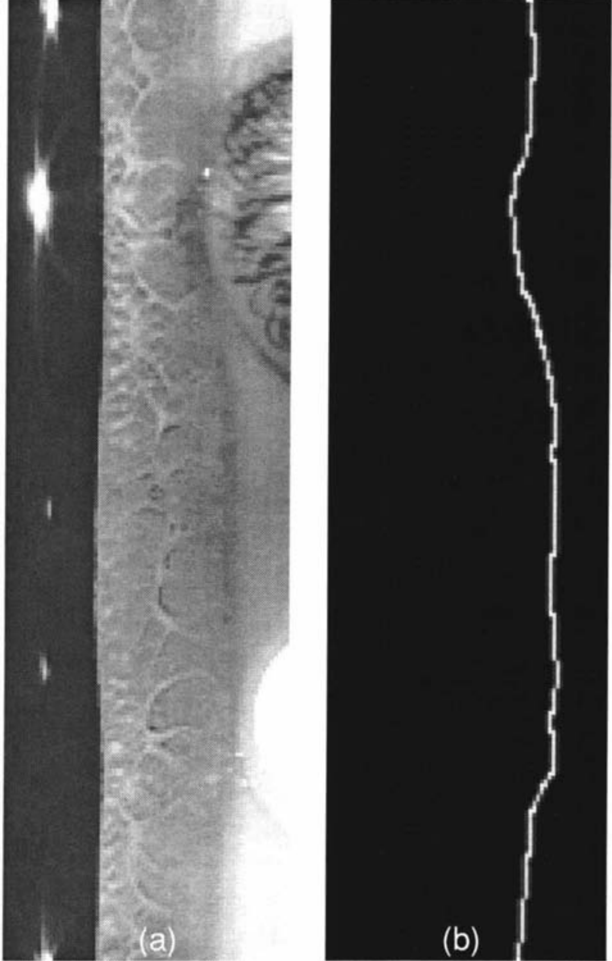


Figure 8.4 Training images for iris detection problem. Image (a) is the input image and image (b) is the target image.

8.3 Development of a figure of merit

A figure of merit was developed that measures the error between a target image and an image generated by processing an input image using a trained FuNN. Figure 8.5 is an image used to describe how the error is calculated. Figure 8.5 – *3* is the area on the image where the detected iris edge can lie. This area covers a range of possible positions where the iris can lie, as the true position of the iris edge is unknown. Depending on where the iris edge lies a false positive or false negative can be generated. The false positives and negatives will be used in the figure of merit. If placement of the pixels in the iris is correct then a false positive or false negative will not be generated.

8.3.1.1 False positive

If a pixel lies in area *1* or *2* and that pixel should not be present in that location then a false positive is generated.

8.3.1.2 False negative

If a pixel is expected but is not present at a location in area *3* then a false negative is generated. In the context of this problem if a pixel is missing from the iris edge a false negative is generated.

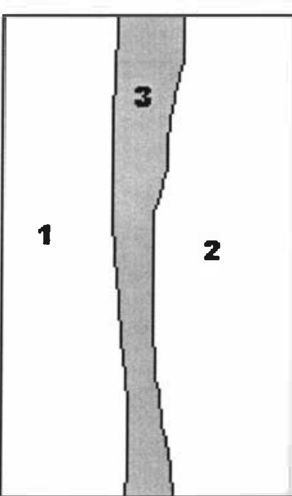


Figure 8.5 Image to demonstrate the error calculation.

The error was calculated using the number of false positives (FP) and false negatives (FN).

$$Error = 0.1 \times FP + 0.5 \times FN$$

The linearised iris image has a height of 250 pixels resulting in a maximum length of 250 pixels for the length of the iris edge. It is assumed the output image generated by processing an iris image using a neural network can have a maximum of 500 false positives. The percentage error is,

$$PercentageError = (0.1 \times FP + 0.5 \times FN) \times 100 / 175$$

The percentage error can be used for comparison purposes. Lower weighting was given to false positives, as they can be removed through a clean-up procedure. The weightings for the equation were determined through an iterative process.

8.4 The experiments

8.4.1 Experiment to determine if a FuNN could detect the iris edge

This experiment was performed to determine if a FuNN initialized with rules could detect the iris edge. Before training, rules representing problem knowledge were generated. The rules were generated from the areas of interest on the input image. Figure 8.6 shows the input image and the areas of interest that were used to generate the membership functions. These areas were chosen because they lie on the iris edge.

The average value of the intensity of the pixels that were on either side of the iris edge in the areas of interest was used to generate the centers of the fuzzy membership functions.

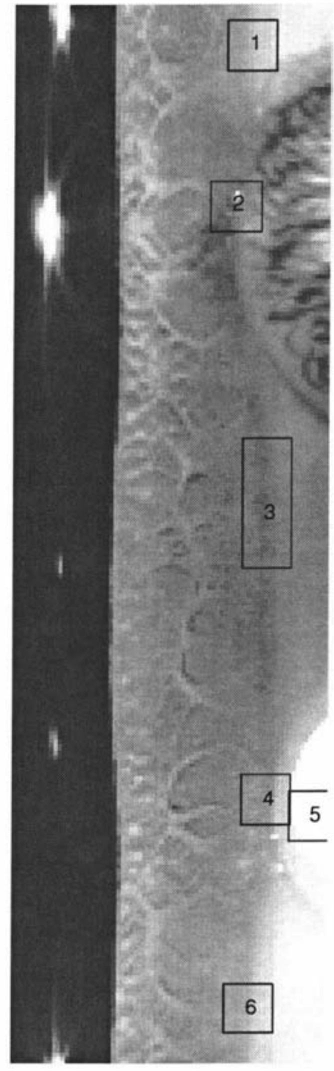


Figure 8.6 Input image showing areas of interest.

8.4.1.1 The fuzzification membership functions

Table 8.1 shows the fuzzification membership functions used to generate the rules for this experiment.

Membership function	Value
M1	0/255 = 0
M2	126/255 = 0.496
M3	129/255 = 0.508
M4	139/255 = 0.547
M5	162/255 = 0.637
M6	175/255 = 0.686

M7	185/255 = 0.727
M8	229/255 = 0.898
M9	255/255 = 1

Table 8.1 Fuzzification membership functions for experiment 8.4.1.

8.4.1.2 The window filter

Figure 8.7 shows the window filter placed over a section of the eye image.

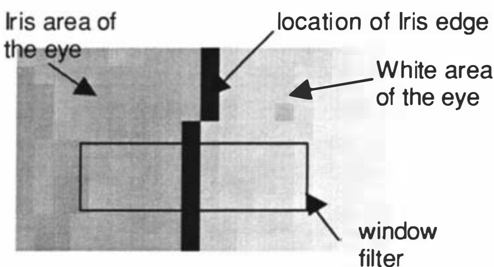


Figure 8.7 Shows the window filter placed over a section of the eye.

A 3x11 (height x width) window filter was used as it covers a section of the iris, the iris edge and the white part of the eye. The iris edge spans three pixels, four pixels were allocated to the iris section and another four were allocated to the white part of the eye. A total of eleven pixels were used as the width of the window. The first four columns from the left of the window were considered as the left of the window, the next three were considered as the middle of the window and the remaining four columns were considered as the right of the window. The rules generated were based on the left, middle and right of the window.

8.4.1.3 The rules

Two defuzzification membership functions were used in the rules, low and high.

Rule	Description
1	If I5=M2 I6=M2 I7=M2 I16=M2 I17=M2 I18=M2 I27=M2 I28=M2 I29=M2 I1=M3 I2=M3 I3=M3 I12=M3 I13=M3 I14=M3 I23=M3 I24=M3 I25=M3 I9=M4 I10=M4 I11=M4 I20=M4 I21=M4 I22=M4 I31=M4 I32=M4 I33=M4 Then output is high (if the middle=M4 and the left=m3 and the right=M4 then output=hi)

2	If I5=M7 I6=M7 I7=M7 I16=M7 I17=M7 I18=M7 I27=M7 I28=M7 I29=M7 I1=M6 I2=M6 I3=M6 I12=M6 I13=M6 I14=M6 I23=M6 I24=M6 I25=M6 I9=M8 I10=M8 I11=M8 I20=M8 I21=M8 I22=M8 I31=M8 I32=M8 I33=M8 Then output is high (If the middle=M7 and left=M6 and right=M8 then output=hi)
3	If I5=M6 I6=M6 I7=M6 I16=M6 I17=M6 I18=M6 I27=M6 I28=M6 I29=M6 I1=M5 I2=M5 I3=M5 I12=M5 I13=M5 I14=M5 I23=M5 I24=M5 I25=M5 I9=M8 I10=M8 I11=M8 I20=M8 I21=M8 I22=M8 I31=M8 I32=M8 I33=M8 Then output is high (if the middle=M6 and left=M5 and right=M8 then output=hi)

Table 8.2 Rules for experiment 8.4.1.

Each rule was generated from a different section of the eye image. The intensity of the pixels in the left, right and middle of the window was used to generate the rules. A quality factor of 0.5 and a saturation value of 1 were used for the above rules and a training time of 300 epochs was selected to train the network. These values were used because they were found to be appropriate when starting a new set of experiments on a new problem.

8.4.1.4 Network structure used

Layer	No. of nodes
Input layer	33 nodes (3x11 window)
Condition layer	9 membership functions for each input = 297 nodes
Rule layer	3 rules + 7 free = 10 nodes
Action layer	2 nodes
Output layer	1 node

Table 8.3 Network structure for experiment 8.4.1.

Figure 8.8 shows the output image generated by processing the input image with the trained network. The neural network performed poorly in detecting the iris edge. Table 8.4 shows the error between the target and output image.

Figure 8.8 Output image generated by processing the input image with the trained neural network.

False positives	238
False negatives	76
Error percentage (%)	35.3

Table 8.4 Errors between target and output image generated by the network.

Another network with the same structure was initialized with random numbers in the range {-1, +1} and trained for 300 epochs. The output image generated by network initialized with random numbers was similar to the image in Figure 8.8. The errors for this image were very close to those in Table 8.4. Network training was not improved by the use of the rules in section 8.4.1.3.

Several other experiments were performed to investigate methods of detecting the iris edge. In these experiments the membership functions remained the same and the rules and the window size was varied. The rules for these experiments were developed using the left, right and middle section of the window similar to that of section 8.4.1.3. In all of these experiments the trained network performed unsatisfactorily. From experience it was thought that the irregularities in the

illumination in the input image were causing the network to perform poorly. The network performance improved after the input image was processed using an illumination correction procedure [Matlab IPT].

8.5 The new training images

Figure 8.9 shows the new training images for the iris edge detection problem. Image (a) is the original eye image. An illumination correction process was used on image (a) to generate image (b). The illumination correction process improved the distinction between the iris and the white area of the eye. Image (b) was used as the input image for training. Image (c) is the target image. The iris edge in the target image was drawn by hand. As a first estimate an iris edge with a width of one pixel was first drawn and then the image was blurred using a Gaussian filter. The position of the pixels in the iris edge drawn by hand has inherent displacement errors. By blurring the image a range of possible iris edge positions were generated. Figure 8.9 (b) and (c) are used as input images for the experiments in the remainder of the chapter.

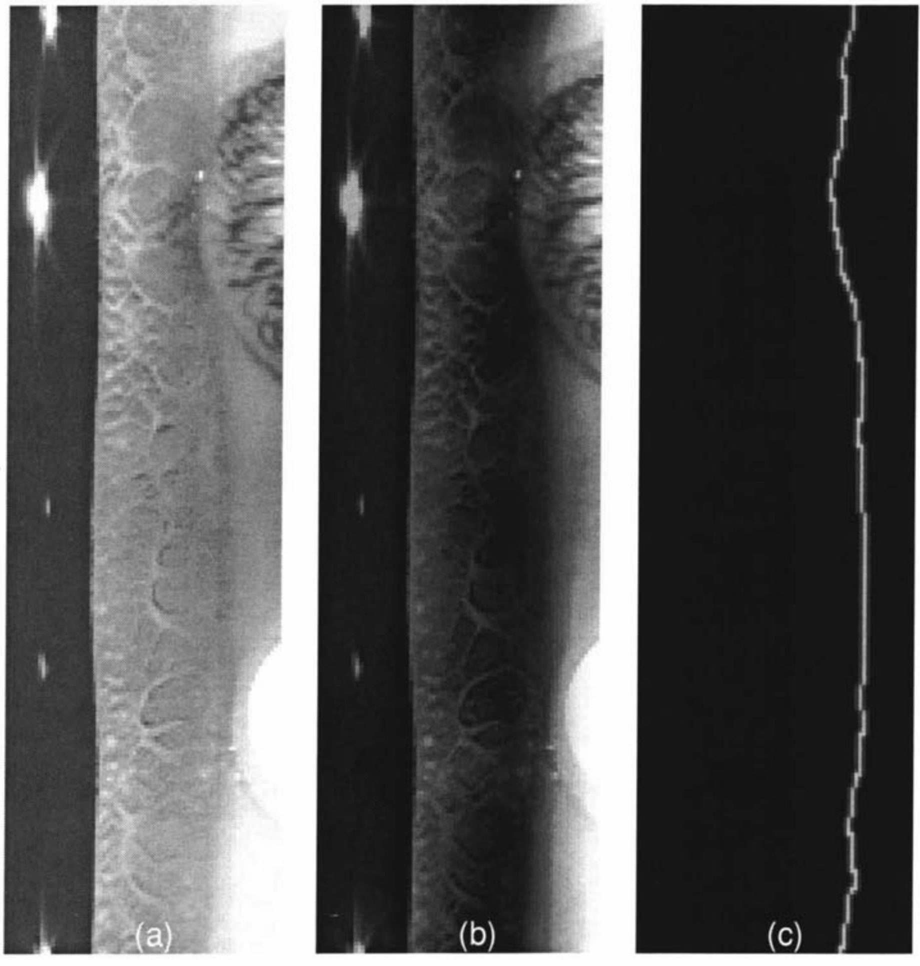


Figure 8.9 Training images for iris detection problem. Image (a) is the original eye image, image (b) was generated by passing an illumination correction process on (a) and is used as the input image and image (b) is the target image.

8.6 Generation of the new membership functions

Before training the FuNN, rules were developed that describe the problem. Figure 8.10 shows the input image used for training. The numbers marked on the input image represent the areas of interest that were used to generate the fuzzification membership functions. Table 8.5 shows the average pixel values around the markers.

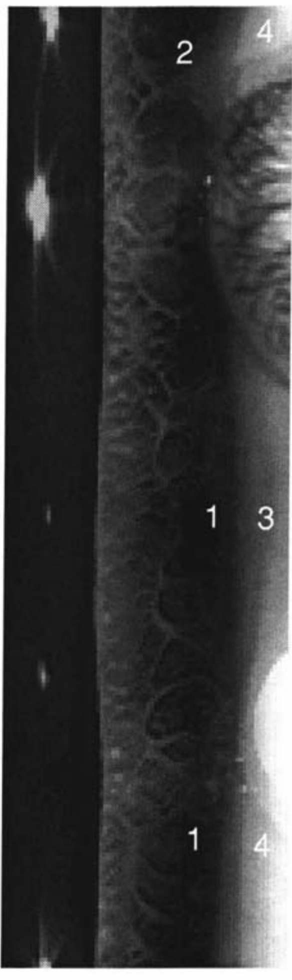


Figure 8.10 Input image displaying areas of interest to generate fuzzification membership functions.

Marker Number	Average pixel value around marker
1	10
2	40
3	80
4	160

Table 8.5 Average pixel values around areas of interest.

8.6.1 Membership functions for fuzzification process

Table 8.6 shows the membership functions that were used for the fuzzification process for this problem. The membership functions were derived from the values in Table 8.1.

Membership function	Value
M1	0
M2	10/255 = 0.039
M3	40/255 = 0.157
M4	80/255 = 0.313
M5	160/255 = 0.62
M6	1

Table 8.6 Fuzzification membership functions used in iris edge detection problem.

8.6.2 Membership functions for defuzzification process

Table 8.7 shows the membership functions used in the defuzzification process of this problem. There are only two states, iris edge *found* and *not found*.

Membership function	Value
M1	0 (iris edge not found)
M2	1 (iris edge found)

Table 8.7 Defuzzification membership functions used in iris edge detection problem.

8.7 The experiments

8.7.1 Experiment to find the iris edge using a FuNN initialized with rules.

In experiment 8.4.1 a 3x11 window filter was used. In this experiment a 3x7 window filter was chosen as this is the minimum size where an intensity transition indicating an iris edge could be noticed.

The membership functions in 8.6.1 and 8.6.2 where used in the rule generation process. The rules were based on the areas of interest in Figure 8.10. The rules represent an intensity transition from area two to four, area one to three and area one to four. The rules are,

Rule	Description
1	If I1=M2 I2=M2 I8=M2 I9=M2 I15=M2 I16=M2 I6=M3 I7=M3 I13=M3 I14=M3 I20=M3 I21=M3 Then O1=M2 (If the intensity of the left side of the window is of M2 and the intensity of the right side of the window is of M3 then output is high – iris edge found)
2	If I1=M2 I2=M2 I8=M2 I9=M2 I15=M2 I16=M2 I6=M4 I7=M4 I13=M4 I14=M4 I20=M4 I21=M4 Then O1=M2 (If the intensity of the left side of the window is of M2 and the intensity of the right side of the window is of M4 then output is high – iris edge found)
3	If I1=M2 I2=M2 I8=M2 I9=M2 I15=M2 I16=M2 I6=M5 I7=M5 I13=M5 I14=M5 I20=M5 I21=M5 Then O1=M2 (If the intensity of the left side of the window is of M2 and the intensity of the right side of the window is of M5 then output is high – iris edge found)
4	If I1=M4 I2=M4 I8=M4 I9=M4 I15=M4 I16=M4 I6=M2 I7=M2 I13=M2 I14=M2 I20=M2 I21=M2 Then O1=M1 (If the intensity of the left side of the window is of M4 and the intensity of the right side of the window is of M2 then output is low – pupil edge found)

Table 8.8 Rule for experiment 8.7.1.

The above four rules were used to initialize the network. A quality factor of 0.5 and a saturation value of 1 were used as these values were found to be appropriate when a new set of rules are used.

8.7.1.1 Network structure used

Layer	No. of nodes
Input layer	21 nodes (3x7 window)
Condition layer	6 membership functions for each input = 126 nodes
Rule layer	4 rules + 6 free = 10 nodes
Action layer	2 nodes
Output layer	1 node

Table 8.9 Network structure for experiment 8.7.1.

Another network with the same structure was initialized with random numbers in the range {-0.1,+0.1} and trained. The performance of the network initialized with rules was compared to that of the network initialized with random numbers. Both networks were trained for 300 epochs. After training the input image was processed using the trained network. Table 8.10 shows the error between the target and output images.

	Network with rules	Network without rules
False positives	86	69
False negatives	51	50
Error percentage (%)	19.5	18.2

Table 8.10 Error between target and output images for experiment 8.7.1.

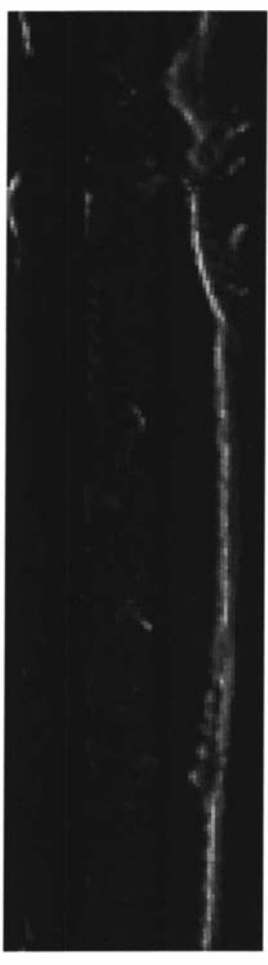


Figure 8.11 Output generated from the trained network for experiment 8.7.1

Table 8.10 shows the network initialized without rules performed better than the network initialized with rules. The performance of the network with rules is better than the performance of the network in experiment 8.4.1. The illumination correction procedure has helped improve network performance.

8.7.2 Experiment to find the iris edge using a FuNN using a larger window than in experiment 8.7.1

This experiment is a continuation of the previous experiment. The size of the window was increased to 3x9. On closer inspection of the eye image it was observed that the area adjacent to the left of the iris edge had an average pixel value of 19. So a new membership function was added to the existing set. The value of the new membership function was 0.078. The membership functions after the new addition were [0, 0.039, 0.078,0.157, 0.3125, 0.625, 1]. These membership functions were used to generate the rules.

Rule	Description
1	If I1=M2 I2=M2 I10=M2 I11=M2 I19=M2 I20=M2 I8=M4 I9=M4 I17=M4 I18=M4 I26=M4 I27=M4 Then O1=M2 (If the intensity of the left side of the window is of M2 and the intensity of the right side of the window is of M4 then output is high – iris edge found)
2	If I1=M2 I2=M2 I10=M2 I11=M2 I19=M2 I20=M2 I8=M5 I9=M5 I17=M5 I18=M5 I26=M5 I27=M5 Then O1=M2 (If the intensity of the left side of the window is of M2 and the intensity of the right side of the window is of M5 then output is high – iris edge found)
3	If I1=M2 I2=M2 I10=M2 I11=M2 I19=M2 I20=M2 I8=M6 I9=M6 I17=M6 I18=M6 I26=M6 I27=M6 Then O1=M2 (If the intensity of the left side of the window is of M2 and the intensity of the right side of the window is of M6 then output is high – iris edge found)
4	If I1=M2 I2=M2 I10=M2 I11=M2 I19=M2 I20=M2 I8=M7 I9=M7 I17=M7 I18=M7 I26=M7 I27=M7 Then O1=M2 (If the intensity of the left side of the window is of M2 and the intensity of the right side of the window is of M7 then output is high – iris edge found)

5	If I1=M3 I2=M3 I10=M3 I11=M3 I19=M3 I20=M3 I8=M5 I9=M5 I17=M5 I18=M5 I26=M5 I27=M5 Then O1=M2 (If the intensity of the left side of the window is of M3 and the intensity of the right side of the window is of M5 then output is high – iris edge found)
6	If I1=M5 I2=M5 I10=M5 I11=M5 I19=M5 I20=M5 I8=M2 I9=M2 I17=M2 I18=M2 I26=M2 I27=M2 Then O1=M1 (If the intensity of the left side of the window is of M5 and the intensity of the right side of the window is of M2 then output is low – pupil edge found)

Table 8.11 Rule for experiment 8.7.2.

The above six rules were used to initialize the network. A quality factor of 0.5 and a saturation value of 1 were again used for the rules.

8.7.2.1 Network structure used

Layer	No. of nodes
Input layer	27 nodes (3x9 window)
Condition layer	7 membership function for each node = 189 nodes
Rule layer	6 rules + 4 free nodes = 10 rule nodes
Action layer	2 nodes
Output layer	1 node

Table 8.12 Network structure for experiment 8.7.2.

Another network with the same structure was initialized with random numbers {-0.1, +0.1} and trained. The performance of the network initialized with rules was compared to that of the network initialized with random numbers. Both networks were trained for 300 epochs. The input image was processed using the trained network. Table 8.13 shows the error between the target and output images.

	Network with rules	Network without rules
False positives	72	128
False negatives	27	16
Error percentage (%)	11.8	11.9

Table 8.13 Error between target and output images for experiment 8.7.2.

Figure 8.12 shows the output image generated by processing the input image using the trained network initialized with rules.

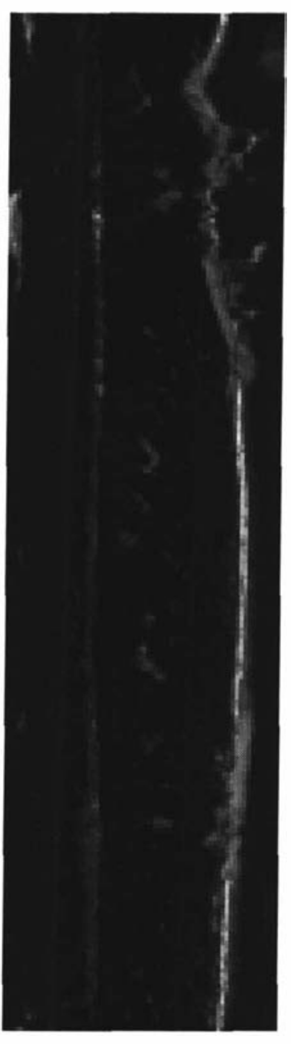


Figure 8.12 Output image generated by the trained network initialized with rules for experiment 8.7.2.

Table 8.13 shows the number of false positives for the network initialized with rules is lower than the network initialized with no rules. However the final error for the network with and without rules in almost the same.

8.7.3 Experiment to find the iris edge using a FuNN using a greater number of free nodes compared to experiment 8.7.2

This experiment is a continuation of experiment 8.7.2. The size of the window was not changed. The iris edge near the eyelashes has not been detected as clearly as the iris edge in the remainder of the image. This experiment was performed to determine if adding a new rule representative of the area near the eyelashes would improve the performance of the network. So a new rule was added to the existing rules and the experiment was rerun. The new rule was,

Rule	Description
7	If I1=M3 I2=M3 I10=M3 I11=M3 I19=M3 I20=M3 I8=M4 I9=M4 I17=M4 I18=M4 I26=M4 I27=M4 Then O1=M2 (If the intensity of the left side of the window is of M3 and the intensity of the right side of the window is of M4 then output is high – iris edge found)

Table 8.14 New rule added for experiment 8.7.3.

The rules were used to initialize the network. A quality factor of 0.5 and a saturation value of 1 were again used for the rules.

8.7.3.1 Network structure used

Layer	No. of nodes
Input layer	27 nodes (3x9 window)
Condition layer	7 membership functions for each input = 189 nodes
Rule layer	7 rules + 9 free = 16 nodes
Action layer	2 nodes
Output layer	1 node

Table 8.15 Network structure used for experiment 8.7.3.

Another network with the same structure was initialized with random numbers in the range {-0.1,+0.1} and trained. The performance of the network initialized with rules was compared to that of the network initialized with random numbers. Both networks were trained for 300 epochs. After training the input image was

processed using the trained network. Table 8.16 shows the error between the target and output images.

	Network with rules	Network without rules
False positives	275	289
False negatives	42	40
Error percentage (%)	27.7	27.9

Table 8.16 Error between target and output images for experiment 8.7.3.

As the table shows the number of false positives for the network initialized with rules is lower than the network initialized with no rules. However the final error for the network with and without rules in almost the same.

Table 8.17 shows the training RMS error for both networks. The network initialized with rules has started from a lower RMS error compared to the network initialized without rules.

	Starting RMS	RMS error after 300 epochs
Network initialized with rules	0.19	0.018
Network initialized without rules	0.308	0.016

Table 8.17 Training RMS error for experiment 8.7.3.

Figure 8.13 shows the output images generated from the trained network. The trained network initialized with rules was used to process the input image to generate image (a-1). Then image (a-1) was thresholded and thinned to produce image (a-2). The threshold value was obtained through an iterative process. The trained network initialized without rules was used to generate image (b-1). Then image (b-1) was thresholded and thinned to produce image (b-2).

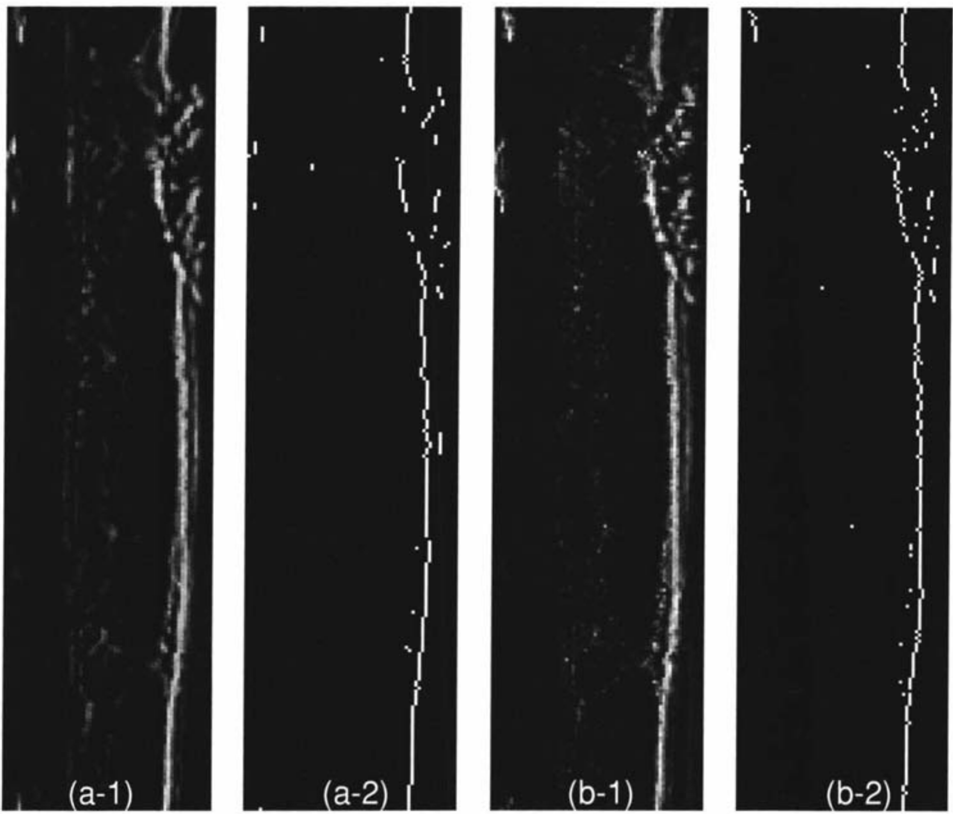


Figure 8.13 Network output for experiment 8.7.3. Image (a-1) was generated by the network initialized with rules, image (a-2) was generated by thresholding and thinning image (a-1). Image (b-1) was generated by the network initialized with no rules and image (b-2) was generated by thresholding and thinning image (b-1).

Figure 8.13 shows that the network did not detect the iris edge near the eyelashes. Several other experiments were performed using experiment 8.7.3 as a basis to try to generate a FuNN to detect the entire iris edge. The iris edge not being detected near the eyelashes seemed to be a common problem in all the other experiments that were performed. So a profile plot across sections along the iris edge was generated to determine if the image was homogeneous. Figure 8.14 shows an eye image with profile plots from different sections along the iris edge.

Figure 8.14 (a), (c) and (d) have the same profile plots, (b) is different. This shows that the area adjacent to the iris edge is not homogeneous. This could be a reason why the network could not detect the iris edge near the eyelashes.

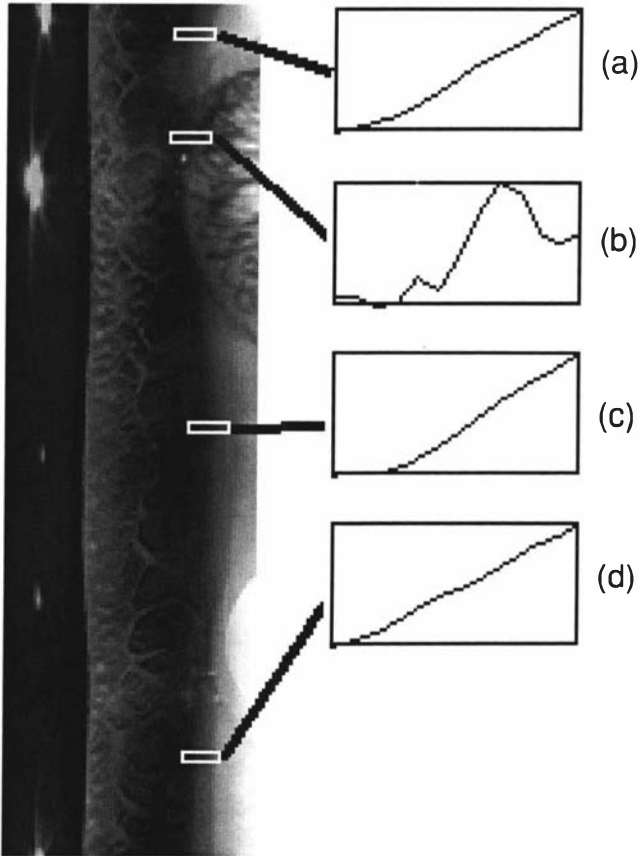


Figure 8.14 Profile plots of different sections along the iris edge.

The next experiment was to determine if a FuNN could detect the iris edge near the eyelashes.

8.7.4 Experiment to train a network to detect the iris edge near the eyelashes

This experiment was performed to determine if a FuNN could detect the iris edge near the eyelashes. The entire iris edge could not be detected because the area adjacent to the iris edge is not homogeneous. Therefore the problem was separated into two parts. One part to detect the iris edge near the eyelashes and the other part to detect the remainder of the iris edge.

8.7.4.1 Fuzzification membership function for experiment 8.7.4

Table 8.18 shows the new set of fuzzification membership functions generated for the area near the eyelashes.

Membership function	Value
M1	0
M2	0.019
M3	0.078
M4	0.195
M5	0.351
M6	1

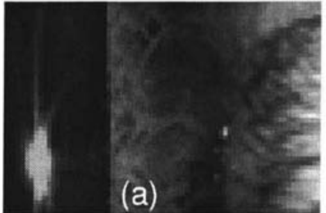
Table 8.18 Fuzzification membership functions used for the area near the eyelashes.

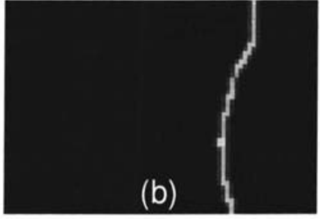
The new membership functions were used to generate the rules.

Rule	Description
1	If I1=M3 I2=M3 I10=M3 I11=M3 I19=M3 I20=M3 I8=M4 I9=M4 I17=M4 I18=M4 I26=M4 I27=M4 Then O1=M2
2	If I1=M2 I2=M2 I10=M2 I11=M2 I19=M2 I20=M2 I8=M4 I9=M4 I17=M4 I18=M4 I26=M4 I27=M4 Then O1=M2
3	If I1=M1 I2=M1 I10=M1 I11=M1 I19=M1 I20=M1 I8=M5 I9=M5 I17=M5 I18=M5 I26=M5 I27=M5 Then O1=M1
4	If I1=M5 I3=M5 I5=M5 I7=M5 I9=M5 I11=M5 I13=M5 I15=M5 I17=M5 I19=M5 I21=M5 I23=M5 I25=M5 I27=M5 Then O1=M1
5	If I1=M1 I3=M1 I5=M1 I7=M1 I9=M1 I11=M1 I13=M1 I15=M1 I17=M1 I19=M1 I21=M1 I23=M1 I25=M1 I27=M1 Then O1=M1
6	If I1=M5 I2=M5 I10=M5 I11=M5 I19=M5 I20=M5 I8=M3 I9=M3 I17=M3 I18=M3 I26=M3 I27=M3 Then O1=M1

Table 8.19 Rules used for experiment 8.7.4.

A saturation value of one and a quality factor of 0.5 were again used for the rules. The network was initialized with the rules and trained for 200 epochs. Figure 8.15 shows the training images used to train the network and the network output. Figure 8.15 (a) shows the input image and (b) shows the target image used to train the network. Figure 8.15 (c) shows the output image generated when the trained network was used to process input image (a).





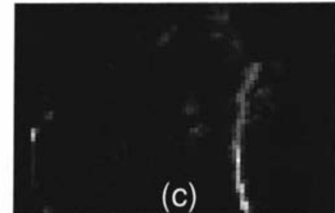


Figure 8.15 The training images and network output for experiment 8.7.4. Images (a) and (b) were the training images and image (c) was the network output.

Figure 8.15 (c) shows that the network detected the iris edge near the eyelashes. In the previous chapters more than one neural network was used to solve a single problem. Similarly, more than one network can be used to detect the iris edge. Experiment 8.7.3 showed that a FuNN could detect the iris edge excluding the area near the eyelashes. Experiment 8.7.4 showed that a FuNN could detect the iris edge near the eyelashes. If these two networks were combined the entire iris edge could be detected. The next experiment will show if two neural networks working on different areas of the image can be used to detect the entire iris edge.

8.7.5 Use of two neural networks to detect the iris edge

Experiment 8.7.4 showed how the network detected the iris edge near the eyelashes. Experiment 8.7.3 showed how the iris edge excluding the area near the eyelashes was detected. This experiment was performed to determine if two neural networks processing different areas of the image could detect the entire iris edge.

For this experiment the input image was segmented into two parts. Figure 8.16 shows the segments in the image. Section 1 is the section near the eyelashes and section 2 is the remainer of the image. Figure 8.16 has dimensions 75x256 pixels (width x height). Section 1 starts from point (1,30) and has a height of 50 pixels and a width of 75 pixels (point (0,0) starts from the top left hand corner of the input image). The two sections were processed using two separate networks. The network trained in experiment 8.7.4 was used to process section 1. The network trained in experiment 8.7.3 was used to process section 2.

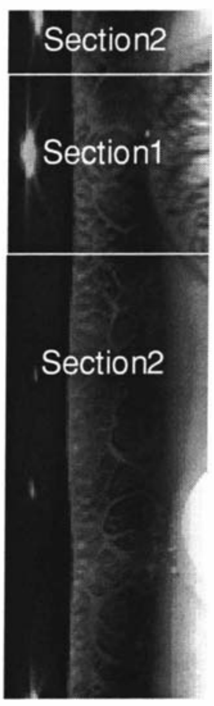


Figure 8.16 Segmented input image showing section 1 and section 2.

Figure 8.17 (a) shows section 1 from the input image. Figure 8.17 (b) shows the output generated by processing image (a) using the trained network in experiment 8.7.4. The iris edge is clearly defined in the output image.

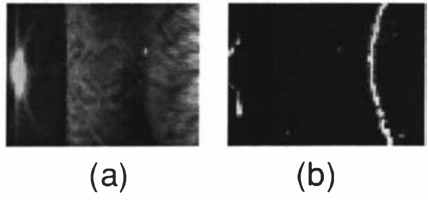


Figure 8.17 Image (a) is section 1 from the input image and image (b) is the output generated by the network.

Since the network in experiment 8.7.3 is used to detect the iris edge in section 2, the area near the eyelashes was excluded from the training set and the network was retrained. Figure 8.18 (a) shows the input image. Figure 8.18 (b) shows the output image generated by processing the input image using the trained network.

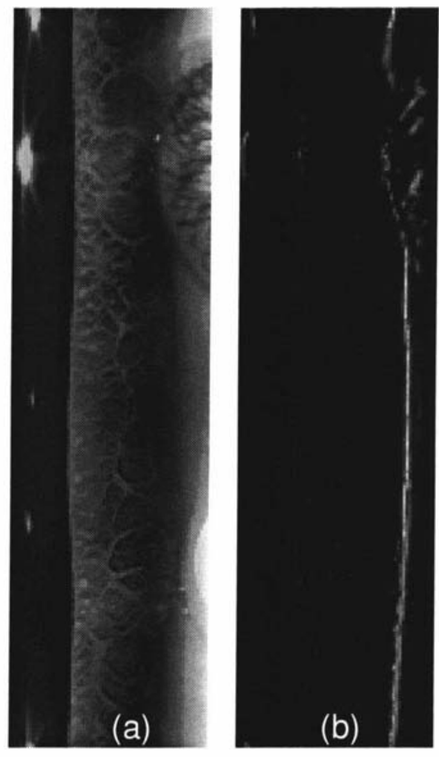


Figure 8.18 Image (a) is the input image and image (b) is the output generated for section 2.

Image 8.17 (b) was inserted into image 8.18 (b) to generate the final image. Figure 8.19 (a) shows the combined image. Figure 8.19 (b) was generated using a post process procedure.

8.7.5.1 Post process procedure

Figure 8.19 (a) was post processed using the following operations,

1. Threshold network output using a threshold value of 20. This value was generated through an iterative process.
2. Skeletonize the thresholded image (the skeletonize function is a built in Matlab command).
3. Generate the negative of the skeletonized image and use the least cost path algorithm to generate the final iris edge.

8.7.5.2 Least Cost Path Algorithm[†]

1. The foreground pixels are given a weighting of 0 and the background pixels are given a weighting of 255.
2. A penalty function is added to each pixel in each row (processing done row by row). In this case a penalty function of [9 4 1 0 1 4 9].

   $E[r,c] =$

   $E[r,c] + Min\{ E[r\text{-}1,j], E[r\text{-}1,j\text{-}1]+1, E[r\text{-}1, j+1]+1], E[r\text{-}1,j\text{-}2]+4, E[r\text{-}1,j+2]+4, E[r\text{-}1,j\text{-}3]+9, E[r\text{-}1,j+3]+9\}$

   Where

   r {1..height of image},

   c {1..width of image}.
3. The pixel with the smallest value at the bottom of the image is used as a starting point to traverse upwards, finding the least cost path through the image.

Figure 8.19 Image (a) is the combined output. Image (b) was generated by post processing image (a). Image (b) was superimposed on the input image to generate image (c).

[†] The least cost algorithm was developed by Dr. D. G. Bailey, Institute of Information Sciences & Technology, Massey University.

Figure 8.19 shows that the iris edge can be detected by using two neural networks.

The above procedure to detect the iris edge was repeated. Random numbers in the range {-1, +1} were used to initialize each network prior to training. Each network was trained for 300 epochs. The network trained to detect the iris edge in section 1 was used to process section 1 and the network trained to detect the iris edge in section 2 was used to process section 2. The outputs from network one and two were combined and post processed to generate the final output. Figure 8.20 (a) shows the combined output from the two networks. Figure 8.20 (b) shows the combined output post processed using the procedure described in 8.7.5.1. Figure 8.20 (c) show the post processed image superimposed on the eye image.

Figure 8.20 Image (a) is the combined output from the two networks. Image (a) was post processed to generate image (b). Image (b) was superimposed on the input image to generate image (c).

Table 8.20 shows the errors between the target and output images generated by networks initialized with and without rules.

		Network with rules	Network without rules
Before post process	False positives	68	193
	False negatives	44	31
	Error (%)	16.5	19.9
After post process	False positives	38	80
	False negatives	34	61
	Error	11.8	22

Table 8.20 Error between the target and output image for experiment 8.7.5.

Table 8.20 shows that the network initialized with rules has a lower error compared to the network initialized with random numbers.

8.7.6 Operational test for iris detection problem using two neural networks

This experiment was performed to determine the operational performance of the neural networks on three new eye images.

As in experiment 8.7.5 two neural networks were used to process the input eye image. All eye images have dimensions 75x256 (width x height). Figure 8.21 shows the original input images used for the operational test. Each eye image was pre-processed using an illumination correction process described in section 8.5. Figure 8.22 shows the pre-processed images. The trained neural network in experiment 8.7.4 was used to process the area near the eyelashes, i.e. the area from point (1,30) to (75,80) in the eye image. The retrained neural network in experiment 8.7.3 was used to process the remainder of the image. The outputs from both networks were combined as described in experiment 8.7.5. The combined output was post processed using the method described in section 8.7.5.1.

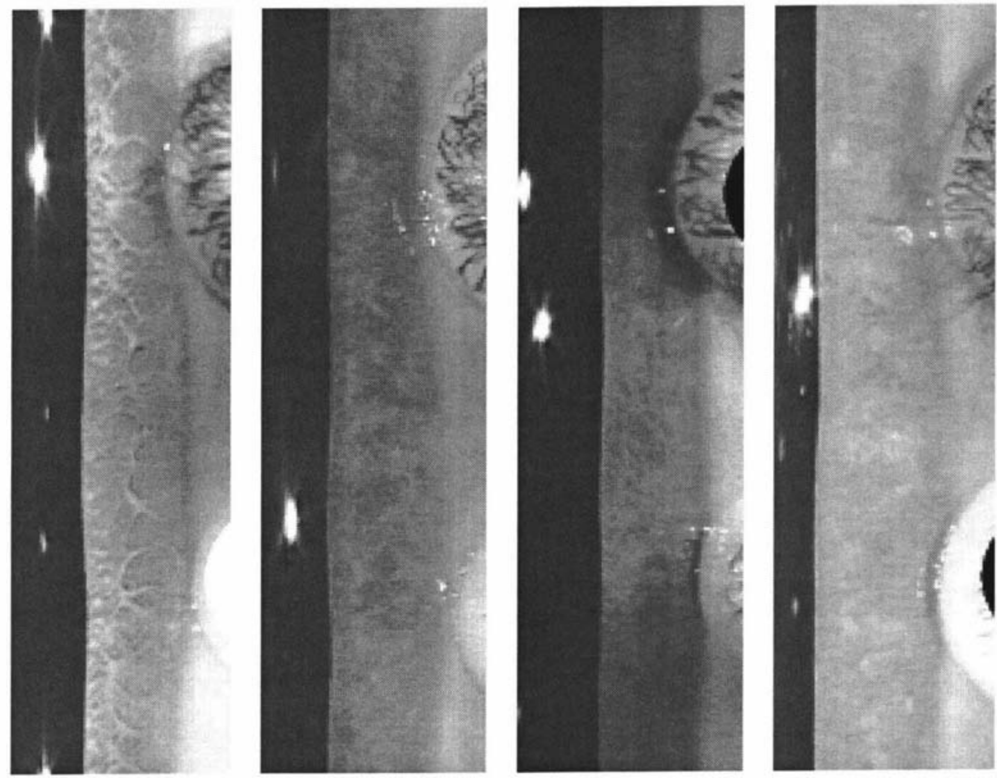


Figure 8.21 New eye images for operational test.

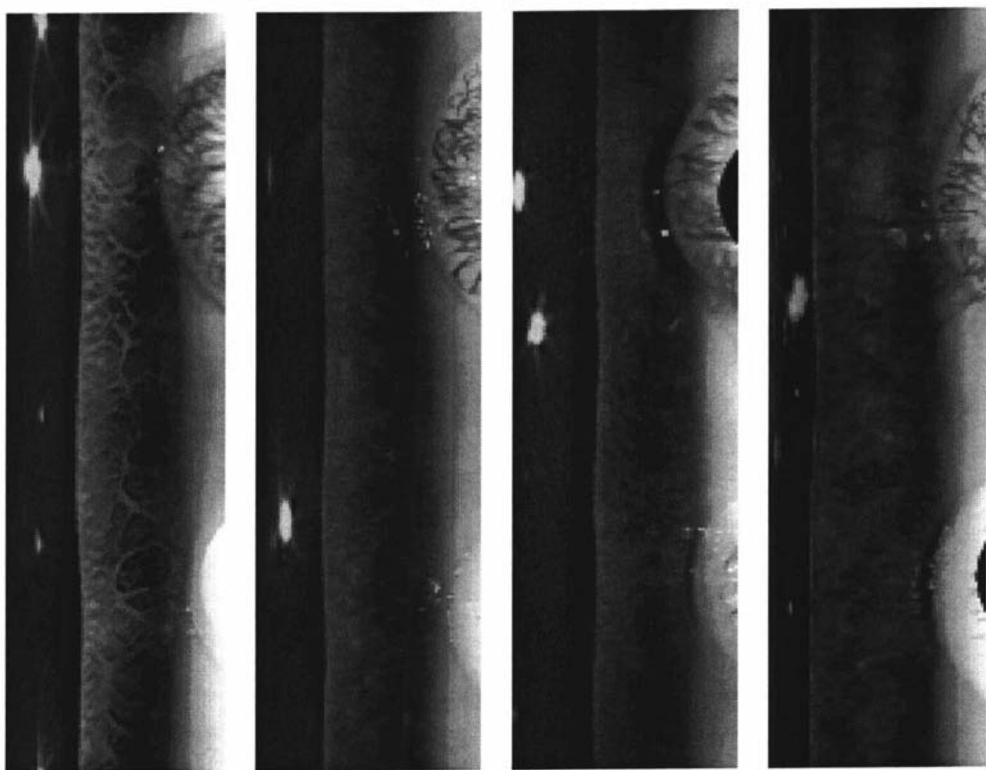


Figure 8.22 Eye images in figure 8.21 post processed.

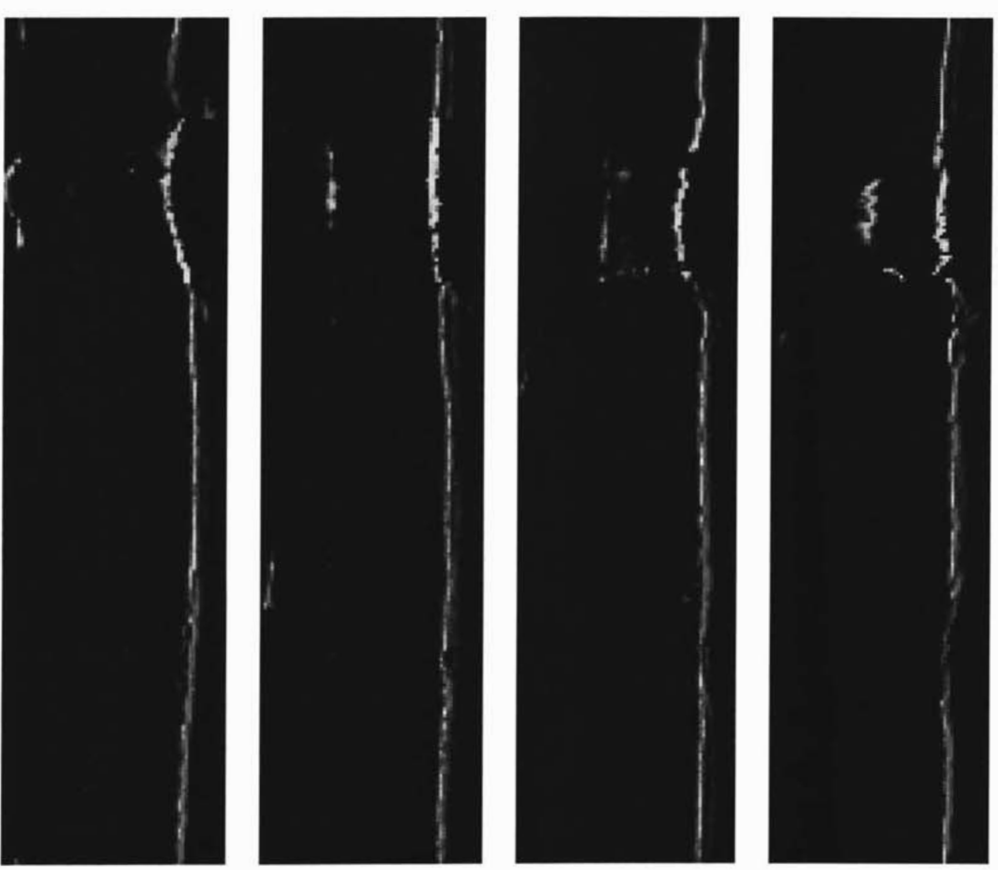


Figure 8.23 Eye images in figure 8.22 processed using network one and network two.

Figure 8.24 Images in figure 8.23 post processed.

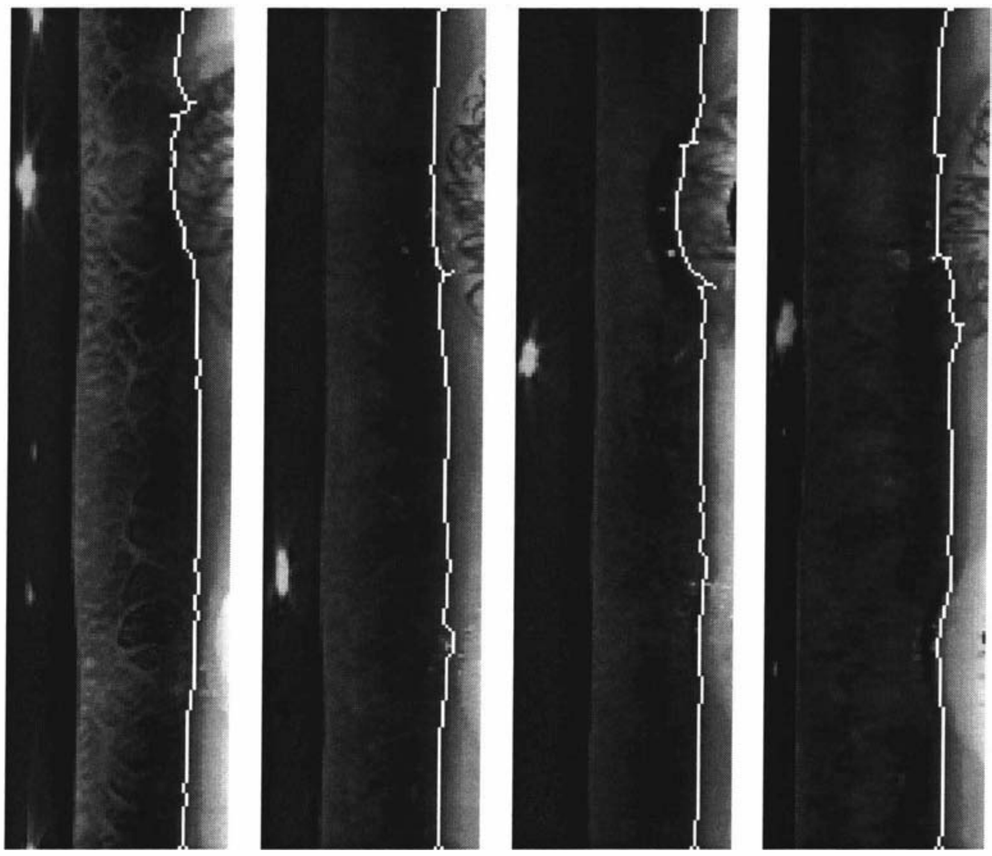


Figure 8.25 Post processed network output overlaid over input images.

The error measured between the target and output images is not shown as the target images for the eye images in figure 8.21 was not available.

8.7.7 Experiment to determine if the least cost path algorithm in the post process procedure could be replaced by a neural network

The post process procedure described in section 8.7.5.1 is used to cleanup the output image generated by processing the input image using the trained FuNN. This experiment was performed to determine if a FuNN could be trained to perform the cleanup process.

The output images generated by the trained networks in experiment 8.7.5 produced and iris edge having an average width of five pixels. The iris edge is also predominantly vertical. Therefore a long thin window filter can be used for this

problem. So an 11x5 (height x width) window filter was chosen for this experiment.

Figure 8.26 shows the output image generated by combining the output from the two networks in experiment 8.7.5. The squares on the image show the areas of interest. The average values of the pixels in these areas will be used as the centers of the membership functions that will be used to generate the rules.

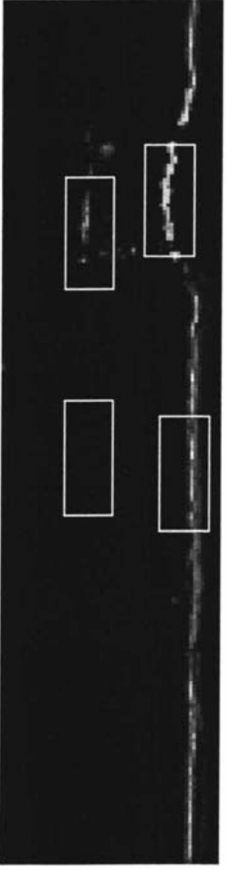


Figure 8.26 Input image showing areas of interest.

8.7.7.1 The fuzzification membership functions used

Membership function	Value
M1	0
M2	0.0977
M3	1

Table 8.21 Membership functions used for experiment 8.7.7.

The membership functions were used to generate the rules.

Rule	Description
1	If I3>M2 I8>M2 I13>M2 I18>M2 I23>M2 I28>M2 I33>M2 I38>M2 I43>M2 I48>M2 I53>M2 Then O1=M2
2	If I23>M2 I28>M2 I33>M2 I27>M2 I29>M2 Then O1=M1
3	If I1>M2 I2>M2 I6>M2 I7>M2 I12>M2 I55>M2 I54>M2 I49>M2 I50>M2 I44>M2 Then O1=M2
4	If I4>M2 I5>M2 I9>M2 I10>M2 I14>M2 I42>M2 I46>M2 I47>M2 I51>M2 I52>M2 Then O1=M2
5	If I19>M2 I20>M2 I24>M2 I25>M2 I31>M2 I32>M2 I36>M2 I37>M2 Then O1=M2
6	If I16>M2 I17>M2 I21>M2 I22>M2 I34>M2 I35>M2 I39>M2 I40>M2 Then O1=M2

Table 8.22

A saturation value of 1 and a quality factor of 0.5 were again used for the rules.

8.7.7.2 The network structure used

Layer	No. of nodes
Input layer	55 nodes (11x5 window)
Condition layer	3 membership functions for each input = 165 nodes
Rule layer	6 rules + 9 free = 15 nodes
Action layer	2 nodes
Output layer	1 node

Table 8.23

The network was trained for 200 epochs. The trained network was then used to process the image in Figure 8.26.

Figure 8.27 (a) shows the image generated by the two networks in experiment 8.7.5 to detect the iris edge. Figure 8.27 (b) shows the output image generated by processing the input image using the trained network.

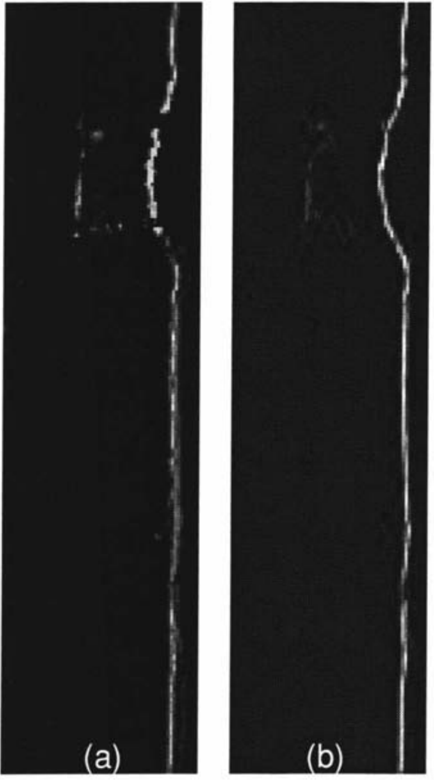


Figure 8.27 Image (a) is the input image and image (b) is output image generated by processing image (a) using the trained network.

The trained network has cleaned up the input image and connected some gaps in the iris edge. But it has not performed as well as the post process procedure that was described in section 8.7.5.1.

8.7.8 Experiment to determine the weight change of the rules during training

This experiment was performed to observe the change of the magnitude of the weights connected to the nodes in the rule layer during training. The hypothesis is that the rule node weights should change less than the free nodes as the rules nodes embody problem knowledge. A saturation value of one was used for the rules, as this was the standard value used in previous similar experiments for other problems. The quality factor for the rules was varied between 0.1 and 1 (0.1, 0.25,

0.5, 0.75, 1). Seven rules were used for this experiment. Five runs were performed, each time the quality factor for the rules was changed and the network was trained for 400 epochs.

8.7.8.1 Cumulative weight change

The cumulative weight change is the Euclidean distance between the initial weight matrix and the weight matrix at the current epoch. It is measured for every node for every epoch in the rule layer.

8.7.8.2 The network structure used

Layer	No. of nodes
Input layer	27 nodes (3x9 window)
Condition layer	7 membership functions for each input = 189 nodes
Rule layer	7 rules + 9 free = 16 nodes
Action layer	2 nodes
Output layer	1 node

Table 8.24 Network structure for experiment 8.7.8.

Figure 8.28 shows the cumulative weight change graphs for the rule and free nodes of the network when a quality factor of 0.5 was used. For clarity only rules one, two and seven are shown in the weight change graph. The weight change for rule one is very close to the horizontal axis. When the quality factor for the rule nodes was 0.5 the weight change for rules three, four, five and six were similar to that of rule one. Figure 8.28 shows that rule two and seven have changed more than the other rules. Except for rule two and seven the other rules have changed less than the free nodes. This suggests that rule two and seven may be a poor representation of problem knowledge. Table 8.25 shows the error between the target and output image of the network for a range of quality factors. The error is the lowest when Q=0.25.

Quality factor	False positives	False negative	Error (%)
0.1	159	45	21.9
0.25	145	37	18.8
0.5	176	31	18.9
0.75	176	35	20
1	187	35	20.7

Table 8.25 The table shows the errors of the network for a range of quality factors.

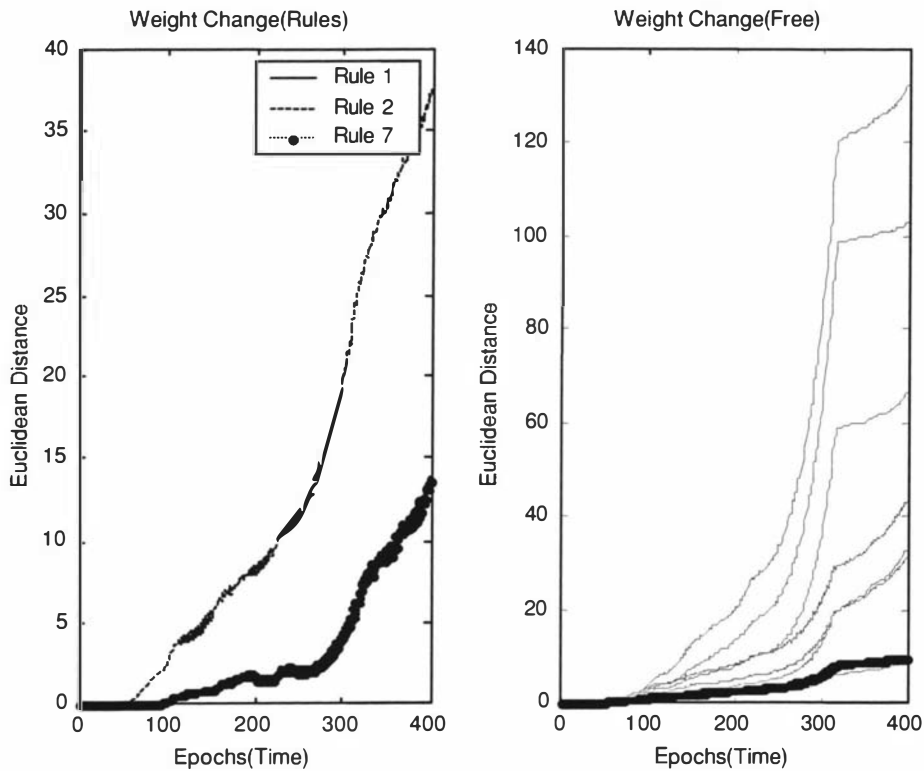


Figure 8.28 The cumulative weight change graph for the network when Q=0.5. The weight change (rules) graph shows the cumulative weight change for the rule nodes and the weight change (free) graph shows the cumulative weight change for the free nodes.

The next experiment involves removing rule two and seven and rerunning the experiment to determine the cumulative weight change.

8.7.9 Experiment to determine if removing rule 2 and 7 from the rule set from experiment 8.7.8 would improve the network performance

This is a continuation of the previous experiment. Rule two and seven from the rule set of experiment 8.7.8 was removed and the experiment was rerun. The

quality factor was again varied between 0.1 and 1 (0.1, 0.25, 0.5, 0.75 and 1). The saturation value was maintained at a value of 1 and the network was trained for 400 epochs. The cumulative weight change for the network was recorded.

Figure 8.29 shows the cumulative weight change of the rule and free nodes of the network when a quality factor of 0.5 was used. Rule 2 and 4 are almost parallel to the horizontal axes.

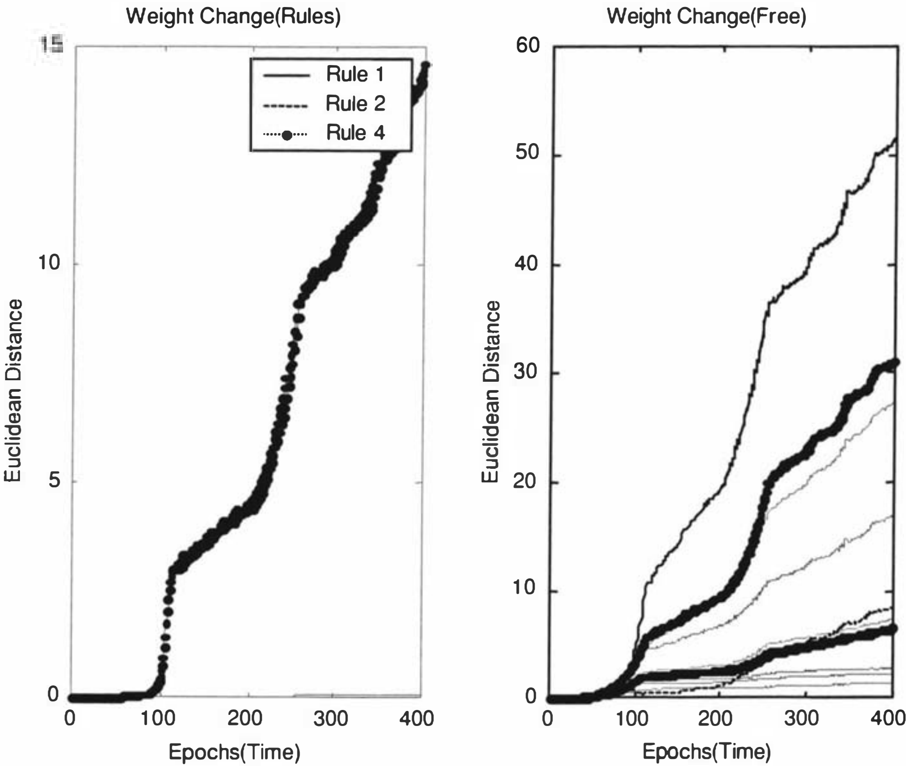


Figure 8.29 The results of the cumulative weight change for experiment 8.7.9 using a quality factor of 0.5. The weight change (rules) graph shows the cumulative weight change for the rule nodes and the weight change (free) graph shows the cumulative weight change for the free nodes.

Figure 8.29 shows that the cumulative weight change for rule four which is rule five from the original rule set has changed more than the other rules. Table 8.26 shows the error between the target and output image for a range of quality factors.

Quality factor	False positives	False negative	Error
0.1	219	40	23.9
0.25	255	30	23.1
0.5	248	36	24.5
0.75	279	40	27.4
1	121	40	18.6

Table 8.26 The table shows the errors of the network for a range of quality factors.

By removing rule two and seven the performance of the network was poorer when the quality factor was between 0.1 and 0.75 compared to experiment 8.7.8, but the performance improved when the quality factor was 1.

Rule seven was removed from the original rule set and the experiment was rerun. Table 8.27 shows the error between a range of quality factors.

Quality factor	False positives	False negative	Error (%)
0.1	224	37	23.4
0.25	285	44	25.8
0.5	284	42	28.2
0.75	284	42	28.2
1	284	42	25.8

Table 8.27 The table shows the errors of the network for a range of quality factors when rule seven was removed and the experiment rerun.

The errors in Table 8.27 are higher than the errors in Table 8.26. Removing rule seven has not improved network performance. Several other experiments were performed that involved removing different combinations of rule two, five, six and seven. These experiments produced results that were similar to the above results.

8.8 Conclusions

- This chapter investigated the use of a fuzzy neural network to solve an industrial problem. The chapter showed how the network gave a poor solution when only one neural network was used to detect the iris edge. Experiment 8.7.3 showed how the problem had to be separated into two parts because the area adjacent to the iris edge was not homogenous. Experiment 8.7.5 showed how each part was processed using a separate neural network. This shows how a problem can be broken down into separate parts and processed using separate networks and how the output can be combined to give the final solution.

- In an image processing problem the areas of interest can be separated into homogeneous sections. A separate neural network can then be used to process each section. A similar process has also been observed in other problems.

- Experiment 8.7.8 and 8.7.9 showed the effects of removing ''weak'' rules from the set of rules. In experiment 8.7.9 removing rule 7 which was a ''weak'' rule resulted in the network producing a poorer solution. The rule had an indirect effect on network performance.

- Rules for image processing tasks are generated from visible problem knowledge. The experiments in section 8.7.8 and 8.7.9 were performed to determine if a rule could be removed from the system. The experiments showed that the ''weaker rules'' could remain in the network.

Chapter 9

9 Calcification detection in mammograms

Several techniques are under development to aid in the early detection of cancerous tissue in mammograms, some of these techniques use neural networks. This chapter investigates the effects of using rules to initialize a fuzzy neural network (FuNN) to detect microcalcifications in digital mammograms. This problem was chosen for study, as it was considered to be a difficult real world problem.

9.1 Introduction

Cancer is a group of diseases in which cells in the body grow, change, and multiply out of control. Usually a cancer is named after the body part in which it originated; thus, breast cancer refers to the erratic growth and proliferation of cells that originate in the breast tissue. A group of rapidly dividing cells may form a lump or mass of extra tissue called a tumor. Tumors can either be cancerous (malignant) or non-cancerous (benign). Malignant tumors penetrate and destroy healthy body tissues. The term, breast cancer, refers to a malignant tumor that has developed from cells in the breast. There are several types of tumors that may develop within different areas of the breast. Most tumors are the result of benign (non-cancerous) changes within the breast [Imaginis]. These are also known as macrocalcifications that are coarse calcium deposits within the breast.

Microcalcification is a second type of calcification and usually signifies the presence of malignant tumors. Microcalcifications are small calcium deposits within breasts, singly or in clusters [Imaginis]. Breast cancer is the most common form of cancer among women [Bazzani 2000]. Mammography is a special type of x-ray imaging used to create detailed images of the breast. Mammography uses a low dose x-ray, high contrast high resolution film and an x-ray system specifically designed for the task. Successful treatment of breast cancer depends on early diagnosis [Imaginis]. The presence of microcalcifications in breast tissue is one of the main features looked for by radiologists in diagnosis [Bazanni 2000]. Many imaging technologies are under development to automate the detection of microcalcifications. Some of these technologies use neural networks for the detection and classification of calcifications. The work presented in this chapter investigates the effects of using problem knowledge to initialize a FuNN to detect clustered microcalcifications in mammograms. The aim of the work was to concentrate on this as a case study in the hope that the FuNN based approach could be shown to have better performance than established techniques.

9.2 The training data

The mammogram images used for the experiments in this chapter were downloaded from the Digital Database for Screening Mammography (DDSM) available at the University of Florida web site [Mammo, Heath 1998]. The images are digitized screening mammograms [Moore 2001]. The database consists of 2620 cases, each of which contains 4 mammograms, two mammograms for each breast. Figure 9.1 shows the two mammographic views taken for each breast. CC represents the Cranio-caudal view and MLO represents the mediolateral oblique (MLO) mammographic view. The cases used in the experiments were from a dataset specifically designed for evaluating clustered microcalcifications [Mammo]. Figure 9.2 shows the CC mammographic view of the right breast from case 1220 from the DDSM. Figure 9.3 shows the MLO mammographic view of the right breast from case 1220.

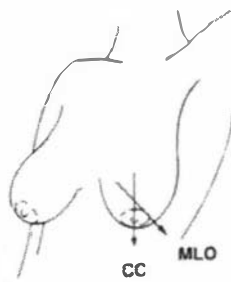


Figure 9.1 The two mammographic views taken for each breast[†].

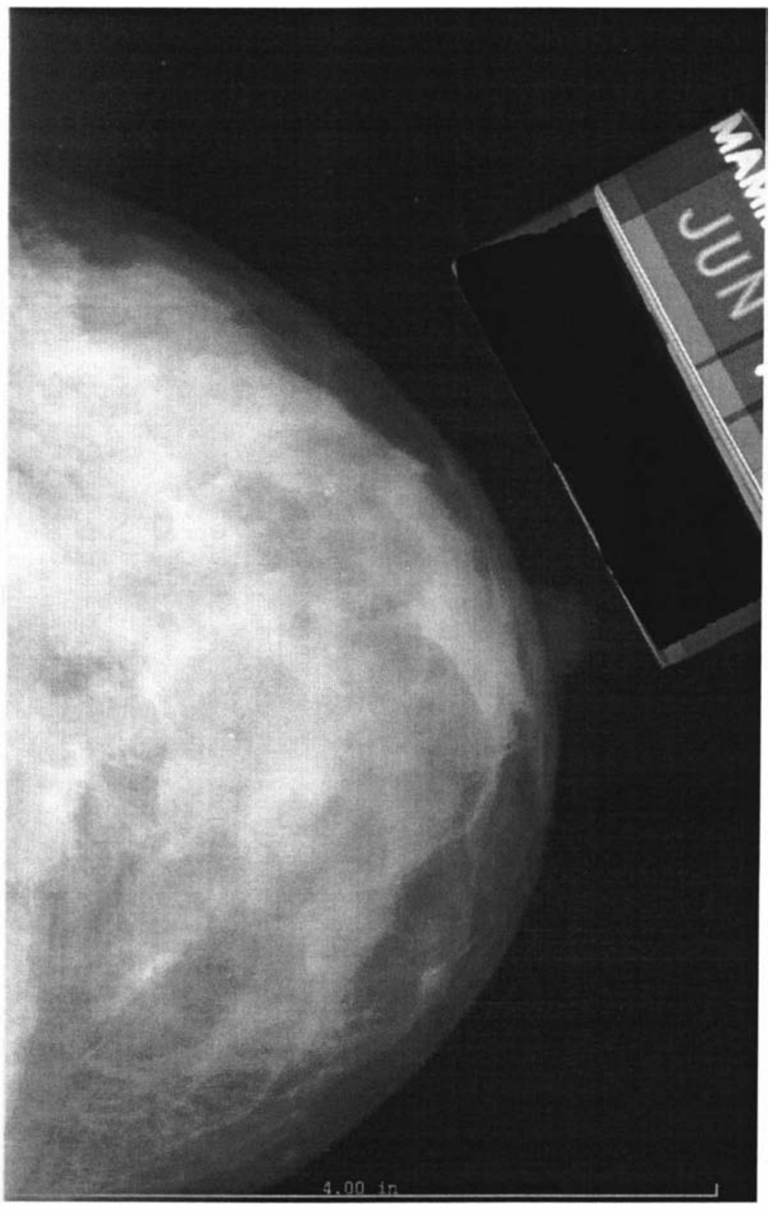


Figure 9.2 CC mammographic view of the right breast from case 1220 from the DDSM.

[†] Image courtesy of http://www.imaginis.com.

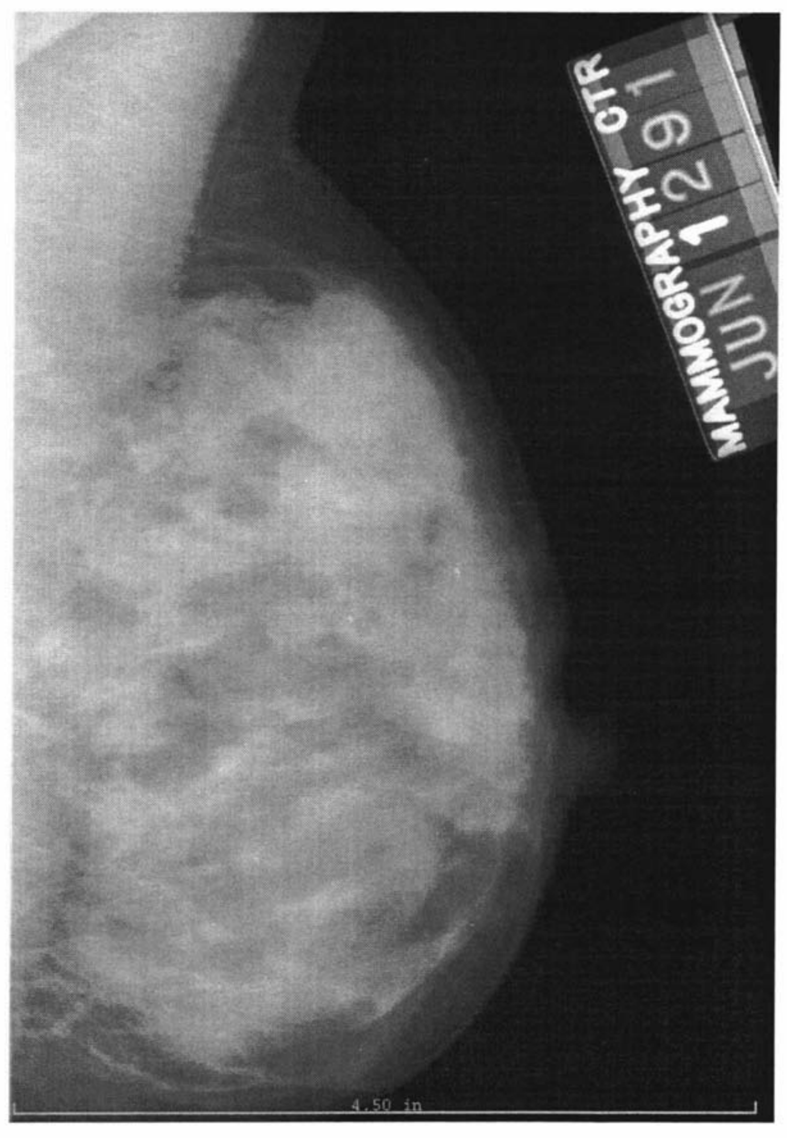


Figure 9.3 MLO mammographic view of the right breast from case 1220 from the DDSM.

Microcalcifications in mammograms appear as small clusters of high intensity dots. Figure 9.4 shows an image of a clustered microcalcification.

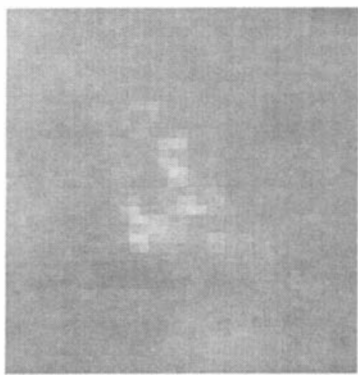


Figure 9.4 An image of a clustered microcalcification.

Figure 9.5 shows the training images used to train the FuNN. The scale at the bottom and the large caption (upper right) were removed before training. Image (a) is the input image and image (b) is the target image. The detection of microcalcifications is an image processing problem, so a fuzzy neural network window filter (FuNNWF) was used.

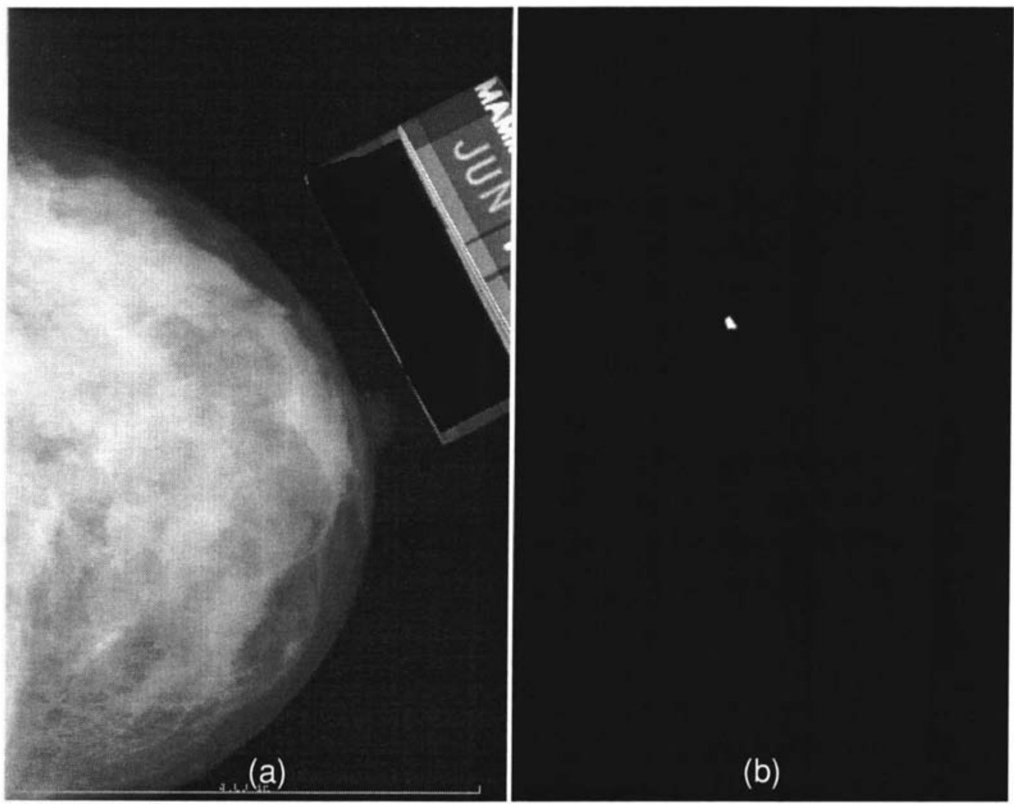


Figure 9.5 Training images for microcalcification detection problem. Image (a) is the input image and image (b) is the target image.

9.2.1 Window filter size

The required size of the window filter that was to be used was initially unknown so several clustered microcalcifications were measured to determine an appropriate size to be used. From the mammograms that were examined it was observed that the average extent of a clustered microcalcification was 23 pixels so a 23x23 window filter was used. It was also observed that the clusters of microcalcifications generally had a circular shape. This information about the shape of the clusters was used as problem knowledge.

9.3 Rule development

The experiments in this chapter were performed to determine how initializing a FuNN with rules effects network training. Rules representing problem knowledge needed to be generated in order to initialize the neural network. The rules for the toy problem, wane edge detection, tree crown detection and iris edge detection were developed by visually inspecting the intensities of the pixels in the window placed over the area of interest. For the wane and iris edge detection problem the rules were designed to detect vertical edges. For the tree crown detection problem the rules were designed to detect tree crowns having circular shapes. The way in which the rules were developed for these problems was similar in concept. Figure 9.4 shows a typical clustered microcalcification. The number of microcalcifications present in a cluster is different in each case. The position of each microcalcification in each cluster also varies for each mammogram. For these reasons developing rules to accommodate the variations in the number and position of the microcalcifications in each cluster would be impractical. The only common feature among the clustered microcalcifications was that they all had a circular shape. It was also observed that the texture of the area containing the clustered microcalcification was different to the rest of the mammogram. This was because clustered microcalcifications appear as clusters of high intensity dots in the mammogram. It was speculated that a textured based method could be used to differentiate clustered microcalcifications.

9.3.1 The texture based method

As explained previously, the circular shape of the cluster of microcalcifications was a common feature. So a circular window with a diameter of 23 pixels was used as a basis for the calculation of texture measures. The general approach is to compute an intermediate matrix, which depicts an aspect of the spatial relationships between the gray levels in an image [Siew 1987]. From such a matrix a set of statistics or features that summarise textural characteristics of the image are calculated. The Spatial Gray Level Dependance Matrix (SGLDM) [Haralick

1973] measures the frequencies with which all transitions in intensity occur between pixels separated by a specified distance in a specified direction [McLaughlin 1998]. The circular window was used to calculate the SGLDM, see below. While the intermediate matrix can directly provide information about the texture of an image, for example its coarseness, it is more fruitful to combine the elements in a systematic fashion to produce a set of statistics or "features". Each feature measures an aspect which can be related to the perceived texture of the original image [Lee 1988].

9.3.2 Spatial Gray Level Dependence Matrix (SGLDM)

The SGLDM calculation method is from [Siew 1987]. The spatial gray level dependence matrix is based in the estimation of the second order joint conditional probability density functions $f(i,j,d,a)$, $a = 0^0, 45^0, 90^0, 135^0$. Each $f(i,j,d,a)$ is the probability of going from gray level i to gray level j, given that the intersample spacing is d and the direction is given by angle a. If an image has Ng gray levels, then the density function can be written as Ng×Ng matrices. Each matrix can be computed from a digital image by counting the number of times each pair of gray levels occur at separation d and in the direction specified by a.

Let Lx = {1,2,...,Nx} and Ly = {1,2,...,Ny} be the x and y spatial domains and Lx×Ly be the set of resolution cells. Let G = {0,1,...,Ng-1} be the set of possible gray levels. Then a digital image I is a function which assigns some gray level to each and every resolution cell; I: Lx×Ly → G.

The initial assumption in characterizing image texture is that all texture information is contained in the SGLD matrices. Hence all the texture features are extracted from these SGLDM matrices. It is assumed that the texture –content information in an image I is contained in the overall or "average" spatial relationship which the gray levels in image I have to one another. More precisely, the assumption is that this texture content information is adequately specified by the matrix of relative frequencies Pij with which two neighbouring resolution cells separated by distance d occurred on the image, one with gray level i and the other

with gray level *j*. Such matrices of gray level dependence frequencies are a function of angular relationship between the neighbouring resolution cells as well as a function of distance between them ($P(i,j,d,a)$). The calculation of $P(i,j,d,a)$ is shown below:

$$P(i,j,d,0) = \#\{((k,l),(m,n))\varepsilon(Ly \times Lx) \times (Ly \times Lx)$$
$$| k-m=0, | l-n=d, | (k,i)=i, | (m,n)=j$$

$$P(i,j,d,45) = \#\{((k,l),(m,n))\varepsilon(Ly \times Lx) \times (Ly \times Lx)$$
$$| k-m=d, | l-n=-d, | (k,l)=i, | (m,n)=j$$

$$P(i,j,d,90) = \#\{((k,l),(m,n))\varepsilon(Ly \times Lx) \times (Ly \times Lx)$$
$$| k-m=d, | l-n=0, | (k,l)=i, | (m,n)=j$$

$$P(i,j,d,135) = \#\{((k,l),(m,n))\varepsilon(Ly \times Lx) \times (Ly \times Lx)$$
$$| k-m=d, | l-n=d, | (k,n)=i, | (m,n)=j$$

Three texture characteristics were used. They are explained below.

$$Energy: E = \sum_{i=0}^{Ng-1} \sum_{j=0}^{Ng-1} [f(i,j,d,a)]^2$$

This is a measure of homogeneity in an image. In a homogeneous image, there are very few dominant gray-tone transitions, hence the matrix for the image will have few entries of large magnitude. However, for an image which is not homogeneous, the matrix will have a large number of small entries and hence the energy feature will be small.

$$Entropy: ENT = -\sum_{i=0}^{Ng-1} \sum_{j=0}^{Ng-1} [f(i,j,d,a) Log(f(i,j,d,a))]$$

The entropy gives a measure of the complexity of the image. A complex image tends to have a higher entropy value than a simple one.

$$Correlation = COR = \sum_{i=0}^{Ng-1} \sum_{j=0}^{Ng-1} \left[\frac{(1-\mu x)(j-\mu y) f(i,j,d,a)}{(\sigma x \sigma y)} \right]$$

This is a measure of gray level linear dependence in an image, where $f(i, j,d,a)$ is a normalized $P(i, j,d,a,)$ matrix, Ng is the number of gray levels in the image from which the spatial gray level dependence matrices were extracted, and

$$\mu x = \sum_{i=0}^{Ng-1} i \sum_{j=0}^{Ng-1} f(i, j,d,a)$$

$$\mu y = \sum_{j=0}^{Ng-1} j \sum_{i=0}^{Ng-1} f(i, j,d,a)$$

$$(\sigma x)^2 = \sum_{i=0}^{Ng-1} (i-\mu x)^2 \sum_{j=0}^{Ng-1} f(i, j,d,a)$$

$$(\sigma y)^2 = \sum_{j=0}^{Ng-1} (j-\mu y)^2 \sum_{i=0}^{Ng-1} f(i, j,d,a)$$

At first the texture measures that should be used to classify the microcalcifications were not known. So the energy, entropy and correlation were calculated from the SGLDM. It was also speculated that the density measure of the pixels in the circular window maybe useful in detecting and classifying microcalcifications, so it too was calculated.

9.3.3 Neural network training method

Figure 9.6 shows how the inputs to train the FuNN were generated.

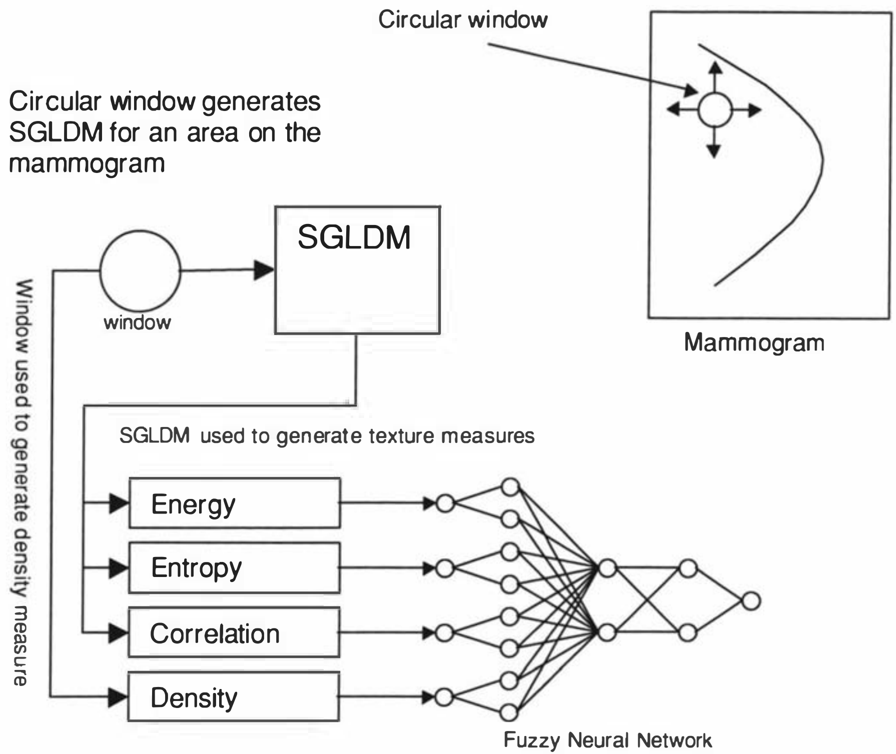


Figure 9.6 Input generation for FuNN for mammogram problem.

The circular window was used to compute both the density and the SGLDM for an area of the mammogram. The SGLDM was then used to calculate the selected texture measures. The position of the centre pixel of the window corresponding to the position of the pixel on the target image was used to generate the target data.

9.3.4 The values for the texture measures

The region where the calcification was present in the mammogram was used to determine the values for the three texture measures. Figure 9.5 (a) shows the CC mammographic view of the right breast for case 1220. This image was used to determine values for the texture measures. The steps to determine the measures is as follows,

1. The circular window was placed over the area where the clustered microcalcification was present and the SGLDM was calculated.
2. Using the SGLDM the energy, entropy, correlation and density was calculated and recorded.
3. The window was displaced by a few pixels by hand in a random direction making sure that the calcification under scrutiny was within the window.
4. Step 1 was repeated.

The above steps were repeated 20 times to record 20 values for each measure. The values for the measures were then normalised between zero and one. The values used to normalise the measure were obtained through a repetitive process.

9.3.4.1 The values used for normalising the texture measures and the density

Measure	Normalizing equation
Correlation	*Correlation * 5*
Energy	*Energy * 10*
Density	*Density /250*
Entropy	*Entropy * -0.2*

Table 9.1 Normalising equations.

The average value for each measure was used to represent the clustered microcalcification in the mammogram. Table 9.2 shows the measures representative of a microcalcification. These values were used to generate the fuzzy membership functions that were used to develop the rules.

Measure	Value
Correlation	0.064
Energy	0.2015
Density	0.72
Entropy	0.88

Table 9.2 Measure representative of a microcalcification.

9.3.4.2 Fuzzification membership functions

Membership function	Value
M1	0
M2	0.064 (Correlation)
M3	0.2015 (Energy)
M4	0.72 (Density)
M5	0.88 (Entropy)
M6	1

Table 9.3 Fuzzification membership functions used for mammogram calcification detection.

9.3.4.3 Defuzzification membership functions

Membership function	Value
M1	0 (microcalcification found)
M2	1 (microcalcification not found)

Table 9.4 Defuzzification membership functions used for mammogram calcification detection.

The membership functions that were explained in sections 9.3.4.2 and 9.3.4.3 were used to generate rules for the experiments.

9.4 The experiments

The experiments performed on microcalcification detection are explained in this section. Table 9.5 shows a summary of the experiments performed. Different combinations of the measures were used to train the network to determine if the calcifications could be detected. The training images in section 9.2 were used to train the networks in runs 1, 2, 3, 4 and 5. For these runs the network did not detect the calcification in the input image. In run 6 when all four measures were used, the network detected the calcification in the input image. Thus it was observed that all four measures were required by the network to discriminate between microcalcifications and normal tissue in the mammogram. Several experiments were performed using the four measures and they are described in this section.

Run number	Measures used for run
Run 1	Density
Run 2	Energy & Density
Run 3	Entropy & Density
Run 4	Correlation & Density
Run 5	Energy, Entropy & Density
Run 6	Energy, Entropy, Correlation & Density

Table 9.5 The measures used for each experimental run.

9.4.1 Experiment to determine if the FuNN could detect microcalcifications in a mammogram

This experiment was performed to determine if a FuNN could reliably detect clustered microcalcifications in a mammogram. Figure 9.5 shows the training images used. The circular window filter with a diameter of 23 pixels was passed over the input image to generate the training data. As explained in section 9.3.3 the circular window was used to generate the SGLDM for an area on the image. The SGLDM was then used to generate the energy, entropy and correlation for that area. The window was also used to generate the density measure for that area. These four measures were used to train the network. The membership functions in section 9.3.4.2 and 9.3.4.3 were used to generate the rules.

9.4.1.1 The rules for experiment 9.4.1

Rule no.	Description
1	If I1=M3 I2=M5 I3=M2 I4=M4 Then O1=M2 (rule designed to detect microcalcifications in mammograms)
2	If I4=M2 Then O1=M1
3	If I4=M3 Then O1=M1
4	If I4=M4 Then O1=M1
5	If I1=M3 Then O1=M1
6	If I1=M4 Then O1=M1

7	If I1=M5 Then O1=M1
8	If I1=M2 I2=M3 I3=M1 I4=M5 Then O1=M1 (rule designed to detect areas that are similar to a microcalcification)
9	If I1=M1 I2=M6 I3=M1 Then O1=M1 (rule designed to detect areas that are similar to a microcalcification)

Table 9.6 Rule used for experiment 9.4.1.

9.4.1.2 The network structure used for experiment 9.4.1

Layer	No. of nodes used
Input layer	4 inputs
Condition layer	6 conditions per input = 24 condition nodes
Rule layer	9 rule and 8 free = 17 rule nodes
Action layer	2 defuzzification nodes
Output layer	1 node

Table 9.7 4-24-17-2-1 network structure used for experiment 9.4.1.

Two networks with the above structure were generated. One network was initialized with the rules in section 9.4.1.1 and the other network was initialized with random numbers in the range {-1, +1}. The network was then trained for a range of epochs. It was determined that the network reached an optimal solution after 200 epochs. Thus a training time of 200 epochs was used for the networks in the remainder of this chapter. The trained network was then used to process the input image to generate the output image.

9.4.1.3 Step processing

In the previous chapters when processing the input image with the trained FuNNWF the window was moved across and down the image pixel by pixel in a raster scan. The input image for this problem was a mammogram with dimensions 990x627 (height x width in pixels). A very long processing time was required for each image as the SGLDM and texture measures were calculated for each circular window position. The image took approximately nine hours to process on a PC with a Pentium III processor running at 800mhz. A stepwise processing approach

was used to reduce the image processing time. The window moved five pixels horizontally and vertically after each calculation. This reduced image processing time to approximately 21 minutes. The step size of five was selected as a result of an iterative process.

9.4.1.4 The error measure

The error measure returns the following measures,

1. 1 is returned if the calcification on the target image was present on the output image, 0 otherwise.
2. The number of points on the output image that belong to a microcalcifications.
3. The number of points on the output image that are not microcalcifications. These are known as false positives.

The error function was developed through an iterative process.

9.4.1.5 Results for experiment 9.4.1

Network	Calcification found	Points in calcification	No. of false positives
Rule run	Yes	8	18
Random run	Yes	8	44

Table 9.8 Results for experiment 9.4.1.

The table shows the results for case 1220. Both networks detected the microcalcification in the mammogram. The network initialized with random numbers generated a larger number of false positives compared to the network initialized with rules. In this respect the network initialized with random numbers was outperformed by the rule initialized network.

9.4.2 Experiment to compare network performance

The previous experiment showed how a trained FuNN could detect the microcalcifications in a mammogram. In this experiment the trained network was tested on mammograms that were not included in the training set. Eight

mammogram cases were used for this experiment; one of these was in the training set. Table 9.9 shows the results for this experiment. The microcalcifications in case 1116, 1223 and 1258 was not detected. This occurred because the calcification in the mammograms for these three cases was overlapped by a speculated mass. The network was trained only to detect microcalcifications.

	Results when network was initialized with rules			Results when network was initialized with random numbers		
Case	Calcification found	Points found in calcification	No. of false positives	Calcification found	Points found in calcification	No. of false positives
Case1256	Yes	1	18	Yes	2	46
Case1213	Yes	9	14	Yes	9	45
Case1214	Yes	6	7	Yes	6	22
Case1219	Yes	4	9	Yes	5	14
Case1220	Yes	8	18	Yes	8	44
Case1223	No	0	26	No	0	40
Case1116	No	0	2	No	0	7
Case1258	No	0	32	No	0	59

Table 9.9 Results for experiment 9.4.1.

Table 9.9 shows that,

- The network initialized with rules generated a lower number of false positives when compared to the results of the network initialized with random numbers.
- Both networks detected the microcalcification in five mammograms. Except for two cases, both networks detected the same number of calcification points.

The results led to the speculation that the network may not need to be initialized with rules to detect the microcalcifications.

9.4.3 Experiment to determine the network performance of a large population of networks

This experiment was performed to determine the performance of a population of networks initialized with random numbers. 100 networks were used for this experiment. The weights of each network were initialized using random numbers in the range {-1, +1} taken from different seedings. The images in section 9.2 were used to train the network. Table 9.10 shows the mean of the error measures for the output images generated by the networks. Each network detected the calcification in the mammogram.

	Calcification found by every network	Average number of points found in calcification	Average number of false positives
Random run	Yes	8	34

Table 9.10 Results for experiment 9.4.3.

The speculation that the network need not be initialized with rules to detect the microcalcification was correct. Ten of the trained networks from the population were used to process the test set to determine the performance of each network. The networks were chosen as follows,

Network to use = N_{i*10}, where i=1..10 and N_i is the ith network from the population of 100.

Table 9.11 shows the average error measures for the test set.

Case	Calcification found	Average points found in calcification	Average number of false positives
Case1116	No	0	6
Case1213	Yes	9	46
Case1214	Yes	6	22
Case1219	Yes	5	15
Case1220	Yes	8	45
Case1223	No	0	40
Case1256	Yes	2	47
Case1258	No	0	57

Table 9.11 Results for experiment 9.4.3 using the ten networks initialized with random numbers.

The results in Table 9.11 are similar to the results in Table 9.9. These results show that the different random seedings for the weights of the network had a minor effect on network training for this problem.

9.4.4 Experiment to determine if the calcification detection rate of the network could be improved

The results from the previous experiments showed that the network could not detect the calcifications in case 1116, 1223 and 1258. This was because the calcifications in those cases were overlapped with speculated masses. For this experiment a new training set was created that contained areas of interest representing microcalcifications and speculated masses. This experiment was performed to determine the detection rate of the network trained on the new training set.

9.4.4.1 Generating the new training set

The new training set was generated to include areas containing calcifications and speculated masses. Two areas of interest were extracted from each mammogram in the test cases. One area of interest contained a calcification with/without an overlapping speculated mass. The other area of interest did not contain any

cancerous tissue. A total of 16 areas of interest were generated from the cases. These were combined to generate the new training set.

Two networks were generated for this experiment. The weights of one network were initialized using rules and the weights of the other network were initialized with random numbers in the range {-1, +1}. Each network was trained for 200 epochs using the new training set. Then each network was used to process the mammograms in the test set. Table 9.12 shows the error measures for the output images generated by each network.

	Results when network was initialized with rules			Results when network was initialized with random numbers		
Case	Calcification found	Points found in calcification	No. of false positives	Calcification found	Points found in calcification	No. of false positives
Case1116	Yes	33	29	Yes	34	30
Case1213	Yes	16	43	Yes	19	45
Case1214	Yes	12	60	Yes	18	49
Case1219	Yes	12	20	Yes	13	24
Case1220	Yes	23	25	Yes	18	23
Case1223	Yes	1	33	Yes	1	37
Case1256	Yes	9	41	Yes	10	53
Case1258	Yes	3	66	Yes	3	64

Table 9.12 Error measures for the output images generated by the two networks for experiment 9.4.4.

The measures in the table show,

- that both networks detected the calcifications in all the mammograms in the test set.
- that for five mammograms the network initialized with random numbers detected a greater number of points in a calcification compared to the network initialized with rules.

- that for five mammograms the network initialized with rules generated a lower number of false positives compared to the number of false positives generated by the network initialized with random numbers.
- that for this run, initializing the network with rules has not resulted in a significant improvement in calcification detection compared to the network initialized with random numbers.

Several other experiments were performed where the weights of the neural network were initialized with rules and random numbers. The results from these experiments showed that the use of rules to initialize the network did not result in a significant improvement in calcification detection. As before the calcifications detected by the network initialized with rules were also detected by the network initialized with random numbers.

9.5 Discussion and conclusions

The experiments in this chapter were performed to determine the effects on training when problem knowledge was used to initialize weights of the FuNN. The experiments showed that the output generated from the trained FuNN had a high number of false positives. This is a common problem encountered by other researchers. The objective of the work done in this chapter was to determine if the use of rules could improve network performance by reducing the number of false positives found in a mammogram.

In the wane edge detection, tree crown detection and iris edge detection problem, problem knowledge was extracted through visual inspection. For this problem rules could not be extracted visually so a different approach was used. The SGLDM is generally considered to be a very powerful method of statistical texture analysis [Lee 1988]. This statistical texture based method was used to generate features that were used to develop rules. A density measure was also used in the rules. The experiments performed in this chapter compared the calcification detection performance of networks initialized with and without rules. The results showed that for this problem the use of rules to initialize a FuNN did not

significantly improve network performance as expected. As a result the values of the features and density for different areas of the mammogram were scrutinised. It was observed that the three texture features and density measure were adequate to discriminate between calcification and non-calcification regions in the mammogram.

The following factors could be considered before using the rule initialization method when training a network,

1. If there is a high discrimination in the training data then the rule initialization method need not be used to initialize the weights of the FuNN. A FuNN initialized with random numbers may be used.
2. The ground truth data for the wane edge detection, tree crown detection and iris edge detection problem was ambiguous, noisy and inaccurate. So using rules to initialize the networks to solve those problems was advantageous. The images in the Digital Database for Screening Mammography are an exemplary set. The ground truth is 100% accurate. In such a situation an advantage is not gained by using rules to initialize a FuNN before training.

Chapter 10

10 Reducing the FuNN to a conventional MLP network

The fuzzy neural (FuNN) is a connectionist model for fuzzy rules implementation and inference [Kasabov 1996]. The FuNN architecture consists of a multilayer perceptron (MLP) network with two additional layers, the fuzzification and defuzzification layers, resulting in a five layer network. The rule mapping procedure developed in chapter 3, maps IF-THEN fuzzy rules to the FuNN architecture. This chapter investigates how the rules developed for the FuNN can be mapped from a five layer FuNN architecture to a four layer MLP network. The mapping process represents an indirect method of using rules to initializing a MLP network. This chapter also investigates the effects rules have on MLP network performance.

10.1 Introduction

The training performance of a multilayer perceptron network depends mainly on the initial values of its weights and biases, learning rate, network topology and on learning rule improvements like the momentum term [Thimm 1995]. Several methods have been proposed to initialize the weights of a MLP network [Yam 2000, Thimm 1997, Nguyen 1990, Weymaere 1994]. Chapter 3 showed how

problem knowledge represented by rules could be used to initialize the weights of a fuzzy neural network. This chapter shows how the weights of the rule initialized FuNN can be mapped to the weights of a MLP network. The method can be seen as an indirect approach to initializing the weights of a MLP network. The process involves mapping the weights of the five layer FuNN network to a four layer MLP network. It was expected that the MLP network would require less computation during training as it is smaller than the FuNN. The effects of using rules to initialize the MLP network were also investigated.

10.2 Funn to MLP mapping

Chapter 3 described how rules developed from problem knowledge could be mapped to a FuNN architecture. The mapping algorithms could be used with MLP networks if an equivalence can be established between the FuNN and a MLP network. As this equivalence is only needed to generate initial weights from rules, it need only be valid at the crisp instantiation of the rules. The equivalence will be determined layer by layer. Figure 10.1 shows the proposed mapping scheme.

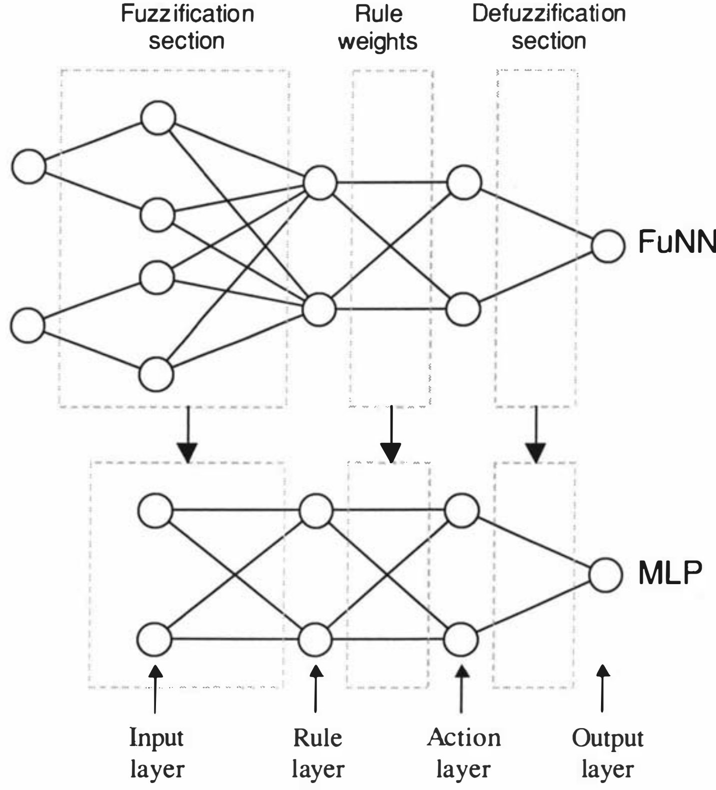


Figure 10.1 The proposed FuNN to MLP mapping.

The weight connections from the rule layer to the action layer in the FuNN remain the same as they can be mapped directly to the hidden layer of a MLP network. An equivalent network need only be developed for the fuzzification and defuzzification layers.

10.2.1 Fuzzification layer equivalence

Figure 10.2 (a) shows a section from the FuNN network and (b) is a section from the MLP network. I_k and I_{k+1} represent the inputs to both networks. $C_{k,1}$ and $C_{k,2}$ represent the fuzzy membership labels for input I_k and $C_{k+1,1}$ and $C_{k+1,2}$ represent the fuzzy membership labels for input I_{k+1}. $Z_{k,1}$, $Z_{k,2}$, $Z_{k+1,1}$ and $Z_{k+1,2}$ represent the condition nodes for the FuNN section. $W_{k,1}$ to $W_{k+1,2}$ represent the weight connections from the condition layer to the rule node R in the FuNN section. V_k and V_{k+1} represent the weight connections from the inputs to the rule node R in the

MLP section. b is the bias connection to rule node R in the MLP section. Figure 10.2 shows the proposed equivalent MLP network for the FuNN section when the number of encoded symbols for each input is restricted to the two extremes of the input range. As the symbols are restricted to the range extremum the fuzzification layer nodes perform a linear calculation in the FuNN, as does the input to the rule node R in the MLP network.

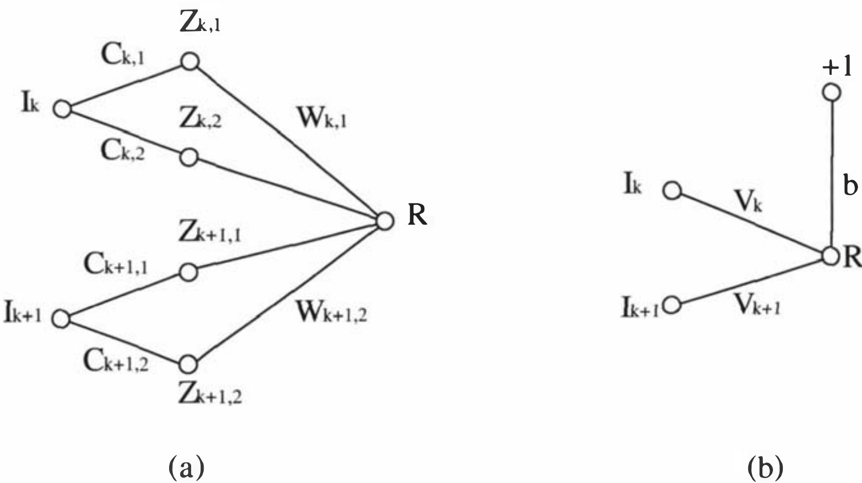


Figure 10.2 The proposed equivalent MLP network for the FuNN. (a) shows a section from the FuNN and (b) shows the equivalent MLP section.

Symbols $C_{k,1}$ and $C_{k,2}$ convert crisp input I_k to the fuzzy values $Z_{k,1}$ and $Z_{k,2}$. The input to the rule node R from the input I_k is,

$$Z_{k,1}W_{k,1} + Z_{k,2}W_{k,2} = R \qquad (1)$$

As $Z_{k,1}$ and $Z_{k,2}$ are spread encoded,

$$Z_{k,1} + Z_{k,2} = 1$$
$$\therefore Z_{k,2} = 1 - Z_{k,1}$$

Equation (1) can be rewritten as,

$$Z_{k,1}W_{k,1} + (1 - Z_{k,1})W_{k,2} = R \qquad (2)$$

The input to the rule node R in the MLP section can be written as,

$$I_k V_k + b = R \qquad (3)$$

At the crisp instantiation equations (2) and (3) are equivalent. Equating these equations and solving gives,

$$Z_{k,1}W_{k,1} + (1 - Z_{k,1})W_{k,2} = I_k V_k + b$$

$$Z_{k,1}W_{k,1} + W_{k,2} - Z_{k,1}W_{k,2} = I_k V_k + b$$

$$Z_{k,1}(W_{k,1} - W_{k,2}) + W_{k,2} = I_k V_k + b$$

By equating the coefficients for I_k and $Z_{k,1}$,

$$V_k = (W_{k,2} - W_{k,1}) \qquad (4)$$

for k=1,2,3,…,N, where N is the number of input nodes,

and the bias,

$$b = \sum_{k=1}^{N} W_{k,2} \qquad (5)$$

10.2.2 Defuzzification layer equivalence

The proposal is to replace each node in the defuzzification layer of the FuNN with a MLP node having a log-sigmoid transfer function. Figure 10.3 shows the proposed MLP network equivalent for the defuzzification section of the FuNN. A_1 and A_2 represent the nodes in the action layer and C_1 and C_2 represent the defuzzification membership functions in the FuNN. W_1 and W_2 represent the weight connections from nodes A_1 and A_2 to O_i in the MLP network.

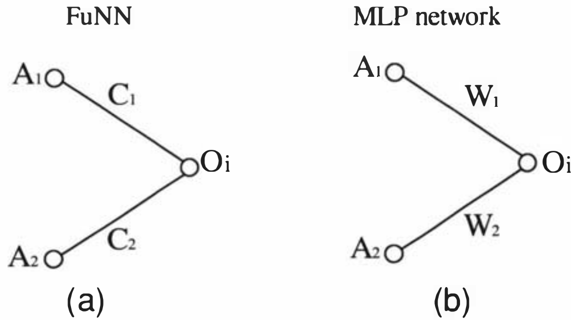


Figure 10.3 Proposed mapping for defuzzification section. (a) is the output layer of the FuNN and (b) is the equivalent output layer for the MLP network.

For the FuNN the outputs are calculated as,

$$O_i = \frac{\sum_{i=1}^{N} A_i C_i}{\sum_{i=1}^{N} A_i} \qquad (6)$$

where N is the number of nodes in the action layer.

For the MLP network the output is,

$$O_i = \frac{1}{1+e^{-\sum AW}} \qquad (7)$$

At the crisp instantiation of the rule only one of the A_i has a non-zero value and the sum of the A_i is one. Equating (6) and (7),

$$\frac{\sum_{i=i}^{N} A_i C_i}{\sum_{i=1}^{N} A_i} = \frac{1}{1+e^{-\sum AW}}$$

$$\frac{A_1 C_1 + A_2 C_2}{A_1 + A_2} = \frac{1}{1+e^{-(A_1 W_1 + A_2 W_2)}}$$

$$e^{(A_1 W_1 + A_2 W_2)} = \frac{A_1 C_1 + A_2 C_2}{A_1 + A_2 - A_1 C_1 - A_2 C_2}$$

$$A_1 W_1 + A_2 W_2 = \ln\left(\frac{A_1 C_1 + A_2 C_2}{A_1 + A_2 - A_1 C_1 - A_2 C_2}\right)$$

When $A_1=0$ and $A_2=1$

$$W_2 = \ln\left(\frac{C_2}{1-C_2}\right)$$

When $A_1=1$ and $A_2=0$

$$W_1 = \ln\left(\frac{C_1}{1-C_1}\right)$$

When the defuzzification membership functions have values zeros or one W_i in equation (8) does not exist. So values close to zero and one must be used to determine the weight values for the MLP network. As the value of $C_i \rightarrow 0$ the value of $W_i \rightarrow$ large negative number and as the value of $C_i \rightarrow 1$ the value of $W_i \rightarrow$ large positive number. So the weights of the output layer are assigned such that the output nodes reaches near saturation.

$$\therefore W_i = Min\left(\ln\left(\frac{C_i}{1-C_i} \right), 4 \right) \qquad (8)$$

for $C_i \neq 0$ and $C_i \neq 1$.

A more automatic method of weight selection was required to initialize the weights of the output layer of the MLP network. So Yam and Chow's [Yam 2000] method was used to initialize the weights in the output layer of the MLP network.

10.2.3 Yam and Chow's weight initialization method

A brief description of Yam and Chow's weight initialization method is given in this section. A detailed description of the method can be found in [Yam 2000]. Yam and Chow's method can be used to initialize the all of the weights in the MLP network. This was primarily used as an automatic process to initialize the weights of the last layer of the MLP network. The notation for the equations describing Yam and Chow's method is the one used in [Yam 2000].

10.2.3.1 Weight initialization method

The method can be used to initialize the weights of a fully connected MLP network.

In Yam and Chow's algorithm the magnitude of the weights θ_p^l for input pattern p is,

$$\theta_p^l = s\sqrt{\frac{\tau}{(n_l+1)\sum_{i=1}^{n_l+1}\left(a_{p,i}^l\right)^2}} \quad (9)$$

where $\tau = 3$ for weights with uniform distribution and $\tau = 1$ for weights with a normal distribution.

For different input patterns, the values of θ_p^l are different. To make sure the outputs of the hidden neurons are in the active region for all patterns, the following value is selected,

$$\theta^l = \min(\theta_p^l) \quad (10)$$

where,

$s \approx 4.59$ for standard sigmoidal function,

n_l is the number of nodes in layer l, where l=1,..,L-1 where L is the total number of layers in the MLP network,

$a_{p,i}^l = f(O)$

$O = AW$

where A is the output from the previous layer and W represents the weight matrix of the previous layer and f is the sigmoidal activation function.

The steps of the weight initialization algorithm are as follows,

1. Evaluate θ^l using the input training patterns by applying equations (9) and (10) with l=1.
2. The weights $w_{i,j}^l$ are initialized with a random number generator with uniform distribution between $\{-\theta^l, +\theta^l\}$ or normal distribution between $\{0, (\theta_p^l)^2\}$.
3. Evaluate ${a_{i,j}^l}^2$ by feedforwarding the input patterns through the network using $w_{i,j}^l$.
4. For l=2,..,L-2
   (a) Evaluate θ^l using the outputs of layer l, i.e. $a_{p,i}^l$ and apply equation (9) and (10).

(b) The weights $w_{i,j}^{l}$ are initialized by a random number generator with uniform distribution between $\{-\theta^{l},+\theta^{l}\}$ or normal distribution between $\{0,(\theta_{p}^{l})^{2}\}$.

(c) Evaluate $a_{p,i}^{l+1}$ by feedforwarding the outputs of $a_{p,i}^{l}$ through the network using $w_{i,j}^{l}$.

5. After finding $a_{p,i}^{L-1}$ or A^{L-1} we can find the last layer of weights W^{L-1} by solving the following equation using a least squares method,

$$\text{minimize } \left\|A^{L-1}W^{L-1}-S\right\|_{2} \qquad (11)$$

where S is a matrix, which has entries,

$$s_{i,j}=f^{-1}(t_{i,j})$$

where $t_{i,j}$ are the entries of the target matrix T.

10.3 The experiments

The following problems were investigated in this section,

1. Sobel filter emulation
2. Wane edge detection
3. Tree crown detection

The use of a fuzzy neural network window filter (FuNNWF) to solve the wane edge detection and tree crown detection problem was investigated in chapter 6 and 7. The use of a FuNNWF to emulate the Sobel filter was investigated by Siroki [Siroki 1998]. The experiments in this section were performed to determine the effects of using different initialization methods on MLP training.

10.3.1 Experiment to determine the network performance of a MLP network initialized with rules

This experiment was performed to determine the network performance of a MLP network initialized with rules. Sobel filter emulation, wane edge detection and the

tree crown detection problem data, were used to train the network. Rules representing each problem were used to initialize the MLP network before training.

10.3.1.1 Rules used for Sobel filter emulation

The Sobel filter is a combination of two linear filters using a 3x3 window, one for vertical edge detection and the other for horizontal edge detection. An edge is detected when the intensity across the window changes from low to high or vice versa. The intensity change can be in the vertical or horizontal direction. The fuzzification membership functions LOW and HIGH were used to represent low and high intensities in the window. The rules describing the edge detector are as follows,

Rule no.	Description
1	If the intensity of the left side of the window is low and the intensity of the right side of the window is high then an edge is detected.
2	If the intensity of the left side of the window is high and the intensity of the right side of the window is low then an edge is detected.
3	If the intensity of the top of the window is low and the intensity of the bottom of the window is high then an edge is detected.
4	If the intensity of the top of the window is high and the intensity of the bottom of the window is low then an edge is detected.

Table 10.1 Rules for Sobel filter emulation.

10.3.1.2 Structure of the neural network window filter used for the Sobel filter emulation problem

Layer	No. of nodes
Input layer	9 nodes (3x3 window)
Condition layer	2 membership functions for each input = 18 nodes
Rule layer	4 rules + 8 free = 12 nodes
Action layer	2 nodes
Output layer	1 node

Table 10.2 Network structure for Sobel filter experiment.

10.3.1.3 Rules used for wane edge detection

The wane edge is predominantly vertical so the rules were designed to detect a vertical edge. A complete description on wane edge detection can be found in chapter 6. The rules used in chapter 6 section 6.6.7 were used for this experiment.

10.3.1.4 Structure of the neural network window filter used for the wane edge detection problem

Layer	No. of nodes
Input layer	81 nodes
Condition layer	2 membership functions for each node = 162 nodes
Rule layer	4 rule + 8 free = 12 nodes
Action layer	2 nodes
Output layer	1 nodes

Table 10.3 Network structure used for wane edge detection experiment.

10.3.1.5 Rules used for tree crown detection

Tree crowns are predominantly circular with an average diameter of 11 pixels. So an 11x11 window filter was used to detect the tree crowns. A complete description on tree crown detection can be found in chapter 7. The rules used in chapter 7 were used for this experiment.

10.3.1.6 Structure of the neural network window filter used for the tree crown detection problem

Layer	No. of nodes
Input layer	121 nodes
Condition layer	2 membership functions for each input = 242 nodes
Rule layer	4 rule + 11 free = 15 nodes
Action layer	2 nodes
Output layer	1 node

Table 10.4 Network structure used for tree crown detection problem.

10.3.1.7 Network initialization methods

The MLP networks generated for problems 10.3.1.1, 10.3.1.3 and 10.3.1.5 were initialized using four different methods. They were,

1. Using the FuNN to MLP mapping technique **(method 1)**.
2. Using the FuNN to MLP mapping technique to initialize the input and rule layer weights and Yam and Chow's method to initialize the weights of the last layer **(method 2)**.
3. Using Yam and Chow's method to initialize the weights of the entire network **(method 3)**.
4. Using random numbers from a normal distribution in the range {-1, +1} **(method 4)**.

10.3.1.8 Result for Sobel filter emulation problem

Figure 10.4 shows a graph of the training performance of the MLP network initialized using the four methods described in section 10.3.1.7. The network initialized using a combination of the FuNN to MLP mapping and Yam and Chow's method (method 2) has started with the lowest RMS error at the beginning of training. After 100 epochs the MLP network initialized with method 2 has reached the lowest RMS error. The training performance of the network initialized with method 3 started from a higher RMS error when compared to the network initialized with method 4; but after 100 epochs the training performance of the network initialized with method 3 has reached a lower RMS error. Table 10.5 shows the RMS error between the target image and the output image generated by the trained network. The table shows that the error between the target and output image is the lowest for the network initialized with the FuNN to MLP mapping combined with Yam and Chow's method (method 2).

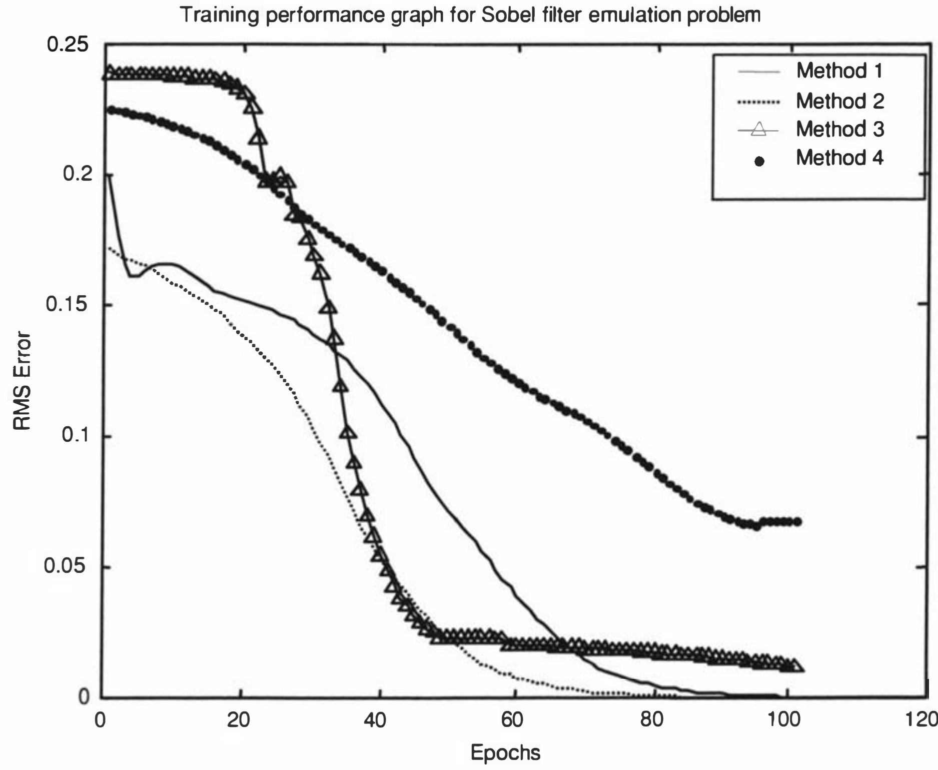


Figure 10.4 Training performance of MLP network for the Sobel filter emulation problem.

Initialization method	RMS error between target and output image
FuNN to MLP mapping	10.1
FuNN to MLP mapping combined with Yam and Chow's method	9.5
Yam and Chow's method	37.9
Random numbers	57.4

Table 10.5 RMS error between output and target image for Sobel filter emulation problem.

10.3.1.9 Results for wane edge detection problem

Figure 10.5 shows a graph of the training performance of the MLP network initialized using the four methods described in section 10.3.1.7. The figure shows that the MLP network initialized with method 2 has started training from a lower RMS error when compared to the starting RMS error of the other initialization methods. The final training RMS error of the network initialized with method 2 is also the lowest compared to the final RMS error of the other networks. Table 10.6

shows the error between the target image and the output image generated by the trained network. The table shows that the error between the target and output image is the lowest for the network initialized with the FuNN to MLP mapping procedure.

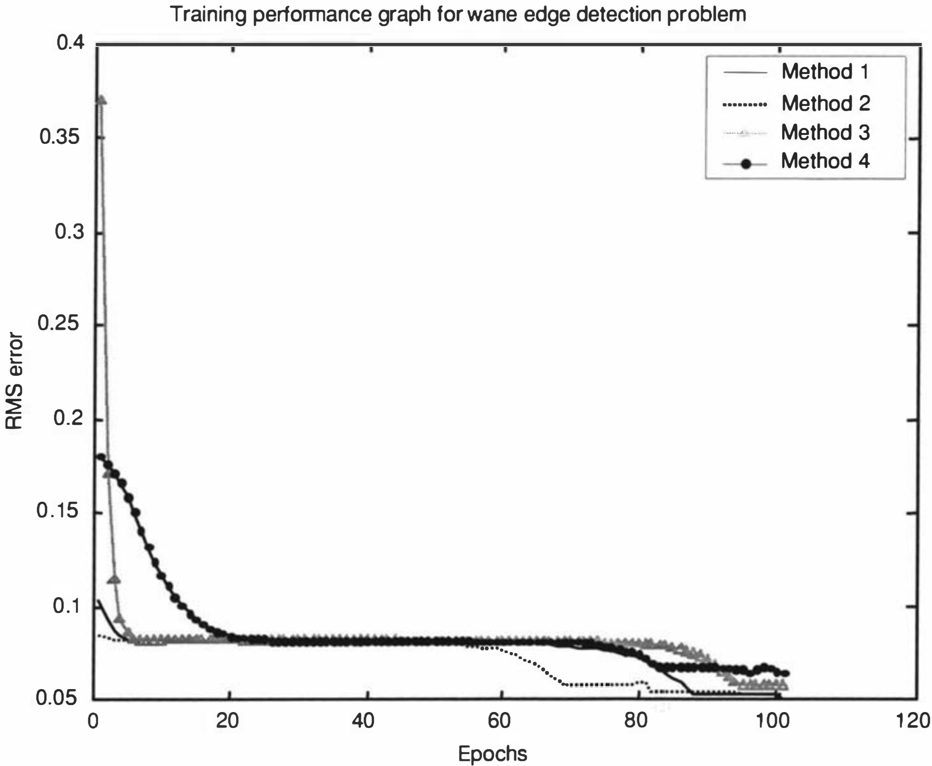


Figure 10.5 Training performance of MLP network for wane edge detection problem.

Initialization method	False positives	Final error
FuNN to MLP mapping	300	62.3
FuNN to MLP mapping combined with Yam and Chow's method	323	67.2
Yam and Chow's method	315	65.6
Random numbers	346	103.6

Table 10.6 Error between output and target image for wane edge detection problem.

10.3.1.10 Results for tree crown detection problem

Figure 10.6 shows a graph of the training performance of the MLP network initialized using the four methods described in section 10.3.1.7. For this problem

the network initialized using Yam and Chow's method (method 3) had a high starting RMS error compared to the other networks.

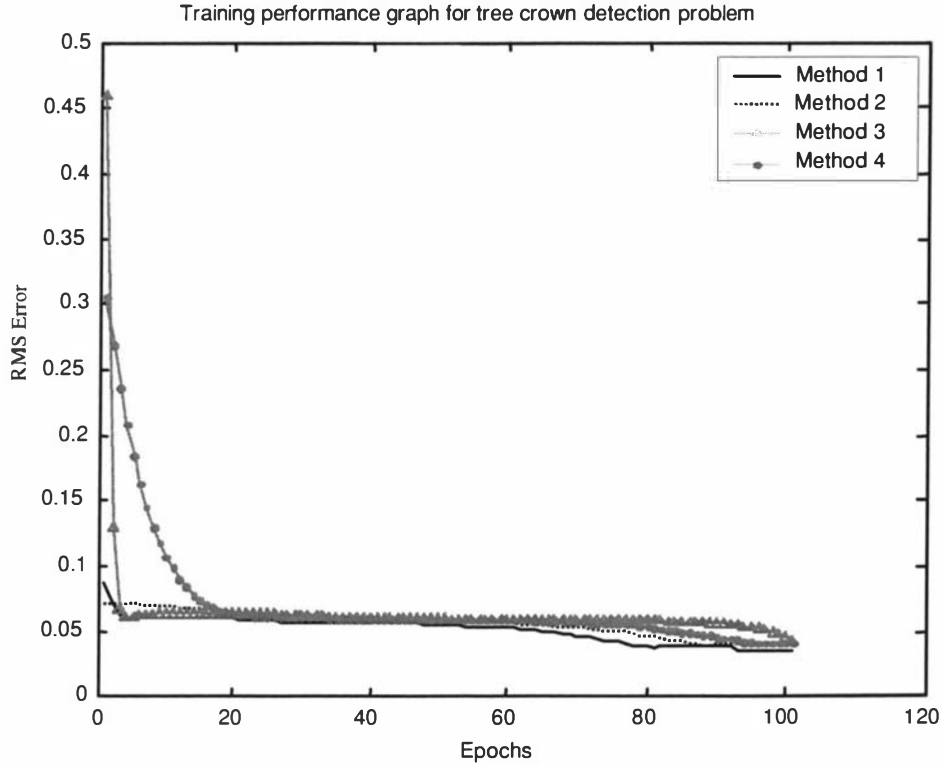


Figure 10.6 Training performance of MLP network for the tree crown detection problem.

The error between the target image and the output image is not shown as only one network from the hierarchical network structure was used for training (see chapter 7 for more details).

Several runs were performed on the above three problems using the four initialization methods. During some training runs it was observed that the networks initialized using Yam and Chow's method (method 2 and 3) got ''stuck'' in a local minimum. Sometimes Yam and Chow's method generated large values in the range of 100 for the weights in the output layer. These large values forced the nodes in the network to saturate causing the network to remain in a local minimum during training. This effect has also been reported by Yam and Chow [Yam 2000].

10.3.2 Experiment to determine the effects on MLP training when a modified version of Yam and Chow's initialization method was used

In the previous experiment it was observed that Yam and Chow's network initialization method generated large values for the weights for some runs causing the network to remain in a local minimum during training. To prevent this the values generated for the output weights were normalised between one and ten. The modification prevented the network from saturating when training started. For this experiment the Sobel filter emulation, wane edge detection and tree crown detection problems were investigated again. A population of 100 networks was trained on each problem and each network from the population was initialized using the four methods described in section 10.3.2.1 before training.

10.3.2.1 Network initialization methods

The MLP networks generated for problems 10.3.1.1, 10.3.1.3 and 10.3.1.5 were initialized using four different methods. They were,

1. Using the FuNN to MLP mapping technique **(method 1)**.
2. Using the FuNN to MLP mapping technique to initialize the input and rule layer weights and the modified Yam and Chow's method to initialize the weights of the last layer **(method 2)**.
3. Using Yam and Chow's method to initialize the weights of the input and rule layer and the modified Yam and Chow's method to initialize the weights of the output layer **(method 3)**.
4. Using random numbers from a normal distribution in the range {-1, +1} **(method 4)**.

The training performance graphs for each problem were similar to the ones in sections 10.3.1.8, 10.3.1.9 and 10.3.1.10. The modified Yam and Chow's initialization method prevented the network from remaining in a local minimum during training.

10.4 Conclusions

The chapter showed how rules developed for a fuzzy neural network architecture could be mapped to the weights of a MLP network structure.

- The process of mapping the output weights from the FuNN to the output weights of the MLP network is an approximation. The output weights in the MLP network were set such that the output node would approach near saturation. This technique is a poor method as it is only an approximation and because the output weights are set manually.
- Yam and Chow's method was used to automate the process of initializing the output weights of the MLP network. Sometimes Yam and Chow's method generated output weights that cause the MLP network to remain in a local minimum during training. For the problems investigated, this was over come by using the modified Yam and Chow's method proposed by the author.
- The experiments showed that the use of rules to initialize a MLP network can improve network performance.
- The experiments showed how the FuNN to MLP mapping could be combined with the modified Yam and Chow's method to initialize the weights of a MLP network. It was observed that a significant improvement in training was not achieved by using the modified Yam and Chow's method. The advantage gained by using this method was the automation of output weight initialisation in the MLP network.

Chapter 11

11 Summary and Conclusions

This thesis describes the research that began with the aim of improving neural network training by incorporating problem knowledge into the initial state of the network. The thesis shows how rules representing problem knowledge can be derived and how these rules can be mapped to the weights of a fuzzy neural network (FuNN) prior to training. As research progressed the mapping procedure was further developed from previous work carried out by a postdoctoral researcher and the related Massey team. The extension to the rule mapping procedure allowed more meaningful relationships to be derived from problems. The rule mapping process was used to initialize the weights of a fuzzy neural network window filter (FuNNWF). Then several image processing problems were investigated using the FuNNWF. The experiments assisted in the development of a methodology based on mapping problem knowledge to a FuNN. The experiments also assisted in defining the appropriate circumstances in which the methodology could be used.

11.1 Summary

The field of fuzzy logic, artificial neural networks and image processing has been explained in the introduction of this thesis. These three fields have been combined to create a fuzzy neural network window filter that was used for the experiments in

this thesis. Problem knowledge was defined and it was shown how rules representing problem knowledge could be generated and mapped to the weights of a fuzzy neural network window filter.

The previous research involved mapping boolean logic rules extracted from problem knowledge to the initial weights of a fuzzy neural network. The research work reported extends the mapping procedure to include conditional rule mapping. The work reported also shows how conditional mapping is more "powerful" than boolean rule mapping. This effect is shown when the same problem is investigated using both boolean and conditional rule mapping. During training the network initialized with conditional rule mapping arrives at the same solution in less iterations during training when compared to the same network initialized using boolean rule mapping.

It is shown that a complete rule set can be generated for problems that are fully determined and fully understood. The complete rule set can be used to initialize all the weights of a fuzzy neural network. In such an ideal situation the network can be used without any training. In a real world environment often problems are not fully understood, nor fully determined. Often complete rule sets cannot be generated for real world problems. Four real world problems were investigated in this thesis.

In the context of this research it is shown how a number of networks can be used either in parallel or sequentially to solve a particular problem. If the networks are used in parallel, each network can be used to process a different section of the problem. If the networks are used sequentially the output from one network can be used as input to the next network. It is shown how problem knowledge can be used to initialize the weights of each network for each ordering.

It is shown when it is appropriate to use problem knowledge to initialize the weights of a fuzzy neural network.

11.2 Conclusion

Fuzzy logic, artificial neural networks and image processing have been combined to generate the fuzzy neural network window filter architecture. It has been shown to be an effective approach for investigating some image processing problems. This thesis investigates the effects of using problem knowledge in a fuzzy neural network for image processing tasks. The effects of using problem knowledge to initialize the weights of a fuzzy neural network before training have been investigated.

Network convergence

Using random numbers to initialize the weights of a neural network may start training from an arbitrary position that may be closer or farther away from an acceptable solution. It is expected that by initializing the weights of the neural network using problem knowledge, the network will be placed closer to an acceptable solution in the solution space. It is shown that for some experiments the network initialized with rules converges to a solution in less iterations when compared to the network initialized with random numbers.

Network solution after training

In the experiments reported here the network initialized with rules converges to a more optimal solution after training when compared to the network initialized with random numbers. The network trained using problem knowledge has better generalisation capabilities as indicated by the error measures. An important characteristic of the network initialized using problem knowledge is that it always converges to a useful solution.

Boolean logic rule vs Conditional rules

The boolean mapping rule procedure was extended to include conditional mapping. Conditional mapping allows better expression of the problem in terms of rules. Thus as expected conditional rules are more "powerful" than boolean logic rules. It has been demonstrated that a network initialized using conditional rules converges to a solution faster compared to a network initialized with boolean logic rules.

Confidence factor (quality factor)

It is shown how a confidence factor can be used to qualify rules. The benefit of using the optimum quality factor for the rules is demonstrated in the experiments. The confidence factor led to the development of an iterative procedure to refine rules. This procedure can be used to fine-tune the rules during network training.

Robustness

Training data obtained from real world problems is sometimes noisy and inconsistent introducing uncertainty. It is shown that in such a situation using problem knowledge to initialize the weights of a fuzzy neural network can compensate for the inconsistency that exists in the training data.

Methodology

The methodology developed maps problem knowledge in the form of rules to the weights of a fuzzy neural network. Four real world problems were investigated where problem knowledge was mapped to a fuzzy neural network before training. For three of these problems the training data was imprecise. For the wane edge detection and the iris edge detection problem the target images were generated by hand as the ground truth data was not available. For the tree crown detection problem the training data was inaccurate, as there was a considerable time difference between aerial image capture and ground truth data collection. It is shown that for these three problems use of problem knowledge had a beneficial effect on network training, as the features under investigation could not be easily discriminated. The features had an inherent fuzziness. For the mammogram calcification detection problem the statistical texture measures provided a high discrimination between the features under investigation. The ground truth data for this problem was also complete and accurate. Thus for this problem the use of problem knowledge did not have a beneficial effect on network training. It can be concluded that the problem knowledge based initialization methodology is most effective when the training data is noisy, unreliable and ambiguous.

11.3 Future work

- **Insertion of global information**

As window filters are local operators the FuNNWF was designed for local processing. The software developed for the FuNNWF can be extended to accommodate global information. Two possible avenues are listed below:

1. Use of additional inputs for global information

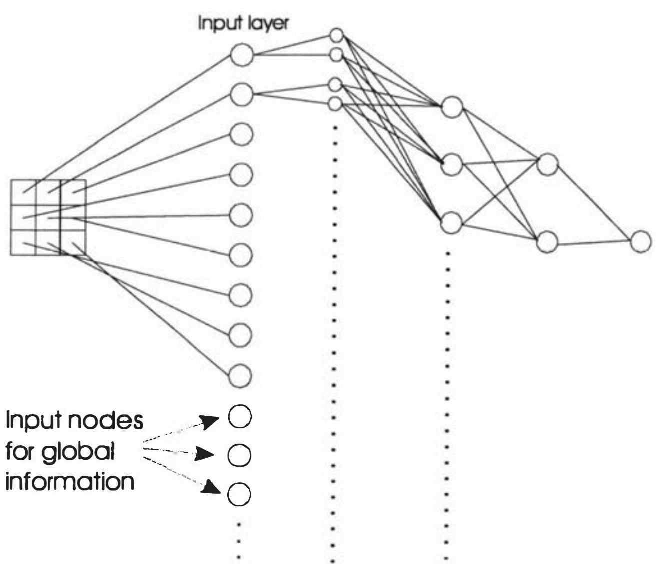


Figure 11.1 Extended input layer of a FuNNWF.

Figure 11.1 shows the input layer of a FuNNWF. The elements in the 3×3 window filter are inputs to the first nine nodes in the input layer. The remaining nodes can be used for global information.

2. Pyramidal vision

The pyramidal machine architecture [Tanimoto 1986] may be regarded as a combination of the architectures of cellular arrays and tree machines. Pyramidal structures that are simultaneously parallel and serial allow gradual formation of more and more global descriptions of image data. A pyramid may be regarded as a collection of images of a single scene at different resolutions. This ensures the use of the most appropriate resolution for required image processing operation. A hierarchy can be introduced for the different levels of resolution. This allows the problem to be solved starting from low spatial

resolutions, having small amounts of data, to higher resolutions until the final result is achieved. It would be interesting to observe the effects of integrating the window filter in to a pyramidal vision system.

- **Automation of rule generation process**

The thesis discusses how problem knowledge can be used to generate rules to initialize the weights of the FuNNWF. Currently, the rules have to be extracted by the user, who may or may not have expertise relating to the problem. Usually for image processing problems only problem knowledge about aspects of images is extracted. There may be more information including structured knowledge. The use of this less apparent problem knowledge could be beneficial for network training. Fuzzy-C means clustering could be an approach that may be used to extract problem knowledge [Yager 1994].

- **Evolutionary algorithms and Evolving Fuzzy Neural Networks (EFuNN)**

Computational intelligence focuses on the use of fuzzy logic (FL), artificial neural networks (ANN) and evolutionary algorithms (EA) to model the behaviour and mechanisms of biologically intelligent organisms. Only two areas, namely FL and ANN have been investigated for the work reported in this thesis. EA could be used to evolve ANN structures [Wolfgang 1999] and discover efficient learning rules [Radi 1999]. It would be interesting to investigate the possibility of integrating EA rule generation methods and the rule intialization method reported in this thesis. EFuNNs [Kasabov 1999] have a five layer structure, similar to the structure of the FuNN. In this new structure nodes and connections are created/connected as data examples are presented. In the FuNN used for the research work, the membership functions were kept fixed during training. In the EFuNN architecture the membership functions can be modified by the network during training. It would be interesting to investigate the effects of using the rule based initialization method on EFuNNs.

- **The rule mapping process**

From the work reported in this thesis, it is apparent that the rule mapping process has a beneficial effect on neural network training. It was shown that rules could compensate for the inconsistencies in training data. The extent of compensation was not investigated. A completely know data set where the inconsistencies could be introduced in a statistically controlled manner would be required to determine a quantitative factor for the extent of compensation. It would also be interesting if a quantitative measure could be placed on the extent of the improvements in network training when conditional mapping is used over boolean logic mapping.

Appendix A

Source code

This section gives a listing of the source code used for the experiments in this thesis. Most of the code was written in Matlab version 5. Some sections were written in C and compiled to a DLL (dynamic link library) which was then called using Matlab.

Network and rule definition file

The following file is a network and rule definition file. The fuzzy neural network structure that will be used for an experiment is defined in this file. The rules that will be used to initialize the weights of the FuNN are also defined in this file.

Example of network and rule definition file

```
ParserVersion2

% Network Information

DataFileName iris001
TypeOfNetwork FNN
NumberOfInputNeurons 33
TotalRuleNodes 10
NumberOfOutputNeurons 1
WindowFilterVerticalSize 3
WindowFilterHorizontalSize 11
Saturation 1
LearningRate 0.8
Momentum 0.8
```

```
Epochs 200
Goal 0

% Fuzzy Membership Function Details

Fuzzification MF=9
DEF=[0,0.4961,0.5078,0.5469,0.6367,0.6863,0.7266,0.8984,1]
Defuzzification MF=2 DEF=[0,1]

Process InitializeMatrix

% Rules

R 1 If I5=M2 I6=M2 I7=M2 I16=M2 I17=M2 I18=M2 I27=M2 I28=M2 I29=M2
I1=M3 I2=M3 I3=M3 I12=M3 I13=M3 I14=M3 I23=M3 I24=M3 I25=M3 I9=M4
I10=M4 I11=M4 I20=M4 I21=M4 I22=M4 I31=M4 I32=M4 I33=M4 Then O1=M2
R 2 If I5=M7 I6=M7 I7=M7 I16=M7 I17=M7 I18=M7 I27=M7 I28=M7 I29=M7
I1=M6 I2=M6 I3=M6 I12=M6 I13=M6 I14=M6 I23=M6 I24=M6 I25=M6 I9=M8
I10=M8 I11=M8 I20=M8 I21=M8 I22=M8 I31=M8 I32=M8 I33=M8 Then O1=M2
R 3 If I5=M6 I6=M6 I7=M6 I16=M6 I17=M6 I18=M6 I27=M6 I28=M6 I29=M6
I1=M5 I2=M5 I3=M5 I12=M5 I13=M5 I14=M5 I23=M5 I24=M5 I25=M5 I9=M8
I10=M8 I11=M8 I20=M8 I21=M8 I22=M8 I31=M8 I32=M8 I33=M8 Then O1=M2

% Calculate Hi & Lo values for matrix

Process ProcessRules
Process ProcessActions
```

Code for Parser

The code listed in this section parses the network and rule definition file to generate a structure that can be used by the programs in section 0 to generate the FuNN.

Function – ParseRuleFile2

```
function r = ParseRuleFile2(fname)
% Function to parse the network and rule definition file
% to generate a network definition structure that can
% be used to generate a fuzzy neural network window
% filter initialized with rules

% ParseRuleFile('FileName')

FilePos=0;

SNN = [];
fid = fopen(fname, 'r');

line = fgetl(fid);
% perform version check on network definition file
```

```
[CurrentWord, Line] = ExtractWord(line);
if upper(CurrentWord) ~= 'PARSERVERSION2'
   fprintf('File being parsed belongs to an older version, please
use PARSER Ver 1\n');
   fclose(fid);
   return;
end

SNN.ParserVersion = '2.0';

while 1
   line=fgetl(fid);
   if ~ischar(line)
      break
   end

   SNN = parse2(line, SNN, fid);
   FilePos = FilePos + 1;
   fprintf('Parsing line %d in parse file \n', FilePos);
end

sname = SNN.DataFileName;
fprintf('Saving parsed file as %s\n', sname);
save(sname,'SNN');

fclose(fid);

r = SNN;
```

Function – ExtractWord

```
function [r1, r2]=ExtractWord(Line)
% Function to extract a word from a line in the
% network definition file

% r1  - returns current word &
% r2  - remaining line

[m,n] = size(Line);
CurrentPos = 1;
CurrentWord = '';

if ~isempty(Line)

while Line(CurrentPos)==' '
      CurrentPos = CurrentPos +1;
end

while 1
   if Line(CurrentPos)==' '
      break;
   end

   CurrentWord = [CurrentWord Line(CurrentPos)];
   CurrentPos = CurrentPos + 1;
   if CurrentPos > n
      break;
   end
end
```

```
end

r1 = CurrentWord;
r2 = Line(CurrentPos:n);
```

Function – ExtractNumber

```
function [r, t] = ExtractNumber(Word)
% Function to extract and convert a word to a number

m = length(Word);
t = Word(1);
r = str2num(Word(2:m));
```

Function – SplitWord

```
function [left, right, opt]=SplitWord(Word)
% Function to split the word into 3 components
% example:
% i1>M1 can be split into i1, >, M1

m=length(Word);
equalPos=0;

for i=1:m
   if Word(i)=='=' | Word(i)=='<' | Word(i)=='>'
      equalPos = i;
      break;
   end
end

left = Word(1:equalPos-1);
right = Word(equalPos+1:m);
opt  = Word(equalPos);
```

Function – Parser2

```
function r = parse2(Line, NN, fid)
% Function to parse the network definition file

while ~isempty(Line)
   [CurrentWord, Line] = ExtractWord(Line);

   switch CurrentWord
   case 'TotalRuleNodes'
      [CurrentWord, Line] = ExtractWord(Line);
      NN.TotalRuleNodes = str2num(CurrentWord);
   case 'NumberOfInputNeurons'
      [CurrentWord, Line] = ExtractWord(Line);
      NN.NumberOfInputNeurons = str2num(CurrentWord);
      NN.IsWindowFilter = 0;
   case 'NumberOfOutputNeurons'
      [CurrentWord, Line] = ExtractWord(Line);
      NN.NumberOfOutputNeurons = str2num(CurrentWord);
```

```
    case 'WindowFilterVerticalSize'
       [CurrentWord, Line] = ExtractWord(Line);
       NN.WFVSize = str2num(CurrentWord);
       NN.IsWindowFilter = 1;
    case 'WindowFilterHorizontalSize'
       [CurrentWord, Line] = ExtractWord(Line);
       NN.WFHSize = str2num(CurrentWord);
    case 'Saturation'
       [CurrentWord, Line] = ExtractWord(Line);
       NN.Saturation = str2num(CurrentWord);
       case 'DataFileName'
       [CurrentWord, Line] = ExtractWord(Line);
       NN.DataFileName = CurrentWord;
    case 'TypeOfNetwork'
       [CurrentWord, Line] = ExtractWord(Line);
       switch upper(CurrentWord)
       case 'MLP'
          NN.TypeOfNetwork = 'MLP';
       case 'FNN'
          NN.TypeOfNetwork = 'FNN';
       end
    case 'LearningRate'
       [CurrentWord, Line] = ExtractWord(Line);
       NN.LearningRate = str2num(CurrentWord);
    case 'Momentum'
       [CurrentWord, Line] = ExtractWord(Line);
       NN.Momentum = str2num(CurrentWord);
    case 'Epochs'
       [CurrentWord, Line] = ExtractWord(Line);
       NN.Epochs = str2num(CurrentWord);
    case 'Goal'
       [CurrentWord, Line] = ExtractWord(Line);
       NN.Goal = str2num(CurrentWord);
    case 'Display'
       [CurrentWord, Line] = ExtractWord(Line);
       NN.Display = str2num(CurrentWord);
    case 'Process'
       [CurrentWord, Line] = ExtractWord(Line);
       switch CurrentWord
       case 'InitializeMatrix'
          NN.RuleMatrix = zeros(NN.TotalRuleNodes,
NN.NumberOfInputNeurons * NN.FuzzyMF);
          NN.ActionMatrix = zeros(NN.DefuzzyMF, NN.TotalRuleNodes);
          if NN.TypeOfNetwork == 'MLP'
             NN.RuleBiasMatrix = zeros(NN.TotalRuleNodes, 1);
          end
          if NN.IsWindowFilter
             if NN.NumberOfInputNeurons ~= (NN.WFHSize*NN.WFVSize)
                   warndlg('Mismatch in Input nodes and Window
Filter size.  FuNN will not work properly!');
                   break;
             end
          end
       case 'ProcessRules'
          NN = ProcessRuleMatrix2(NN);
       case 'ProcessActions'
          NN = ProcessActionMatrix(NN);
       end
    case '%'
       % ignore Comment line
       Line = [];
```

```
    case 'Fuzzification'
        % define the number of membership functions for each node in
 fuzzification layer
        [CurrentWord, Line] = ExtractWord(Line);
              [left, right, opt] = SplitWord(CurrentWord);
        if left == 'MF' & opt == '='
           NN.FuzzyMF = str2num(right);
        end

        % triangular membership function definition
        [CurrentWord, Line] = ExtractWord(Line);
        [left, right, opt] = SplitWord(CurrentWord);
        if left == 'DEF' & opt == '='
           NN.FuzzyMFDEF = str2num(right);
           %tright = str2num(right);
        end

        if size(NN.FuzzyMFDEF,2)~=NN.FuzzyMF
           warndlg('Fuzzy membership function definition error. FuNN
will not work properly!');
        end
        % future expantion
        %NN.FuzzyMFDEF = [];
        %for FMc=1:NN.NumberOfInputNeurons
        %   NN.FuzzyMFDEF = [NN.FuzzyMFDEF tright];
        %end
    case 'Defuzzification'
        % define the number of membership functions in
defuzzification layer
        [CurrentWord, Line] = ExtractWord(Line);
              [left, right, opt] = SplitWord(CurrentWord);
        if left == 'MF' & opt == '='
           NN.DefuzzyMF = str2num(right);
        end

        % triangular membership function definition
        [CurrentWord, Line] = ExtractWord(Line);
        [left, right, opt] = SplitWord(CurrentWord);
        if left == 'DEF' & opt == '='
           NN.DefuzzyMFDEF = str2num(right);
        end

        if size(NN.DefuzzyMFDEF,2)~=NN.DefuzzyMF
           warndlg('DeFuzzy membership function definition error.
FuNN will not work properly!');
        end

    case 'R'
        % Rule Line\
        [CurrentWord, Line] = ExtractWord(Line);
        CurrentRule = str2num(CurrentWord);
        fprintf('Processing Rule %d \n', CurrentRule);

        [CurrentWord, Line] = ExtractWord(Line);
        if upper(CurrentWord)~='IF'
           fprintf('Syntax Error in Rule -%d', Currentrule);
           break;
        end

        ProcessConsequent = 0;
```

```
        while ~isempty(Line)
           [CurrentWord, Line] = ExtractWord(Line);

           if upper(CurrentWord(1,1:3)) == 'THE'
              ProcessConsequent = 1;
           end

           switch ProcessConsequent
           case 0
              [left, right, opt] = SplitWord(CurrentWord);
              left = ExtractNumber(left);
              right = ExtractNumber(right);
              if (isempty(left) | isempty(right))
                 fprintf('Error!!!! Input or membership funtion not
a numeric\n');
              end
              switch opt
              case '='
                 NN.RuleMatrix(CurrentRule, ((left-
1).*NN.FuzzyMF+1):(left.*NN.FuzzyMF)) = ones(1,NN.FuzzyMF).*-1;
                 NN.RuleMatrix(CurrentRule, ((left-1).*NN.FuzzyMF +
right)) = 1;
              case '<'
                 NN.RuleMatrix(CurrentRule, ((left-
1).*NN.FuzzyMF+1):(left.*NN.FuzzyMF)) = ones(1,NN.FuzzyMF).*-10;
                 NN.RuleMatrix(CurrentRule, ((left-
1).*NN.FuzzyMF+1):((left-1).*NN.FuzzyMF + right)) = 10;
              case '>'
                 NN.RuleMatrix(CurrentRule, ((left-
1).*NN.FuzzyMF+1):(left.*NN.FuzzyMF)) = ones(1,NN.FuzzyMF).*-20;
                 NN.RuleMatrix(CurrentRule, ((left-1).*NN.FuzzyMF +
right):left.*NN.FuzzyMF) = 20;
              end
           case 1
              [CurrentWord, Line] = ExtractWord(Line);
              [left, right] = SplitWord(CurrentWord);
              left = ExtractNumber(left);
              right = ExtractNumber(right);

              NN.ActionMatrix(((left-
1).*NN.DefuzzyMF+1):(left.*NN.DefuzzyMF), CurrentRule) =
ones(NN.DefuzzyMF,1).*-1;
              NN.ActionMatrix(((left-1).*NN.FuzzyMF + right),
CurrentRule) = 1;

           end
        end
    end
end

r=NN;
```

Function – ProcessRuleMatrix2

```
function r = ProcessRuleMatrix2(NN)
% Function to calculate the weight values for the rule layer
% using the NN structure

fprintf('Calculating weights for Rule Matrix.\n');
```

```
% check if calculations are to me made for MLP or FuNN Structure
% calculations for MLP structure not implemented in this function

switch NN.TypeOfNetwork
case 'FNN'
   % FuNN structure calculations
   for i=1:NN.TotalRuleNodes
      ActiveNodes =
size(find(NN.RuleMatrix(i,:)~=0),2)/NN.FuzzyMF;

      % Calculate on and off values
      if ActiveNodes ~= 0
         % Boolean calculation
            BOn  = NN.Saturation/ActiveNodes;
         BOff = -NN.Saturation*(1 + ((ActiveNodes -
1)/ActiveNodes));
         % Conditional less than
         LOn  = NN.Saturation/ActiveNodes;
         LOff = -(2*ActiveNodes-1).*NN.Saturation/ActiveNodes;
         % Conditional greater than
         GOn  = NN.Saturation/ActiveNodes;
         GOff = -(2*ActiveNodes-1).*NN.Saturation/ActiveNodes;

            % Weight values for Boolean logic case
            NN.RuleMatrix(i,find(NN.RuleMatrix(i,:)==1)) = BOn;
            NN.RuleMatrix(i,find(NN.RuleMatrix(i,:)==-1)) = BOff;

            % Weight values for Conditional less than case
            NN.RuleMatrix(i,find(NN.RuleMatrix(i,:)==10)) = LOn;
            NN.RuleMatrix(i,find(NN.RuleMatrix(i,:)==-10)) = LOff;

            % Weight values for Conditional less than case
            NN.RuleMatrix(i,find(NN.RuleMatrix(i,:)==20)) = GOn;
            NN.RuleMatrix(i,find(NN.RuleMatrix(i,:)==-20)) = GOff;
            end
   end
end

r = NN;
```

Function – ProcessActionMatrix

```
Function - function r = ProcessActionMatrix(NN)
% Function to calculate the action layer weight values
% using the NN structure

fprintf('Calculating weights for Action Matrix.\n');

% AND case not implemented
for i=1:NN.TotalRuleNodes
   ActiveNodes =
size(find(NN.ActionMatrix(:,i)~=0),1)/NN.DefuzzyMF;
   if ActiveNodes ~= 0
      BOn = NN.Saturation*(1 + ((ActiveNodes - 1)/ActiveNodes));
      BOff = -NN.Saturation/ActiveNodes;

      a = find(NN.ActionMatrix(:,i)==1);
```

```
        b = find(NN.ActionMatrix(:,i)==-1);
        NN.ActionMatrix(find(NN.ActionMatrix(:,i)==1),i) = BOn;
        NN.ActionMatrix(find(NN.ActionMatrix(:,i)==-1),i) = BOff;
    end
end

r = NN;
```

Code for FuNN

The code listed in this section extends the MLP network in the Matlab Neural Network toolbox to a FuNN.

Function – CreateFuNN

```
function [r] = CreateFuNN(NN)
% function to create a FuNN network

% r   - returns a structure initialized using NN
% NN  - neural network definition structure
NN = InitializeFuzzificationMatrix(NN);
NN = InitializeDefuzzificationMatrix(NN);
NN = InitializeMLPSection(NN);

r = NN;
```

Function – InitializeFuzzificationMatrix

```
function [r] = InitializeFuzzificationMatrix(NN)
% function to initialize Symbol Layer Weights

% Function assigns the membership functions stored in the
structure NN
% to the fuzzification layer.

% create fuzzification matrix
CW =
zeros(NN.NumberOfInputNeurons,NN.NumberOfInputNeurons*NN.FuzzyMF);
k=0;
for i=1:NN.NumberOfInputNeurons
   for j=1:NN.FuzzyMF
      k=k+1;
      CW(i,k)=NN.FuzzyMFDEF(j);
   end
end

NN.FuzzyMatrix = CW;

r = NN;
```

Function – InitializeDefuffificationMatrix

```
function [r] = InitializeDefuzzificationMatrix(NN)
% function to initialize the defuzzification layer

% Function to assign the membership functions stored in the
structure
% NN to the defuzzification layer
NN.DefuzzyMatrix = NN.DefuzzyMFDEF';

r = NN;
```

Function – ExtractRuleMatrix

```
function [r] = ExtractRuleMatrix(FNN)

% Function to extract the weight values of the rule layer fro the
% FNN structure

r = FNN.MLP.IW{1};
```

Function - ExtractActionMatrix

```
function [r] = ExtractActionMatrix(FNN)

% Function to extract the weight values of the action layer from
the
% structure FNN

r = FNN.MLP.LW{2};
```

Function - FuzzifyInput

```
function [r] = FuzzifyInput(Input, FNN)
% function to convert crisp input to fuzzy data

% function used DLL file compiled using VC++ ver 5.

F = cFuzzInputs(Input,
zeros(FNN.FuzzyMF.*FNN.NumberOfInputNeurons,1), FNN.FuzzyMF,
FNN.FuzzyMatrix);

r = F;
```

Function – DefuzzifyTarget

```
function [r] = DefuzzifyTarget(Target, FNN)
% function to convert crisp target to fuzzy data

F = cDeFuzzTargets(Target, FNN.DefuzzyMatrix);
```

```
r = F;
```

Function - InitializeMLPSection

```
function [r] = InitializeMLPSection(NN)
% function to create and initialize MLP section of FuNN

% Function will create the MLP section of the FuNN network
% using the definitions in the NN structure

Mil = generateMLPInputLayer(NN);
% create MLP section using Matlab NN toolbox function
MLP = newff(Mil,[NN.TotalRuleNodes NN.DefuzzyMF],{'logsig'
'logsig'},'traingdx');

% set initial params for MLP section
MLP.biasConnect = [0;0];
MLP.trainParam.goal = NN.Goal;
MLP.trainParam.epochs = NN.Epochs;
MLP.trainParam.lr = NN.LearningRate;
MLP.trainParam.mc = NN.Momentum;
MLP.trainParam.show = 25;

NN.MLP = MLP;

r = NN;

% sub function to create input layer for MLP initialization
function [tr]=generateMLPInputLayer(tNN)
FinalMatrix = [];
s = tNN.FuzzyMF .* tNN.NumberOfInputNeurons;
for i=1:s
   FinalMatrix = [FinalMatrix; [0 1]];
end
tr=FinalMatrix;
% end of sub function
```

Function – ModifyRuleMatrix

```
function [r] = ModifyRuleMatrix(FNN, NRM)
% function to modify rule matrix in MLP section

% r  - returns fuzzy neural network structure
% FNN - fuzzy neural network structure to modify using NRM
% NRM - weight matrix for rule layer

FNN.MLP.IW{1} = NRM;

r = FNN;
```

Function – ModifyActionMatrix

```
function [r] = ModifyActionMatrix(FNN, NAM)
% function to modify the action matrix in MLP section

% r  - returns fuzzy neural network structure
% FNN - fuzzy neural network structure to modify using NAM
% NAM - weight matrix for action layer

FNN.MLP.LW{2} = NAM;

r = FNN;
```

Function – ModifyEpochs

```
function [r] = ModifyEpochs(FNN, E)
% function to modify training epochs

% r   - modified fuzzy neural network structure
% FNN - fuzzy neural network structure to modify using E
% E   - number of epochs

FNN.MLP.trainParam.epochs = E;

r = FNN;
```

Function – ProcessImageUsingFuNN

```
function [r] = ProcessImageUsingFuNN(Pimage, FNN)
% function to process process image using FuNN

% convert image to double precision
Pimage = double(Pimage);

Soutput=[];
[m,n]=size(Pimage);

hW = waitbar(0,'Processing image ...');
if m<n      % if height is less than width then process image row
by row
   %fprintf('Processing rows\n');
   for PICounter=1:m-FNN.WFVSize+1
      Vinputs=[];
      % normalize values between 0 - 255
      Cinputs=Pimage(PICounter:PICounter+FNN.WFVSize-1,1:n)./256;
      [V,T]=processArea(Cinputs, Cinputs, FNN.WFVSize,
FNN.WFHSize);
      % process vector using trained FuNN
      s = SimFuNN(FNN, V);
      Soutput=[Soutput s'];
      waitbar(PICounter/(m-FNN.WFVSize+1), hW);
   end
   Soutput = Soutput';
```

```
else        % if height is greater than width then process image
column by column
   %fprintf('Processing Columns\n');
      for PICounter=1:n-FNN.WFHSize+1
      Vinputs=[];
      % normalize values between 0 - 255
      Cinputs=Pimage(1:m, PICounter:PICounter+FNN.WFHSize-1)./256;
      [V,T]=processArea(Cinputs, Cinputs, FNN.WFVSize,
FNN.WFHSize);
      % process vector using trained FuNN
      s = SimFuNN(FNN, V);
      Soutput=[Soutput s'];
      waitbar(PICounter/(n-FNN.WFHSize+1), hW);
   end
end
close(hW);

Soutput = Soutput.*256;

r = Soutput;
```

Function – RuleToFuNN

```
function [r] = RuleToFuNN(FNN)
% function to initialize FuNN using Rule and Action Matrix from
FNN

FNN = ModifyRuleMatrix(FNN, FNN.RuleMatrix);
FNN = ModifyActionMatrix(FNN, FNN.ActionMatrix);

r = FNN;
```

Function – SimFuNN

```
function [r] = SimFuNN(FNN, Input)
% function to simulate FuNN

% fuzzify input
V = FuzzifyInput(Input, FNN);

% simulate MLP section
ALayer=sim(FNN.MLP, V);

% Defuzzification process
% Uses centroid defuzzification
NetOutput=FNN.DefuzzyMatrix'*ALayer;
ZigmaCons=sum(ALayer);
S=zeros(1,size(V,2));
if ZigmaCons~=0
      S=NetOutput./ZigmaCons;
end

r = S;
```

Function – TrainFuNN

```
function [r, tr] = TrainFuNN(FNN, Input, Target)
% function to train FuNN

% r          - returns trained fuzzy neural network
% FNN        - fuzzy neural network to train
% Input      - train FNN using input matrix
% Target     - train FNN using target matrix

FInputs = FuzzifyInput(Input, FNN);
FTargets = DefuzzifyTarget(Target, FNN);

[FNN.MLP, tr] = train(FNN.MLP, FInputs, FTargets);

r = FNN;
```

Function – smat.c

```
int mat2long(int rowNumber,int colNumber,int totalRows)
{
      /* function gets row, column and total rows             */
      /* in matric and returns the location in the     */
      /* contigous array passed to the function.              */

      int t;

      rowNumber;
      t=totalRows*(colNumber)+rowNumber;
      return(t);
}
```

Function – cFuzzyInputs.c

```
/* $Revision: 1.0 $ */
/*
 *
 * cFuzzyInputs.c
 *
 *
 */

#include <math.h>
#include "c:\matlab\extern\include\mex.h"
#include "smat.c"

/* Input Arguments */

#define     INPUTLAYER_IN                  prhs[0]
#define CONDITIONLAYER_IN            prhs[1]
#define CONDITIONSPERNEURON_IN       prhs[2]
#define CONDITIONWEIGHTS_IN          prhs[3]
```

```
/* Output Arguments */

#define FUZZL_OUT              plhs[0]



static void cFuzzInputs(
                    double FuzzL[],
                    double InputLayer[],
                    int nVectors,
                    int INeurons,
                    double ConditionLayer[],
                    int CPNeurons,
                    int CNeurons,
                    double ConditionWeights[]
                    )
{
      int c, k, i, j;
      double x;

      for(c=0;c<nVectors;c++) {
            k=-1;
            for(i=0;i<INeurons;i++) {
                  x=InputLayer[mat2long(i,c,INeurons)];
                  for(j=0;j<CPNeurons;j++) {
                        k=k+1;
                        if
(x==ConditionWeights[mat2long(i,k,INeurons)])
                              FuzzL[mat2long(k,c,CNeurons)] = 1;
                        else {

      if(x>ConditionWeights[mat2long(i,k,INeurons)])
                                    FuzzL[mat2long(k,c,CNeurons)]
= 1 - (x-
ConditionWeights[mat2long(i,k,INeurons)])/(ConditionWeights[mat2lo
ng(i,k+1,INeurons)]-ConditionWeights[mat2long(i,k,INeurons)]);
                              else {
                                    if
(x<ConditionWeights[mat2long(i,k,INeurons)])

      FuzzL[mat2long(k,c,CNeurons)]=1-
(ConditionWeights[mat2long(i,k,INeurons)]-
x)/(ConditionWeights[mat2long(i,k,INeurons)]-
ConditionWeights[mat2long(i,k-1,INeurons)]);
                                    else

      FuzzL[mat2long(k,c,CNeurons)]=0;
                              }
                        }


                  }
            }
      }

  return;
}


/*
```

```
int checkMatrix(double const cMatrix[])
{
      int t;

      t=((!mxIsNumeric(cMatrix) || mxIsComplex(cMatrix) ||
      mxIsSparse(cMatrix)  || !mxIsDouble(cMatrix)));
      return t;
}
*/

void mexFunction(
                 int nlhs,       mxArray *plhs[],
                 int nrhs, const mxArray *prhs[]
             )
{

      /*    Args in           - 4
            InputLayer              - Input vectors to be
fuzzifsied
            ConditionLayer          - Condition Layer
            ConditionsPerNeuron     - Conditions Per Neuron
            ConditionWeights  - Condition Weights

            Args out    - 1
            FuzzL                   - Fuzzified Vectors

      */

  double    *InputLayer, *ConditionLayer, *ConditionsPerNeuron,
*ConditionWeights;
  double    *FuzzL;

  unsigned int    nVectors, CNeurons, INeurons, CPNeurons;

  /* Check for proper number of arguments */

  if (nrhs != 4) {
    mexErrMsgTxt("cFuzzInputs requires 4 input arguments.");
  } else if (nlhs > 1) {
    mexErrMsgTxt("cFuzzInputs requires 1 output arguments.");
  }

  /* Check if matrix is okay. */

  if (!mxIsNumeric(INPUTLAYER_IN) || mxIsComplex(INPUTLAYER_IN) ||
      mxIsSparse(INPUTLAYER_IN)  || !mxIsDouble(INPUTLAYER_IN))
  {
        mexErrMsgTxt("Error using cFuzzInputs.");
  }

  if (!mxIsNumeric(CONDITIONLAYER_IN) ||
mxIsComplex(CONDITIONLAYER_IN) ||
      mxIsSparse(CONDITIONLAYER_IN)  ||
!mxIsDouble(CONDITIONLAYER_IN))
  {
        mexErrMsgTxt("Error using cFuzzInputs.");
  }

  if (!mxIsNumeric(CONDITIONWEIGHTS_IN) ||
mxIsComplex(CONDITIONWEIGHTS_IN) ||
```

```
       mxIsSparse(CONDITIONWEIGHTS_IN)  ||
!mxIsDouble(CONDITIONWEIGHTS_IN))
    {
        mexErrMsgTxt("Error using cFuzzInputs.");
    }

    if (!mxIsNumeric(CONDITIONSPERNEURON_IN) ||
mxIsComplex(CONDITIONSPERNEURON_IN) ||
       mxIsSparse(CONDITIONSPERNEURON_IN)  ||
!mxIsDouble(CONDITIONSPERNEURON_IN))
    {
        mexErrMsgTxt("Error using cFuzzInputs.");
    }




    /* Assign pointers to the various parameters */

    InputLayer = mxGetPr(INPUTLAYER_IN);
    nVectors = mxGetN(INPUTLAYER_IN);
    INeurons = mxGetM(INPUTLAYER_IN);

    ConditionLayer = mxGetPr(CONDITIONLAYER_IN);
    CNeurons = mxGetM(CONDITIONLAYER_IN);

    ConditionsPerNeuron = mxGetPr(CONDITIONSPERNEURON_IN);
    CPNeurons = (int) ConditionsPerNeuron[0];

    ConditionWeights = mxGetPr(CONDITIONWEIGHTS_IN);

    /* Create a matrix for the return argument */

    FUZZL_OUT = mxCreateDoubleMatrix(CNeurons, nVectors, mxREAL);
    FuzzL = mxGetPr(FUZZL_OUT);

    /* Do the actual computations in a subroutine */

    cFuzzInputs(FuzzL, InputLayer, nVectors,
INeurons,ConditionLayer, CPNeurons, CNeurons, ConditionWeights);

    return;
}
```

Function – cDefuzzTargets.c

```
/* $Revision: 1 $ */
/*
 *
 * cDeFuzzTargets
 *
 */

#include <math.h>
#include "c:\matlab\extern\include\mex.h"
#include "smat.c"

/* Input Arguments */
```

```
#define     TARGET_IN                  prhs[0]
#define OUTPUTWEIGHTS_IN       prhs[1]

/* Output Arguments */

#define TLAYER_OUT                     plhs[0]



static void cDeFuzzTargets(
                    double TLayer[],
                    double Target[],
                    int nVectors,
                    int ANeurons,
                    double OutputWeights[]
                    )

{
      int c, i;
      double y;

      for(c=0;c<nVectors;c++) {
            for(i=0;i<ANeurons;i++) {
                  y=Target[c];
                  if (y==OutputWeights[mat2long(i,0,ANeurons)])
                        TLayer[mat2long(i,c,ANeurons)]=1;
                  else if
(y<OutputWeights[mat2long(i,0,ANeurons)])
                              TLayer[mat2long(i,c,ANeurons)]=1-
(OutputWeights[mat2long(i,0,ANeurons)]-
y)/(OutputWeights[mat2long(i,0,ANeurons)]-
OutputWeights[mat2long(i-1,0,ANeurons)]);
                              else if
(y>OutputWeights[mat2long(i,0,ANeurons)])

      TLayer[mat2long(i,c,ANeurons)]=1-(y-
OutputWeights[mat2long(i,0,ANeurons)])/(OutputWeights[mat2long(i+1
,0,ANeurons)]-OutputWeights[mat2long(i,0,ANeurons)]);
                                    else

      TLayer[mat2long(i,c,ANeurons)]=0;
            }
      }

  return;
}


void mexFunction(
                  int nlhs,        mxArray *plhs[],
                  int nrhs, const mxArray *prhs[]
              )
{

      /*    Args in             - 2
            Target                    - Required Target
            OutputWeighs        - Output Weights

            Args out    - 1
```

```
            TLayer                  - DeFuzzified targets

        */

    double    *Target, *OutputWeights;
    double    *TLayer;

    unsigned int    nVectors, ANeurons;

    /* Check for proper number of arguments */

    if (nrhs != 2) {
      mexErrMsgTxt("cDeFuzzTarget requires 3 input arguments.");
    } else if (nlhs > 1) {
      mexErrMsgTxt("cDeFuzzTargets requires 1 output arguments.");
    }

    /* Check if matrix is okay. */

    if (!mxIsNumeric(TARGET_IN) || mxIsComplex(TARGET_IN) ||
        mxIsSparse(TARGET_IN)   || !mxIsDouble(TARGET_IN))
    {
      mexErrMsgTxt("Error using cDeFuzzTargets.");
    }

    if (!mxIsNumeric(OUTPUTWEIGHTS_IN) ||
mxIsComplex(OUTPUTWEIGHTS_IN) ||
        mxIsSparse(OUTPUTWEIGHTS_IN)   ||
!mxIsDouble(OUTPUTWEIGHTS_IN))
    {
      mexErrMsgTxt("Error using cDeFuzzTargets.");
    }

    /* Assign pointers to the various parameters */

    Target = mxGetPr(TARGET_IN);
    nVectors = mxGetN(TARGET_IN);

    OutputWeights = mxGetPr(OUTPUTWEIGHTS_IN);
    ANeurons = mxGetM(OUTPUTWEIGHTS_IN);

    /* Create a matrix for the return argument */

    TLAYER_OUT = mxCreateDoubleMatrix(ANeurons, nVectors, mxREAL);

    TLayer = mxGetPr(TLAYER_OUT);

    /* Do the actual computations in a subroutine */

    cDeFuzzTargets(TLayer, Target, nVectors, ANeurons,
OutputWeights);

    return;
}
```

Function – processArea.c

```
/* Version 1.2 */
/*    processArea.c
 *
 *    Author - Sanj Gunetileke
 */

#include <math.h>
#include "c:\matlab\extern\include\mex.h"
#include "smat.c"

/* Input Arguments */

#define     PIMAGES_IN  prhs[0]
#define TIMAGES_IN      prhs[1]
#define WFSIZER_IN      prhs[2]
#define WFSIZEC_IN      prhs[3]

/* Output Arguments */

#define PVECTORS_OUT    plhs[0]
#define     TVECTORS_OUT      plhs[1]



static void processArea(
                  double      PImageS[],
                  double      TImageS[],
                  double      PVectors[],
                  double      TVectors[],
                  int         Pr,
                  int         Pc,
                  int         WFsizeR,
                  int         WFsizeC
                  )
{

      int r, c, l,m, p, t;

      p=0; t=0;
      for(r=0;r<(Pr-WFsizeR+1);r++) {
            for(c=0;c<(Pc-WFsizeC+1);c++) {
                  /* l=m*r+(c) */;
                  for(l=0;l<WFsizeR;l++)
                        for(m=0;m<WFsizeC;m++) {

      PVectors[p]=PImageS[mat2long(r+l,c+m,Pr)];
                              p++;
                  }

                  /* center of window filter */

      TVectors[t]=TImageS[mat2long(r+(int)ceil(WFsizeR/2),c+(int)c
eil(WFsizeC/2),Pr)];
                  t++;
      }
      }
  return;
```

```
}

void mexFunction(
                    int nlhs,       mxArray *plhs[],
                    int nrhs, const mxArray *prhs[]
               )
{

      /*    Args in            - 3
            PImageS            - Section from PImage
            TImageS            - Section from TImage
            WFsizeR            - Window filter Row size
            WFsizeC            - Window filter Column size

            Args out      - 2
            PVectors      - Input Vectors
            TVectors      - Target Vectors

      */

  double     *PImageS, *TImageS;
  double     *PVectors, *TVectors;
  double     *WFsizeRP, *WFsizeCP;
  int NumberOfVectors, WFsizeR, WFsizeC;

  unsigned int     Pm,Pn;

  /* Check for proper number of arguments */

  if (nrhs != 4) {
    mexErrMsgTxt("PROCESSAREA requires 4 input arguments. ( Input
Image, Target Image, Row, Column");
  } else if (nlhs > 2) {
    mexErrMsgTxt("PROCESSAREA requires two output arguments.");
  }

  /* Check if matrix is okay. */

  if (!mxIsNumeric(PIMAGES_IN) || mxIsComplex(PIMAGES_IN) ||
      mxIsSparse(PIMAGES_IN)   || !mxIsDouble(PIMAGES_IN))
      {
    mexErrMsgTxt("Error using PROCESSAREA.");
  }

  if (!mxIsNumeric(TIMAGES_IN) || mxIsComplex(TIMAGES_IN) ||
      mxIsSparse(TIMAGES_IN)   || !mxIsDouble(TIMAGES_IN))
      {
    mexErrMsgTxt("Error using PROCESSAREA.");
  }


  Pm = mxGetM(PIMAGES_IN);
  Pn = mxGetN(PIMAGES_IN);

  WFsizeRP = mxGetPr(WFSIZER_IN);
  WFsizeR = (int) *WFsizeRP;
  WFsizeCP = mxGetPr(WFSIZEC_IN);
  WFsizeC = (int) *WFsizeCP;
  NumberOfVectors = (Pm-(WFsizeR-1))*(Pn-(WFsizeC-1));

  /* Create a matrix for the return argument */
```

```
  PVECTORS_OUT = mxCreateDoubleMatrix(WFsizeC*WFsizeR,
NumberOfVectors, mxREAL);
  TVECTORS_OUT = mxCreateDoubleMatrix(1, NumberOfVectors, mxREAL);


  /* Assign pointers to the various parameters */

  PImageS = mxGetPr(PIMAGES_IN);
  TImageS = mxGetPr(TIMAGES_IN);

  PVectors = mxGetPr(PVECTORS_OUT);
  TVectors = mxGetPr(TVECTORS_OUT);


  /* Do the actual computations in a subroutine */

  processArea(PImageS, TImageS, PVectors, TVectors, Pm, Pn,
WFsizeR, WFsizeC);

  return;
}
```

Miscellaneous functions

The following functions were used in the program scripts for each experiment.

Function – getInputImage

```
function [r, map]=getInputImage(hHandle)

% function to load input image into Input Figure Window
% use InputFigure Window

[filename,path] = uigetfile('*.bmp','Input Image');
[P,pmap] = imread([path,filename]);
P = double(P);
figure(hHandle);
image(P);
colormap(pmap);
r=P;
map = pmap;
```

Function – getTargetImage

```
function r=getTargetImage(hHandle)

% function to load target image to TargetFigure Window

[filename,path]=uigetfile('*.bmp','Target Image');
[T,tmap]=imread([path,filename]);
figure(hHandle);
```

```
image(T);
colormap(tmap);
T=double(T);
r=T;
```

Function – pickCluster

```
function [finalV, finalT] = pickcluster(Pimage, Timage,
figureNumber, WFsize)
% function to pick a cluster from the input image

% figure(figureNumber);

fV=[];
fT=[];

while 1
   Vc=[];
   Tc=[];
   [x1,y1,x2,y2]=getTrainingSegment(figureNumber);
   [Vc,Tc]=getTrainingVectors(x1,y1,x2,y2,Pimage,Timage,WFsize);

   if ~isempty(Vc)
      R=input('Use area (Y/N/Q)','s');
      R=upper(R);
      if (R=='Y')
            fV = [fV Vc];
            fT = [fT Tc];
      end
      if (R=='Q')
            break;
      end
   end
end

finalV = fV;
finalT = fT;
```

Function – getTrainingSegment

```
function [x1,y1,x2,y2]=getTrainingSegment(H)

% funtion to get the area to train using the mouse

figure(H);
[x,y]=ginput(2);
x1=round(x(1));
y1=round(y(1));
x2=round(x(2));
y2=round(y(2));

while ((x2<x1) | (y2<y1)),
   figure(H);
   [x,y]=ginput(2);
      x1=round(x(1));
      y1=round(y(1));
```

```
      x2=round(x(2));
   y2=round(y(2));
end
```

Function – getTrainingVectors

```
function
[i,t]=getTrainingVectors(x1,y1,x2,y2,inputImage,targetImage,
WFsize)

% function to convert area picked to input & target vector format
fprintf('\nGenerating Input Vectors and Target Vectors, Please
wait .......');

% get input vectors from image
Cinputs=inputImage(y1:y2,x1:x2);
Ctargets=targetImage(y1:y2,x1:x2);

if ((x1+WFsize(1,1))>x2) | ((y1+WFsize(1,2))>y2)
   fprintf('Area chosen is too small\n');
   i=[];
   t=[];
else
   % processArea - c mex file
   [V,T]=processArea(Cinputs, Ctargets, WFsize(1,2), WFsize(1,1));
   [m,n]=size(V);
   fprintf('\nTraining Vectors Generated:%d\n',n);
   i=V./255;
   t=T./255;
end
```

Example of an experiment

The following steps show the method in which an experiment can be performed.

Step 1

The first step is to generate a network and rule definition file according to section 0. After the network and rule definition file has been created the network definition structure must be generated.

If the network and rule definition file was named *iris001.txt*, the following command can be used to generate the network definition structure. At the Matlab command prompt type,

>> parseRuleFile2('iris001.txt')

The network definition structure will be created and stored in the file name given in the definition file.

Step 2

Next the following script can be used to perform an experiment.

```
% Experiment to determine the performance of a network with rules

savefilename = 'iriswork001.mat';
load iris001
F = CreateFuNN(SNN);
Pf = figure(1);
Pimage = getInputImage(Pf);
Tf = figure(2);
Timage = getTargetImage(Tf);

[Iv, Tv] = pickCluster(Pimage, Timage, Pf, [F.WFHSize F.WFVSize]);

randn('state',0);
R = SNN.RuleMatrix;
R = R + randn(size(SNN.RuleMatrix))*0.1;
A = SNN.ActionMatrix;
A = A + randn(size(SNN.ActionMatrix))*0.1;
F = ModifyRuleMatrix(F, R);
F = ModifyActionMatrix(F, A);
```

```
F = ModifyEpochs(F, 300);

[F, RulePerf] = TrainFuNN(F, Iv, Tv);

RuleO = ProcessImageUsingFuNN(Pimage, F);
NT = CropTargetToOutput(Timage, RuleO, [F.WFVSize F.WFHSize]);
save(savefilename);
```

The experiment is saved in the file *irisworkOO1.mat*.

References

This contains a list of the references made in the thesis.

[Bazzani 2000] Bazzani A., Brancaccio R. & Riccardo A., Automatic Detection of Clustered Microcalcifications in Digital Mammograms Using an SVM Classifier, European Symposium on Artificial Neural Networks, Belgium, 2000, pp 195-200.

[Bishop 2000] Bishop A. W. Bailey D. G. & Hodgson R. M., Verification by Iris Recognition. Projects, Vol. 9, College of Science, Massey University, New Zealand, ISSN 1172-8426, 2000.

[Chaplin 2000] Chaplin R. I. Gunetileke S. & Hodgson R. M., Initializing Neural Networks with A-Priori Problem Knowledge, *in* Neural Networks for Signal Processing X, Proceedings of the 2000 IEEE Signal Processing Society Workshop, Australia, 2000, pp 165-174.

[Daugman 1999] Daugman J., Recognizing Persons by Their Iris Pattern, *in* Biometrics, Personal Identification in Networked Society, edited by Jain A. K., Bolle R. & Pankanti S., Kluwer Acedemic Publishers, Massechusetts, 1999, pp 103-122.

[Demuth 1998] Demuth H. and Beale M., MATLAB Neural Network Toolbox ver 3, User's Guide, 1998.

[Denoeux 1993] Denoeux T. & Lengelle R., Initializing Back Propagation Networks with Prototypes, Neural Networks, Vol.6, 1993, pp 351-363.

[Emboss 5] Jasc Software, Emboss Filter in PaintShop Pro Ver 5.0, http://www.jasc.com.

[Fogel 1999] Fogel D. B., Fukuda T. & Guan L., Scanning the Special Issue/Technology on Computational Intelligence, Special Issue on Computational Intelligence, Proc. of the IEEE, Vol. 87, No. 9, 1999, pp 1415-1422.

[Haralick 1973] Haralick R. M., Shanmugam K. & Dinstein I., Textural Features for Image Classification, IEEE Transactions on Systems, Man and Cybernetics, Vol. SMC-2, 1973, pp 610-621.

[Haykin 1994] Haykin S., Neural Networks a Comprehensive Foundation, Prentice Hall, New Jersey, ISBN 0-02-352761-7, 1994.

[Heath 1998] Heath M., Bowyer K. W. & Kopans D., Current State of the Digital Database for Screening Mammography, Digital Mammography, Kluwer Acedemic Publishers, 1998, pp 457-460.

[Imaginis] http://www.imaginis.com,

[Iridian T] Iridian Technologies Inc.,
http://www.iriscan.com/questions/q1/iris_recognition_demo.html

[**Kasabov 1996**] Kasabov N. K., Foundations of Neural Networks, Fuzzy Systems, and Knowledge Engineering, MIT Press, Cambridge, Massechusetts, ISBN 0-262-11212-4, 1996.

[Kasabov 1997] Kasabov N. K., Kim J., Watts M. J. & Gray A. R., FuNN/2 - A Fuzzy Neural Network Architecture for Adaptive Learning and Knowledge Acquisition, Information Sciences - Applications, 1997.

[Kasabov 1999] Kasabov N., Evolving Connectionist Systems for On-line, Knowledge-Based Learning: Principles and Applications, Technical Report TR99/02, Department of Information Science, University of Otago, 1999.

[Kim 2000] Kim J., Neuro-fuzzy Methods for Buidling Adaptive Intelligent Systems, Phd thesis, Department of Information Sciences, Otago University, 2000, pp 92-163.

[Kohonen 1989] Kohonen T., Self Organisation and Associative Memory, Third Edition, Springer-Verlag, New York, ISBN 0387513876, 1989.

[Kolen 1991] Kolen J. & Pollack J., Back Propagation is Sensitive to Initial Conditions, Advances in Neural Information Processing Systems 3, 1993, pp 860-867.

[Lee 1988] Lee H. S., Hodgson R. M. & Wood E. J., Texture Measurement for Carpet Wear Assesment, Special Issue of IEEE Transactions, PAMI on Industrial Machine Vision and Computer Vision Technology, Vol. 10, No. 1, 1988, pp 92-105.

[Lodewky 1992] Lodewky F. A., Barnard W. & E., Avoiding False Local Minima by Proper Initialization of Connections, in IEEE Transactions on Neural Networks, Vol. 3, No. 6, 1992, pp 899-905.

[Mammo]
http://maraton.csee.usf.edu/Mammography/DDSM/BCRP/bcrp_calc_01.html.,

[Matlab IPT] Matlab, Image Processing Toolbox, Ver. 2.2.1.

[McLaughlin 1998] McLaughlin J. L., A Tutorial on Texture, Final Year Project Institute of Information Sciences and Technology, Massey University, 1998, pp 8-16.

[Minsky 1969] Minsky M. & Papert S., Perceptrons; An Introduction to Computational Geometry, MIT Press, Cambridge, Massachusetts, 1969.

[Moore 2001] Moore S. K., Better Breast Cancer Detection, IEEE Spectrum, May 2000, pp 51.

[Nguyen 1990] Nguyen D. & Widrow B., Improving the Learning Speed of 2-Layer Neural Networks by Choosing Initial Values of the Adaptive Weights, International Joint Conference on Neural Networks, Vol. 3, San Diego, CA, 1990, pp 21-26.

[Nguyen 1997] Nguyen H. T. & Walker E. A., A First Course in Fuzzy Logic, CRC Press, ISBN 0849394775, 1997, pp 1-15.

[Pugmire 1995] Pugmire R. H., A Neural Network Based Window Filter and its Training for Image Processing Tasks, Phd Thesis, Massey University, 1995.

[Pugmire 1997] Pugmire R. H., Hodgson R. M. & Chaplin R. I., The Properties and Training of a Neural Network Based Window Filter Developed for Image Processing Tasks, *in* Brain Like Computing and Intelligent Information Systems, Edited by Shun-ichi Amari and Nikola Kasabov, Springer-Verlag, Singapore, ISBN 981-3083-58-1, 1997, pp 49-80.

[Radi 1999] Radi A. & Poli R., Genetic Programming Discovers Efficient Learning Rules for the Hidden and Output Layers of Feedforward Neural Networks, *in* Genetic Programming: Proceedings of the Second Europen Workshop, EuroGP '99, Sweden, May 1999.

[Rumelhart 1986] Rumelhart D. E. Hinton G. E. & Williams R. J., Learning Internal Representation by Error Propagation, in Parallel Distributed Processing Volume 1, Cambridge MA, MIT Press, 1986, pp 318-362.

[Russ 1995] Russ J. C., The Image Processing Handbook, 2nd Edition, CRC Press, ISBN 0849325161, 1995.

[Siew 1987] Siew L. H., Texture Measures for Carpet Wear Assesment, ME thesis, University of Canterbury, 1987, pp 25-31.

[Siroki 1998] Siroki M., Neural Network Based Image Processing, Postdoctoral Report, Massey University, 1998.

[Tanimoto 1986] Tanimoto S L., Paradigms for Pyramid Machine Algorithms, in Pyramid Systems in Computational Vision, Springer-Verlag, New York, ISBN 0387171657, 1986, pp 173-194.

[Thimm 1995] Thimm G. & Fiesler E., Neural Network Initialization, *in* Natural to Artificial Neural Computation International Workshop on Artificial Neural Networks, Spain, June 1995, pp 535-542.

[Thimm 1997] Thimm G. & Fiesler E., High Order and Multilayer Perceptron Initialization, IEEE Transactions on Neural Networks, Vol. 8, No. 2, 1997, pp 349-359.

[Wessels 1992] Wessels L. F. A. & Etienne B., Avoiding False Local Minima by Proper Initialization of Connections, IEEE Transactions on Neural Networks, Vol. 3, No. 6, 1992, pp 899-905.

[Weymaere 1994] Weymaere N. & Martens J., On the Initialization and Optimization of Multilayer Perceptrons, IEEE Transactions on Neural Networks, Vol. 5, No. 5, 1994, pp 738-751.

[Wolfgang 1999] Wolfgang G. & Feuring T., Evolving Neural Network Structures by Means of Genetic Programming, *in* Genetic Programming: Proceeding of the Second European Workshop, EuroGP '99, Sweden, May 26-27, 1999.

[Yager 1994] Yager R. R. & Dimitar P. F., Approximate Clustering via the Mountain Method, IEEE Transactions on Systems, Man, and Cybernetics, Vol. 24, No. 8, August 1994, pp 1279-1284.

[Yam 2000] Yam J. Y. F. & Chow T. W. S., A Weight Initialization Method for Improving Training Speed in Feedforward Neural Network, Neurocomputing, Vol. 30 (1-4), 2000, pp 219-232.

Bibliography

This is a list of the works that the author studied during the course of the research.

[Anthony 1999] Anthony M. & Bartlett P.,Neural Network Learning: Theoretical Foundations, Cambridge University Press, Cambridge, New York, ISBN 052157353X, 1999.

[Bouchon 2000] Bouchon-Meunier B., Yager R. R. & Zadeh L. A., Information, Uncertainty and Fusion, Kluwer Academic Publishers, Boston, ISBN 079238590X, 2000.

[Carling 1992] Carling A., Introducing Neural Networks, Sigma, ISBN 1850581746, 1992.

[Choueiki 1999] Choueiki M. H. & Mount-Campbell C. A., Training Data Development with the D-Optimality Criterion, IEEE Transactions on Neural Networks, Vol. 10, No. 1, 1999, pp 56-63.

[Eaton 1992] Eaton H. A. C. & Oliver T. L., Learning Coefficient Dependence on Training Set Size, Neural Networks, Vol. 5, 1992, pp 283-288.

[Fu 1996] Fu L., Learning Capacity and Sample Complexity on Expert Networks, IEEE Transactions on Neural Networks, Vol. 7, No. 6, 1996, pp 1517-1520.

[Golubski 1999] Golubski W. & Feuring T., Evolving Neural Network Structures by Means of Genetic Programming, *in* Genetic Programming: Proceeding of the Second European Workshop, EuroGP '99, Gotenborg, Sweden, 1999, pp 211-220.

[Ho 1188] Ho K. H. L. & Ohnishi N., FEDGE - Fuzzy Edge Detection by Fuzzy Categorization and Classification of Edges, Lecture Notes in Artificial Intelligence 1188, pp 182.

[Keller 1992] Keller M. J. & Tahani H., Implementation of Conjunctive and Disjunctive Fuzzy Logic Rules with Neural Networks, *in* Fuzzy Models for Pattern Recognition: Methods that Search for Structures in Data, IEEE Press, New York, ISBN 0780304225, 1992, pp 513-528.

[Kelly 1996] Kelley A. & Pohl I., C By Dissection, Third Edition, Addison Wesley, CA, ISBN 0-8053-3149-2, 1996.

[Mitra 2000] Mitra S. & Hayashi Y., Neuro-Fuzzy Rule Generation: Survey in Soft Computing Framework, IEEE Transactions on Neural Networks, Vol. 11, No. 3, 2000, pp 748-768.

[Philpot 1997] Philpot D. & Hendtlass T., Ensembles of Neural Networks for Digital Problems, Proceeding of the International Conference on Artificial Neural Nets and Genetic Algorithms, Norwick, UK, 1997, pp 31-34.

[Purushothaman 1997] Purushothaman G. & Karayiannis B., Quantum Neural Networks (QNN's): Inherently Fuzzy Feedforward Neural Networks, IEEE Transactions on Neural Networks, Vol. 8, No. 3, 1997, pp 679-693.

[Rao 1990] Rao A. R., A Taxonomy for Texture Description and Identification, Springer-Verlag, New York, ISBN 0-387-97302-8, 1990.

[Robins 1997] Robins A. & Frean M., Learning and Generalization in a Stable Network, *in* Progress in Connectionist Based Information Systems, Proceedings of

the International Conference on Neural Information Processing and Intelligent Information Systems, 1997, pp314-317.

[RoyChowdhury 1999] RoyChowdhury P., Singh Y. P. & Chansarkar R. A., Dynamic Tunneling Technique for Efficient Training of Multilayer Perceptrons, IEEE Transactions on Neural Networks, Vol. 10, No. 1, 1999, pp 48-55.

[Shimodaira 1994] Shimodaira H., A Weight Value Initialization Method for Improving Learning Performance of the Back Propagation Algorithm in Neural Networks, Proceedings of the 6th International Conference On Tools in Artificial Intelligence, 1994, pp 672-675.

[Sinha 1999] Sinha N. K. & Gupta M. M., Softcomputing & Intelligent Systems: Theory and Applications, Acedemic Press Series in Engineering, San Diego, California, ISBN 0126464901, 1999.

[Spreeuwers 1995] Spreeuwers L. J., van der Zwag B. J. & Heijden F., Context Dependant Learning in Neural Networks, Proceedings of IEE 7th International Conference on Image Processing and its Applications, Manchester, UK, 1995, pp 632-636.

[Takagi 1992] Takagi H. & Hayashi I.,NN-Driven Fuzzy Reasoning, from Fuzzy Models for Pattern Recognition, IEEE Press, 1992, pp 497-512.

[Van Ooyen 1992] Van Ooyen A. & Nienhus B., Improving the Convergence of the Back-Propagation Algorithm, Neural Networks, Vol. 5, 1992, pp 465-471.

[Verikas 1997] Verikas A., Gelzinis A. & Malmqvist K., A Random Search Technique for Training Neural Networks, Progress in Connectionist Based Information Systems, Proceedings, International Conference on Neural Information Processing and Intelligent Information Systems, 1997, pp322-325.

[Watanabe 1997] Watanabe E., Adaptive Determination of the Amount of Forgetting in the Learning Algorithm with Forgetting, *in* Progress in Connectionist

Based Information Systems, Proceedings of the 1997 International Conference on Neural Information Processing and Intelligent Information Systems, Springer-Verlag, Singapore, ISBN 981-3083-64-6, 1997, pp237-240.

[Yager 1994] Yager R. R. & Zadeh L. A., Fuzzy Sets, Neural Networks and Soft Computing,1994.

[Yang 1997] Yang H. H. & Amari S., Training Multi-Layer Perceptrons by Natural Gradient Descent, *in* Progress in Connectionist Based Information Systems, Proceedings of the 1997 International Conference on Neural Information Processing and Intelligent Information Systems, Springer-Verlag, Singapore, ISBN 981-3083-64-6, 1997, pp211-214.

[Yoon 1997] Yoon S. & Lee S., Two Learning Algorithms for Robust Classification with Small and Incomplete Training Data, *in* Progress in Connectionist Based Information Systems, Proceedings of the 1997 International Conference on Neural Information Processing and Intelligent Information Systems, Springer-Verlag, Singapore, ISBN 981-3083-64-6, 1997, pp217-220.

[Yu 1997] Yu X. & Chen G., Efficient Backpropagation Learning Using Optimal Learning Rate and Momentum, Neural Networks, Vol. 10, 1997, pp 517-527.

[Zhao 1997] Zhao S & Dillon T.S., Making Neural Networks More Intelligent by Incorporating Prior Knowledge, *in* Progress in Connectionist Based Information Systems, Proceedings of the 1997 International Conference on Neural Information Processing and Intelligent Information Systems, Springer-Verlag, ISBN 981-3083-64-6, 1997, pp770-773.