City Research Online

City St George's, University of London

Citation: Che, F. N. (1996). Object-oriented analysis and design of computational intelligence systems. (Unpublished Doctoral thesis, City, University of London)

This is the accepted version of the paper.

This version of the publication may differ from the final published version. To cite this item please consult the publisher's version.

Permanent repository link: https://openaccess.city.ac.uk/id/eprint/29367/

Copyright and Reuse: Copyright and Moral Rights remain with the author(s) and/or copyright holders. Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge, unless otherwise indicated, provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way. For full details of reuse please refer to City Research Online policy.

City Research Online: http://openaccess.city.ac.uk/ publications@citystgeorges.ac.uk

OBJECT-ORIENTED ANALYSIS AND DESIGN OF COMPUTATIONAL INTELLIGENCE SYSTEMS

by

Fidelis Ndeh Che

This thesis is submitted for the degree of

Doctor of Philosophy

at

Department of Electrical, Electronics and Information Engineering

City University

August, 1996

City University London

Abstract

Machine learning from data, neuro-fuzzy information processing, approximate reasoning and genetic and evolutionary computation are all aspects of computational intelligence (also called soft computing methods). Soft computing methods differ from conventional computing in that they are tolerant of imprecision, uncertainty and partial truths. These characteristics can be exploited to achieve tractability, robustness and low solution costs when the solution to a complex (in machine terms) problem is required. The principal constituents of soft computing include: Neural Networks, Fuzzy Logic and Probabilistic Reasoning Systems. Genetic Algorithms (GAs), Evolutionary Algorithms, Chaos Theory, Complexity Theory and parts of Learning Theory all come under Probabilistic Reasoning Systems. Hybrid systems can be designed incorporating 2 or more aspects of soft computing that are more powerful than any of the components used in a stand alone fashion. A unified framework is needed to implement and manipulate such systems. Such a framework will allow for easy visualisation of the underlying concepts and easy modification of the resulting computer models. In this thesis, an investigation of the major aspects of computational intelligence has been carried out. The main emphasis has been placed on developing an object-oriented framework for architecting computational intelligence systems. Object models for Neural Networks, Fuzzy Logic Systems and Evolutionary Computation systems have been developed. Software has been written in C++ to realise sample implementations of the various systems. Finally, practical applications and the results of using the Neural Networks, Fuzzy Logic systems and Genetic Algorithms developed in solving real world problems are presented. A consistent notation based on the Object Modelling Technique (OMT) is used throughout the thesis to describe the software architectures from which the computer implementation models have been derived.

TABLE OF CONTENTS

TABLE OF CONTENTS ii
LIST OF FIGURES viii
LIST OF TABLES xiv
ACKNOWLEDGMENTS xv
Declaration xvi
Chapter I: INTRODUCTION 1
1.1 Summary of Main Contributions 5
LIST OF PUBLICATIONS 6
Chapter II: NEURAL NETWORKS 9
2.1 Introduction 9
2.2 A Taxonomy of Neural Network Architectures 9
2.2.1 Classifying Neural Networks 10
2.2.2 Categorisation based on the Arrangement of Neurons 10
2.2.2.1 Feedforward Multilayer Neural Networks 11
2.2.2.2 Feedback Neural Networks 12
2.2.2.3 Cellular Neural Networks 12
2.2.3 New Scheme for Classifying Neural Networks 13
2.2.3.1 Memoryless Neural Models 14
2.2.3.2 Memory Neural Models 14
2.3 Structure of Neural Networks 16
2.4 Discussion and Conclusions 23
Chapter III: LEARNING IN NEURAL NETWORKS 24
3.1 Introduction 24
3.2 Data Representations in Neural Networks 24
3.3 What Neural Networks Learn 26
3.4 Learning algorithms in Neural Networks 28
3.4.1 Associative Learning Networks 29
3.4.1.1 The Linear and Non-Linear Associative Memories 30
3.4.1.2 The Bi-directional Associative Memory (BAM) 31

3.4.1.3 The Hopfield Memory Network 32
3.4.2 Unsupervised Learning Networks 33
3.4.2.1 Competitive Learning 34
3.4.2.2 Clustering Networks 34
3.4.2.3 Vector Quantisation 35
3.4.2.4 SOM Learning 36
3.4.3 Stochastic Learning 38
3.4.4 Supervised Learning 40
3.4.4.1 Decision-based supervised learning 40
3.4.4.2 Approximation-based supervised learning 42
3.4.4.3 The backpropagation learning algorithm for multilayer perceptron networks 44
3.4.4.4 Supervised Learning Parameters 45
3.4.4.5 Speeding up Supervised Learning 48
3.5 Discussion and Conclusions 48
Chapter IV:ANALYSIS AND DESIGN OF NEURAL NETWORKS 48
4.1 Introduction 48
4.2 Software Architecture 49
4.3 System Development 50
4.3.1 Software Life-Cycle Models 51
4.3.1.1 The Waterfall Life-Cycle Model 51
4.3.1.2 Evolutionary Life-Cycle Model 52
4.3.1.3 Spiral Life-Cycle Model 52
4.3.1.4 Prototyping 53
4.3.2 Phases of Software Development 54
4.3.2.1 Requirements Definition 54
4.3.2.2 Analysis 54
4.3.2.3 Design 55
4.3.2.4 Implementation 56
4.3.2.5 Testing 56
4.4 Object-Oriented Systems Development 56

4.4.1 Object-Oriented Analysis 57
4.4.1.1 Finding Objects 57
4.4.1.2 Organising Objects 58
4.4.1.3 Describing Object Interaction 58
4.4.1.4 Defining Operations on Objects 58
4.4.2 Object-Oriented Design 59
4.4.3 Implementation 59
4.4.4 Object-Oriented Testing 59
4.5 Object-Oriented Analysis and Design of Associative Neural Networks 60
4.5.1 Domain of Associative Neural Networks 60
4.5.2 Problem Statement 60
4.5.3 Identifying Objects 60
4.5.4 Organising the Objects 63
4.5.5 Describing Object Interactions 64
4.5.6 Defining Operations on Objects 66
4.5.7 Neural Network Systems Design 68
4.5.8 Implementation and Testing 69
4.6 Discussion and Conclusions 74
Chapter V: APPLICATIONS OF NEURAL NETWORKS 75
5.1 Introduction 75
5.2 Application of Neural Networks to Fault Diagnosis in HVDC systems 75
5.2.1 Introduction 76
5.2.2 The HVDC System 76
5.2.3 Data Pre-processing 82
5.2.4 Fault Diagnostic Neural Network 83
5.2.5 Results 84
5.2.6 Discussion 85
5.2.7 Future directions 86
5.3 Application of Neural Networks to Systems Identification 86
5.3.1 Introduction 86
5.3.2 Problem Scope 87

5.3.3 Neural Network for Systems Identification 88
5.3.4 Simulation Results 92
5.4 Discussion and Conclusions 96
Chapter VI: FUZZY LOGIC 97
6.1 Introduction 97
6.2 Fuzzy Logic Theory 98
6.2.1 Fuzzy Subsets 98
6.2.2 Linguistic Variables 100
6.2.3 Fuzzy Numbers 101
6.2.4 Fuzzy Inference 102
6.2.5 Constructing the Rulebase 105
6.3 Object-Oriented Analysis and Design Fuzzy Inference Systems 107
6.3.1 Introduction 107
6.3.2 Problem statement: The domain of Fuzzy Inference Systems 107
6.3.3 Identifying Objects 108
6.3.4 Organising the objects in the Fuzzy Inference System 109
6.3.5 Determining operations on Objects 110
6.3.6 Design of Fuzzy Inference System 113
6.3.7 Implementation of the Fuzzy Inference Systems 114
6.4 A Fuzzy Inference System for predicting harmonics in AC/DC networks 116
6.4.1 Introduction 116
6.4.2 Constructing the Rulebase 119
6.4.3 Results 130
6.5 Discussion and Conclusions 133
Chapter VII: EVOLUTIONARY COMPUTATION 134
7.1 Introduction 134
7.2 Genetic Algorithms 135
7.2.1 Introduction 135
7.2.1.1 Path based models 135
7.2.1.2 Point based models 136
7.2.1.3 Population based models 136

7.2.2 Features of Genetic Algorithms 136

7.2.3 Object-Oriented Analysis of Genetic Algorithms 140

7.2.3.1 Identifying objects in the GA domain 140

7.2.3.2 Discovering Object Operations 142

7.2.4 Object-Oriented Genetic Algorithm Design 143

7.2.4.1 Object Design 144

7.2.4.2 Implementation 146

7.2.5 Object-Oriented Testing 148

7.3 Genetic Learning Classifier Systems 166

7.3.1 Introduction 166

7.3.2 Classifier Systems 167

7.3.3 Learning in Classifier Systems 168

7.4 Object-Oriented Analysis and Design of Classifier Systems 169

7.4.1 Object-Oriented Analysis of Learning Classifier Systems 169

7.4.1.1 Learning Classifier System: Problem Description 170

7.4.1.2 Identifying Classifier domain objects 170

7.4.1.3 Identifying the relationships between domain objects 171

7.4.2 Object-Oriented Design of Learning Classifier Systems 175

7.4.2.1 Classifier Systems Design 175

7.4.2.2 Object Design 177

7.4.2.3 Describing Object Interactions 178

7.4.2.4 Learning Classifier Systems Implementation 181

7.4.2.5 Object-Oriented Testing 183

7.4.3 Learning Classifier System applied to autonomous vehicle control
in an obstacled environment 184

7.4.3.1 The Environment 184

7.4.3.2 Credit Assignment 186

7.4.4 Preliminary Results 187

7.5 Discussion and Conclusions 189

Chapter VIII:ACHIEVEMENTS AND FUTURE DEVELOPMENTS 190

8.1 Achievements 190

8.2 Suggestions for Improvement 192
8.3 Conclusions 194
8.4 References 196
Appendix A : Key Elements of the Object Modelling Technique (OMT) 208
Analysis Models 209
The Object Model 209
The Dynamic Model 212
Functional Model 214
Design Models 216
System Design 216
Object Design 217
Implementation Model 217
Appendix B : Speed of Convergence of Algorithms 218
Order of Convergence 218
Linear Convergence 218
Average Rates 219
Convergence of Vectors 219
The Method of Steepest (Gradient) Descent 219
Global Convergence 220
Appendix C : Derivation of the Backpropagation Algorithm 221
Appendix D : Object-Oriented Design Methodologies 224
Object-Oriented Software Engineering (OOSE) 224
Booch Object-Oriented Design 226
The Booch Notation 227
Other Object-Oriented Development Methodologies 229
Appendix E : Class Declarations for Associative Neural Network Classes 231

LIST OF FIGURES

Number	*Page*
Figure 2.2-1: Classification of neural networks by neuron arrangement	11
Figure 2.2-2: Block diagram of MLP feedforward neural network	11
Figure 2.2-3: Block diagram of a recurrent feedback neural network	12
Figure 2.2-4: Block diagram of a cellular neural network	13
Figure 2.2-5: Class hierarchy of neural network architectures	15
Figure 2.2-6: Class hierarchy for associative neural networks	16
Figure 2.3-1: Schematic diagram of a simple neuron	17
Figure 2.3-2: Logistic activation function	18
Figure 2.3-3: Hyperbolic tangent activation function	19
Figure 2.3-4: Guassian activation function	19
Figure 2.3-5: Neural network class diagram	20
Figure 2.3-6: Dynamic view of neural network architecture	21
Figure 2.3-7: Context diagram of typical neural network system	22
Figure 2.3-8: Data flow diagram of a typical neural network system	22
Figure 3.2-1: Hierarchy of input coding schemes	26
Figure 3.4-1: A simple clustering neural network	35
Figure 3.4-2: Hybrid Network: vector quantisation and feedforward network	36
Figure 3.4-3: Schematic diagram of a SOM network	37
Figure 3.4-4: Mexican-hat function used as neighbourhood function in SOM learning	38
Figure 3.4-5: Energy surface of a hypothetical optimisation problem	40
Figure 3.4-6: A 2-layer fully connected multilayer perceptron network.	45
Figure 4.3-1: Phases in software systems development	50
Figure 4.3-2: The waterfall life-cycle model of software development	51
Figure 4.3-3: Incremental software delivery using evolutionary life-cycle model	52
Figure 4.3-4: Spiral life-cycle model for software development	53
Figure 4.3-5: Overview of system analysis	55
Figure 4.5-1: System object model for an associative neural network	64
Figure 4.5-2: Event trace diagram for a training scenario	65

Figure 4.5-3: Event trace diagram for a *retrieve* scenario 65
Figure 4.5-4: Dynamic model of associative neural network 66
Figure 4.5-5: System context for an associative neural network 67
Figure 4.5-6: Level 1 data flow diagram that describes the operation an associative neural network 67
Figure 4.5-7: Information flow among the different subsystems in a neural network 69
Figure 4.5-8: System Block diagram of neural network 69
Figure 4.5-9: Unit test results for the associative memory network 71
Figure 4.5-10: Unit test results for the BAM network 73
Figure 5.2-1: Configuration of a six-pulse HVDC Converter 76
Figure 5.2-2: Voltage and current waveforms for the different fault conditions 81
Figure 5.2-3: Block diagram of fault diagnostic neural network system 83
Figure 5.2-4: Neural network learns mapping between training patterns and fault types 83
Figure 5.2-5: RMS training error, 11 hidden neurons, learning rate = 0.0175, momentum = 0.02 84
Figure 5.2-6: RMS training error, 21 hidden neurons, learning rate = 0.015, momentum = 0.03 84
Figure 5.3-1: Representations of system dynamics: (a) z-transfer function representation; (b) state-variable representation; (c) state-vector representation. 88
Figure 5.3-2: Architecture of supervised neural network for identification 92
Figure 5.3-3: Plot of state values of the dynamical system 93
Figure 5.3-4: Plot of state vectors for dynamical system 94
Figure 5.3-5: Extract from training data 94
Figure 5.3-6: Training curve using linear activation functions (learning rate = 0.15) 95
Figure 5.3-7: Training curve using sigmoid activation functions (learning rate = 0.15) 95
Figure 6.2-1: Fuzzy subsets with triangular, trapezoidal and gaussian membership functions respectively 99
Figure 6.2-2: Membership functions associated with TALL 101
Figure 6.2-3: Triangular fuzzy number 3 101

Figure 6.2-4: Gaussian fuzzy number 3 101
Figure 6.2-5: An L-R fuzzy number M with value m 102
Figure 6.2-6: Structure of a fuzzy inference system 103
Figure 6.2-7: Fuzzy reasoning: product inference with SUM composition 105
Figure 6.2-8: Fuzzy reasoning: min inference with MAX composition 105
Figure 6.3-1: Domain object model for fuzzy system 110
Figure 6.3-2: Context diagram for fuzzy inference system 110
Figure 6.3-3: Level 1 dataflow diagram for fuzzy inference system 111
Figure 6.3-4: State transition diagram for the fuzzy inference system 112
Figure 6.3-5: Information flow amongst the different subsystems 114
Figure 6.3-6: Declaration of a simple Rule class 115
Figure 6.3-7: Declaration of membership functions and fuzzy Rulebase 115
Figure 6.3-8: Triangular memberships for prototype system 116
Figure 6.3-9: Sampled output membership showing SUM Composition 116
Figure 6.4-1: Switched-capacitor filter 118
Figure 6.4-2: Thyristor controlled converter with switched-capacitor filter. 120
Figure 6.4-3: Plot of total harmonic distortion (THD) for all capacitance values 121
Figure 6.4-4: Variation of harmonic distortion with capacitance for different switching times 122
Figure 6.4-5: Variation of harmonic distortion with resistance 123
Figure 6.4-6: Variation of harmonic distortion with switching time 124
Figure 6.4-7: Fuzzy subsets for the different system parameters 126
Figure 6.4-8: Normalised harmonics Vs pattern number 130
Figure 6.4-9: Predicted harmonics distortion using normalised inputs 131
Figure 6.4-10: Predicted harmonic distortion for normalised inputs for all patterns 131
Figure 6.4-11: Finer partitioning of universe of discourse for capacitance 132
Figure 6.4-12: Even finer partitioning of universe of discourse for capacitance 132
Figure 7.2-1: Pie charts comparing the different selection mechanisms 137
Figure 7.2-2: Schematic diagram depicting the reproduction process 138
Figure 7.2-3: One point crossover 139
Figure 7.2-4: Two point crossover 139

Figure 7.2-5: Uniform Ccossover 139
Figure 7.2-6: Genetic algorithm domain object model 141
Figure 7.2-7: Genetic algorithm state transition diagram 142
Figure 7.2-8: System object model of genetic algorithm 144
Figure 7.2-9: Object interaction diagram for CREATE scenario 145
Figure 7.2-10: Object interaction diagram for RUN scenario 145
Figure 7.2-11: Fully specified object structures for some GA objects 146
Figure 7.2-12: Sample C++ implementation of Alphabet 147
Figure 7.2-13: Sample C++ implementation for generalisation/specialisation relationships 148
Figure 7.2-14: Results of testing the GA Alphabet class 149
Figure 7.2-15: Test results for individual object 150
Figure 7.2-16: Results of testing on Chromosome class 151
Figure 7.2-17: Genetic algorithm input screen showing the different options 152
Figure 7.2-18: Plot of gaussian objective function 153
Figure 7.2-19: Maximum fitness vs. generation for random population search (crossover =0, mutation =1, roulette wheel selection, generational replacement) 154
Figure 7.2-20: Maximum fitness vs. generation for random search (crossover =0, mutation =1, tournament selection, generational replacement) 154
Figure 7.2-21: Variation of fitness vs. generation for random search (crossover = 0, mutation = 1, random selection, generational replacement) 155
Figure 7.2-22: Variation of maximum and average fitnesses with generation (crossover = 0.9, mutation = 0.1, roulette selection, one point crossover, generational replacement) 155
Figure 7.2-23: Variation of maximum and average fitnesses with generation (crossover = 0.9, mutation = 0.01, roulette selection, two point crossover, generational replacement) 156
Figure 7.2-24: Variation of maximum and average fitnesses with generation

(crossover = 0.7, mutation = 0.1, tournament selection,
two point crossover, generational replacement) 156
Figure 7.2-25: Variation of maximum and average fitnesses with generation
(crossover = 0.7, mutation = 0.1, roulette selection,
two point crossover, partial replacement) 157
Figure 7.2-26: Variation of maximum and average fitnesses with generation
(crossover = 0.9, mutation = 0.1, tournament selection,
two point crossover, elitist replacement) 157
Figure 7.2-27: Variation of maximum and average fitnesses with generation
(crossover = 0.9, mutation = 0.1, tournament selection,
uniform crossover, generational replacement) 158
Figure 7.2-28: Objective function plot for u1 = 3, u2 = 2, and s = 0.15 159
Figure 7.2-29: Behaviour of the objective function in the neighbourhood of the
function maximum. 160
Figure 7.2-30: Random search with generational replacement 161
Figure 7.2-31: Random search with elitism 161
Figure 7.2-32: Variation of maximum and average fitness values with generation
(crossover = 0.8, mutation = 0.1, roulette selection,
two point crossover, generational replacement) 163
Figure 7.2-33: Variation of maximum and average fitness values with
generation (crossover = 0.8, mutation = 0.1, tournament
selection, twopoint crossover, generational replacement) 163
Figure 7.2-34: Variation of maximum and average fitness values with
generation (crossover = 0.8, mutation = 0.1, tournament selection,
one point crossover, generational replacement) 165
Figure 7.2-35: Variation of decoded parameters with generation. 165
Figure 7.3-1: Schematic diagram of a learning classifier system 167
Figure 7.4-1: Domain object model of learning classifier system 172
Figure 7.4-2: Learning classifier system: state-transition diagram 173
Figure 7.4-3: Learning classifier system: context diagram 174
Figure 7.4-4: Learning classifier system: level 1 dataflow diagram 175

Figure 7.4-5: Concept map showing the interaction between subsystems in the learning classifier system 176
Figure 7.4-6: Design object model of learning classifier system 177
Figure 7.4-7: Object interaction diagram for *Create LCS* use case 179
Figure 7.4-8: Object interaction diagram for the *Create Classifier* use case 179
Figure 7.4-9: Object interaction diagram for *Match/Execute* cycle use case 180
Figure 7.4-10: Object interaction diagram for *Regenerate Classifier* use case 181
Figure 7.4-11: C++ declaration for chromosome class 182
Figure 7.4-12: C++ declaration for the classifier condition class 182
Figure 7.4-13: C++ declaration for classifier class 183
Figure 7.4-14: Object modelling diagram describing the environment 185
Figure 7.4-15: Encoding of the sensor information 185
Figure 7.4-16: Direction encoding of effector messages 186
Figure 7.4-17: Calculating the payoff 187
Figure 7.4-18: A sample environment 188
Figure 7.4-19: Autonomous vehicle movement 189
Figure 8.2-1: Classification meta model for computational intelligence systems 193
Figure A-1: OMT process pverview 208
Figure A-1: Multilayer perceptron neural network trained by backpropagation 221
Figure A-1: A use case model in OOSE 224
Figure A-2: OOSE process overview 225
Figure A-3: Object types in OOSE 225
Figure A-4: An example object interaction diagram in objectory 226
Figure A-5: Booch process overview 226
Figure A-6: An example of a Booch class diagram showing the different relationships 227
Figure A-7: A Booch state diagram 228
Figure A-8: CRC Process overview 229
Figure A-9: Fusion process of object-oriented development methods 230

LIST OF TABLES

Table 4.5-1: Associative neural network domain objects 61

Table 5.2-1: Representation of each fault type in the training/test sets 82

Table 5.2-2: Training results 85

Table 6.3-1: Objects as nouns in the problem description 108

Table 6.4-1: Format of Rule file 128

Table 6.4-2: The Rulebase for inferring harmonics distortion 129

Table 7.2-1: Comparison between roulette wheel and tournament selection mechanisms 137

Table 7.2-2: Best of generation fitness and (x, y) values vs. generation number for genetic algorithm search. 162

Table 7.4-1: Objects in the learning classifier system 170

ACKNOWLEDGMENTS

The author wishes to thank his wonderful supervisor Dr L. L. Lai for his help and guidance throughout the course of this work.

A whole bunch of thanks to my benefactor Mr Owen Farrelly, for rescuing me financially on more than one occasion and in more ways than I could possible thank him for.

Thanx.

Special thanks to Dr Barbara Hienzen for all the prep talk and those vitamins.

Special thanks also to Mr Mark Littman QC, Miss Margaret Misodi and Dr Peter Rajroop for their support.

Final word of thanks to Mr Harald Braun for help on those tricky 3-D plots.

THIS THESIS IS DEDICATED TO MY LATE FATHER, PAPA VINCENT CHE WHOSE TRAGIC AND UNTIMELY DEATH HALFWAY THROUGH THE WRITEUP ONLY SERVED TO INCREASE MY DETERMINATION TO SEE IT THROUGH TO THE END.

HE WOULD HAVE LIKED THAT...

Declaration

The author hereby grant powers of discretion to the City University Librarian to allow this thesis to be copied in whole or in part without further reference to the author. This permission covers only single copies made for study purposes, subject to normal conditions of acknowledgement.

Chapter 1

INTRODUCTION

This thesis is the result of research that grew out of the practical need for ways in which intelligent systems could be applied to the solution of real problems in the field of Power Systems Engineering and Control in the Energy Systems Group at City University. The intelligent systems methodologies that were being considered included Neural Networks, Fuzzy Logic, Genetic Algorithms, Evolutionary Programming and Chaos Theory.

At the time when this research was first started, there were a number of significant papers, textbooks and PhD dissertation theses that were considered to be an embodiment of the knowledge in the area of neural networks. The most notable of these are given in references [1-15]. There were also a few conferences and even fewer international journals dedicated solely to the promotion of neural network technology. Neural networks were commonly seen as black box systems and were treated as such. Numerous theories were proposed about what exactly it was that neural networks learnt and how they could be better designed to perform these. Neural network research was very closely associated with research areas in neurobiology such as neuroscience, neuropsychology and the theories of learning and complexity. Neural networks were seen as an attempt to mimic the way the brain works and it was widely believed that a better understanding of artificial neural networks would lead to great discoveries about the working of their biological counterpart, namely the human Brain. There were greatly exaggerated reports in the mainstream scientific journals and magazines about the capabilities of neural network technology and it was some times difficult to separate fact from fantasy. Nevertheless, the fact remains that, neural networks offer solutions to some problems that were hitherto considered to be intractable. This has led to resurgence in research in neural technology and its applications. The pace of research has been breathtaking. There is now a clear distinction between biological neural models and artificial or connectionist neural models. Biological neural models are concerned with networks that mimic biological neural systems such as early vision and audio functions in the brain. Their main objective is to develop synthetic elements for verifying hypothesis concerning biological systems. Artificial or connectionist

models on the other hand are more application driven. For these models the architectures are largely dictated by the needs of a particular problem or application. This thesis is solely concerned with the later, i.e., connectionist or computational neural models. The current popularity of neural networks has mostly been fuelled by developments in connectionist models. Currently, there are neural network applications in fields as diverse as medicine, power systems, finance, industrial process control, signal processing and network management to name but a few. Since 1992, there has been a number of international journals dedicated to the publication of ongoing research and applications in the field of neural networks. The two most common ones include IEEE Transactions on Neural Networks and Journal of Neural Computation. Further more, there has been a constant recurrence of special issues and invited papers on neural networks in mainstream scientific journals such as IEEE ASSP magazine, IEEE Computer, Scientific American, IEEE Transactions on Systems, Man and Cybernetics, IEEE Control magazine, IEEE Transactions on Signal Processing, etc. Finally, there has been a number widely publicised international conferences on neural networks and its applications in recent years. These have all contributed to an even wider interest in the application of neural network technology by industry. Still, there are major hurdles that have to be overcome before neural technology can be adopted, whole sale, by industry. Major problems include the proliferation of neural network paradigms and the black box stigma that is still attached to neural networks. Each neural network paradigm almost inevitably results in a different neural network architecture with different learning modes and range of applications to which it can be successfully applied. With so many different paradigms to choose from, selecting the correct paradigm so that the resulting neural network architecture is suitable for a particular problem requires a tremendous amount of creative insight and luck. Otherwise repeated experiments are needed to arrive at an architecture and a correct set of learning parameters to solve a given problem. This process can be significantly improved if an appropriate set of discriminating features can be found to provide a robust classification of neural network paradigms and architectures. Secondary to this is the use of a standard, yet easy notation for the classification and for communicating design decisions so that "good" designs can be easily communicated and successfully reused. The black box stigma that is attached to neural networks also constitutes a major obstacle. The main reason for this is the fact that the majority of neural network systems exist only as lines of program code in neural network software. Such descriptions of neural network systems are

not practical as a bases for communicating design decisions to users of the neural network. Without a proper basis for making informed decisions about the choice of learning parameters, network size, activation function, etc. users end up resorting to experimentation and blind application of neural networks. The consequences on the long term development and application of neural network technology can be extremely damaging. Here again, a robust software architecture for neural networks supported by a standard set of notational symbols will go a long way to enable end users to make informed decisions about the choice of parameters when neural networks are applied to solve difficult real world problems.

Chapter II of this thesis deals with the problem of classifying neural networks. An appropriate set of discriminating features has been identified on which a sensible classification of neural networks can be made. A standard notation based on the Object Modelling Technique has been proposed as the solution both for depicting the classification and for communicating design decisions about the structure of neural networks.

In Chapter III, learning algorithms in neural networks has been presented. Chapter III is a review chapter that surveys current research in the field of neural networks, major issues associated with neural network learning, and proposed solutions to resolve them. The chapter includes a discussion of the factors which affect the operation of neural networks and the methods which are accepted as standard for resolving them. The chapter also presents an insight into a fundamental problem associated with neural networks; *what neural networks learn*, and its implications to solving real problems.

In chapter IV, an object-oriented approach to analysis and design of neural networks is described. This departs from the conventional approach where neural network systems are treated merely as algorithms without neither structure nor architecture. Neural network designs are described using flow charts or pseudo-code or just equations and are usually incomprehensible especially where very complex algorithms are involved. The construction of an object-oriented software architecture for neural network systems will have a major impact in the communication of neural network architecture and designs to other designers and users alike. The significance of this development on the long term adoption and application of neural networks in industry remains to be seen.

Chapter V presents two distinct approaches to applying neural networks in solving real world problems. The first case study approaches the neural network as a black box which is configured to learn a mapping between fault data and a set of fault conditions in an HVDC system. The identification of faults in HVDC systems in real time is a difficult learning problem to which neural networks have been successfully applied. The application doesn't make use of any *a priori* information about the problem even where this is available. In the second case study, an application specific neural network is developed to identify unknown parameters in linear and non-linear dynamical systems. This approach uses *a priori* information about the problem to aid the design of the neural network. This leads to the concept of application specific neural networks and can result in a higher rate of success than the blind application approach.

In chapter VI, an introduction to fuzzy logic concepts is presented. Despite their current popularity and their robust nature, neural network solutions become impractical when the data available to solve a problem are incomplete or imprecise or both. A large proportion of real world problems fall under this category and with the preclusion of a neural network solution, an alternative is required. Fuzzy Logic has been proposed as a means of dealing with the uncertainty that besets real world problems. Fuzzy logic is a generalisation of conventional logic that has been extended to deal with the concept of imprecision and vagueness. In chapter VI, an object-oriented software architecture for constructing fuzzy inference systems is presented. Along with a sample implementation in the C++ programming language. Finally, the procedure and results of applying fuzzy inference to the prediction of harmonics in AC systems is presented.

Chapter VII deals with the evolutionary computation aspects of computational intelligence. Neural network learning can be regarded as negative hill climbing or gradient descent procedure where the network weights converge to a local minima of the error/performance function surface. For extremely difficult learning problems, the performance function surface is usually neither continuos nor convex/concave. This makes it impossible for hill climbing procedures to converge. Even where the performance function is continuos, there are usually more than one optima so that the convergence of the neural network becomes over dependent on the starting position in weight space. The net result is that convergence to a global optima is highly unlikely and the performance of the neural network fluctuates markedly with starting conditions or

becomes trapped in local optima. Evolutionary Computation (EC) strategies are robust search mechanisms modelled on Darwinian theory of evolution and natural selection that can be used to search for global optima in very complicated search spaces. EC strategies include: Genetic Algorithms (GA), Genetic Programming (GP), Evolutionary Strategies (ES), Evolutionary Programming (EP), Classifier Systems (CFS) and several other problem solving strategies based on natural selection and natural evolution. Chapter VII presents an object-oriented architecture for constructing Genetic Algorithms and Genetic Learning Classifier systems. The emphasis in this chapter as with the precious chapters is on effective design communication and design reuse. The chapter demonstrates the reuse of an existing Genetic Algorithm design in the construction of the Genetic Learning Classifier Systems. Successful application of the GA to searching and optimisation problems have been presented along with preliminary results of the application of Genetic Learning Classifier Systems to control an autonomous robot vehicle in an obstacled environment.

As with any research work, there a few things, that, with the benefit of hindsight could have been done much better and there are others which, given more time and enough resources can eventually be achieved. Chapter VIII presents ways in which this research work can been improved and provides suggestions about future research that can be undertaken to realise the full potential of this research.

1.1 Summary of Main Contributions

The word "Architecture" has become a mainstay in the construction of robust software systems that transcends organisational and national boundaries. If intelligent software systems are to survive beyond public domain and one off (bespoke) applications to make the leap into industrial systems design, a shift in the construction methods and the way in which design ideas are communicated is desperately required. This thesis has proposed a solution to the communication problems using a new set of discriminating features for categorising neural networks. The categorisation uses a descriptive diagrammatic notation based on a standard and yet easy to understand set of notational symbols for describing neural network classification hierarchies. A further major contribution is the development of robust object-oriented architectures for constructing intelligent systems. The conventional approach based on the implementation of very complicated algorithms from flowcharts or algorithm descriptions is replaced by a more durable approach based on

object-oriented analysis and design. These architectures serve as an ideal vehicle for communicating computation intelligence systems' designs relating to both designers and users. Furthermore, neural network, fuzzy logic and genetic algorithm software systems have been developed in C++ which has formed the basis of a number of undergraduate and postgraduate projects at the Energy Systems Group at the City University and also practical applications to the solution of difficult real world problems in the field of power systems engineering and control. Work on this research has led in part or whole to the publications in the next section.

LIST OF PUBLICATIONS

1. **F Ndeh-Che**, L L Lai and K H Chu, 'The design of neural networks with object-oriented techniques', IEE Colloquium on Recent Progress in Object Technology, Dec 1993.
2. **F Ndeh-Che,** 'Application of neural networks to financial decision making', Report to Marks and Spencers Financial Services, 1994.
3. K Ramar, A Koppurajulu, C Venkataseshaiah, L L Lai, **F Ndeh-Che** and K L Lo, 'Power system simulators', IEE Colloquium on Simulation of Power Systems, Digest No 1992/221, London, Dec. 1992.
4. L L Lai, **F Ndeh-Che**, K S Swarup and H S Chandrasekharraiah, 'Fault diagnosis for HVDC systems with neural networks', Preprints of Papers, Vol 9, 12th International Federation of Automatic Control (IFAC) World Congress, July 1993, Australia, 179-182.
5. K S Swarup, H S Chandrasekharraiah, L L Lai and **F Ndeh-Che**, 'Application of neural networks to fault diagnosis for HVDC systems', Neural Networks and Genetic Algorithms, Springer-Verlag, Wien, New York. 1993, 227-234.
6. L L Lai, **F Ndeh-Che**, Tejedo Chari, P J Rajroop and H S Chandrasekharraiah, 'HVDC systems fault diagnosis with neural networks' Proceedings of the 5th European Conference on Power Electronics and Applications, The European Power Electronics Association, Vol. 8, Sept 1993, 145-150.
7. L L Lai and **F Ndeh-Che**, 'A new approach to protecting transmission systems with fault generated noise', Proceedings of the Second International Conference on

Advances in Power System Control, Operation and Management, IEE, Pub No 388, Dec 1993, 170-175.

8. L L Lai, **F Ndeh-Che** and Tejedo Chari, 'Fault identification in HVDC systems with neural networks', Proceedings of the Second International Conference on Advances in Power System Control, Operation and Management, IEE, Pub No 388, Dec 1993, 231-236.

9. L L Lai, **F Ndeh-Che**, K H Chu and W Butt, 'Application of ferrite material to protection', Proceedings of the Power and Energy 94, IEE Japan, July 1994.

10. L L Lai, **F Ndeh-Che** and K H Chu, 'Modelling, design and simulation of generator excitation control system by using object-oriented techniques and a genetic algorithm', The First International Conference on Power Electronic and Motion Control, IEEE, China, June 1994.

11. L L Lai, **F Ndeh-Che** and K H Chu, 'Reactive power control by selecting parameters of excitation control systems using a genetic algorithm', Proceedings of the 10th CEPSI, Vol 3, The Association of the Electricity Supply Industry of East Asia and the Western Pacific, New Zealand, Sept 1994, pp264-274.

12. L L Lai, **F Ndeh-Che**, K H Chu, P J Rajroop and X F Wang, 'Design neural networks with genetic algorithms for fault section estimation', Proceedings of the 29th Universities Power Engineering Conference, Ireland, Vol 2, Sept 1994, 569-599.

13. L L Lai, **F Ndeh-Che** and K H Chu, 'Application of ferrites and high frequency signals to protection power systems', Proceedings of the 10th CEPSI, Vol 3, The Association of the Electricity Supply Industry of East Asia and the Western Pacific, New Zealand, Sept 1994, pp210-220.

14. L L Lai, **F Ndeh-Che** and K H Chu, 'Improving power system stability by selecting parameters of excitation control systems using a genetic algorithm', International Conference on Power System Technology, IEEE, China, Oct 1994.

15. E Georges, L L Lai and **F Ndeh-Che**, 'Implementation of neural networks with VLSI', Fourth International Conference on Neural Networks, IEE, June 1995.

16.L L Lai, **F Ndeh-Che**, H Braun, R Hui and A B Serrano, 'Application of neural networks to predicting harmonics', Sixth European Conference on Power Electronics and Applications, Sept 1995, pp533-538.

17.L L Lai, **F Ndeh-Che**, K H Chu, R Yokoyama and M Zhao, 'Application of Ferrite and High-Frequency Signals to Protecting Power Systems', accepted for the European Transactions on Electrical Power Engineering, VDE VERLAG, Germany.

18.L L Lai and **F Ndeh-Che**, 'An application of neural networks to improving power system stability', IEE Colloquium on Advances in neural networks for control and system. April 1993.

19.**F Ndeh-Che**, 'Neural networks and fuzzy logic', PhD Transfer Report, July 1995.

20.**F Ndeh-Che,** 'Computational Intelligence', Internal Report, Energy Systems Group, 1994.

Chapter II

NEURAL NETWORKS

2.1 Introduction

The power of neural networks lies in their ability to find a general solution to a given problem. Neural networks are massively parallel networks of simple processing elements designed to emulate the functions and structure of the brain. They combine properties such as fault tolerance and robustness to solve very complex problems in polynomial time. Artificial neural networks (ANNs) have their roots in the theory of function approximation and pattern classification. Neural network models are characterised by a variety of factors. This determines the type of network and the range of applications in which they can be successfully used. A unique contribution of this chapter is the use of the Object Modelling Technique (OMT) [16] to provide a clear and unambiguous classification of the neural network architectures and in the description of the static structure and operation of neural networks.

2.2 A Taxonomy of Neural Network Architectures

Intelligent classification is a fundamental part of all science and the same holds for a study of neural networks. An ever present problem in science is to construct meaningful classification of observations in order to facilitate human understanding [17]. Because of the proliferation in the development of neural network models, the need for an intelligent classification of neural networks has become unavoidable. There are too many difficulties inherent in any attempt to classify neural networks. The most prominent of these is the fact that the boundaries that distinguish one neural network paradigm from another are often quite fuzzy. It is very difficult to tell where one paradigm stops and the next begins. Furthermore, there is no such thing as a "perfect" classification although some classifications are definitely better than others. Finally, intelligent classification requires a tremendous amount of creative insight and knowledge about neural networks. There are potentially as many ways to classify neural networks as there are people willing to undertake the task. A further problem to do with classification concerns the use of notations. A "Victorian novel" style textual description of all the different neural network architectures and paradigms will take up whole volumes and is just as ineffective to an

Engineer whose main interest lies in applying neural networks to a real world problem. Diagramming techniques have been used since the early days of computing to express requirements for computer software systems and also in their analysis and design [18,19] and a graphical notation is more appropriate. On the other hand, purely graphical notations will lead to a proliferation of diagrams and symbols which places a great burden on the reader in that they have to remember the meanings of the large number of symbols that make up the diagrams. Having a well-defined and expressive notation makes it easier to communicate design decisions and also facilitates consistency and correctness checking of these decisions using automated tools. The robustness of the notation can depend on its relative independence from specific methods and technology as these are more changeable and evolve more rapidly and unexpectedly [20]. The Object Modelling Technique (OMT) provides a comprehensive notation that make it easier to visualise both the structure and the functions of complex neural network software systems. The key elements of OMT are presented in appendix A.

2.2.1 Classifying Neural Networks

There are a large number of different criteria that can be used for classifying neural networks. In [21, 22], an overview of different neural network paradigms is presented along with data structures and algorithms necessary for software implementation. In this thesis, only two different criteria for classifying neural networks will be considered. The first is based on the arrangement of the neurons and the connection weights linking them. This is a generally accepted standard for classifying neural networks. The second criterion is much broader. It is based on the presence or absence of memory[1] in the neural network and it gives a much better insight into the different types of neural networks.

2.2.2 Categorisation based on the Arrangement of Neurons

A categorisation of neural network architectures can be made based on the arrangement of the neurons and the connection of the weights linking them. This kind of categorisation distinguishes between three different categories of neural networks:

1. Feedforward Neural Networks
2. Feedback Neural Networks and
3. Cellular Neural Networks

Figure 2.2-2 shows a classification hierarchy for different neural network paradigms classified according to the arrangement of their neurons.

---

[1] Memory in the neural network is considered with respect to how data patterns are treated with time.

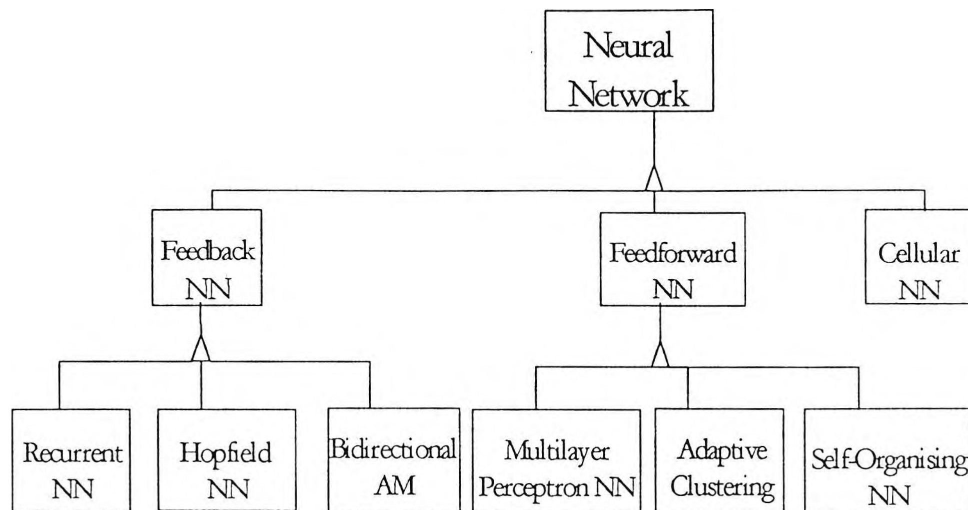


Figure 2.2-1: Classification of Neural Networks by neuron arrangement

2.2.2.1 *Feedforward Multilayer Neural Networks*

In feedforward networks, the neurons are arranged in a feed forward manner, usually in the form of layers. Each neuron may receive inputs from the external environment or from other neurons in preceding layers. Neurons are not allowed to have feedback connections or connections from neurons within the same layer. Feedforward neural networks compute an output pattern in response to a given input pattern. Examples of feedforward neural networks include multilayer perceptron networks (MLP), temporal dynamic neural networks (TDNN) and Hamming networks.

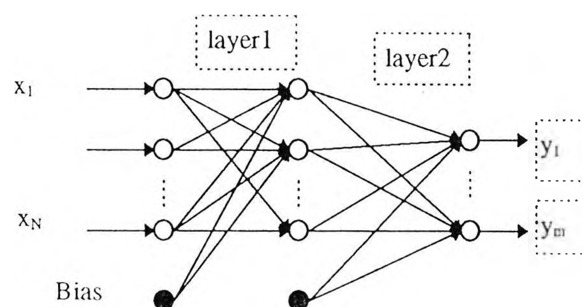


Figure 2.2-2: Block diagram of MLP feedforward neural network

2.2.2.2 *Feedback Neural Networks*

Feedback neural networks, as the name suggests, allow feedback connections between neurons in the same or succeeding layers. Feedback or recurrent neural networks consist of neurons with dynamic building blocks. Their dynamic properties are described by a system of non-linear ordinary differential or difference equations along with an associated

computation (Lyapunov) energy function which is minimised during computation. The evaluation with time of the system of dynamic equations results in a minimisation of the Energy function. Some examples of feedback neural networks include recurrent neural networks, bi-directional associative memory (BAM) and the Hopfield networks.

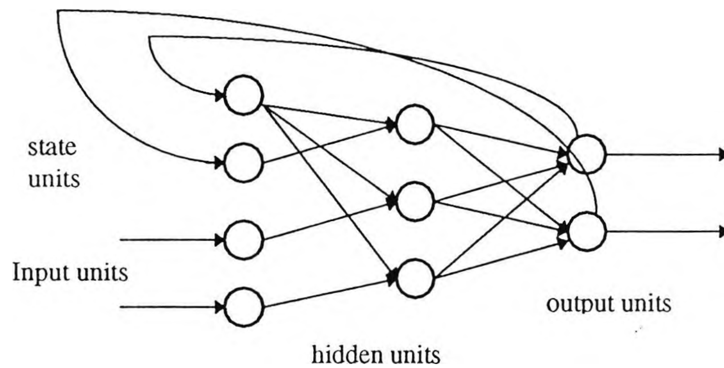


Figure 2.2-3: Block diagram of a recurrent feedback neural network

2.2.2.3 Cellular Neural Networks

Cellular neural are similar to cellular automata. They consist of regularly spaced artificial neurons called cells. The cells communicate only with neurons in their immediate neighbourhood. Adjacent cells interact with each other through mutual lateral interconnections. Cells which are further apart can still affect each other during the propagation of transient signals. The cells are usually organised in a two-dimensional array of rectangular or other regular grid. Every cell in the grid is affected by its own signals and by signals flowing from adjacent cells due to the local connectivity. Mutual interactions causes the processed signals to propagate in time within the whole array of the cellular neural network.

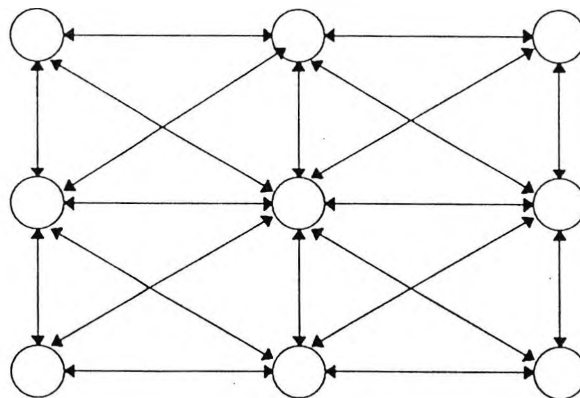


Figure 2.2-4: Block diagram of a Cellular Neural Network

2.2.3 *New Scheme for Classifying Neural Networks*

The classification of neural networks by the arrangement of neurons is severely limiting. For instance, recurrent neural networks are classified as being very similar to BAM and Hopfield networks and much different from temporal dynamic neural networks (TDNN). In practice, recurrent neural networks are structurally and conceptually very similar to TDNN. Furthermore, necessary information that can be used to make informed decisions about the choice of neural network for a particular application is lost in such a classification. In this new scheme for categorising neural networks, the main discriminating criterion is the presence or absence of memory capability in the neural network. At the top level, neural networks are classified into two main categories: memory and memoryless neural networks. The categorisation is successively refined in a tree-like manner to include most of the neural network paradigms that are currently known. Figure 2.2-5 shows a class hierarchy of neural network architectures based on the new classification scheme. A great advantage of this method is the fact that almost any neural network architecture can be classified as a leaf node at the bottom of the hierarchy. Also new nodes can be added as necessary or as new neural network architectures are developed. Furthermore, whole sections can be collapsed or expanded as shown in Figure 2.2-6 to facilitate browsing of particular sections of the class hierarchy. This scheme lends itself to automation support making it easier for neural network designers to navigate the class hierarchy to select a network architecture which is suitable for a particular application.

2.2.3.1 *Memoryless Neural Models*

Memoryless neural models view an input pattern as a random point in n-dimensional feature space. Models of this type are static with respect to time and thus represent the class of static neural networks. Memoryless neural networks can be further divided into supervised or unsupervised learning networks. Supervised learning networks require the presence of a teaching signal at the output in order to adapt or compute the network weights. Using the classification, multilayer perceptrons, radial basis function networks and all the associative memory networks fall under this category. Unsupervised learning networks on the other hand do not require a teaching signal in order to adapt the network weights. Learning in such networks is achieved through a process of competition and self-organisation. Examples of unsupervised neural networks include vector quantiser, the ART (Adaptive Resonant Theory) networks and the Kohonen Self-Organising Feature Map (SOM) network. Still under Memoryless neural networks, a number of hybrid networks

can be classified. These include CMAC (Cerebral Model Articulated Controller) neural networks and the Counter Propagation Network (CPN).

2.2.3.2 Memory Neural Models

Memory neural models incorporate some form of memory unit which is used for processing time dependent (transient or temporal) signals. In this classification, temporal neural models fall under memory neural networks. Temporal neural models are further classified into deterministic or stochastic networks based on the nature of the memory mechanism. In deterministic temporal neural models, the memory mechanism is in the form of time delay units. Neural models of this type are called temporal dynamic models (TDMs). Examples of deterministic temporal neural networks include the Temporal Dynamic Neural Network (DNN) and Recurrent Neural Network (RNN). In stochastic temporal neural networks such as Hidden Markov Models (HMMs), the memory mechanism is in the form of a state transition matrix which is trained to model the temporal behaviour [12, 23, 24].

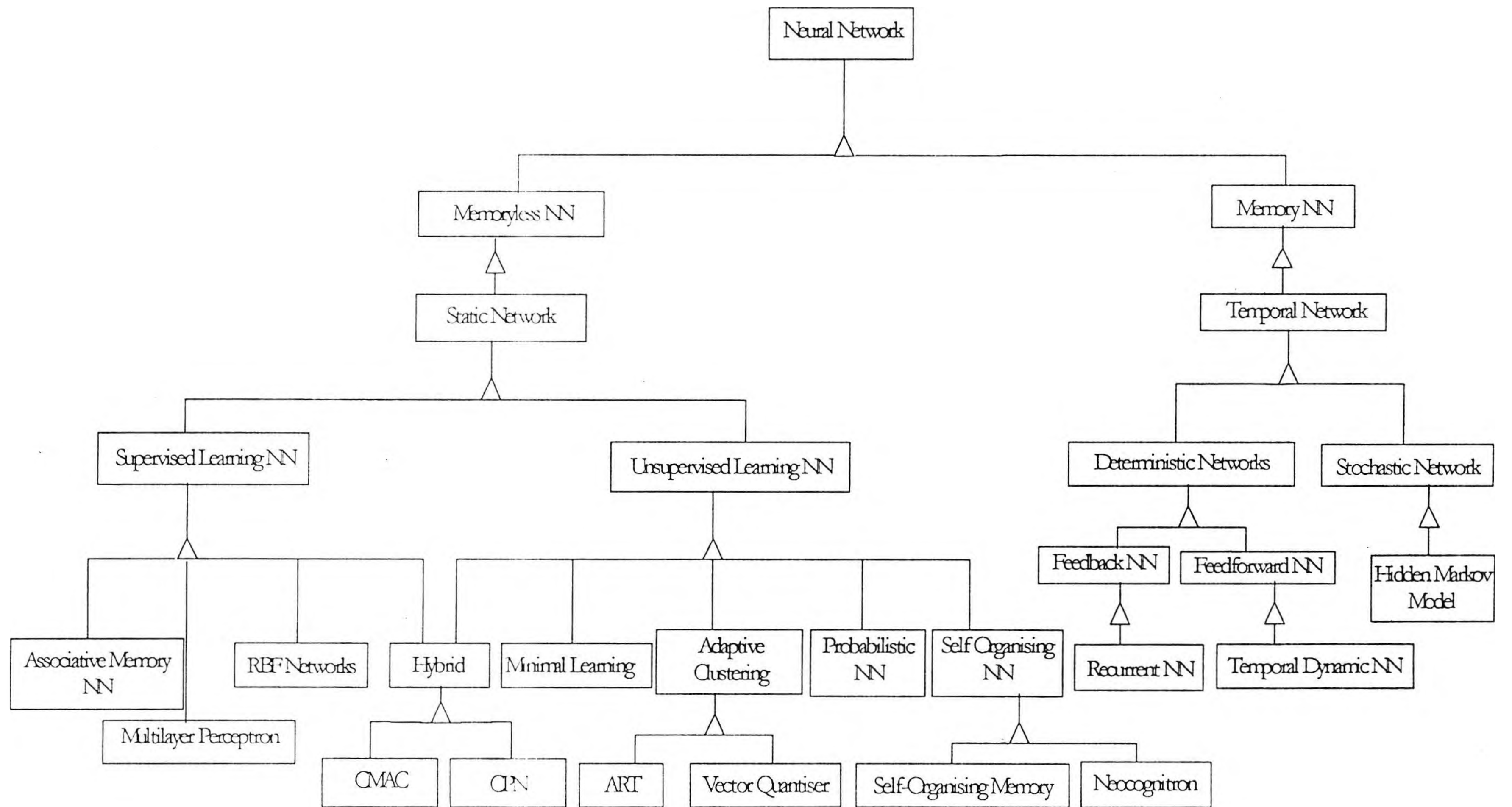


Figure 2.2-5: Class hierarchy of neural network architectures

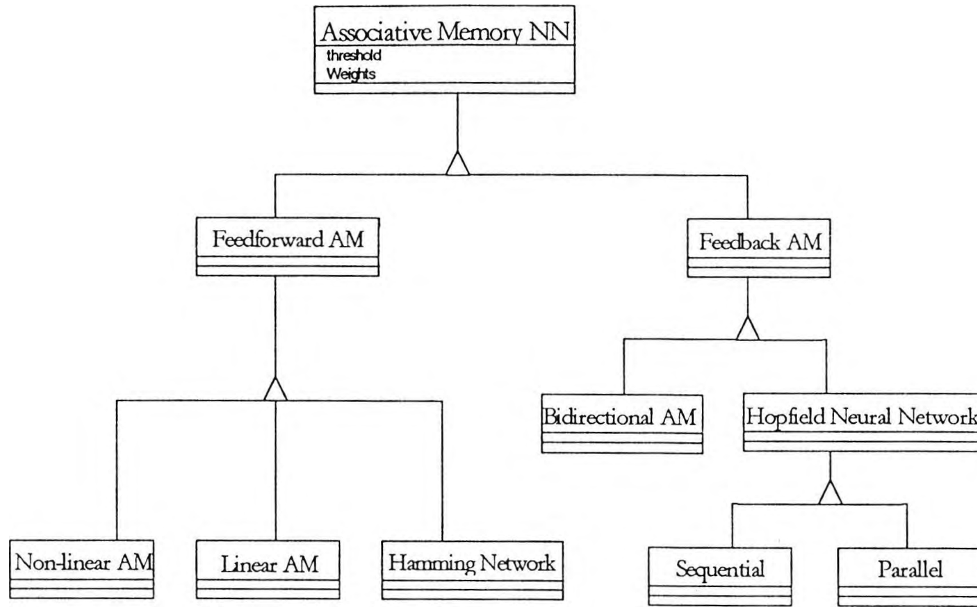


Figure 2.2-6: Class hierarchy for associative neural networks

2.3 Structure of Neural Networks

Every neural network is characterised by a set of coupled differential equations that describe the dynamics of the neural network [25]. These are, the generalised neural network state equation,

$$\frac{d\mathbf{x}}{dt} = \mathbf{F}(\mathbf{x}, \mathbf{W}, \mathbf{U}) \tag{2.3-1}$$

and the general purpose learning equation,

$$\frac{d\mathbf{W}}{dt} = \mathbf{G}(\mathbf{x}, \mathbf{W}, \mathbf{U}) \tag{2.3-2}$$

where $\mathbf{x} = [x_1(t), x_2(t), \ldots, x_n(t)]^{\mathrm{T}}$ is the activation state space vector,

$\mathbf{W} = [w_{ij}(t)]_{n \times n}$ is the weight matrix and

$\mathbf{U} = [u_1, u_2, \ldots, u_n]^{\mathrm{T}}$ is the time independent external inputs.

This system of dynamical equations is guaranteed to converge by a **Lyapunov** or **Energy** function $E = E(\mathbf{x}, \mathbf{W}, \mathbf{U})$ defined on the state space which is monotonically decreasing and bounded from below.

Physically, a neural network consists of layers of artificial neurons connected by synaptic weights. The basic artificial neuron is modelled as a multi-input non-linear processing element with weighted interconnections w_{ij}. The neuron processes its input to produce an output according to the following equation

$$y_j = \Psi\left(\sum_{i=1}^{n} w_{ji} x_i + \theta_j\right) \qquad (2.3\text{-}3)$$

Where Ψ is the activation function,

θ_j is the bias or offset,

x_i are the inputs $(i = 1, 2, \ldots, n)$, n is the number of inputs,

w_{ij} are the synaptic weights and

y_j are the outputs.

The above equation is often written in a more precise manner by treating the bias as a weight connected to an input which is permanently set 1.

$$y_j = \Psi\left(\sum_{i=0}^{n} w_{ji} x_i\right) \qquad (2.3\text{-}4)$$

where $w_{i0} = \theta_j$ and $x_0 = 1$

A schematic diagram of an artificial neuron is shown in Figure 2.3-1.

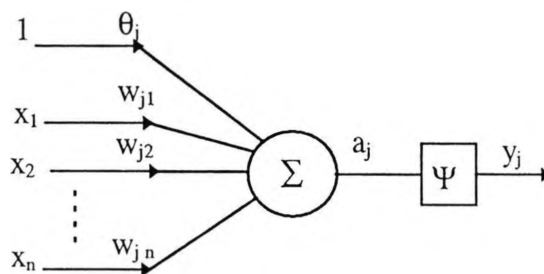


Figure 2.3-1: Schematic diagram of a simple neuron

The neuron activation function determines the kind of information that one neuron can signal to another. Activation functions may be linear or non-linear. Linear activation functions can be used to provide an approximation of the operations of non-linear models. Non-linear activation functions are used to generate variable and complex performance in neural networks. Sigmoidal functions are normally used as neuron activation functions, but, any function that is continuos, differentiable, monotone increasing, step-like can be used. Two examples of sigmoidal neuron activation functions are shown below:

$$\text{logistic function:} \quad y = f(x) = \frac{1}{1 + e^{-kx}} \qquad (2.3\text{-}5)$$

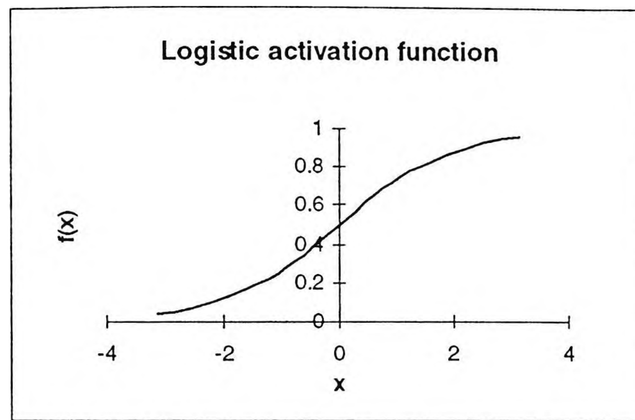


Figure 2.3-2: Logistic activation function

hyperbolic tangent:

$$y = f(x) = \frac{e^{kx} - e^{-kx}}{e^{kx} + e^{-kx}} = \tanh(kx) \qquad (2.3\text{-}6)$$

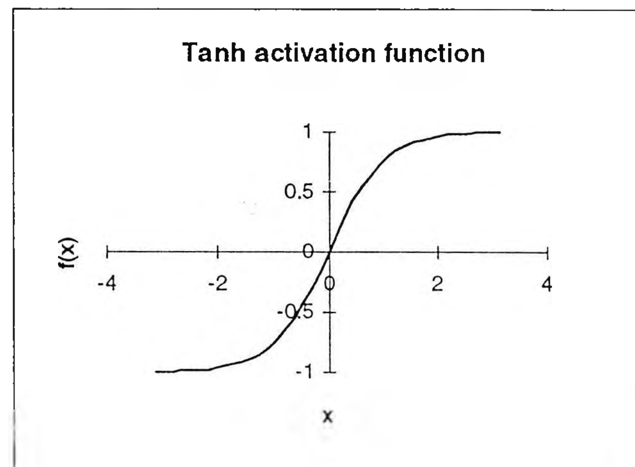


Figure 2.3-3: Hyperbolic tangent activation function

Neurons with sigmoidal activation functions produce real valued outputs which gives the neural network its ability to construct complicated decision boundaries in n-dimensional feature space. This is important because the smoothness of the generalisation function produced by the neurons and hence its classification ability is directly dependent on the nature of the decision boundaries. Another popular class of neurons use Gaussian activation functions [26]. Figure 2.3-4 shows a sample Gaussian activation function with zero mean and a unity standard deviation. The local properties of Gaussian activation functions make them more desirable than sigmoidal activation functions in pattern recognition problems. This is due to the fact that Gaussian neurons are able to produce sharper decision boundaries than their sigmoidal counterparts.

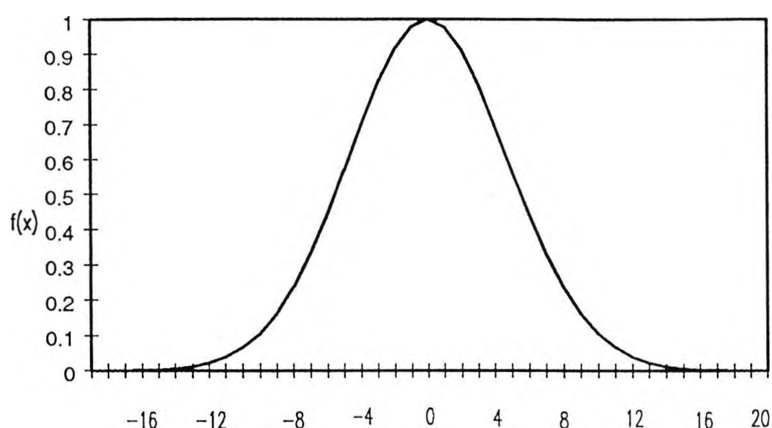


Figure 2.3-4 : Guassian activation function

The input vectors, $\mathbf{x} \in \Re^n$, presented to the neural network can be binary data or scaled/normalised real data ranging between -1.0 and +1.0 or 0.0 and +1.0 depending on the nature of the neuron activation function. Most neural learning algorithms have no self-normalising qualities hence scaling or normalisation is necessary to prevent saturation of the neuron activation function. In some cases such as unsupervised learning, the algorithms actually require that the input data and the initial network weights are normalised before learning can take place. Ordinarily, the neuron weights are initialised to small random numbers when the neural network is first created.

The structure and operation of a general neural network can be described in graphical terms using models in the Object Modelling Technique. Three separate models are provided so that different aspects of the system can be visualised. The models include: the object model, the dynamic model and the functional model. The object model is expressed in a class or object diagram as shown in Figure 2.3-5. This model shows the static structure of a neural network and serves to describe the constituent parts without the details of its operation. The class diagram shows that a neural network consists of one or more neuron layers. Each layer is modelled as a weight matrix containing a pair of activation vectors that represent the input and output activations. The activation vectors are acted on by activation functions which can be linear or non-linear in nature. The neural network uses patterns both during training and testing or validation. Patterns can thus be either training patterns or test patterns. Each pattern is just an aggregation of Vector pairs which in turn consists of between 0 and 2 vectors.

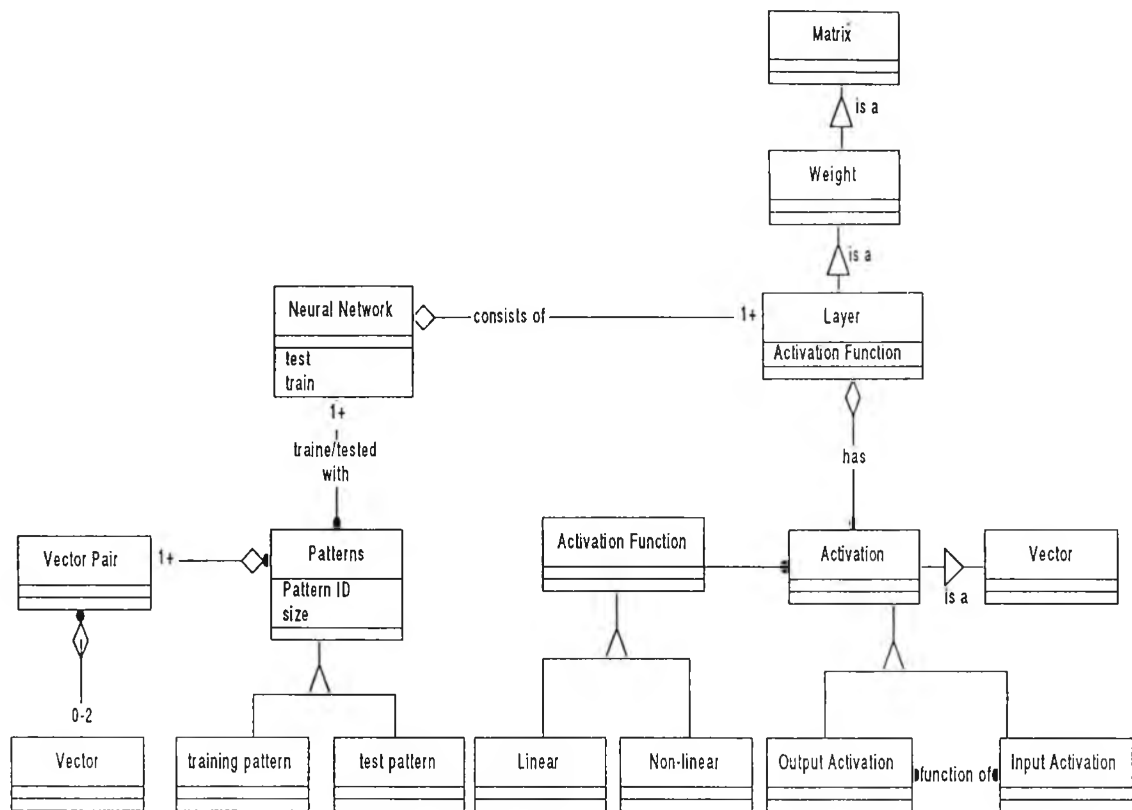


Figure 2.3-5: Neural Network Class Diagram

The second view shown in Figure 2.3-6 represents the dynamic model of the neural network system. This view shows the different states that the system can be in, the events that the system responds to whilst it is in that state and the possible transitions between the states. The state of the system at any one time is the sum total of the values held by all its attributes and associations at that time. For a large system, the number of possible states can be very large. For modelling purposes, all the major states in the main objects are represented. While in a state, the neural network can perform certain activities until an event is received that causes a state transition. For example, in state *Training*, the neural network can *do: train, do: test or do: print error.*

Neural Network

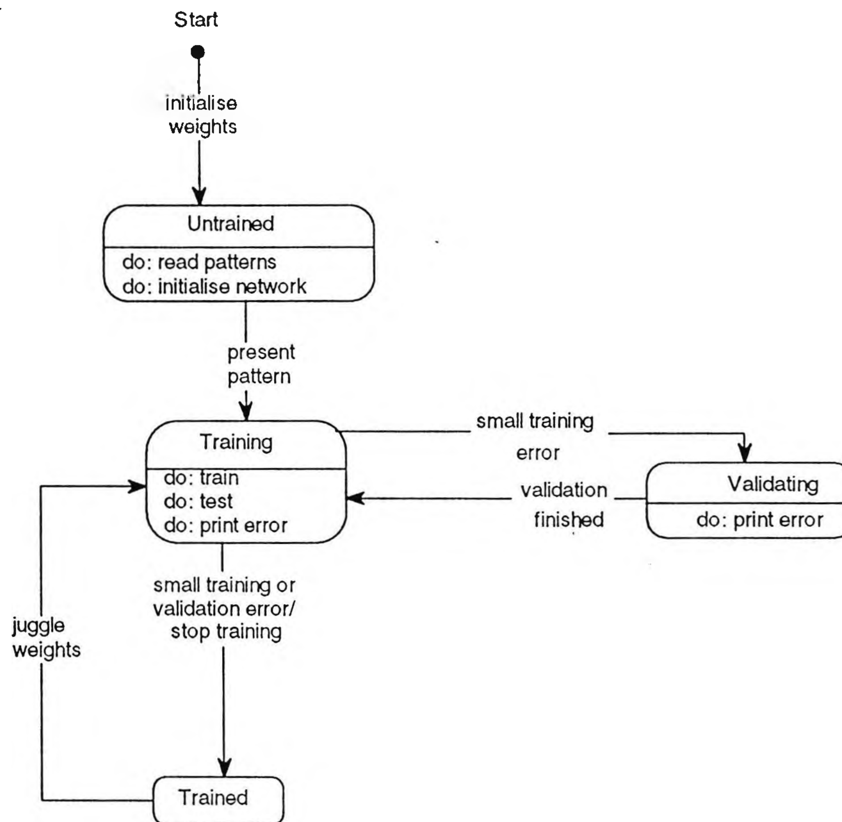


Figure 2.3-6: Dynamic view of Neural Network architecture

The final view in the neural network system description represents the functional model of a neural network system. This model describes the functionality of a neural network in terms of the data it accepts and the transformations that it performs on the data. Functional models are expressed using dataflow diagrams. A dataflow diagram is a directed graph where the nodes represent functions that carry out operations and the edges represent the flow of data or resources between the functions. Dataflow diagrams exist at different levels. The highest level dataflow diagram is the system context diagram. This shows what inputs are required by a system, the sources of the inputs, the main outputs produced, and finally, the main users or sinks of the output. The system context can be successively refined into lower level dataflow diagrams that describe the system in greater detail. At the lowest level, the node transformations in the dataflow diagram degenerate into functions or algorithms that can be utilised in the finished design. The neural network context diagram is shown in Figure 2.3-7. while Figure 2.3-8 shows the level 1 data flow diagram.

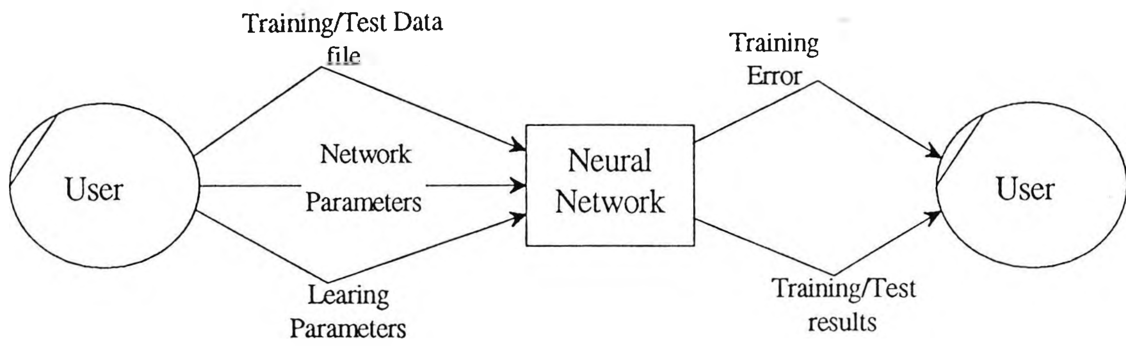


Figure 2.3-7: Context Diagram of Typical Neural Network System

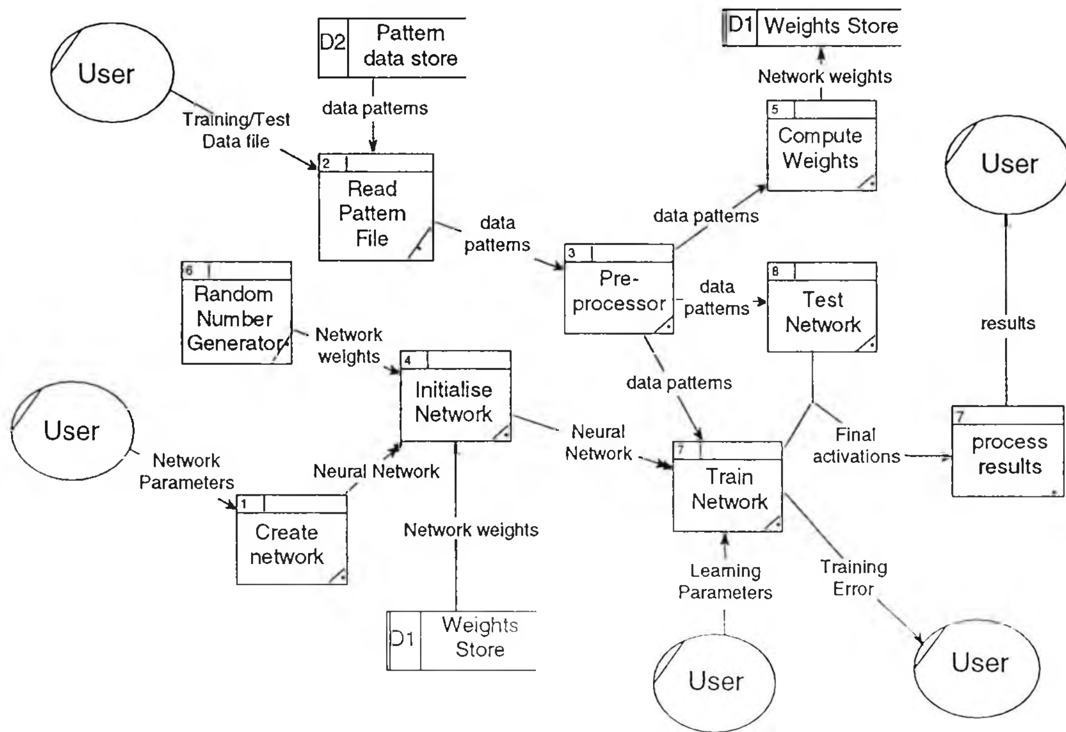


Figure 2.3-8: Data flow diagram of a Typical Neural Network System

2.4 Discussion and Conclusions

Classifying neural network models is difficult because of the wide range of neural network architectures currently available and the diverse and contrasting features or criteria that can be used to describe these models. Furthermore, any kind of classification is very subjective and very dependent on the problem area. Also, where there is no standard set of features or standard notation available to perform the classification as is currently the case, ambiguities are bound to arise. This chapter has attempted to rectify the situation by proposing a new scheme for classifying neural networks using conventional diagramming techniques. The method is based on a standard notation with a few easily recognised symbols as described by the Object Modelling Technique (OMT). Using such a notation, more expressive relationships between different neural network paradigms can be easily described in a diagrammatic form. A set of appropriate discriminating features between the different neural models has be used to create a hierarchy of Neural Network models with significant improvement on the standard method of classifying Neural Networks. It provides a finer level of detail on which informed decisions can be made on the choice of Neural Network architecture for a particular application. It also ensures that similar Neural Networks architectures or models are classified close to each other, which is the ultimate goal of any classification scheme.

Finally, the general structure of a neural network has been described. In describing a complex system, multiple views are required to portray different aspects of the system. In this description, three different views have been presented to completely describe the structure, operation and functionality of a typical neural network. The static view shows the structure of the neural network, the dynamic view show the different states a neural network can be in and the events it responds to. The functional view shows the data that a neural network accepts and the different transformations performed on the data.

Chapter III

LEARNING IN NEURAL NETWORKS

3.1 Introduction

Given a network of neurons connected by weighted unidirectional links, some of the neurons are regarded as input neurons, others as hidden neurons and the rest are output neurons. The object of learning is to train the neural network to respond to an input vector $\mathbf{X}_p$ with a specified output vector $\mathbf{Y}_p$. This can be accomplished by adapting the neural network weights $w = \{w_{ij}\}$ to learn the required mapping between the input and the output vectors. A qualitative mechanism that describes the way in which synaptic weights are modified to reflect the process of learning was first proposed by D.O.Hebb [14]. Questions about what artificial neural networks really learn have been asked since the early days of neural computing [8]. In this review chapter, the question of data representation and learning in neural networks is examined. This chapter discusses data representations and learning in neural networks and presents the different coding mechanisms that enables neural networks to learn meaningful representations. The final part of this chapter presents a discussion of the different neural network learning paradigms and the associated network models and algorithms that make use of these learning paradigms.

3.2 Data Representations in Neural Networks

Much of the brain's power comes not so much from powerful general mapping algorithms but from powerful representation. In general, neural network mappings are easier to establish if similar inputs give rise to similar outputs. Many application areas where neural networks have been used have involved some sort of binary representation of data. This simplifies the analysis and possibly the learning process but can result in similar inputs having quite different representations [27]. Biological neural systems make use of a proportional form of coarse coding by way of locally-tuned but overlapping receptive fields [28]. In coarse coding systems, each neuron responds to a range of input values, in between, but overlapping with those of its neighbours and the representation is said to be distributed [29]. A given input will thus be represented by the relative activity of a number of neighbouring neuron cells. Coarse coding neurons output a 1 value if the stimulus is

within their receptive field, and a 0 otherwise. Such a coding can be self-organised by the neural network. A self-organised coding has no obvious meaning to a human observer, without the use of exhaustive analysis or some form of translation device. On the other hand, coding imposed on the neural network by an implementer can be and usually is meaningless to the neural network. However, if the neural network has some means of interacting with the environment that it is a part of, then, the internal representations may begin to have some meaning. Data representation internally in neural networks can either be local or distributed or both. In a localised representation, inputs are mapped consistently onto a single category. In such a representation, localised constituents of the representation can be associated with specific operations appropriate for the category. In a distributed representation, a single input causes many hidden neurons to become activated. The network can thus encode unknown feature representations with each hidden neuron responding to one or more aspects of features in the input representation. The choice of input representation is thus important in determining the internal representations. Input representations can also be distributed, local or both. A localised representation usually produces a larger feature set but could facilitate training. A distributed representation produces more compact feature vectors but could be difficult to train, especially where similar inputs in state space tend to be represented differently. The input representation depends to a large extent on the coding strategy used. Coding strategies could either be discrete or continuous.

Discrete coding schemes assume that units can only be on or off. As such, they are relatively immune to noise. On the other hand, they are severely limited in resolution and range. Some examples of discrete coding schemes include: value unit encoding and discrete thermometer encoding.

Continuous coding schemes encode the input as a triangular or Gaussian fuzzy number. Triangular fuzzy numbers are specified by their widths which determines the fraction of the number encoded by neighbouring neurons. Gaussian fuzzy numbers are specified with a given standard deviation that determines the spread of the number to the neighbouring neurons. A description of coding schemes is given in [27].

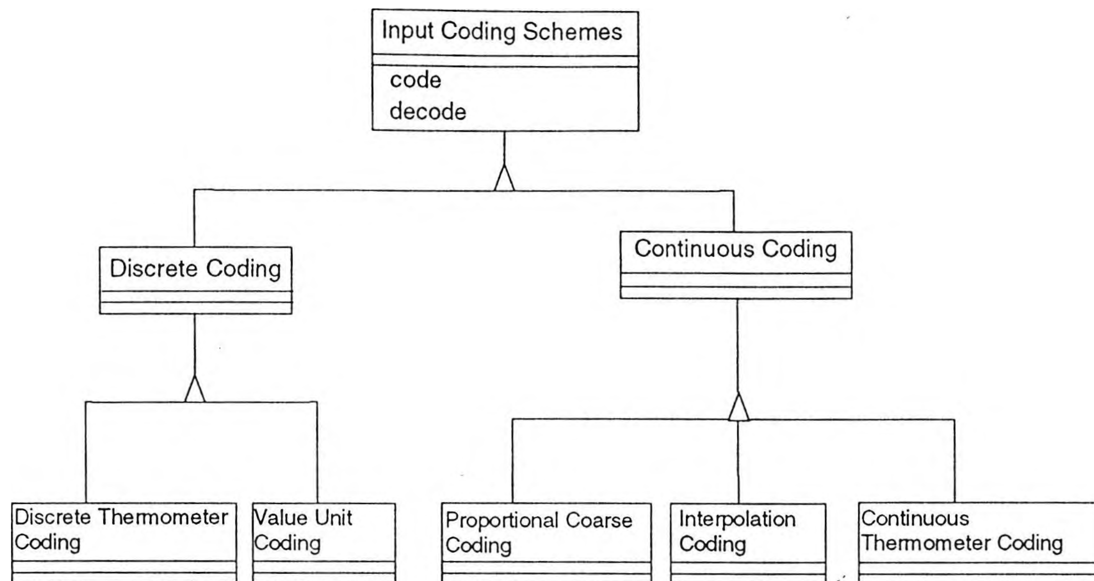


Figure 3.2-1: Hierarchy of Input Coding Schemes

3.3 What Neural Networks Learn

Neural networks learn representations [30]. For a particular problem, the space of possible representations is very large. Researchers into neural network learning are interested in the conditions under which a useful representation for a problem would be learned. Neural network learning is based on a broad class of parametric search techniques that may be recursive, non-linear, biased and even inefficient. Still, this surpasses statistical learning which exists, in principle, only when representative samples and detailed knowledge about the environment are available [31]. Features of a neural network, including the type of neurons, the architecture and the learning rules, are all independent of the problem to be solved. This has meant that when a neural network is being designed to solve a given problem, the features of the network are chosen in an ad hoc manner. A better understanding of what a neural network actually learns will provide a basis for selecting neural networks for solving specific problems. An integrated theory of learning and representation in neural networks is thus needed.

Underlying human cognition is a set of condition-action rules called production rules. Rule-based approaches provide an impressive array of tools for knowledge representation. Unfortunately, they have not been very successfully applied to the study of learning. Rule-based approaches are adequate for problems involving a small number of rules that need to be fine-tuned. In more realistic tasks, they face the characteristic problems that tend to arise in AI research. The recurrent difficulties are related to the size, nature and interdependencies of the rules whenever either the task or the rules change. Learning

systems should be able to acquire and update their rules automatically on the basis of existing rules, while learning to solve new problems. New rules have to be introduced into the learning system. The interactions of the rules have to be tested. In rule-based systems, the introduction of new rules usually creates a bottleneck.

In neural networks, the way knowledge is represented is integrated with the way knowledge can be modified. A characteristic learning task for a neural network is that of categorisation. Given a set of stimuli defined in an arbitrary feature space, a neural network can be used to sort the stimuli into pre-defined categories. Each stimulus is encoded as a unique pattern of pre-defined features. The activation of an input neuron in turn produces patterns of activity throughout the neural network. Learning consists of changes in certain properties of the hidden and output neurons. In the neural network, knowledge is stored as a pattern of connections or connection strengths amongst the neurons. The information learned directly determines how the neurons interact. Neurons have very little information stored internally. Typically, only a scalar activation-level is stored. The activation level is used as a sort of short term memory. The long term storage of information is accomplished by altering the pattern of interconnections or modifying the weights associated with each neuron. There are three kinds of constraints on neural network learning:

1. The training data set is usually incomplete and erroneous. This means that the neural network must constantly update parameter estimates with data which may represent only a small sample from a possible population.

2. The conditional distribution of categories with respect to the input stimuli and features are *a priori* unknown and have to be determined from sample that is unrepresentative.

3. Local information maybe varying or misleading. This would lead to a poor trade off between using data available at the time or waiting for more information to become available.

Single layer perceptron networks have an inherent capability to learn any function that can be represented but they are extremely limited in terms of what they can represent [8]. Multilayer neural networks use hidden neurons to increase their representation capability and hence their computational power. The presence of hidden neurons allows the network to perform more complicated input-to-output mappings, by constructing more complicated decision boundaries in state space. It is thus possible for a neural network in principle to be able to perform any classical computation [32].

3.4 Learning algorithms in Neural Networks

Neural Networks are capable of constructing models of arbitrary systems, represented by time-varying stochastic processes over some vector space. This is done by learning a mapping between the input vector space and the output vector space. There are five common learning paradigms in neural networks [1, 33].

1. **Auto Associator:** Under this paradigm, patterns are stored by repeatedly presenting them to the neural network during learning. The network learns to represent them internally. During recall, an arbitrary pattern is presented to the neural network and the network is supposed to recover the stored pattern which is closest to the one presented. The input and output patterns span the same vector space.

2. **Pattern Associator:** Under this paradigm, pairs of patterns are presented and stored in the network during learning. One of the patterns in a pair represents the key and the other represents an associated pattern. During recall, presentation of a complete or partially complete(corrupted) key should enable the network, in principle to reproduce the associated pattern.

3. **Pattern Classifier:** In this case, patterns are presented to the network which is supposed to categorise them according to some pre-defined set of classes. Usually, the learning process is supervised by a teacher signal. During training several patterns are presented along with their correct classification. During recall, the network should, in theory, be able to correctly classify patterns that are different than, but similar enough to the exemplars in each category used in the training phase. Algorithms that implement this kind or learning are referred to as supervised learning algorithms.

4. **Regularity Detector:** Given an arbitrary probability distribution of patterns, each pattern is presented to the neural network with the probabilities in which they occur in that distribution. The neural network is supposed to discover salient features in the distribution, and thus classify each pattern according to these features. In this case, there is no teacher signal provided. The different categories and what they represent has to be learned. Algorithms that implement this kind of learning are referred to as self-learning or unsupervised learning algorithms.

5. **Reinforcement Learning:** Under this paradigm, the network accepts input signals which are processed and transformed into output signals. The only clue to the correctness of this transformation is an extra reinforcement signal. The reinforcement signal acts to reward or penalise the neural network or learning system, depending on

the success or failure of its current computation. The network adapts to try and minimise the penalties, and/or maximise the rewards. Eventually, only the correct transformations are produced.

3.4.1 Associative Learning Networks

Associative neural networks are also called associative memory or content addressable memory [34]. They are used for storing and retrieving associations. An association is an ordered pair of patterns $(\mathbf{X}, \mathbf{Y})$ where $\mathbf{X} = [x_1, \dots x_m]^T$ is the key pattern and $\mathbf{Y} = [y_1, \dots y_n]^T$ is the associated pattern. Associations are stored in the connection weights of the neurons. The representation of each association will be distributed over many connections, and every connection will be involved in storing each association. This distributed representation provides the network with its robustness and graceful degradation properties as well as allowing it to discover certain regularities in the training set. Recall takes place after all associations have been learned or stored. Different key patterns are presented to the network which in turn produces the corresponding associated pattern. If the key pattern is corrupted by noise, the net work produces the stored associated pattern which best matches the presented key. There are three major types of associative memory networks:

- The linear/non-linear associator,
- The bi-directional associative memory and
- The Hopfield memory

3.4.1.1 The Linear and Non-Linear Associative Memories

Linear associative memory networks are suitable for storing lists of associations. In a network having *n* neurons with *m* inputs, every neuron receives the same input vector, **x**, and computes its output y_i according to

$$y_i = \sum_{j=1}^{m} w_{i,j} x_j, \qquad i = 1, \dots, n. \tag{3.4-1}$$

where w_{ij} is the connection weight from input j to neuron i. A simple form of the Hebbian learning rule is employed. The weights are computed as follows:

$$w_{i,j} = \sum_{k=1}^{p} s_i^k r_j^k \qquad i = 1, \dots, n; j = 1, \dots, m. \tag{3.4-2}$$

Linear associative memories will exhibit perfect recall if the input vectors are orthonormal [35]. Given L pairs of vectors $\{(\mathbf{x}_1,\mathbf{y}_1),(\mathbf{x}_2,\mathbf{y}_2),\ldots,(\mathbf{x}_L,\mathbf{y}_L)\}$, with $\mathbf{x}_i \in \Re^n$, and $\mathbf{y}_i \in \Re^m$, [21] distinguish three types of associative memory:

1. **Hetero-associative memory:** This type of associative memory implements a mapping, Φ, of $\mathbf{x}$ to $\mathbf{y}$ such that $\Phi(\mathbf{x}_i) = \mathbf{y}_i$. If an arbitrary $\mathbf{x}$ is closest (in terms of hamming distance) to $\mathbf{x}_i$ then $\Phi(\mathbf{x}) = \mathbf{y}_i$.

2. **Interpolative associative memory:** This type of associative memory also implements a mapping, Φ, of $\mathbf{x}$ to $\mathbf{y}$. If the input vector differs from one of the stored exemplars by the vector $\mathbf{d}$, such that $\mathbf{x} = \mathbf{x}_i + \mathbf{d}$, then the output of the memory also differs from one of the exemplars by some vector $\mathbf{e}$ such that $\Phi(\mathbf{x}) = \Phi(\mathbf{x}_i + \mathbf{d}) = \mathbf{y}_i + \mathbf{e}$.

3. **Auto-associative memory:** Autoassociative memories assume that $\mathbf{x}_i = \mathbf{y}_i$ and implements a mapping Φ, of $\mathbf{x}$ to $\mathbf{x}$, such that $\Phi(\mathbf{x}_i) = \mathbf{x}_i$. If some arbitrary $\mathbf{x}$ is close to $\mathbf{x}_i$, then $\Phi(\mathbf{x}) = \mathbf{x}_i$.

Non-linear associative memories are also used for storing lists of associations. They employ a non-linear processing element which helps to eliminate unwanted perturbation [23]. Assuming L pairs of vectors $\{(\mathbf{x}_1,\mathbf{y}_1),(\mathbf{x}_2,\mathbf{y}_2),\ldots,(\mathbf{x}_L,\mathbf{y}_L)\}$, with $\mathbf{x}_i \in \Re^n$, and $\mathbf{y}_i \in \Re^m$, have been stored in the network, if a test pattern $\mathbf{t} \in \Re^n$ is presented to the network, a matching score vector $\mathbf{s} \in \Re^L$ is computed as an inner-product product of $\mathbf{t}$ and all the $\mathbf{x}_i$.

$$\mathbf{s} = [<\mathbf{x}_1,\mathbf{t}>,<\mathbf{x}_2,\mathbf{t}>,\ldots,<\mathbf{x}_L,\mathbf{t}>]$$

where

$$<\mathbf{x}_r,\mathbf{t}> \equiv \mathbf{x}_r^T\mathbf{t} \equiv \sum_{i=1}^{L} x_{ri}\, t_i \qquad (3.4\text{-}3)$$

The non-linear processing element acts on the score vector to produce a binary decision vector v.

$$\mathbf{v} = N(\mathbf{s}) \qquad (3.4\text{-}4)$$

The pattern to be retrieved is the vector

$$\mathbf{y} = \mathbf{A}\mathbf{v} \qquad (3.4\text{-}5)$$

where $\mathbf{A}$ is a matrix formed from the column vectors $\mathbf{y}_i$. The non-linear element selects a winning node and simultaneously suppresses all others. This suppresses any noise on the test signal and thus leads to holographic retrieval.

3.4.1.2 The Bi-directional Associative Memory (BAM)

The bi-directional associative memory (BAM) consists of two layers of processing elements that are fully interconnected between the layers. The neurons may, or may not, have feedback connections to themselves. The connection weights between the neurons can be determined in advance if all the training exemplars are available. Given L pairs of vectors $\{(\mathbf{x}_1,\mathbf{y}_1),(\mathbf{x}_2,\mathbf{y}_2),\ldots,(\mathbf{x}_L,\mathbf{y}_L)\}$, with $\mathbf{x}_i \in \Re^n$, and $\mathbf{y}_i \in \Re^m$, that is to learned, the weight matrix, W, is given by

$$\mathbf{W} = \mathbf{y}_1\mathbf{x}_1^T + \mathbf{y}_2\mathbf{x}_2^T + \ldots + \mathbf{y}_L\mathbf{x}_L^T \tag{3.4-6}$$

where w_{ij} is the weight on the connection from the jth neuron on the **x** layer to the ith neuron on the **y** layer. Once the weight matrix has been constructed, the BAM can be used for recall. A partial or noisy key vector presented as input to the net will yield both the correct input and the associated pattern vector, assuming that the network has not been overloaded with exemplars. The network is overloaded when the exemplars are too close together in terms of the hamming distance between them. Interactions between the patterns in an overloaded network will result in the creation of spurious stable states or local minima as the network tries to converge to a solution. This phenomenon is known as crosstalk and could cause the network to stabilise on meaningless vectors. An unknown pattern presented to the network during recall may require several passes before the network stabilises on a final solution. In such a situation, the **x** and **y** vectors change with time and so form a dynamical system. The energy[1] in the system is given by the BAM energy function as:

$$E(\mathbf{x},\mathbf{y}) = -\mathbf{y}^T\mathbf{W}\mathbf{x}$$

$$= -\sum_{i=1}^{m}\sum_{j=1}^{n} y_i w_{ij} x_j \tag{3.4-7}$$

The BAM energy theorem [34] ensures the convergence of the BAM energy function by assuring the existence of stable solutions for the BAM processing equations. The BAM energy theorem can be stated as:

1. Any change in **x** or **y** during BAM processing results in a decrease in E.
2. E is bounded below by $E_{min} = -\sum_{i,j} |w_{ij}|$.
3. When E changes, it must change by a finite amount.

[1] The states in the system are associated with an energy surface that can be calculated from the weight values. Stable states are seen to be on some sort of minimum in the energy surface.

3.4.1.3 The Hopfield Memory Network

The Hopfield memory is an associative neural network having a fully connected set of neurons [36]. The inputs to each neuron are fed by the outputs of all the other neurons in the network. The arrangement creates a recurrent dynamical system capable of storing and recalling abstract association lists. In the Hopfield memory network, a set of equations and an update policy defines the dynamics of the system. The net input to a neuron is given by

$$net_i = \sum_{j=1}^{n} w_{ij}\, x_j + I_i \qquad (3.4\text{-}8)$$

The output value of each neuron depends on both the net input value and the current output value as well as a threshold value assigned for each neuron. The new output value is given by:

$$x_i(t+1) = \begin{cases} +1 & \text{net}_i > T_i \\ x_i(t) & \text{net}_i = T_i \\ -1 & \text{net}_i < T_i \end{cases} \qquad (3.4\text{-}9)$$

where T_i is the threshold value.

The behaviour of the Hopfield network can be characterised by means of an energy function analysis where the energy function

$$\mathrm{E}(\mathbf{x}) = -\sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij}\, x_i x_j + 2\sum_{i=1}^{n} T_i x_i \qquad (3.4\text{-}10)$$

decreases whenever the state of any processing element changes. It has been shown that when the weights are symmetrical, i.e., $w_{ji} = w_{ij}$ the network always converges [10]. The shape of the convergence region provides useful information which has been used in methods for suppressing spurious[2] states in the Hopfield networks [37].

3.4.2 Unsupervised Learning Networks

Unsupervised or competitive learning takes place in a context of sets of hierarchically layered neurons. Unsupervised learning algorithms have the property that a competition process, involving some or all of the processing elements, always takes place before each episode of learning. For problems where correct examples of input and output do not exist or are not readily available or where it is necessary to provide insights into the nature of the data in order to determine the kind of classifier to design, an unsupervised learning

[2]Spurious states are local minima on the error surface. The network is caught between stable states and never reaches a global minimum. This usually happens when the network is overloaded. with patterns or when the hamming distance between pairs is not insufficient.

system may be used to provide the categorisation. Also, if the characteristics of the patterns can change slowly with time, an unsupervised learning system that can track these changes will provide better performance [31]. The available information is in the form of a set of input patterns $\mathbf{x}^p \in \mathbf{X}$, a p-dimensional input vector where $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$ is a data set of n items. The operations that could be performed on **X** include:

- **Clustering**: Clustering in **X** means the identification of an integer *c*, $2 \leq c < n$, and a partitioning of **X** by *c* mutually exclusive, collectively exhaustive subsets of **X** called clusters [38]. The learning system has to find inherent clusters in the input data. The output of the system is the cluster label for an available input pattern.

- **Classification:** If **S** denotes the data space from which **X** was drawn, i.e., $\mathbf{X} \subset \mathbf{S}$, classification in **S** is a process whereby **S** is partitioned into a number of decision regions. The unsupervised learning system delineates the decision regions in **S** in an attempt to discover structure in **S**.

- **Vector Quantisation**: In this sort of categorisation, continuous space has to be discretised. The input of the system is the p-dimensional vector, **x**. The unsupervised learning system has to find an optimal discretisation of the input space.

- **Dimensional Reduction:** The input vectors have to be grouped into a subspace that has lower dimensionality than that of the data. The unsupervised learning system learns an optimal mapping that preserves the variance of the input data in the output space.

- **Feature Extraction:** The unsupervised learning system is used to extract features from the input data. This usually leads to a dimensional reduction of the input space.

3.4.2.1 Competitive Learning

The basic competitive learning scheme is described in [1]. In this scheme, binary vector patterns are presented to the neural network and a competition is held amongst the neurons. The neuron with the maximum output wins the competition and its weights are adapted in the direction of the input using a simple form of the Hebbian learning rule. Weight adaptation is limited to neurons that win the competition. All weights are normalised such that $\sum w_{ij} = 1$. Learning occurs by shifting a proportion of the total weights associated with the winning neuron onto its active lines.

3.4.2.2 Clustering Networks

Figure 3.4-1 shows a simple clustering network. The network is fully connected, with weights w_{ij} connecting neuron i in the output layer to neuron j in the input layer. Both the input and weight vectors are normalised to unit length initially.

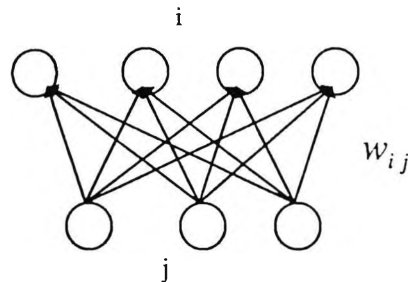


Figure 3.4-1: A simple clustering neural network

The activation of neuron i in the output layer is given by

$$a_i = \sum_j w_{ij}\, x_j = \mathbf{w}_i^T \mathbf{x}. \qquad (2.\ 1)$$

The competition process selects the neuron with the maximum activation as the winner and its activation is given a maximum value of 1. All other neurons have their activation reset to 0. The learning process adapts the weight vector of the winning neuron so that it is closer to the input vector each time an input vector is presented. Consequently, the weight vectors move towards an area which has a higher density of input vectors, thus forming a cluster. The simple clustering network has inherent stability problems as the cluster centres continue to move around indefinitely. Furthermore, a malicious input vector can completely change the centres of a the clusters causing misclassification of previous training vectors. The ART networks [39] are an advanced implementation of the clustering network with built-in vigilance and gain to control prevent cluster centres from getting infinitesimally close and to provide better support for incremental learning [21].

3.4.2.3 Vector Quantisation

Vector quantisation algorithms are used to find natural groupings in a data set. Every feature vector is associated with a point in n-dimensional feature space. Vectors $\vec{\mathbf{x}}$ belonging to the same class are assumed to form a cluster in feature space. Vector quantisation discretises the input space so that the clusters can be separated. A vector quantisation neural network approximates a mapping of an n-dimensional input space to an m-dimensional output space. The vector quantisation algorithm presupposes that

vectors belonging to the same class are distributed normally with mean $\vec{\mu}_i$. Feature vectors are classified on the basis of their Euclidean distance, $\|\vec{\mathbf{x}} - \vec{\mu}\|$, from a pre-selected set of mean vectors. A feature vector $\vec{\mathbf{x}}$ is assigned to the class of the nearest mean. Whenever a feature vector has been wrongly classified, the mean vectors are updated by moving the correct mean towards the feature vector and the wrong mean away from it. The learning rule used to update the mean vectors is given by

$$w_{ij}(t+1) = w_{ij}(t) + \eta(o_i - w_{ij}(t)) \quad (3.4\text{-}11)$$

where η is the learning rate or step size, taken in the direction of the input vector and

o_i is the output of the network.

Vector quantisation has been combined with feedforward networks to form the counter propagation networks [5].

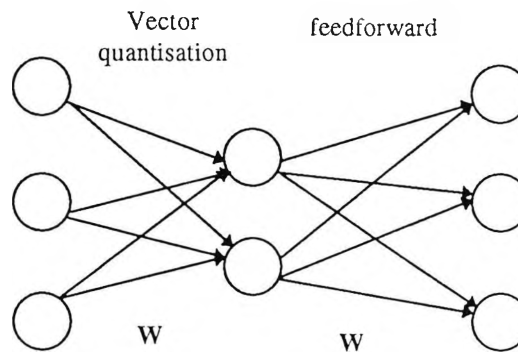


Figure 3.4-2: Hybrid Network: Vector Quantisation and Feedforward Network

3.4.2.4 *SOM Learning*

Self-Organising feature maps or Self-Organising Memories are a unique class of neural networks because they can construct topology-preserving maps of the input data set, where the location of the neuron carries semantic information [3]. A SOM network is made up of two layers of neurons as shown in Figure 3.4-3. The input layer is a one dimensional vector, while the output layer is a two dimensional array or grid of neurons. They can be used to cluster data, obtaining a 2-dimensional display of the input space that is easy to visualise.

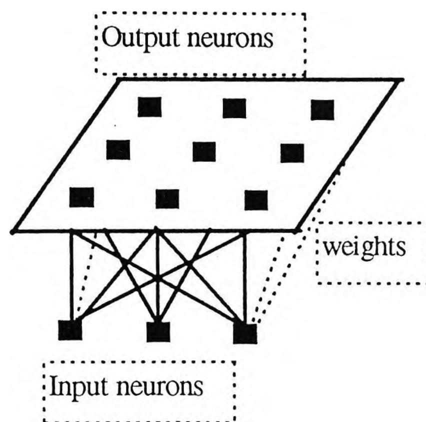


Figure 3.4-3: Schematic diagram of a SOM network

In a SOM network, learning is accomplished by the application of input data alone; no expected output data is used as a teacher to signal the network that it has made an error. A competitive algorithm is used to produce a topology preserving map of the input data using the competing neurons. In a topology preserving map, neurons located physically next to each other will respond to classes of input vector that are close to each other. Larger dimensional input vectors are projected down on the two dimensional map in a way that maintains the natural order of the input vectors. This dimensional reduction allows easy visualisation of the relationships amongst the data. The SOM learning algorithm produces a spatial ordering of the output neurons that are close to each other in output grid. Both the weights and the input data have to be normalised before the network can be trained. During learning, an input vector is presented in parallel to all units in the output layer. The activation of each output neuron is given by

$$a_i(t) = \sum_{i=1}^{n} x_i(t) w_{ij}(t) = \mathbf{x} \cdot \mathbf{w}_j \qquad (3.4\text{-}12)$$

The neuron with the largest activation function is chosen as the winner. The topological ordering is achieved by using a spatial neighbourhood relation (N_c) between the competitive units during learning. Adaptation takes place on the weights of both the winning neuron and the surrounding neurons during the training process. The most commonly used neighbourhood function is the Mexican-hat function shown in Figure 3.4-4. This function has a peak at the centre and tapers off towards the edges. This allows maximum learning for the winning neuron while enabling the closest neighbours to participate in the learning process.

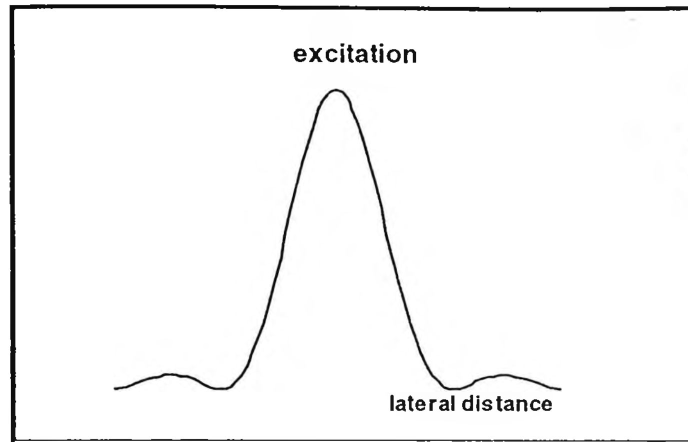


Figure 3.4-4: Mexican-hat function used as Neighbourhood function in SOM learning

All neurons within the neighbourhood of the winning neuron participate in the weight update process. The size of the neighbourhood is diminished as learning progresses until it encompasses only a single neuron. For a winning neuron c, with neighbourhood N_c, the weight update equations are given by

$$\mathbf{w}_i(t+1) = \begin{cases} \mathbf{w}_i(t) + \alpha(t)(\mathbf{x} - \mathbf{w}_i(t)) & i \in N_c \\ 0 & \text{otherwise} \end{cases} \quad (3.4\text{-}13)$$

Weight vectors participating in the update process rotate towards the input vector **x**. After training, the weight vectors converge to a value which is representative of points close to the physical location of the winning neurons. Thus, an advantage of the SOM is that large numbers of unlabelled data can be organised quickly into a configuration that illuminates underlying structure within the data [40]. As a result of SOM learning, the point density function of the weight vectors tend to approximate the probability density function p(**x**) of the input vectors, **x**, and the weight vectors end up being ordered according to their mutual similarity.

3.4.3 Stochastic Learning

Statistical mechanics is the central discipline of condensed matter physics, a body of methods for analysing aggregate properties of the large numbers of atoms found in liquid or solid matter. Because the number of atoms is extremely large, only the most probable behaviour of the system in thermal equilibrium at a given temperature is observed in experiments [41]. This can be characterised by the average and small fluctuations about the average behaviour of the system over a given ensemble of identical systems. In this ensemble, each configuration of the system is defined by a set of atomic positions $\{r_i\}$ weighted by its Boltzmann probability factor, given by

$$\exp\left(-E(\{r_i\})\Big/k_b T\right) \quad (3.4\text{-}14)$$

where $E(\{r_i\})$ is the energy of the configuration,

k_b is the Boltzmann constant and

T is the temperature of the configuration.

At high temperatures, there are very few ground states (states of low energy) in the system. As T is lowered, the Boltzmann distribution collapses into its lowest energy state or states. Stochastic learning networks make use of statistical mechanics techniques such as stochastic simulated annealing (SSA) and mean field annealing (MFA) to solve combinatorial optimisation problems [22]. A combinatorial optimisation problem for a discrete system involves searching for a state that minimises a predetermined energy criterion. Each state of the network is associated with a computed energy level that depends on the temperature parameter *T*. The network has a non-zero probability to go from one state to another. The probability function depends on the temperature and the energy difference between two states. Learning takes place by state update according to the Boltzmann state-transition rule. For a stochastic network, if $P(a)$ denotes the state-distribution function and $\Pr ob(a \to a')$ denotes the state-transition function from one state a to another state a', the Boltzmann state-transition rule is given by

$$\Pr ob(a \to a') = \frac{1}{1+\exp(\Delta E/T)} \equiv f(-\Delta E/T) \quad (3.4\text{-}15)$$

The Boltzmann state-transition rule ensures that, in thermal equilibrium, the relative probability of two global states is determined solely by their energy difference and temperature, and the probability of being in a state follows a Boltzmann distribution. As can be seen in Figure 3.4-5, the global minimum has lower energy than any local minima and so is able to attract higher-energy local optima.

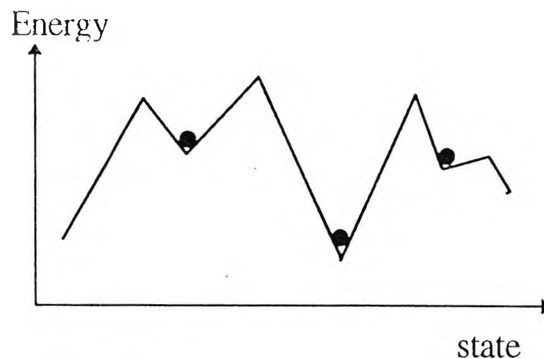


Figure 3.4-5: Energy surface of a hypothetical optimisation problem

3.4.4 Supervised Learning

Supervised learning networks adapt their connection weights to learn the relationship between a set of example patterns. They are able to apply the relationship learnt when presented with novel input patterns. This is because they focus on features of arbitrary input patterns that resemble those of the examples used during training. The output of the network is a function of its inputs and the connection weights between its neurons, i.e.

$$y = \Phi(\mathbf{x}, \mathbf{w}) \qquad (3.4\text{-}16)$$

where Φ is termed the discriminant function. During training, example patterns are presented as pairs of input/teacher vectors, $[\mathbf{X},\mathbf{T}] = \{[\mathbf{x}_1,\mathbf{t}_1],[\mathbf{x}_2,\mathbf{t}_2],\ldots[\mathbf{x}_M,\mathbf{t}_M]\}$, M is the number of training pairs. The network learns a mapping between the input vectors $\mathbf{x}$ and the teacher vectors $\mathbf{t}$. There are two approaches to supervised learning which differ only in the nature of the teaching signal. The first approach is based on the correctness of the solution and is termed decision based learning. The second approach is based on the optimisation of a training or cost function and is said to be approximation based.

3.4.4.1 Decision-based supervised learning

In this form of supervised learning, the teaching signal only serves to determine whether each training pattern is correctly classified or not hence binary decision vectors are used as teaching signals. The objective of training is to find a set of weights that yield a correct classification of the input patterns. For a sample classification problem, the pattern space is divided into decision regions separated by decision boundaries. The discriminant function acts as a hyperplane or hypersurface separating the decision regions. If the pattern classes are linearly separable, then, a linear discriminant function can be used. Two classes of patterns are said to be linearly separable if the can be separated by a linear hyperplane decision boundary. The decision boundary is characterised by a linear discriminant function

$$\Phi(\mathbf{x}, \mathbf{w}) = \sum_{i}^{P} w_i x_i + \theta = 0 \qquad (3.4\text{-}17)$$

The classification is decided based on the values of the discriminant functions at the network's output. A binary decision is made as to the correctness of the classification.

$$d = \begin{cases} 1 & y>0 \\ 0 & y\leq 0 \end{cases} \qquad (3.4\text{-}18)$$

The network weights are updated according to the perceptron learning rule [9].

$$w^{(m+1)} = w^{(m)} + \eta\,(t^{(m)} - d^{(m)})x^{(m)} \qquad (3.4\text{-}19)$$

where η is the learning rate. Because both $t^{(m)}$ and $d^{(m)}$ are binary, the network weights will only be updated when a pattern is misclassified in a particular epoch. If a pattern belongs to a class but is misclassified by the network, then the network weights will be reinforced by adding a fraction of the input pattern to the weights. On the other hand if the pattern does not belong to the class but is misclassified as belonging to the class, then the weights will be anti-reinforced by subtracting a fraction of the pattern from the weights. Training stops when all patterns are correctly classified and no more weight updates take place. The learning process is said to converge. It has been proven [8] that if the classes of patterns are linearly separable, then the learning process is guaranteed to converged to a correct solution in a finite number of steps. This is known as the Perceptron Convergence Theorem [8, 9] and only holds when the discriminant functions are linear. Perceptron networks composed of only linear discriminant functions are severely limited because they require that the input patterns should be linearly separable if the learning algorithm is to converge. The perceptron learning rule has been generalised in [22] into the decision-based learning rule which as applicable to both linear and non-linear perceptrons. Weight update is still by reinforced and anti-reinforced learning. The decision boundaries are adjusted by adapting the network weights in the direction of the positive gradient of the discriminant function during reinforced learning or in the opposite direction during anti-reinforced learning.

$$\Delta\mathbf{w} = \pm\eta\nabla\Phi(\mathbf{x},\mathbf{w}) \qquad (3.4\text{-}20)$$

where η is the learning rate,

$\nabla\Phi(\mathbf{x},\mathbf{w}) = \dfrac{\partial\,\Phi(\mathbf{x},\mathbf{w})}{\partial\,\mathbf{w}}$ is the gradient vector of function Φ with respect to $\mathbf{w}$.

The teaching signal is still a binary decision vector that determines the class of a particular input pattern. When a pattern is correctly classified, no weight update takes place. When the pattern is misclassified, the network weights are updated according to the following equations:

$$\begin{aligned} &\text{reinforced learning:} && \mathbf{w}(t+1) = \mathbf{w}(\mathrm{t}) + \Delta\mathbf{w} \\ &\text{anti-reinforced learning:} && \mathbf{w}(t+1) = \mathbf{w}(t) - \Delta\mathbf{w} \end{aligned} \qquad (3.4\text{-}21)$$

When the network discriminant functions are linear, the perceptron learning rule is obtained.

3.4.4.2 Approximation-based supervised learning

In approximation-based supervised learning the training patterns are also given in input/teaching signal pairs. The teaching signals are the desired or target values at the output nodes which correspond to a given input pattern. The objective of the learning process is to find an optimal set of weights to minimise the error between the teaching signal and the actual response of the network. The learning problem can usually be reposed as a function minimisation problem. For a multilayer perceptron network with inputs vectors $\mathrm{x_p} \in \Re^{N}$ and target vectors $t^{p} \in \Re^{K}$ the error due to the pth pattern vector is given by

$$E_p = \sum_k (t_{pk} - o_{pk})^2 \qquad (3.4\text{-}22)$$

where $o_{pk} = f(net_j) = f(\sum_i \overline{w}_{ji} a_i + \theta_j)$ is the actual output of the kth neuron,

t_{pk} is the neuron's desired or target value,

$\mathrm{f(net_j)}$ is the neuron activation function,

$\mathrm{net_j}$ is the net input to the jth neuron,

a_i is the activation of ith neuron and

θ_j is a learnable bias weight.

E_p can be expressed in vector notation as

$$\begin{aligned} \mathrm{E_p} &= (\mathbf{t}-\mathbf{o})^{T}(\mathbf{t}-\mathbf{o}) \\ &= (\mathbf{t}-\mathbf{o})^{T}\mathbf{I}(\mathbf{t}-\mathbf{o}) \end{aligned} \qquad (3.4\text{-}23)$$

where $\mathbf{I}$ is the identity matrix. E_p is a quadratic error function and solutions of minimise $\mathrm{E_p} = \mathrm{E}(\mathbf{w})$ provide an optimal set of weights for a given problem [42]. Gradient descent methods or methods of steepest descent can be used to solve the learning problem. These methods transform the minimisation problem into an associated system of first-order ordinary differential or difference equations.

$$\frac{dw_j}{dt} = -\sum_{k=1}^{K} \mathbf{u}_{jk} \frac{\partial E_p}{\partial w_j} \qquad (3.4\text{-}24)$$

with initial conditions $w_j(0) = w_j^{(0)}$. Equation 2.2 can be written in compact vector form as follows

$$\frac{\mathrm{d}\mathbf{w}}{\mathrm{d}t} = -\mathbf{u}(\mathbf{w},t)\nabla_{\mathbf{w}} E_p(\mathbf{w}) \qquad (3.4\text{-}25)$$

where $\mathbf{u}(\mathbf{w},t)$ is a symmetric positive definite matrix called the learning matrix. In the steepest descent method, $\mathbf{u}(\mathbf{w},t)$ is assumed to be a unitary matrix multiplied by a positive constant η which is the learning rate. The above system of differential equations can be shown to be stable by considering the change of the error (Energy or Lypunov) function E_p with time:

$$\frac{\mathrm{d}E_p}{\mathrm{d}t} = \sum_j^N \frac{\partial E_p}{\partial w_j} \frac{\mathrm{d}w_j}{\mathrm{d}t} \tag{2. 3}$$

Substituting (2. 4) in (2. 5) and enforcing the condition that $\mathbf{u}(\mathbf{w},t)$ is symmetric and positive definite, the following condition holds

$$\frac{\mathrm{d}E_p}{\mathrm{d}t} = -[\nabla_w E_p(\mathbf{w})]^T \mathbf{u}(\mathbf{w},t) \nabla_w E_p(\mathbf{w}) \leq 0 \tag{3.4-26}$$

The above condition guarantees that the error function decreases in time and eventually converges to a stable local minima. The speed of convergence to the minimum depends on the choice of values for the learning matrix $\mathbf{u}(\mathbf{w},t)$. Appendix B discusses convergence theory of algorithms and proof that for a quadratic error measure, the steepest descent method is guaranteed to converge.

The discrete-time version of the gradient descent method is

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta^{(k)} \nabla_{\mathbf{w}} E_p(\mathbf{w}^{(k)}) \tag{3.4-27}$$

where $\eta^{(k)}$ is the learning step size which should be bounded in a small range to ensure the stability of the algorithm. In most implementations of the steepest descent learning algorithm $\eta^{(k)}$ is determined at each time step by one-dimensional line search as the value of $\eta \geq 0$ that minimises

$$\psi(t) = E_p(\mathbf{w}^{(k)} - \eta \nabla_w E_p(\mathbf{w}^{(k)})) \tag{3.4-28}$$

3.4.4.3 The backpropagation learning algorithm for multilayer perceptron networks

Multilayer perceptron networks consists of layers of neurons connected by synaptic weights. An example of a fully connected network is shown in Figure 3.4-6. The sum-of-squares error is normally used as the error or energy function at the output of the network and is given by

$$E = \sum_{p=1}^{P} E_p = \sum_{p=1}^{P} \sum_{k=1}^{K} (t_{pk} - o_{pk})^2 \tag{3.4-29}$$

where P is the number of patterns in the training set.

The dynamic gradient descent algorithm described above can be used to minimise the sum-of-squares error criterion. This method requires the computation of the gradient $\nabla_{\mathbf{w}} E(\mathbf{w})$ for all the weights in the network during each learning iteration. This is very inefficient and computationally very expensive. It is also impractical for large networks. The backpropagation algorithm first proposed in [7] and later popularised by [43, 44] offers an effective approach to the computation of gradients and hence a relatively efficient speedup to the training of multilayer neural networks. The backpropagation algorithm makes use of a recursive formulation to compute the error at the output of lower layer neurons when the error at the output neurons is known. Hence the computation of gradients of the energy function with respect to the lower layer weights is thus avoided. The network weights are then updated according to the Generalised Delta Learning Rule [43]. Appendix C shows the derivation of the backpropagation algorithm for training multilayer neural networks.

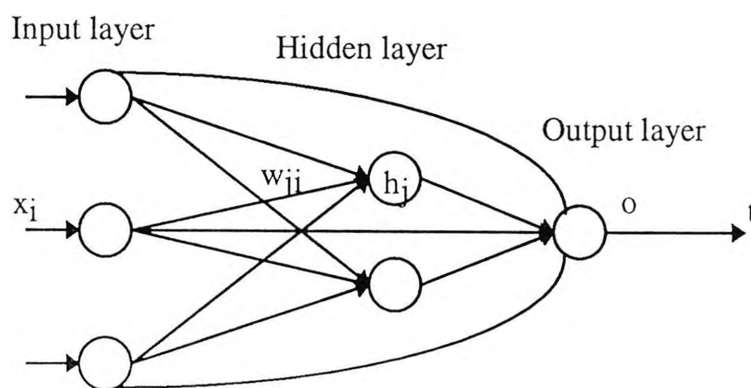


Figure 3.4-6: A 2-layer fully connected multilayer perceptron network. There are 3 neuron layers and 2 weight layers. The number of layers in the network usually refers to the number of weight layers

3.4.4.4 Supervised Learning Parameters

Certain formulations have to be adopted when supervised learning networks are applied to solve real problems. The selection of the learning parameters have a significant effect on the network performance. The learning parameters are varied and very problem dependent. Some of the factors that influence learning in supervised networks include:

- **Learning rate η**: This specifies the step size that is taken in the downhill slope or gradient in weight space, when weights are updated. The learning procedure requires that the weight change be proportional to $\nabla_{\mathbf{w}} \mathrm{E}(\mathbf{w})$. True gradient descent requires

that infinitesimal steps are taken. The learning rate acts as the constant of proportionality. In practical applications, the largest learning rate that does not lead to oscillations during training should be chosen. Alternatively, a one-dimensional line search, learning rate adaptation [45], relaxation methods [22] or optimal learning methods [46] can be used to determine the optimal learning rate at each time step.

- **Momentum term:** To prevent oscillations in the learning process, the weight update equations are usually modified to include a momentum term. The momentum term specifies the fraction of the previous weight change that is added to the weights during update. The momentum term helps keep the weight changes going in the same direction and prevents oscillations when reasonably high learning rates are used. This enables the network to converge to a final solution much faster.

- **Range of initial weights:** Gradient descent methods require the provision of initial conditions to solve the system of dynamic equations that represent the neural network. The conditions are provided by initialising the network weights to small random values. This represents the starting point of the iterative descent algorithm. Large values for the initial weights is not recommended because the neural network could be initialised to a state that is either unstable or far removed from the final solution state. This will lead to very long training times and possible difficulties in learning the required mapping. Techniques have been proposed to statistically control weight initialisation based on the training patterns [47].

- **Frequency of weight update:** An Iteration involves a single training pattern presented to the system. An epoch or sweep covers the presentation of an entire block of training data to the system. The network weights can be updated after each training pattern is presented i.e., after each iteration. This is known as data-adaptive updating and provides very fast learning and better response for on-line or real-time learning applications. On the other hand, it is numerically unstable for large problems and extremely sensitive to network learning parameters and noise effect on individual patterns. Block-adaptive methods, on the other hand, only update the network weights after one epoch i.e., the presentation of a block of patterns or all the training patterns to the network. The network tends to be slower learning but learning is more predictable and robust as the training step is averaged over all the patterns.

- **Scaling and normalisation of input patterns:** Normalisation or scaling is necessary for gradient descent algorithms that have no self-normalising qualities [48].

Normalisation or scaling of the input patterns can significantly speed-up numerical convergence of the gradient descent algorithm [42]. The simplest form of normalisation is to convert all input vectors to unit length. This tends to destroy the variance of the different features that make up the input vector. The input vectors can also be scaled so that the variance between the features in each input vector are preserved [49].

- **Network architecture:** The generalisation capability of a multilayer network depends on architectural parameters. This includes the number of weight layers, the number of hidden neurons, the type of the discriminant functions and the nature of the activation functions. The ability of the network to generalise on the training set is very sensitive to number of hidden neurons. A rule of thumb for obtaining good generalisation in backpropagation networks is to use the smallest network that will fit the data. If the number of hidden neurons is too large, the network will take too long to train and will memorise instead of generalise on the training data. The test performance on noisy input or patterns not seen before will thus be very poor. On the other hand, if the number of hidden neurons is too small, the network will not be able to learn or will be very sensitive to the network initial conditions. Small networks are also more likely to become trapped in a local minimum. A number of techniques have been proposed for determining an optimal number of hidden units to learn a particular task. These include algebraic projection analysis [50], network growing [51, 52] and pruning [53] and network Evolution [54]. The nature of the discriminant functions coupled with the kind of activation function mostly affects quality of the classification performed by the neural network. Conventional multilayer networks use linear discriminant or linear basis functions with linear or non-linear activation functions. As mentioned above, such networks produce hyperplane decision boundaries to separate classes of input patterns. More recently, multilayer perceptron networks using radial, elliptic, spline or wavelet basis functions with Gaussian activations have been proposed [26, 55, 56, 57, 58, 59, 60]. Radial basis function networks have a great advantage in that they are much faster to train than multilayer perceptrons using back-propagation and are thought to be more suitable for approximating and learning smooth continuos functions from sparse data. The main reason is due to the fact that while multilayer perceptrons construct hyperplane decision boundaries in order to separate class categories, the corresponding decision boundaries are hyperspherical or hyperellipsoidal when radial basis networks are used. The resulting classification is much more accurate and can be constructed much faster because of typical clusters found in data.

3.4.4.5 Speeding up Supervised Learning

The greatest single obstacle to the application of multilayer networks trained by backpropagation in real-world problems is the slow speed at which current algorithms learn. Even on relatively simple problems, standard backpropagation often requires a lengthy training process in which the complete set of training exemplars is presented a large number of times. One solution is to run the network on even faster computers, but the sequential nature of conventional digital computers is a limiting factor to the speed at which the networks can be run. Hardware implementation using VLSI and optics [61, 62] with programmable interconnections would significantly increase the computations speed. Faster learning variations of the backpropagation algorithm such as quickprop [63], rprop and backpercolation [64] have been proposed to overcome some of the speed limitations. Second order methods such as conjugate gradients [63, 46] and generalised projections [65] can search the for the minimum faster than conventional gradient descent used by back-propagation. Unfortunately, the rate of convergence for conjugate gradients is only linear in n, where n is the number of variables and so the algorithm has to be constantly restarted [66] for fast learning to occur. For static pattern recognition problems, faster learning with improved accuracy can be obtained using basis function networks [56].

3.5 Discussion and Conclusions

Our understanding of learning in neural networks is far from complete. Different theories about what neural networks really learn have been proposed but these remain theories. Hard evidence that corroborates these theories are very difficult to obtain. Most of the questions posed about learning and convergence in neural networks in [8] remain unanswered despite over 20 years of research and development. This chapter has surveyed the large body of literature that is concerned with learning in artificial neural networks. The general conclusion is that neural networks learn representations. The representations can either be external or internal, explicit or implicit, local or distributed. External representations created by designers and imposed on a neural network have very little chance of being learnt except where there is a high correlation between the external representation and one of a possible choice of internal representations that the neural network would have arrived at for the initial learning problem. Another possible conclusion from this survey is that the neural network is more likely to create distributed rather than local representations for a given learning problem. This implies that neural network designers should be wary about creating local representations for problems where a neural network solution is required. It is obvious that local representations are not

consistent with the graceful degradation with which neural information processing has come to be associated. The third conclusion concerns neural network parameters. There is a high correlation between parameters of the neural network such as size, activation function, topology, etc. and the range of representations that it can learn. Highly complex networks with larger degrees of freedom and a bigger range of free parameters can obviously learn highly complex representations. The final conclusion drawn from the survey is that for any given problem, if a suitable representation can be found, then it is possible to configure a neural network to learn the representation and hence solve the problem. This implies that a neural network solution is possible for any problem where a suitable representation can be found. The implications to neural network designers are potentially very severe as difficult learning problems can be construed as poor solution design rather than any inability on the part of the neural network to solve the problem.

The theoretical and fundamental aspects of learning, which falls in the realm of psychology and complexity theory, have not been explored in this chapter. Instead, emphasis has been placed on learning and learning algorithms in artificial neural networks. A survey of learning algorithms has been carried out. The stability and learning issues have been explored and the large body of literature that covers learning in neural networks thoroughly examined with some potentially far reaching conclusions of practical significance to neural network designers.

ANALYSIS AND DESIGN OF NEURAL NETWORKS

4.1 Introduction

Neural Networks are often regarded as black box systems. The current generation of neural network systems exist mostly as legacy[1] software or code libraries that have been programmed to solve specific sets of problems [67, 68]. Despite the prevalence of general purpose neural network software both in the public domain and commercially, there has been little or no attempt to create a robust software architecture for neural network systems which can be integrated into general commercial and industrial systems development. Some notable efforts include [68] where dataflow diagrams were presented for neural network operations. In [69] class diagrams based on the Booch notation [17] have been used to describe the design of a general neural network, while in [67] an object-oriented framework for neural network systems is presented. This chapter describes how a robust object-oriented architecture for neural networks can be constructed. Such an architecture will allow neural networks and other intelligent systems to be widely used as components in general purpose commercial and industrial systems development efforts, whether they are in power systems monitoring and protection, power systems control, industrial plant control systems, financial software systems or data mining and visualisation systems in distributed databases including the World Wide Web. With a clearly defined and open architecture, it will be possible for system designers to directly incorporate neural network components in their designs or call on the services of neural networks and other intelligent systems as part of a distributed object system using an object middleware[2] mechanism such as Microsoft's Object Linking and Embedding (OLE) or an object request broker that subscribes to

---

[1] Legacy systems are software systems currently in use but in need of replacing or extensive maintenance. Because there is no well defined architecture, maintaining or extending such systems are extremely costly because there is a very steep learning curve involved, primarily in understanding the operation of the system by examining the source code before any changes can be made.

[2] Where Object-oriented programs or databases running on different platforms such as DOS or UNIX™ have to communicate with each other, a third party program, an object middleware can be installed to translate requests from one object system to another. The Object Management Group (OMG) defines a standard to which object systems have to adhere if their requests are to be easily translated from one object system to another. The Common Object Request Broker Architecture (CORBA) is an object middleware mechanism that encapsulates the standard proposed by the OMG. OLE is a rival standard proposed by Microsoft mostly on PC platforms.

the Object Management Group's Common Object Request Broker Architecture (CORBA) [70]. The implications of such an architecture on the future use of neural networks are phenomenal. Self-learning or trained neural networks that aid intelligent decision making will become a standard part of many new software packages as commercial software designers seek to incorporate intelligence into their products. Neural networks will become an indispensable utility as more people seek software with a minimum level of intelligence that can adapt to their everyday needs. More innovative and practical uses will be developed for neural networks as a lot more people get to understand the advantages and limitations of the technology.

4.2 Software Architecture

All software systems should have an architecture so that they can be easily integrated with existing systems or easily modified or customised when the requirements of an an organisation or application changes. In [71], architecture is described as the structuring paradigms, styles and patterns that describe or make up a software system. A software architecture proclaims and enforces system-wide rules regarding the organisation and access of data and the overall control of the software system [72]. The software architecture serves a number of extremely useful purposes. Some of the more common ones are as follows:-

- It acts as a basis for communication between different designers and between designers and users of a software system.
- It serves as a high level documentation of the system and as a starting point from which the effectiveness of the system can be measured or changed.
- It serves to provide users of the system with a certain degree of confidence about the robustness of the system and ensures that their investment in the software is protected.
- It acts as a boundary that delimits both the expectations and the implementation of the software system.
- It serves as a starting point from which modifications or maintenance changes to the system are made.

In this chapter, a new approach to constructing artificial neural network systems based on object-oriented techniques is presented. Detailed descriptions of the analysis and design process are presented. The approach makes use of the Object Modelling

Technique as a basis for the development of software neural network architectures. An object-oriented architecture typically derives it properties from the classes and class hierarchies found within class clusters in the problem domain.

4.3 System Development

Software systems development is a very complicated task. The complexity of software is due to the fact that, usually, the developers of a software system are different from eventual users of the system and also from the people who have to maintain the system. The system has to be developed from a problem statement where the true requirements are often intermixed with design decisions [73]. Most of the time, the problem statements are ambiguous, inconsistent or even wrong because the users are unsure of what the final system will do. Some requirements although precisely stated, will have unpleasant consequences on the system behaviour or will impose unreasonable implementation costs. The complexity is further increased by the very flexible nature of software itself, which can be constructed in an almost unlimited number of ways. This leads to high maintenance costs as any changes to the system will require a thorough understanding of the structure of the system if unpleasant side effects are to be avoided. The complexity has to be reduced or handled in an organised way if reliable, robust and maintainable software systems are to be constructed [74]. This is done by a process of decomposition where both the software system and the development process is divided into smaller, more manageable parts. The software development process can be divided into a number of phases as shown in Figure 4.3-1. The product of each development phase is a model or view of the system in miniature represented by a document or set of documents that become the input to the next development phase [19] .

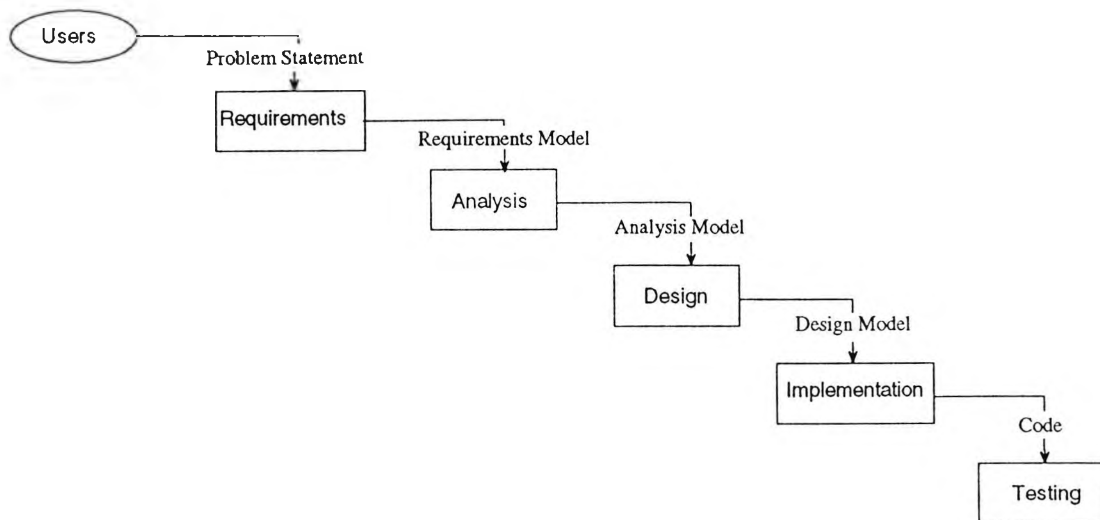


Figure 4.3-1: Phases in Software Systems Development

4.3.1 *Software Life-Cycle Models*

The term software life-cycle can be defined as a model used to help explain and understand the software development and maintenance process [75]. A software life-cycle model describes how the software development process progresses through the different phases. The aim of life-cycle models is to reduce the inherent complexity in the software development process in an attempt to deliver correct, reliable and robust software systems on schedule and according to budget. Different life-cycle models have been proposed with different characteristics. The most common of these life-cycle models are described below.

4.3.1.1 *The Waterfall Life-Cycle Model*

A schematic diagram of the waterfall model is shown in Figure 4.3-2. In the waterfall model, each phase has to be completed before the next phase can begin. Sometimes, the phases are allowed to overlap but the ordering of phases is strictly maintained. Very often, the different development phases are carried out by different groups of people. The strict waterfall model is not practical for large software projects firstly because the boundary between the phases are often fuzzy and it is extremely difficult to determine where one phase ends and the other begins. Secondly, large software projects require a long time to complete and this inevitably leads to changing requirements as the economic and business conditions change. Changes in requirements earlier on in the development cycle are cheaper and easier to incorporate but when changes are required later on, the effect on the budget, design, delivery schedule and documentation of the software system are usually disastrous.

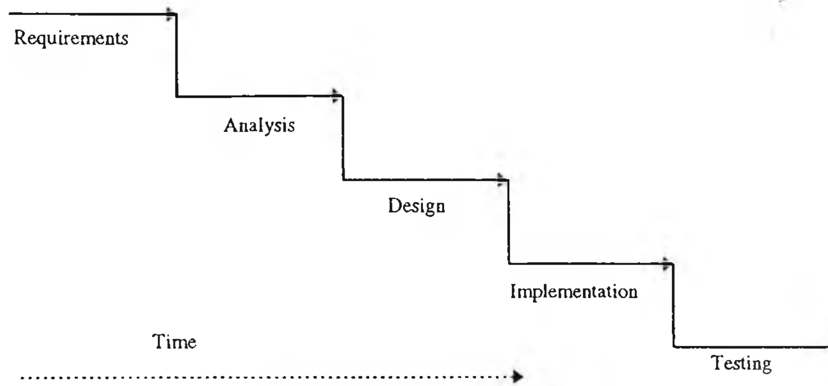


Figure 4.3-2: The Waterfall Life-Cycle Model of Software Development

4.3.1.2 Evolutionary Life-Cycle Model

The development of a large software system is a slow process that can take a long time to finish. Where the requirements are completely known at the start of the project, the waterfall model can be easily applied to develop the system. In most cases, the requirements cannot be completely determined at the start of the development process. Such systems are better developed in a step by step manner beginning with a few of its core functions. As understanding of the system functions evolve, new functions can be added. In this way, the system is incrementally enlarged until the desired level of functionality is attained. Such an incremental strategy also provides faster feedback during the development process. In practice, the system can be divided into parts according to requested services. The completion of each part extends the system functionality up to the finished product which comprises the whole of the system functionality.

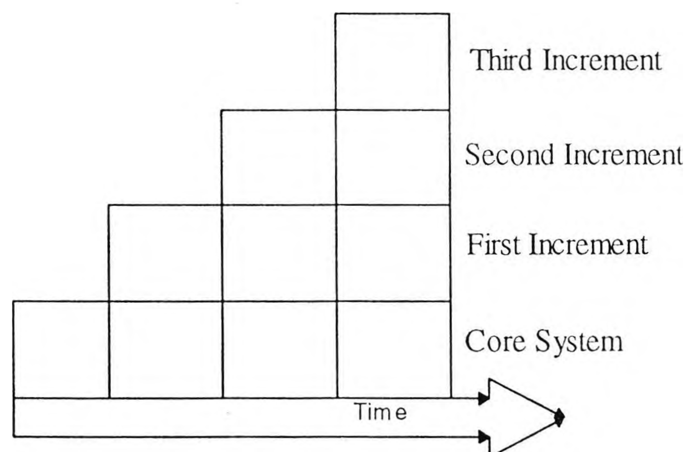


Figure 4.3-3: Incremental Software Delivery using Evolutionary Life-Cycle Model

4.3.1.3 Spiral Life-Cycle Model

The spiral model for software development describes how a software system can be developed to derive new versions and also how new versions of the software system can be developed from prototypes. The underlying concept of the spiral model is to minimise the risks associated with developing a software system either by use of prototypes or other necessary means [75]. The basis of the spiral model is a multistage stage representation through which the development process spirals [76]. At each stage, objectives, constraints and alternatives are identified, the alternatives are evaluated to identify and resolve risks, then the next version of the software system is developed and finally the next phases are planned. A very simplified schematic of the spiral model due to [74] is shown in Figure 4.3-4.

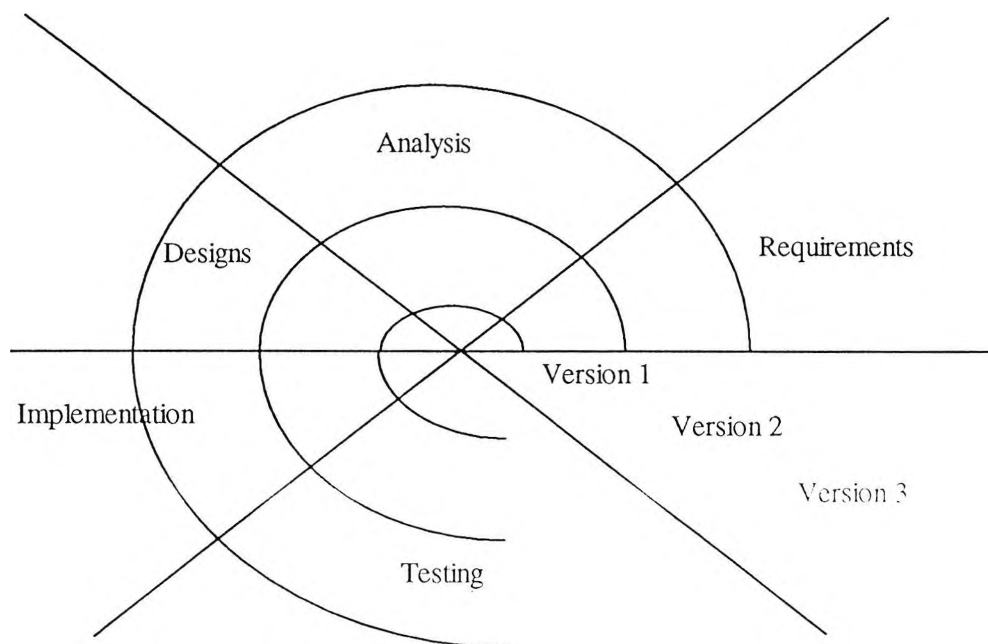


Figure 4.3-4: Spiral Life-Cycle Model for Software development

4.3.1.4 Prototyping

Where it is difficult to determine how a system is suppose to work either due to technical or functional reasons it is helpful to develop a prototype of the intended system. This is especially true when user interfaces and highly interactive systems are to be developed. The prototype focuses on the properties of the system that require

further insight. It allows the system developers to experiment with a number of design options and thus serves as a complement to incremental system development [74]. Prototyping is a useful technique for understanding an application and can act as a means of communication between the developer and the eventual users of a software system. Prototyping differs from incremental development in that the aim is not to create a fully working product, but to emphasise and demonstrate certain aspects of the intended system.

4.3.2 Phases of Software Development

The different phases in the software systems development process are briefly described below.

4.3.2.1 Requirements Definition

Requirements definition or requirements specification is usually required at the start of any software development process. A requirements document can be developed from information about the environment in which the software system will be used. A need for a new system or modification of an existing system is identified by users of the system. This leads to the creation of a problem statement which becomes the starting point for the software development process.

4.3.2.2 Analysis

Analysis is the study of a problem prior to designing a solution for it. The aim is to build a problem model, i.e., to create a description of what is required and what will eventually be built without attempting to say how it will be built. The product of the analysis process is a document or sets of documents that describe the problem in a clear, unambiguous and easily understood manner. The analysis document represents the analysis model. These analysis documents are completely in the problem domain and the vocabulary used in the description is consistent with those found in the problem statement. This allows the eventual users to easily understand and comment on the analysis model so that the correct system can be designed and subsequently implemented. Figure 4.3-5 shows an overview of the process of analysis. The inputs and sources of information into this phase include the following:

- The problem statement.
- Interviews with prospective users of the system.

- Knowledge about the problem domain, from similar systems that already exist.
- Real-world experience of the developers.
- Random data

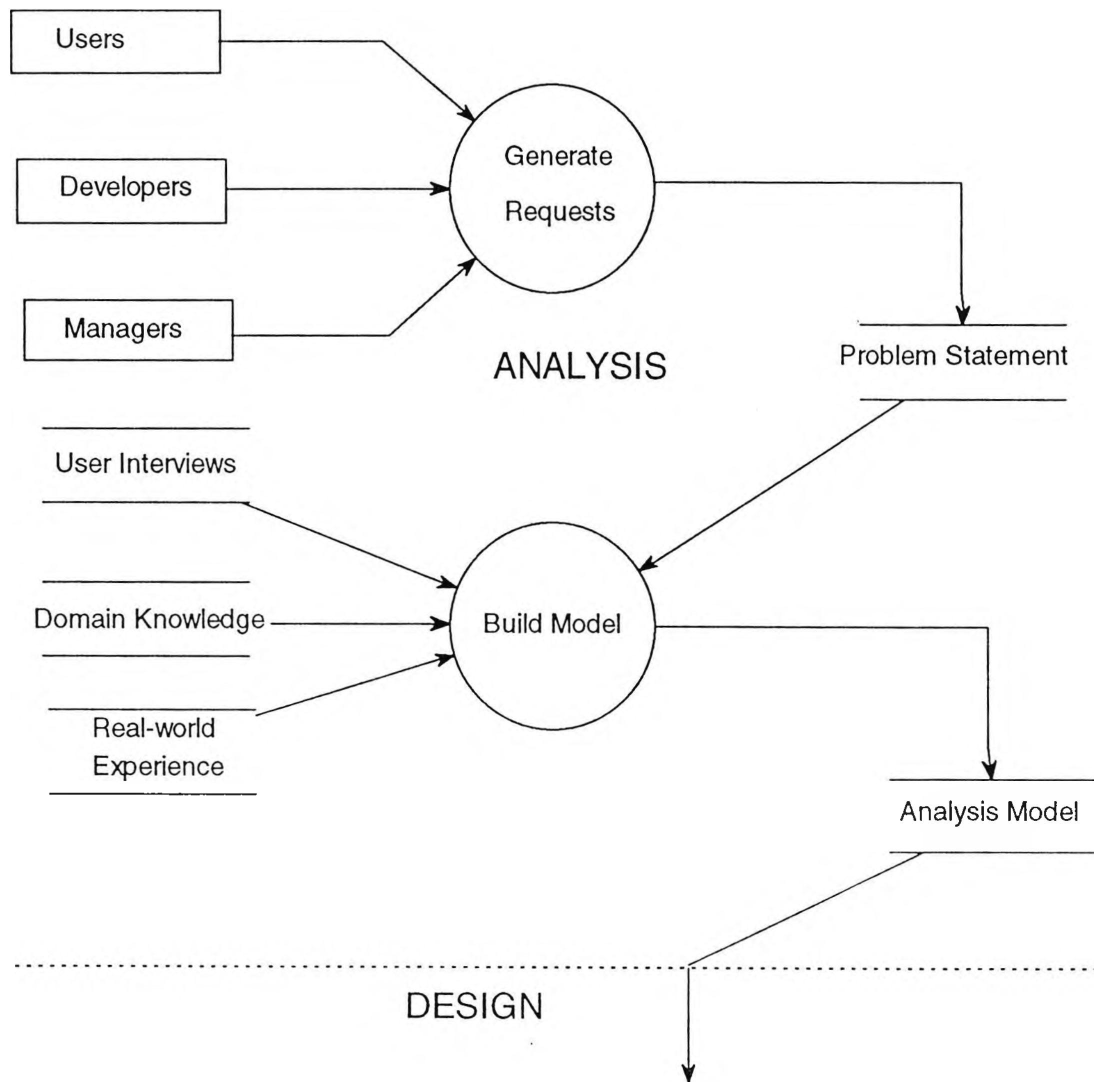


Figure 4.3-5: Overview of System Analysis

4.3.2.3 Design

The input to the design phase is the analysis model. The aim of the design is to construct a solution model by deriving a high-level strategy for solving the problem and building the solution. The process of design will refine the analysis model to take advantage of a particular implementation environment, which consists of the available hardware, operating system, network resources, programming languages and database management systems [16, 77]. The design phase is entirely in the solution domain and serves to structure and organise the software system to give it an overall architecture.

Two main stages of design can be identified; a systems design stage and a detailed design stage [16]. In the systems design stage, high level strategic decisions are taken about the overall architecture of the system, the implementation environment and the mixture of hardware and software required to realise the system. The different subsystems that make up the complete system are also identified and organised. In the detailed design stage, lower level decisions that directly affect the implementation of the systems such as choice of data structures and efficiency of algorithms and algorithm designs are carried out.

4.3.2.4 Implementation

The goal of the implementation phase is to realise the system according to the blueprints set out in the design phase. This can be done using a programming language or database management system.

4.3.2.5 Testing

The input to the testing phase is the finished program code or packages from the implementation phase. In the testing phase, the system is checked to make sure that the performance of the system meets the specifications laid out in the requirements document.

4.4 Object-Oriented Systems Development

Object models are abstractions built to understand a problem before devising a solution. Object modelling differs from traditional systems analysis and design techniques in that traditional systems development methods separate functions from data. Functions are active and have a specific behaviour while data is passive and holds information on which the functions act. Traditional design decomposes the system according to functions and the data is passed between these functions. If-Then or Case statements are required to reconcile differences between data formats. These statements do not add any functionality to the system. On the other hand, they make the finished program harder to read or understand and even harder to modify because of the possibility of side effects. The resulting systems are very unstable as changes in the data structure will require modifications in all functions which access the data structure. In object-oriented design, the emphasis is placed on building systems around objects rather than around functionality [21, 78, 79, 80, 81]. Objects encapsulate both data and the necessary functions required to access the data. Object-oriented systems are thus more resilient to

change because the objects correspond more closely to real world objects in the problem domain. For a particular problem domain, the set of objects are pretty much constant over time and modifications to an object's structure tend to be localised thus requiring only small modifications in the software system. Furthermore, information about objects can easily be conveyed from users to developers where as conveying information about functionality is very prone to errors and ambiguity. Finally, well designed objects can be easily reused between applications in the same problem domain which can lead to major reductions in the cost of developing new software systems. An object-oriented systems development process goes through similar stages of analysis, design, implementation and testing, as do conventional systems development. The major difference is that, in traditional systems analysis and design, different tools with different notations are used in the different phases. The result is a large semantic gap between the phases which leads to ambiguities and errors in the transition from one development phase to another. In object-oriented systems development, there is a direct translation between analysis, design and implementation models. This means that once analysis is completed the process of design and implementation can be significantly speeded up or automated. This alone provides much needed consistency and traceability to the systems development effort. Another major difference is that while traditional development projects usually follow a waterfall life-cycle model, the object-oriented paradigm follows an incremental and iterative life-cycle model. The end result is that working versions of the system are delivered orders of magnitude quicker than in traditional development projects.

Different methodologies have been proposed for carrying out object-oriented systems development. The methods differ in terms of the notation, diagramming types and the underlying process support for developing object-oriented software. A survey of the major object-oriented development methodologies is presented in Appendix D. The different steps in an object-oriented development process are described below.

4.4.1 Object-Oriented Analysis

Object-oriented software is organised as a collection of distinct objects that co-operate through message passing to solve a given problem. Each object is an abstraction that encapsulates both data and the operations that act on the data. The analysis process consists of the following activities:

4.4.1.1 *Finding Objects*

Objects can be found as naturally occurring entities or items in the problem domain. An object is typically a noun which exists in the problem statement or problem domain. A domain and/or behaviour analysis can be carried out to help uncover relevant objects in a particular application domain. Domain analysis is an attempt to identify objects, operations and relationships that domain experts perceive to be important about a particular problem or application domain [21]. It involves a survey of existing systems and domain experts to identify the key abstractions and mechanisms that have been previously employed and to evaluate which were successful. Behaviour analysis on the other hand seeks to identify objects by first understanding system behaviours, then assigning or attributing the behaviours to different parts of the system and finally recognising the parts which play a significant role in terms of their behaviours as objects. A further method for identifying objects is by Use Case analysis [70]. A Use Case is a particular pattern of usage of the system. It is a scenario that begins with some user initiating an action or sequence of events in the system. Objects in the system are identified by walking through each of the scenarios to find participating objects, their responsibilities and how they collaborate with other objects in the scenario.

4.4.1.2 *Organising Objects*

After the objects in the system have been identified, the objects need to organised into meaningful collections by discarding unnecessary objects and mapping out relationships between the remaining objects. Objects can be grouped together into classes based on a number of criteria that stem from classification theory. Using classical categorisation, the objects that have a given property or collection of properties are grouped together into a common category or class. Clustering techniques can be used to formulated conceptual descriptions of the classes and then classifying the objects according to how closely they fit a particular description. Finally using prototype theory, a class of objects is represented by a single prototypical object and the other objects classified based on their similarities to the prototype.

4.4.1.3 *Describing Object Interaction*

In order to determine how the different objects fit into an object-oriented software system, different scenarios can be described showing the involvement of the object of interest and its interaction with other objects.

4.4.1.4 Defining Operations on Objects

An object's operations can be naturally determined when the behaviour of the object and the interface to it are considered. The behaviour of the an object represents the complete set of messages that can operate on an object or that the object can respond to. A number of techniques including Use Cases and Class-Responsibility-Collaborators (CRC) [79] cards have been successfully used during analysis not only for the discovery of objects but also to determine object operations and the interactions between objects.

4.4.2 Object-Oriented Design

The aim of Object-Oriented Design (OOD) is to make high level decisions about how the problem just analysed will be solved. The input to the design phase is the analysis models. Object-Oriented Design defines the architecture of the system and addresses implementation issues concerning object interaction, relationships between objects and choice of implementation environment and implementation language. The design process also organises the objects in the system into subsystems and assigns these to physical processors based on the system's architecture.

4.4.3 Implementation

The implementation phase converts the design into code. An object-oriented programming language like C++, Smalltalk or Common Lisp Object System (CLOS) is preferred because they include constructs to directly realise the design into code. The objects in the analysis and design model can be directly represented in C++ or Smalltalk classes. Constructs also exist to directly represent complicated aggregation and generalisation/specialisation relationships between objects in the problem domain.

4.4.4 Object-Oriented Testing

Testing is a defined, repeatable and measurable process that is performed on the software to qualitatively demonstrate that the software functions or fails to function as specified in the requirements. Classes and objects in object-oriented programs must be tested to a greater degree that traditional programs because they are subject to reuse in projects and environments which are very different from those for which they were designed. Unit testing is carried out on individual classes to ensure that the satisfy their required behaviour. The test data can be obtain from test cases developed during the requirements determination phase or from random data. Finally, integration testing is carried out on the finished system consisting of all the different objects to demonstrate that the software meets the initial specifications.

4.5 Object-Oriented Analysis and Design of Associative Neural Networks

This section presents the application of object-oriented techniques to the analysis and design of Associative Neural Networks. The Object Modelling Technique (OMT) is applied to the creation of both the analysis and design models. The finished design is implemented using the C++ programming language on a SUN SPARC 10. Unit testing is carried out on the individual classes to ensure correct behaviour of the ensuing objects. Finally, integration testing is carried out on the finished neural network using both random data and standard test data.

4.5.1 Domain of Associative Neural Networks

Associative neural networks are a class of fixed weight neural networks. These networks have a common property that the weights and/or thresholds (bias) are usually pre-calculated and pre-stored. Associative networks could be feedforward or feedback neural networks. A class hierarchy for associative neural networks was presented in chapter 1. The inputs to the networks can be binary values (0/1), bipolar (-1/1) or real numbers. The weights are derived from the input patterns usually by taking a correlation of the input vectors pairs. Processing of the input vectors to compute the weight is similar, for both bipolar and real values. A slight modification is required for binary values. The computed weights are pre-stored in a weight matrix for use during associative retrieval.

4.5.2 Problem Statement

An associative neural network can be used to store and retrieve associations. An association is a vector or a pair of vectors that are presented at the input to the network. Associations can be binary, bipolar or real valued. The network operates in two phases: a storing phase and a retrieving phase. During the storing phase, the network is presented with a training file containing a set of associations to be stored. These are the patterns or exemplars that the network is required to learn. During the retrieving phase, one part of an association (the key) is presented and the network has to retrieve the second part of the association. The key could simply be a corrupted version of one of the stored vectors in which case a holographic retrieval can take place.

4.5.3 Identifying Objects

The main objects in an associative neural network can be easily identified by careful analysis of the problem statement and from domain knowledge. They include all the different classes of associative neural networks shown in the class hierarchy and some

key nouns found in the problem statement. An enumeration of the domain objects is as follows [82]:

Table 4.5-1: Associative neural network domain objects

Associative Neural Network	Feedforward Neural Networks	Training File
Linear Associative Memory	Non-Linear Associative Memory	Hamming Network
Hopfield Network	Bi-directional Associative Memory	Associations
Weight	Feedback Neural Networks	Key
Vector	Patterns	Exemplars
Threshold		

Careful analysis of the domain objects shows that some of the listed objects are aliases for previously listed objects e.g. Associations and Exemplars. There could also be more than one named object performing the same function in the context of associative neural networks and so only one of them needs to be listed. It is necessary to prepare a data dictionary in order to decide and eliminate unnecessary domain objects. A dictionary entry clarifies the role an object plays in the system and so reduces misinterpretations, ambiguities and name clashes. A dictionary entry for an object consists of the objects name, a short description, associations with other objects and any constraints that are imposed on the object. The objects attributes and operations can also be included.

The Data Dictionary

Feedforward Associative Neural Networks — An associative memory network where neurons have no connections to themselves nor to neurons in the same layer or preceding layers. Feedforward networks may or may not have threshold elements.

Feedback Associative Neural Networks — An Associative neural network where the neurons can have connections to themselves and to neurons in the same or preceding layers. Feedback networks have thresholds, states, and an energy which is minimised in the retrieval process.

Linear Associative Memory — A basic feedforward Associative neural network. Takes pairs of vectors as input in the storing phase. The weights are computed by correlation of the input vectors.

Non-Linear Associative Memory — A Feedforward Associative memory containing non-linear neurons. The network does not require weights to be computed. The training patterns are simply stored in the network and retrieved as required. During recall, the key is multiplied by each X vector in the training pattern to produce a score which undergoes non-linear processing resulting in a binary decision vector used to select a stored output.

Hamming Network — A Feedforward associative memory network containing a hamming distance calculator. In the storing phase, only X vectors are presented and stored in the network. Hence no weight computation is required. In the retrieving phase, a key is presented and the output vector is determined as the stored vector which is closest to the key in terms of hamming distance.

Bi-directional Associative Memory — A Feedback Associative neural network with two weight layers and capable of bi-directional recall.

Hopfield Network — A single layer Feedback associative neural network. Neurons are not allowed to have connections to themselves even though the have connections to other neurons in the same layer.

Associations — An association is a single vector or a pair of vectors to be stored in a network.

Exemplars — Another name for an association, used in the context of multilayer perceptron networks.

Patterns — A set of associations stored in a data file. Could be either training patterns or test patterns.

Weight — The value of a connection between two neurons. A weight matrix connects layers of neurons.

Threshold — A vector of values used for non-linear decision making in Associative Neural Networks.

Training File — A plain text or binary data file containing Patterns.

Vector — A data type or object created to represent a one dimension array of values.

Key — A named vector.

4.5.4 *Organising the Objects*

Objects do not exist in isolation. The next step in the analysis process is to organise the domain objects and map out the associations between them. The domain objects and their relationships can be represented by a domain object model. A domain object model describes the static structure of objects and the relationships between objects found in the domain of the problem. Further objects can be added which do not directly relate to the problem to be solved but are required to provide operations and services to the domain objects. These are called system objects and the combination of system and domain objects forms the system object model. Figure 4.5-1 shows a system object model for the class of associative neural networks. The associative neural network is described as consisting of one or more weight layers. In the dictionary description, a weight is considered to be a single value. Domain knowledge shows that the weights between layers of neurons can be represented by a matrix. Even though matrices are not domain objects, they are still valid objects in the system object model as they help to efficiently describe and express the structure of the system. A similar explanation can be used to justify a generalisation relationship between vector and threshold. For each neuron, there can be only one threshold value. For a layer of neurons, all the threshold values can be collected to form a vector. The vector representation is more efficient in terms of expression when communicating design decisions. In the design, an associative neural network object has been created which forms the basis for all other types of associative memory networks. This object defines the interface and representation of associative neural networks. All other feedforward and feedback associative networks are designed as special cases of the base associative neural network. The interface and representations are inherited and then refined by adding the extra operations and data required to exhibit a specific behaviour. There is thus a generalisation/specialisation relationship between the base Associative memory object and each of the derived neural network objects. This relationship can be read as follows: A Feedforward Associative Memory is a "kind of" Associative Memory. The same relationship holds for Feedback Associative Memories. Linear, Non-linear and Hamming neural networks are all a "kind of" Feedback Associative Memory network which automatically makes all three a "kind of" Associative Memory.

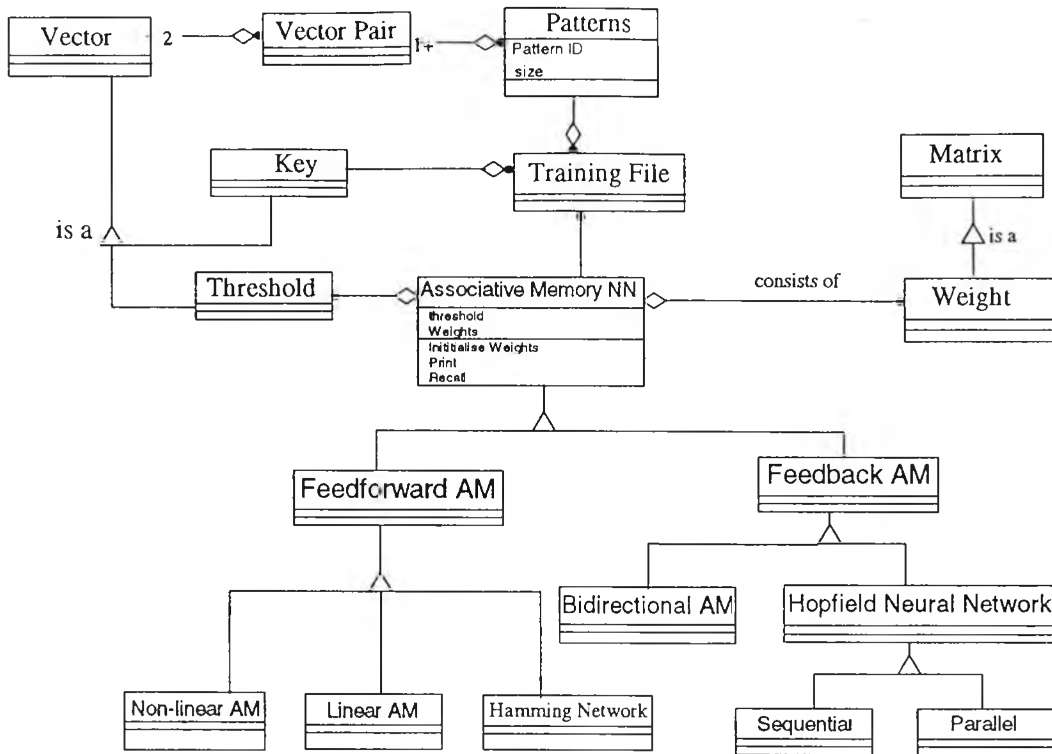


Figure 4.5-1: System Object Model for an Associative Neural Network

4.5.5 Describing Object Interactions

Object-oriented systems are made up of collections of autonomous or semi-autonomous objects that interact or co-operate to solve a specific problem. Object interactions include the set of messages that an object should respond to and the set of behaviours that an object can exhibit. Object interactions can be discovered using scenarios or Use Cases. A scenario is a sequence of events that occur during one particular execution of a system. Scenarios use event trace diagrams to describe sequences of events and the objects exchanging those events during a particular execution of the system. Two main scenarios can be envisaged for the Associative neural network classes. A *train* scenario shows the sequence of events that take place when the network is used to store a set of patterns. Figure 4.5-2 shows the an event trace diagram for the *train* scenario. In a *retrieve* scenario the sequence of events that lead to a recall is shown in Figure 4.5-3.

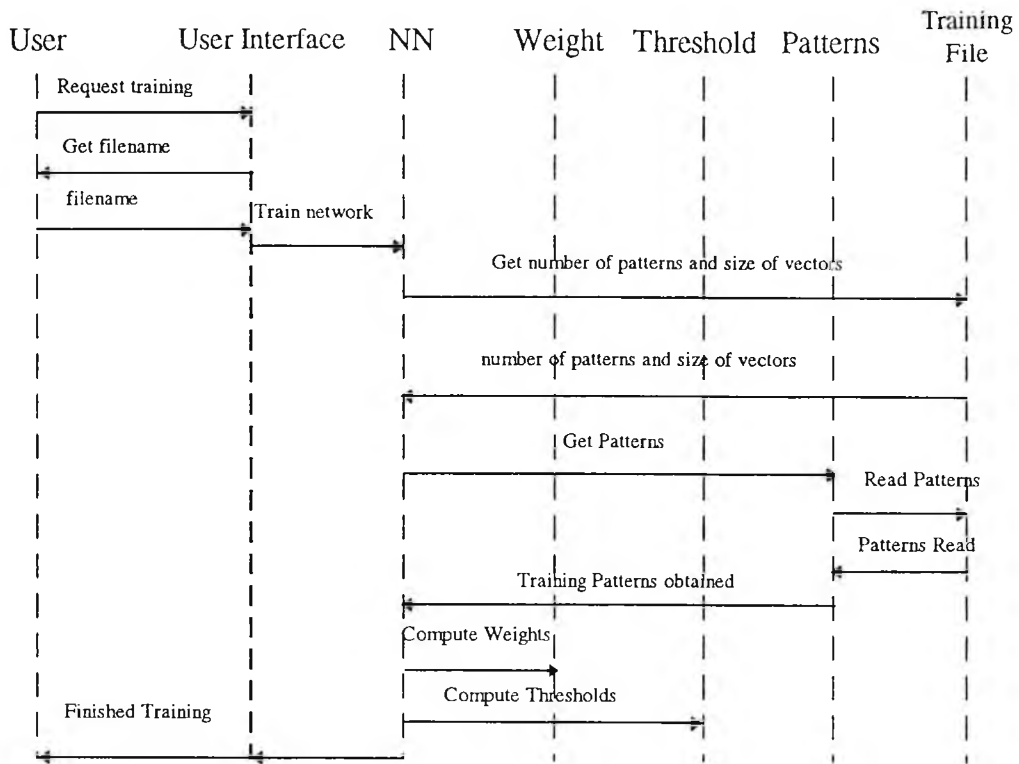


Figure 4.5-2: Event Trace Diagram for a Training Scenario

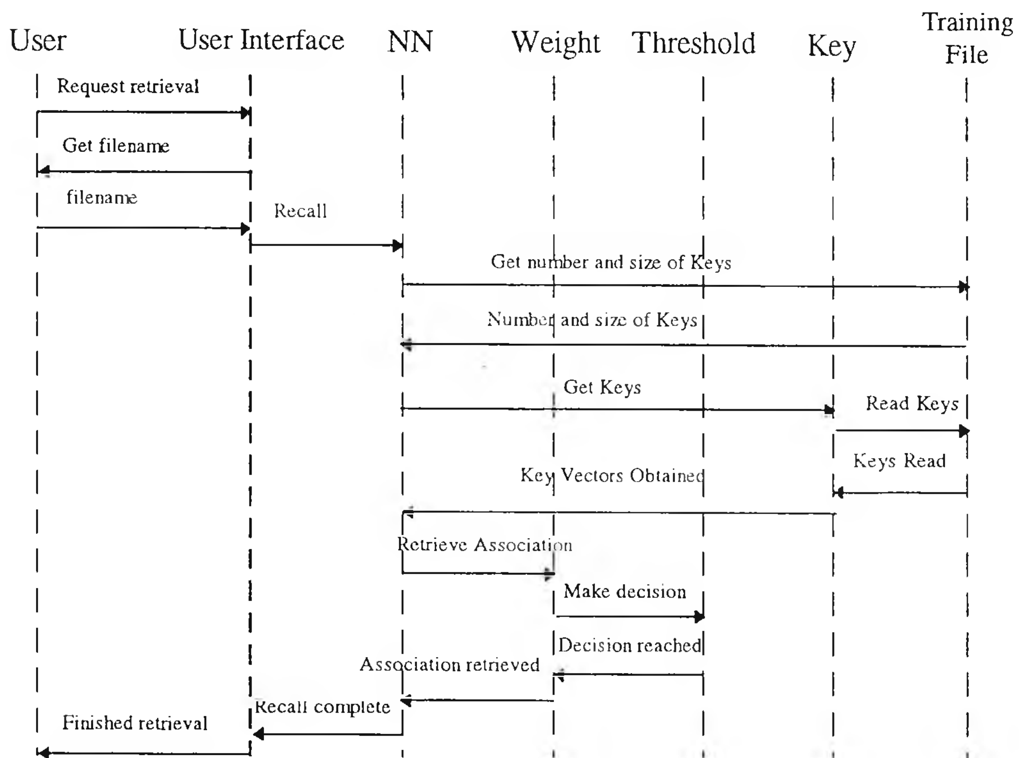


Figure 4.5-3: Event trace diagram for a *retrieve* scenario

4.5.6 Defining Operations on Objects

Event trace and object interaction diagrams aid in refining the relationships between different objects. Most of the operations required of an object can also be obtained from an examination of the flow of events to and from the object. This however does not represent the complete picture. Where an object has very complicated dynamic behaviour, all its operations cannot be captured simply by using event trace diagrams. In such cases, a state-transition diagram can be used to represent a dynamic model of the system. A state diagram for an object is a graph whose nodes represent the different states in which the object can be in, and whose directed edges are transitions between the states. State transitions are caused by labelled events entering a state. For the Associative Neural Network, a dynamic model has been developed for the base Neural Network object as shown in Figure 4.5-4.

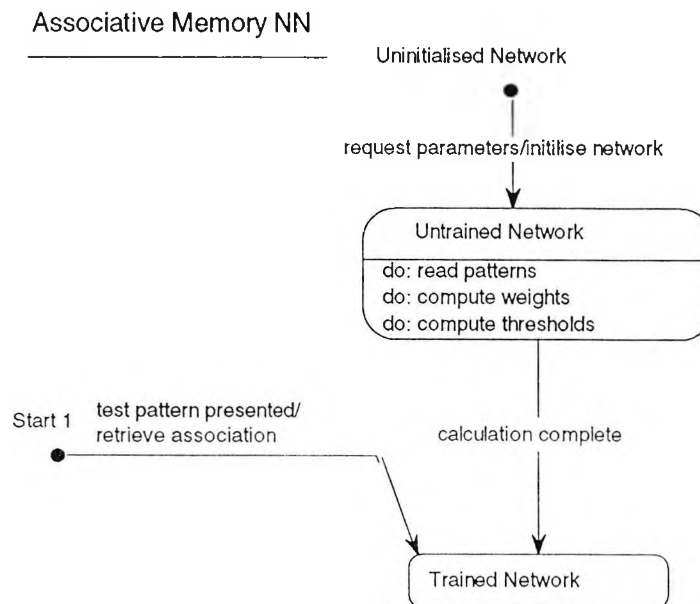


Figure 4.5-4: Dynamic Model of Associative Neural Network

The dynamic model shows the need for an *initialise* member function which uses requested network parameters to initialise the network. Extra functionality and further objects are also required to obtain and process network parameters from the user and also for pre-processing of data patterns and post-processing of test results if necessary. Finally, a functional model can be used to describe what happens in the Neural Network. The functional model shows what the different inputs to an Associative Neural Network are and the computation required to transform these inputs into output

values. The functional model consists of the System Context shown in Figure 4.5-5 and the level 1 dataflow diagram in Figure 4.5-6.

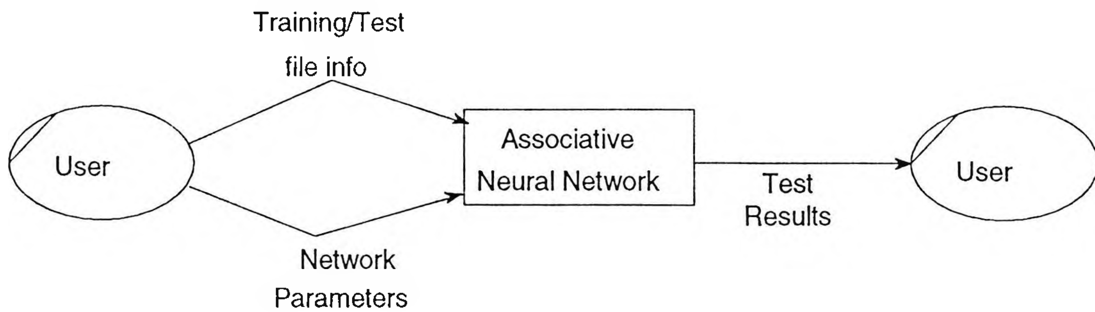


Figure 4.5-5: System Context for an Associative Neural Network

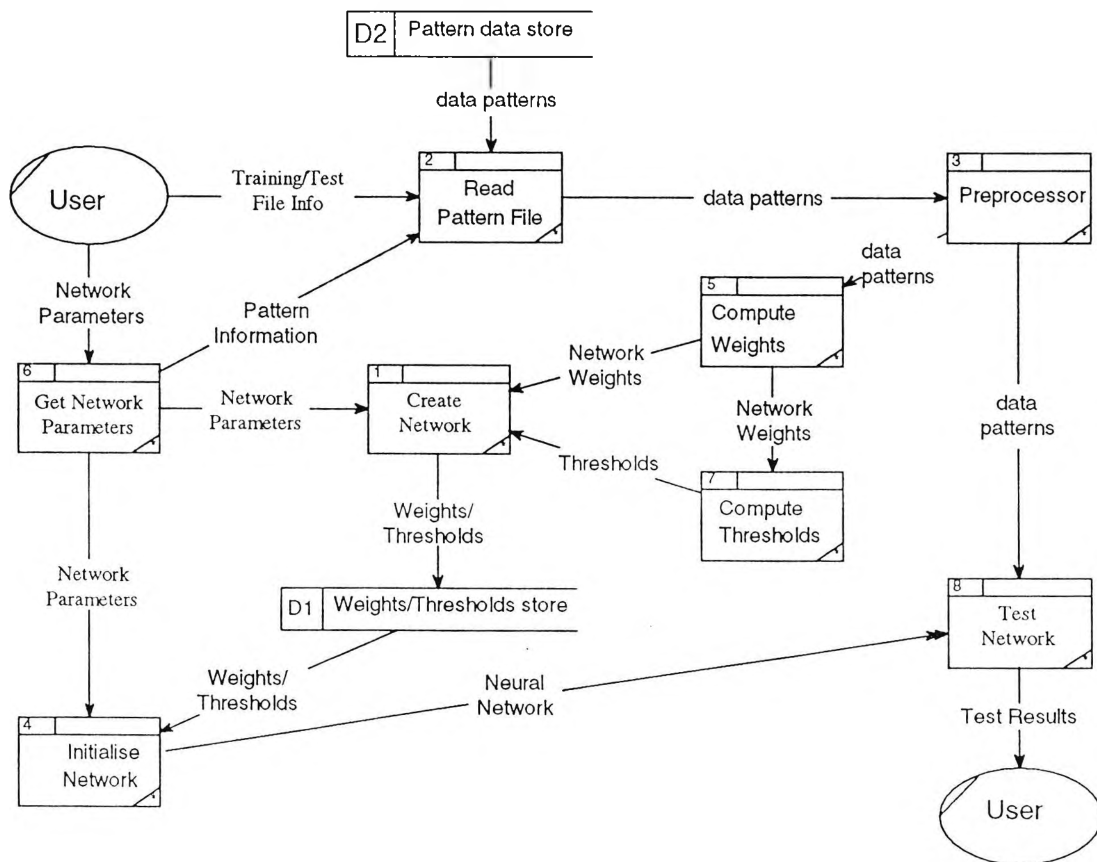


Figure 4.5-6: Level 1 data flow diagram that describes the operation an Associative Neural Network

4.5.7 Neural Network Systems Design

In the design process, an architecture for the Associative Neural Network is produced based on information contained in the different models constructed during analysis. The design process consists of partitioning the Neural Network System into subsystems that perform specific functions in the context of the Neural Network. In this design, the following subsystems have been identified:

- The Input subsystem,
- The Data Pre-processing subsystem,
- The Training/Testing subsystem
- The Analysis subsystem.

The Input subsystem consists of object(s) that handle interactions with the user to obtain network parameters and information relating to the data patterns. The Data Pre-processing subsystem consists of objects that handle the reading of desired training/test patterns from a disk file or other media such as sensors and its subsequent pre-processing. The Training/Testing subsystem includes objects that make up the neural network, weight initialisation, reading and writing of network weights and final activation values. The Analysis subsystem includes objects that handle post-processing of activation values and test results to produce statistical information about the performance of the neural network. The flow of information amongst the different subsystems in the Neural Network is shown in Figure 4.5-7. This architectural description is applicable to the design of most of classes of Neural Networks including Multilayer Perceptron Networks trained by backpropagation and Kohonen Self-Organising Maps. The different subsystems are layered on top of the operating system to create the complete Neural Network system as shown in Figure 4.5-8. Such layering of partitions constitute a horizontal decomposition of the neural network system. It is also possible to vertically decompose the neural network system into weakly coupled subsystems. The choice of partitioning depends on the implementation environment which includes the operating system, hardware, software and network resources.

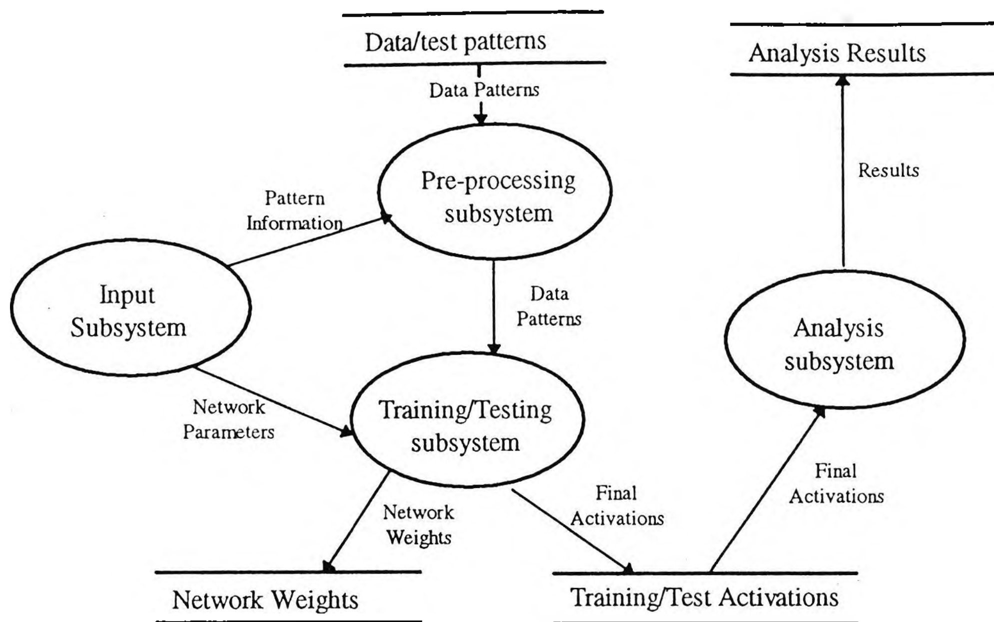


Figure 4.5-7:Information flow among the different subsystems in a Neural Network

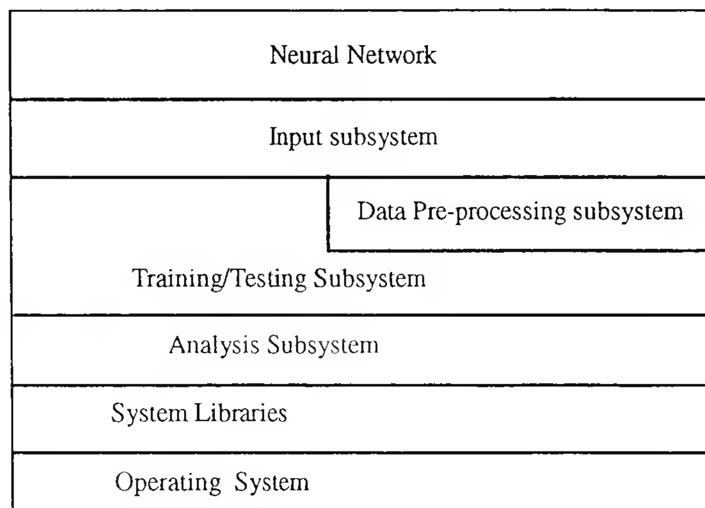


Figure 4.5-8: System Block diagram of Neural Network

4.5.8 Implementation and Testing

The finished design has been realised using the C++ programming language on a Sun SPARC™ 10 running at 50 MHz. The different objects that make up the system have been implemented using the class mechanism provided in C++. Class declarations for the objects that make up the Neural Network System are shown in Appendix E. Because the neural network has been developed as an object-oriented program, testing takes a different approach from that used in conventional systems design. Each object exposes a certain behaviour which is expressed in the C++ class. The classes are

instantiated into objects which can then be independently tested for the required behaviour. Tested objects are then pooled together into the different subsystems for testing. Finally, the subsystems are pooled together into the complete Neural Network System for an integration test. This form of testing relies on the fact that the objects that make up the system are independent entities which can exist outside the system. Thus their creation and testing is completely decoupled from the system development process leading to more robust architectures and eliminating the need for exhaustive testing on a grand scale to prove correctness. Figure 4.5-9 and Figure 4.5-10 show unit test results for the base Associative Memory network and the BAM network respectively. In each case, the neural network is supplied with a pattern file containing 2 training exemplars. The network weights are computed by correlation or outer product of the input patterns and then stored in a weights file. A separate file can be supplied with the test vectors and the neural network performs associative recall if the test vectors are sufficiently close to the original input vectors.

The default processing assumes that the input vectors are binary. If real input vectors are to be used, a separate instance of the neural network is required. The default binary processing will have to be overridden with the appropriate arguments when the associative neural network constructor is called. In this way, the neural network object does not have to carry out any pre-processing or sanity checks on the input patterns. This is a design decision to prevent excessive and unnecessary computation in the neural network object which needs to be kept small and compact so that it an be easily understood and modified and can be reused without excessive overheads. Furthermore, the pre-processing and error checking can be handled by a separate subsystem.

```
Script started on Wed Sep 25 10:22:13 1996eeisun10% m ex0.run
eeisun10% tassoc
 This is the test version of the Associative Neural Network
   The program requires both training and test input pattern files.
   Pattern files are of the form:
            number of patterns: #
            number of inputs:   #
            number of outputs:  #
            .
            input vector
            output vector pattern_ID
            .
            .
Enter Pattern filename: twt.pp
Training Patterns
1 1 1 1 0 1 1 1 1
1 1 1 1 0 1 1 1 1  1
1 0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0 1  2
Weights stored
9 9 1
 2  0  2  0  0  0  2  0  2
 0  2  0  2 -2  2  0  2  0
 2  0  2  0  0  0  2  0  2
 0  2  0  2 -2  2  0  2  0
 0 -2  0 -2  2 -2  0 -2  0
 0  2  0  2 -2  2  0  2  0
 2  0  2  0  0  0  2  0  2
 0  2  0  2 -2  2  0  2  0
 2  0  2  0  0  0  2  0  2

-4 -3 -4 -3 3 -3 -4 -3 -4
Testing network
Enter test pattern filename: tswt.pp
Enter number of test patterns: 2
 1  0  1  1  0  0  1  0  1
 0  1  0  1  0  1  1  1  1
 1  0  1  0  1  0  1  0  1
 1  1  1  1  0  1  1  1  1
eeisun10% ^D
script done on Wed Sep 25 10:22:53 1996
```

Figure 4.5-9: Unit test results for the Associative Memory network

```
Script started on Wed Sep 25 10:25:34 1996eeisun10% m bam.pp
Number of Patterns: 2
Number of Inputs: 10
Number of Outputs: 6

1 0 0 1 0 1 1 0 0 1
1 0 0 0 0 1 1
1 1 1 0 0 0 1 1 0 0
1 1 1 1 0 0 2
eeisun10% tbam
 This is the test version of the BAM Neural Network
          .
          .
          .
  Enter Pattern filename: bam.pp
Training Patterns
1 0 0 1 0 1 1 0 0 1
1 0 0 0 0 1  1
1 1 1 0 0 0 1 1 0 0
1 1 1 1 0 0  2
10 6 1
  2  0  0  0 -2  0
  0  2  2  2  0 -2
  0  2  2  2  0 -2
  0 -2 -2 -2  0  2
 -2  0  0  0  2  0
  0 -2 -2 -2  0  2
  2  0  0  0 -2  0
  0  2  2  2  0 -2
 -2  0  0  0  2  0
  0 -2 -2 -2  0  2

-0 -2 -2 2 -0 2 -0 -2 -0 2
  2  0  0  0 -2  0  2  0 -2  0
  0  2  2 -2  0 -2  0  2  0 -2
  0  2  2 -2  0 -2  0  2  0 -2
0  2  2 -2  0 -2  0  2  0 -2
 -2  0  0  0  2  0 -2  0  2  0
  0 -2 -2  2  0  2  0 -2  0  2
```

```
-0 -0 -0 -0 -0 -0

Emin -64
initial vector  0 0 0 1 0 1 1 0 0 1

Out zero 2 -6 -6 -6 -2 6

Out zero 1 0 0 0 0 1
1 0 0 1 0 1 1 0 0 1
Energy -64
Out zero 4 -6 -6 -6 -4 6

Out zero 1 0 0 0 0 1
1 0 0 1 0 1 1 0 0 1
Energy -64
Out zero 4 -6 -6 -6 -4 6

Out zero 1 0 0 0 0 1
1 0 0 1 0 1 1 0 0 1
Energy -64
Out zero 4 -6 -6 -6 -4 6

Out zero 1 0 0 0 0 1
1 0 0 1 0 1 1 0 0 1
Energy -64
Out zero 4 -6 -6 -6 -4 6

Out zero 1 0 0 0 0 1
1 0 0 1 0 1 1 0 0 1
Energy -64      1 0 0 1 0 1 1 0 0 1
eeisun10% ^D°
script done on Wed Sep 25 10:26:17 1996
```

Figure 4.5-10: Unit test results for the BAM network

4.6 Discussion and Conclusions

The pace of Neural Network development remains unabated. This has been fuelled by an increasing requirement by industry for supra intelligent applications in order to gain a competitive edge. Early adopters of Neural Network technology have hitherto been content with hiring PhDs to develop bespoke applications. Others have turned to off-the-shelf packages such as Neural Works Professional™, NeuronLine™ and Neural Network toolboxes on standard mathematical packages as a base from which Neural Network solutions to specific problems can be developed. Finally, vast amounts of public domain Neural Network software have been made available at various Internet sites around the world. There is a major difference between those and the approach adopted in this thesis. Off-the-shelf Neural Network software and toolboxes can be regarded as specific instances of a Neural Network development effort with hard coded design decisions. The net effect is that they are extremely inflexible and very difficult to customise or integrate into other systems development efforts. In this thesis, Neural network software development has been approached from a systems development perspective. The analysis methods used above can be applied to the development of most neural network architectures. The system design process produced a software architecture for associative neural networks. The same software architecture can be reused in the design of ART networks or Kohonen neural networks as well as multilayer perceptron (MLP) networks trained by backpropagation. The software architecture makes the completed system easier to understand and extend. It also provides a framework in which modifications in the design or implementation can be incorporated so that the system does not disintegrate when necessary maintenance or upgrade is carried out. Finally and most importantly, the software architecture makes it easier to incorporate neural network technology into other industrial systems development efforts by reducing the effort required from non-technical and non expert application designers wishing to incorporate neural technology into their designs.

Chapter V

APPLICATIONS OF NEURAL NETWORKS

5.1 Introduction

This chapter presents two case studies where supervised learning Neural Networks have been successfully applied to solving real world problems. During the course of this thesis, a number of different projects have been carried out involving the application of the neural networks developed to different problem areas [62, 83-89]. This section presents the application of Multilayer Perceptron (MLP) networks trained by backpropagation to solve problems in the field of power systems engineering. The first case study describes procedures and results of applying neural networks to fault diagnosis in HVDC systems. In the second case study, an application specific neural network is developed to identify unknown parameters in linear and non-linear dynamic systems such as generators in power systems. Training data is obtained by simulation using both EMTP (the Electromagnetic Transients Program) and the C++ programming language. Supervised learning networks trained by backpropagation are very widely used in application areas because they are comparatively easier to model, implement, set up and train than competing network paradigms.

5.2 Application of Neural Networks to Fault Diagnosis in HVDC systems

In this section, the results of applying artificial neural networks to fault diagnosis in HVDC systems is presented. Fault diagnosis is carried out by mapping input data patterns, which represents the behaviour of the system to one or more fault conditions. Each fault condition (including no fault) is represented by a 4 bit binary number which acts as the fault identifier. The behaviour of the HVDC system is described in terms of time varying patterns of conducting thyristors and AC & DC fault characteristics. A three-layer MLP network trained by backpropagation is used for the classification. The network is configured with 20 input nodes, 12 hidden nodes and 4 output nodes. The use of a 4 bit binary number implies that a total of 16 different fault conditions can be represented.

5.2.1 Introduction

Neural network applications in power systems have increased dramatically in the past five years. Applications have spread into such diverse areas as load forecasting [90, 91] and turbogenerator regulators [92] to power system stabilisers [87] and on-line fault section estimation [93]. Fault detection and diagnosis in real time has traditionally been done using fault diagnostic expert systems. The exhaustive search process carried out by expert systems is too time consuming to be done in real-time. A neural network approach provides a fast alternative when real-time processing is required. In [94], a neural network solution was proposed for detecting high impedance faults in power distribution feeders. In this section, a fault diagnostic neural network is designed to classify faults in an HVDC system based on the AC and DC voltages and also the conduction patterns of the thyristors.

5.2.2 The HVDC System

The HVDC system consists of three subsystems; the converter subsystem, the transmission subsystem and the inverter subsystem. The converter is the basic component involved in the conversion of power from AC to DC. The behaviour of the converter under normal and fault conditions provides data that is used for fault diagnosis in the HVDC system. Figure 5.2-1 shows the configuration of a 6-pulse converter.

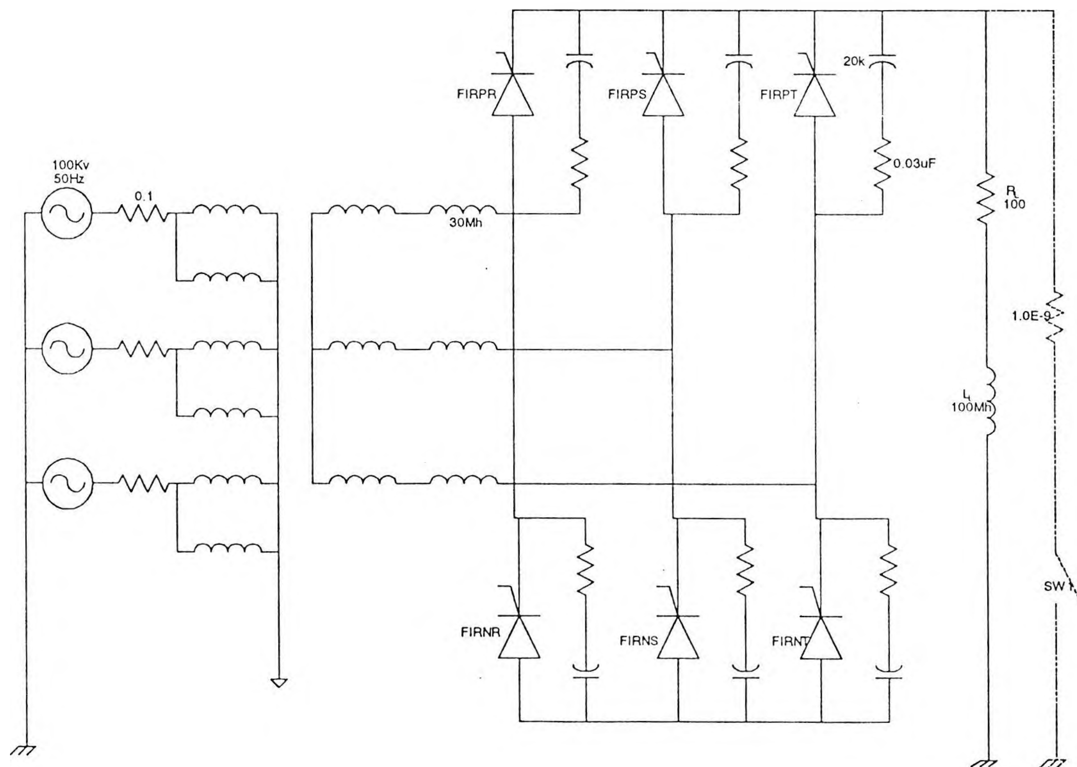


Figure 5.2-1: Configuration of a six-pulse HVDC Converter

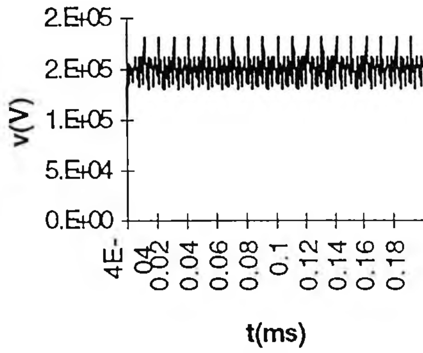
Faults external to the converter are treated as system faults. Both the switching sequence and the conduction pattern of the thyristors that make up the converter are periodic under normal conditions, but the conduction pattern tends to be unpredictable when a fault occurs. Data for both normal and the desired fault conditions is used to train the network. It is hoped that the variation between pre-fault data and fault data provides sufficient discrimination on which the neural network learning can be based. The data is obtained from a computer simulation of the system using the Electromagnetic Transients Program (EMTP) [95]. The relevant data is extracted from the simulation data and pre-processed for use as input to the neural network. Under normal operations, training data is extracted at random at any point in time from simulation data. On the other hand, only the immediate pre-fault and post fault data is extracted for a fault condition. Below is list of possible faults that can occur in an HVDC system.

- **no fault**
- **External fault**
  - AC fault, single-phase to ground
  - AC fault, double-phase to ground
  - AC fault, three-phase to ground
  - DC fault
- **Converter fault**
  - single commutation failure
  - single repetitive commutation failure
  - double repetitive commutation failure
  - double successive commutation failure
  - double not successive commutation failure
  - double not successive repetitive commutation failure
  - arc quenching
  - arcthrough
  - arcback
  - misfire

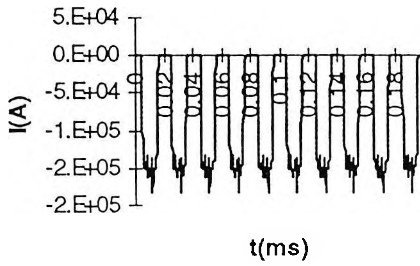
Figure 5.2-2 shows some current and voltage waveforms for no fault and a variety of fault conditions.

Voltage Waveforms

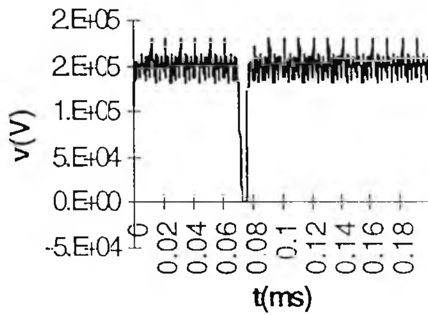
No Fault



Thyristor Voltage Conduction Pattern

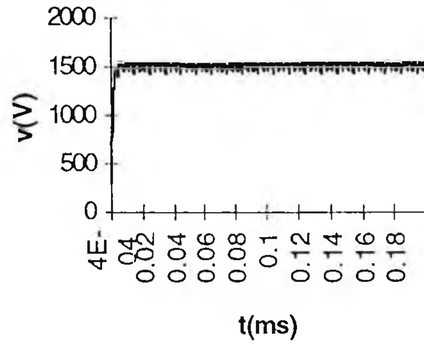


Single Commutation Failure

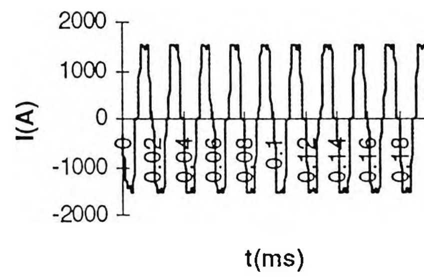


Current Waveforms

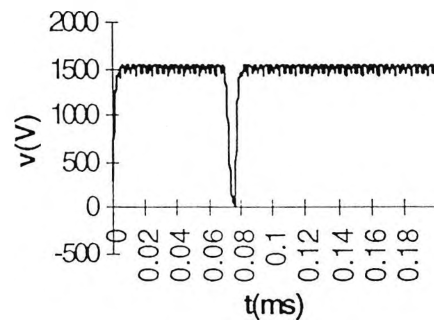
No Fault



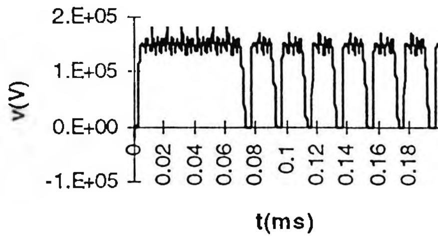
Thyristor Current Conduction Pattern



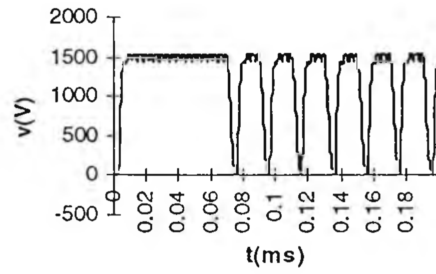
Single Commutation Failure



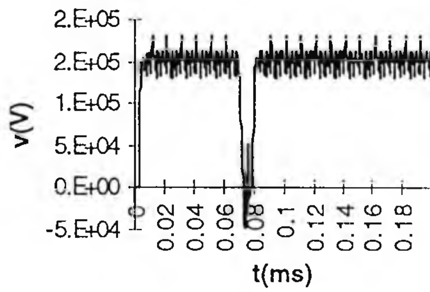
Double Not Succesive Commutation Failure



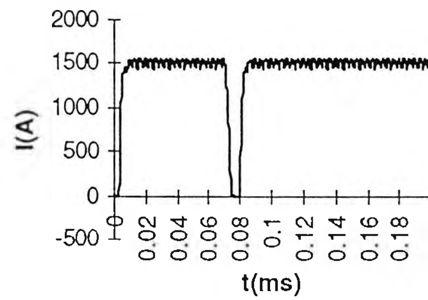
Double Not Succesive Commutation Failure



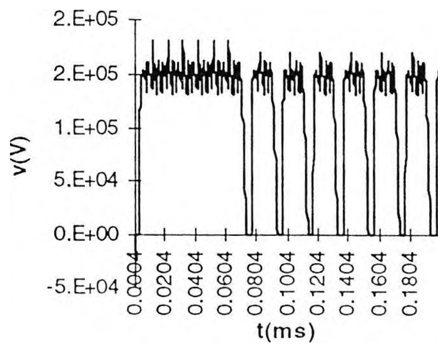
Double Succesive Commutation Failure



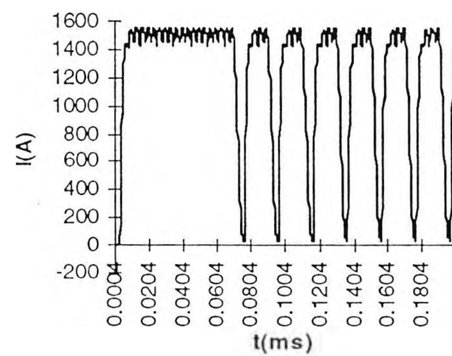
Double Succesive Commutation Failure



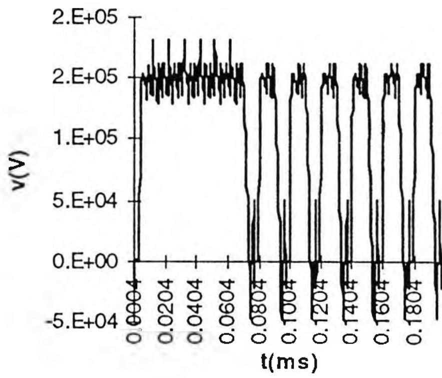
Single Repetive Commutation Failure



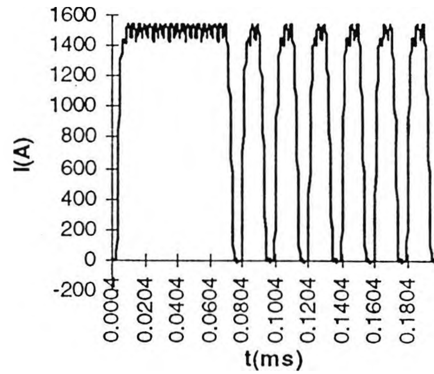
Single Repetive Commutation failure



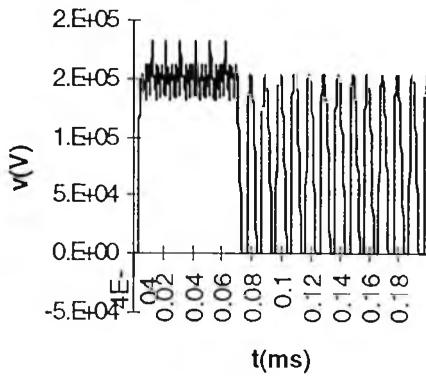
Double Succesive Repetitive Commutation Failure



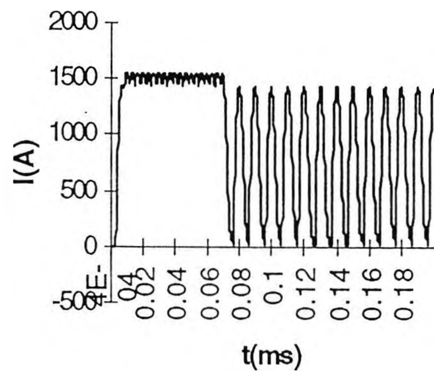
Double Succesive Repetitive Commutation Failure



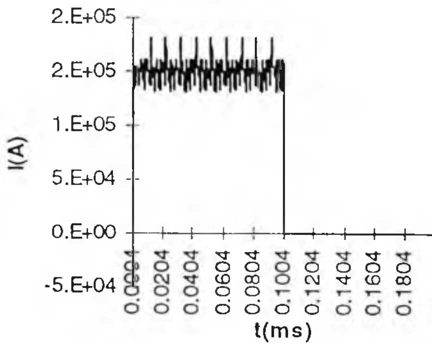
Double Not Succesive repetitive Commutation Failure



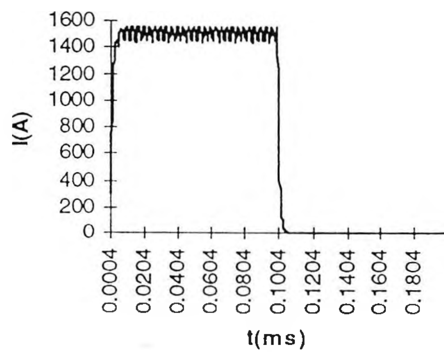
Double Not Succesive repetitive Commutation Failure

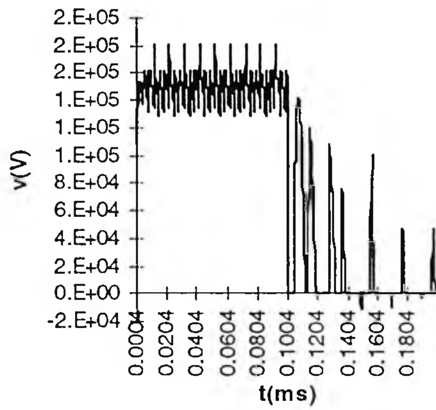


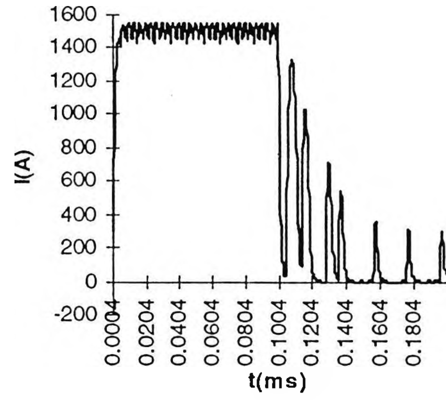
DC Fault

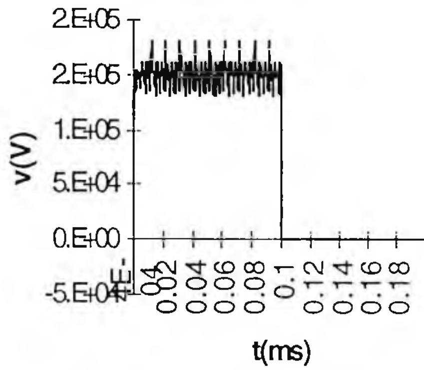


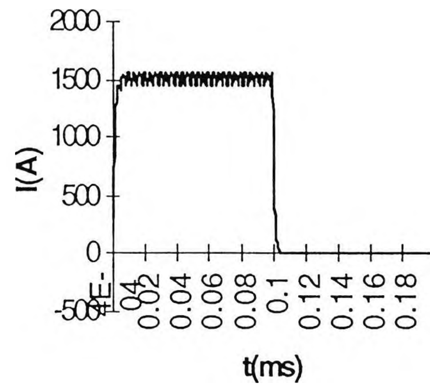
DC Fault

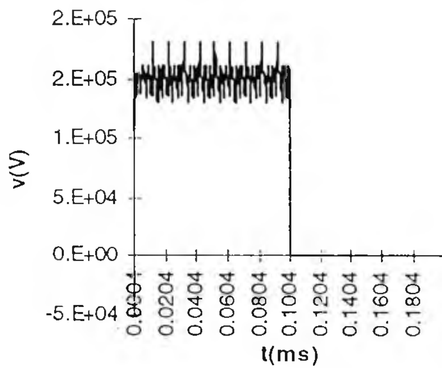












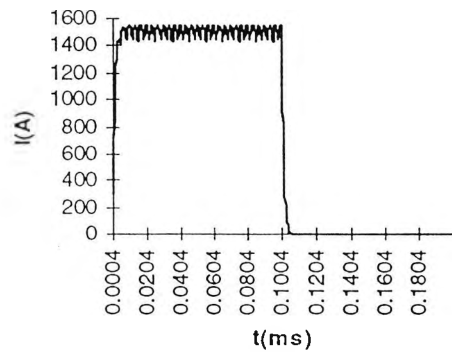


Figure 5.2-2: Voltage and Current Waveforms for the Different Fault Conditions

5.2.3 Data Pre-processing

The raw fault data obtained from EMTP simulation of the HVDC system consists of 500 samples for each fault type over a 0.2ms time interval. For a no fault condition, the whole 0.2ms interval is randomly sampled to select patterns for the training and test sets. For the other fault conditions, only samples in the immediate vicinity of the fault are of interest. This means that, for some fault cases, it is difficult to obtain enough samples to completely represent that case. To prevent a high occurrence of false alarms, data from the no fault or pre-fault data situation is given the highest representation in the training set sampled over the complete range of the simulation. The number of training patterns for the different fault types is severely limited by the sampling window during which fault data can be collected and by the resolution of the sampling interval. Table 5.2-1 shows the representation of each fault type in the training and test sets.

Table 5.2-1: Representation of each fault type in the training/test sets

Fault type	**Total**	**Training Set**	**Test Set**
No fault	500	80	22
Single commutation failure	20	9	8
Double successive commutation failure	46	13	11
Double not successive commutation failure	37	13	11
Single repetitive commutation failure	237	58	21
Double successive repetitive commutation failure	37	13	11
Double not successive repetitive failure	237	57	21
DC Fault	183	40	19
Single phase to ground fault	183	40	19
Double phase to ground fault	183	40	19
Three phase to ground fault	183	40	19
Total	1846	403	181

Using the random sampling approach mentioned above, it was then possible to generate a number of different but overlapping training and test sets which can provide a true picture of the generalisation capability of the network. Pre-processing takes the form of column-wise scaling of the data. For each column, the maximum and the minimum values are obtained. The data in the columns is then scaled according to the following formula:

$$\frac{x - \min}{\max - \min} \qquad (5.2\text{-}1)$$

so that individual features in each training vector is in the range $0\cdots1$ or $-1\cdots1$.

5.2.4 Fault Diagnostic Neural Network

The block diagram in Figure 5.2-3 shows the Neural Network incorporated into the HVDC system.

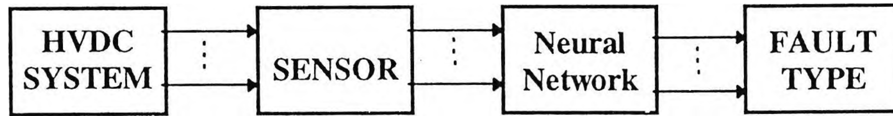


Figure 5.2-3: Block Diagram of Fault Diagnostic Neural Network System

The layout of the fault diagnostic Neural Network system to learn the mapping between the training patterns and the fault type is as shown in Figure 5.2-4.

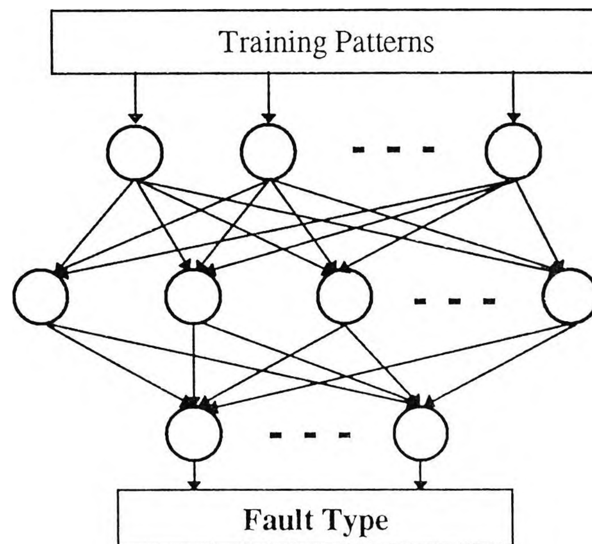


Figure 5.2-4: Neural Network learns mapping between training patterns and Fault Types

For training purposes, a single hidden layer neural network having 12 hidden neurons is used. There neural network has 20 input nodes and 4 output nodes. The input patterns consist of the following

- The voltage across and the current through each thyristor (9 nodes),
- The output dc voltage and current (2 nodes),
- The three phase voltages and currents through the transformer (6 nodes)
- The conduction pattern of the thyristors over one period (3 nodes).

The output of the network is a four bit binary number that represents the fault condition. The number of training iterations was set at 5000. The network was then trained for different learning rates and momentum factors.

5.2.5 Results

Training curves for two different sets of learning parameters are shown in Figure 5.2-5 and Figure 5.2-6 respectively. In the first run, 11 hidden neurons are used with a learning rate of 0.0175 and a momentum factor of 0.02. These values are changed to 21, 0.015 and 0.03 respectively in the second simulation run. As can be seen, the RMS error declines rapidly during both training sessions but the curve is much smoother when more neurons are used.

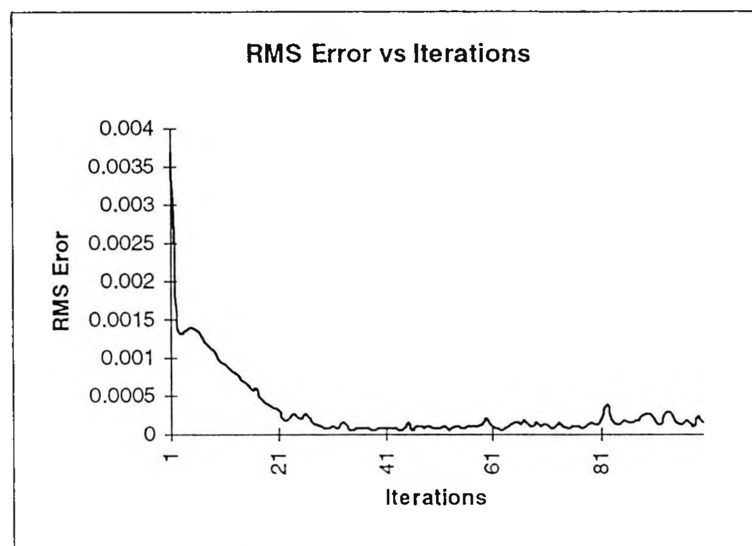


Figure 5.2-5: RMS training error, 11 hidden neurons, learning rate = 0.0175, momentum = 0.02

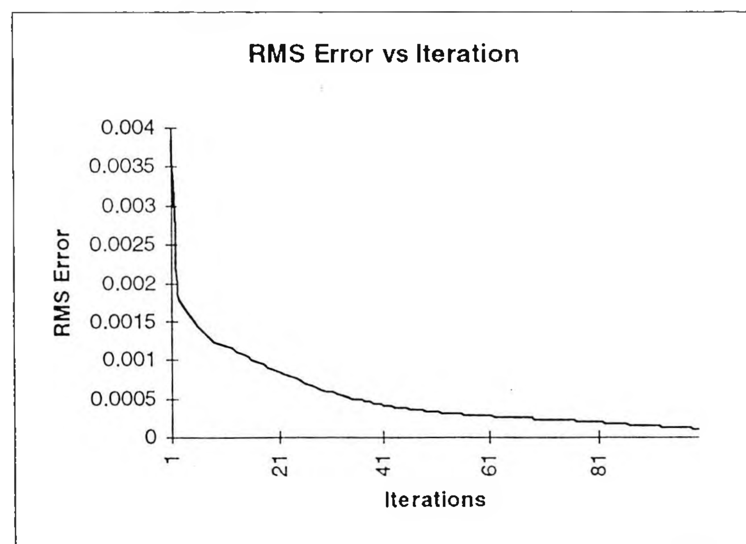


Figure 5.2-6: RMS training error, 21 hidden neurons, learning rate = 0.015, momentum = 0.03

The performance of the network is measured in terms of the percentage of misclassified patterns after a specified number of iterations. This performance index was almost always mirrored when independent testing was carried out using novel data in the test set. For test purposes, the output space was partitioned into 5 decision regions representing the percentage accuracy of the trained network for a single pattern. Properly classified patterns are those trained to within 10% of the desired value. Partially classified patterns are within 40% of their desired value. Boundary patterns are between 40 and 60% of their desired values. Partially and totally misclassified patterns are more than 60% and 90% respectively from their desired values. This partitioning was necessary to identify difficult learning patterns. By examining the partially classified patterns and those in the boundary, the patterns that needed further training could be identified. This helps concentrate training time on the non-learning or difficult learning patterns either by increasing their representation in the training set or by specialised training in a separate sub net. Table 5.2-2 shows the training results for the two simulation runs presented above. A sample training set consisting of 86 patterns as been used and training is done for 5000 iterations with the same learning rates.

Table 5.2-2: Training results

Decision boundary	Simulation run 1	Simulation run 2
Properly classified	23	25
Partially classified	56	57
Boundary	7	4
Partially misclassified	0	0
Totally misclassified	0	0

The results show that, in both cases, approximately 25% of the patterns are properly classified. The use of binary values in the outputs makes it difficult to attain output values which are close to the desired. This is due to the squashing nature of the sigmoidal activation function used. If a threshold element of 0.5 is used, then 100% classification is achieved in both cases.

5.2.6 Discussion

For this simulated system, the neural network could easily identify the system faults. The classification accuracy on system faults and the no fault case was always greater than 99%.

However, the converter faults could not be uniquely identified. The average classification accuracy on converter faults was less than 50%. The neural network could not learn the difference between the repetitive and non repetitive cases for the converter faults over one period. Three separate solutions were considered to overcome this problem. In the first instance, one of either the repetitive or non-repetitive cases was taken out of the training set. The network was then trained to maximum accuracy with rest of the training data. In this case, the network was able to correctly a identify a commutation failure. A separate classifier was then used to determine if it was a repetitive or non-repetitive fault. In the second instance, the sampling window from which the training data was extracted was increased to cover an extra period. The network was thus able to correctly classify both the repetitive and non-repetitive failure cases. The last case was to train the network on a slowly moving window of the time-stepped simulation data after pre-processing. While this method was also able to correctly identify the repetitive and non-repetitive commutation failures, the percentage error on the system fault and no fault cases was unacceptable. The method was thus discarded.

5.2.7 Future directions

Neural networks are fast enough for real time fault identification once they have been properly trained. The training set contains enough examples of normal operation to reduce the number of false alarms (i.e. no faults classified as faults) to a minimum. On the other hand the number of simulated fault conditions cannot necessarily cater for every fault that can be encountered in the field. The costs of on-line monitoring equipment and the associated difficulties mean that good fault data is difficult to come by. An inverse mapped or deductive neural network at the output of the fault classifying network can greatly reduce the chances of misclassification[93].

5.3 Application of Neural Networks to Systems Identification

5.3.1 Introduction

The previous case study described the application of neural networks to fault diagnosis in HVDC systems. In this application, the supervised neural network is treated as a black box system where training patterns are presented at the inputs, corresponding teaching signals are at the output and the network learns a mapping between them. In such an approach, a generic supervised learning architecture is configured with certain parameters for the number of input, hidden and output neurons, the activation function, and learning

parameters. The network weights are then initialised to small random values which is then adapted during learning to solve the problem. This is the approach used by most applications of neural networks where the problem to be solved is a specific case of generic static pattern recognition or simple optimisation. No *a priori* information about the problem or the process generating the data patterns can be used even where this is available to aid the neural network in arriving at the solution. For difficult learning problems, additional information can be incorporated in the design of the neural network to improve the chances of converging to an optimal solution. This leads to the concept of an application specific neural network, where the neural network is designed to solve a specific problem. Information about the problem domain is incorporated into the network learning algorithm to improve the chances of convergence to an optimal solution. This section presents an application specific supervised learning network for identifying the parameters of linear/non-linear dynamical systems. The dynamical system is described as a system of difference equations expressed in state variable format. The system is assumed to be time-invariant so that the state matrices are constant. For an arbitrary system where the input, current state and next state are known, the problem of identifying the system parameters is formulated as a neural network learning problem whose solution enables the system parameters to be identified. The network error and weight update equations are derived from empirical knowledge about the system configuration. The learning process is guaranteed to converge to a local minima of the error surface by the gradient descent algorithm. The rate of convergence is determined by the choice of learning parameters.

5.3.2 Problem Scope

In many engineering and scientific applications, a system or plant having an unknown structure has measurable or observable input and output signals. One way to obtain information about the plant dynamics is to simulate it by a possibly flexibly-structured model which will imitate the unknown dynamic system [25]. Matching a measurable plant and a specific model occurs by adaptively updating the model parameters so as to minimise a specified error/performance function. System identification is the process of selecting a model for an unknown dynamic system and estimating the model parameters from experimental data, i.e. from measurement of input and output data. The model is represented either as a transfer function block, a state variable representation or a state-vector representation [97].

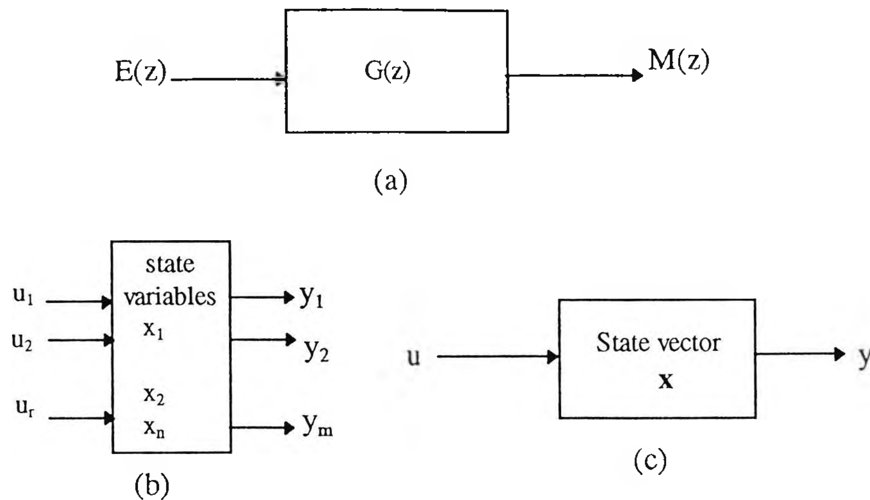


Figure 5.3-1: Representations of system dynamics: (a) z-transfer function representation; (b) state-variable representation; (c) state-vector representation.

5.3.3 Neural Network for Systems Identification

In the design of control systems where measurement of the full state vector is not practical, the states of the system have to be estimated using practical measurements. System identification techniques are thus widely used in control systems design. A number of papers have been published that discuss the use of neural networks in systems identification. In [97] a binary neural network is used to identify the discrete inverse dynamics of an unknown plant. The neural network is configured to operate in two separate modes: The first is an adaptation/learning mode where the weights of the network are configured to learn a mapping between the input to the plant and its measured response. The second is the controller mode where the neural network acts as an inverse controller of the plant. The desired response of the plant is fed as input to the neural network and the output is a signal used to drive the output of the plant to its desired value. The simulation results showed that neural networks are capable of learning an optimal mapping even when the dynamical systems are non-linear. In [98] a dynamic neural network is employed in the identification and control of static and dynamical systems. Both state-vector and difference equation representations of the system have been used. The identification problem is formulated as a suitably parameterised model whose parameters are adjusted to minimise an error function between the plant to be identified and the model outputs. In [99] a backpropagation neural network is trained to produce the unknown parameters of the system to be identified at its output. In a similar manner to [98], a parameterised model of the plant is used. The neural network is then

configured to estimate the unknown parameters of the parameterised model from which the real parameters are derived. In the field of power systems, [100] presents an approach to estimate parameters of synchronous machines using online small disturbance response data. Their estimation is also based on the minimisation of a cost function of the estimation error but a Maximum Likelihood rather than a gradient descent algorithm was adopted for the optimisation.

The approach discussed in this thesis for identifying the parameters of dynamical systems follows from the work described in [101]. In general, the equations that describe the state of a dynamical system at any time k+1 is given by the single valued functional relationship [97]

$$\mathbf{x}(k+1) = \mathbf{f}[\mathbf{x}(k), \mathbf{u}(k)] \tag{5.3-1}$$

The output response of the system is given by

$$\mathbf{y}(k) = \mathbf{g}[\mathbf{x}(k), \mathbf{u}(k)] \tag{5.3-2}$$

where $\mathbf{u}$ is the set of values that in input vector may assume as a function of time,

$\mathbf{y}$ is the output vector and

$\mathbf{x}$ is the state vector.

If the system is linear, the equations can be written as

$$\dot{\mathbf{x}}(k) = \mathbf{A}\,\mathbf{x}(k) + \mathbf{B}\,u(k) \tag{5.3-3}$$

$$\mathbf{y}(k) = \mathbf{C}\,x(k) + \mathbf{D}\,u(k) \tag{5.3-4}$$

where $\mathbf{x}(k) \in \Re^{n}$, $u(k) \in \Re$ and $\mathbf{y}(k) \in \Re^{m}$.

$\mathbf{A} \in \Re^{nxn}$

$\mathbf{B} \in \Re^{n}$

$\mathbf{C} \in \Re^{rxn}$

$\mathbf{D} \in \Re^{r}$

The next state description of the system in Equation (5.3-3) can be expressed in matrix form as

$$\begin{bmatrix} \dot{x}_1(k) \\ \vdots \\ \dot{x}_n(k) \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1(k) \\ x_2(k) \\ \vdots \\ x_n(k) \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} u(k) \tag{5.3-5}$$

The aim of the neural network learning is to determine the values of a_{ij} and b_j of the unknown system with given value of state vector $\mathbf{x}(k)$, next state vector $\dot{\mathbf{x}}(k)$ and input $u(k)$. The approach, suggested in [101] is used to derive the network performance (error) function and the weight update equations. The error function is formulated in such a way that it is quadratic in terms of the parameters to be estimated. The neural network dynamical equations are as follows:

$$\begin{bmatrix} \dot{x}_{e1}(k) \\ \vdots \\ \dot{x}_{en}(k) \end{bmatrix} = \begin{bmatrix} a_{e11} & a_{e12} & \cdots & a_{e1n} \\ a_{e21} & a_{e22} & \cdots & a_{e2n} \\ \vdots & & \ddots & \vdots \\ a_{en1} & a_{en2} & \cdots & a_{enn} \end{bmatrix} \begin{bmatrix} x_{e1}(k) \\ x_{e2}(k) \\ \vdots \\ x_{en}(k) \end{bmatrix} + \begin{bmatrix} b_{e1} \\ b_{e2} \\ \vdots \\ b_{en} \end{bmatrix} u(k) \qquad (5.3\text{-}6)$$

where a_{eij} and b_{ej} are the estimated values of the unknown parameters. The error in the estimation is given by

$$\mathbf{e}(k) = \mathbf{x}(k) - \mathbf{x}_e(k) \qquad (5.3\text{-}7)$$

The derivative of the error is given by

$$\begin{bmatrix} \dot{e}_1(k) \\ \dot{e}_2(k) \\ \vdots \\ \dot{e}_n(k) \end{bmatrix} = \begin{bmatrix} \dot{x}_1(k) \\ \dot{x}_2(k) \\ \vdots \\ \dot{x}_n(k) \end{bmatrix} - \begin{bmatrix} \dot{x}_{e1}(k) \\ \dot{x}_{e2}(k) \\ \vdots \\ \dot{x}_{en}(k) \end{bmatrix} \qquad (5.3\text{-}8)$$

By substituting for $\dot{\mathbf{x}}_e(k)$ from (5.3-6) into (5.3-8), the error can be written as

$$\begin{bmatrix} \dot{e}_1(k) \\ \dot{e}_2(k) \\ \vdots \\ \dot{e}_n(k) \end{bmatrix} = \begin{bmatrix} \dot{x}_1(k) \\ \dot{x}_2(k) \\ \vdots \\ \dot{x}_n(k) \end{bmatrix} - \begin{bmatrix} a_{e11} & a_{e12} & \cdots & a_{e1n} \\ a_{e21} & a_{e22} & \cdots & a_{e2n} \\ \vdots & & \ddots & \vdots \\ a_{en1} & a_{en2} & \cdots & a_{enn} \end{bmatrix} \begin{bmatrix} x_1(k) \\ x_2(k) \\ \vdots \\ x_n(k) \end{bmatrix} - \begin{bmatrix} b_{e1} \\ b_{e2} \\ \vdots \\ b_{en} \end{bmatrix} u(k) \qquad (5.3\text{-}9)$$

The ith component of the derivative of the error is

$$\dot{e}_i(k) = \dot{x}_i(k) - (a_{ei1}x_1(k) + a_{ei2}x_2(k) + \cdots + a_{ein}x_n + b_{ei}u(k)) \qquad (5.3\text{-}10)$$

This error component can be expressed in more compact notation as

$$\dot{e}_i = \dot{x}_i - \sum_{j=1}^{n} a_{eij} x_j(k) + b_{ei} u(k) \qquad (5.3\text{-}11)$$

The above equation can equally be expressed as

$$\dot{e}_i = \dot{x}_i - \mathbf{w}_i^T \mathbf{z} \qquad (5.3\text{-}12)$$

where $\mathbf{w}_i^T = [a_{ei1}, a_{ei2}, \cdots, a_{ein}, b_{ei}]$ and

$\mathbf{z}^T = [x_1, x_2, \cdots, x_n, u]$

This equation is analogous to the general supervised neural network processing equation where $\dot{x}_i$ is the ith target value, $\mathbf{w}_i^T$ is the weight of the ith neuron, with $\mathbf{z}$ as its input. The objective or energy function can be formulated as the sum-of-squares error criterion

$$\mathbf{E}_p = \frac{1}{2} \sum_{i=1}^{n} \left(\dot{e}_i \right)^2 \qquad (5.3\text{-}13)$$

where $\mathbf{E}_p$ is the error per pattern presented. The total error of the network after all patterns have been presented is simply the sum of the individual pattern errors:

$$\mathbf{E} = \sum_{p=1}^{P} \mathbf{E}_p \qquad (5.3\text{-}14)$$

substituting for $\dot{e}_i$ in the equation (5.3-13) above, we have

$$\mathbf{E}_p = \frac{1}{2} \sum_{i=1}^{n} \left(\dot{x}_i - \mathbf{w}_i^T \mathbf{z} \right)^2 \qquad (5.3\text{-}15)$$

which is simply a quadratic error measure that can be minimised by gradient descent. The minimisation of $\mathbf{E}_p$ with respect to time enables the unknown parameters of $\mathbf{w}_i^T = [a_{ei1}, a_{ei2}, \cdots, a_{ein}, b_{ei}]$ to be determined as the final weights of the trained neural network. Figure 5.3-2 shows the architecture of the neural network and how the parameters are determined after training.

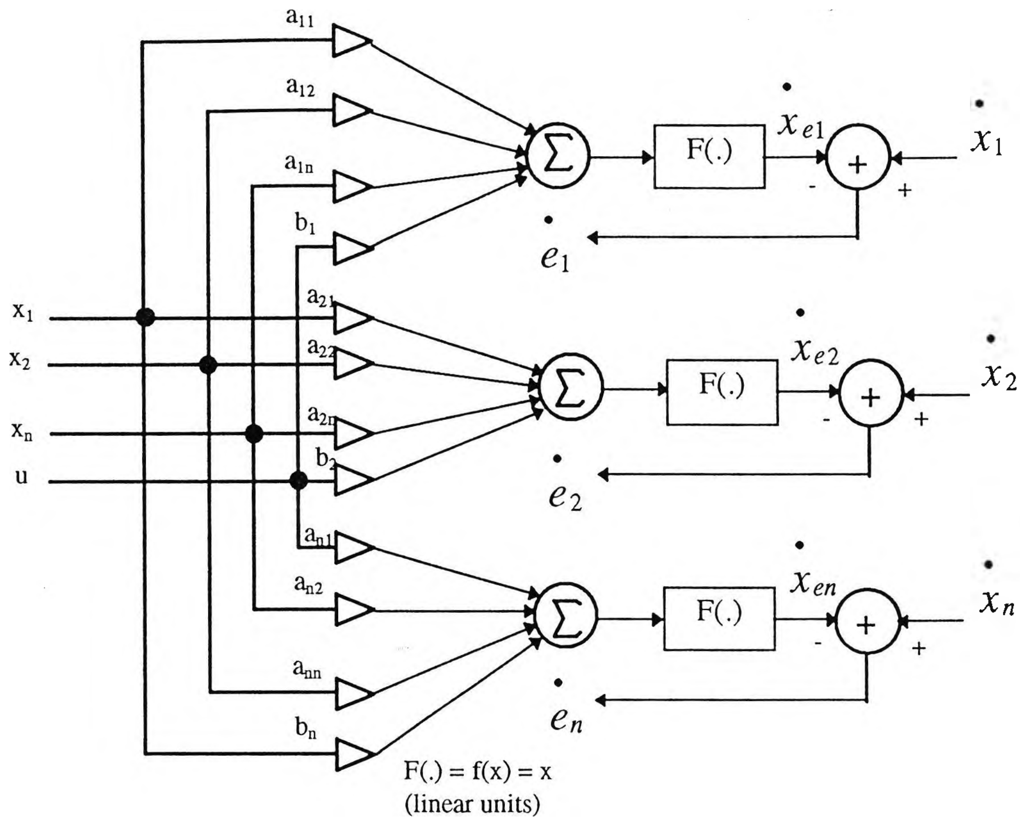


Figure 5.3-2: Architecture of Supervised Neural Network for Identification

The quadratic error measure in Equation (5.3-15) can be optimised by first differentiating and then setting the gradient equal to zero. The gradient of the error function w.r.t the weight, $\mathbf{w}_i$ is

$$\nabla_{\mathbf{w}_i} \mathbf{E}_p = \frac{\partial \mathbf{E}_p}{\partial \mathbf{w}_i} \tag{5.3-16}$$

Under this condition, the values obtained for the unknown parameters will be optimal. Adopting first-order gradient descent, the weight change in each iteration is given by

$$\Delta \mathbf{w}_i = -\eta \nabla_{\mathbf{w}_i} \mathbf{E}_p = -\eta \frac{\partial \mathbf{E}_p}{\partial \mathbf{w}_i} \tag{5.3-17}$$

where η is the learning rate or size of step to be taken in the direction of the negative gradient. The weight update equation for each learning iteration is given by

$$\mathbf{w}_i(t+1) = \mathbf{w}_i(t) + \Delta \mathbf{w}_i \tag{5.3-18}$$

5.3.4 Simulation Results

This section presents the results of applying the parameter estimation neural network to determining the parameters of an unknown second order linear dynamical system. To be able to do this, values for the state vector $\mathbf{x}(k)$, the derivative of the state vector $\dot{\mathbf{x}}(k)$

and the input, $u(k)$, must be known. The neural network was tested using a number of different second-order systems. The following is an example of a second-order dynamical system used as an example:

$$\begin{bmatrix} \dot{x}_1(k) \\ \dot{x}_2(k) \end{bmatrix} = \begin{bmatrix} -0.9420 & 12.566 \\ -12.566 & -0.942 \end{bmatrix} \begin{bmatrix} x_1(k) \\ x_2(k) \end{bmatrix} + \begin{bmatrix} 1.0 \\ 2.0 \end{bmatrix} u(k) \qquad (5.3\text{-}19)$$

To simplify the processing, a step input signal is used. The state vector and derivative of state vector are obtained by simulation. The system is expressed in the form of a difference equation and a C++ program written to recursively compute the values of the state vectors for a specified number of iterations. Figure 5.3-3 shows the state vector plots of the dynamical system obtained by the C++ simulation.

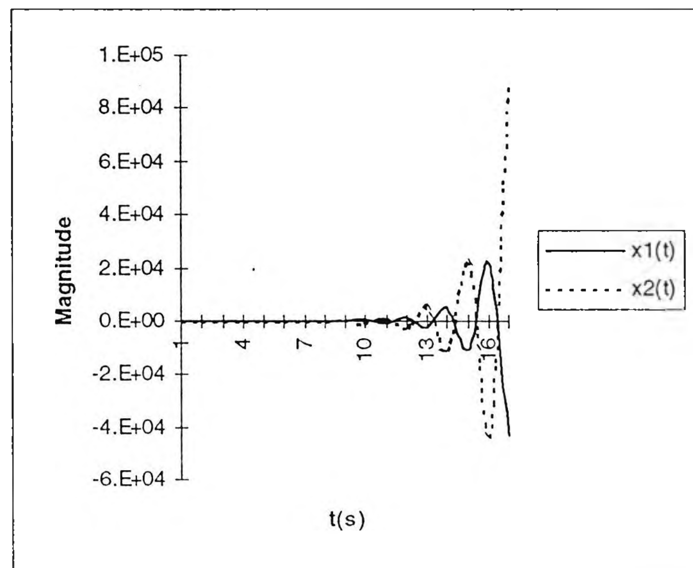


Figure 5.3-3: Plot of state values of the dynamical system

Equation 5.3-19 was modified to a similar second order dynamical system by changing the values for the parameters as follows:

$$\begin{bmatrix} \dot{x}_1(k) \\ \dot{x}_2(k) \end{bmatrix} = \begin{bmatrix} 0.0 & 1.0 \\ -0.72 & 1.7 \end{bmatrix} \begin{bmatrix} x_1(k) \\ x_2(k) \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u(k) \qquad (5.3\text{-}20)$$

Figure 5.3-4 shows a plot of the state vectors for the modified system.

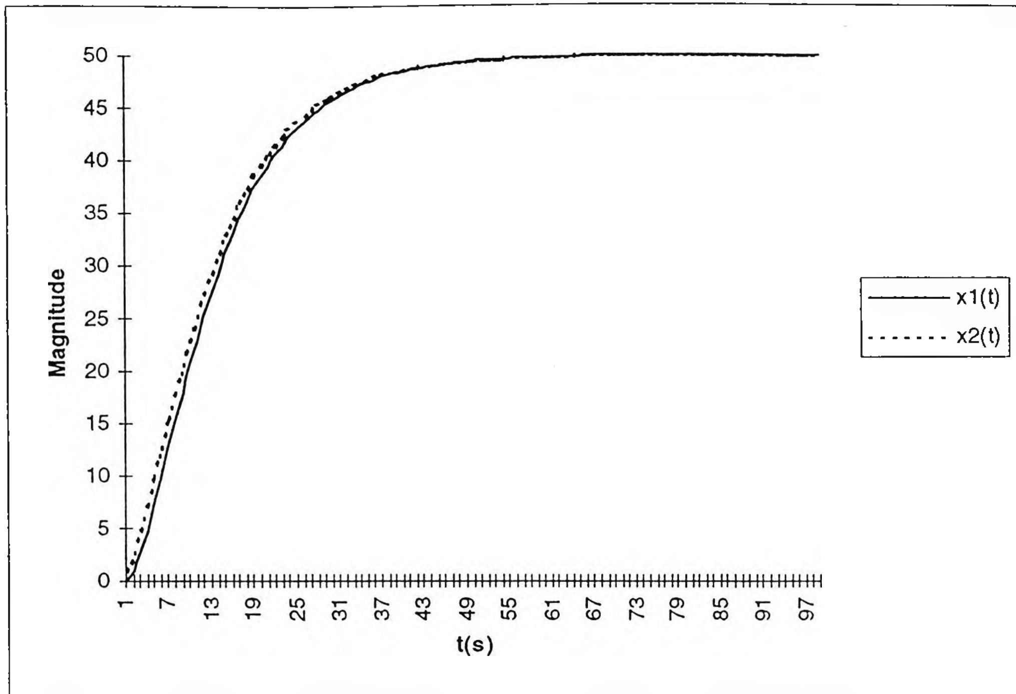


Figure 5.3-4: Plot of state vectors for dynamical system

Supervised training data is obtained by considering state vectors at time step *k* as the input vectors and state vectors at time step *k+1* as the corresponding target vectors. A total of 1000 training patterns was used for the training process. Figure 5.3-5 shows an extract from the training file. A single layer neural network with 2 inputs, 2 outputs and a learnable bias was constructed to train the samples.

```
Number of patterns: 1000
Number of inputs: 2
Number of outputs: 2
0 0                       0.00101344 0.00199187 1
0.00101344 0.00199187     0.00204947 0.00396984 2
0.00204947 0.00396984     0.00310928 0.0059328   3
0.00310928 0.0059328      0.00419266 0.00788049 4
0.00419266 0.00788049     0.00529938 0.00981259 5
•
•
•
0.0991016 -0.0434478      0.0994671 -0.0426549 998
0.0994671 -0.0426549      0.0998421 -0.0418673 999
0.0998421 -0.0418673      0.100227 -0.0410852  1000
```

Figure 5.3-5: Extract from training data

The network was configured with Linear units and training carried out, then the activation function was changed to Sigmoid functions. Figure 5.3-6 and Figure 5.3-7 show the training curves for linear and Sigmoid activation functions respectively.

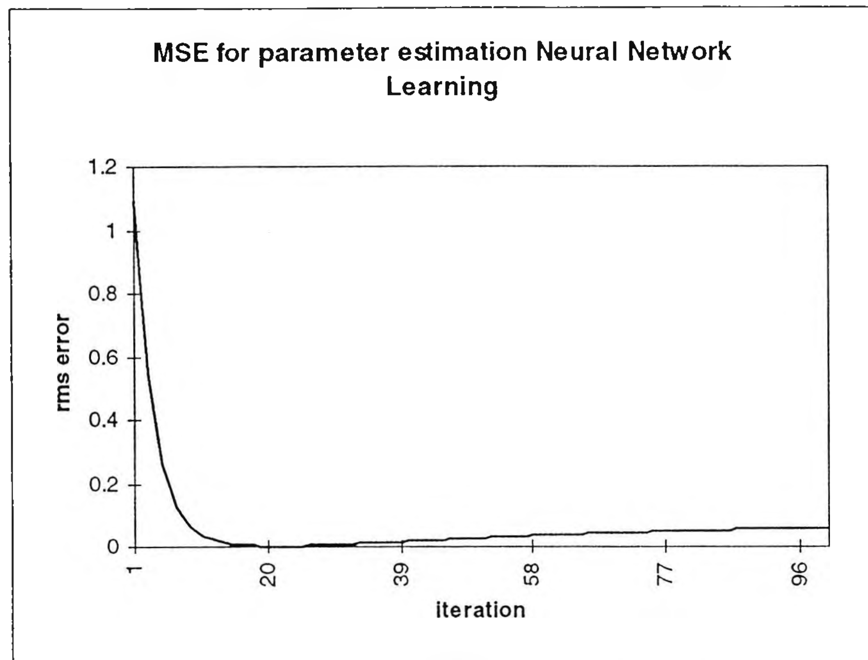


Figure 5.3-6: Training curve using linear activation functions (learning rate = 0.15)

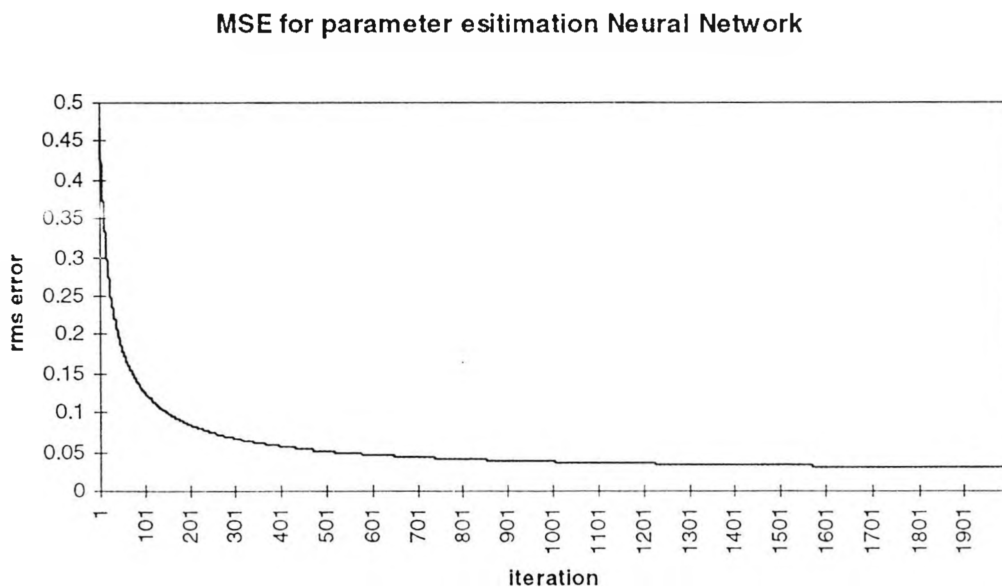


Figure 5.3-7: Training curve using Sigmoid activation functions (learning rate = 0.15)

Preliminary training results shows that the learning is much faster with Linear activation functions proposed in the learning equations. Learning still takes place when non-linear

units with Sigmoid activation functions are used but learning process is slow and erratic and some of the weights often fail to converge to the optimal parameters values.

5.4 Discussion and Conclusions

The chapter has demonstrated the successful application of artificial neural networks to two practical problems in the field of power systems. In the first case study, the neural network was able to learn the required mapping between input fault data and a binary fault identifier. The use of neural networks in fault identification has comparative advantages over fault diagnostic expert systems in terms of the speed at which they can operate. A pre-trained neural network requires just a single forward pass through the network during production in order to identify a fault. Fault diagnostic expert systems, on the other hand, perform the identification based on the firing of pre-stored rules in a rulebase. For a particular input, the number of rules that fire and the order in which the rules fire cannot be predetermined. This makes artificial neural networks more suitable for real-time problems such as fault diagnosis in power systems.

In the second case study, an application specific neural network has been developed for identifying unknown parameters in linear and non-linear dynamical systems. The network uses *a priori* information about the problem to derive the neural processing and weight update equations. In this particular case, the *a priori* information is in the form of the next state equations that describe a state space representation of the system whose parameters are to be identified. Preliminary results have shown that the neural network is capable of converging to the correct parameters for a stable second order system.

Chapter VI

FUZZY LOGIC

6.1 Introduction

The previous chapters have describe the design, implementation and application of Artificial Neural Networks. Neural networks are often construed as black boxes which can store and retrieve associations or learn a mapping between an input vector space and an output space. Where data required to train the network is imprecise, incomplete or uncertain, the implicit rules that describe the relationship between the input space and the output space become very complicated and the neural network is usually unable to learn the correct mapping or correct set of rules to represent this relationship. In recent years, the focus of much research has moved to the design of systems which can not only cope with uncertainty but that can explicitly model the uncertainty and imprecision that is inherent in real world problems. Such systems include fuzzy systems, probabilistic reasoning systems, possibility theory, belief networks, etc.

Fuzzy systems like their neural network counterparts are still encapsulations of low level algorithms. Despite their widespread use in consumer and industrial systems, the majority of software-based fuzzy systems exists only as program code in class libraries [102]. Fuzzy class libraries are useful as a means of promoting code reuse and also facilitates large scale adoption of fuzzy logic technology. On the other hand, code in class libraries constrains new applications to a particular design architecture and implementation language. There are also significant overheads associated with learning class libraries as well as run-time and performance penalties incurred due to their generality. This chapter describes how analysis and design can be carried out using OMT to construct a robust architecture for a fuzzy inference system. A number of sample implementations have been done using the C++ programming language. Finally, an inference system is developed to predict harmonics in switched-capacitor AC-DC converter systems where the mapping is very non-linear and the rules are initially unknown. A method for obtaining the required Rules in application areas where an expert or real world experimental data is not readily available is described.

6.2 Fuzzy Logic Theory

Lotfi Zadeh [103] defines fuzzy logic to be

> "…a Logical system which is aimed at providing a model for modes of reasoning which are approximate rather than exact…",
>
> "…Fuzzy Logic = Fuzzy Set Theory, the theory of classes with fuzzy rather than sharp boundaries…".

Fuzzy logic is a superset of conventional logic that has been extended to handle the concept of partial truths, i.e., truth values between "completely true" and "completely false". Fuzzy logic is used to provide robustness and ease of use in large complex systems where high precision is not a prerequisite.

6.2.1 Fuzzy Subsets

Associated with fuzzy logic is the concept of a fuzzy subset. Fuzzy subsets are functions that map a number or value that might be a member of a set to a number between 0 and 1 which indicates its actual degree of membership in the set. A membership value of zero indicates that the number is not in the set while a membership value of one means that the value is completely representative of the set. The Fuzzy Logic FAQ (frequently asked questions) @ ftp.cs.cmu.edu gives the following formal definition for a fuzzy subset.

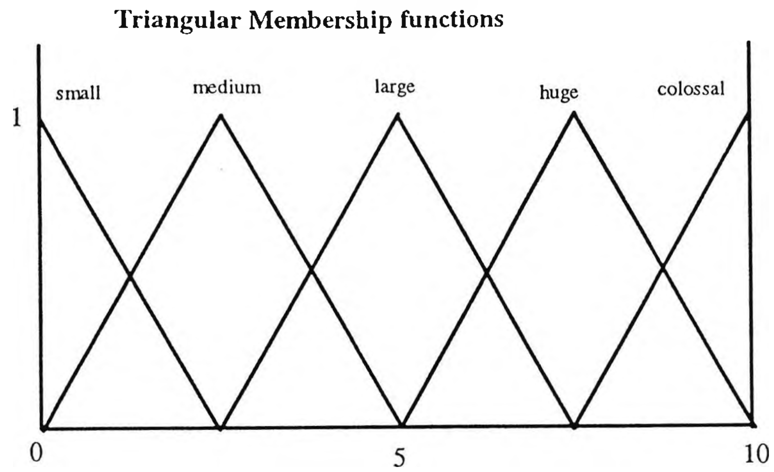
Formal definition

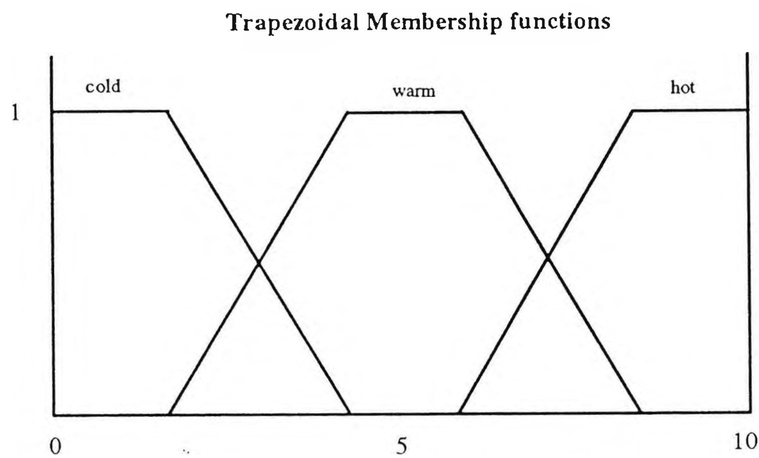
A fuzzy subset F, of a set S is defined as the set of ordered pairs, each with the first element from S and the second element from the interval [0,1], with exactly one ordered pair present for each element of S. This defines a mapping between elements of the set S and values in the interval [0,1]. A zero value is used to represent complete non-membership while a one value represents complete membership. Values between 0 and 1 are used to represent intermediate degrees of membership. The set S is referred to as the universe of discourse of the fuzzy subset F. The mapping is described as a function which is the membership function of F. The degree to which the statement

x is in F

*is true is determined by finding the ordered pair whose first element is **x**. The degree of truth of the statement is the second element of the ordered pair.*

Fuzzy membership functions could be triangular, trapezoidal or Gaussian in shape as shown in Figure 6.2-1.





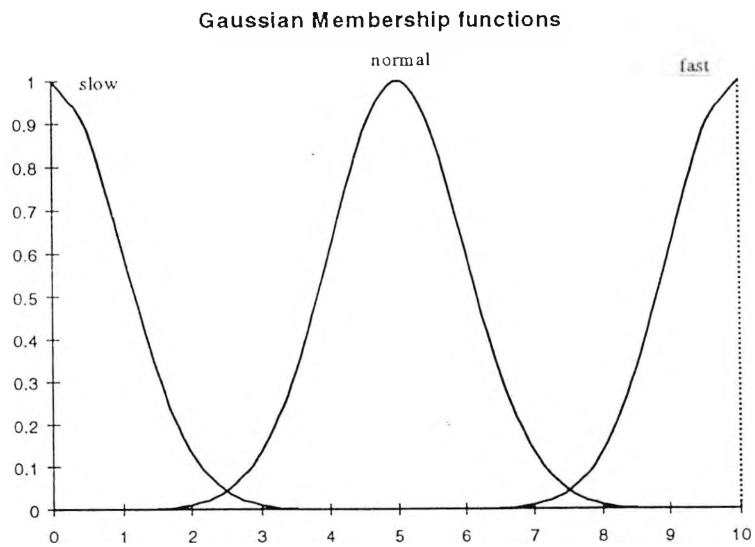


Figure 6.2-1: Fuzzy subsets with triangular, trapezoidal and Gaussian membership functions respectively

Triangular and Trapezoidal membership functions are more widely because they are mathematically easier to process. Gaussian membership functions, on the other hand, are much preferred due to their local properties and smoother transitions which in theory should provide better performance in a similar manner to Gaussian units Radial Basis function neural networks. The membership functions in Figure 6.2-1 could be used to represent a number of real world input variables such as salaries, temperature, and pulse rate, etc. It is necessary to determine universe of discourse in of a fuzzy set in order for the fuzzy variables to be meaningful. For instance given a fuzzy variable, salary, and a given value of salary, the value cannot described as Small, Medium, Colossal etc. unless a range of values which salary can take are know *a priori*. The set of all values that the fuzzy variable, salary can take determines its universe of discourse.

6.2.2 Linguistic Variables

Fuzzy logic provides support for the concept of a linguistic variable. Linguistic variables provide a way in which natural language expressions are can be handled mathematically using fuzzy logic. Examples of Linguistic variables include: TALL, HIGH, COOL, SMALL. Fuzzy linguistic variables are associated with a corresponding membership function. For example, the linguistic variable TALL which describes the universe of TALL people has an associated membership function which has been derived based on a person's height. To each person in the universe of discourse, a degree of membership in the fuzzy subset TALL can be assigned based on the person's height.

$$\text{TALL}(x) = \begin{cases} 0, & \text{if height}(x) < 5\text{ft.} \\ \dfrac{\text{height}(x) - 5\text{ft.}}{2\text{ft.}}, & \text{if } 5\text{ft.} < \text{height}(x) \text{ and } \leq 7\text{ft.} \\ 1, & \text{if height}(x) > 7\text{ft.} \end{cases} \quad (6.2\text{-}1)$$

Figure 6.2-2 shows a graph of the membership function associated with the linguistic variable of TALL. An expression such as "Person X is TALL" can be interpreted as a degree of truth which depends on the person's height. The shape of the membership function associated with a linguistic variable can be modified using linguistic hedges such as VERY, ALMOST, QUITE, EXTREMELY, etc. The effect of a hedge is tune the membership function respond properly to a given range of inputs. For instance, the rule "IF X is TALL then..." is less restrictive than "IF X is VERY TALL then..." or "IF X is EXTREMELY TALL..." but more restrictive than "IF X is QUITE TALL" and will respond slightly differently for similar values of height.

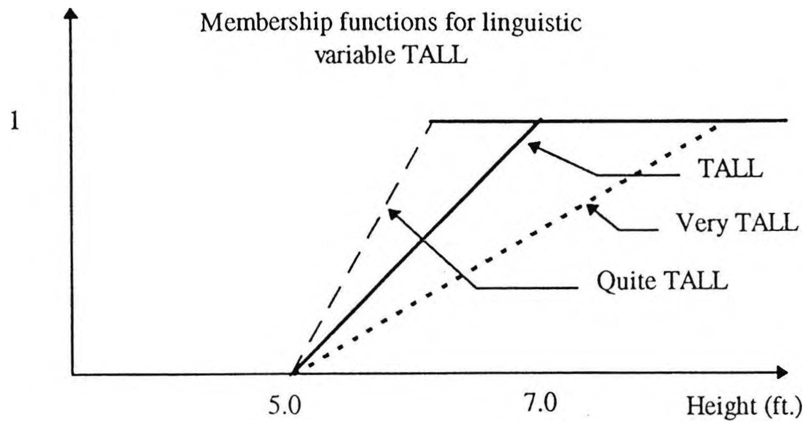


Figure 6.2-2: Membership functions associated with TALL

6.2.3 Fuzzy Numbers

Fuzzy Numbers are fuzzy subsets of the real line. They have a peak or plateau where the grade of membership equals 1. The membership function falls away on both sides of the peak. The common types of fuzzy numbers are triangular, Gaussian and L-R fuzzy numbers. Both triangular and Gaussian fuzzy numbers are symmetrical around a mean value as shown in Figure 6.2-3. They are more specialised and therefor computationally easier to process.

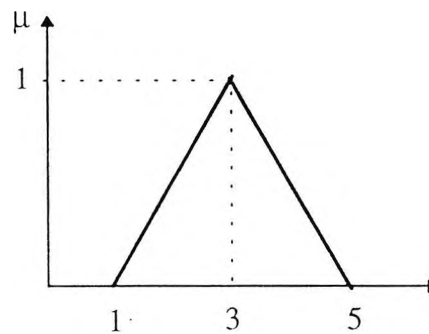


Figure 6.2-3: Triangular fuzzy number 3

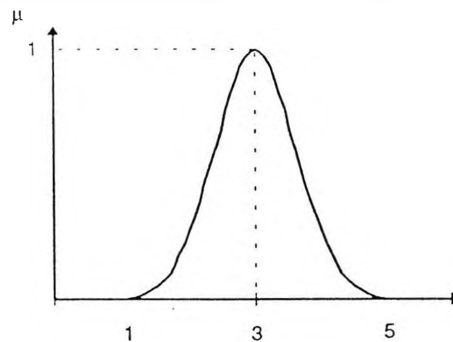


Figure 6.2-4: Gaussian Fuzzy number 3

L-R fuzzy numbers are more general. They consist of a mean value as well as left and right spread functions. Figure 6.2-5 gives an illustration of an L-R fuzzy number. An L-R fuzzy number M, written $(m,\alpha,\beta)_{LR}$, is given by

$$\mu_M(x) = \begin{cases} L\left(\frac{m-x}{\alpha}\right) & x \le m, \quad \alpha > 0 \\ R\left(\frac{x-m}{\beta}\right) & x \ge m, \quad \beta > 0 \end{cases} \tag{6.2-2}$$

where m is the mean value,

α and β are the left and right spreads respectively and

$L(.)$ and R(.) are left and right shaped functions.

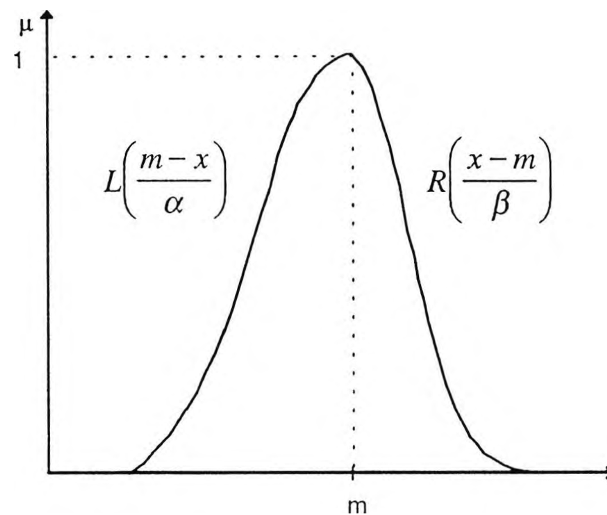


Figure 6.2-5: An L-R fuzzy number M with value m

When the left and right spreads, α and β, are zero, the L-R fuzzy number is reduced to an ordinary number, m, which is the mean value.

6.2.4 Fuzzy Inference

There are two generally recognised methods of fuzzy reasoning[104]:

- reasoning based on compositional rules of inference.
- reasoning based on fuzzy logic.

A fuzzy inference system uses a collection of fuzzy *if...then* rules (fuzzy rulebase) to reason about data. For the sake of computational simplicity, most fuzzy inference systems use the first form of reasoning, i.e. based on computational rules of inference. The structure of a simple fuzzy inference system is shown in Figure 6.2-6.

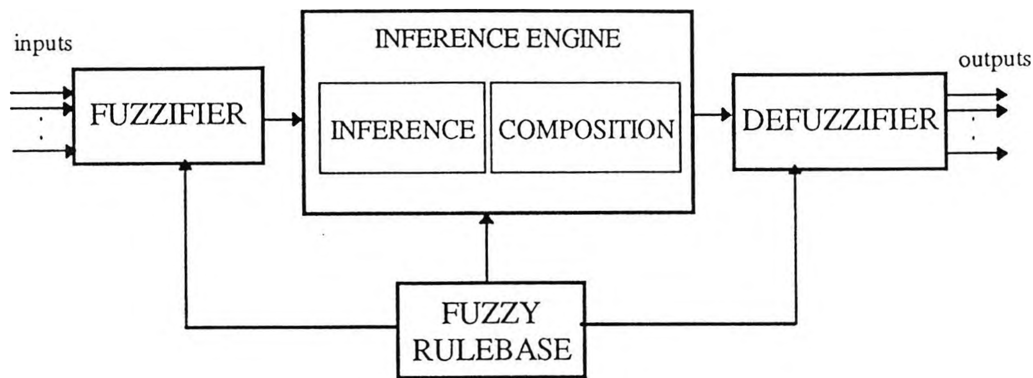


Figure 6.2-6: Structure of a Fuzzy Inference System

The fuzzy rules are of the form:

If Premise (Antecedent) *then* Conclusion (Consequent).

For example, in the following rule:

If x is A AND y is B *then* z is C,

the Premise is "x is A AND y is B" and the Conclusion is "z is C"; x and y are the input variables, z is the output variable and A, B and C are fuzzy subsets or membership functions defined on x, y and z respectively. The rule's premise describes to what degree the rule applies while the conclusion assigns a membership function to each of one or more output variables. The set of rules in the inference system forms the Rulebase. The fuzzy inference process generally consists of the following four steps: Fuzzification, Inference, Composition and Deffuzification[105].

During Fuzzification, the membership functions defined on the input variables are applied on their actual values to determine the degree of truth for each premise.

In the inference process, the truth value for the premise of each rule (the weight) is computed and applied to the conclusion part of each rule. Either MIN or PRODUCT inference is used. When MIN inference is used, the output membership functions are clipped at a height which corresponds to the weight of the rule's premise. When PRODUCT inference is used, the weight of the rule's premise is used to scale down the output membership functions.

During composition, all the fuzzy subsets assigned to each output variable are combined together to form a single fuzzy subset for the output variable. As with the inference stage, there are two possible methods of composition; MAX composition and SUM composition. MAX composition constructs the output fuzzy subset by taking the point-

wise maximum over all the fuzzy subsets assigned to a variable by the inference rule. SUM composition, on the other hand, uses the point-wise sum of all the fuzzy subsets assigned to a given variable to construct the output fuzzy subset.

The final defuzzification stage converts the output fuzzy subset into a crisp (non-fuzzy) value or number. Several different methods of defuzzification exist in the literature. The two most common include: Centre Of Gravity (COG) and Mean of Maxima (MOM). COG defuzzification computes the crisp value of a fuzzy subset by finding the centre of gravity of the membership function. In MOM defuzzification, the crisp value is obtained by calculating the mean value at all points where the membership function has it's highest value. Other defuzzification methods include Semi-Linear Defuzzification (SLIDE) and Basic Defuzzification Distribution (BADD) transformations [106].

Figure 6.2-7 and Figure 6.2-8 illustrates the process of fuzzy reasoning based on compositional rules of inference. For a system with 2 fuzzy *if...then* rules [104, 107], the rules are of the form:

if x_1 is A_{11} and x_2 is A_{12} *then* y is B_1;

if x_1 is A_{21} and X_2 is A_{22} *then* y is B_2.

Where x_1 and x_2 are the inputs,

y is the output and

A_{11}, A_{12}, A_{21}, A_{22}, B_1, B_2 are the fuzzy membership functions.

In Figure 6.2-7 the reasoning process uses a combination of Product inference with Sum composition while in Figure 6.2-8 Min inference coupled with Max composition is illustrated.

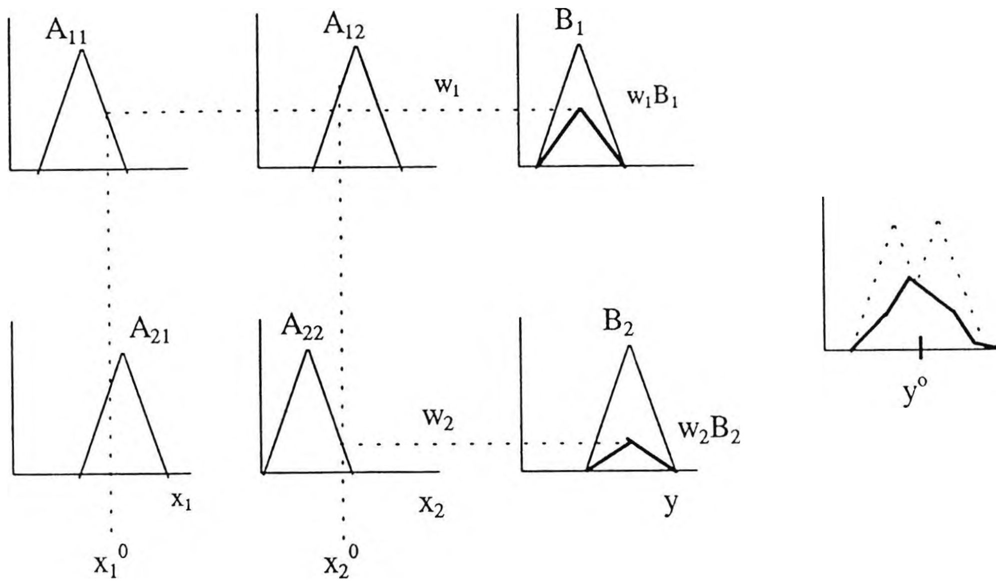


Figure 6.2-7: Fuzzy reasoning: Product Inference with Sum Composition

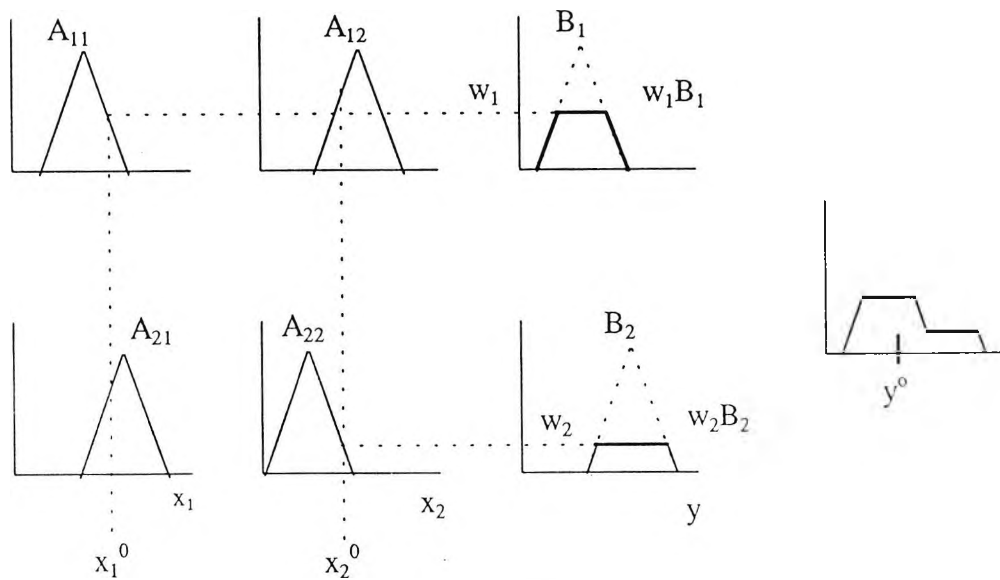


Figure 6.2-8: Fuzzy reasoning: Min Inference with Max Composition

6.2.5 Constructing the Rulebase

The most important activity involved in the design of a fuzzy inference system is the selection of rules that make up the Rulebase. There are the following four general methods available for deriving the rules [104, 108].:

1. Experts knowledge and advice: The format of fuzzy rules make them suitable as a descriptive language to express an experts domain knowledge and thinking. The fuzzy *if...then* rules can be formulated by two heuristic approaches. The first involves the

explicit verbalisation of human experience by a domain expert. The second approach involves a careful interrogation of a pool of domain experts and then subsequently processing their responses to obtain the relevant rules for a particular application.

2. Modelling an operator's action: In most complex natural and man-made systems the exact input-output relationships are not known and so it is extremely difficult to create mathematical or quantitative models to describe their operation. Skilled human operators have learned to control such systems using heuristics and experience. Their actions and operations can be modelled by fuzzy *if...then* rules which form the Rulebase for reasoning about and subsequently controlling these systems.

3. Modelling a process: A linguistic description of the dynamic characteristics of a controlled process may be viewed as a fuzzy model of the process. The characteristics of process and the necessary variables required for it's control can be modelled by simulation. Based on the model, a set of fuzzy *if...then* rules can be generated for reasoning about or controlling the process.

4. Learning and adaptation: These involve the automatic generation or learning of fuzzy *if...then* from data sets. Neural networks can be used to learn a process from examples of sample inputs and outputs and then generate the required fuzzy rules [109, 110] or a fuzzy controller can be designed by heuristics and then tuned using reinforcement learning neural networks [111].

6.3 Object-Oriented Analysis and Design Fuzzy Inference Systems

6.3.1 Introduction

In this section, an architecture for fuzzy inference systems based on OMT is presented. Object-oriented analysis is performed in order to discover pertinent objects and the relationships between objects found in the domain of Fuzzy Inference Systems. An object-oriented designed is described to produce a robust architecture for software implementations of fuzzy inference systems. An object-oriented architecture ensures that even when the fuzzy processing algorithms evolve, the modifications required for any prior implementation will be minimal. This is because the basic structure of a fuzzy inference system and hence its representation in object diagrams is relatively very stable over time compared to the algorithms used in the processing.

6.3.2 Problem statement: The domain of Fuzzy Inference Systems

In Logic, a Statement or Proposition is a verbal or mathematical Sentence that is either true or false [112]. Examples of a statement include:

x = 6;

The weather is hot

A compound statement is a combination of simple statements using various rules of combination. The statements are combined using connectives, AND, OR, EITHER...OR, etc. An implication (conditional) refers to the statement "*if p then q*". Both p and q must be propositions; p is called the premise or antecedent and q is the conclusion or consequent. An inference rule is defined to be an implication or *if...then* rule. It has a premise part and a conclusion. The premise part is either a single statement or a compound statement combined using logical connectives such as AND and OR.. The conclusion is usually a single statement but can also be compound. A statement is of the form "*A is High*" where *A* is a fuzzy input variable and *High* is a fuzzy Linguistic variable realised as a Membership Function over a given universe of discourse. The premise and conclusion parts are thus components of a Rule. The structure of the membership functions can be altered using modifiers or qualifiers such as Very, Almost, Quite, Extremely...; called fuzzy hedges. Each Rule has a weight which is the calculated truth value of its premise when the rule is activated. A Rulebase is an aggregation of Rules that describe the operation of a plant or system. Rules in the Rulebase are recognised by their rule number which acts as an index for inserting and retrieving Rules into the

Rulebase. A fuzzy inference system uses the Rulebase to infer or reason about data at its input resulting in an output which can be a fuzzy number or a crisp number.

6.3.3 Identifying Objects

As suggested before, the first step in identifying and discovering objects in the problem domain is by careful analysis of the problem statement and from domain knowledge. The simplest analysis process utilises nouns found in the description of the problem as a source of domain objects. These are further analysed to eliminate pseudo objects, aliases and spurious objects (objects in the problem description which have no bearing in the application or which are really attributes of other objects).

Table 6.3-1: Objects as Nouns in the problem description

Logical Connective	Proposition	Sentence	Statement
Simple Statement	Antecedent	Implication	Logic
Membership Function	Implication	Premise	Conclusion
Fuzzy Inference System	Consequent	Rule	Qualifier
Compound Statement	Rulebase	Data	Fuzzy Hedge
Linguistic Variable			

Analysis shows that some of the identified objects are aliases for previously identified objects in the context of the Fuzzy Inference System, e.g. Proposition and Statement, Implication and Rule, Premise and Antecedent, Conclusion and Consequent. Other objects including Sentence, Logic and Logical Connective are either not relevant to the application domain or are not really objects. A data dictionary can be constructed so that the role each object plays in the system is clearly defined.

The Data Dictionary

Rule — A fuzzy implication or conditional of the form "*if p then q*".

Proposition — A verbal or mathematical sentence that can be either True or False.

Statement — Another name (alias) for a Proposition, also a simple statement.

Compound Statement — Two or more statements combined using logical connectives.

Premise — The first half of a Rule, also known as antecedent.

Conclusion — The second half of a Rule, also known as consequent.

Linguistic Variable — A fuzzy predicates whose values are words or sentences in a natural or synthetic language.

Membership Function — A membership function realises a fuzzy linguistic variable in some universe of discourse.

Qualifier — A fuzzy qualifier or Modifier alters the shape of a membership function to fine tune its response to a particular set of inputs.

Rulebase — A collection of rules for use in an inference system.

Fuzzy Inference System — A software/hardware system that makes use of pre-stored or generated Rules to reason about data presented at its input.

Data — Patterns presented at the input of a Fuzzy Inference System

6.3.4 Organising the objects in the Fuzzy Inference System

A domain object model for the fuzzy inference system is in Figure 6.3-1. As mentioned before, domain objects are objects which are generally accepted by experts in the domain as pertinent to any application in the domain. The figure shows the different relationships that exist between objects in the fuzzy inference system. For instance, a "uses" relationship exists between the inference system and both Data and Rulebase. An aggregation relationship exists between Rulebase and Rules while the relationship between Linguistic Variables and Membership functions is one of generalisation or specialisation. Each Rule in turn is composed of a Premise part and a Conclusion part. Both Premise and Conclusion are related by virtue of the fact that they are all part of a Rule and also because Conclusions are usually with respect to preceding Premises. Both Premise and Conclusion have a Statement as an integral component, i.e. Statements make up both the Premise and the Conclusion. A Statement in turn consists of a Membership function and one or more Qualifiers (Modifiers). Finally, Membership functions are realisations of Linguistic Variables while Fuzzy hedges are a special kind of Qualifier.

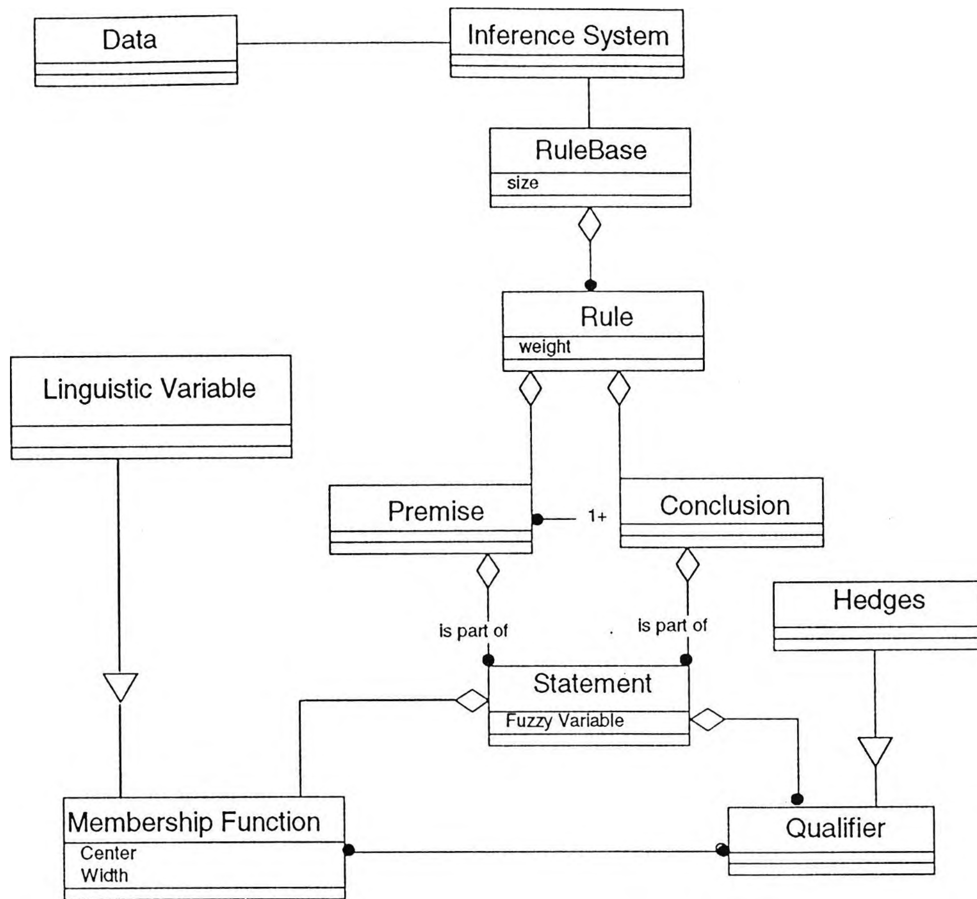


Figure 6.3-1: Domain Object Model for Fuzzy System

6.3.5 Determining operations on Objects

The different objects in the Fuzzy Inference System co-operate at a low level to perform a high level function. This function is not evident when the individual objects or the object diagram is examined. A functional model of the system shows what inputs are required and the transformations that the system performs on these inputs. Functional models are expressed in terms of dataflow diagrams. Figure 6.3-2 and Figure 6.3-3 show the context and level one dataflow diagrams respectively for the Fuzzy Inference System.

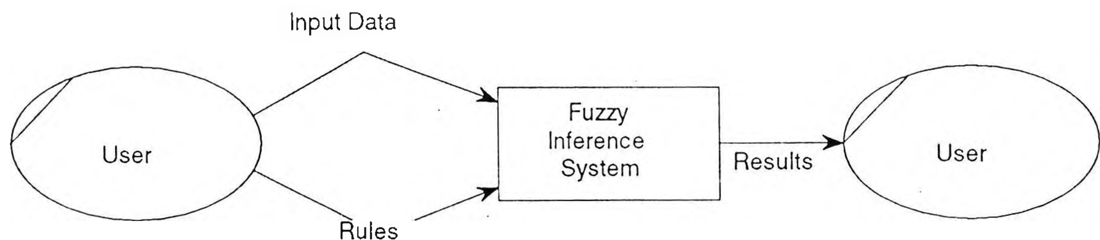


Figure 6.3-2: Context Diagram for Fuzzy Inference System

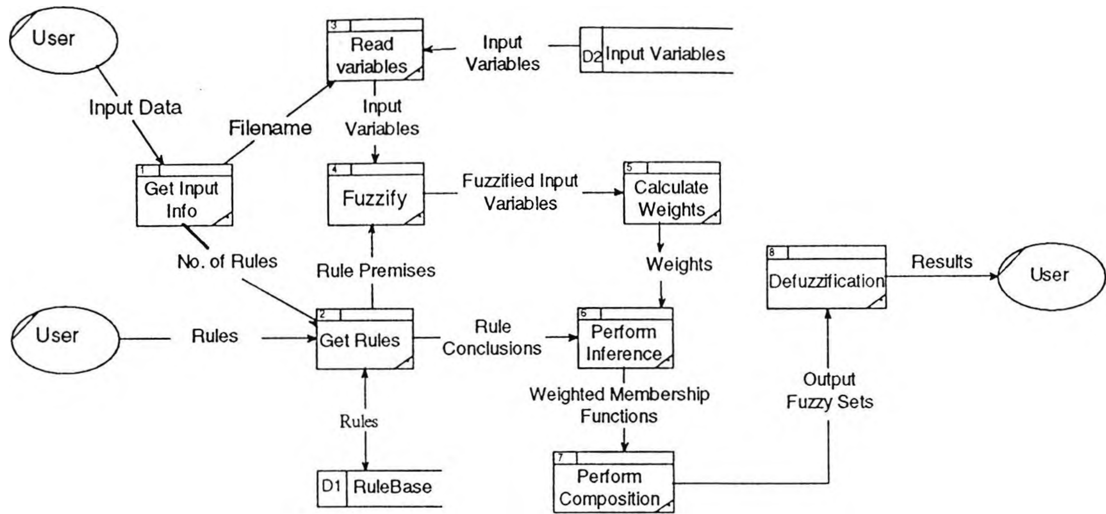


Figure 6.3-3: Level 1 dataflow diagram for Fuzzy Inference System

The system context shows that Input Data and Rules have to be entered into the system. These are transformed into inference results which are fed back to the User. The level 1 dataflow diagram shows the sequence of transformations that take place to convert the inputs to the final output seen by the User. Some of the processes can be further decomposed to give even lower level processes that describe the transformations in greater detail.

A dynamic model can be constructed for some of the system objects in order to portray the different states in which the objects can be in at different times. The behaviour of an object and hence the operations required of the object states is completely defined by knowledge of the states, the activities performed within those states and the events that cause transitions to other states.

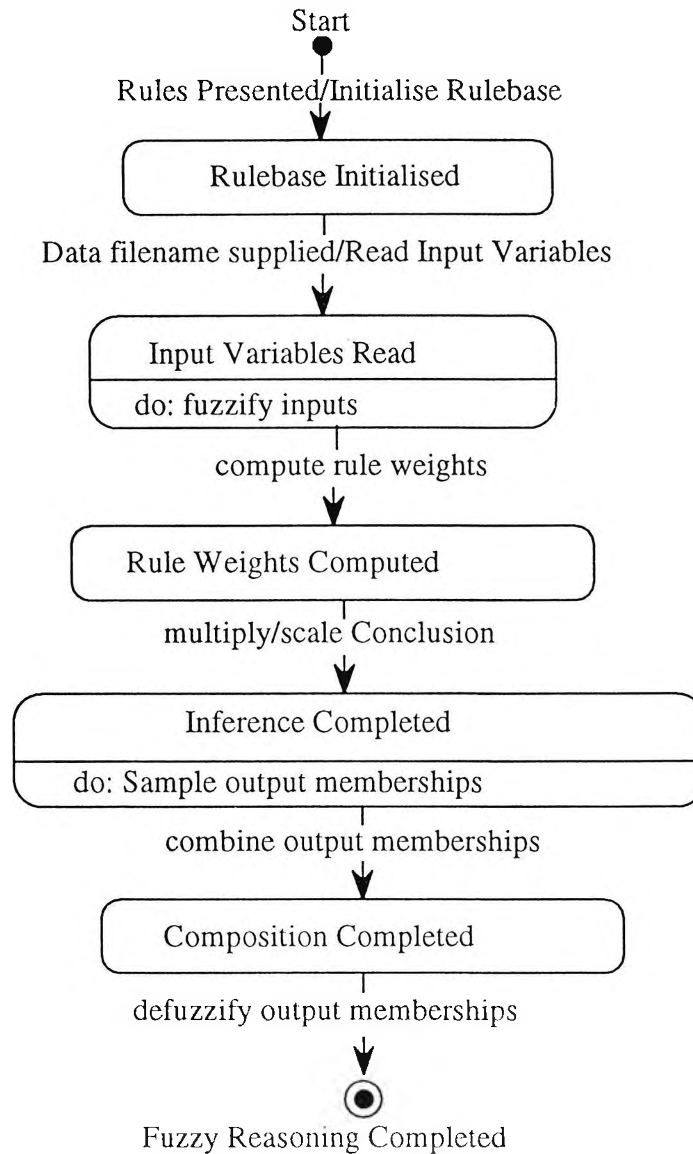


Figure 6.3-4: State transition diagram for the Fuzzy Inference System

Dynamic models are expressed in the form of state transition diagrams. Even for small systems, with few objects, an enumeration of all the possible states of all objects is extremely tedious because the number of possible states can be quite large. In theory, for a system with n objects each having m attributes, the number of possible states can approach $n!*m!$ as each change in an object's attribute is a possible cause of change in the state of the object and hence the system. The net effect is that, state transition diagrams are only constructed for those objects that are deemed to have important dynamic characteristics. Even then, only the major states and the most pertinent transitions are represented in the state diagram. Figure 6.3-4 shows the major states in the Fuzzy Inference System and the events that cause transitions between the states.

6.3.6 Design of Fuzzy Inference System

The purpose of an object-oriented design is to produce an architecture for a software system. As was the case with the Neural Network system, the design process seeks to produce a robust architecture for fuzzy systems based on information contained in the different analysis models. In the design process, the domain objects are moderated into a form suitable for implementation in a target language. Other supporting objects, not found in the description of the problem but necessary for efficient implementation of the system are also incorporated. The system is then partitioned into subsystems that can be easily implemented and that will remain robust and intact during it's lifetime as modifications and maintenance take place. The following subsystems were identified:

- Input Subsystem
- Rule Processing Subsystem
- Rule Activation Subsystem
- Sampling Subsystem
- Analysis Subsystem

As with the Neural Network design, the Input Subsystem handles interactions with the User. It incorporates objects to handle user inputs and hold parameters required for initialising other system objects. The Rule Processing Subsystem handles the insertion, modification and removal of rules from the Rulebase. It also deals with pre- and post-processing of the rules. The pre-processing translates the rule numbers into the mask which depends on the membership functions in the rule premise. The post processing converts the rules into English language *if...then* form which is understandable to the user. Pre- and Post- processing of the rules is necessary because the Rules are stored as a pattern of numbers which are offsets pointing into a table of membership functions. This avoids the need for dedicated lexical analyser and parser to understand and translate Rules written in English into a suitable format for use in the Inference System. The Rule activation Subsystem is really the inference engine. It handles the fuzzification, inference, composition, defuzzification required for fuzzy reasoning. It makes use of the Sampling Subsystem used to discretise output membership functions during the composition and defuzzification. The Analysis Subsystem simply processes and displays the results from the reasoning process. It doesn't have to be integrated in the system design and for the moment consists of an off-the-shelf spreadsheet to process and plot the results. Figure

6.3-5 shows how information flows amongst the different subsystems that make up the Fuzzy Inference System.

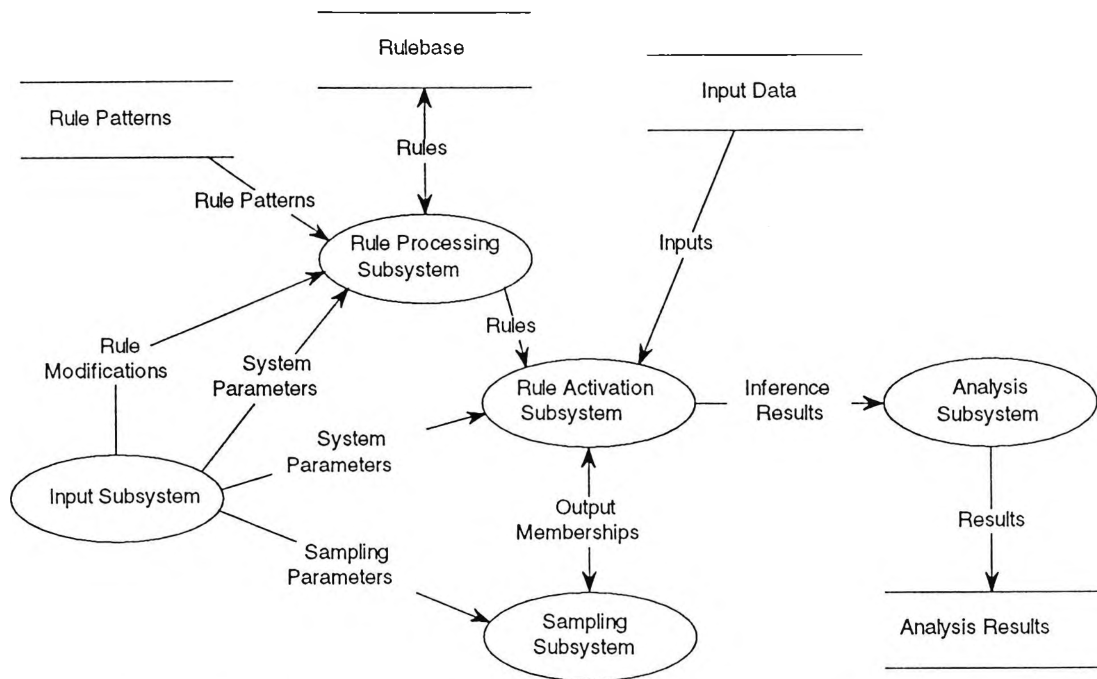


Figure 6.3-5: Information flow amongst the different subsystems

6.3.7 Implementation of the Fuzzy Inference Systems

To test the design, a prototype system with the four rules described in [105] was implemented using C++ on a SUN SPARC. The rules represent the benchmark XOR problem with two input variables and one output variable. The rules are as follows:

- If X is Low and Y is Low then Z is High
- If X is Low and Y is High then Z is Low
- If X is High and Y is Low then Z is Low
- If X is High and y is High then Z is High

X and Y are the input variables, Z is the output variable, and High and Low are the Membership functions. The universe of discourse for both X, Y and Z is given $0 \le (X|Y|Z) \le 10$. Both High and Low are Triangular membership functions given by the following equations:

$$\text{Low}(x) = 1 - \tfrac{x}{10}$$
$$\text{High}(x) = \tfrac{x}{10} \qquad (6.3\text{-}1)$$

For this simple system, each Rule object is implemented as a simple C++ class whose attributes are offsets in a membership function table. The Rule class declaration is show in Figure 6.3-6 below

```
class Rule {
public:
        int x;
        int y;
        Rule(int a=0, int b=0) {    // default constructor
                x = a;
                y = b;
        };
        ~Rule() {}                  // destructor
};
```

Figure 6.3-6: Declaration of a simple Rule class

The input fuzzy subset is partitioned into two; Low and High membership functions for each fuzzy variable. The rulebase has the four rules presented above. Figure 6.3-7 shows C++ declarations for the membership functions, the table of membership functions and the input and output membership functions tables that make up the rulebase.

```
Typedef float T;                                  // T is an alias for float

T lowt(T val) { return 1.0 - val/10.0 ; }         // implementation of low membership function

T hi_t( T val) { return val/10.0 ; }              // implementation of high membership function

T (*mf[])(T) = { lowt, hi_t };                    // mf is an array of membership functionss
                                                  // mf[0] = lowt(), mf[1] = hi_t()

static int rulebase[] = { 1, 0, 0, 1 };           // The rulebase, initialised with four rules
                                                  // membership functions, e.g. rulebase[0] = lowt()

static Rule r[] = {Rule(0,0), Rule(0,1), Rule(1,0), Rule(1,1)};    // initialised array of 4 rules
```

Figure 6.3-7: Declaration of membership functions and fuzzy Rulebase

In the figure, *lowt* and *hi_t* are the LOW and HIGH membership functions respectively, *mf* is the table of membership functions, *rulebase[]* is an array that represents the four output membership functions required for the conclusion part of the Rule and finally *r[]* is an array that represents four pairs of membership functions for each rule's premise.

Figure 6.3-8 shows the membership functions plotted over the given universe of discourse. The system was tested using both Min Inference with Max Composition and Product Inference with Sum composition. A plot of the sampled output membership function for X = 0.0 and Y = 3.2, after Sum composition is given by Figure 6.3-9. A

thousand samples were used to represent the universe of discourse. Centre of Gravity defuzzification was used to obtained the final crisp output. To calculated the centre of gravity for the sampled system, each sample is multiplied by the sampling step and all the products added together to give the area under the output membership function. This is then divided by the sum of all the samples.

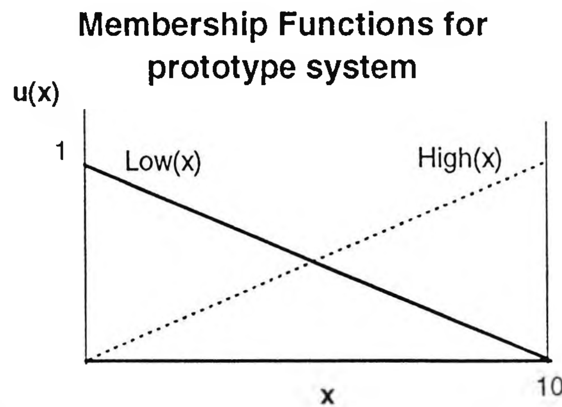


Figure 6.3-8: Triangular Memberships for Prototype system

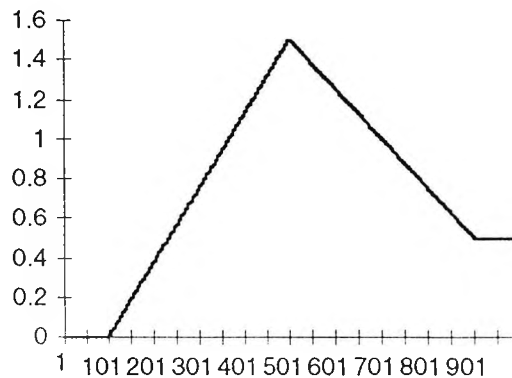


Figure 6.3-9: Sampled output membership showing Sum Composition

6.4 A Fuzzy Inference System for predicting harmonics in AC power networks

6.4.1 Introduction

Switching elements present AC-DC systems are a primary source of harmonics that are fed into the power networks. Detrimental effects of these harmonics include overheating, resonant overvoltages, communication interference and false tripping of relays. HVDC converters produce harmonics in the conversion process from AC to DC current. The harmonics can propagate into both the AC and DC transmission lines and can cause electromagnetic interference in their vicinity. In Flexible AC Transmission Systems (FACTS), the use of use of modern reactive power compensators such as thyristor-controlled reactors and thyristor-switched capacitors further injects harmonics at the

point of common coupling. This can cause harmonic instability by creating resonant overvoltages at lower order harmonic frequencies. To minimise interruptions and improve the reliable operation of the power system, the harmonics must be eliminated or suppressed. When the harmonic frequencies are known, the required harmonic component may be easily extracted from the polluted electrical waveform. The usual practice is to install a range of filters tuned to the frequencies of the harmonic components. Passive filters are used to supply a low impedance path at certain harmonic frequencies so that the harmonic currents can be short-circuited within the HVDC station. Unfortunately, the cost of installation and maintenance of these filters is prohibitively high.

Research and development into active filters have been boosted by major advances in key technologies including pulse width modulation (PWM) and digital signal processing (DSP). Active filters have been proposed as a means of overcoming some of the limitations of conventional filters in power supply systems [113]. The operation of a general active filter can be described as follows: The harmonic current is measured by a transducer. The measured signal is transmitted through optical fibres to DSP units. The power part of the active filter usually incorporates a high frequency transformer and a PWM power amplifier that functions as the harmonic voltage source. The voltage source sends harmonic currents on to the power line through bypass switches and any existing passive filters. A control unit ensures that the harmonic currents generated active filter have the opposite phases to the harmonics generated in the power network. In this way, the line harmonic currents are compensated. Figure 6.4-1 shows an example active filter using switched-capacitor circuits to generate the harmonic currents. The switched-capacitor filter consists of two capacitors in parallel with bi-directional switches in series with each capacitor. A third branch containing a small resistor is switched into the circuit to facilitate the smooth transfer of current between the capacitor branches. The filter is connected to the main power network via a small transient limiting inductor. The three switches are controlled to generate antiphase harmonic currents into the power network.

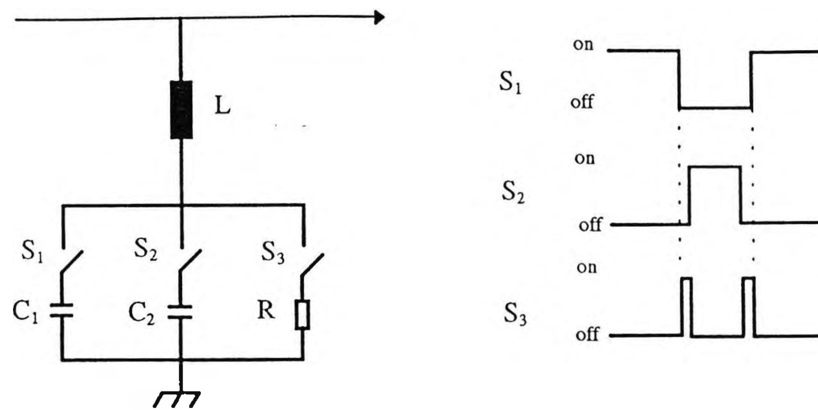


Figure 6.4-1: Switched-Capacitor Filter

In designing harmonic suppression filters, individual harmonic distortion or the total harmonic distortion expressed with reference to the fundamental frequency is an important index [114]. In theory, the nature of the harmonics and the frequencies present can be determined, given that the input current type is known. For instance, if the input current is a square waveform, then odd harmonics will be generated with amplitudes in inverse proportions to their orders. In practical implementations, the calculations become extremely complicated as thyristor firing angles and commutation times cannot be exactly controlled. Also, it is standard practice to have filters attached to the line to suppress all or some of the harmonics. Where passive filters are used, single tuned filters are used to eliminated low individual low frequency harmonics and a low pass filter to cut off higher order harmonics. The presence of the filters adds to the complexity of the resultant harmonics and make it even more difficult to predict. Many transformation techniques including FFTs [115] are available for computing the Fourier components of the harmonics. In the field of power systems, state estimation approaches that make use of Kalman filters have usually been employed as a means of identifying the harmonic contents [116]. Recently, computation intelligence techniques especially neural networks have proposed as a means of estimating the harmonics present in power networks [89, 117, 118]. It has been shown that trained neural networks can be used to estimated harmonic components in real time. In [89], the neural network training data is obtained by simulation over the operating range of the power network with different load, supply voltage and filter parameters. The level and content of the harmonics is measured for each case. The neural network learns a mapping between the system configuration and the harmonic contents. This approach is extremely tedious even for small systems with very few parameters because the of number simulations runs required. Furthermore, when the structure of the network changes, the training data will have to be

regenerated and a new neural network trained to predict the harmonics. The cost of obtaining data and retraining the networks is a major limitation to the neural network approach.

The approach proposed here to determine the implicit rules that govern the nature and content of harmonics in a network. These rules are expressed explicitly in the form of fuzzy *if...then* rules which can be processed and stored in a fuzzy rulebase. A fuzzy inference engine is then constructed using the rules to predict the harmonics in the network. The use of a rule-based fuzzy inference engine as opposed to an expert system offers considerable speedup in the processing of rules. This is because expert systems perform symbolic processing which is considerably slower than the equivalent fuzzy arithmetic operations performed in a fuzzy inference engine. The rules can be obtained through empirical studies of harmonics in power networks or through expert knowledge which comes from real world experiences of harmonics in power systems. In the absence of both, a third alternative is to simulate harmonic production in a small network while varying the system configuration. This gives an idea of harmonics content with different system configurations. As with the neural network approach, this is a vary tedious process but with the exception that the rules need to be determined just once from the simulation data. This approach presupposes that the same rules govern the production of harmonics irrespective of the structure of the system. Hence, a change in the structure of the system will not necessarily require a completely new set of rules.

6.4.2 Constructing the Rulebase

This section presents the process by which the explicit rules that govern the nature and content of harmonics in a single phase-controlled converter can be obtained. Figure 6.4-2 shows a schematic diagram of the phase-controlled converter with the active filter attached.

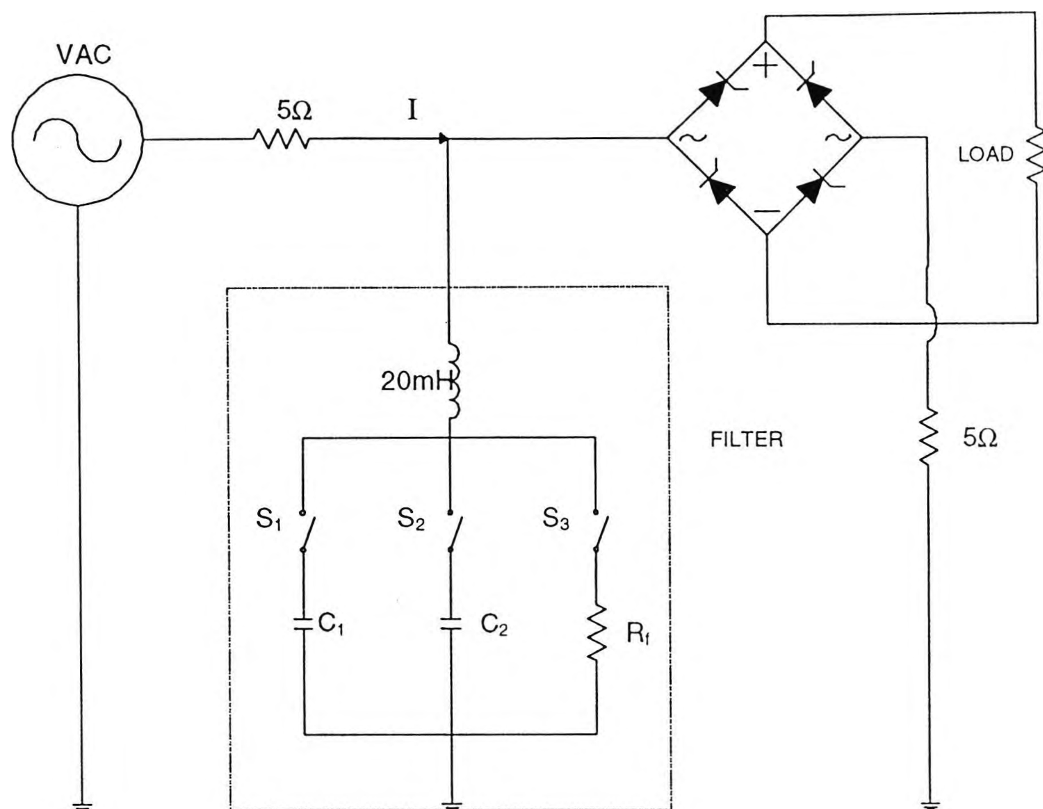


Figure 6.4-2: Thyristor controlled converter with switched-capacitor filter.

The presence of the filter can significantly alter the harmonic content and so the effects must be considered. The system is simulated using the EMTP. The simulation is set to 80ms. The input current, I, is in theory a pure sinusoid but the presence of the switching elements causes harmonics to be generated. For simulation purposes, only a single capacitance is used. The value is doubled and it is switched twice in each cycle to simulate the presence of a parallel capacitance branch. For this simple system, there are very few parameters to be varied. These include: switching time, capacitance, load resistance and the filter resistance as the most important system parameters. The input signal on the generator side is obtained from EMTP each simulation run. Fourier analysis is then carried out to obtain the harmonic contents. Only the first 25 harmonics components are of interest. These are used to compute a harmonic index which in this case is the total harmonic distortion (THD). It was realised that the harmonic content was pretty much independent of the load which meant that the load could be kept constant. A total of 394 different values for harmonic distortion were obtained by varying the resistor value, capacitance and switching times. The capacitor values range from 50uF to 800uF. The resistor values range from 10Ω to 100Ω in 10Ω increments. Finally, the switching time for S_1 and S_2 are made to vary between 0.2ms and 1.0ms. Figure 6.4-3 shows the general

trend of the harmonic distortion for all switching times and all resistance values when plotted against capacitance.

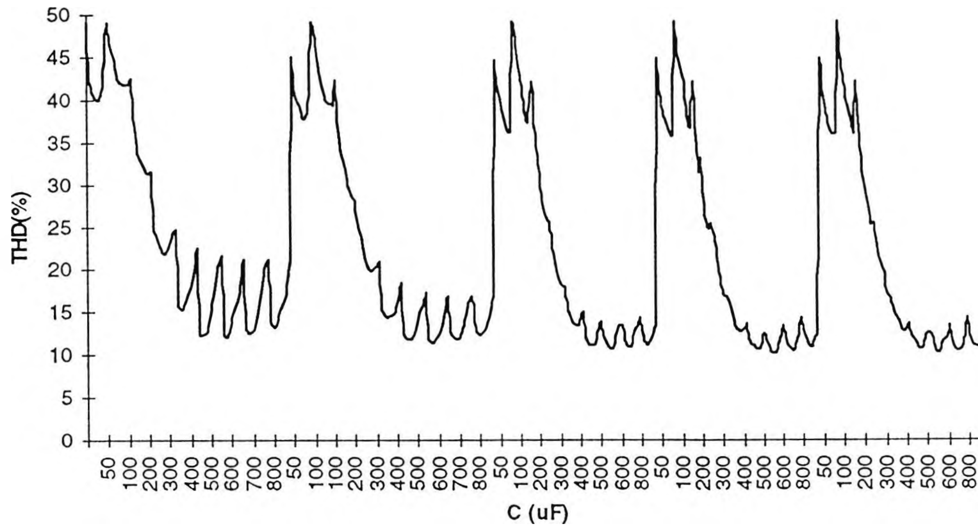
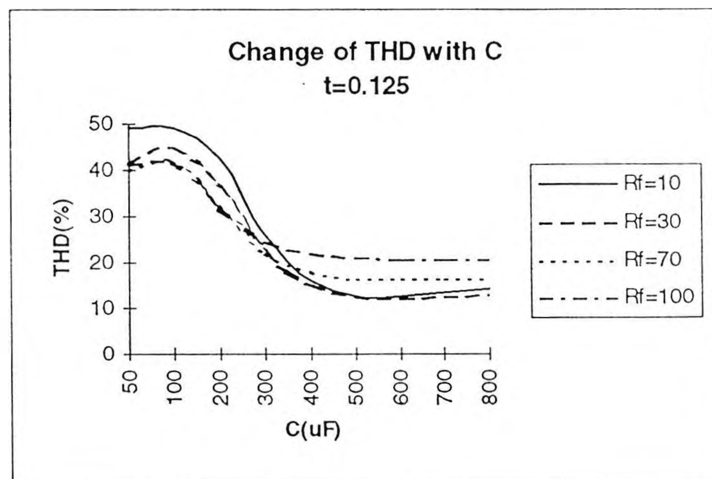
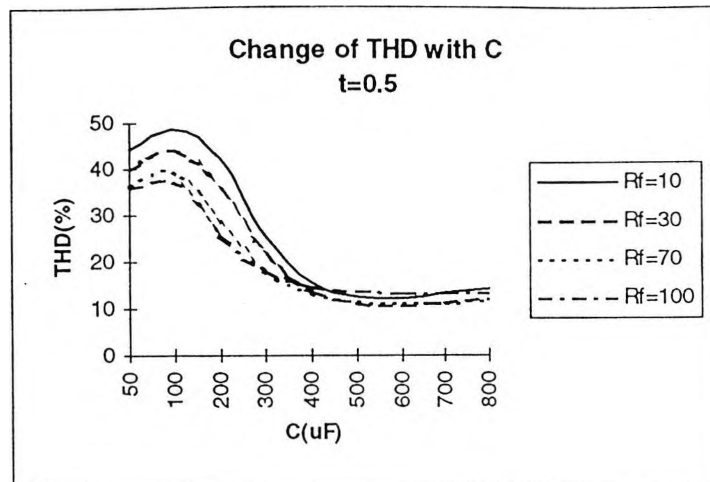


Figure 6.4-3: Plot of Total Harmonic Distortion (THD) for all Capacitance values

The required rules are obtained by observation and careful analysis of the way the harmonic index changes with the different parameters. The observations are in the form of trend graphs for THD as the different parameters vary. The first set of graphs in Figure 6.4-4 show how the harmonic distortion varies with capacitance for different resistance values and different switching times.





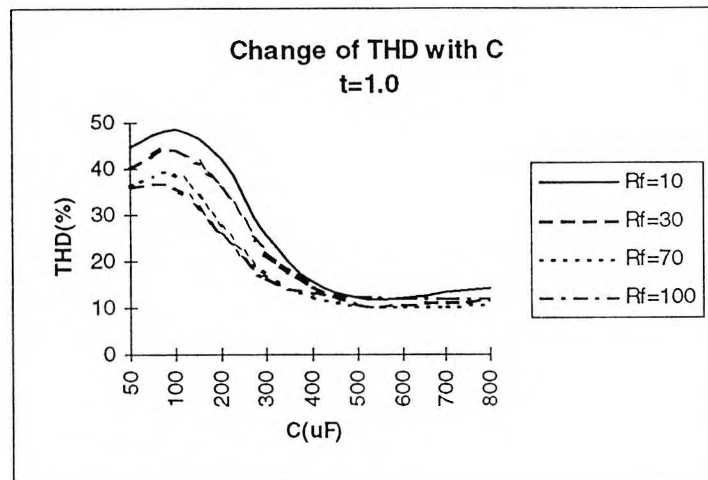
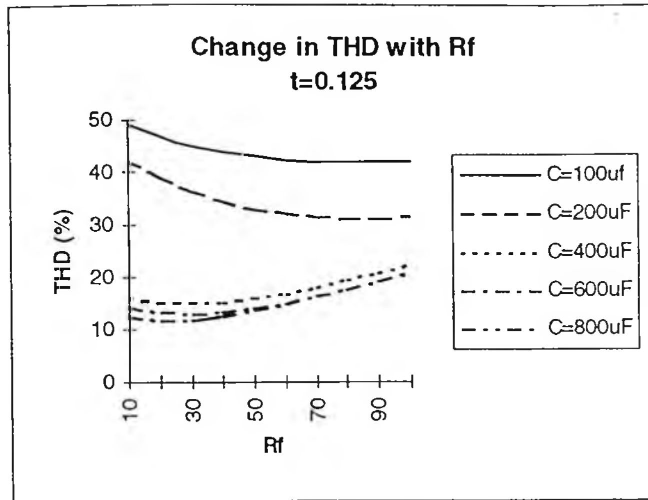
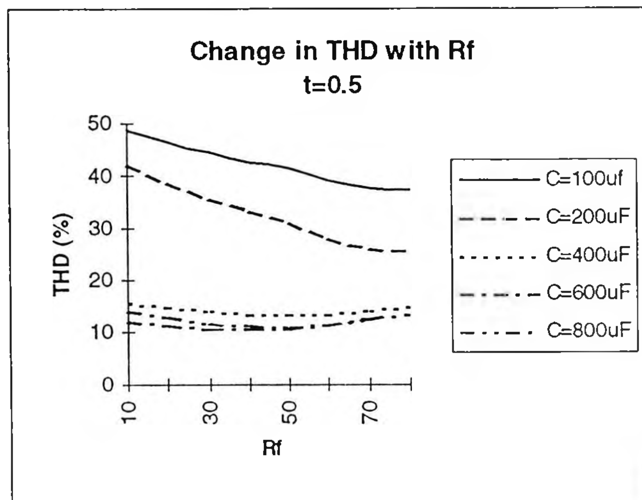


Figure 6.4-4: Variation of harmonic distortion with capacitance for different switching times

It can be seen from the graphs that the harmonics values tend to be high when the capacitance is low. As the capacitance increases, the harmonic distortion tends to decrease until a trough of around 500uF. This can be considered as medium values of capacitance. For higher values of capacitance, i.e. >500, the harmonics distortion increases again but much more slowly. A similar trend can be established for the way the harmonics change with the resistance values. From the graphs , it can be seen that when the resistance is low, the harmonics values tend to be higher and the harmonic values fall slightly as the resistance increases.





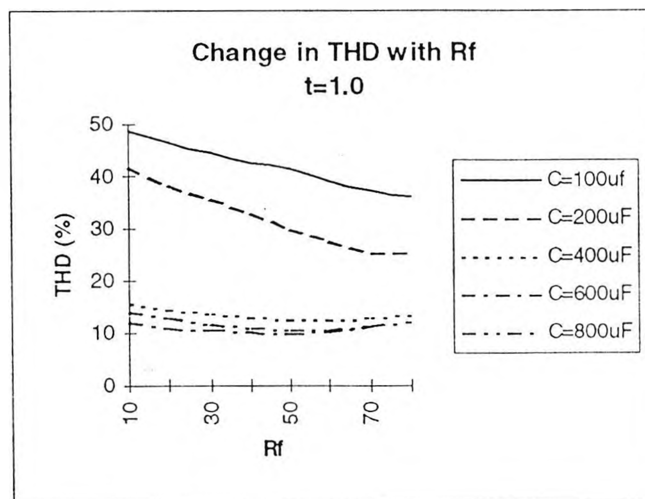
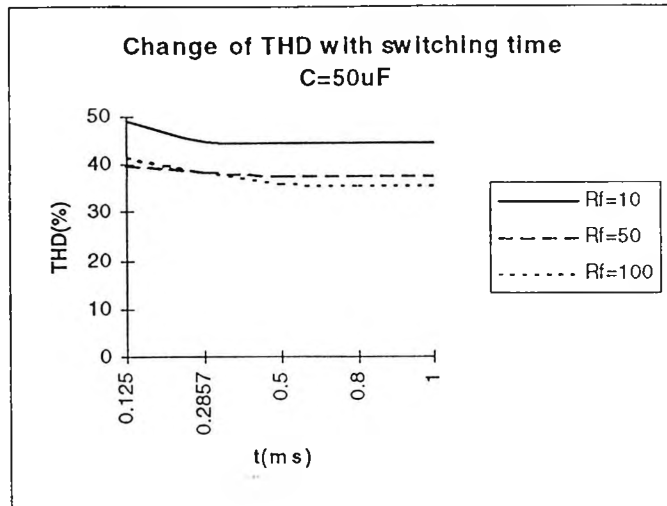
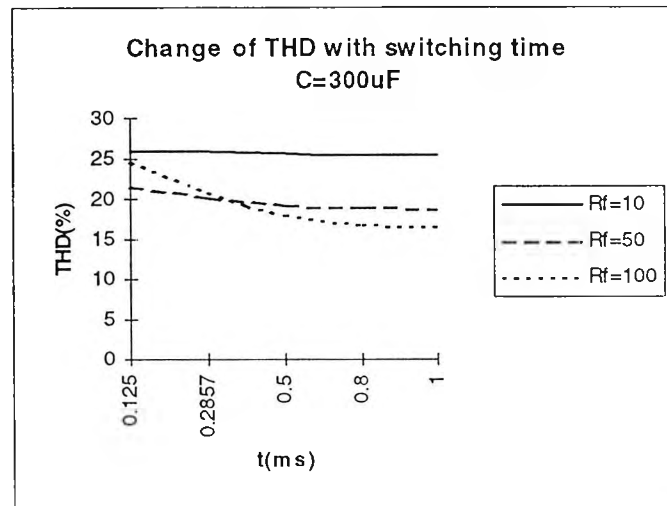


Figure 6.4-5: Variation of Harmonic Distortion with Resistance

The graphs in Figure 6.4-5 demonstrate how the harmonic distortion decreases as the resistance increases. The change in the three graphs for different values of switching time further demonstrates that the harmonics change only gradually with the time.





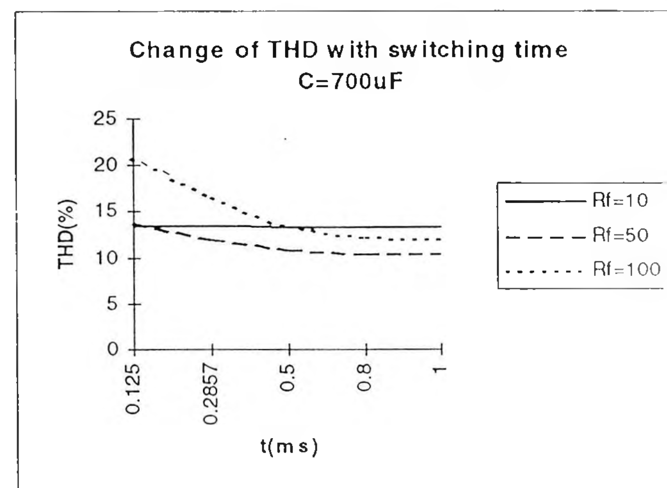
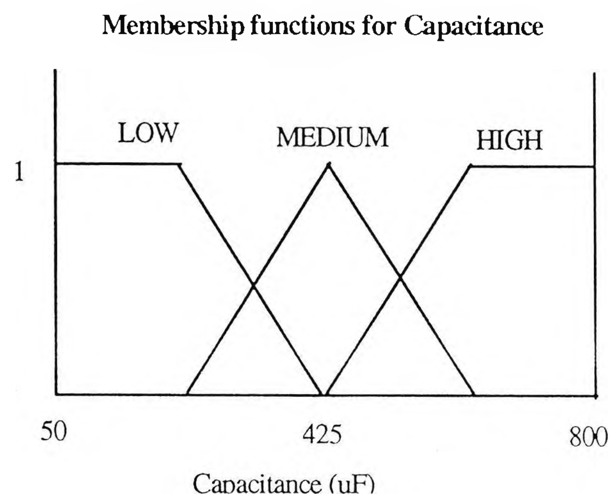


Figure 6.4-6: Variation of Harmonic Distortion with switching time

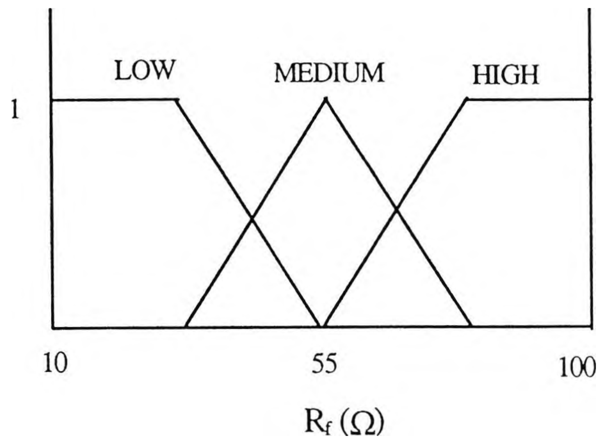
The last set of graphs confirm the slow variation of the harmonic distortion with switching time.

The next step in the construction of the fuzzy Rulebase involves setting up the fuzzy subsets. A fuzzy subset is constructed for each of the variables in the system. Input fuzzy subsets are constructed for the capacitance, resistance and switching time while an output fuzzy subset is constructed for the harmonic distortion. Each fuzzy subset consist of 3 fuzzy or linguistic variables; namely Low, Medium and High. Triangular/trapezoidal membership functions have been used to construct the both the input and the output fuzzy subsets because they are mathematically easier to process. For each variable, the universe of discourse is partitioned into 3 equal portions. Each linguistic variable is expressed as a membership function over a single fuzzy partition. The combination of the 3 membership functions a linguistic variable over its universe of discourse represents the fuzzy subset for that variable.

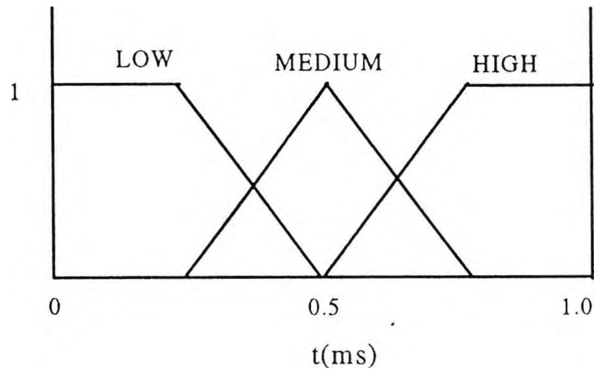
Membership functions for Capacitance



Membership functions for Resistance



Membership functions for the switching time



Membership functions for Harmonic Distortion

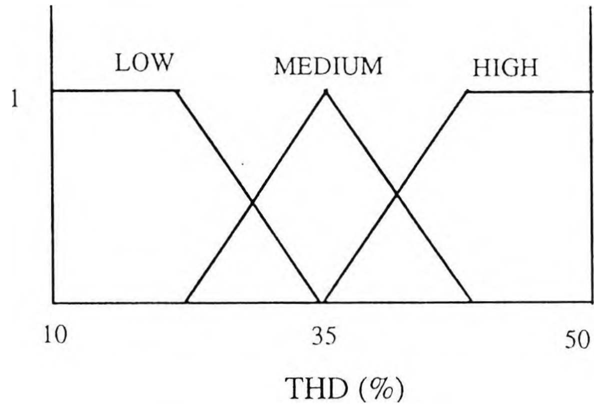


Figure 6.4-7: Fuzzy subsets for the different system parameters

The membership functions for the Capacitance are as follows:

$$LOW(C) = \{ \begin{array}{ll} 1, & \text{if } C < 212.5uF, \\ -\frac{1}{212.5}C + 2, & \text{if } C \geq 212.5uF \text{ and } C \leq 425uF, \\ 0, & \text{if } C > 425uF \} \end{array} \quad (6.4\text{-}1)$$

$$MEDIUM(C) = \{ \begin{array}{ll} 0, & \text{if } C < 212.5uF, \\ \frac{1}{212.5}C - 1, & \text{if } C \geq 212.5uF \text{ and } C < 425uF, \\ -\frac{1}{212.5}C + 3, & \text{if } C \geq 425uF \text{ and } C < 637.5uF, \\ 0, & \text{if } C \geq 637.5uF \} \end{array} \quad (6.4\text{-}2)$$

$$HIGH(C) = \{ \begin{array}{ll} 0, & \text{if } C < 425uF, \\ \frac{1}{212.5}C - 2, & \text{if } C \geq 425uF \text{ and } C < 637.5, \\ 1, & \text{if } C \geq 637.5 \} \end{array} \quad (6.4\text{-}3)$$

Similar equations were written to describe the membership functions for Resistance, switching time and Harmonics distortion. By combining the trend graphs in Figure 6.4-4, Figure 6.4-5 and Figure 6.4-6 with the input and output fuzzy subsets in

Figure 6.4-7, the fuzzy *if...then* that govern the nature and content of the harmonics can be written.

Some examples of the rules that were deduced are as follows:

if C is **High** and R is **Medium** and t is **Low** *then* Harmonics is **Low**

if C is **Medium** and R is **Low** and t is **High** *then* Harmonics is **Medium**

if C is **Low** and R is **Medium** and t is **Low** *then* Harmonics is **High**

if C is **Medium** and R is **Medium** and t is **Medium** *then* Harmonics is **Medium**

For a system with 3 input variables, each represented by a fuzzy set with membership functions, a maximum 27 rules can be written from all combinations of input variables and membership functions. The output membership function (i.e. for harmonic distortion) that correspond to each rule was deduced directly from the actual data as represented by the trend graphs in Figure 6.4-3, Figure 6.4-4, Figure 6.4-5 and Figure 6.4-6. The set of rules in the Rulebase is shown in Table 6.4-2. Once the rules have been deduced, tables of membership functions are created for Capacitance, Resistance

Switching time and Harmonic Distortion. This is necessary because, even though the input and output fuzzy subsets have similar shapes and the same number of input variables, the universes of discourse are different, hence different membership functions are required. For example, the universe of discourse for Capacitance is $50uF \leq C \leq 800uF$ while that for Resistance is $10\Omega \leq R \leq 100\Omega$. The rules are stored in a file as patterns of numbers in the range 0-2:

where 0 represents low,

1 represents medium and

2 represents high.

An extract from the Rules file for the four example rules above is shown in Table 6.4-1

Table 6.4-1: Format of Rule file

Rule Number	Capacitance	Resistance	Switching time	Harmonics
22	2	1	0	0
12	1	0	2	1
4	0	1	0	2
13	1	1	1	1

A Rule Processing Subsystem reads the patterns and converts them to the appropriate membership functions using a lookup table of functions. As mentioned before, this avoids the need for a dedicated lexical analyser and parser in order to translate and understand *if...then* rules written in the English Language. The Rulebase can then be constructed when the inference system is initialised by reading in the rules from file.

Table 6.4-2: The Rulebase for inferring Harmonics Distortion

No	Capacitance	Resistance	Time	Harmonic Distortion
1	low	low	low	high
2	low	low	medium	high
3	low	low	high	high
4	low	medium	low	high
5	low	medium	medium	high
6	low	medium	high	medium
7	low	high	low	medium
8	low	high	medium	medium
9	low	high	high	high
10	medium	low	low	medium
11	medium	low	medium	medium
12	medium	low	high	medium
13	medium	medium	low	low
14	medium	medium	medium	medium
15	medium	medium	high	medium
16	medium	high	low	low
17	medium	high	medium	medium
18	medium	high	high	medium
19	high	low	low	low
20	high	low	medium	low
21	high	low	high	low
22	high	medium	low	low
23	high	medium	medium	low
24	high	medium	high	low
25	high	high	low	medium
26	high	high	medium	low
27	high	high	high	low

6.4.3 Results

The first 90 patterns input patterns from the EMTP simulation are used as inputs to the fuzzy inference system to test its operations. This represents the complete range of capacitance and resistance values for one value of switching time. The normalised predicted harmonics plotted against pattern number as shown in Figure 6.4-8. When compared with the first cycle of the harmonics Vs time from the simulation (Figure 6.4-3), the graph shows that the fuzzy system is capable of inferring the general trend of the harmonics even though the exact values are not correct.

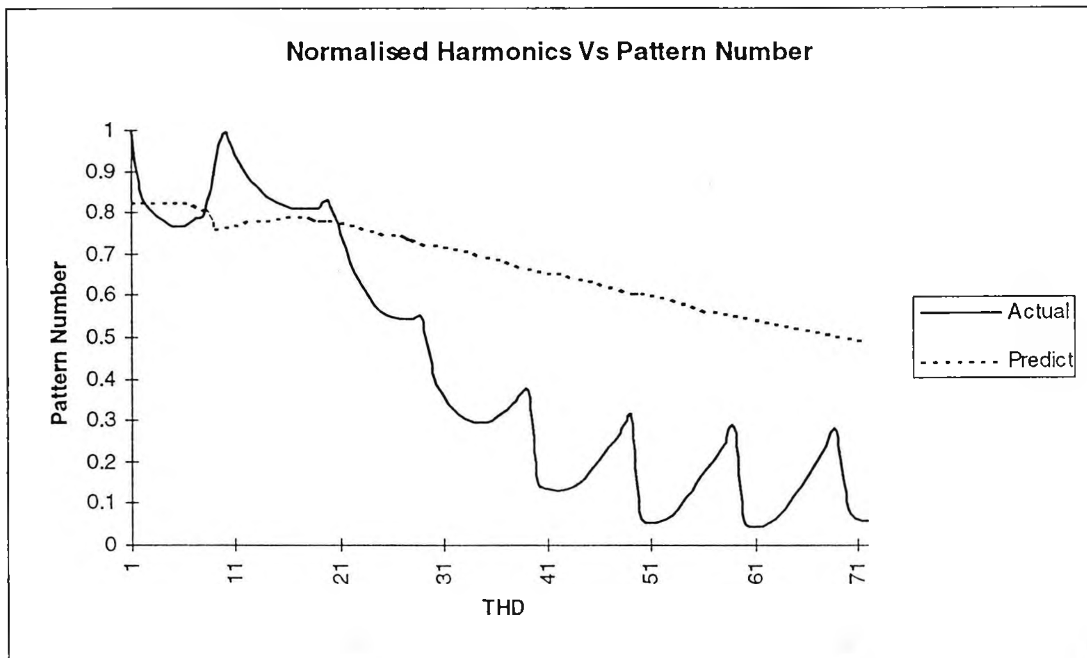


Figure 6.4-8: Normalised Harmonics Vs Pattern Number

Over the complete range and especially for normalised input values, the predicted harmonic values are very poor as can be seen in Figure 6.4-9 and Figure 6.4-10. There are a number of possible reasons for the poor performance. It is possible that the Rules that describe how the harmonic distortion changes with time are not correct. The number of simulations that could be carried out by changing the switching time is severely limited by the tedious nature of the data collection process. An exhaustive simulation over a wider switching time, Capacitance and Resistance range could well have yielded more accurate trends in the way the harmonic distortion varies with the different parameters and hence better rules for the inference process.

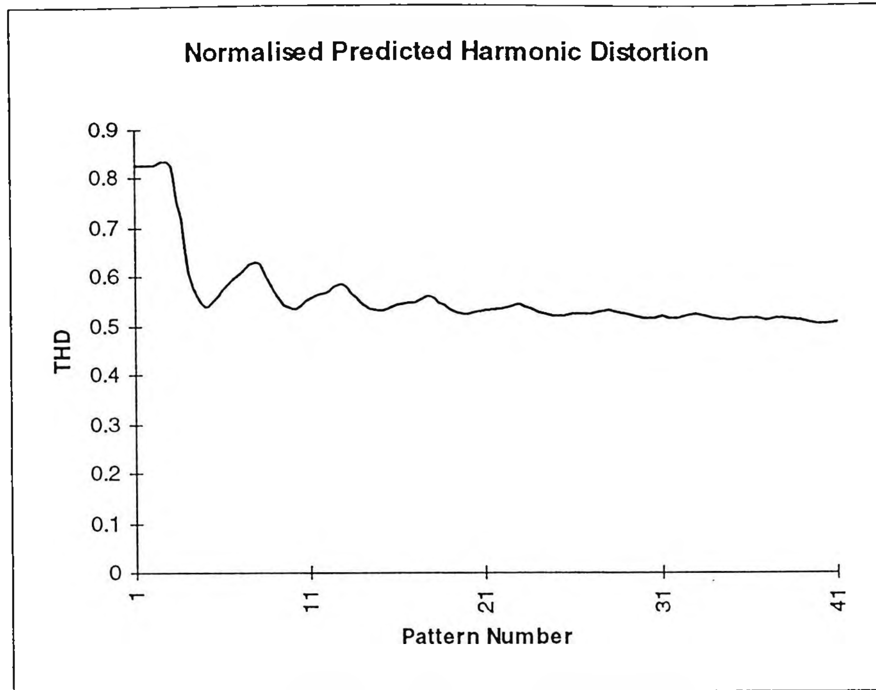


Figure 6.4-9: Predicted Harmonics Distortion using Normalised inputs

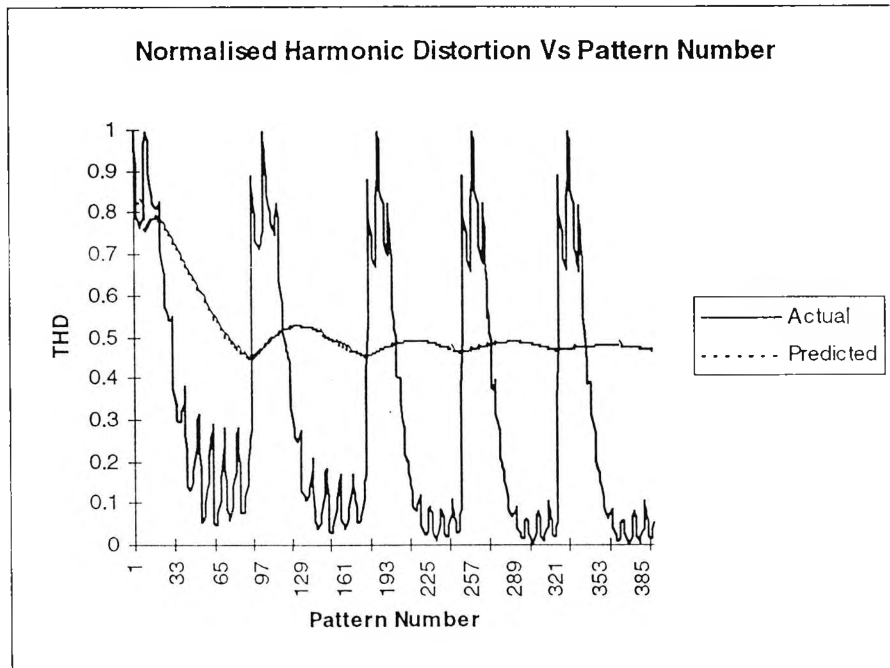


Figure 6.4-10: Predicted Harmonic Distortion for Normalised Inputs for all Patterns

It is also possible that the partitioning of the input and output fuzzy subsets was insufficient to correctly represent the required mapping. The use of three linguistic

variables Low, Medium and High results in a very coarse partitioning of the input and output fuzzy subsets. A finer partitioning over the same universe of discourse, with say five or seven linguistic variables and/or smooth membership functions would result in smoother transitions between rule firings. Unfortunately, considerable effort is required to derive the very large number of rules that would result makes from finer partitioning. Example fuzzy sets with five and seven linguistic variables to represent Capacitance is shown in Figure 6.4-11: Finer partitioning of Universe of Discourse for Capacitance and Figure 6.4-12 respectively.

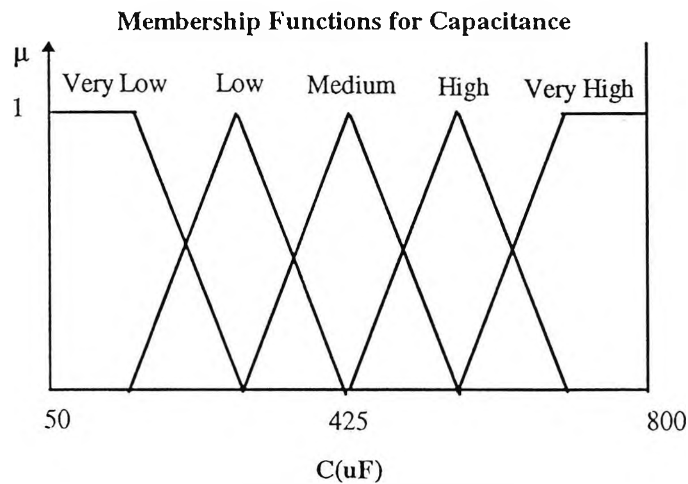


Figure 6.4-11: Finer partitioning of Universe of Discourse for Capacitance

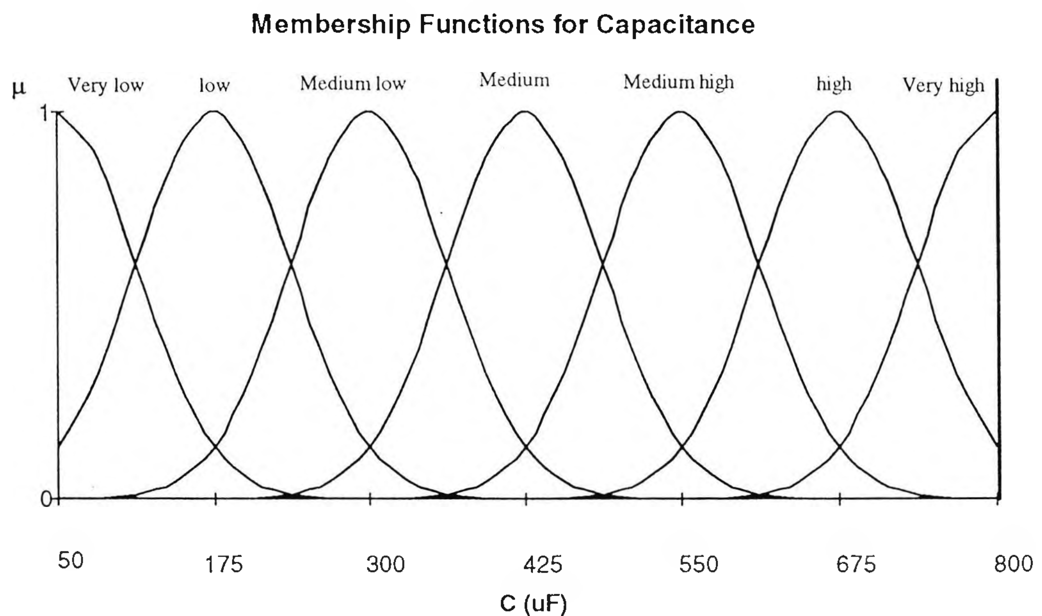


Figure 6.4-12: Even finer partitioning of Universe of Discourse for capacitance

6.5 Discussion and Conclusions

Conventional inference systems that rely on symbolic processing of pre-stored rules have been criticised for being too slow because of the huge computational burden required by symbolic computation and the sequential firing of rules necessary to infer a single decision. They are also very tedious to design and implement because of the disproportionate amount of work required to collect, collate and formulate the necessary rules even for average sized inference systems. Fuzzy inference systems provides a faster alternative. Fuzzy rules are much more simpler to formulate because of the built-in vagueness and imprecision. Furthermore, fuzzy rules are processed numerically rather than symbolically which makes fuzzy rule processing not only less demanding of storage space but also orders of magnitude faster than their symbolic counterparts. Finally rules in a fuzzy rulebase usually fire in parallel whereas symbolic rules are more often sequential. The combination of numeric processing, small storage requirements and intrinsic parallelism, makes it possible to implement fuzzy inference systems on dedicated VLSI hardware where even higher speeds have been attained.

This chapter has demonstrated how fuzzy systems can be analysed, designed and implemented using object-oriented techniques. Two sample implementations have been realised using the C++ programming language. The first is a simple test system with four rules that demonstrates that such analysis and design procedure can be used to realise working fuzzy inference systems. The second represents the development of an inference system for a practical problem where the rules have to be derived. Even though the initial results are not very good, they still show the potential high benefits of fuzzy inference in solving practical problems.

Chapter VII

EVOLUTIONARY COMPUTATION

7.1 Introduction

Natural evolution produces increasingly fit organisms in complex environments which are highly uncertain for individual organisms [119]. The individual organisms have to learn to adapt to changing environmental conditions if they are to survive. Global optimisation algorithms imitating certain principles of nature have proved their usefulness in various applications domains [120]. Evolutionary Computation (EC) is useful for optimizing problem solutions when other techniques like gradient descent or direct analytical discovery are not possible [121]. Evolutionary Computation may currently be characterised by the following: Genetic Algorithms (GA), Evolutionary Programming (EP), Evolution Strategies (ES), Classifier Systems (CFS), Genetic Programming (GP), and several other problem solving strategies, based upon biological observations that date back to Charles Darwin's discoveries in the 19th century: the means of natural selection and the survival of the fittest, i.e. the "theory of evolution." The resulting algorithms are thus termed Evolutionary Algorithms (EA).

Evolutionary Algorithms use computational models of evolutionary processes as key elements in the design and implementation of computer-based problem solving systems. A variety of evolutionary computational models have been proposed. They share a common conceptual base of simulating the evolution of individual structures via processes of selection, mutation and reproduction. The processes depend on the perceived performance of the individual structures as defined by a given problem environment.

Genetic Algorithms are the most popular and widely used of all the evolutionary algorithms. They have been widely applied to solve complex nonlinear optimisation problems in a number of engineering disciplines [122]. Like other computational intelligent systems such as Neural Networks and Fuzzy Systems, Genetic Algorithms currently exist as lines of computer code written to solve particular problems. The previous chapters have attempted to redress the situation by providing a software engineering base from which Neural and Fuzzy systems can be constructed. In this chapter, a similar analysis and design philosophy, using object-oriented approach is presented for the construction of Genetic

Algorithms and Learning Classifier Systems. The use of object-oriented techniques means that a GA component can be created and incorporated into other designs where an evolutionary solution is required. The high level of reuse exemplified by such an approach is demonstrated in the construction of a Genetic Learning Classifier System incorporating the GA design.

7.2 Genetic Algorithms

7.2.1 Introduction

Genetic Algorithms (GA) are robust search mechanisms based on the principle of population genetics, natural selection and evolution [123]. Genetic Algorithms perform global search on the solution space of a given problem domain [124]. Genetic search is capable of satisfying strong hidden constraints with reasonable efficiency. The characteristics of genetic search mechanisms that enable them to be able search globally and still converge to a solution include :

Learning while searching: As GA search progresses, the scope of the search is narrowed as information about the function space accumulates.

Sustained exploration: The GA incorporates mechanisms that prevent the search space from being irrevocably narrowed so far as to let the optimal solution states to slip permanently through the GA search net.

Genetic search is characterised by three components:

Ongoing state(Ω): The ongoing state represents knowledge that contains information previously acquired during the search and also a means of feeding information from past searches to aid the current or future search.

Search Function (S): Uses the ongoing state at time (generation) k to generate the next point to search. $\{S(\Omega) = x_{k}\}$.

Learning Function (L): Uses the ongoing state and the location and value of the most recently searched point to update the ongoing state.

Search strategies can broadly be divided into three categories [125]:

7.2.1.1 Path based models

In path based models, the search space is recursively defined by a starting state and a set of state to state transition operators. The search strategy then has to find a solution state and

a path to it from the starting state. Examples of path based search include tree searching techniques such as depth first search.

7.2.1.2 Point based models

Point-based models maintain the location and value of one point in the search space defined to be the current point. A neighbourhood is defined around this point to be the promising region. The current point is used as the standard for comparison. New points are generated at each step of the search. Better points get credit by becoming the current point while bad points are discarded.

7.2.1.3 Population based models

Population-based models maintain a set of locations and values in the function space. The average value of the population can be used as the standard of comparison and a discovered good point can be added to the population according to some rule or rules.

7.2.2 Features of Genetic Algorithms

Genetic algorithms belong to the class of population-based search strategies. They operate on population of strings (chromosomes) that encode the parameter set of the problem to be solved over some finite alphabet. Each encoding represents an individual in the GA population. The population is initialised to random individuals (random chromosomes) at the start of the GA run. The GA searches the space of possible chromosomes (hamming space) for better individuals. The search is guided by "fitness" values returned by the environment. This gives a measure of how well adapted each individual is in terms of solving the problem and hence determine its probability of appearing in future generations. A binary encoding of the parameters of the problem is normally used. It has been mathematically proven [123] that the cardinality of the binary alphabet maximises the number of similarity templates (schemata) on which the GA operates and hence improves the search mechanism.

Two types of rules are used by Genetic Algorithms in their search for highly fit individuals; selection rules and combination rules. The selection rule is used to determine the individuals that will have a representation in the next generation of the GA. The combination rules operate on selected individuals to produce new individuals that appear in the next generation. The selection mechanism is based on a fitness measure or objective function value, defined on each individual (chromosome) in the population. Two major selection mechanisms are commonly adopted in GA search. Roulette wheel selection and

tournament selection. In roulette wheel selection, the probability of being selected is proportional to an individual's fitness value. Therefore, highly fit individuals have a higher probability of being selected and hence of being represented in the next generation. In tournament selection, a fraction of the individuals in the population are randomly selected into a sub population and competition carried out to select the fittest individuals in each sub population. Table 7.2-1 and Figure 7.2-1 show a comparison between roulette wheel and tournament selection for a population of 10 individuals with randomly initialised fitness values between 0 and 100. The population is sampled 1000 times in each generation and the number of wins per individual is averaged over 10 generations and tabulated. For tournament selection, the sub population size is set to half of the population size.

Table 7.2-1: Comparison between Roulette Wheel and Tournament selection mechanisms.

	Fitness	Roulette Wheel	Tournament
	9	15	0
	74	147	133
	32	83	0
	93	199	487
	40	70	0
	61	119	18
	75	139	304
	41	63	0
	48	69	2
	62	96	56
Total	535	1000	1000

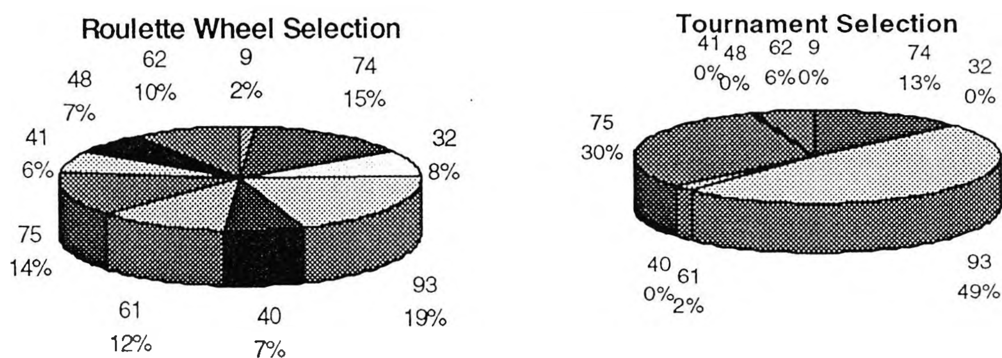


Figure 7.2-1: Pie Charts comparing the different selection mechanisms

The results show that roulette wheel selection gives even the least fit members of the population a chance of getting represented in the next generation. Tournament selection,

on the other hand, is strongly biased in favour of the fittest individuals with a subset of the least fittest individuals guaranteed to disappear from the population in each generation.

Combination rules are used to introduced new individuals into the current population or to create a new population based on the current population. The genetic algorithm uses certain operators (genetic/search operators) [123, 125] in the combination process. The most commonly used genetic operators are **crossover** and **mutation**. Other less commonly used ones include inversion and deletion. The combination rules act on individuals that have been previously selected by the selection mechanism. A reproduction process takes place between the selected individuals in the current generation to produce offspring that become individuals in the next generation (Figure 7.2-2).

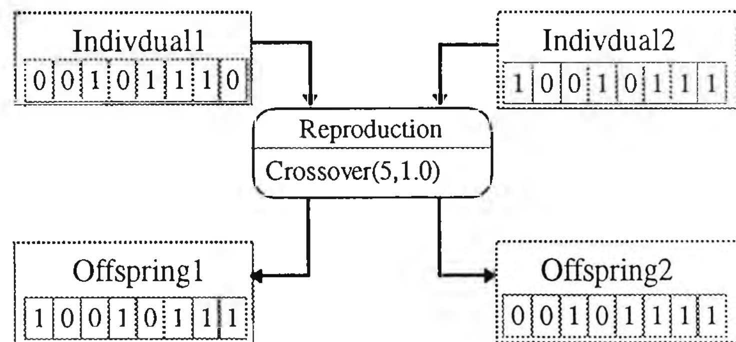


Figure 7.2-2: Schematic diagram depicting the reproduction process

Crossover is a combination rule that produce offspring in the hamming interpolation of the parents. The crossover operator produces a non-uniform sampling probability distribution over the subspace. When standard or one point crossover is applied on a pair of chromosomes, a real value generated at random is compared with the crossover threshold, if the value is less than the threshold, then a random site is picked for crossover of genetic material between the two chromosomes. A number of alternative crossover strategies have also been suggested [126]. These include: two point crossover, multi-point crossover and uniform crossover. In two point crossover, two sites are selected at random and the participating chromosomes swap genetic material between the sites. Multi-point crossover is similar to two point crossover. Here, the number of crossover points is selected at random and alternative sections of genetic material swapped between participating chromosomes. Finally, with uniform crossover, a crossover mask is generated initially for all chromosomes. For each mask position, a real value is generated at random and compared to the crossover threshold. If the value is less than the threshold, then the

mask at the position is set otherwise the mask is cleared. For a pair of participating chromosomes, exchange of genetic material takes place only at positions where the mask is set. Figure 7.2-3, Figure 7.2-4 and Figure 7.2-5 show how the different crossover strategies are applied to a pair of participating chromosomes and the resulting offspring.

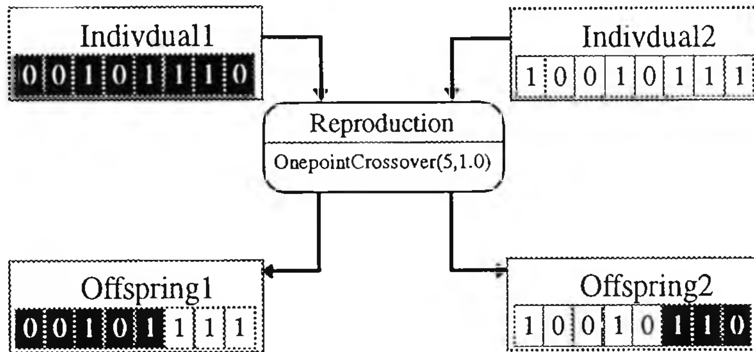


Figure 7.2-3: One point Crossover

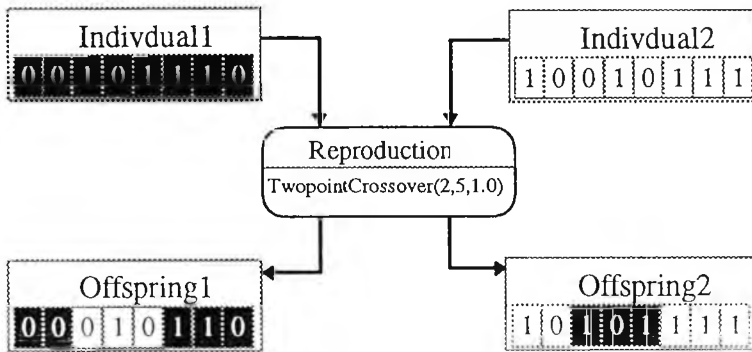


Figure 7.2-4: Two point Crossover

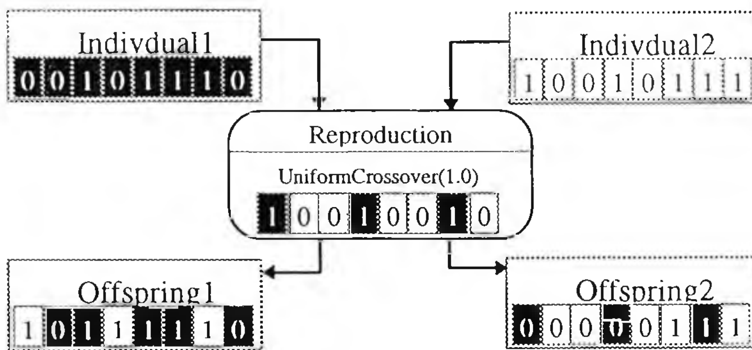


Figure 7.2-5: Uniform Crossover

Mutation is the random occasional alteration of the information contained in the chromosome. The mutation probability determines how often mutation occurs. By itself, mutation is a random walk through the hamming space but when mutation is combined with crossover, it acts to improve the performance of the Genetic Algorithm by preventing premature loss of genetic information from the GA population.

7.2.3 Object-Oriented Analysis of Genetic Algorithms

An object-oriented analysis of Genetic Algorithms is performed to:

- understand the problem,
- identify objects which will remain important in the life of the application,
- identify the relationships between the different objects and the ways in which the objects interact,

so that a correctly working, robust and easy to maintain GA system can be designed and subsequently constructed.

7.2.3.1 Identifying objects in the GA domain

The process of analysis begins by identifying the objects found in the GA problem domain. As mentioned before, the objects in the domain can be obtained by a careful analysis of the problem description combined with expert knowledge of the particular problem domain. In the absence of the later, domain analysis can be carried out to identify the objects which experts in the domain perceive to be important to applications in the domain. This is done by studying previous applications, by talking to the experts and by reading the literature. In the GA domain, five main objects have been identified: Genes, Chromosomes, Individuals, Population and Genetic Algorithm (GA). A dictionary has been created for the objects so that their use in the description of the Genetic Algorithms is unambiguous.

The Data Dictionary

Genetic Algorithm — Robust search mechanism based on population genetics and natural selection. A Genetic Algorithm object is an algorithmic object that evolves highly fit individuals that represent a parameter encoding of a problem to be solved.

Population — The collection of individuals on which the GA acts.

Individual — A structure representing a parameter coding of the problem to be solved.

Chromosome — In biology, a Chromosome is a locus of alleles that determine the phenotype of an individual organism. In a GA, a Chromosome is a collection of Genes that holds structural information about individuals found in the GA Population.

Gene — A Gene or Allele is the information contained in a single position or locus of a Chromosome. It represents the lowest level coding of the parameters of the problem to be solved by the Genetic Algorithm.

It should be stressed that analysis is done in the problem domain, and analysis domain objects are problem domain objects. The aim is to uncover important objects and relationships so that a correct GA system can be designed. A domain object model is used to describe the static structure of the system and the relationships between the domain objects. A class diagram representing the domain object model is shown in Figure 7.2-6.

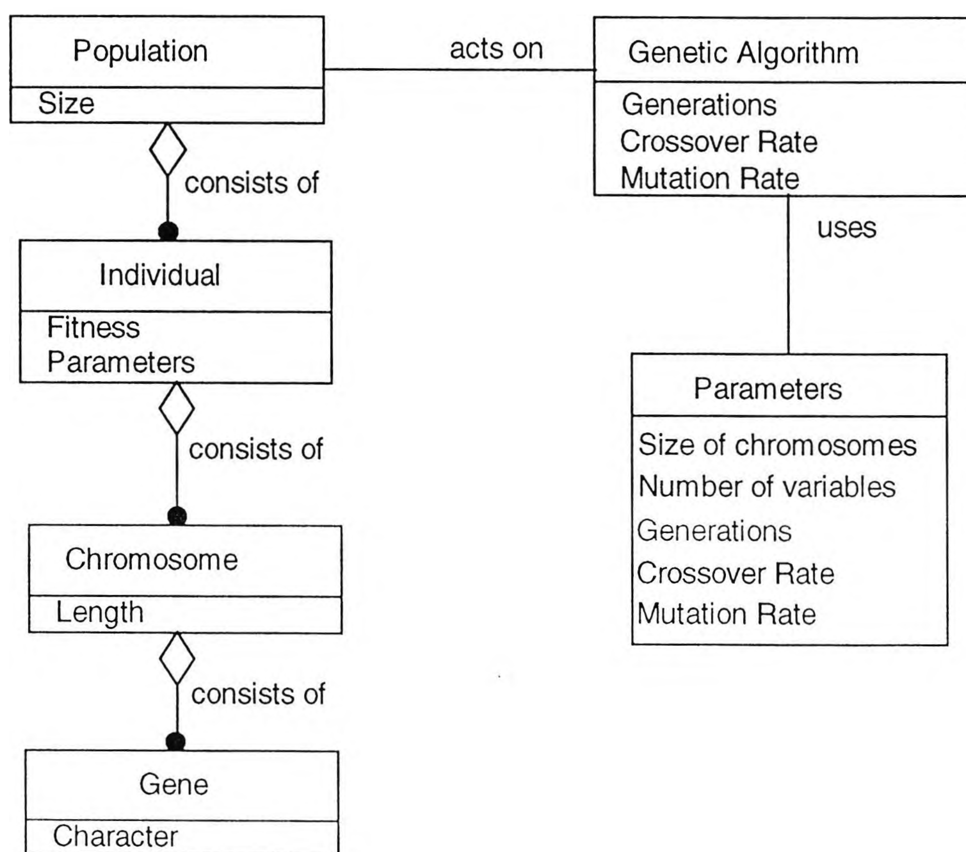


Figure 7.2-6: Genetic Algorithm Domain Object Model

The domain object model shows that a GA system makes use of a Population of Individuals. Each Individual is composed of Chromosomes and have a fitness value. Chromosomes in turn are a certain Length and are made up of Genes. Each Gene has a character which represents the information that the gene encodes.

7.2.3.2 Discovering Object Operations

A dynamic model of the Genetic Algorithm can be created to show the different states that a GA can be in and the events that it responds to. The behaviour of the GA is dependent on the operations that can be performed on it and the events that it can respond to. The dynamic model, enumerates the important states which need to be captured in any GA implementation and hence the operations that are required in the GA interface. Dynamic models are expressed in terms of state-transition diagrams. Figure 7.2-7 shows a state-transition diagram for the Genetic Algorithm.

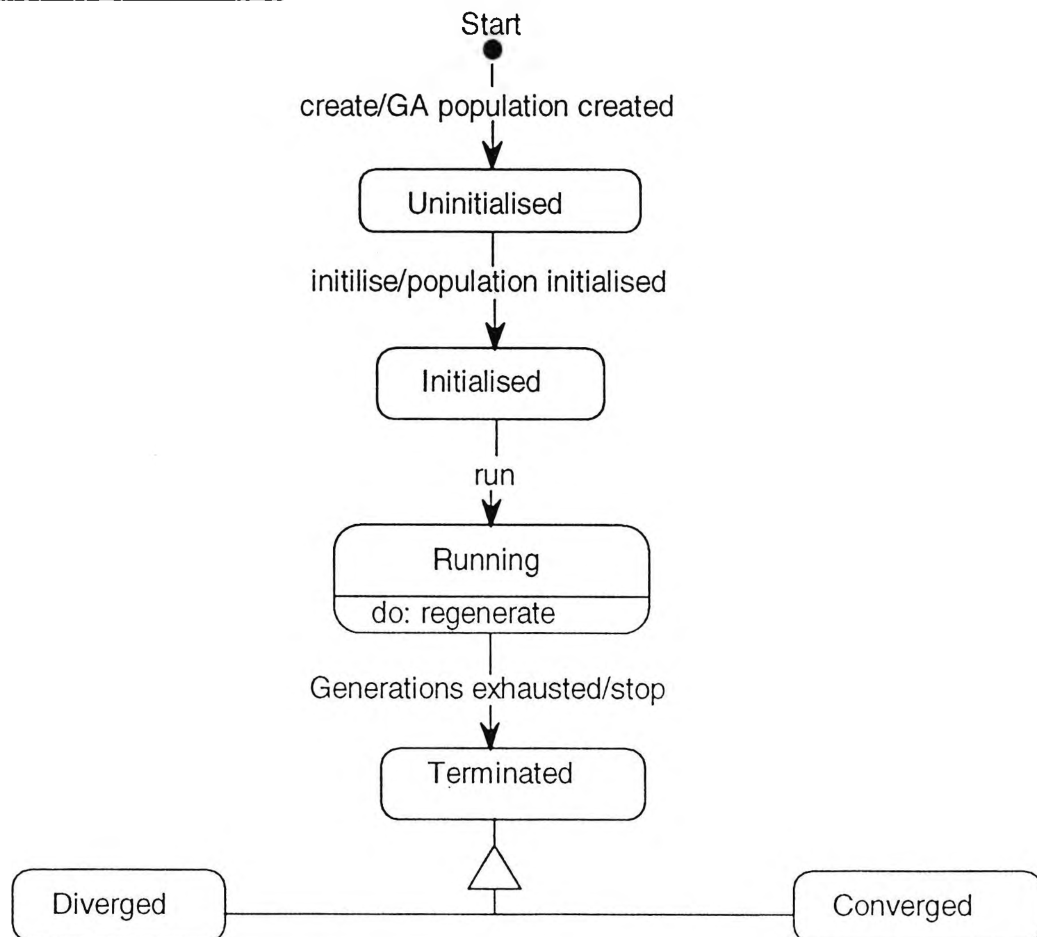


Figure 7.2-7: Genetic Algorithm State Transition Diagram

The different events, *create*, *initialise*, and *run*, will cause a state transition in the GA when they are received. The action parts of the event (where present) will be carried out as a result of the state transition. For example, when a *create* event is received, the GA population is created and the state of the GA changes to *Un-initialised*. On receipt of an *initialise* event, the population is initialised and causes a state transition to *Initialised*. The *Running* state is entered when the *run* event is received. While in the *Running* state, the

regenerate activity takes place to continually regenerate the GA population. When the activity finishes, the state automatically changes to *terminated.* The transition to state *terminated* occurs when the event *generations exhausted* is received. The states, *Converged* and *Diverged* are substates or specialisations of *Terminated.*

7.2.4 Object-Oriented Genetic Algorithm Design

The models developed in the analysis phase are used as inputs to the design phase. The aim of the design phase is to create a computer implementable blue print for the GA system. During object-oriented design, the attributes of the domain objects are determined and the objects themselves are reorganised into class and inheritance hierarchies for easy and efficient implementation. New objects are also added to the analysis domain object model which allow the GA to be constructed efficiently and robustly. The new objects, sometimes described as system objects, are not found in the vocabulary of the problem but enable a modular and reusable system to be built. The design process will also seek to determine efficient data structures and the correct level of granularity for realising the domain objects and the relationships between them. Finally, the system is partitioned into subsystems which can be separately constructed and tested so that the overall complexity of the system is further reduced. The static structure of the system is given by the system object model shown in Figure 7.2-8. The design diagram shows a more detailed object model with the associated relationships. In the design, an Alphabet object is used to represent Genes in the classifier system domain. Alphabets can be a '1', '0' or '#' where '#' represents a 'don't care'. An Array data structure is used to efficiently implement collections. Chromosomes and Vectors are special collections of Alphabets and numbers respectively and so inherit their behaviour from Arrays. Each Individual is made up of a Chromosome and a fitness value. The GA Population consists of zero or more Individuals. The Genetic Algorithm has a *uses* relationship with both the Population and the Parameter object used for collecting GA and problem parameters such as Population size, chromosome string size, number of generations, crossover and mutation probabilities, etc. from the users.

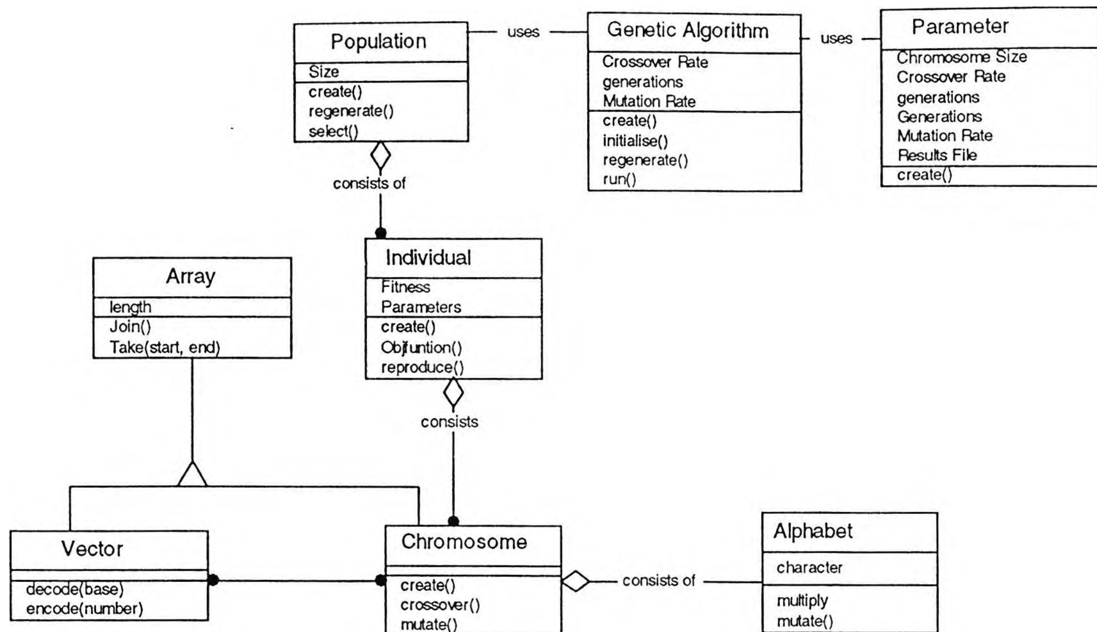


Figure 7.2-8: System object model of Genetic Algorithm

7.2.4.1 Object Design

For the Genetic Algorithm, the system design phase which will require the partitioning of the GA into subsystems is not necessary. It is not easy to visualise a partitioning of the GA into meaningful subsystems that simplify the implementation and can stand on their own. The input subsystem consists of a single Parameter object that requests GA and problem parameters from the User. The remaining portion constitutes what is traditionally regarded as a Genetic Algorithm and so further partitioning is not necessary. Object design is carried out to fully specify the classes and relationships that have been identified so that a solution can be implemented. The objects can be fully specified by creating actual scenarios in which the objects will be used so as to determine the nature of the messages received by the different objects. Two scenarios, CREATE, when the GA is created and RUN when the GA is running are presented to show how the different objects interact. The scenarios are expressed in terms of object interaction diagrams. In the CREATE scenario shown in Figure 7.2-9, a Parameter is initialised with a request for the User to enter the system and problem parameters. A create() call to the GA object results in a create() call to the Population object. The Population in turn issues create() calls for each Individual in the population. The creation of an Individual object leads to a further create() call to the Chromosome object. Finally, the Chromosome calls create() for Array and Alphabet objects to complete the sequence of create() calls. This sequence results in the creation of the GA object at the highest level. In creating the interaction diagrams for a particular scenario, only the very important messages have been enumerated. Furthermore

the messages are at a high level of abstraction so that the design diagrams can be kept simple and focused. Each high level message can be subsequently refined into a number of lower level messages which can be depicted as separate scenarios or Use Cases. At the lowest level, the messages are simply member functions in the objects public interface.

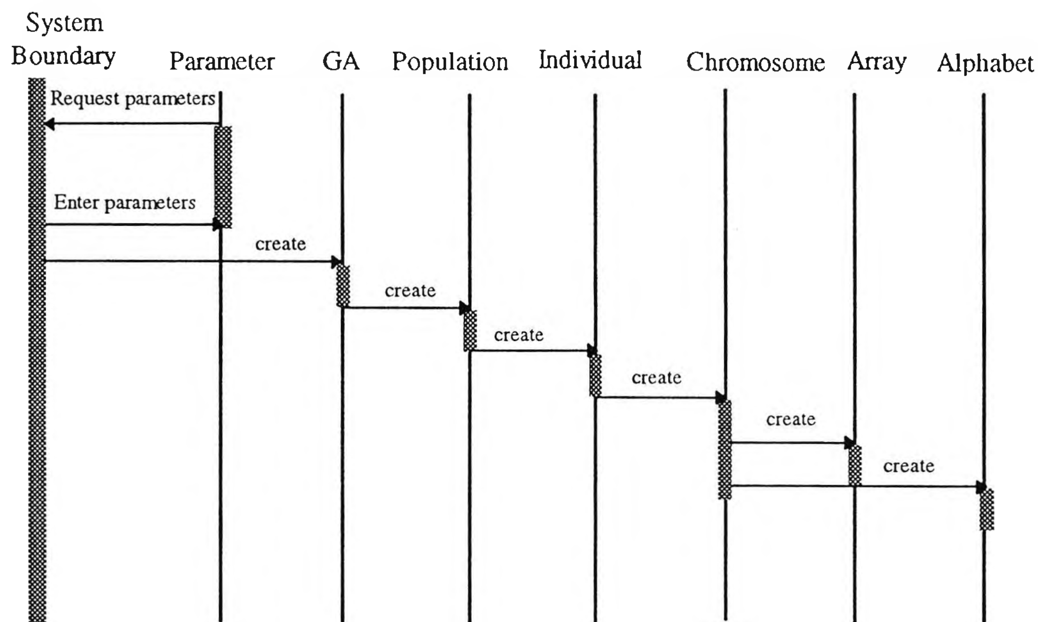


Figure 7.2-9: Object Interaction Diagram for CREATE scenario

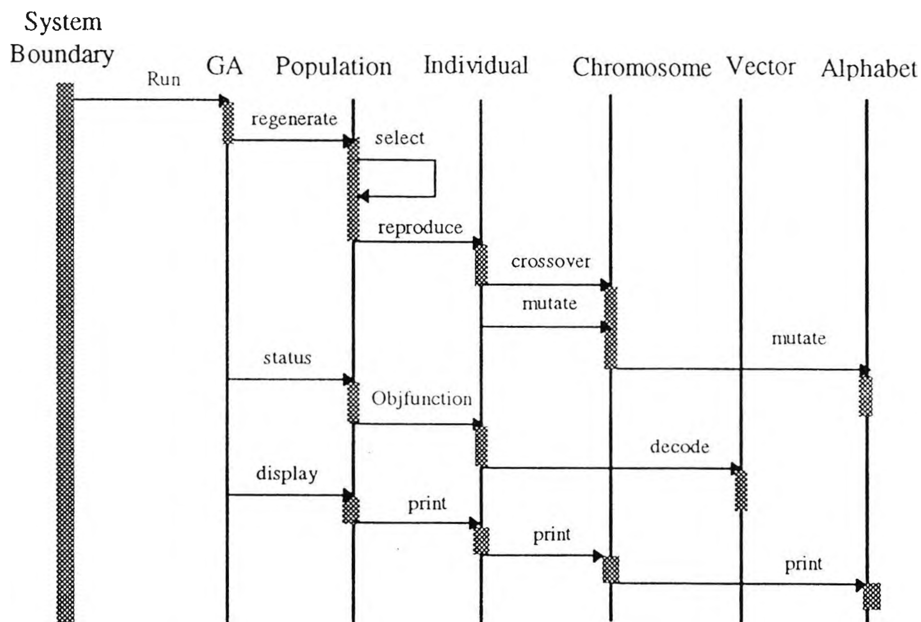


Figure 7.2-10: Object Interaction Diagram for RUN scenario

The running of the GA can be described by a RUN scenario expressed in Figure 7.2-10. When the GA is first run, a regenerate() message is sent to the Population object to create

a new population from the current population. Population issues select() messages to itself to obtain Individuals for reproduction and sends a reproduce() message to Individual. Individual in turn sends crossover() and mutate() messages to Chromosome. To honour any mutation requests, Chromosome has to send a mutate() request to Alphabet which causes a random change in the Alphabet. When the population has been regenerated, a status() message is sent from GA to Population to perform the house keeping. Population then sends an Objfunction() message to Individual to update the fitness values of the new individuals in the population. Individual issues a decode() message to Vector so that the values of their Chromosomes can be decoded as binary vectors. Finally, the results of each generation are displayed from a sequence of display() messages sent between the objects. The interaction diagrams show how the Genetic Algorithm objects co-operate at a lower level to realise the required high level behaviour. The diagrams also show the important messages that each object has to respond to and hence the nature of the member functions required in the objects' public interface. Other member functions are also added to the objects interface to support construction, destruction and general house keeping activities like copying, storage and retrieval for each object. Fully specified structures for Alphabet, Individual and Genetic Algorithm objects are shown in Figure 7.2-11.

Alphabet
character
construct destruct multiply initialise copy isA store retrieve display mutate

Individual
fitness
construct destruct initialise copy reproduce objectivefunction setfitnessfunction store retrieve display

Genetic Algorithm
generations mutation rate crossover rate size
construct destruct initialise regenerate store retrieve status

Figure 7.2-11: Fully specified object structures for some GA objects

7.2.4.2 *Implementation*

The finished design as represented in the design object model (Figure 7.2-8) was implemented using the C++ programming language on a SUN SPARC 10. There is a one

for one translation between the objects in the design object model and classes in the C++ programming language. As an example, the C++ class that represents an Alphabet is given in Figure 7.2-12.

```
static char Alfa[] = { '0', '1', '#' };
class Alphabet {
protected:
        int offset;                                                     // offset in the Alfa array
public:
        void init(Bool msg =False) { offset = rnd(msg); }               // initialise
        void init(int of) {
                assert(of >= 0 && of <= 2);
                offset = of;
        }
        Alphabet(Bool msg =False) { offset = rnd(msg); }                // constructor
        virtual ~Alphabet() {}                                          // destructor
        Alphabet (const Alphabet &Alf) { offset = Alf.offset; }         // copy constructor
        Alphabet& operator=(const Alphabet &Alf)  {                     //Alphabet assignment
                offset = Alf.offset;
                return *this;
        }
        Bool operator==(const Alphabet &Alf) { return (offset == Alf.offset) ? True : False; }
        virtual void print(ostream& o = cout) const  { o << Alfa[offset]; }
        virtual void scan(istream& s = cin)  {
                s >> offset;
                offset = (offset == 0 || offset == 1) ?  offset : 2;
        }
        operator int() const { return offset;  }                        // conversion operator
        void mutate(float p, Bool Msg =False)  { offset = ( flip(p) ? rnd(Msg) : offset); }
        operator char() const { return Alfa[offset]; }                  // conversion operator
        Bool isA() const { return (offset >= 0 && offset <= 2) ? True : False;  }
        friend Alphabet operator*(const Alphabet&, const Alphabet&);    //Alphabet multiplication
};
```

Figure 7.2-12: Sample C++ implementation of Alphabet

Other GA objects have been similarly realised. The C++ language also provides direct support and language constructs for implementing the relationships between the objects. Composition/Aggregation relationships such as that between Population and Individual are realised using pointers and references while inheritance hierarchies are used to implement Generalisation/Specialisation relationships. For example, the object diagram showed that an Array is a general data structure used to implement collections while Chromosomes and Vectors are special cases of Arrays. The relationship between Arrays on the one hand and Vectors and Chromosomes on the other is said to be a Generalisation/Specialisation relationship. Figure 7.2-13 shows a sample implementation of the relationship between Arrays, Vectors and Chromosomes. As a result of the inheritance relationship, data and functions declared in Array are reused in Vectors and

Chromosomes without the need for redefinition. Functionality such as crossover() which is specific to Chromosomes and decode() which is specific to Vectors are then added to Chromosomes and Vectors respectively. Functions that have been defined in the Array class can be refined in the Vector and Chromosome classes to be more specific or more efficient or both. The Array class is called the Base or superclass while the Chromosome and Vector classes are known as Derived or sub classes.

```
template <class T>
class Array {
public:
        virtual void init(const T*, int);
        Array();
        virtual ~Array();
        void print() const;
        virtual Array Join(const Array&, const Array&);
protected:
        int length;
        T *array;
};      /** End of Array class declaration **/

template <class T>
class Chromosome : public Array<T> {
public:
        Chromosome(int, int =1);
        ~Chromosome() { }
        void mutate(float);
friend void crossover(const Chromosome&, const Chromosome&, Chromosome&, Chromosome&);
}; /** End of Chromosome class declaration **/

template <class T>
class Vector : public Array<T> {
public:
        Vector(int);;
        ~Vector();
        int decode(int base);
}; /** End of Vector class declaration **/
```

Figure 7.2-13: Sample C++ implementation for generalisation/specialisation relationships

7.2.5 Object-Oriented Testing

Unit testing is carried out on each GA object to ensure satisfactory behavior. The set of messages that the object needs to respond to, can be simulated and sent to an object independent of the other objects in the Genetic Algorithm. Closely coupled objects and objects in inheritance relationships such as Arrays, Chromosomes and Vectors in the GA are tested together. The test process is thus simplified and predictable. Test results for

Alphabet, Individual and Chromosome objects are shown in Figure 7.2-14, Figure 7.2-16 and Figure 7.2-16 Respectively.

```
eeisun10% talfa
Initial display of both Alphabets
0 0 0 # 1 # # 1 1 1
1 0 # 1 0 1 # # # #
Alphabets after initialisation
Original I : 1 1 0 0 0 0 1 0 1 0
Original II: 1 # 1 0 0 1 # # # 0
Mutation test
Mutation Prob      Original I             Original  II
                   1 1 0 0 0 0 1 0 1 0    1 # 1 0 0 1 # # # 0
   0.001           1 1 0 0 0 0 1 0 1 0    1 # 1 0 0 1 # # # 0
   0.005           1 1 0 0 0 0 1 0 1 0    1 # 1 0 0 1 # # # 0
   0.009           1 1 0 0 0 0 1 0 1 0    1 # 1 0 0 1 # # # 0
   0.01            1 1 0 0 0 0 1 0 1 0    1 # 1 0 0 1 # # # 0
   0.05            1 1 0 0 0 1 1 0 1 0    1 # 1 0 0 1 # # # 0
   0.09            1 1 0 0 0 1 1 0 1 0    1 # # 0 0 1 # # # 0
   0.1             1 1 0 0 0 1 1 0 1 0    1 # # 0 0 1 # # # 0
   0.5             1 # # # 0 1 1 0 1 #    1 0 # 0 1 1 # # # 0
   0.9             0 # 1 # # # 0 # # #    0 1 1 # 1 1 1 # 0 #
   1               1 0 0 # 0 1 0 # 1 0    # # 1 1 1 # 0 0 0 1
Multiplication test!
1 0 0 # 0 1 0 # 1 0
# # 1 1 1 # 0 0 0 1
 Result : 1 1 0 1 0 1 1 1 0 0
testing file input /output
 aArray, bArray, cArray have been written to file alfa.dat
Reading test from file
The alfabets read are
1 0 0 # 0 1 0 # 1 0
# # 1 1 1 # 0 0 0 1
1 1 0 1 0 1 1 1 0 0
eeisun10%
```

Figure 7.2-14: Results of testing the GA Alphabet class

Figure 7.2-14 shows the behaviour of the Alphabet object when the different messages in its interface are issued. Two strings of alphabet are created initially at random. Then the strings are initialised with a user supplied string of characters stored in the test file. Mutation, Multiplication, Storage and Retrieval tests are then carried out. The results confirm that the Alphabet object behaves according to the required specification. As can also be seen, low rates of mutation cause little or no change in the original strings. As the rate of mutation increases, the discrepancies between the original strings and mutated strings also increases confirming that the mutation operation performs satisfactorily.

```
ecisun10% tindividual
 Chromosome  5   4 1 1 1 0 1 0 0 0 1 0 0 1 0 0 0 1 0 0 1 1
 Individual      5   4 1 1 1 0 1 0 0 0 1 0 0 1 0 0 0 1 0 0 1 1    10.5
Individual before
  5   5 0 0 1 0 1 0 0 0 0 0 0 1 0 0 0 0 1 1 0 1 1 0 0 1 0   0.005
Individual After fitness calculation
  5   5 0 0 1 0 1 0 0 0 0 0 0 1 0 0 0 0 1 1 0 1 1 0 0 1 0    0.5
Mutation      Individual
0.001     5   5 0 0 1 0 1 0 0 0 0 0 0 1 0 0 0 0 1 1 0 1 1 0 0 1 0    0.5
0.01      5   5 0 0 1 0 1 0 0 0 0 0 0 1 0 0 0 0 1 1 0 1 1 0 0 1 0    0.5
0.05      5   5 0 0 1 0 1 0 0 0 0 0 0 1 0 0 0 0 1 1 0 1 1 0 0 1 0    0.5
0.095     5   5 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1 1 0 1 1 0 0 0 0    0.5
0.105     5   5 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1 1 0 1 1 0 0 1 0    0.5
0.125     5   5 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 1 0 0 1 0    0.5
0.45      5   5 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 0 1 1 0    0.5
0.7       5   5 1 0 1 0 1 1 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1 0 0 0 0    0.5
0.95      5   5 1 1 1 1 0 0 0 1 0 0 1 1 0 1 1 1 0 0 0 1 0 1 0 1 0    0.5
1         5   5 1 1 0 1 1 1 1 1 0 0 0 0 1 0 1 1 0 1 0 0 0 0 1 1 1    0.5
Stored Individuals in Population
  5   5 0 0 1 0 1 0 0 0 0 0 0 1 0 0 0 0 1 1 0 1 1 0 0 1 0    0.5
  5   5 0 0 1 0 1 0 0 0 0 0 0 1 0 0 0 0 1 1 0 1 1 0 0 1 0    0.5
  5   5 0 0 1 0 1 0 0 0 0 0 0 1 0 0 0 0 1 1 0 1 1 0 0 1 0    0.5
  5   5 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1 1 0 1 1 0 0 0 0    0.5
  5   5 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1 1 0 1 1 0 0 1 0    0.5
  5   5 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 1 0 0 1 0    0.5
  5   5 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 0 1 1 0    0.5
  5   5 1 0 1 0 1 1 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1 0 0 0 0    0.5
  5   5 1 1 1 1 0 0 0 1 0 0 1 1 0 1 1 1 0 0 0 1 0 1 0 1 0    0.5
  5   5 1 1 0 1 1 1 1 1 0 0 0 0 1 0 1 1 0 1 0 0 0 0 1 1 1    0.5
Stored Offspring Individuals in new Population
  5   5 0 0 1 0 1 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 1 0 1    0.5
  5   5 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0 0 1 0    0.5
  5   5 0 0 1 0 1 1 0 0 0 1 0 0 0 0 0 1 1 1 1 0 1 0 0 0 0    0.5
  5   5 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 1 0 0 1 0 1 1 0    0.5
  5   5 0 0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 1 1 0 1 0 0 0 1 0    0.5
  5   5 1 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 1 1 0 1 1 0 0 1 0    0.5
  5   5 0 0 1 0 0 1 0 0 0 0 0 1 0 0 0 0 1 1 0 1 0 0 0 0 0    0.5
  5   5 1 0 1 0 1 0 0 0 0 0 0 1 0 0 0 0 1 1 0 1 1 0 1 1 0    0.5
  5   5 1 1 1 1 0 0 1 0 0 0 1 1 0 1 0 1 0 0 0 1 0 1 0 0 0    0.5
  5   5 1 1 0 1 1 1 1 1 0 1 1 0 1 0 1 1 0 1 0 1 0 0 0 1 0    0.5
ceisun10%
```

Figure 7.2-15: Test results for Individual Object

```
eeisun10% tchrom
Static initialised alfabet: # 0 1 0 # 1 # 1 0 #
0 1 1 0 0 0 1 1 0 1
     /* testing the constructors */
Chromosome aChrom           5   4 0 0 0 0 1 1 0 0 1 1 0 1 1 1 0 1 1 0 1 0
Chromosome bChrom(10)       10   1 0 1 0 0 0 1 1 0 0 1
Chromosome cChrom(5, 4)     5   4 0 0 0 0 1 1 0 0 1 1 0 1 1 1 0 1 1 0 1 0
aChrom  = cChrom            5   4 0 0 0 0 1 1 0 0 1 1 0 1 1 1 0 1 1 0 1 0
Chromosome dChrom(aChrom)        5   4 0 0 0 0 1 1 0 0 1 1 0 1 1 1 0 1 1 0 1 0
Chromosome eChrom = bChrom       10   1 0 1 0 0 0 1 1 0 0 1
Chromosome fChrom                10   1 0 1 1 0 0 0 1 1 0 1
     /* mutatation of chromosomes */
aChrom.mutate(0.001)     5   4 0 0 0 0 1 1 0 0 1 1 0 1 1 1 0 1 1 0 1 0
bChrom.mutate(0.002)    10   1 0 1 0 0 0 1 1 0 0 1
cChrom.mutate(0.003)     5   4 0 0 0 0 1 1 0 0 1 1 0 1 1 1 0 1 1 0 1 0
dChrom.mutate(0.004)     5   4 0 0 0 0 1 1 0 0 1 1 0 1 1 1 0 1 1 0 1 0
eChrom.mutate(0.005)    10   1 0 1 0 0 0 1 1 0 0 1
fChrom.mutate(0.006)    10   1 0 1 1 0 0 0 1 1 0 1
aChrom = xover(aChrom, cChrom, 7, 0.005)   5   4 0 0 0 0 1 1 0 0 1 1 0 1 1 1 0 1 1 0 1 0
dChrom = xover(bChrom, fChrom, 4, 0.001)  10   1 0 1 0 0 0 0 1 1 0 1
parents:
  5   4 0 0 0 0 1 1 0 0 1 1 0 1 1 1 0 1 1 0 1 0
  5   4 0 0 0 0 1 1 0 0 1 1 0 1 1 1 0 1 1 0 1 0
Production offspring
  5   4 0 0 0 0 1 1 0 0 1 1 0 1 1 1 0 1 1 0 1 0
  5   4 0 0 0 0 1 1 0 0 1 1 0 1 1 1 0 1 1 0 1 0
Testing mutation
 Enter starting value for mutation: 0.5
  5   4 0 1 1 0 1 1 0 1 1 1 1 0 1 1 1 1 0 0 0 1   450289
  5   4 0 0 1 0 0 1 0 1 0 1 1 1 1 1 1 1 0 0 0 1   153585
  5   4 0 0 1 0 0 1 0 1 0 1 1 0 1 1 1 1 0 1 1 1   153335
  5   4 0 1 1 1 0 0 1 1 1 1 1 0 1 0 0 1 0 1 0 0   474772
  5   4 1 1 1 1 0 0 1 0 1 1 1 0 1 0 0 1 0 0 1 0   994962
  5   4 1 1 1 0 0 0 0 1 1 1 1 1 0 0 1 0 0 0 1 0   925474
  5   4 1 1 1 0 0 1 0 1 0 1 0 1 0 1 0 0 0 0 1 0   939330
  5   4 1 1 1 0 0 1 1 1 0 1 0 1 0 1 0 0 1 0 1 0   947530
  5   4 1 1 0 0 0 1 1 1 0 1 0 0 1 1 0 1 1 1 1 0   816350
  5   4 0 0 0 0 0 1 1 1 0 1 1 0 0 1 0 1 1 1 0 1   30301
  5   4 0 1 0 0 0 1 1 1 0 0 0 0 0 0 0 1 1 1 0 1   290845
  5   4 0 1 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 1 0 1   323589
  5   4 0 1 0 0 1 1 1 1 0 0 0 1 0 1 0 0 0 1 0 1   323909
  5   4 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 1 1 0   65286
  5   4 0 0 0 0 1 0 1 0 1 1 1 1 0 0 0 0 0 1 1 0   44806
  5   4 0 0 0 0 1 0 1 0 1 1 1 1 0 0 0 1 1 1 1 0   44830
  5   4 0 0 0 0 1 0 1 0 0 1 1 1 0 0 0 1 1 1 1 0   42782
  5   4 1 0 0 0 1 0 1 0 0 1 1 1 0 0 0 1 1 1 1 0   567070
  5   4 1 0 0 0 1 0 1 1 0 1 1 1 0 0 0 1 1 0 1 0   571162
  5   4 1 0 0 1 1 0 1 0 0 1 1 1 0 0 0 0 1 0 1 0   632586
  5   4 1 0 0 1 1 0 1 1 0 1 1 1 0 0 0 1 1 0 1 0   636698
  5   4 1 0 0 1 1 0 0 1 0 1 1 1 0 0 0 1 1 1 1 1   628511
  5   4 1 0 0 1 1 0 0 1 0 1 1 0 0 0 0 1 1 1 1 1   628255
  5   4 1 0 0 1 1 0 0 1 0 1 1 0 0 1 0 1 1 1 1 1   628319
eeisun10%
```

Figure 7.2-16: Results of testing on Chromosome class

After unit testing, integration testing is carried out. All the finished classes are put together as an application to check its operation on real problems and to remove subtle mistakes. Because of the flexible architecture, the GA implementation can be configured to solve a wide variety of searching problems. Options have been included to set the number of objective function parameters, the crossover strategy, the population replacement strategy and even the nature of the selection function for selecting fit individuals from the GA population. Furthermore, the size of the encoding for the parameters in an optimisation problem can be varied to improve the resolution of the search. A sample input screen for the GA is shown in Figure 7.2-17.

GENETIC ALGORITHMS: INPUT SCREEN

GA Population Size:
Number of function parameters:
Encoding Size for each parameter:
GA crossover Probability:
GA mutation Probability:
Number of generations:
Selection Strategy: (R)oulette, (T)ournament, Ra(N)dom :
Crossover Strategy: (O)nepoint, (T)wopoint, (U)niform:
Replace Strategy: (G)eneration, (P)artial, (O)verlap, (E)litist:
Results Filename :

Figure 7.2-17: Genetic Algorithm input screen showing the different options

The Genetic Algorithm has been tested on both simple and complex function optimisation. In the simplest example, the Gaussian function shown Equation 7.2-, is used as the objective function. The function has a mean value of 2 and a standard deviation of 1.

$$f(x) = \tfrac{1}{2} e^{-(x-2)^2} \qquad (7.2\text{-}1)$$

The function is unimodal and has a singular maximum value of 1 at x = 2 as shown in Figure 7.2-18.

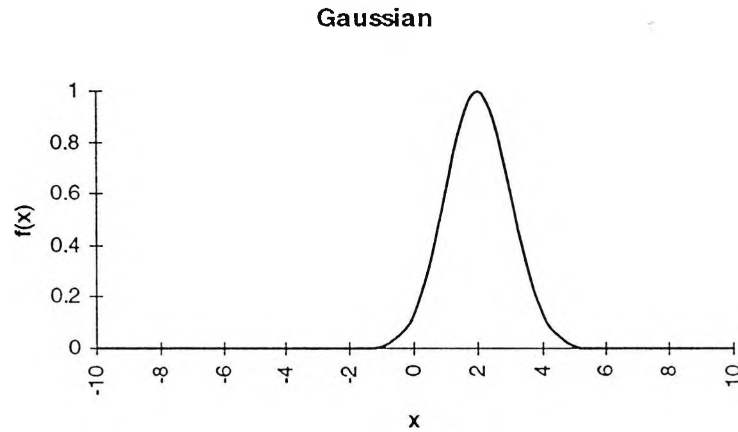


Figure 7.2-18: Plot of Gaussian Objective Function

The GA was configured to search for this maximum in the range -10 to 10. The parameter, x, is encoded as 16 bit string. The fitness values are obtained by evaluation of the decoded strings. The decoded values are scaled according to Equation 7.2-2 so that they are between -10 and 10 before being passed to the Gaussian function for evaluation.

$$fitness = f\left(-10 + \frac{decoded\ value}{2^{16}-1} * 20\right) \qquad (7.2\text{-}2)$$

The population is initialised with 10 individuals at random. The GA is then run a number of times with the simulation parameters varied in each run. The results are compared with random population search where the individuals in each generation are selected at random from the population and only the mutation operator is used throughout the run of the GA to evolve new individuals. This is achieved by setting the crossover and mutation rates to 0 and 1 respectively and choosing either Random selection, Roulette wheel selection or Tournament selection strategy. Figure 7.2-20 and Figure 7.2-21 show how the maximum and average fitnesses vary from one generation to another for random search with Roulette wheel and tournament selection respectively. The equivalent graph with completely Random selection is shown in Figure 7.2-21. The total fitness which represents the sum of all fitness values in the population is also shown. The graph demonstrates that for random search, the total fitness of the population can be completely dominated by the fitness of the best individual and so the search process will never converge and has to be stopped arbitrarily. Furthermore, the individual with the maximum fitness in the final generation represents a solution for the search problem and for random search, this can be far from optimal as generational replacement means that the best solution in each generation is not retained.

For GA search, experiments have been carried out using different values for crossover, mutation, population selection strategy, chromosome crossover strategy and population replacement strategy. Figure 7.2-22 - Figure 7.2-27 are example plots that show how the average and maximum fitness values in each generation varies with different parameter selections.

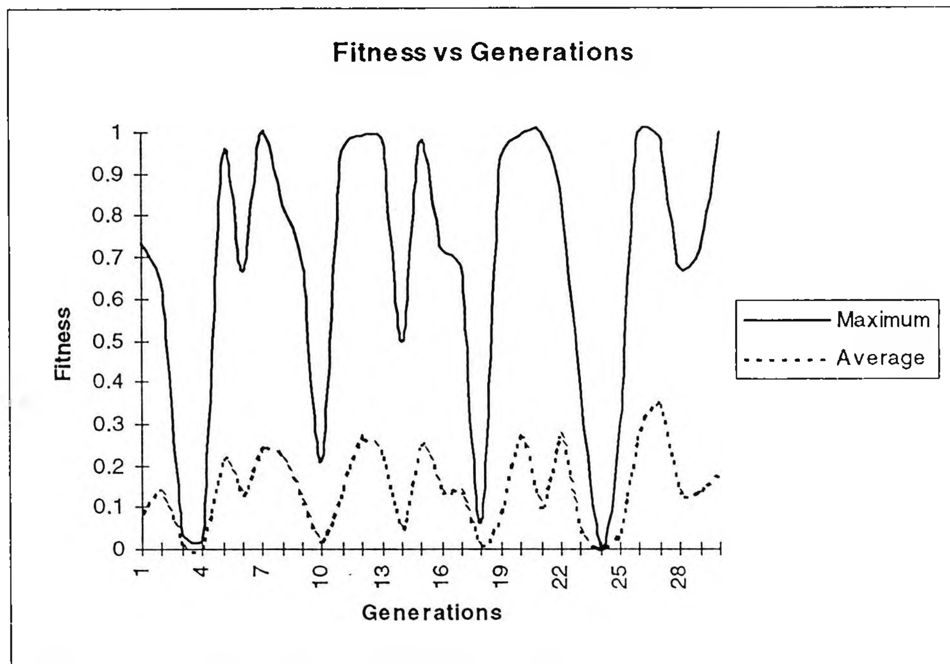


Figure 7.2-19: Maximum Fitness vs. Generation for random population search (crossover =0, mutation =1, Roulette wheel selection, Generational replacement)

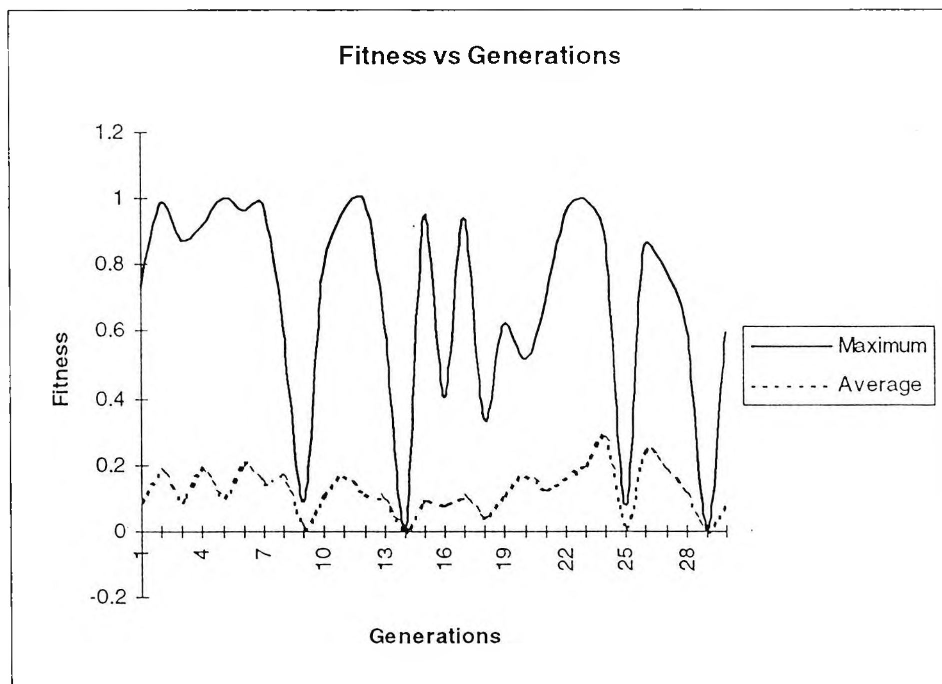


Figure 7.2-20: Maximum Fitness vs. Generation for Random search (crossover =0, mutation =1, Tournament selection, Generational replacement)

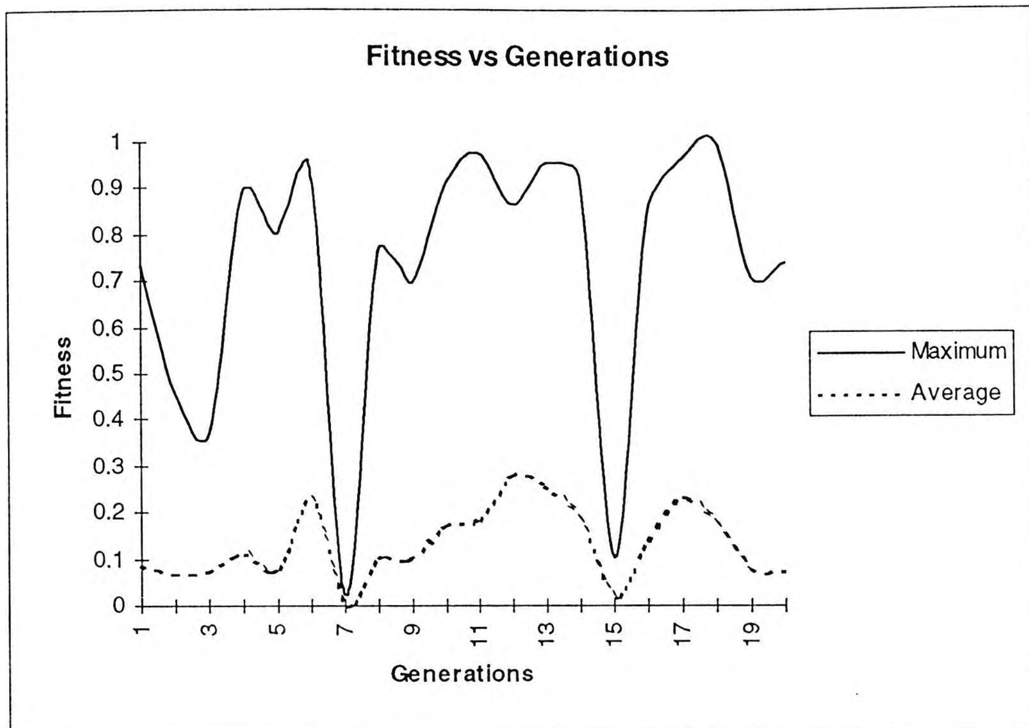


Figure 7.2-21: Variation of Fitness vs. Generation for Random Search (crossover = 0, mutation = 1, Random selection, Generational replacement)

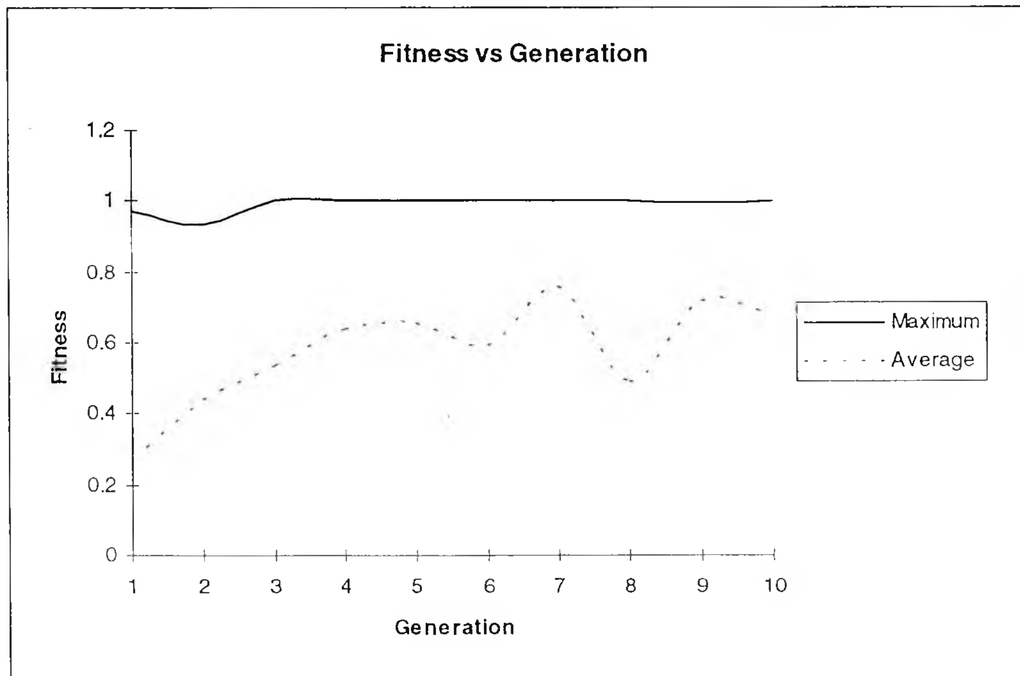


Figure 7.2-22: Variation of Maximum and Average Fitnesses with Generation (crossover = 0.9, mutation = 0.1, Roulette selection, one point crossover, Generational replacement)

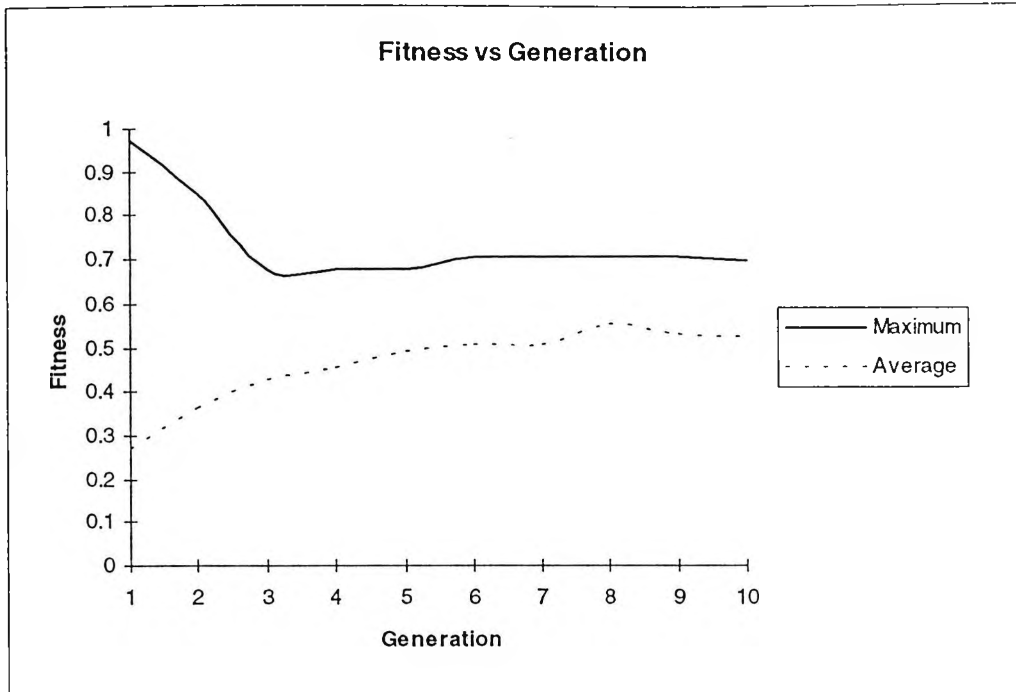


Figure 7.2-23: Variation of Maximum and Average Fitnesses with Generation (crossover = 0.9, mutation = 0.01, Roulette selection, two point crossover, Generational replacement)

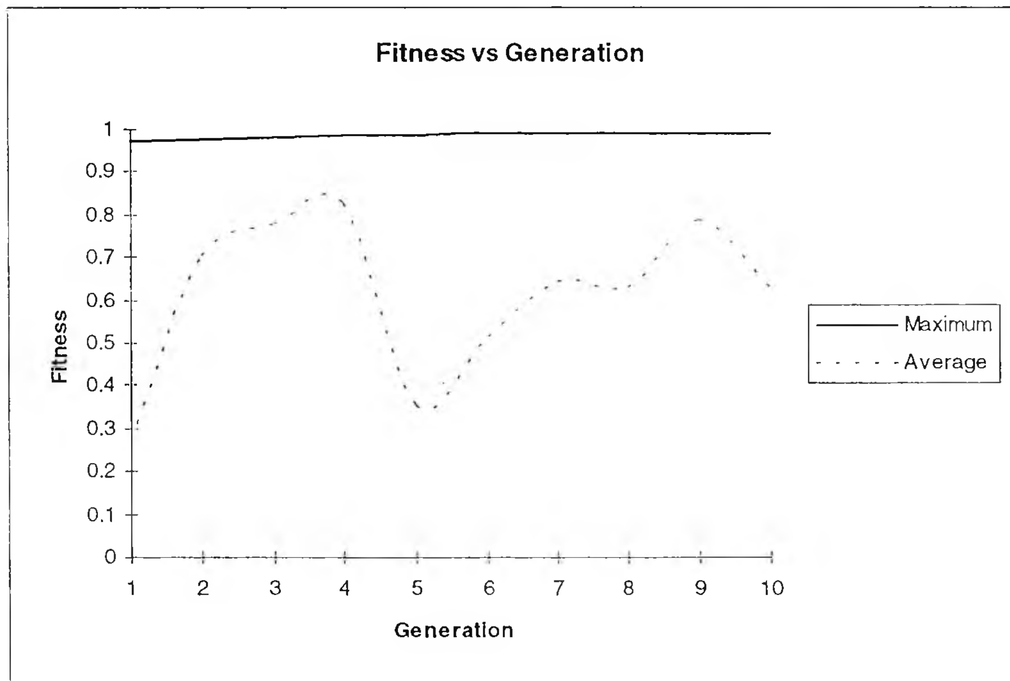


Figure 7.2-24: Variation of Maximum and Average Fitnesses with Generation (crossover = 0.7, mutation = 0.1, Tournament selection, two point crossover, Generational replacement)

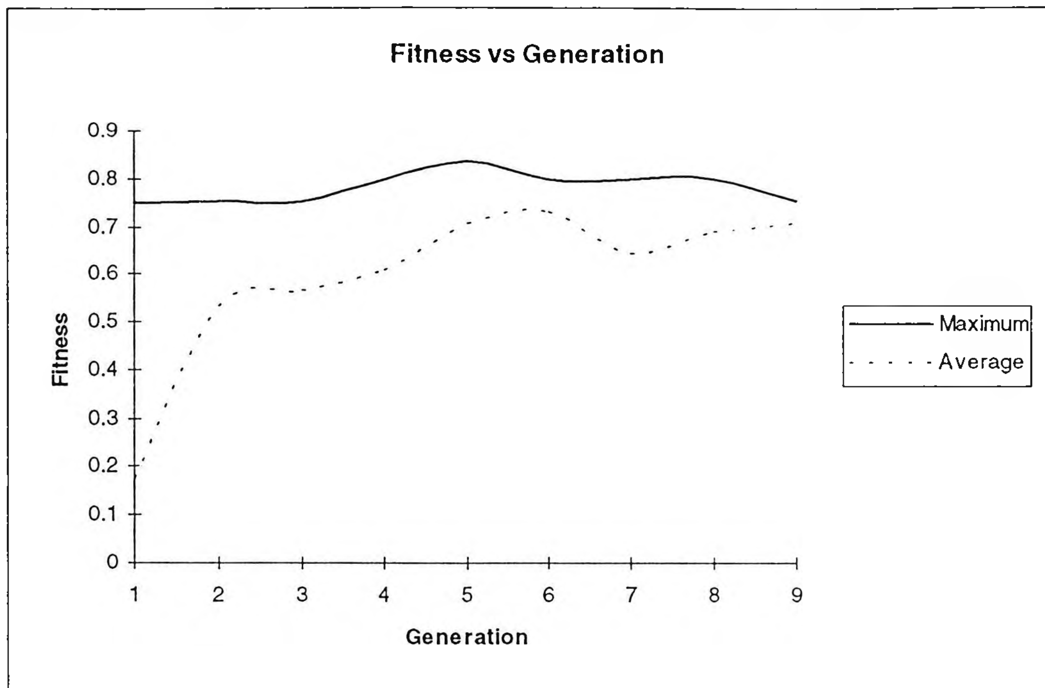


Figure 7.2-25: Variation of Maximum and Average Fitnesses with Generation (crossover = 0.7, mutation = 0.1, Roulette selection, two point crossover, Partial replacement)

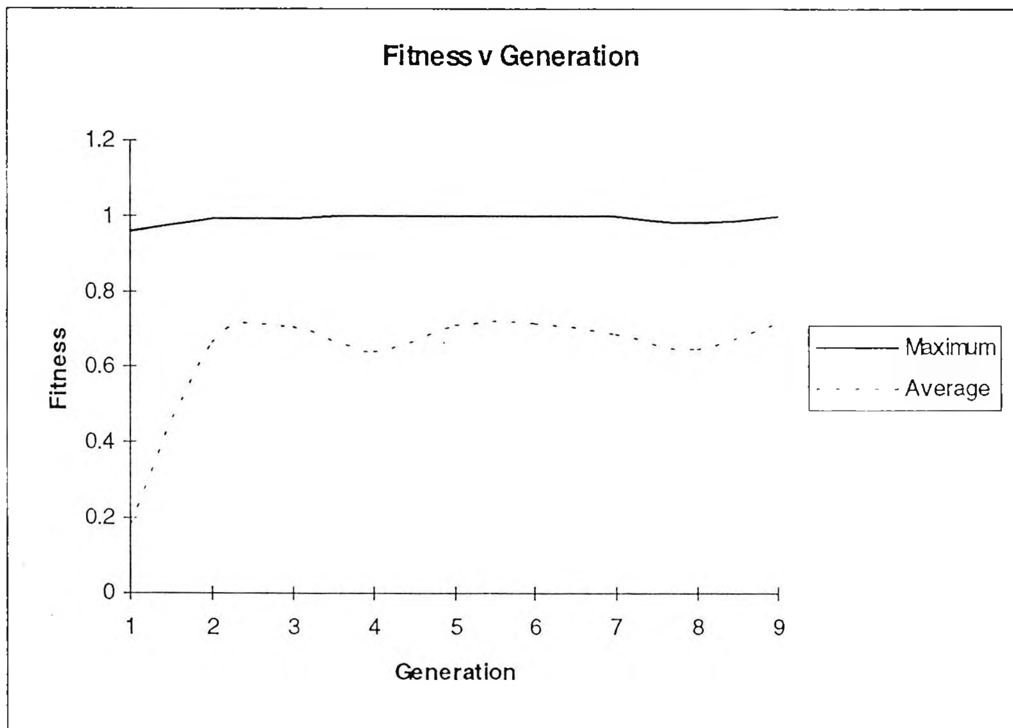


Figure 7.2-26: Variation of Maximum and Average Fitnesses with Generation (crossover = 0.9, mutation = 0.1, Tournament selection, two point crossover, Elitist replacement)

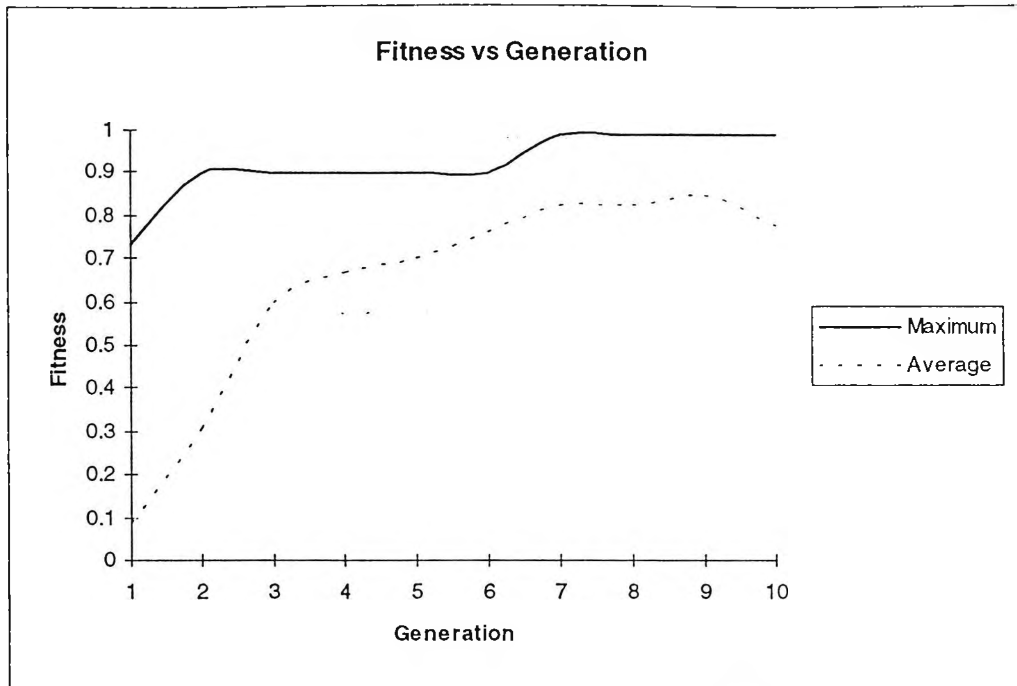


Figure 7.2-27: Variation of Maximum and Average Fitnesses with Generation (crossover = 0.9, mutation = 0.1, Tournament selection, uniform crossover, Generational replacement)

The graphs confirm that, for this simple function, the GA can successfully converge to an optimal solution which is very close to the required solution in only 10 generations. The convergence characteristics are very similar, irrespective of the method of selection and the crossover and mutation rates despite the fact that the search space is very large. As expected, the average fitness of the population is very low initially since the population is initialised at random. The GA evolves increasingly fitter individuals in subsequent generations and the average fitness of the population thus increases. In some cases, mutation and crossover actually produce less fit individuals in the next generation and this results in a fall in the population average fitness. With random search, it is possible to find optimal or near optimal solutions in a very short time, but as is clearly evident in Figure 7.2-20 the average fitness of the population remains very low. Furthermore, there is a distinct possibility that the optimal solution will not be found.

In the next example, the GA is used to search for an optimal value in a space where the objective function is more complicated. The objective function consists of a Gaussian mixture contaminated by noise. The noise function is also a Guassian mixture with similar means as the original mixture but with one tenth the amplitude and five times the standard deviation of the mixture components. The objective function is expressed as shown in Equation 7.2-3.

$$f(x,y) = 0.9 * \frac{1}{\sqrt{2\pi\sigma_1^2}} \exp\left(-\frac{(x-\mu_1)^2}{\sigma_1^2} \right) * \frac{1}{\sqrt{2\pi\sigma_2^2}} \exp\left(-\frac{(y-\mu_2)^2}{\sigma_2^2} \right) +$$
$$0.1 * \frac{1}{\sqrt{2\pi(5\sigma_1)^2}} \exp\left(-\frac{(x-\mu_1)^2}{(5\sigma_1)^2} \right) * \frac{1}{\sqrt{2\pi(5\sigma_2)^2}} \exp\left(-\frac{(y-\mu_2)^2}{(5\sigma_2)^2} \right) \quad (7.2\text{-}3)$$

where μ_1 and σ_1 are the mean and variance, of the first mixture component

μ_2 and σ_2 are the mean and variance of the second mixture component.

A plot of the objective function is shown in Figure 7.2-28. The GA is configured to search the two dimensional space represented by the objective function for the maximum value. The aim of the search is to find two real values x and y in the range 0...5 representing the co-ordinates of the maximum point of the objective function. Figure 7.2-29 shows how the objective function behaves in the vicinity of the maximum. A numerical solution using Mathematica shows that the function has a single maximum at 6.39449, when x = 2 and y = 3. The objective function then falls off very rapidly to zero all around the maximum.

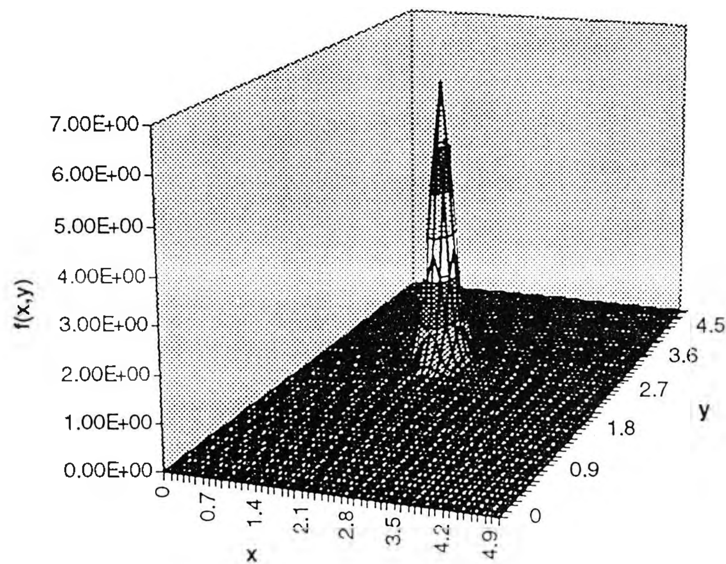


Figure 7.2-28: Objective function plot for u1 = 3, u2 = 2, and s = 0.15

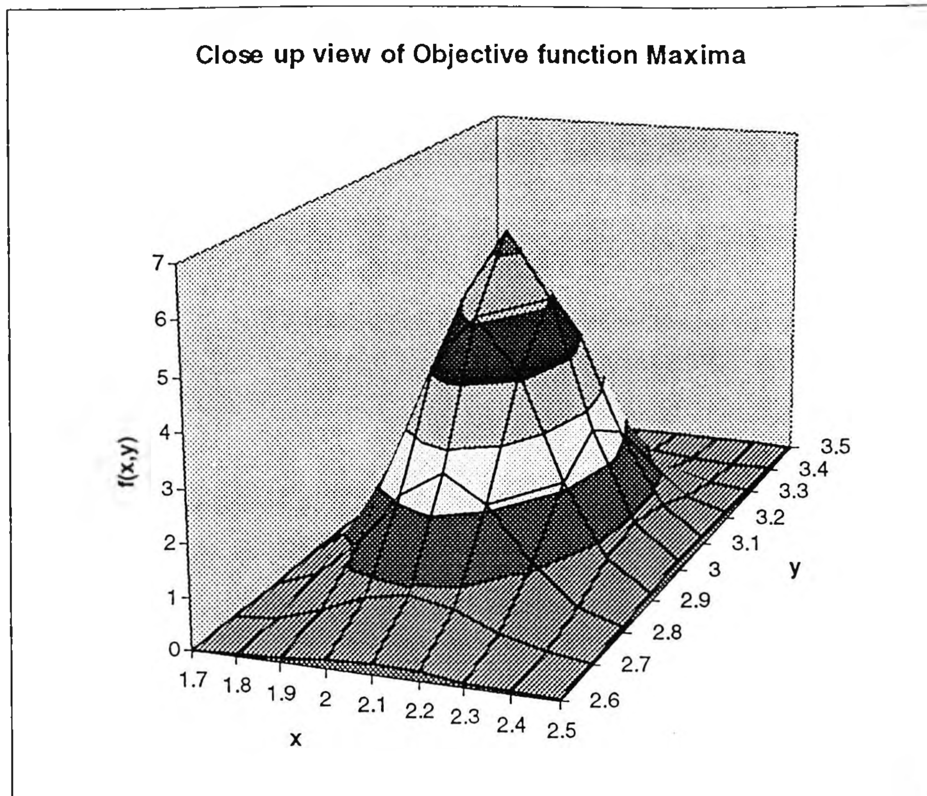


Figure 7.2-29: Behaviour of the objective function in the neighbourhood of the function maximum.

In the GA, both x and y are encoded as 20 bit strings which are then concatenated to form a 40 bit chromosome for each individual. To calculate the individual's fitness value, each chromosome is split into its two constituent parts which are then decoded and scaled according to Equation 7.2-4 into the x and y values. The values obtained are evaluated by the Gaussian mixture function to obtain the fitness of the individual.

$$x = y = \left(\frac{decoded_value}{2^{20} - 1}\right) * 5 \qquad (7.2\text{-}4)$$

The GA is initialised at random with a population of 30 individuals. The number of generations is set to 30 and the GA is then run a number of times with different values for the GA parameter, i.e., mutation rate, crossover rate and population replacement strategy. As with the previous objective function, the results are compared with a random population search of the function space where the crossover rate is set to 0, the mutation is set to 1 and a Random selection strategy is used. The population replacement strategy can either be Generational or Elitist. Figure 7.2-30 shows how the maximum and average fitnesses vary when random population search is used in conjunction with Generational replacement. As mentioned before, Generational replacement completely replaces the GA population in each generation and because no information is passed from one generation

to another, the search will never converge. In Figure 7.2-31 random population search is carried out in conjunction with Elitist population replacement, i.e., the best individual in each generation is retained in the next generation. The figure shows the random population search can still arrive at an optimal or near optimal solution if the best individual in each generation is retained, but the variation of total and average fitnesses on a per generation basis is still very erratic.

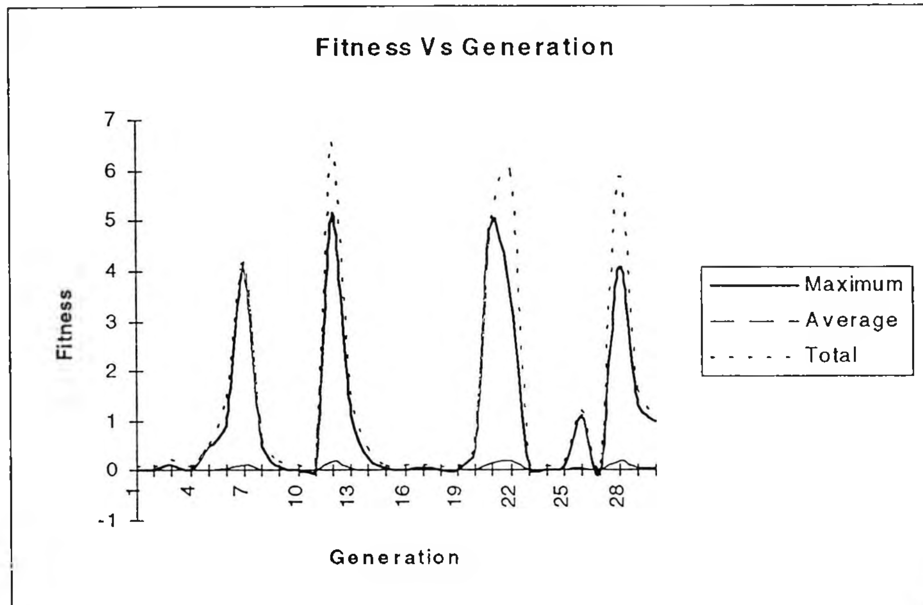


Figure 7.2-30: Random search with Generational replacement

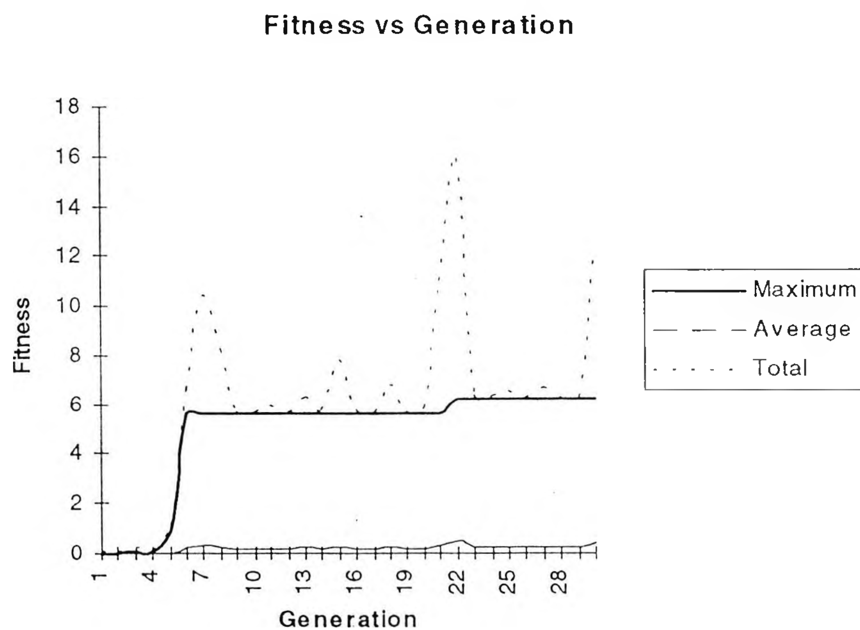


Figure 7.2-31: Random search with Elitism

Figure 7.2-32 - Figure 7.2-34 show the equivalent graphs for Genetic Algorithm search. The convergence of the fitness and x and y values with generation are also tabulated in Table 7.2-2. In the graphs, the total fitness has been left out so that the Maximum and Average fitnesses are visible.

Table 7.2-2: Best of Generation Fitness and (x, y) values vs. Generation number for Genetic Algorithm search.

Gen	Figure 7.2-32	Figure 7.2-33	Figure 7.2-34	Figure 7.2-34 (x)	Figure 7.2-34 (y)
1	0.728291	0.025201	0.506987	2.4353	1.89206
2	4.17134	0.027658	1.08185	3.3728	1.89084
3	4.04754	2.20231	1.68198	2.79052	1.89077
4	5.97665	2.54161	3.18862	3.13842	1.89046
5	5.87153	4.6623	4.83984	3.13842	1.89298
6	6.36692	5.51842	5.0433	3.1042	2.0482
7	6.37606	5.66355	5.56414	3.09444	2.04091
8	6.26938	6.19981	5.78558	3.09444	2.0471
9	6.3662	6.34124	5.83347	3.02551	1.9711
10	6.36927	6.34922	6.00828	3.02551	1.96775
11	6.37617	6.35054	6.33573	3.02427	1.96716
12	6.39095	6.38343	6.28355	3.02512	1.96653
13	6.38138	6.38343	6.29152	3.02591	1.96896
14	6.38314	6.39276	5.89751	3.02608	1.96896
15	6.38754	6.39407	6.04052	2.98702	1.96896
16	6.38754	6.39407	6.04052	3.0228	1.98669
17	6.38478	6.39445	5.8	2.98642	1.98669
18	6.39216	6.39419	5.8	2.98642	1.98648
19	6.39137	6.39436	6.05812	2.98642	1.98648
20	6.39371	6.39438	5.98166	2.98642	1.98667
21	6.39244	6.39438	6.00832	2.98642	1.98632
22	6.3937	6.39438	5.88713	3.01687	1.98667
23	6.39369	6.39447	6.09555	3.0261	1.98423
24	6.39373	6.39447	5.91747	3.01503	2.00948
25	6.39253	6.39448	6.15635	3.01621	2.01714
26	6.39337	6.39447	5.98371	3.01621	2.01714
27	6.39323	6.39446	6.12896	3.01507	2.00204
28	6.3938	6.39386	5.99622	3.01684	1.99147
29	6.39382	6.39447	6.17239	3.02549	2.00296
30	6.39386	6.39447	6.20304	2.98658	2.00376

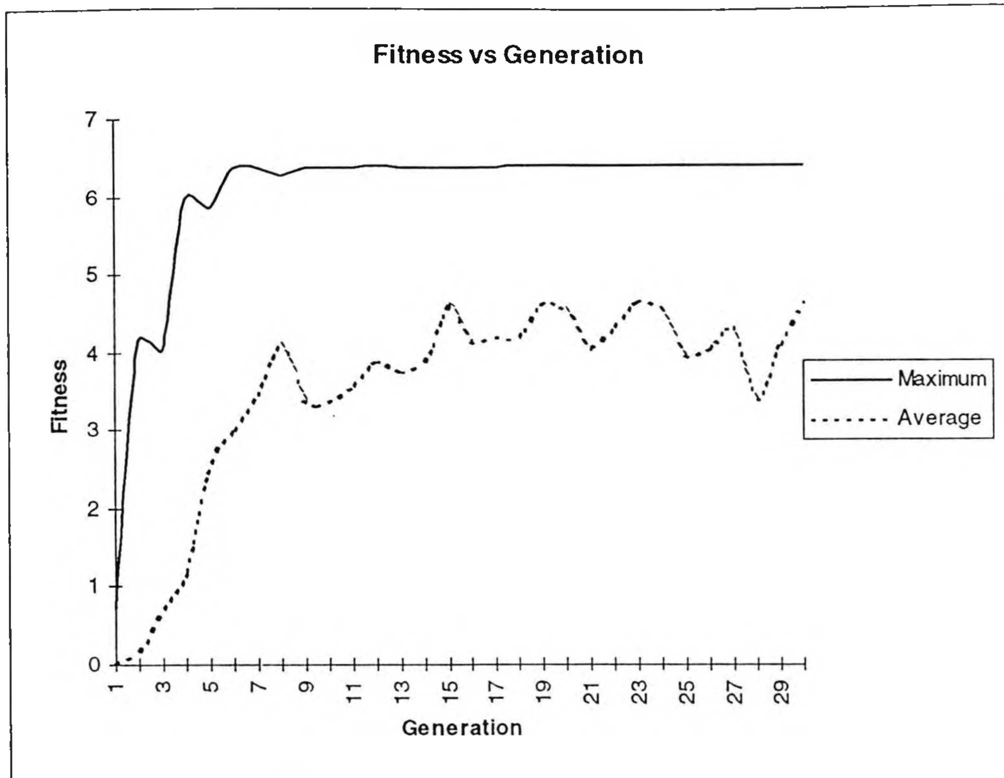


Figure 7.2-32: Variation of Maximum and Average Fitness values with Generation (Crossover = 0.8, mutation = 0.1, Roulette selection, Two point crossover, Generational replacement)

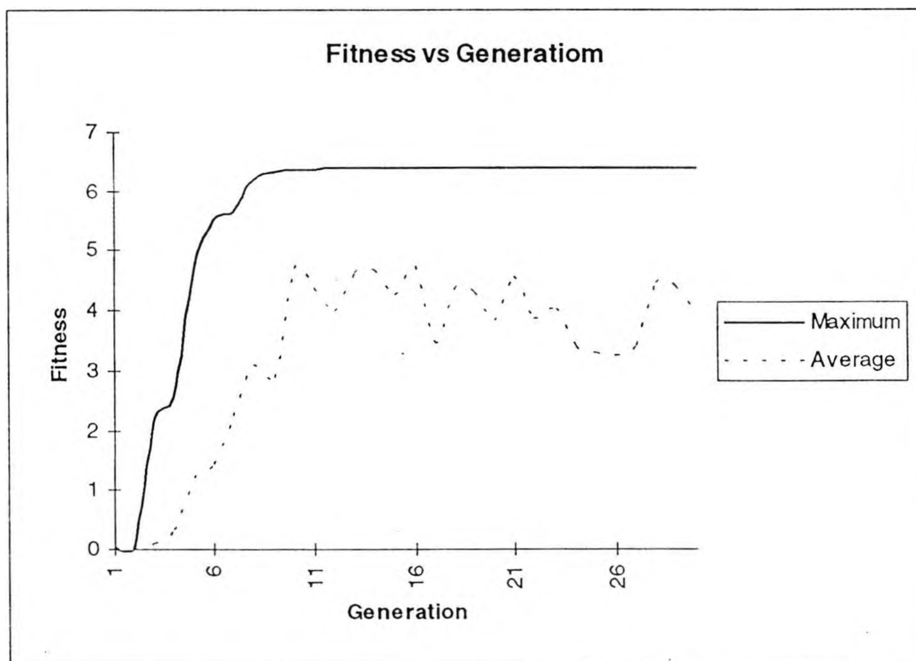


Figure 7.2-33: Variation of Maximum and Average Fitness values with Generation (Crossover = 0.8, mutation = 0.1, Tournament selection, twopoint crossover, Generational replacement)

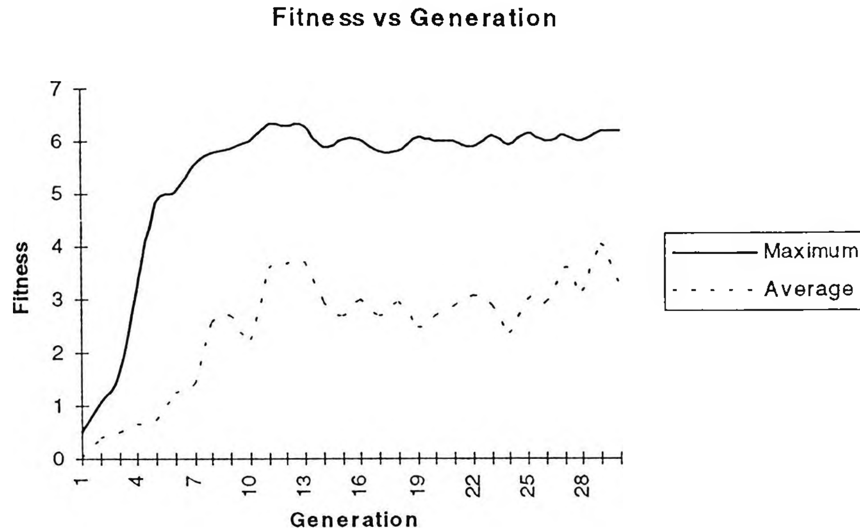


Figure 7.2-34: Variation of Maximum and Average Fitness values with Generation (Crossover = 0.8, mutation = 0.1, Tournament selection, one point crossover, Generational replacement)

The results show that random search with Generational replacement has a very small probability of arriving at an optimal solution at the end of 30 generations. This is because, no information is stored between generations and the search results in the 30th generation are just as likely as those in the first generation. On the other hand, when Elitism is added to the Random search, the best individual in each generation is retained and there is thus a significant performance improvement in the search process. However, the chances of arriving at an optimal solution within 30 generations is still very small. For a search resolution accurate to two decimal places, the search space contains 250000 points. If the resolution is increased to 3 decimal places, the cardinality of the search space increases to 25 million points. Also, if either the search range or the search resolution is increased, the number of points can easily become unbounded. It becomes almost impossible for a brute force search method such as random search to locate a single point in such large search spaces. GA search on the other hand converges to an optimal or near optimal solution in a very short time. Figure 7.2-32 - Figure 7.2-34 show the equivalent graphs for Genetic Algorithm search. The convergence of the fitness and x and y values with generation are also tabulated in Table 7.2-2. In the graphs, the total fitness has been left out so that the Maximum and Average fitnesses are visible.

Table 7.2-2 show how the genetic search converges to an optimal solution in 30 generations for 3 different runs of the GA. The table also shows how the decoded values for the x and y co-ordinates of the maximum vary with generation for the third run of the GA. In Figure 7.2-32 the GA converges to a final value of 6.39386. This value is identical

to three decimal places to the analytically derived optimum value and is just 0.00063 or 0.000985% different than the maximum. In the second run shown in Figure 7.2-33 the search results are even more impressive. The combination of Tournament selection, two point crossover and Generational replacement causes the GA to converge to within 0.00002 of the required optimum value. In the third run in Figure 7.2-34, tournament selection combined with one point crossover and Generational replacement causes a convergence to a sub-optimal value. Even here, the converged GA is within 2.1% of the required value and can be considered as near optimal for most purposes. The optimality of the solution can be appreciated when the variation of the best of generation parameters are examined. The optimal values for x and y are 2 and 3 respectively. As shown in Figure 7.2-35, the error in the decoded values for x and y are relatively large at the start of the GA run. As the GA converges, the values tend to stabilise with very little fluctuation around the optimal values.

The above examples demonstrate that the GA implementation is capable of successfully searching higher dimensional spaces in the solution of complex optimisation problems and that the results are significantly better than random or brute force search. The test results of applying the Genetic Algorithm to determine the optimal parameters of an excitation control system and neural network parameter selection have been presented in [127, 128, 129]. The next section describes how the Genetic Algorithm design is reused in developing a Learning Classifier System.

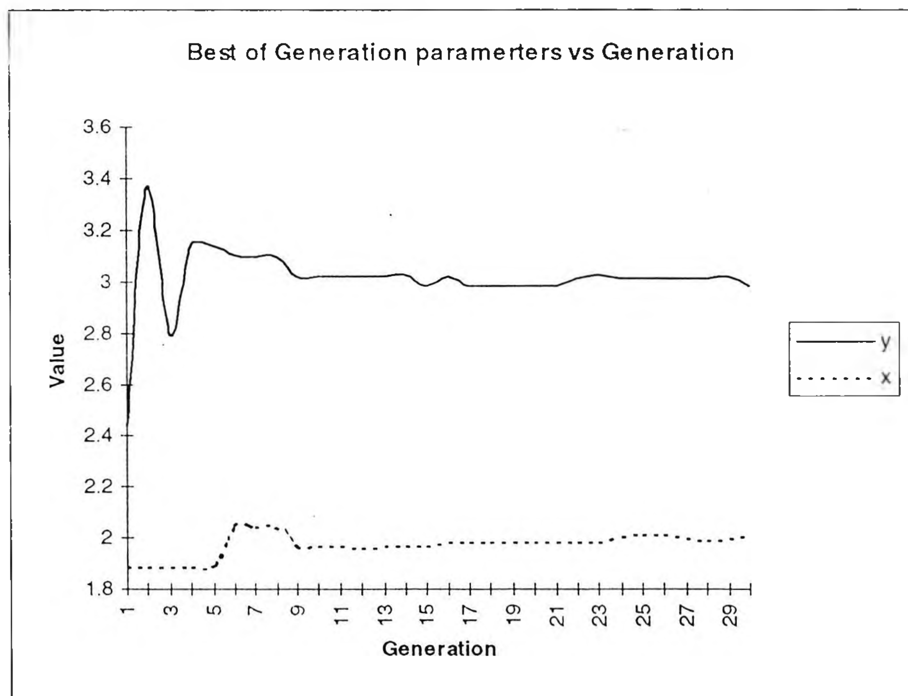


Figure 7.2-35: Variation of decoded parameters with Generation.

7.3 Genetic Learning Classifier Systems

This section presents a new way of constructing Genetic Learning Classifier Systems using object-oriented techniques. Object-oriented analysis and design techniques are applied to the construction of a robust software architecture for Classifier Systems. The design demonstrates how a high level of reuse can be achieved in object-oriented software construction by reusing the GA design and incorporating features necessary for Learning classifier systems. Also, the basic classifier system is extended using Object-Oriented techniques to allow for inexact matching [130] by extending the classifier grammar to include a multiplication operator. This simplifies the classifier matching and bidding processes. The matching process is reduced to a simple inner product calculation between the classifier string vectors and message string vectors. This procedure is very similar to the inner product multiplication performed between weights and activations in connectionist models and also the fuzzy t-norms and t-conorms that take place during the inference process in fuzzy models. This brings classifier systems closer in terms of their construction and operation to other intelligent systems.

7.3.1 Introduction

Knowledge representation is an important problem domain in which the parallelism of classifier systems can be applied. Artificial intelligent techniques have traditionally been applied to static off-line problems where the statistics of the problem domain remain constant or vary only slowly throughout the problem solving process. Current technology for building knowledge-based systems lend themselves only partially to continuos real-time operation in a dynamic environment. Reliable real-time operation is difficult to achieve because the performance of knowledge-based systems can vary dramatically with problem configurations [131]. Classifier systems can, in theory, be used in the solution of problems requiring real-time interactions with ongoing processes. This section presents the analysis, design and implementation of a learning classifier system of the kind discussed in [123]. In the implementation, the basic classifier architecture is modified to allow for inexact matching and population wide bidding. Also, the basic classifier grammar has been extended using object-oriented techniques to concretise the abstract classifier alphabet into an object incorporating a multiplication method to perform simple alphabet matching. Closure is achieved by ensuring that multiplication produces results which are in the scope of the classifier grammar.

7.3.2 *Classifier Systems*

A classifier system is a machine learning system that learns syntactically simple string rules (called classifiers) to guide its performance in an arbitrary environment.

The classifier system is made up of 2 main components as shown in Figure 7.3-1

- Rule and message subsystem,
- Learning subsystem

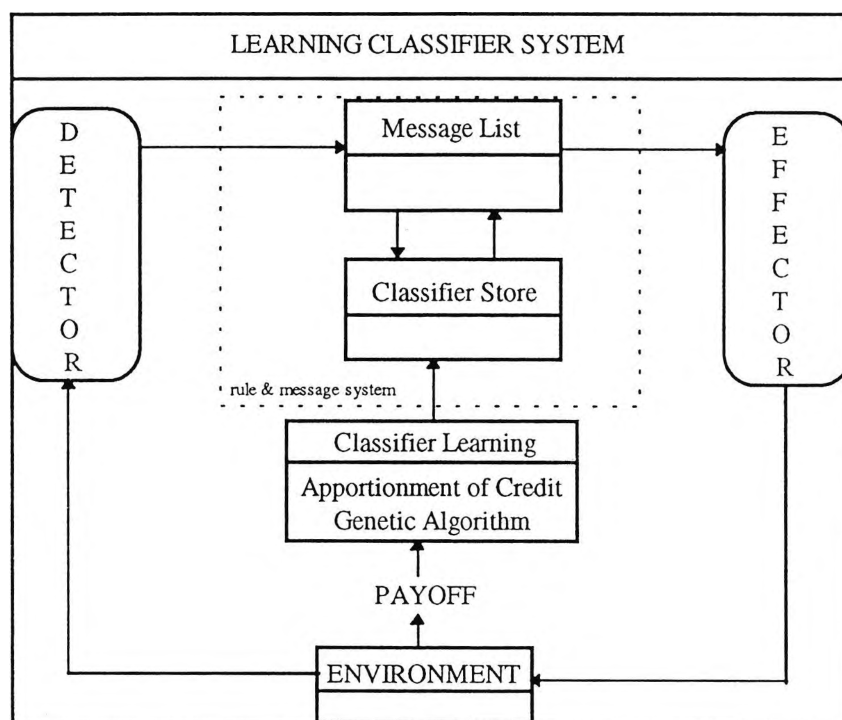


Figure 7.3-1: Schematic Diagram of a Learning Classifier System

The rule and message system is a specialised production system. A production system is defined by a set of rules (productions) that form the production memory and a database of current assertions called the working memory [132]. The production rules are of the form :

if <condition> *then* <action>

The condition part of each rule contains pattern elements which are matched against the working memory while the action part contains directives for updating the working memory. An update of the working memory consists of adding and removing facts about the current state of the system and also includes a directive for effecting external side effects. Classifier systems use large but finite global message lists but at any one time, only a small proportion of these are active [133]. The Learning subsystem is made up of an apportionment of credit subsystem and a Genetic algorithm subsystem. The

apportionment of credit subsystem comprises the bucket brigade algorithm which apportions credit amongst competing classifiers in order to distribute external reward to the rules that contribute to successful behaviour. The bucket brigade algorithm assigns a strength to each individual classifier, modifying the strength on the basis of the classifier's overall usefulness as the system evolves. The genetic algorithm subsystem is in charge of creating new classifiers. The genetic algorithm is used to search the space of existing classifier rules to produce new rules which are innovative combinations of the existing rules.

Classifier systems are able to achieve massive parallelism through a parallel rule matching process while implicit parallelism occurs when many classifiers share the same message.

7.3.3 *Learning in Classifier Systems*

Learning algorithms control the action of classifier systems by controlling write access to the Message list and/or by controlling which classifiers are in the database of rules [134]. Access to the Message list is controlled by placing an upper bound on the number of messages that can be active at any one time. The classifiers that are potentially active then bid for the right to post their messages on the Message list. A competition is held after which the winners are allowed to post their messages. Various factors associated with each classifier determine which classifiers get to post messages. These factors include; the strength of the classifier, the support due to previously matched classifiers and the specificity of the classifier's condition(s). Specificity is scale factor that represents the number of '#' ('don't cares') in a classifier's condition string. The bucket brigade algorithm adjusts the strength of the classifiers over time, rewarding those classifiers that have contributed to good solutions and penalising those that have not proved useful. The Genetic algorithm controls the choice of classifiers that are in the database or classifier store. The GA is used periodically throughout the operation of the classifier system to evolve the classifier base by introducing new rules into the system. The GA evaluation function uses the strength of each classifier as the "fitness" measure to eliminate weak classifiers from the database. New classifiers are generated from the remaining 'strong' classifiers by application of genetic recombination operators such as crossover and mutation.

7.4 Object-Oriented Analysis and Design of Classifier Systems

A great motivation behind the widespread use of object-oriented techniques in modern software construction is the high level of reuse which can be achieved. Most attempts at reuse in object-oriented software construction has tended to concentrate on reuse of individual objects or components. In [135] it has been proposed that a higher level of reuse can be achieved by reusing whole designs or patterns found in existing designs. This section demonstrates how a robust software architecture for the Learning Classifier System can be constructed by combining reuse of component objects at a lower level of abstraction and design reuse at a much higher level. Component reuse is achieved by reusing common or similar objects found in the design of the Genetic Algorithm while design reuse is achieved by the whole sale reuse of the GA design as part of the learning system in the construction of the Learning Classifier System.

7.4.1 Object-Oriented Analysis of Learning Classifier Systems

As mentioned before, the analysis process seeks to understand the system before a solution can be designed. A proper understanding of the system will not only ensure that the correct system is designed and subsequently implemented but will also make the system less likely to require changes soon after it has been built. The analysis process consists of identifying the important objects and relationships in the classifier systems domain. Different methods have been proposed for identifying objects in a problem domain [17,79]. The most common approach is to analyse a textual description of the particular problem statement. Nouns in the description are usually objects in the Classifier domain whereas the verbs tend to be operations on objects. After the objects have been identified, a data dictionary is created to describe the roles that the objects play in the system. The results of the analysis process are captured in models which will be further enhanced in the system design process. Different methodologies have different modelling diagrams and notations for capturing the results of the analysis process. In OMT, three analysis models are used. They include an object model, a dynamic model and a functional model. The object model describes the static structure of the system, i.e., the relationships between the objects found in the system. The dynamic model shows the states that the system can be in and different events that the system needs to respond to. Finally, the functional model shows the different inputs and outputs of the system and the sequence of transformations or processes that result in the outputs being produced from the system's inputs.

7.4.1.1 Learning Classifier System: Problem Description

A learning classifier system consists of string rules called classifiers in a classifier store that co-operate to solve problems in a given environment. The main portion of the learning classifier system is a rule and message system which processes messages from the environment. Environment messages are received through detectors and posted onto finite length message lists where they are used to activate the string rules or classifiers. Each classifier consists of a condition part, an action part and equivalent strength representing the usefulness of the string rule encoded by the classifier. Learning is achieved by employing a Bucket Brigade Algorithm (BBA) and a Genetic Algorithm (GA). The BBA is used to modify rule strengths depending on past usefulness of the rule while the GA is used to evolve new rules (classifiers) from existing rules. Activated classifiers send messages or actions to the environment through effectors and depending on the usefulness of the message, a reward or penalty is sent back to the system via detectors and is distributed amongst active rules. Both the condition part and the action part of the classifiers are finite length strings over a given alphabet. In the case of the simple classifier system, the alphabet is very simple and consists of 3 characters; '0', '1' and '#' (don't care). Despite this, the classifier grammar is capable of representing very complex rules (classifiers) by simply concatenating elements of the alphabet. Closure is achieved in the classifier grammar by ensuring that all operations on the alphabet produces results in the scope of the alphabet.

7.4.1.2 Identifying Classifier domain objects

Table 7.4-1 show the classifier domain objects that were identified by textual analyses based on the above problem description and also from descriptions of classifier architecture in the literature [119, 123, 136].

Table 7.4-1: Objects in the Learning Classifier System

Classifier Alphabet	Classifier Condition	Classifier	Bucket Brigade Algorithm
Classifier Store	Classifier Action	Message	Genetic Algorithm
Message List	Effectors	Detectors	Learning Classifier System
Environment			

The Data Dictionary

Classifier Alphabet — A '0', '1' or '#' (don't care) that represent the lowest level encoding of information in the classifier system.

Message — A finite length string over some finite Alphabet. <Message> ::= $\{0,1\}^k$

Condition — A simple pattern recognition device. <Condition> ::= $\{0, 1, \#\}^k$.

Action — Another name for a Message used in the context of a classifier.

Classifier — A production or string rule <Classifier> ::= <Condition>:<Action>

Classifier Store — A collection of Classifiers that make up Learning Classifier System's Rulebase. <Classifier Store> := $\{\text{Classifier}\}^k$.

Message List—Temporary memory used by the classifier system to control rule actions.

Environment— Metaphor to describe the problem to be solved by the classifier system.

Detectors— Input interface for obtaining Environment messages into the classifier system.

Effectors— Output interface for sending classifier actions to the Environment.

Bucket Brigade Algorithm— A system for apportioning credit amongst classifiers by modifying their rule strengths based on pass or perceived usefulness.

Genetic Algorithm — A search mechanism to evolve new rules or classifiers into the classifier store based on classifier strengths.

7.4.1.3 Identifying the relationships between domain objects

The static relationships between the domain objects in the learning classifier system are captured by the domain object model shown in Figure 7.4-1. The model shows the very complicated architecture of a Learning Classifier system. Some of the domain objects such as Genetic Algorithm or Bucket Brigade Algorithm are object hierarchies in their own right with constituent objects still being related to other domain objects. The object model is used to describe the static structure of the system without any consideration for how the system behaves in time or the functions that the system is to perform.

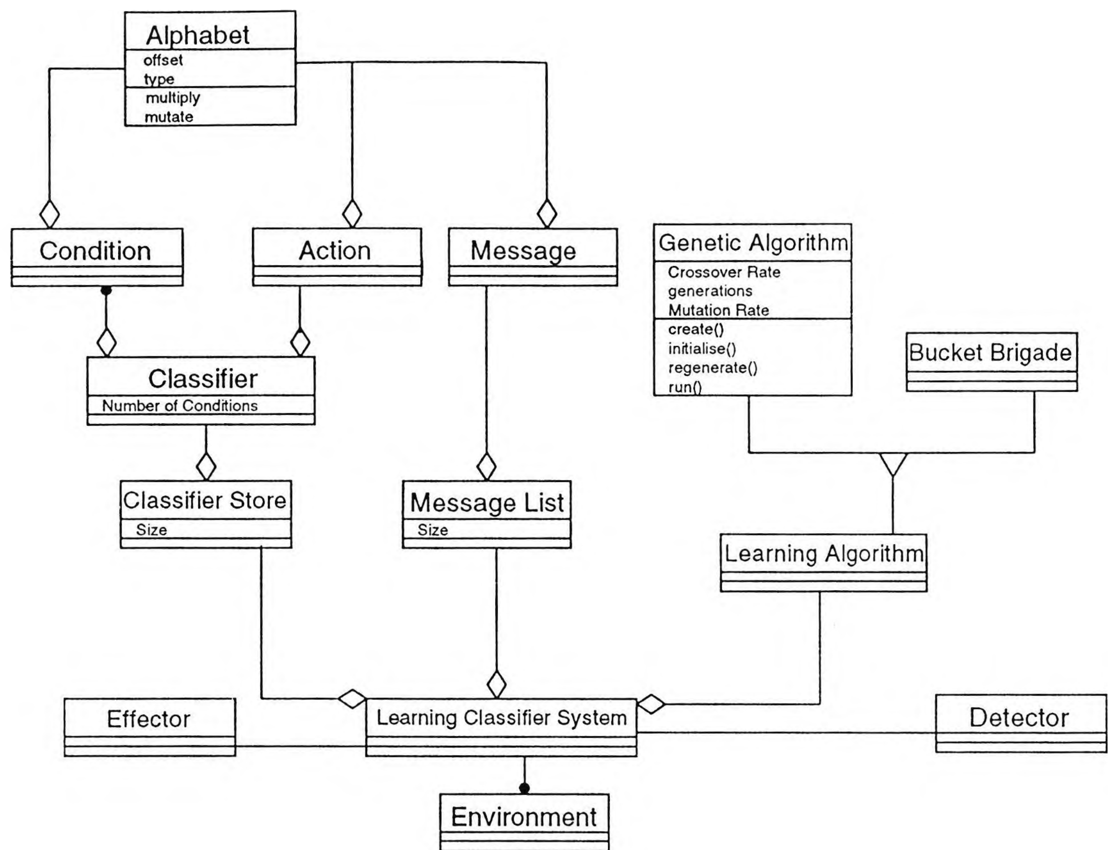


Figure 7.4-1: Domain object model of Learning Classifier System

The time behaviour of the system is captured by the dynamic model. The dynamic model enumerates the important states that the system can be in and the events that the system can respond to. When in a particular state, the system can perform certain activities or wait for an event to occur. The occurrence of an event causes a state-transition to a new state but can also cause an action to be performed. If the new state is a final state, the system terminates and processing is complete. Dynamic models are expressed as state-transition diagrams. Figure 7.4-2 shows a state-transition diagram for the Learning Classifier System.

Learning Classifier System

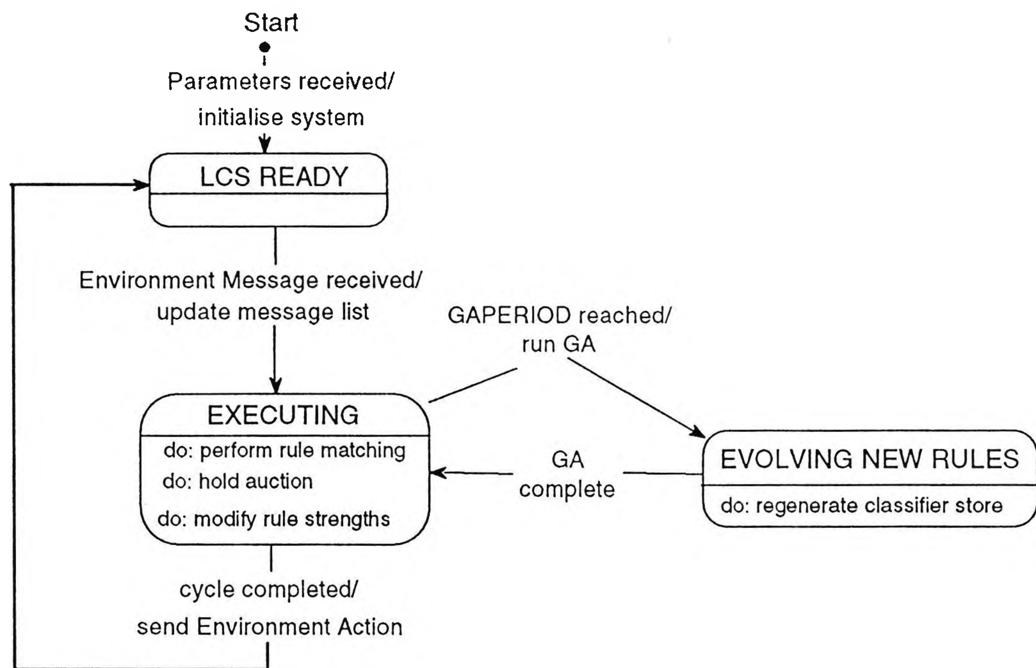


Figure 7.4-2: Learning Classifier System: State-transition diagram

The dynamic model shows the three main states that the system can be in; *Ready, Executing* and *Evolving*. At the start, the user supplies system and problem parameters to initialise the system. The state of the system changes to *Ready*. When an environment message is received by posting onto the message list, a state transition to *Executing* occurs. In this state, one of two events can occur; The first event, *GAPERIOD reached*, will cause the action *run GA* to be performed and results in a state-transition to *EVOLVING NEW RULES*. In this state, the activity, *regenerate classifier store* is performed where new classifiers are introduced into the system using genetic selection and recombination operators. When the Genetic Algorithm is complete, the event, *GA complete* is received causing a state-transition back to *EXECUTING*. The second event, *cycle complete*, marks the end of the execution cycle and causes a state transition back to *LCS Ready* after an action has been sent to the environment. There are potentially many more states in this system than cab be shown in the state-transition diagram. For example, states due to error conditions or interruptions from users have not been shown.

The functionality of the system is described with the aid of a functional model. Functional models show the various inputs to the classifier system and the sequence of transformations that result in the outputs being produced. Functional models are expressed in the form of dataflow diagrams. For such a complicated system, a single data flow diagram that describes all the inputs and transformations in detail would be

incomprehensible. Hence, the dataflow diagrams are levelled to facilitate both their creation, understanding and subsequent modification. The highest level dataflow diagram (system context diagram) for the learning classifier system is shown in Figure 7.4-3. The diagram shows that inputs to the classifier system come from both users and the environment. Environmental input is in the form of an environmental message and a payoff. User inputs include problem parameters, system parameters and learning parameters. The only output from the system is the classifier action which is used to effect some action in the environment.

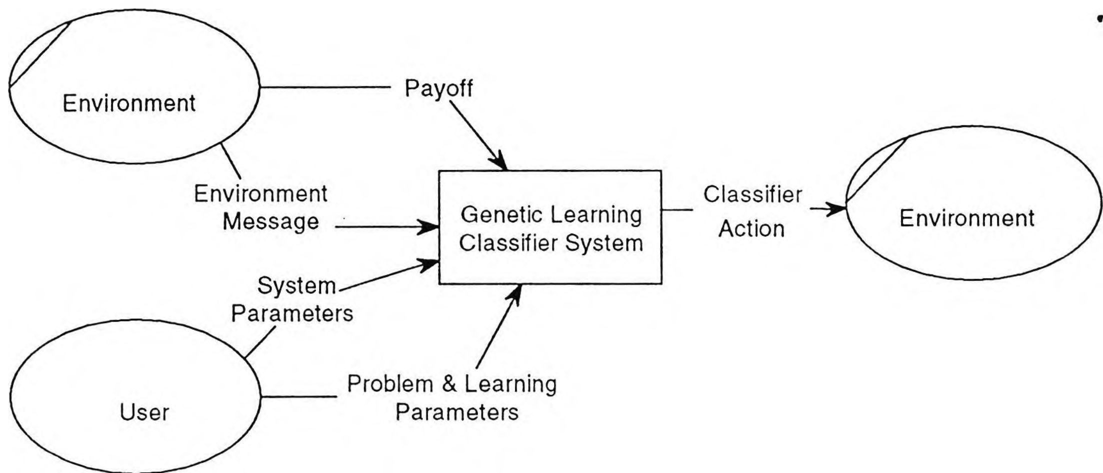


Figure 7.4-3: Learning Classifier System: Context Diagram

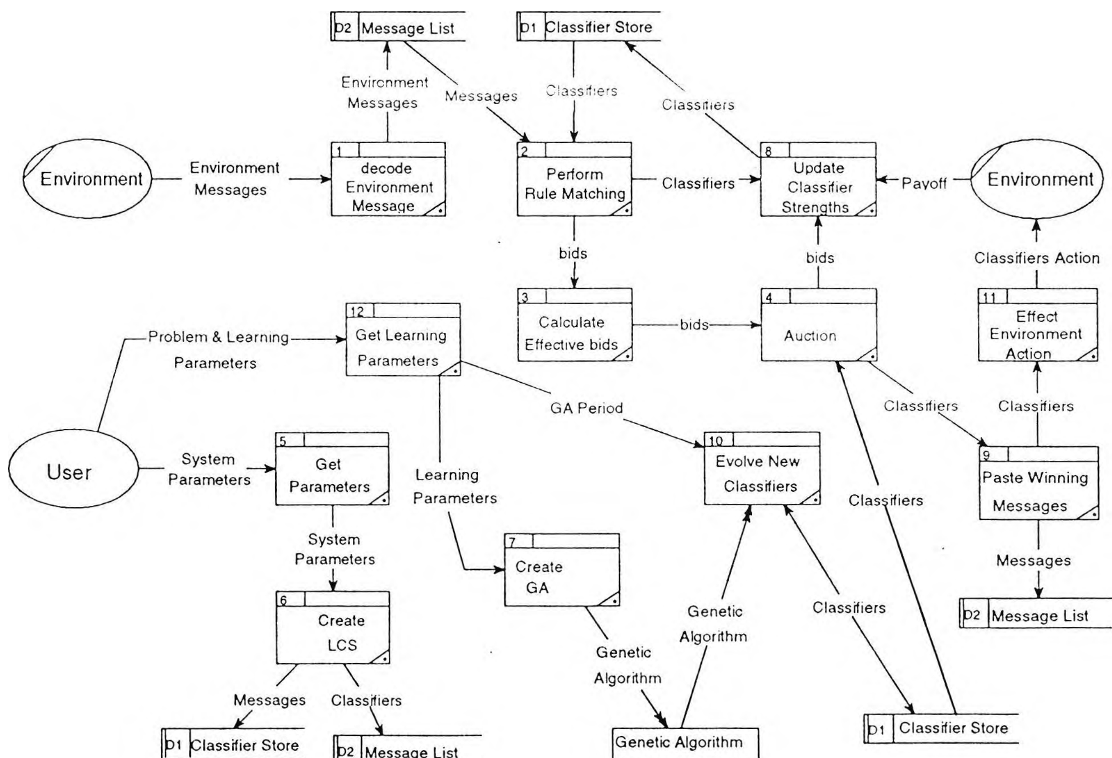


Figure 7.4-4: Learning Classifier System: Level 1 Dataflow diagram

Figure 7.4-4 shows a level 1 dataflow diagram for the learning classifier system. Here, the functionality of the system is described in a greater level of detail. The sequence of transformations that occur to convert User and Environmental inputs into the final classifier action which is sent to the environment is clearly outlined. For example, the diagram shows how environmental messages are decoded and sent to the message list. The function, *Perform Rule Matching* matches classifiers from the classifier store to messages in the message list. Its outputs are bids and winning classifiers. The bids are sent to *Calculate Effective Bids* where the effective bids are calculated. The function, *Update Classifier Strengths* uses the bids and environment payoff to update the strengths of the active classifiers which are then written to the classifier store. The dataflow diagram can be levelled even further to show high levels of detail for each of the function transformations in Figure 7.4-4. At the lowest level, the transformations will be equivalent to functions or subroutines in conventional programs and class member functions or methods in Object-Oriented programs.

7.4.2 Object-Oriented Design of Learning Classifier Systems

The aim of the design process is to create a software architecture for the Learning Classifier System. The design process makes use of the analysis models to produce design model. The design model forms the blue print which will be implemented in a computer programming language to realise the system. The first stage of the design process is to construct a software architecture for the system. The system architecture is the overall organisation of the system into subsystems or partitions that can be easily constructed and independently tested. The second stage of the design adds extra detail to the analysis domain models as well as further objects to allow for a flexible, reusable and efficient implementation of the Learning Classifier System. Also, in this design stage, decisions are made about the interface and the representation of all the objects in the application and how they interact to realise the system operation. In the design of the Learning Classifier System, a very high level of reuse has been achieved by reusing both the GA design and the component GA objects.

7.4.2.1 Classifier Systems Design

As mentioned above, the systems design process seeks to establish an overall software architecture for the Classifier System. By partitioning the Classifier System into subsystems that can be easily and independently analysed, designed and tested, the complexity and hence the possibilities for errors are greatly reduced. Where the partitioning results in

meaningful subsystems, this will facilitate the reuse of whole designs at the subsystem level as opposed to component reuse at the object level. Even though the unit of production in object-oriented analysis and design is based around individual objects, a partitioning along object boundaries is not feasible for even moderate sized systems where the number of objects can be quite large. For the Learning Classifier System, natural partitions into subsystems already exist and these are evident when the literature is examined. The following major subsystems have been identified:

- The Rule and Message subsystem and
- The Genetic Algorithm subsystem
- The Input subsystem
- The Classifier Interface subsystem
- The Environment subsystem

The concept mapping technique [78] provides a diagrammatic way in which the interaction between the main system and the different subsystems can be expressed. In a concept map, the key concept will usually correspond to the main system while the less general concepts will correspond to subsystems. Each less general concept can in turn be analysed as a key concept. The concept map in Figure 7.4-5 shows the activities performed in the different subsystems and the flow of data between the subsystems.

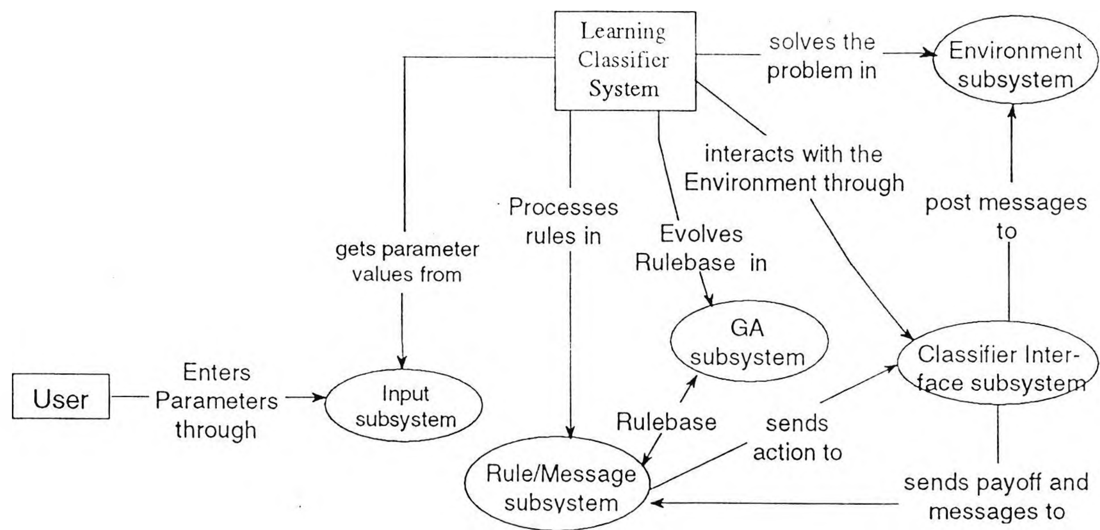


Figure 7.4-5: Concept map showing the interaction between subsystems in the Learning Classifier System

7.4.2.2 Object Design

The object design phase expands the analysis object model by adding extra objects so that the Classifier system can be efficiently implemented. These objects are not found in the Classifier System vocabulary or problem description but are necessary if design or component reuse is to be achieved during implementation. Extraneous objects usually include collection objects such as Arrays and Vectors and Adaptor objects that make it possible to reuse existing objects by either modifying their interface or their representation. Other objects result from a reorganisation of the analysis model objects into inheritance or object composition hierarchies. Some analysis domain objects can thus become component parts of the design domain objects through object composition or special cases of design domain objects through inheritance. The expanded model is known as the system or design object model. Figure 7.4-6 shows the design object model for the Learning Classifier System.

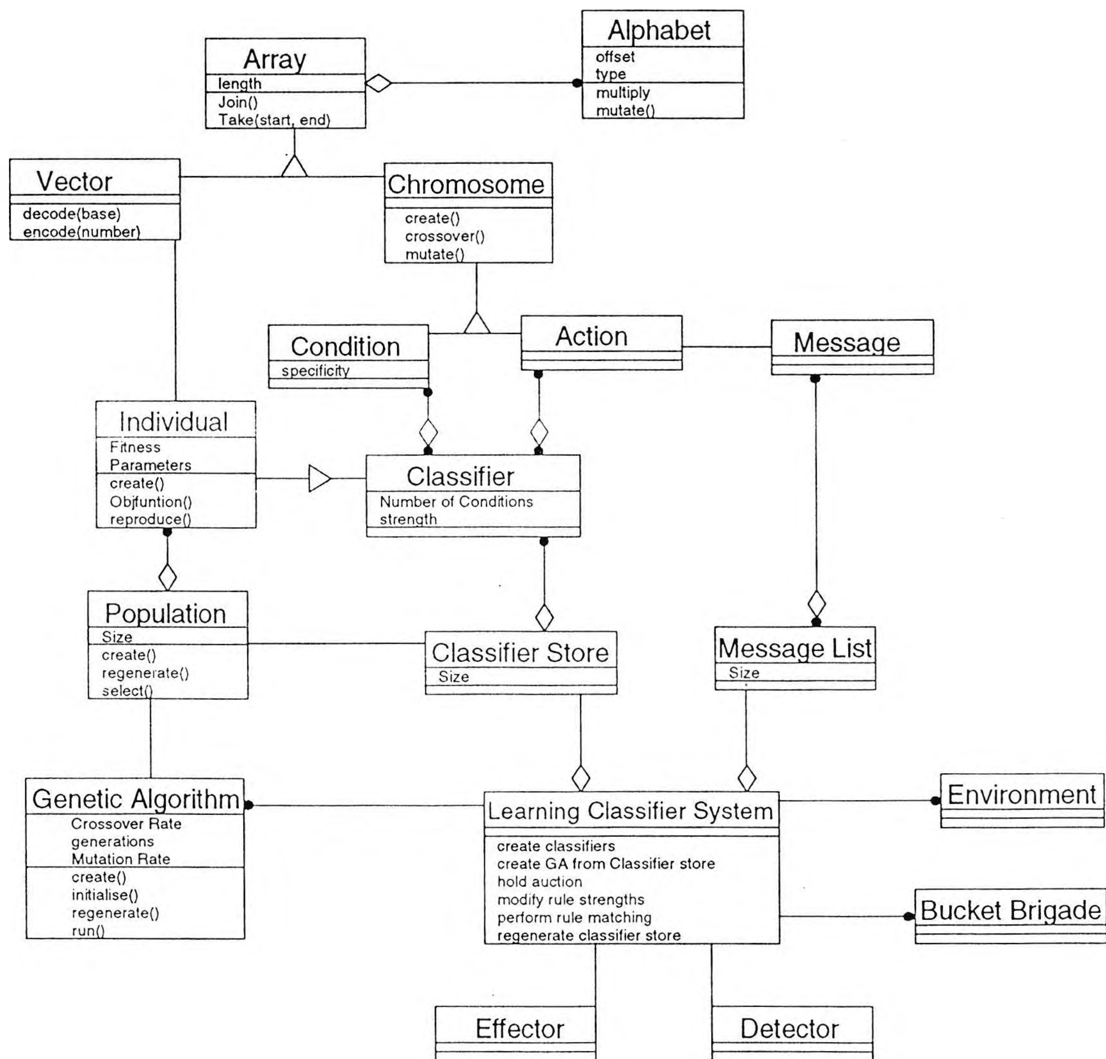


Figure 7.4-6: Design object model of learning classifier system

The figure shows how objects in the GA design, including Alphabets, Arrays and Chromosomes are reused in the design of classifier system components. Both Classifier Condition and Classifier Action inherit a large part of their interface and their functionality from the previously designed Chromosome. The design diagram also shows how individuals in the GA population inherit the operations and implementation of Classifiers. The Individual acts as an Adaptor object to convert Classifiers into Individuals in a standard GA Population. The strength of the Classifier becomes the fitness of the Individual, while the Condition and Action part of the Classifier are used as the Individual's information encoders. The Population and Genetic Algorithm in the previous design are thus reused with very little modification in the Classifier Systems design. It will be necessary in the implementation to modify the interface to the Individual object so that the underlying Classifier can be properly initialised when an Individual object is created.

7.4.2.3 Describing Object Interactions

Before the design can be implemented, the objects in the system object model have to be fully specified. A fully specified object has all its attributes, associations and member functions defined. The set of all attributes and associations make up the implementation or representation of the object while the object's interface is given by the set of operations that can act upon or be acted on by an object. The operations that make up the objects interface can be obtained by examining the events or messages that the object sends and receives from other objects when the application is run. Unfortunately, for a large system with many objects, an enumeration of all the events is not feasible. On the other hand, without any such examination, key operations required of an object for the proper functioning of the system or application can be easily omitted. The effects on the overall quality and usefulness of the software system are potentially disastrous. The solution is to create Use Cases for all the important requirements that the system or application has to satisfy. As mentioned before, a Use Case is a particular mode of interaction or use of the system. Use Case design is expressed using Object Interaction Diagrams (OID). For the Genetic Learning Classifier System described in this design, the three main Use Cases are:

1. Create LCS: whereby the different objects that make up the Learning Classifier System are constructed.
2. BBArun: This is the Match/Execute cycle where the Bucket Brigade Algorithm is used to modify classifier strengths.
3. Regenerate Classifier Store: where the Genetic Algorithm is used to evolve new Classifiers which are inserted into the Classifier database.

Figure 7.4-7 shows the OID used to depict the create LCS Use Case. A dashed line has been used to represent the system boundary. The diagram shows the sequence of messages that are sent between the different objects in order for the Classifier System to be created. The first message, *initialise*, to Parameters will result in the creation and initialisation of the Parameters object with user supplied system and problem parameters. A *create* message to LCS will result in a series of *create* messages to the constituent components of the Learning Classifier System. Because there are potentially too many object interactions than can be shown on a single object interaction diagram, a Probe has been created for the *create* message to Classifier which is then expanded in a separate object interaction diagram in Figure 7.4-8. Each *create* message can take a list of zero or more arguments.

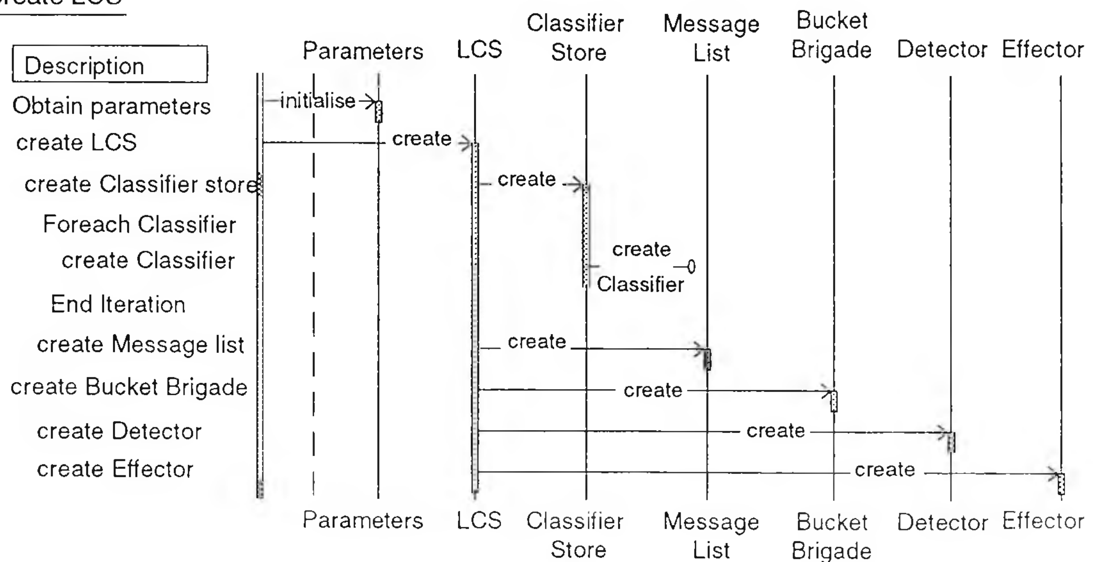


Figure 7.4-7: Object Interaction Diagram for Create LCS Use Case

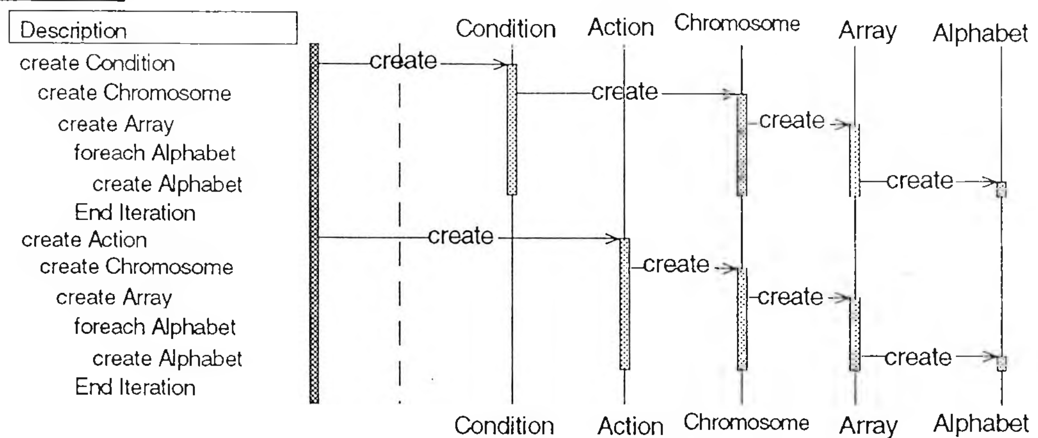


Figure 7.4-8: Object Interaction Diagram for the create Classifier Use Case

In the second Use Case, depicted in the OID in Figure 7.4-9, a *run* message is sent to Bucket Brigade Algorithm to initiate the match/execute cycle. The BBA first sends a *get Message* request to Detector to obtain the Environment message which is then posted by an *insert* request from Detector to Message list.

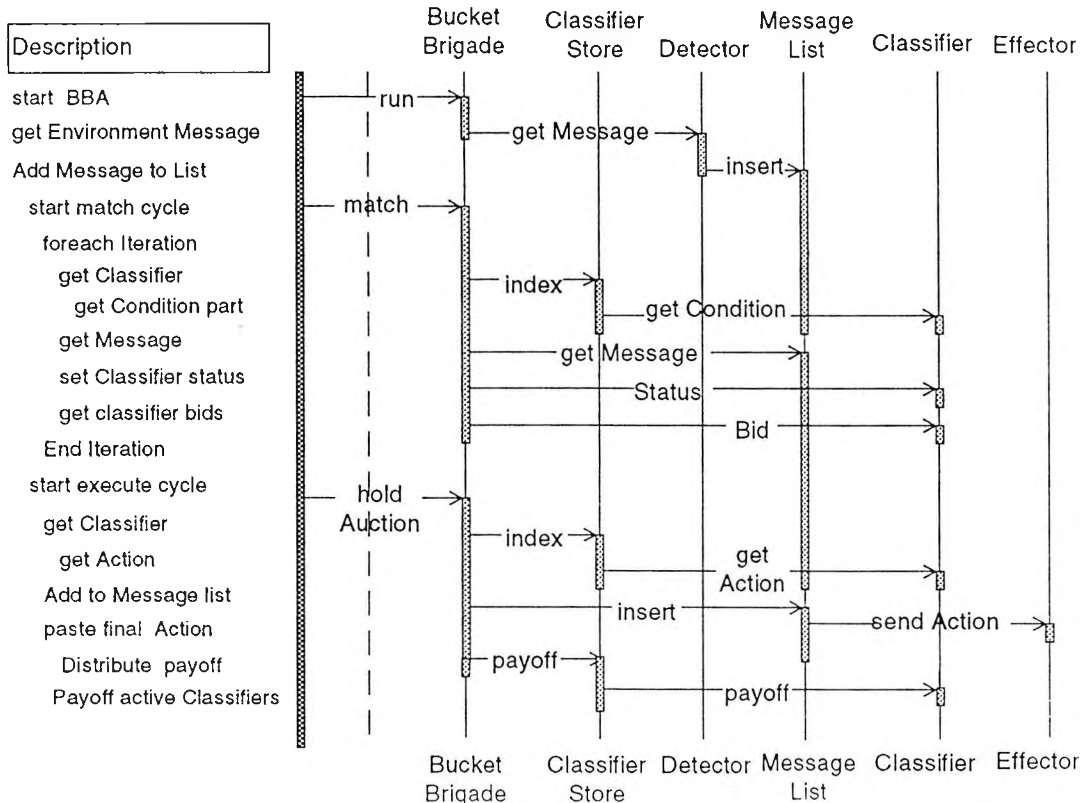


Figure 7.4-9: Object Interaction Diagram for Match/Execute cycle Use Case

The BBA then co-ordinates the message passing between the relevant objects to complete the match/execute cycle.

The final Use Case documents the process of regenerating the Classifier Store. The Learning Classifier System periodically invokes the Genetic Algorithm component so that new rules or Classifiers can be generated to replace consistently weak Classifiers. The GA uses the strength of Classifiers as a fitness measure and new Classifiers are generated as novel combinations of existing fit Classifiers using the genetic operators. Only Partial or Overlapping replacement is used in conjunction with Tournament selection strategy so that previously useful Classifiers are preserved in the Classifier Store after the GA has been invoked.

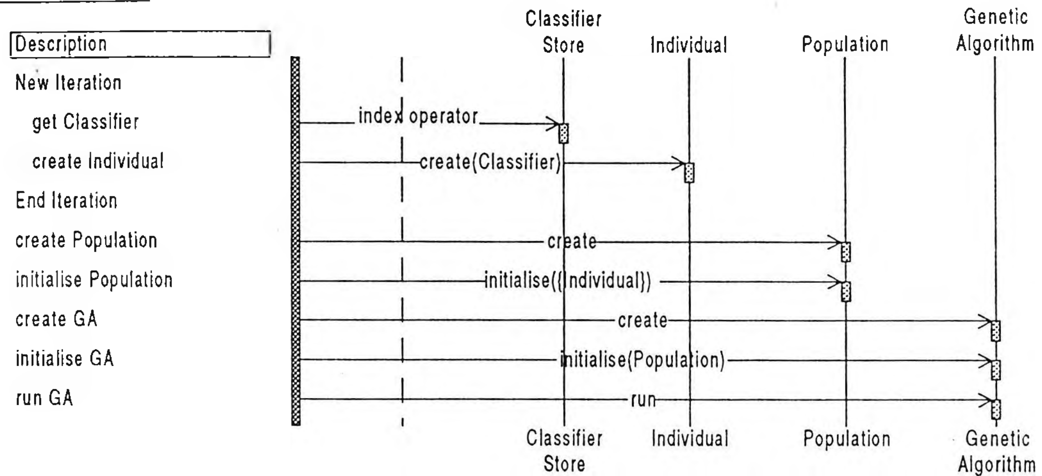


Figure 7.4-10: Object Interaction Diagram for regenerate Classifier Use Case

7.4.2.4 Learning Classifier Systems Implementation

The Genetic Learning Classifier System was implemented using the C++ programming language on a SUN SPARC 10. In the implementation, there is a direct translation between objects in the design object model and classes in the C++ programming language. The C++ language also provides direct support for implementing the different Relationships between classes[137, 138, 139]. For example, Pointers and References in C++ are used to implement Aggregation or composition relationships while class Inheritance is used to implement Generalisation/Specialisation relationships [81]. The listing in Figure 7.4-11 shows how C++ class inheritance is used to implement the generalisation relationship between a Chromosome and an Array object in the design object model. The Array class is described as the super or base class while Chromosome is the sub or derived class. Inheritance is used in a similar manner to implement the relationship between Chromosome objects and Condition Objects as shown in the listing in Figure 7.4-12. Finally, Figure 7.4-13 shows how C++ pointers are used to implement the Aggregation or composition relationship between a Classifier object and its constituent Condition and Action parts.

```
#include "Array.h"
class Chromosome : public Array {
public:
	/* similar to array declarations */
	Chromosome():Array(), nparams(0), szparams(0) { }
	Chromosome(int sz): Array(0, sz), nparams(1), szparams(sz) {}
	Chromosome(const Chromosome &c) { init(c.a, c.szparams, c.nparams); }
	Chromosome& operator=(const Chromosome&);		// Chromosome assignment
	~Chromosome() {}					// destructor
public:
	void init(const T*, int, int, Bool =False);
	Chromosome(int szp, int np, Bool Msg) { init(0, szp, np, Msg); }
	Chromosome(const T *ar, int szp, int np =1) { init(ar, szp, np); }
	Chromosome& mutate(float, Bool);
	virtual int Nparams(ChromParams which =nVariables) const { // return parameters
		return (which == nVariables) ? nparams : szparams;
	}
	static void initmask(int, float);
	static void setxover(Xover);
	static Chromosome& getmask() { return *crossmask; }
	float Specificity() const { return 0.0; }	// no don't cares in ordinary Chromosomes
	static Crossover xover;
protected:
	int nparams;				// number of parameters
	int szparams;				// string size of each parameter
	static Chromosome *crossmask;		// mask for uniform crossover
};
```

Figure 7.4-11: C++ declaration for Chromosome class

```
#include "Chromosome.h"
class Condition : public Chromosome {
public:
	Condition():Chromosome() { }
	Condition(int sz):Chromosome(0, sz, defnparams) {}
	Condition(const Condition &c) { Chromosome::init(c.a, c.szparams, c.nparams, Msg); }
	Condition& operator=(const Condition&);
	~Condition() {}
public:
	void init(const T*, int, int);
	Condition(int szp, int np) { Chromosome::init(0, szp, np, Msg); }
	Condition(const T *ar, int szp, int np =1) { Chromosome::init(ar, szp, np, Msg);	}
	Condition& mutate(float p) {
		Chromosome::mutate(p, Msg);
		return *this;
	}
	int Specificity() const; // number of don't cares
protected:
	static Bool Msg;
};
```

Figure 7.4-12: C++ declaration for the Classifier Condition class

```
#include "Condition.h"
#include "Action.h"

class Classifier {
friend void xover(const Classifier&, const Classifier&, Classifier&, Classifier&, float, float);
public:
        static double bidTotal;
        static double bidnsd;                   // bidding noise standard deviation
public:
        virtual void init(int, int, int, double);
        virtual void init(const Condition&,const Action&,Bool,double,int =0);
        virtual void init(const Classifier &C) {
                init(*C.condition, *C.action, C.status, C.fitness, C.active);
        }
        Classifier() : condition(0), action(0), status(False), fitness(0.0), active(0) {}
        Classifier(const Classifier &C) { init(C); }
        Classifier& operator=(const Classifier&);
        operator Condition() const { return *condition; }
        operator Action() const { return *action; }
        int Specificity() { return condition->Specificity(); }
        Action& paste();
        int state() const { return active; }
        double bid() { return (double) bidConst*fitness*Specificity(); }
        virtual ~Classifier() { delete condition; delete action; }
        operator double() { return fitness; }
        Chromosome& operator[](int idx) const {
                return idx == 0 ? (Chromosome) *condition : (Chromosome) *action;
        }
        virtual void scan(ifstream&);
        virtual void retrieve(ifstream&);
        virtual void print(ostream& = cout) const;
        void update(double f);
        void reset() { status = False; active = 0; }
        }
protected:
        Condition *condition;           // condition part of classifier
        Action *action;                 // action part of classifier
        double fitness;                 // classifier's strength
        Bool status;                    // determine status of classifier
        int active;                     // number of times classifier is active in each cycle
};
```

Figure 7.4-13: C++ declaration for Classifier class

7.4.2.5 Object-Oriented Testing

As with the Genetic Algorithm, unit testing is carried out on individual objects in the Classifier System to check for satisfactory behaviour. Extensive testing is not required on component objects reused from the GA design except where modifications have been made. In such cases, only the new functionality has to be tested. The preliminary results of applying the Learning Classifier System to control the movement of an autonomous vehicle in a simulated obstacled environment is presented in the next section.

7.4.3 Learning Classifier System applied to autonomous vehicle control in an obstacled environment

The computational requirements of autonomous vehicle navigation in a real time environment[140, 141] provide an interesting problem on which the massive parallelism implicit in learning classifier systems can be applied. In [142], a genetic algorithm solution has been proposed as an alternative to dead reckoning techniques for estimating the position of an autonomous robot vehicle. There, the problem formulation is to estimate the vehicle orientation and location from measured sensor values and a given geometric model of a simulated environment. The learning problem is facilitated by constraining the robot motion to prescribed paths. In this thesis, a Learning Classifier System is applied to control the movement of the autonomous vehicle in a simulated obstacled environment. The environment is represented by a rectangular grid of cells in which the vehicle is placed at random. The goal of the Classifier System is to navigate the vehicle through the obstacles to a pre-specified finish point. The state of the cells determines the presence or absence of an obstacle. The goal is expressed as the relative distance at any time from the finish point. Sensors attached to the vehicle obtain information about its immediate surroundings and also payoff related information. The payoff information depends both on the distance of the vehicle from the final position and the state of current position in terms of the number of obstacles in the vicinity of the autonomous vehicle.

7.4.3.1 The Environment

The learning classifier system is applied to the control of a mobile vehicle in a 2-dimensional environment. The environment is made up of a 100x100 grid of cells in which obstacles have been placed at random. Each cell is implemented as a location in Cartesian space and a Boolean value which determines the presence or absence of an obstacle. Figure 7.4-14 shows the static structure of the simulated environment. The Environment is an Aggregation of Cells which in turn contain a Location object that determines their position in Cartesian space. Action strings are received from the Effector and messages and payoff sent to the Detector. The Environment also keeps track of the position of the vehicle and the co-ordinate of finish point.

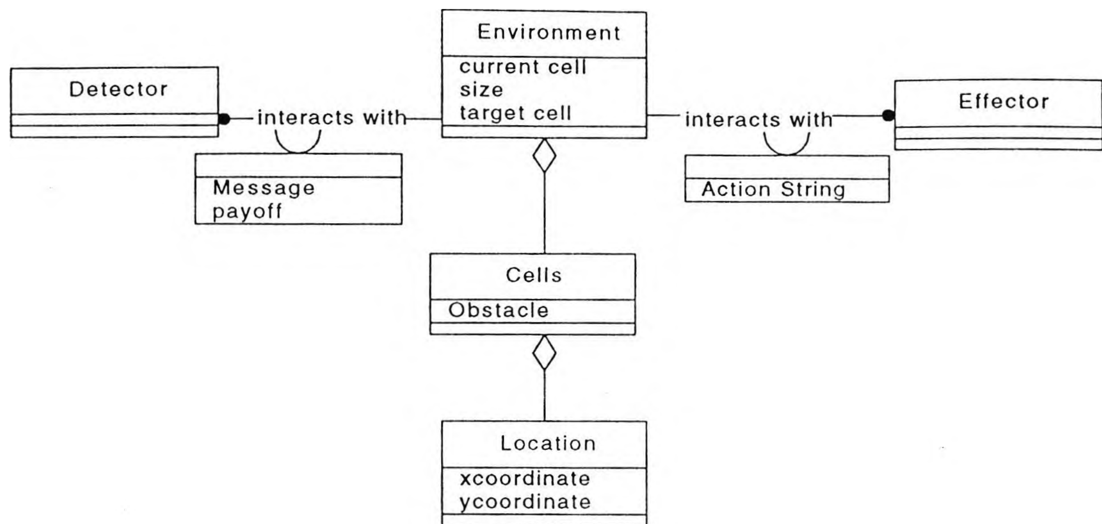


Figure 7.4-14: Object modelling diagram describing the Environment

The goal of the classifier system is to trace a path to a desired end point from a random start position in the shortest possible time while avoiding the obstacles. The input interface to the Classifier System, the Detector, utilises a 20 bit environmental message. The message is decoded as one string of 8 bits that represent the sensor information and a second string of 12 bits that encodes the distance of the vehicle from the desired end point. The sensor information is a direct encoding of the state of the 8 adjacent cells to the vehicle at time t. The remaining 12 bits are used for encoding the distance of the vehicle from the finish point. Figure 7.4-15 shows how the sensor information from the environments is coded as part of the Environment message string.

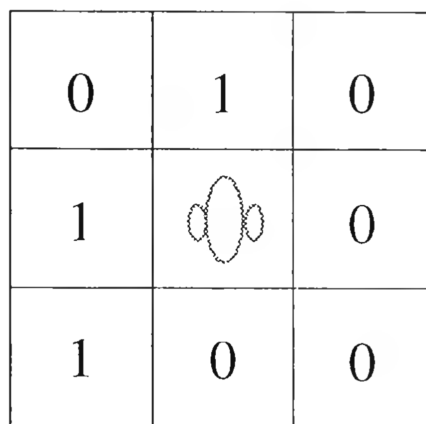


Figure 7.4-16: Encoding of the sensor information

The first 8 bits of the input message for an environment represented by Figure 7.4-17 will thus be 01010100. The Classifier System processes the input message for a fixed number of cycles and issues commands to the output interface or Effector. An Effector message is 3 bits long and encodes the direction along which the vehicle will travel until it encounters an obstacle. Figure 7.4-18 shows the direction that the different messages encode.

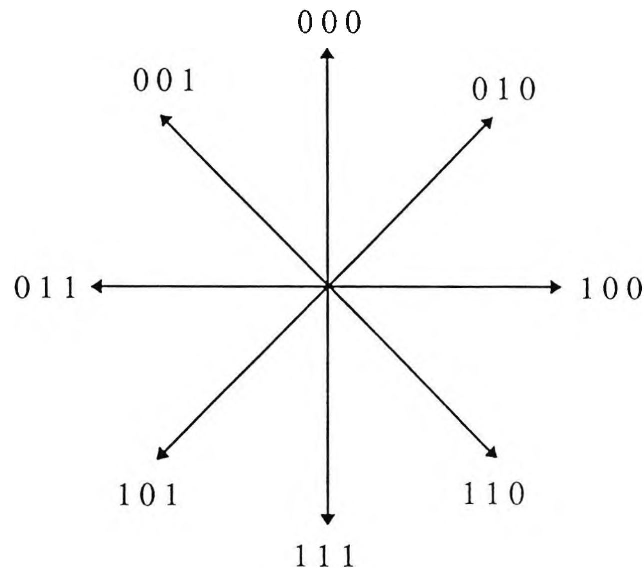


Figure 7.4-18: Direction encoding of Effector messages

At this point, the payoff for the current action is calculated and a new environment message is generated for input to the Classifier System and statistical information about Classifiers performance collected. A record of the best and worst classifiers (in fitness terms) and also the average fitness of the classifier population is kept at each step of the match/execute cycle. The Genetic Algorithm is invoked when the average fitness of the population drops below a pre-defined threshold.

7.4.3.2 Credit Assignment

When an action string is sent to the environment, the autonomous vehicle's position can either improve, remain constant or degrade with respect to the finish position. Also, the state of adjacent cells might become more or less favourable with respect to the presence of obstacles. An ideal credit assignment scheme will recognise action strings that lead to more favourable conditions for the vehicle and penalise strings that either move away from the target or move the vehicle to an area with a large concentration of obstacles. However, a move away from the target can sometimes be beneficial to the overall goal of reaching the finish position if this takes the vehicle away from areas of high obstacle

density. Unfortunately, it is difficult to reward such actions except where information about the entire environment is available at the instance when payoff is calculated. In this thesis, a fixed value equivalent to one fifth of the initial strength of the Classifiers is used as the maximum reward with the negative of the value as the maximum penalty. After the move associated with a posted action string has been carried out, the payoff is calculated, based on the resultant distance moved, as a fraction of the maximum reward or penalty with extra penalties for any obstacles in the vicinity. As shown in Figure 7.4-19, movement towards the finish point results in a reward while a move away incurs a penalty.

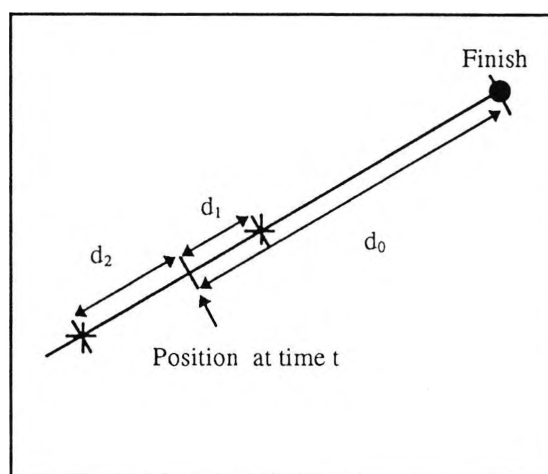


Figure 7.4-19: Calculating the payoff

For example the message a message that results in move d_1 if Figure 7.4-19 gets the following reward minus any penalties for obstacles in the vicinity

$$\text{payoff} = \text{reward} = \frac{d_1}{d_0} \mathit{MaximumPayoff} \qquad (7.4\text{-}1)$$

while a message resulting in move d_2 receives a penalty in addition to penalties for obstacles in its vicinity

$$\text{payoff} = \text{penalty} = \frac{-d_2}{d_0} \mathit{MaximumPayoff} \qquad (7.4\text{-}2)$$

When the calculated value for payoff exceeds the maximum permitted, the payoff will be clipped at the pre-set maximum.

7.4.4 Preliminary Results

In testing the learning classifier system, we wish to establish that the areas of high obstacle density are avoided because of the steep penalties associated. On the other hand, the simple movement scheme means that the classifier only stops when an obstacle is

encountered or the environment boundary is reached. Also, because both the environment and the target are different from one run to another, there is no set route that the classifier can memorise. Figure 7.4-20 shows a sample environment with randomly placed obstacles. The environment consists of 20*20 = 400 cells. There is a 0.05 probability of a cell containing an obstacle so that approximately 5% of cells have obstacles. The classifier store is initialised with 20 classifiers and each classifier has an initial fitness of 200. The message list is limited to a maximum of 5 messages. The GA is called into action every time the average fitness of the population is goes below 75% of its possible maximum. The GA uses an overlapping replacement strategy to replace less fit classifiers from the classifier store. It was observed that the autonomous vehicle's motion tended to avoid areas of high obstacle density. Figure 7.4-21 shows a sample motion of the vehicle over the first 20 cycles. The results show that the initial motion of the vehicle is more akin to random walk with obstacle avoidance.

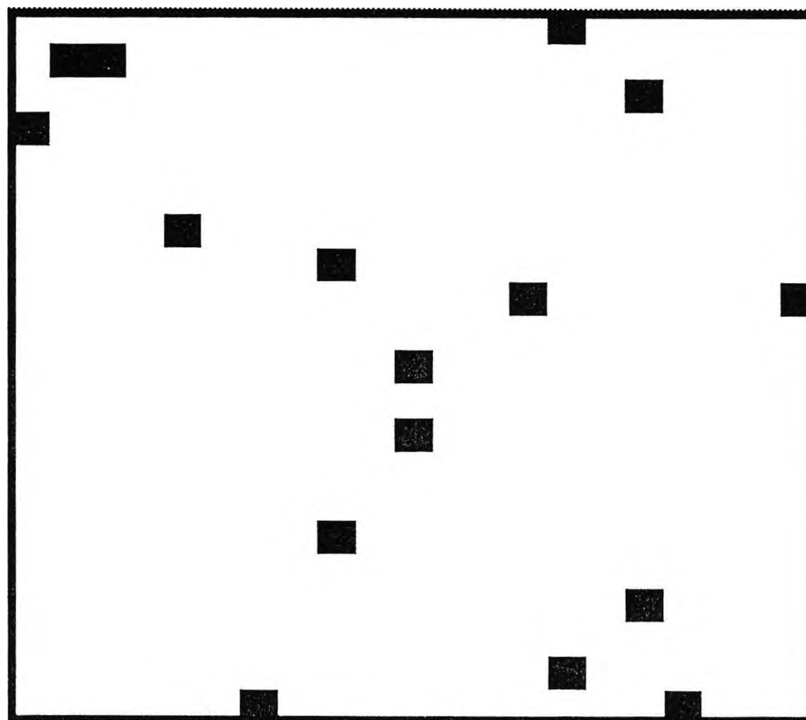


Figure 7.4-20: A sample environment

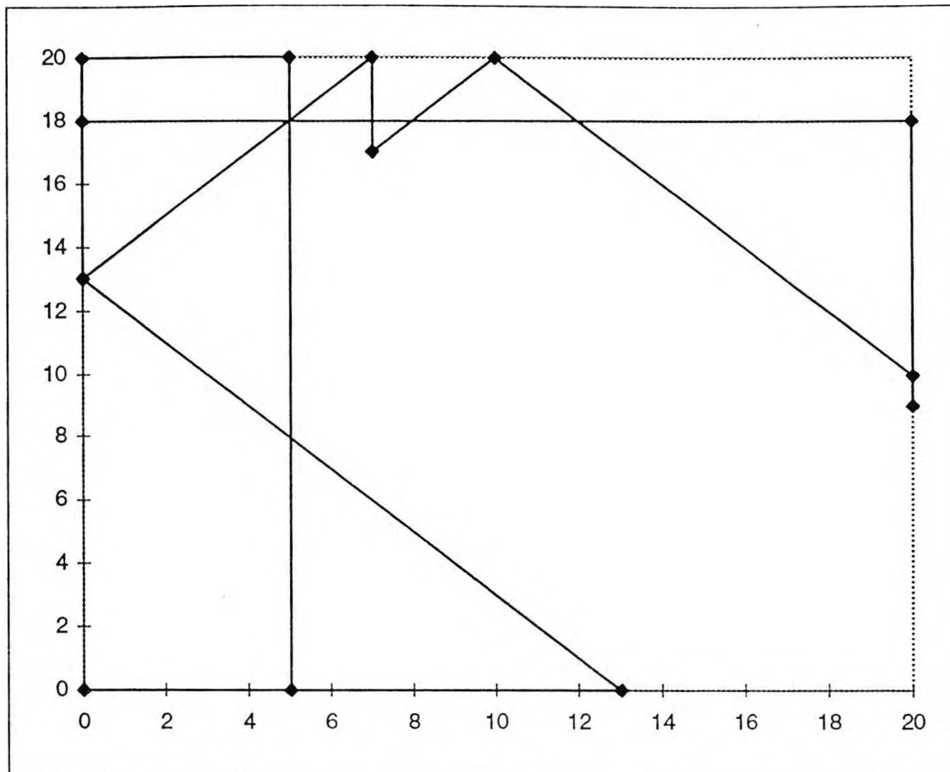


Figure 7.4-21: Autonomous vehicle movement

7.5 Discussion and Conclusions

Robust search mechanisms are a major development in the field of intelligent computing. The breadth of problems to which genetic algorithms have been applied is an indication of their versatility. The ability to search for the global optima of an objective function makes it possible to use genetic algorithms where analytical solutions are not possible. This chapter has demonstrated how an open software architecture for genetic algorithms can be designed and constructed using object-oriented techniques. Reusable components and designs have been constructed for the GA that can become parts of new designs for other evolutionary computation design efforts. The effectiveness of a GA solution has been clearly demonstrated in two sample applications where the GA is used to search for function maxima. The concept of design reuse, popularised in [73, 135] has been demonstrated in the design of the Genetic Learning Classifier System. Genetic Learning Classifier Systems are very complex, both in their structure and operation. Object-oriented analysis has been used to improve an understanding of the classifier domain before design was carried out. Object-oriented design follows directly from object-oriented analysis. There is no semantic gap or unnecessary translations from analysis domain to the design domain which can result in errors or ambiguities. There is also a direct translation from design domain to implementation in the C++ programming language.

Chapter VIII

ACHIEVEMENTS AND FUTURE DEVELOPMENTS

8.1 Achievements

The field of computational intelligence is very large, so much so that no single thesis can possibly touch ever aspect of it. While other research efforts burrow deep into individual or a small combination of some of the technologies that constitute the field of computational intelligence, this thesis is unique in that it proposes methods that are applicable to the broad range of computational intelligence technology. Issues which apply to the construction of robust software architectures for neural networks apply with equal fervour when probabilistic, fuzzy or genetic learning systems need to be constructed. The major achievements of this thesis include :

- **A proposed new scheme for classifying neural networks** based on a new set of discriminating features and a novel application of notations found in the Object Modelling Technique. The scheme boasts a more intuitive and more descriptive classification using a standard set of easily recognised diagrammatic symbols. The notation lends itself to automation support where features and capabilities of different neural network paradigms can be easily and conveniently displayed. The scheme also ensures that similar neural network paradigms are classified 'close' to each other and can be extended by introducing new discriminating features or refining existing ones to classify other new or unclassified neural network paradigms.

- **A new approach to constructing intelligent systems based on object-oriented techniques**. The proposed approach is based on the OMT and relies on object-oriented analysis to identify robust and stable objects which are found in different computational intelligence domains.

    - The algorithms which these computational intelligence systems implement evolve in time as new research comes to fruition but the problem vocabulary and hence the domain objects are usually very stable. Robust architectures for these systems can thus be designed based on the domain objects so that the need for modifications is diminished. Also the effects of modifications, when they are required do not destroy the structure of the system or impose overwhelming costs in terms of constraints to the users of the systems. OMT

provides a simple method which can be used to develop object-oriented systems. Complex methods as reported in Booch[17], SSADM [143], and Fusion[144] which rely on a large number of diagram types and heavy textual content are unacceptable to mainstream designers who rely on a more pragmatic and simplified approach to software systems development.

- The thesis has also shown how design and component reuse can be achieved in the construction of intelligent systems. By constructing generic reusable components in the design of GAs, a basis for both design and component reuse was laid out. The GA components can be reused in the construction of other evolutionary computation systems because of the overlap in the domains. This was demonstrated in the design of the genetic learning classifier systems. Design reuse can drastically reduce development time while component reuse further reduces the time required for both implementation and testing.

- **The development of software to realise sample systems**. Software has been developed in C++ on both SUN SPARCs and PCs to realise different neural network paradigms, a fuzzy inference engine, a genetic algorithm and a genetic learning classifier system. The following software has also been developed and has been used to introduce both undergraduate and post graduate project students to computational intelligence systems:
  - a sample supervised learning network based on backpropagation with just four nodes that clearly outlines the processing steps performed by the neural network.
  - a sample inference system with four rules that outlines the processing steps performed by a fuzzy inference system.

- **The novel applications of computational intelligence to real world problems**. This thesis has presented the results of applying different aspects of computational intelligence to different problems in the field of power and control systems. Neural networks have been applied to fault identification in HVDC systems and to identifying unknown parameters in linear and non-linear dynamical systems. A fuzzy system has been design for harmonic prediction in AC systems. A genetic algorithm has been constructed for function optimisation and two examples of successful application in the search for function optima have been demonstrated. Finally a learning classifier system

has been constructed for controlling the motion of an autonomous vehicle in an obstacle laden environment.

8.2 Suggestions for Improvement

In the first chapter of this thesis, intelligent classification has been suggested for distinguishing between different neural network paradigms. This is necessary if implementers are to make informed decisions about the suitability of neural network architectures for specific applications. More often, what is required is a decision on whether or not an intelligent solution can be used and which aspect of computational intelligence is best suited for a given problem domain. While heated debates continue about the best methods for solving particular problems, the choice of solution is often very problem dependent. The people best suited to make such decisions are those who know the minute details of the problem to be solved. An extension of the classification scheme to provide information about computational intelligence paradigms as a whole will provide implementers with a starting tool on which such decisions can be made. The neural network classification scheme thus become a sub collapsible branch of the overall computational intelligence classification scheme as shown Figure 8.2-1. The diagram shows neural networks, fuzzy systems and evolutionary computation systems to be all special cases of computational intelligence systems. It is debatable whether the relationship should be a "kind-of" or "part-of" but the notation can just as easily be changed to handle either case. Hybrid systems that incorporating more than one aspects of the technology are also easily classified in such a scheme. An automated browsable database built on such a scheme can become an indispensable resource to industry.

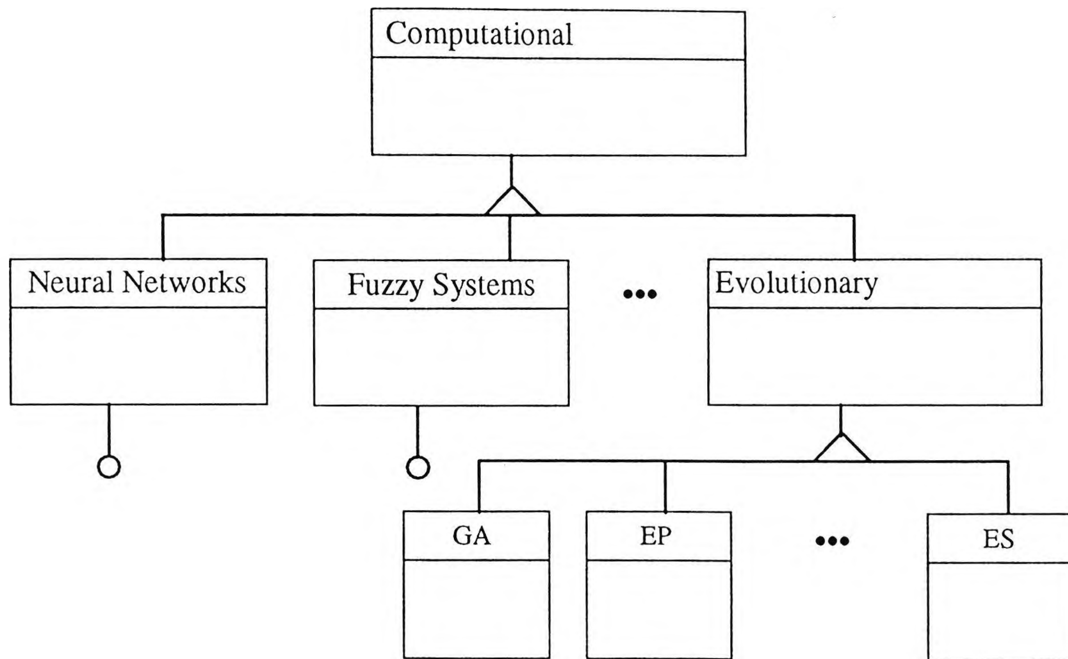


Figure 8.2-1: Classification meta model for computational intelligence systems

In Chapters IV, VI and VII, the construction of robust architectures for individual neural networks, fuzzy logic systems, genetic algorithms and genetic learning classifier systems has been presented. It has become evident that better performance and improved solution times can be obtained by combining the learning capabilities of neural networks with the robustness of fuzzy systems and the global search possible in evolutionary computation systems. The work begun in this thesis can be developed further to produce an integrated object-oriented software architecture for computational intelligence systems. Such an architecture will make it possible to utilise the individual technologies or any viable combination of technologies to achieve robustness and low solution cost in real world problems.

In chapter V, a neural network solution was presented for fault detection in HVDC systems. This approach to fault detection assumes that the statistics of the signals that make up the training patterns are stationary[1] with time. This assumption is valid in off-line applications where data patterns have been previously collected and pre-processed to extract features and reduce the inherent variations. The HVDC systems dynamic

[1] Signals are stationary if their properties do not change during the course of the signals. The concept of stationarity is well defined in the theory of stochastic processes. A stochastic process is called strict-sense stationary if its statistical properties are invariant to a shift of the origin of the time axis. A stochastic process is called wide-sense (weak) stationary if its second order statistics depend only on the time difference.

characteristics can change due to harmonic pollution and other transient phenomena which will render the assumptions invalid for on-line and real-time applications. The wavelet transform is of particular interest in the analysis of non-stationary and fast transient signals. It is local in both time and frequency domain and thus makes it possible to describe short duration and non-stationary signals with fewer wavelet transform coefficients [115]. It is possible that neural networks with wavelet bases functions will provide an improved scope for real-time fault detection in HVDC and other systems with non-stationary or fast transient signals.

In chapter VI, the derivation of necessary fuzzy rules for predicting the harmonics was carried out without any prior knowledge as to how harmonics behave in real AC power networks. The blind application of trend graphs and heuristics to data generated by simulation will lead to the wrong rules being derived if the data happened to be erroneous, incomplete or out of range. Even automated rule generation from data will still result in rules that are not completely representative of the system to be modelled. Where a domain expert is not readily available, it will be necessary to find ways in which the rules can be independently validated before they are stored in the fuzzy rulebase. On-line learning of rules using neural networks or adaptive evolution of rules using GA are possible solutions which need to be explored.

8.3 Conclusions

The future of computational intelligence systems lies in applications that combine the learning capabilities and graceful degradation found in neural networks with the robustness and support for imprecision found in fuzzy systems and the global non gradient-based searching in evolutionary computation systems to solve difficult learning problems in real time. With the ongoing and very active research in this area, it is just a matter of time before such systems become available. Unfortunately, the results are all too predictable. One-off or bespoke implementation of an algorithm or a series of complicated algorithms in different programming languages, with flow charts or pseudo-code to describe their operation and finally the usually un-interpretable and non-reproducible results of applying the algorithm to mundane application problems, which is then published in a mainstream journal or conference. While this thesis has presented intelligent software architecture issues from a viewpoint where development and maintenance efficiency have been stressed, there are very strong arguments for traceability and easy communication of ideas and design decisions to the readers of any such publications. A shift in emphasis to

architectural issues will go a long way towards improving the acceptance, understanding and applicability of computational intelligence technology.

8.4 References

[1] D Rumelhart & J McClelland, (eds), "Parallel distributed processing", Cambridge, MA, MIT Press, Vol. 1 &2, 1986.

[2] J Anderson & E Rosenfeld, (eds) "Neurocomputing: foundations of research", Cambridge, MA, MIT Press, 1988.

[3] T Kohonen, "Self-organisation and associative memory", 3rd Edition, Springer-Verlag, 1989.

[4] K Fukushima, S Miyake, T Ito, "Neocognitron: a neural network model for a mechanism of visual pattern recognition", IEEE Transactions on Systems, Man and Cybernetics, 1983, No 13, pp826-834.

[5] R Hetcht-Nelson, "Neurocomputing", Addison-Wesley, 1990.

[6] B Widrow & M Hoff, "Adaptive switching circuits", in Proceedings of WESCON Convention Record, New York, Vol. 4, 1960, pp 96-104.

[7] P Werbos, "Beyond regression: new tools for prediction and analysis in behavioral sciences", Ph.D. Thesis, Harvard University, Boston, 1974.

[8] M Minsky and S Papert, "Perceptrons", Expanded Edition, MIT Press, 1988.

[9] F Rosenblatt, "Principles of neurodynamics", Spartan Books, 1959.

[10] J Hopfield, "Neural networks and physical systems with emergent collective computational abilities", Proc. of the National Academy of Sciences, USA, 79, 1982, pp 2554-2558.

[11] J Hopfield, "Neurons with graded response have collective computational properties like those of two state neurons", Proc. of the National Academy of Sciences, USA, 82, 1984, pp 3088-3092.

[12] R Lippman, "Introduction to computing with Neural Networks", IEEE ASSP Magazine, 1987, pp4-20.

[13] G Carpenter and S Grossberg, "The ART of adaptive pattern recognition by self-organising neural network", Computer, March 1988, pp77-88.

[14] D Hebb, "Organisation of behaviour", Wiley 1949, Reprinted in J Anderson & E Rosenfeld, (eds) "Neurocomputing: foundations of research", Cambridge, MA, MIT Press, 1988.

[15] W S McCullogh and W Pitts, " A logical calculus of the ideas imminent in logica calculus", Bulletin of Mathematical Biophysics, 1943, Vol. 5, pp115-133.

[16] J Rumbaugh, M Blaha, W Premerlani & W Lorensen, "Object-oriented modelling and design", Prentice-Hall Int. Edition, 1991.

[17] G Booch, "Object-Oriented analysis and design with applications", Second Edition, Benjamin/Cummings publishing Company, 1994

[18] T DeMarco, "Structured Analysis and system specification", Engelwood Cliffs, N.J: Prentice-Hall, 1979.

[19] E Yourdon, "Modern structured analysis", Prentice-Hall International Editions, 1989.

[20] L Constantine & N Henderson-Sellers, "Notation matters part 1: framing the issues", Report on Object Analysis and Design, Vol. 2, No. 3, Sept-Oct 1995, pp25-29.

[21] J Freeman and D Skapura, "Neural networks: algorithms, applications, and programming techniques", Addison-Wesley Publishing Company, 1991.

[22] S Kung, "Digital Neural Networks", PTR Prentice Hall Inc., 1993.

[23] L Rabiner & J Biing-Huang, "An introduction to hidden markov models", IEEE ASSP Magazine, January 1989, pp4-16.

[24] J Biing-Huang & L Rabiner,"Mixture autoregressive hidden markov models for speech signals", IEEE Transactions on Acoustics, Speech and Signal Processing, Vol. 33, No 6, Dec. 1985, pp1404-1413.

[25] A Cichocki & R Ubenhauen, "Neural networks for optimisation and signal processing", John Wiley and Sons, 1993.

[26] J Moody & C Darken, "Faster learning in neural networks of locally-tuned processing units", Neural Computation 1, 1989, pp281-294.

[27] J Hancock, "Data representation in neural networks: an empirical study", in D Touretzky et. Al. (eds), Proceedings of the 1988 connectionist models summer school, Morgan Kaufman, 1988, pp 3-11.

[28] J Moody & C Darken, "Learning with localised receptive fields", in D Touretzky et. al. (eds), Proceedings of the 1988 connectionist models summer school, 1988, pp133-143.

[29] R Rosenfeld & D Touretzky, "A survey of coarse coded symbol memories", in D Touretzky et al (eds), Proceedings of the 1988 connectionist models summer school, 1988, pp256-264.

[30] J Hanson and D Burr, "What connectionist models learn: learning and representation in connectionist neural networks", Brain and behavioural sciences,13(3), pp471-511.

[31] R Duda & P Hart, "Pattern classification and scene analysis", Wiley, New York, 1973.

[32] K Hornik, M Sticchcombe, H White, "Multilayer feedforward networks are universal approximators", Neural Networks, Vol. 2, No 5, 1989, pp259-366.

[33] A Carmago, "Learning algorithms in neural networks", Draft Internal Report, DCC Computer Science Laboratory.

[34] B Kosko, "Bi-directional associative memories", IEEE Transactions on Systems, Man and Cybernetics, SMC-L8, Jan/Feb 1988, pp49-60.

[35] T Kohonen, "Correlation matrix memories", IEEE Transactions on Computers, Vol. c-21, No 4, Apr 1972, pp353-358.

[36] D Tank & J Hopfield, "Collective computation in neuron like circuits", Scientific American, pp62-69.

[37] S Abe, "Global convergence and suppression of spurious states of the Hopfield neural networks", IEEE Transactions on Circuits and Systems, Vol. 40, No 4, April 1993.

[38] J Bezdek, "Pattern recognition with fuzzy objective function algorithms", Plenum Press, 1981.

[39] G Carpenter & S Grosberg, "ART2 Self-Organisation of Stable category recognition codes for analog input patterns", in M. Caudill and C. Butler, (eds), Proceedings of the IEEE first international conference on Neural Networks, San Diego, 1987, ppII-735-II-745.

[40] D Hush & B Horne, "Progress in unsupervised learning", IEEE Signal processing magazine. Vol. 10, No 1, Jan 1993, pp 8-39.

[41] S Kirkpatrick, C Gelatt & M Vecchi, "Optimisation by simulated annealing", in J. Anderson and E. Rosenfeld (eds), "Neurocomputing: Foundations of research", MIT Press, 1988, pp554-567.

[42] D G Luenberger, "Linear and nonlinear programming", Addison Wesley, second edition, 1984.

[43] D Rumelhart, G Hinton & R Williams, "Learning internal representations by error backpropagation", in Rumelhart D. And McClelland J. (Eds), Parallel distributed processing, MIT press, Vol. 1, chap 5, 1986, pp152-193.

[44] P J Werbos, "The roots of backpropagation: from ordered derivatives to neural nets and political forecasting", John Wiley & Sons, 1994.

[45] D Woods, "Back and counter propagation aberrations", Proceedings of the IEEE First International Conference on Neural Networks, San Diego, June 1988, pp 473-479.

[46] S Kung et al, "Generalised perceptron networks with nonlinear discriminant functions", in Mammone J. (eds), Neural Networks, Theory and Applications, Academic, press, 1994, pp245-279.

[47] G Drago & S Ridella, "Statistically controlled weight initialization (SCAWI)", IEEE Transactions on Neural Networks. Vol. 3, No 4, July 1992, pp983-986.

[48] L Maurice, "Cluster analysis for social scientists: techniques for analysing and simplifying complex blocks of data", Jossey-Bass publishers, 1983.

[49] L Lai, **F Ndeh-Che** et al, "HVDC systems fault diagnosis with neural networks", Proceedings of the European Power Electronics and Applications Conference, Vol 8, IEE Pub No 377, UK, 1993, pp145-150.

[50] S Kung and J Hwang, "An algebraic projection analysis for optimal hidden units, size and learning rates in backpropagation learning", IEEE International Conference on Neural Networks, ICNN '88, San Diego, Vol.1, 1988, pp363-370.

[51] S Fahlman & C Lebiere, "The cascade correlation learning architecture", Technical report, CSU CM-CS-91-100, Canergie Mellon University.

[52] A Sankar & R Mammone, "Neural tree networks" in Mammone J. (eds), Neural Networks, Theory and Applications, Academic, press, 1994.

[53] R Reed, "Pruning algorithms, a survey", IEEE Transactions on Neural Networks, Vol. 4, No 5, Sep 1993, pp740-747.

[54] C Jacob & J Rehder, "Evolution of neural net architectures by a hierarchical grammar-based genetic system", in Albrecht R. F. et al (eds) "Artificial Neural Networks and Genetic Algorithms", Proceedings of the 1993 international conference in Innsbruck, Austria, Springer-Verlag, pp72-79.

[55] C Bishop, "Improving the generalisation properties of radial basis function networks", Neural Computation 3, MIT, 1991, pp579-588.

[56] M Servin & F Cuevas, "A new kind of neural networks based on Radial Basis functions", Investigacion, Revista Mexicana de Fisca 39, No. 2, 1993, pp235-249.

[57] P Jokinen, "On the relation between radial basis functions and fuzzy systems", IEEE International Joint Conference on Neural Networks, Vol. 1, 1992, pp220-225.

[58] J Park & I Sandberg, "Approximation and radial basis functions", Neural Computation 5, MIT, 1993, pp305-316.

[59] M Mak, G Allen & G Sexton, "Comparing multi-layer perceptrons and radial basis function networks in speaker recognition", Journal of Microcomputer Applications, Vol. 16, 1993, pp147-159

[60] S Watkins & P Chau, "Different approaches to implementing a radial basis function neurocomputer", IEEE Symposium on Neuroinformatics and Neurocomputers, Vol. 2, 1992, pp1149-1155.

[61] H Graf et. al., "VLSI implementation of a neural network model", Computer, March 1988, pp41-48.

[62] E Georges, L. Lai, **F Ndeh-Che**, H Braun, "Implementation of neural networks with VLSI', Fourth International Conference on Neural Networks, IEE, June 1995, pp489-494.

[63] S Fahlman, "Faster learning variations on backpropagation: an empirical study", in D Touretzky et. al. (eds), Proceedings of the 1988 connectionist model summer school, Morgan Kaufman,1988, pp3-11.

[64] Z Andreas et. al. "SNNS neural network simulator, user manual", version 3.2, 1993

[65] S Yeh and H Stark, "A fast learning algorithm for multilayer neural networks based on projection methods", in J Mammone (eds), Neural Networks, Theory and Applications, Academic, press, 1994, pp323-342.

[66] M Powell, "Restart procedures for the conjugate gradient method", Mathematical programming, Vol. 12, April 1977, pp241-54.

[67] A Blum, "Neural networks in C++: an object-oriented framework for building connectionist systems", John Wiley and Sons, Inc., 1992.

[68] R C Ebehart & R W Dobbins, (eds), "Neural networks PC tools: a practical guide", Academic Press, San Diego, 1990.

[69] M Watson, "C++ power paradigms", McGraw-Hill Book Company, 1994.

[70] R Bindu, "Object-Oriented databases: technology, applications and products", McGraw-Hill, Inc. 1994, pp207-224.

[71] S Gossain, "The emergence of the system architect", Object Expert, SIGS Publications, Vol. 1, No. 1, Nov-Dec 1995, pp58-60.

[72] A Kennedy & C Carter, "Object-Oriented Analysis and Recursive Development: A Formalism for Understanding Software Architectures", Paper and Presentation, IEE Colloquium on "Recent Progress on Object Technology", 1992.

[73] L Peters, "Advance structured analysis and design", Prentice-Hall International Editions, 1988.

[74] I Jacobson, M Christerson, P Johnson & G Övergard, "Object-oriented software engineering: a use case driven approach", Addison-Wesley, 1992.

[75] S Schach, "Classical and object-oriented software engineering", IRWIN, 1996.

[76] A Takang & P Grubb, "Software maintenance: concepts and practice", International Thompson Computer Press, 1996.

[77] J Coplien, "Advanced C++ programming styles and idioms", Addison-Wesley Publishing Company, 1992.

[78] D Duffy, "From chaos to classes: object-oriented software development in C++", McGraw-Hill Book Company, 1995.

[79] N Wilkinson, "Using CRC cards: an informal approach to object-oriented software development", SIGS Books, 1995.

[80] K Derr, "Applying OMT: a practical step-by-step guide to using the object modelling technique", SIGS Books, 1995.

[81] D Papurt, "Inside the object model: the sensible use of C++", SIGS Books, 1995.

[82] **F Ndeh-Che**, L L Lai and K Chu, "The design of neural networks with object-oriented techniques", IEE Colloquium on Recent Progress in Object Technology, Dec 1993.

[83] K S Swarup, H S Chandrasekharraiah, L L Lai, **F Ndeh-Che**, "Application of neural networks to fault diagnosis in HVDC systems", Neural Networks and Genetic Algorithms, Springer Verlag, Wien, New York. 1993, pp227-234.

[84] L L Lai, **F Ndeh-Che**, K S Swarup and H S Chandrasekharraiah, "Fault diagnosis for HVDC systems with neural networks", Pre-prints of papers, Vol. 9, 12th International Federation of Automatic Control (IFAC) world congress, July 1993, Australia, pp179-182.

[85] L L Lai, **F Ndeh-Che**, Tejedo Chari, P Rajroop, and H S Chandrasekharraiah, "HVDC systems fault diagnosis with neural networks", Proceedings of the 5th European Conference on Power Electronics and Applications, The European Power Electronics Association, Vol. 8, Sept 1993, pp145-150.

[86] L L Lai and **F Ndeh-Che**, Tejedo Chari, "Fault identification in HVDC systems with Neural Networks", Proceedings of the Second International Conference on Advances in Power Systems Control, Operations and Management, IEEE, Pub No 388, Dec 1993, pp231-236.

[87] L L Lai and **F Ndeh-Che**, "An application of neural networks to improving power system stability", IEE Colloquium on Advances in Neural Networks for Control System. April 1993.

[88] **F Ndeh-Che,** "Application of neural networks to financial decision making", preliminary report to Marks & Spencers Financial Services, 1994.

[89] L L Lai, **F Ndeh-Che**, H Braun, R Hui and A B Serrano, "Application of neural networks to predicting harmonics", Sixth European Conference on Power Electronics and Applications, Sept 1995, pp533-538.

[90] Y Hsu and C Yang, "Design of artificial neural networks for short term load forcasting. part 1: Self-organising feature maps for day type identification", IEE Proceedings-C, Vol. 138, No. 5, September 1991, pp407-413.

[91] Y Hsu and C Yang, "Design of artificial neural networks for short term load forecasting. part 2: Multilayer feedforward networks for peak load and valley load forecasting", IEE, Proceedings-C, Vol. 138, No. 5, September 1991, pp407-413.

[92] Q Wu, B Hogg and G Irwin, "A neural network for turbogenerators", IEEE Transactions on Neural Networks, Vol. 3, No. 1, Jan 1992, pp95-101.

[93] H Yang, W Chang & C Huang, "A neural network approach to on-line fault section estimation using information of protective relays and circuit breakers", IEEE Transactions on Power Delivery, Vol. 9, No 1, 1994, pp220-229.

[94] S Ebron, D Lubkeman, M White, " A neural network approach to the detection of incipient faults in power distribution feeders", IEEE Transactions on Power Delivery, Vol. 5, No 2, April 1990, pp905-914.

[95] EMTP Rule book, BPA, USA, 1987.

[96] C Philips & H Nagle Jr., "Digital control systems analysis and design", Prentice-Hall, Ch 8, 1984.

[97] E Levin, R Gerwitzman & G Inbar, "Neural network architecture for adaptive system modelling and control", Neural Networks, Vol. 4, pp185-191.

[98] K Narendra & K Parthasarathy, "Identification and control of dynamical systems using neural networks", IEEE Transactions on Neural Networks, Vol. 1, No 1, March 1990, pp4-27.

[99] S Kim & J Lee, "Unknown parameter identification of parameterised system using multi-layered neural networks", IEEE International Conference on Neural Networks, Vol. 1-3, 1993, pp438-443.

[100] H Tsai et. al., "On-line synchronous machine parameter estimation from small disturbance operating data", IEEE Transactions on Energy Conversion, Vol. 10, No. 1, March 1995, pp25-36.

[101] S Bhama & H Singh, "Single layer neural network for linear system identification using gradient descent techniques", IEEE Transactions on Neural Networks, Vol. 4, No. 5, Sept 1993, pp884-888.

[102] E Cox, "The Fuzzy systems handbook: a practitioner's guide to building, using and maintaining fuzzy systems", Academic Press, 1994.

[103] L Zadeh, "Fuzzy sets", Information and Control, Academic Press, Vol. 8, New York, 1965, pp338-353.

[104] M Sugeno, "An introductory survey of fuzzy control", Information Sciences 36, 1985, pp59-83.

[105] "Fuzzy Logic: Frequently Asked Questions", A list of Frequently Asked Questions (FAQ)", USENET : comp.ai.fuzzy. Anonymous FTP from rtfm.mit.edu: /pub/usenet/news.answers/ai-faq/fuzzy/.

[106] R Yager and D Filev, "SLIDE: A simple adaptive defuzzification method", IEEE Transactions on Fuzzy Systems, Vol. 1, No 1, Feb. 1993, pp65-78.

[107] T Yamakawa, "A fuzzy inference engine in non-linear analog mode and its application to a fuzzy logic controller", IEEE Transactions on Neural Networks, Vol. 4, No 3, May 1993.

[108] C Lee, "Fuzzy logic in control systems: fuzzy logic controller-Part I", IEEE Transactions on Systems, Man and Cybernetics, Vol. 20, No 2, 1990, pp404-418.

[109] S Halgamuge, W Poechmueller & M Glesner, "An alternative approach for generation of membership functions and fuzzy rules based on radial and cubic basis function networks", Technical Report of the Institute of Microelectronic Systems, Darmstadt University of Technology, Germany, 1994.

[110] C Lin & C Lee, "Reinforcement structure/parameter learning for Neural Network based fuzzy logic control systems", IEEE Transactions on Fuzzy Systems, Vol. 2, No 1, Feb. 1994, pp46-63.

[111] H Berenji & P Khedkar, "Learning and tuning fuzzy logic controllers through reinforcements", IEEE Transactions on Neural Networks, Vol. 3, No 5, Sept. 1992, pp724-739.

[112] R Backhouse, "Program construction and verification", Prentice-Hall International Editions, 1986.

[113] M Darwish, "Switched capacitor filters power application", PhD thesis, 1987, Brunel University, UK.

[114] K Narendra & H Chandrasekharaiah, "Simple method of selective harmonic tracking (SHT) of signals in an integrated AC-DC power system", IEE Proceedings-C, Vol. 140, No 5, Sept 1993, pp399-403.

[115] A Poularikas, "The transforms and application handbook", CRC Press & IEEE Press, 1993.

[116] H Beides and G Heydt, "Dynamic state estimation of power system harmonics using Kalman filtering methodology", IEEE Tansactions on Power Delivery, Vol. 6, No 4, Oct 1991, pp1663-1670.

[117] R Hartana & G Richards, "Harmonic source monitoring and identification using neural networks", IEEE Transactions on Power Systems, Vol. 5, No 4, Nov 1990, pp1098-1104.

[118] S. Osowski, "Neural network for estimation of harmonic components in a power system", IEE Proceedings, Part C, Vol. 139, No 2, March 1992, pp129-135.

[119] J Holland, "Adaptation in natural and artificial systems", First MIT press Edition, 1992.

[120] J Heitkoetter and D Beasley, eds. (1994) "The hitch-hiker's guide to evolutionary computation: A list of Frequently Asked Questions (FAQ)", USENET : comp.ai.genetic. Anonymous FTP from rtfm.mit.edu: /pub/usenet/news.answers/ai-faq/genetic/.

[121] V Miranda, D Srinivasan and L M Proenca, "Evolutionary computation in power systems", Proceedings of the 12th Power Systems Computation Conference, Germany, Aug 1996, pp25-35.

[122] C Reeves & H Karatza, "Dynamic sequencing of a multiprocessing system: A Genetic Algorithm Approach", in Albrecht R. F. et al (eds) "Artificial Neural Networks and Genetic Algorithms", Proceedings of the 1993 international conference in Innsbruck, Austria, Springer-Verlag, pp491-495.

[123] D Goldberg, "Genetic algorithms in search, optimisation and machine learning", Addison-Wesley publishing company, 1989.

[124] L Davis, "A handbook of genetic algorithms", Van Nostrand Reinhold, New York, 1991.

[125] A Ackley, "A connectionist machine for genetic hill climbing", Kluwer Academic Publishers, 1987.

[126] H Adeli & S Hung, "Machine learning: neural networks, genetic algorithms and fuzzy systems", John Wiley and Sons, Inc. 1995.

[127] L L Lai, **F Ndeh-Che** and K H Chu, "Improving power system stability by selecting the parameters of excitation control systems using a genetic algorithm",

International Conference on Power System Technology, IEEE/CSEE, China, Oct 1994, pp286-290.

[128] L L Lai, **F. Ndeh-Che**, K Chu, P Rajroop, and X F Wang, "Design of neural networks with genetic algorithms for fault section estimation", Proceedings of the 29th UPEC, Ireland, 1994, Vol 2, pp596-599.

[129] **F Ndeh-Che,** "Computational intelligence report", Internal Report, Energy Systems Group, Department of Electrical, Electronic & Information Engineering, City University, 1994.

[130] M Valenzuela-Rendón, The fuzzy classifier system: motivations and first results", in . Schwefel & R. Manner (Eds), Proceedings of the 1st Workshop on Parallel Problem Solver from Nature, PPSN1, Dortmund, October 1990.

[131] S Forrest, "Parallelism and programming in classifier systems", Pitman Publishing, 1991.

[132] D Miranker, "TREAT: A new and efficient match algorithm for AI production systems", Pittman London, 1990.

[133] H Zhou, "A prototype of long-lived, rule-based learning systems", in A. Martelli and G. Valle (eds), Computational Intelligence, I, North-Holland, 1989, pp83-92.

[134] J Holland, "Escaping brittleness: the possibilities of general-purpose learning algorithms applied to parallel rule-based systems", in R. Michalski et. al (eds), Machine Learning Vol. II, M. Kauffman Publishers, 1986, pp593-623.

[135] E Gamma, R Helm, et. al, "Design patterns", Addison Wesley publishing Company, 1994.

[136] R Zitar & M Hassoun, "Neurocontrollers trained with rules extracted by genetic assisted reinforcement learning system", IEEE Transactions on Neural Networks, Vol. 6, no 4, July 1995, pp859-879.

[137] B Stroustrup, "The C++ programming language", Addison,-Wesley, Second Edition, 1991.

[138] H Schildt, "C++: The complete reference", Osborne McGraw-Hill, 1991.

[139] S Lippman,"C++ primer", Addison-Wesley Publishing Company, 1991.

[140] C Malcom et. al., "An emerging paradigm in robot architecture", in Kande T. et. al (eds) Intelligent Autonomous Systems, Vol. 2, 1990, pp545-564.

[141] A Moizer & B Pagurek, "An onboard navigation system for autonomous underwater vehicles", in Hertzberger L and Goren F. (eds), Intelligent Autonomous Systems, North-Holland, 1987, pp449-458.

[142] J Alexander, "On robot navigation using a genetic algorithm", in Albrecht R. F. et al (eds) "Artificial Neural Networks and Genetic Algorithms", Proceedings of the 1993 international conference in Innsbruck, Austria, Springer-Verlag, pp471-478.

[143] C Bentley, "Introducing SSADM4+", National Communications Centre, Blackwell, 1996.

[144] D Coleman, P Arnold, S Bodoff, C Dollin, H Gilchrist, F Hayes and P Jeremes, "Object-oriented development: the fusion method", Prentice-Hall International Editions, 1994.

Appendix A : Key Elements of the Object Modelling Technique (OMT)

The Object Modelling Technique (OMT) is an object-oriented software development methodology/process that models the software system to be built from three related but different viewpoints, each capturing important aspects of the system, but all required for a complete description. The three viewpoints are expressed as three separate modelling types, namely: the object model, the dynamic model and the functional model. OMT also divides the software development cycle into 3 phases: the analysis phase produces an analysis model, the design produces the design model and the implementation phase produces the implementation model or program code. The analysis phase is concerned with modelling the real world. The design phase is concerned with decisions about the subsystems and the overall architecture of the software system while the implementation phase encodes the design in a programming language. Figure A-1 presents an overview of the OMT process.

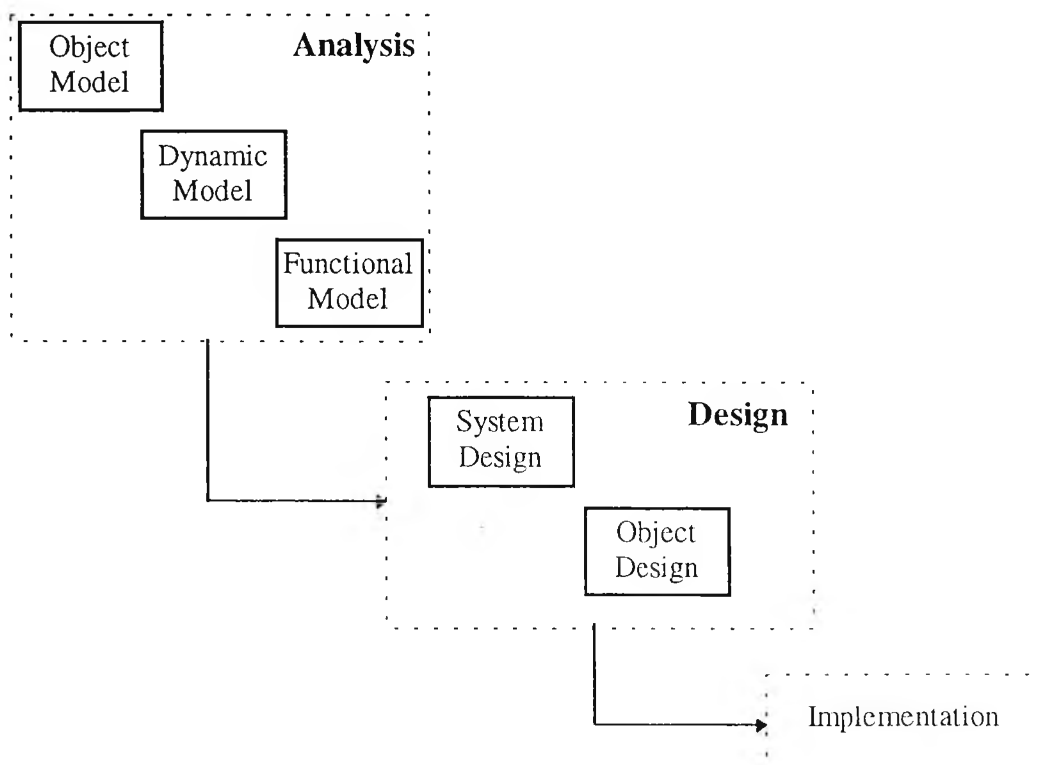


Figure A-1: OMT Process Overview

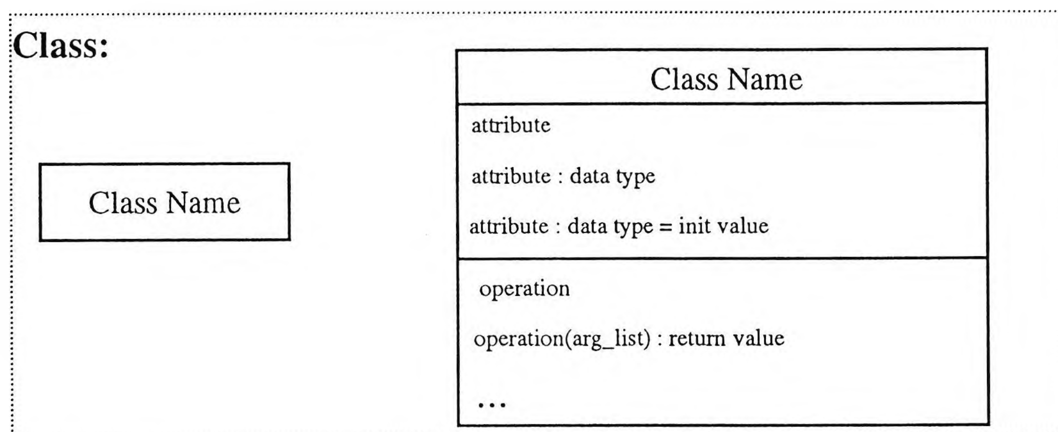
The word model in OMT is used to describe both a view of the software system and a stage in the systems life cycle.

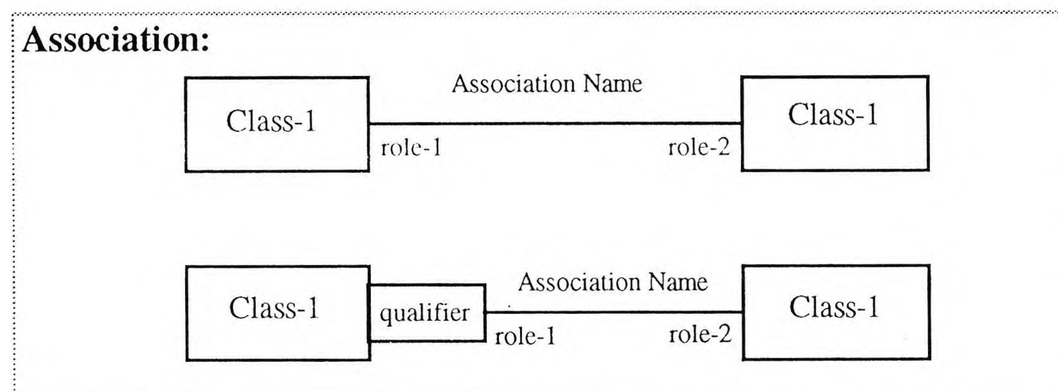
Analysis Models

The Object Model

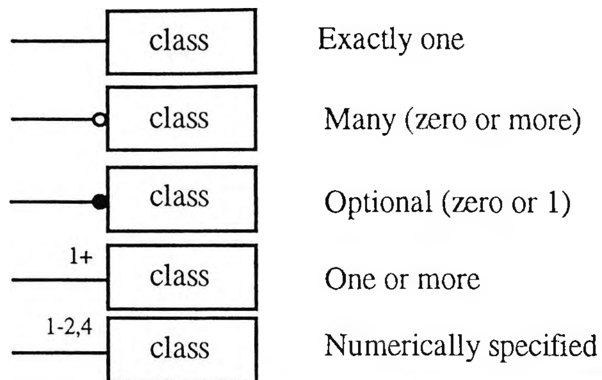
The object model describes the structure of objects in the system—their identity, their relationships to other objects, their attributes and their operations. The goal in constructing an object model is to capture those concepts from the real world which are important to an application. The object model is represented graphically with object diagrams containing object classes.

Object Model Notation

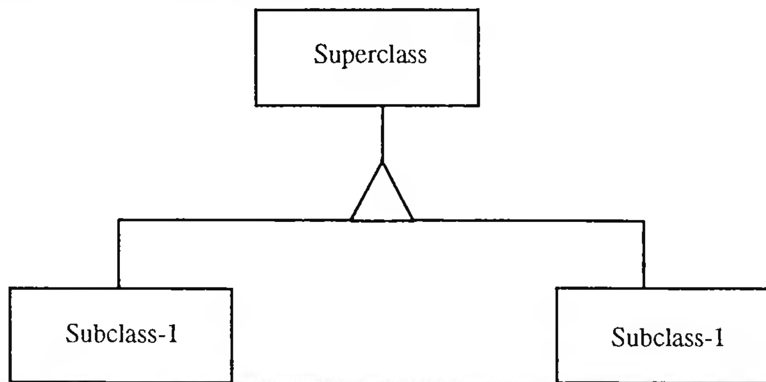




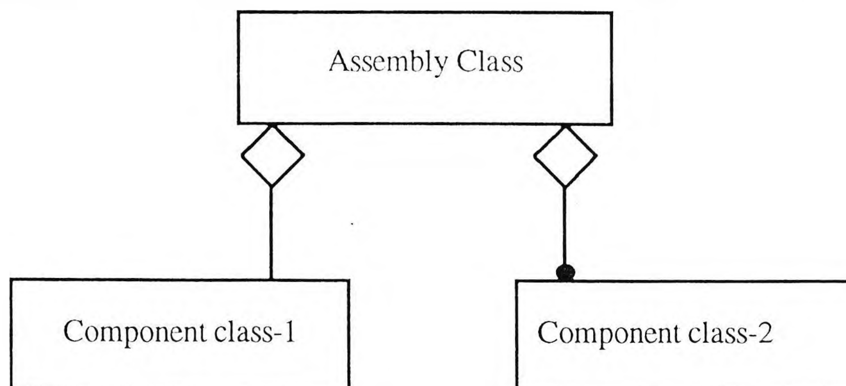
Multiplicity of Associations:



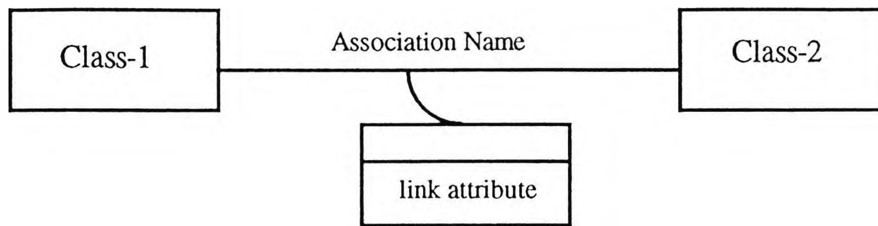
Generalisation (Inheritance):



Aggregation:



Link Attribute:



Object Model Definitions

Object—a tangible entity that exhibits a well defined behaviour and that has meaning for a particular problem domain. An object is characterised by an identity, an interface (behaviour) and a implementation (representation).

Class—a description of a group of objects with similar properties, common behaviour, common relationships and common semantics.

Attribute—a named property of a class describing a data value held by each object of the class.

Operation—a function or transformation that may be applied to objects in a class.

Link—a physical or conceptual connection between two objects; an instance of an association.

Association—a relationship among instances of two or more classes describing a group of links with common structure and common semantics.

Aggregation—a special association between a composite object and its constituent parts.

Generalisation/Specialisation—a relationship or special association between a class and one or more specialised versions of it. The more general class is called the superclass or base class while a specialised version is called the subclass or derived class.

Inheritance—an object-oriented construct that permits classes to share attributes and operations based on generalisation/specialisation relationship between them.

Role—one end of an association.

Role Name—a name that uniquely identifies one end of an association.

Qualifier—an attribute of an object that distinguishes amongst a set of objects in the many end of an association.

Qualified Association—an association that relates two classes and a qualifier. A binary association in which one part is a composite comprising a class and a qualifier and the second part is a class.

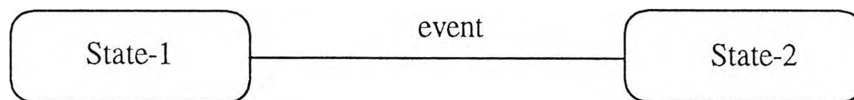
Data Dictionary—a textual description of each class, its associations, attributes and operations.

The Dynamic Model

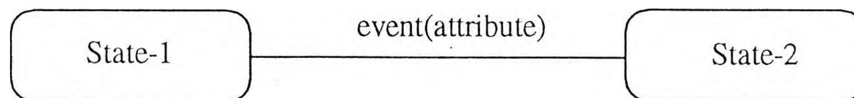
The dynamic model describes the aspects of a system to do with time and the sequencing of operations. The dynamic model is concerned with the thread of control in the system based on the organisation of events and states. Events and sequences of events cause changes to the state of the system (state transitions) while the states provide context for the events. The dynamic model is represented graphically with state diagrams. Each state diagram shows the state and the event sequences permitted in the system for one class of objects.

Dynamic Model Notation

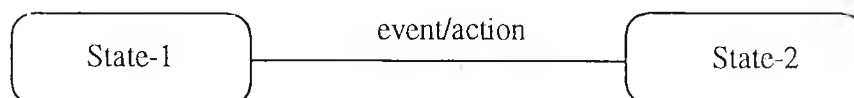
Event causes Transition between States:



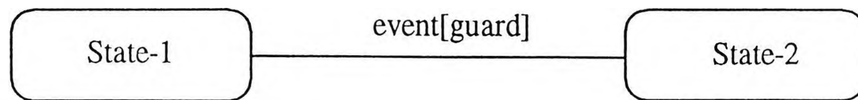
Event with Attribute:



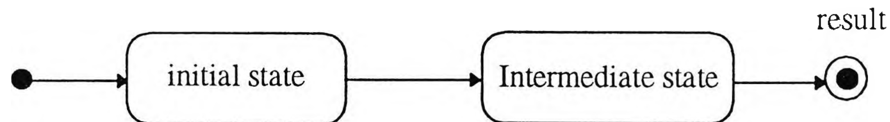
Action on a Transition:



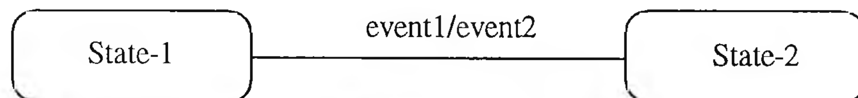
Guarded Transition:



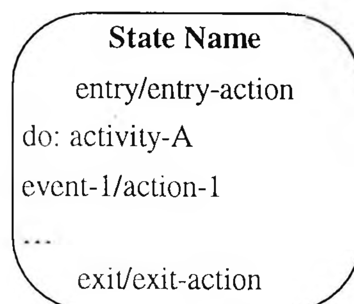
Initial and Final States:



Output Event on a Transition:



Action and Activity while in a State:



Dynamic Model Definitions

Event—an instantaneous occurrence.

Action—an instantaneous operation usually associated with an event.

Activity—an operation that takes time to complete, usually associated with a state and represents real world accomplishments.

Event Attribute—data values conveyed by an event from one object to another.

Event Trace— a diagram that shows the senders and receivers of different events and the sequence in which events are sent.

Transition—a change of state caused by an event.

Control—the aspect of a system that describes the sequences of operations that occur in response to a stimuli.

Guard Condition—a Boolean condition that must be satisfied before a transition can occur.

Guard Transition—a transition that fires only if a guard condition is true.

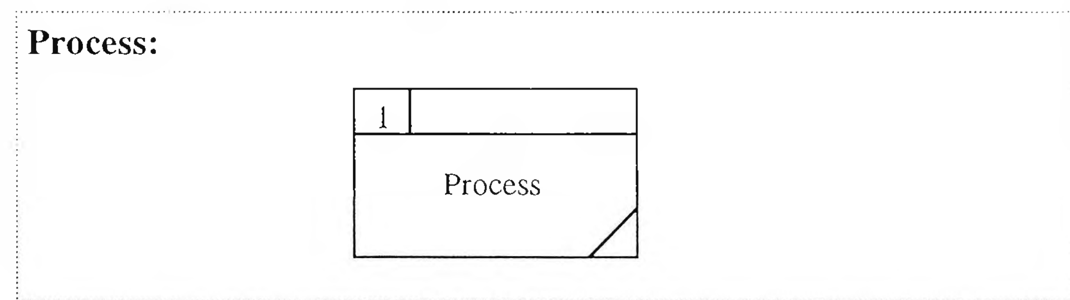
State—the values held by the attributes and the links of an object at a particular time.

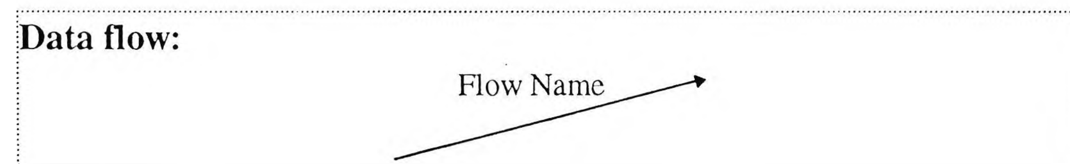
State Diagram—a directed graph in which nodes represent system states and arcs represent the transitions between states.

Functional Model

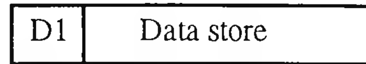
The functional model is concerned with those aspects of a system concern with transformations of values. These include functions, function mappings, constrains and functional dependencies. The functional model captures what a system does without regard for how or when it is done. Functional models is represented with data flow diagrams. A data flow diagram is a modelling tool that allows us to picture a system as a network of functional processes connected to one another by pipelines and holding tanks of data. Data flow diagrams show the dependencies between values, and how output values are computed from input values and functions, without regard for when or if the functions are executed. Components of a data flow diagram include: processes, flows, data stores and terminators.

Functional Model Notation

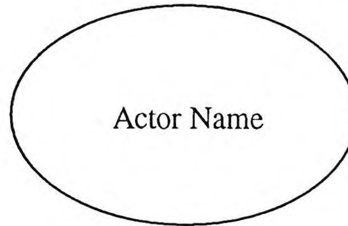




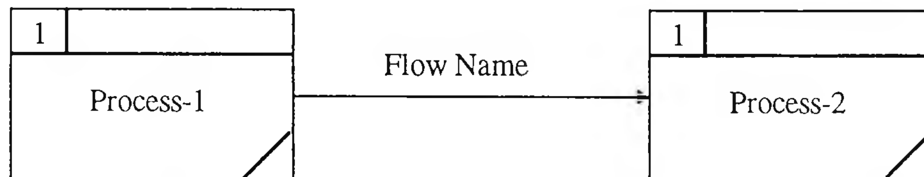
Data Store:



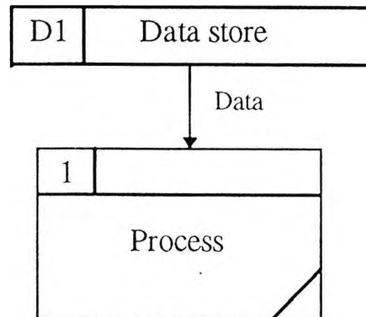
Terminator (Actor):



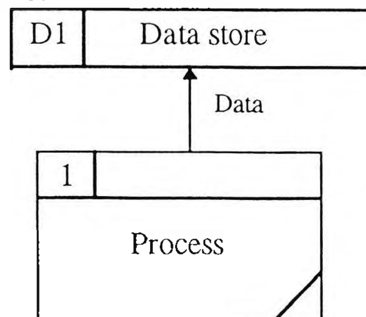
Data flow between Processes:



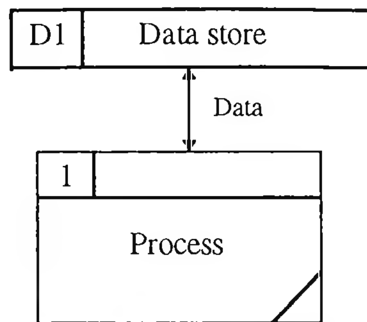
Access of Data Store:



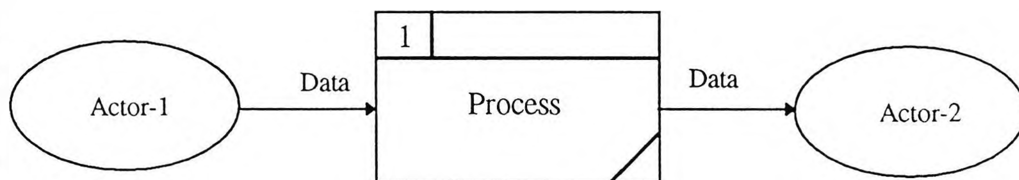
Update of Data Store:



Access and Update of Data Store:



Terminators as sources and sinks of Data Store:



Functional Model Definitions

Process—transformations that represent the individual functions carried out by a system.

Data flow—the connection between the output of one process/data store/external entity and the input to another.

Data Store—a passive entity that stores data that the system must remember over a period of time.

External Entity/Actor—an active entity that dives a data flow diagram by producing or consuming information.

Design Models

System Design

System design is the first design stage in which the basic approach to solving the problem is selected. Decisions are made at a high level about the overall organisation of the system into subsystems that determines the system's architecture. Each subsystem is a package of classes, associations, operations and constraints that are interrelated and that have a well defined interface with other subsystems. Subsystems are identified by the service they provide. A service is a group of related functions that share a common purpose. The interface to a subsystem specifies the form of all interactions and the information flow across subsystem boundaries but does not specify how subsystems are to be implemented

internally. The decomposition of a system into subsystems can be organised as a sequence of vertical partitions or horizontal layers. A layer is an ordered set of virtual worlds, each built in terms of the one below it and providing the basis for the implementation of ones above it. Partitions, on the other hand, divide a system into several weakly coupled system, each providing one kind of service.

Object Design

The object design phase determines the full definitions of the classes and associations used in the implementation as well as the interfaces and algorithms of the methods used to implement the operations. Object design evolves the analysis object model into a system object model by adding detail and making implementation decisions. Redundant objects may be added to the model for efficiency reasons. Also, during object design, decisions have to be made about how to best implement the operations in the functional model need and the choice of algorithms to use for their implementation. Complex operations need to be successively decomposed into simpler operations until they can be implemented as methods in the classes.

Implementation Model

The implementation model is the finish program in an object-oriented programming language such as C++ or Smalltalk. During implementation, fully specified objects in the design model are converted into classes in the programming language. Object-oriented programming languages include direct support for object modelling concepts. For example, C++ classes for object model objects, pointers/references for aggregation relationships, constructors and destructors for object creation and termination models, const for object immutability in the object model, inheritance for generalisation/specialisation relationships in the object model etc.

Appendix B : Speed of Convergence of Algorithms

The study of speed of converge is an important but very complex subject. There is however, a rich and yet elementary theory of convergence rates that makes it possible to predict the relative convergence rate of a wide class of algorithms with confidence. There are a number of concepts that form the basis of measurements for speed of convergence.

Order of Convergence

Given a sequence of real numbers $\{r_k\}_{k=0}^{\infty}$ converging to the limit r^*, several notions relating to the speed of convergence of such a sequence can be defined.

Definition. Let the sequence $\{r_k\}$ converge to r^*. The order of convergence is defined as the supremum of the nonnegative numbers p satisfying

$$0 < \overline{\lim_{k\to\infty}} \frac{|r_{k+1} - r^*|}{|r_k - r^*|^p} < \infty \qquad \text{(AB-1)}$$

The above equation is stated in terms of limit superior rather than just limit to ensure that the definition is applicable to any sequence. It should be noted that the order of convergence is as with all other notions related to speed of convergence that are introduced, is determined solely by the properties of the sequence that hold as $k \to \infty$, referred to as the tail of the sequence. The order of convergence is thus a measure of how good the worst part of the tail is. Larger values of the order p imply faster convergence since the distance from the limit r^* is reduced, at least in the tail, by the pth power in a single step. If the sequence has order p and the limit

$$\beta = \lim_{k\to\infty} \frac{|r_{k+1} - r^*|}{|r_k - r^*|^p} \qquad \text{(B-2)}$$

exists, then asymptotically we have $|r_{k+1} - r^*| = \beta |r_k - r^*|^p$.

Linear Convergence

If the sequence $\{r_k\}$ converges to r^* in such a way that

$$\lim_{k \to \infty} \frac{|r_{k+1} - r^*|}{|r_k - r^*|^p} = \beta < 1, \tag{B-3}$$

the sequence is said to converge linearly to r^* with *convergence ratio* β. Linear convergence is the most important type of convergence behaviour. A linearly convergent sequence, with convergence ratio β can be said to have a tail that converges as fast as the geometric sequence $c\beta^k$ for some constant c.

Average Rates

Definition: Given a sequence $\{r_k\}$ that converges to r^*, the *average order* of convergence is the infinum of the numbers p>1 such that

$$\overline{\lim_{k \to \infty}} |r_k - r^*|^{\frac{1}{p^k}} = 1 \tag{B-4}$$

This order is infinity if the equality holds for no p > 1.

The most important case is that of unity order of convergence, and in this case the *average convergence ratio* is defined as $\overline{\lim_{k \to \infty}} |r_k - r^*|^{\frac{1}{p^k}}$

Convergence of Vectors

Given a sequence $\{\mathbf{x}_k\}_{k=0}^{\infty}$ of vectors in E^n that converges to a vector $\mathbf{x}^*$, the convergence properties are defined w.r.t a some particular function that converts the sequence of vectors into a sequence of numbers. Thus, if f is a continuos function in E^n, the convergence properties of $\{\mathbf{x}_k\}$ can be defined w.r.t to f by analysing the convergence of $f(\mathbf{x}^k)$ to $f(\mathbf{x}^*)$. The function f used in this way to measure convergence is called the error function. In optimisation theory, it is common to choose the error function by which to measure convergence as the same function that defines the objective function of the original optimisation problem. Convergence is then regarded as a measure of how fast the optimisation function converges to a its minimum. Generally, the order of convergence of a sequence is insensitive to the particular error function used; but for step-wise linear convergence, the average convergence ratio is not.

The Method of Steepest (Gradient) Descent

Let f have continuos first partial derivatives in E^n. For convenience, the following simplifying notation has been assumed for the gradient vector $\nabla f(\mathbf{x})$ of f. $\nabla f(\mathbf{x})$ is defined

as n n-dimensional *row* vector; $\mathbf{g}(\mathbf{x}) = \nabla f(\mathbf{x})^T$ is defined as n-dimensional *column* vector. When there is no ambiguity, $\mathbf{g}(\mathbf{x}_k) = \nabla f(\mathbf{x}_k)^T$ is written as $\mathbf{g}_k$.

The method of steepest descent is written as the iterative algorithm

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \mathbf{g}_k \qquad \text{(B-5)}$$

where α_k is a nonnegative scalar minimising $f(\mathbf{x}_k - \alpha \mathbf{g}_k)$. The steepest descent algorithm can be described as a search from the point $\mathbf{x}_k$ along the direction of the negative gradient $\mathbf{g}_k$ for the minimum point on this line which is taken to be $\mathbf{x}_{k+1}$.

In formal terms, the overall algorithm $\mathbf{A}: E \to E^n$ which gives $\mathbf{x}_{k+1} \in \mathbf{A}(\mathbf{x}_k)$ can be decomposed in the form $\mathbf{A} = \mathbf{SG}$. Here, $\mathbf{G}: E^n \to E^{2n}$ is defined by $\mathbf{G}(\mathbf{x}) = (\mathbf{x}, -\mathbf{g}(\mathbf{x}))$ giving the initial point and the direction of a line search. This is followed by line search $\mathbf{S}: E^{2n} \to E^2$.

Global Convergence

If $\nabla f(\mathbf{x}) \neq \mathbf{0}$ then $\mathbf{S}$ is closed and since it is clear that $\mathbf{G}$ is continuos, the steepest algorithm $\mathbf{A}$ is closed. The solution is defined to be the points $\mathbf{x}$ where $\nabla f(\mathbf{x}) = \mathbf{0}$. Any continuos real valued function $Z(\mathbf{x}) = f(\mathbf{x})$ is a descent function for $\mathbf{A}$, since for $\nabla f(\mathbf{x}) \neq \mathbf{0}$

$$\min_{0 \le \alpha < \infty} f(\mathbf{x} - \alpha \mathbf{g}(\mathbf{x})) < f(\mathbf{x}) \qquad \text{(B-6)}$$

Thus, by the Global Convergence Theorem, if the sequence $\{\mathbf{x}_k\}$ is bounded, it will have limit points and each of these is a solution.

Appendix C : Derivation of the Backpropagation algorithm

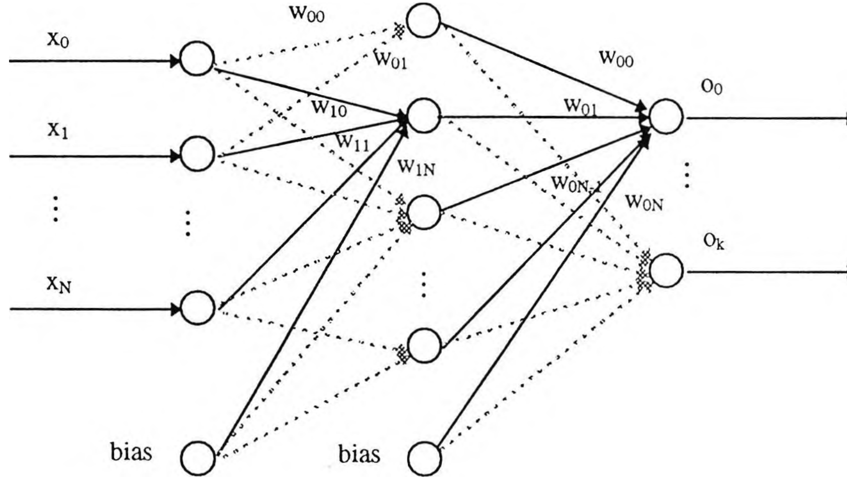


Figure A-1: Multilayer Perceptron neural network trained by backpropagation

Consider the multilayer perceptron network in Figure A-1, with N input neurons, H hidden neurons and K output neurons. The input vectors are $\mathbf{x}_p \in \Re^N$, the weight matrix $\mathbf{W} = \{\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_H\}$ where $\mathbf{w}_p \in \Re^N$ and target vectors $\mathbf{t}_p \in \Re^K$, the error due to the pth training pattern is given by

$$E_p = \sum_{i=0}^{k-1} \left(t_{pi} - o_{pi}\right)^2 \qquad \text{(C-1)}$$

An input vector $\mathbf{x}_p = (x_{p1}, x_{p2}, \ldots x_{pN})^t$, is applied at the input of the network. The input neurons distribute the values to the hidden layer neurons. The net input to the jth hidden neuron is given by

$$net_{pj}^h = \sum_{i=0}^{N-1} w_{ji}^h x_{pi} + \theta_j^h \qquad \text{(C-2)}$$

where w_{ji}^h is the weight on the connection from the ith input neuron and

θ_j^h is the optional bias term.

The output of the hidden neurons is non-linear function of the input

$$o_{pj} = f_j^h(net_{pj}^h) \qquad \text{(C-3)}$$

where f is any activation function which is monotone, non-decreasing and differentiable in the required range. The hidden neurons in turn distribute their activation values to the output neurons. The processing equations for the output neurons are given by

$$net_{pk}^{o} = \sum_{j=0}^{H-1} w_{kj}^{o} o_{pj} + \theta_{k}^{o} \tag{C-4}$$

$$o_{pk} = f_{k}^{o}(net_{pk}^{o}) \tag{C-5}$$

The error at a single output neuron is the difference between its actual output and the desired value and is given by $\delta_{pk} = t_{pk} - o_{pk}$.

The performance function is taken to be the sum of squared errors for all output neurons

$$E = \frac{1}{2}\sum_{p=1}^{P} \mathrm{E}_{p} = \frac{1}{2}\sum_{p=1}^{P}\sum_{k}\delta_{pk}^{2} = \sum_{p=1}^{P}\sum_{k}(d_{pk} - o_{pk})^{2} \tag{C-6}$$

The weight values are updated in a direction that reduces the total error at the output. This is done by calculating the negative gradient of the total error, $\nabla \mathrm{E}_{p}$, with respect to the weights, w_{kj}. This gradient is calculated for all the connection weights in the network. The magnitude of the weight change is taken to be proportional to the negative gradient.

$$\Delta w_{kj} = -\eta \frac{\partial E_{p}}{\partial w_{kj}} \tag{C-7}$$

where n is a scale factor. The partial derivatives can be evaluated as a function of net_{pk}^{o} using the chain rule as follows

$$\frac{\partial E_{p}}{\partial w_{kj}} = \frac{\partial E_{p}}{\partial net_{pk}^{o}} \frac{\partial net_{pk}^{o}}{\partial w_{kj}} \tag{C-8}$$

but

$$\frac{\partial net_{pk}^{o}}{\partial w_{kj}^{o}} = \frac{\partial}{\partial w_{kj}^{o}}\left(\sum_{j=0}^{H-1} w_{kj}^{o} i_{pj} + \theta_{k}^{o}\right) = o_{pj} \tag{C-9}$$

and writing a delta term $\partial_{k} = -\frac{\partial E_{p}}{\partial net_{pk}^{o}}$ we have $\Delta w_{kj} = \eta \partial_{k} o_{pj}$. The chain rule is used again to evaluate ∂_{k} as follows

$$\partial_{k} = -\frac{\partial E_{p}}{\partial net_{pk}^{o}} = -\frac{\partial E_{p}}{\partial o_{pk}} \frac{\partial o_{pk}}{\partial net_{pk}^{o}} \tag{C-10}$$

where $\frac{\partial E_{p}}{\partial o_{pk}} = -(t_{pk} - o_{pk})$ and $\frac{\partial o_{pk}}{\partial net_{pk}^{o}} = f'\left(net_{pk}^{o}\right)$.

Thus, for any node in the output layer, the following can be written

$\partial_{k} = (t_{pk} - o_{pk}) f'(net_{pk}^{o})$ and $\Delta w_{kj} = \eta \partial_{k} o_{pj}$.

For nodes in the output layer, $\frac{\partial E_p}{\partial o_{pk}} = -(t_{pk} - o_{pk})$ can be directly calculated. For hidden layer nodes, this is not possible and so the error at the output must be expressed in terms of known quantities. The delta term for the hidden layer nodes may be expressed as

$$\partial_j = -\frac{\partial E_p}{\partial net^h_{pj}} = -\frac{\partial E_p}{\partial o_{pj}} \frac{\partial o_{pj}}{\partial net^h_{pj}} \tag{C-11}$$

The above equations can be decomposed as follows

$$\frac{\partial E_p}{\partial o_{pj}} = \frac{\partial E_p}{\partial o_{pk}} \frac{\partial o_{pk}}{\partial net^o_{pk}} \frac{\partial net^o_{pk}}{\partial o_{pj}} = -\sum_k \partial_k w_{kj} \tag{C-12}$$

while

$$\frac{\partial o_{pj}}{\partial net^h_{pj}} = f'(net^h_{pj}) \tag{C-13}$$

therefore,

$$\partial_j = f'(net^h_{pj}) \sum_k \partial_k w_{kj} \tag{C-14}$$

Thus the required change in weight in the hidden layer is $\Delta w_{ji} = \partial_j x_{pj}$

In general, the weights on the connections in the output layer are updated according to

$$w^o_{kj}(t+1) = w^o_{kj}(t) + \eta \delta^o_{kj} i_{pj} + \alpha \Delta_p w^o_{kj}(t-1) \tag{C-15}$$

where η is the learning rate factor

α is the momentum factor and

δ is the error at the output of the neuron and is given by

$$\delta^o_{pk} = (d_{pk} - o_{pk}) f^{o'}_k (net^o_{pk})$$

This error is recursively fed back through to the lower layers of the network and used to determine the appropriate weight changes for each layer.

Appendix D : Object-Oriented Design Methodologies

The number of object-oriented development methodologies have increased markedly in recent years as a result of the current high interest in object technology. Each development method is usually supported by a process and has its own set of notations. The underlying process describes how object-oriented software can be developed using the method. The notation acts as a user interface to the method. The main object-oriented development methodologies include:

- The Object Modelling Technique (OMT)
- Object-Oriented Software Engineering (OOSE)
- Object-Oriented Design (Booch)
- Responsibility Driven Design (CRC Cards)
- Object-Oriented Development: The Fusion Method (HP)
- Object-Oriented Systems Analysis (Schlaer/Mellor)
- Object-Oriented Analysis (Coad/Yourdon)
- Hierarchical Object-Oriented Design (HOOD)

The key elements of the Object Modelling Technique have already been presented in **Error! Bookmark not defined.**. This section describes the other two most common object-oriented development methodologies, namely Object-Oriented Software Engineering and Booch Object-Oriented Design.

Object-Oriented Software Engineering (OOSE)

OOSE as a method is based on the notion of a Use Case. A Use Case is a sequence of transactions between the system and an Actor, which is carried out to achieve some goal. Actors can be users or other systems interacting with the system.

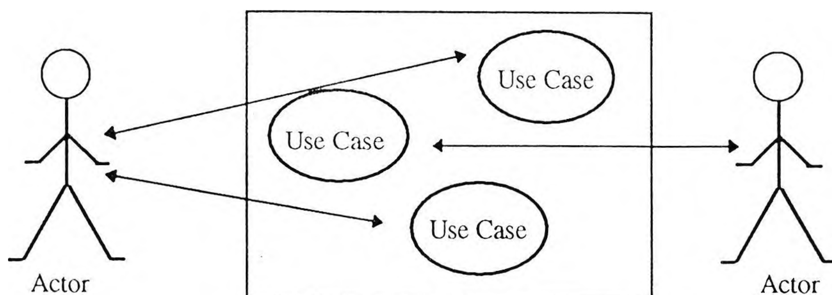


Figure A-1: A Use Case model in OOSE

The underlying process for object-oriented software development is known as Objectory. Objectory partitions the software development process into 3 phases: requirements,

analysis and construction represented by the requirements, analysis and construction models respectively. A overview of the Objectory process is shown in Figure A-2.

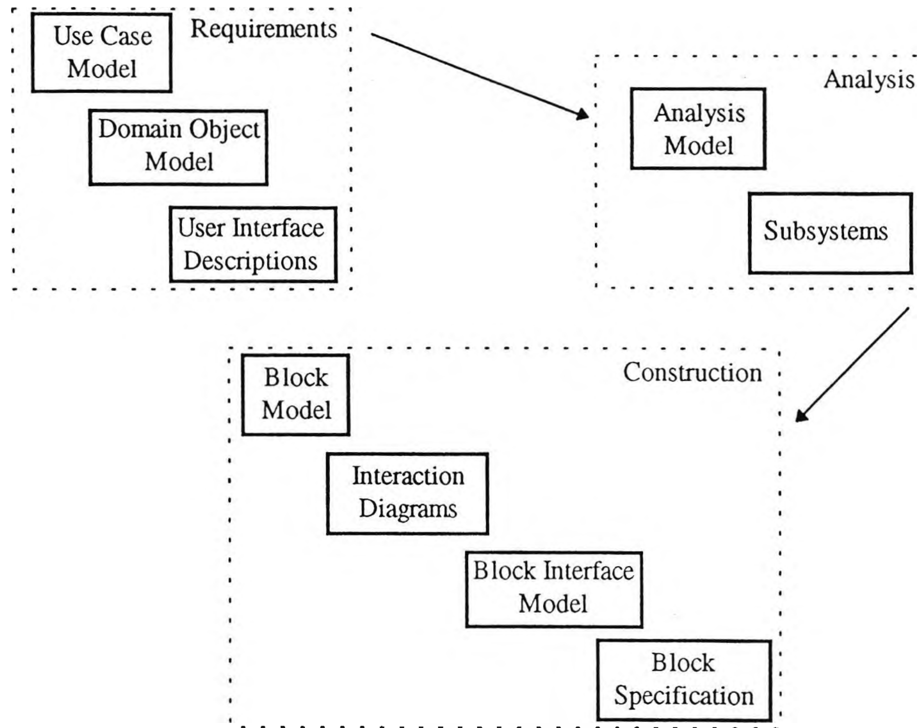


Figure A-2: OOSE process overview

The requirements model captures the functional requirements from the users. It consists of a Use Case model, interface descriptions and a domain object model. The analysis model aims to structure the system from its requirements independently of how it will be implemented. It uses 3 object types to describe the system: an entity object, an interface object and a control object . Figure A-3 shows the different object types in Objectory.

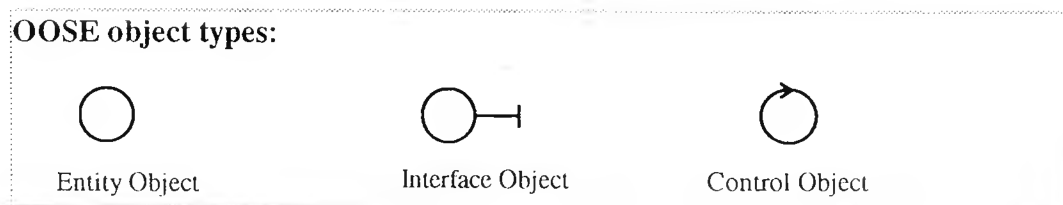


Figure A-3: Object types in OOSE

The construction phase consists of the design, implementation, and test models. During design, a block model is constructed that mirrors analysis domain objects. The block model is then refined into design objects. Object interaction diagrams are created for each Use Case in the requirements model. An object interaction diagram documents all objects that participate in a particular Use Case. It is formalised to describe all stimuli (events) sent between objects and specifies the effect of each operation on the different objects.

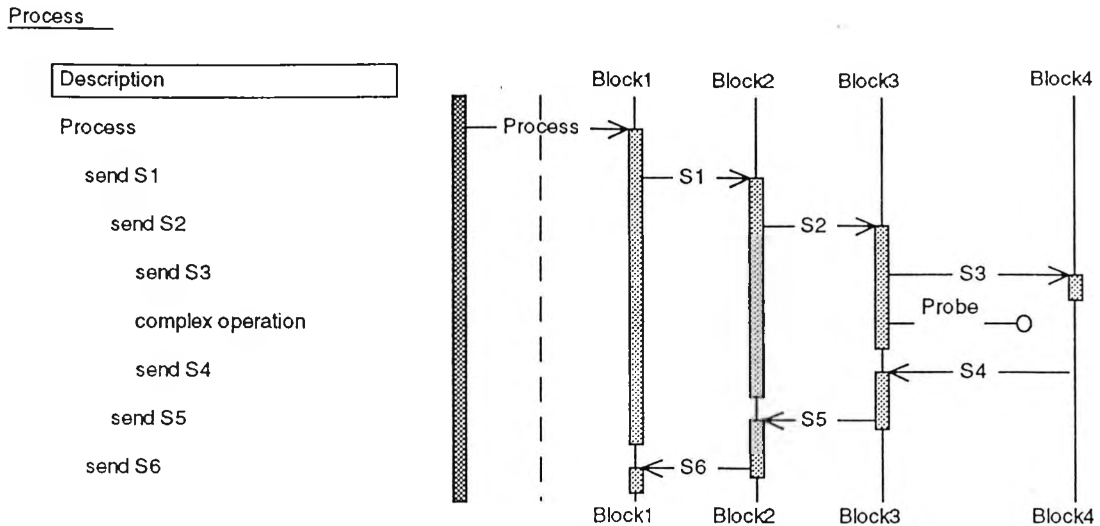


Figure A-4: An example Object Interaction Diagram in Objectory

Booch Object-Oriented Design

The Booch method approaches object-oriented software construction as an iterative process repeated over the analysis and design stages until the correct system has been designed. Figure A-5 shows an overview of the Booch object-oriented development process.

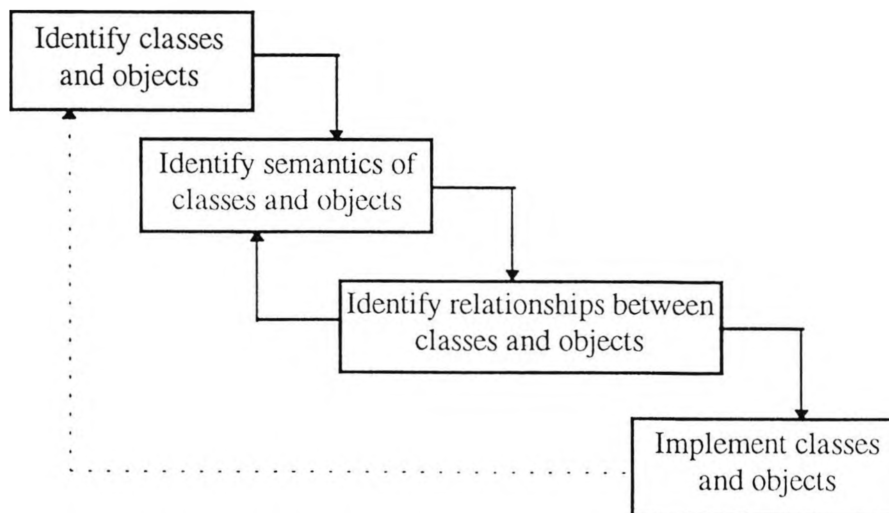


Figure A-5: Booch Process Overview

In the first stage of development, the objects and classes that form part of the application are identified. Different ways are proposed for identifying objects, namely: domain analysis, expert knowledge, textual analysis. The second development stage constructs the class interface. The third development stage deals with identifying the relationships between objects by organising the objects into class or object hierarchies. The discovery of

relationships usually cause new interfaces to be added and so the second and third development stages are iterated until a satisfactory state is achieved. The implementation stage decides on the representation of the class attributes and methods. This in turn may result in the whole process being repeated for an individual class.

The Booch Notation

The Booch method is very descriptive in nature and provides six different modelling diagrams for describing the different aspects of a system's architecture. These include: object diagrams, class diagrams timing diagrams, state diagrams, module diagrams and process diagrams.

Class Diagrams

Class diagrams show classes and their relationships. A class diagram is a notational variation of an entity-relationship diagram to include inheritance, instantiation and using relationships. The notation allows class categories that group classes into namespaces so that large complex systems can be modelled. Classes are depicted by an amorphous blob icon. Four different relationship types can be defined between objects. These include: association, inheritance, has and using relationships. Figure A-6 shows an example class diagram with the different types of relationships.

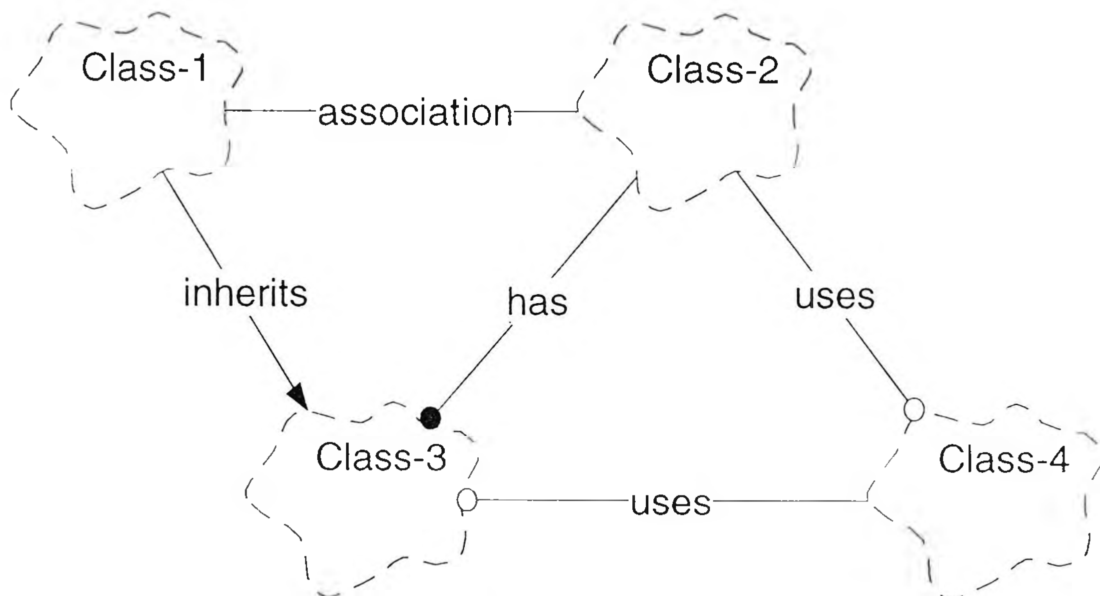


Figure A-6: An example of a Booch class Diagram showing the different relationships

Object Diagrams

An object diagram shows objects and their relationships. The difference between class and object diagrams is manifested in the nature of their relationships. While class relationships

are static, object relationships are dynamic and vary widely during the life of a system as the objects are created and destroyed. Object diagrams depict the behaviour of typical objects by showing the different objects and the relationships between them. There are annotations to show visibility between objects, object sharing semantics and synchronising information necessary in real time systems. Recently, the method has evolved to include notations for depicting systems with distributed architectures including client/server systems. The method emphasises the discovery of key mechanisms in the design. A key mechanism is described as any structure whereby objects work together to provide some behaviour which satisfies a requirement of the problem.

Timing Diagrams

While object diagrams show possible communication between objects, the flow of control and timing of operations can be depicted on a timing diagram. A timing diagram has time on the abscissa and different objects on the ordinates. Lines in the diagram represent the flow of control between objects.

State Diagrams

State transition diagrams show how instances of objects move one state to another when events are received and the actions that occur as a result of the state changes. An example of a Booch state diagram is shown in Figure A-7. To prevent a proliferation of states for large complex systems, a similar notation to OMT has been suggested for representing state transition diagrams.

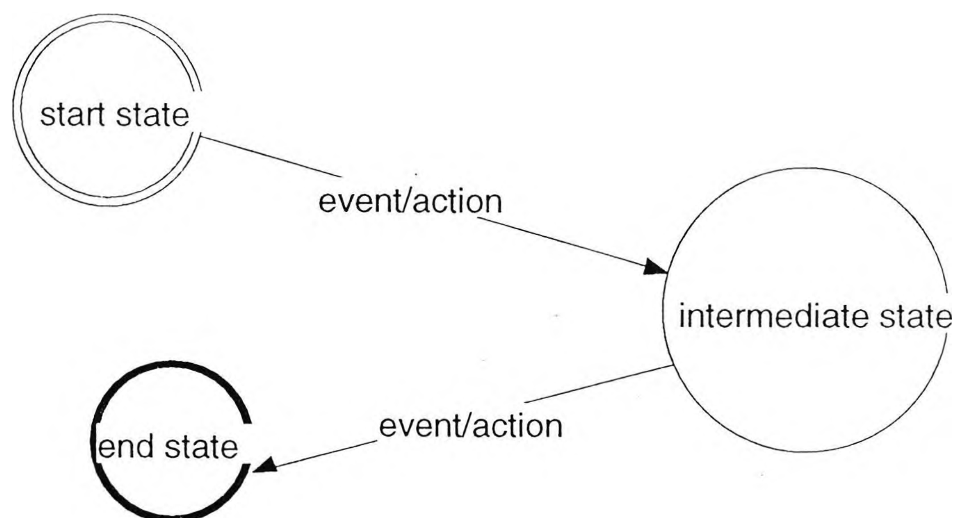


Figure A-7: A Booch state diagram

Module and Process Diagrams

The Booch method distinguishes between logical and physical views of a system. The logical view. Class, object, timing and state diagrams describe the logical view of a system. The logical view is concerned with the different objects that exist and how the collaborate to solve a given problem. The physical view describes the physical hardware and software components of the system. Design decisions about where classes should be declared and the allocation of physical resources such as processors to processes come under the physical view of the system. Module and process diagrams are simple graphs produced during the implementation phase to describe a physical view of the system. A module diagram shows the allocation of classes to modules and the compile-time dependency relationships between modules. A process diagram shows the communication connections between processors and other physical devices.

Other Object-Oriented Development methodologies

Responsibility driven design is an exploratory method based on CRC (Class Responsibility Collaborator) cards in which project teams enact typical scenarios that exist in the system in order to identify classes and their interactions. The classes, their sub and super classes as well as their operations and interactions are recorded on index cards which form the bases of the techniques.

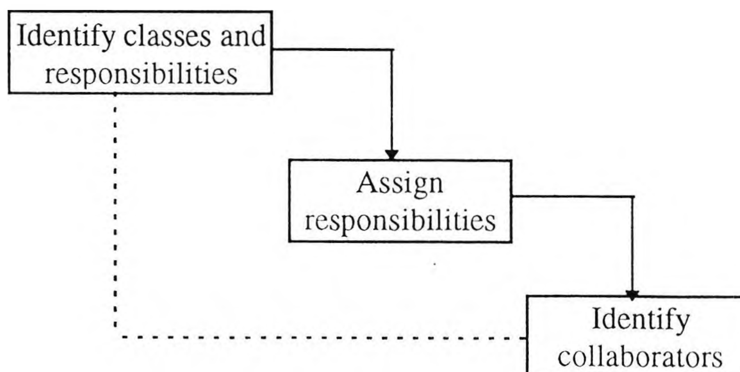


Figure A-8: CRC Process overview

The fusion method for object-oriented [144] software development is a full coverage method that incorporates some aspects of the different object oriented development methods in addition to using formal methods. Fusion divides the object-oriented software development process into 3 phases: analysis design and implementation. Analysis is done with the aid of 3 modelling types: an object model, an operation model and a life cycle

model. In the design phase, three other modelling types are used. These include object interaction graphs, visibility graphs and class descriptions and inheritance graphs.

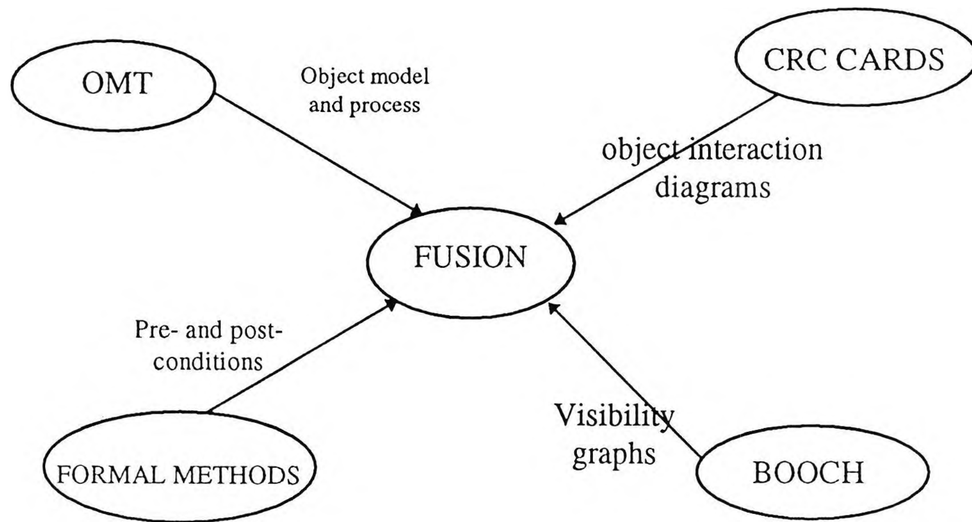


Figure A-9: Fusion process of object-oriented development methods

Appendix E : Class Declarations for Associative Neural Network classes

The header files present class descriptions for the class of fixed weight neural networks. These are the associative memory networks. The following networks can be categorised as associative memory networks:-

- Feedforward Associative Memory Networks
    - Linear Associative Memory (LAM)
    - Non Linear Associative Memory (NLAM)
    - Hamming Networks
- Feedback associative Memory Networks
    - Sequential Hopfield Model
    - Parallel Hopfield Model
    - Bi-directional Associative Memory (BAM)

```
/* ASSOCIATIVE neural networks are a class of fixed wt neural networks. That     #
# include Bidirectional Associative Memories, (BAM), both Linear and non-linear  #
# Associative Memories, Hamming networks, sequential and parallel Hopfield       #
# models. sizex and sizey in these networks is equivalent to the number of       #
# inputs and outputs respectively.      Author: F. Che                           */

#ifndef ASSOC_H_
#define ASSOC_H_

#include "vecpair.h"
#include "matrix.h"
#include "param.h"

#ifndef BOOLEAN
#define BOOLEAN
enum Bool {False=0, True };
#endif

class Assoc {
protected:
        Matrix *weights;                // matrix of stored weights
        Vector *threshold;              // vector f threshold elements.
        int sizex;                      // number of rows in weights matrix
        int sizey;                      // number of columns in weight matrix
        Bool Binary;                    // true Binary inputs
public:
        Assoc() : weights(0), threshold(0) {}
        void init(const Param&) {}
        Assoc(int sz, Bool bin = True):             // first constructor
        sizex(sz),sizey(sz),
        weights( new Matrix(sz,sz)),
        threshold (new Vector(sz)),
        Binary(bin) {}
        Assoc(int sz, int sy, Bool bin = True):
        sizex(sz),sizey(sy),
        weights( new Matrix(sz,sy)),
        threshold (new Vector(sz)),
        Binary(bin) {}
        Assoc(const Patterns &p, Bool bin =True):Binary(bin)
        {
                sizex  = (p[0].VectorX()).Size();       // query first elt of Patterns for the
                sizey  = (p[0].VectorY()).Size();       // get sizes of X and Y vectors
                weights =  new Matrix(sizex,sizey);     // initialise the weights matrix
                threshold = new Vector(sizex);          //initialise threshold vector
        }

        virtual void initwts(const Patterns&) = 0;              // matrix of input patterns
        virtual ~Assoc()
        {
                delete weights;
                delete threshold;
        }
```

```
        virtual void print(ostream& s= cout) const
        {       s << sizex << " " << sizey <<" " << (Binary ? 1 : 0 ) << endl;
                weights->print(s);
                threshold->print(s);
        }
        virtual void scan(istream &s)
        {
                int bin;
                s >> sizex >> sizey >>bin;
                Binary = bin ? True : False;
                weights->scan(s);
                threshold->scan(s);
        }

        virtual Vector recall(const Vector&) const = 0;
        virtual Matrix recall(const Matrix &m) const =0;
};

inline ostream& operator<<(ostream &s, const Assoc &net)
{
        net.print(s);
        return s;
}
/* Assoc constructors: The patterns object is used to supply the dimensions of the
weights and threshold objects. This is more secure than passing the sizes separately but
means more work has to be done to retrieve the information required. */
#endif

/*  Linear associative memory (LAM) class. This is part of a class of          #
#   fixed wt networks. derived from the associative memory class.              #
#   Author: F. Che                                                             */

#ifndef LAM_H_
#define LAM_H_

#include "assoc.h"
class LAM : public Assoc  {
        public:
        LAM(int sz, Bool bin =False):
        Assoc(sz, bin)  {}
        LAM(int sz, int sy, Bool bin =False):
        Assoc(sz, sy, bin)      {}
        LAM(const Patterns& p, Bool bin =False):
        Assoc(p, bin)   { initwts(p); }
        ~LAM() {}
        void initwts(const Patterns&);
        Vector recall(const Vector&) const;
        Matrix recall(const Matrix &m) const ;
};

#endif
```

```
/* Bidirectional associative memories perform pattern association (hetero)        #
#  BAM's include the parallel hopfield as a special case.                          #
# Author: F. Che                                                                   */
#ifndef BAM_H_
#define BAM_H_

#include "assoc.h"

class BAM : public Assoc {
protected:
        Matrix *tweights;                       // second layer weights
        Vector *tthreshold;                     // second layer threshold
        float energy;                           // minimum energy of the network
public:
        BAM(): Assoc(),
        energy(0),
        tweights(0),
        tthreshold(0) {}
        BAM(int sx, int sy, Bool bin = False):
        Assoc(sx, sy, bin),
        tweights(new Matrix(sy,sx)),
        tthreshold(new Vector(sy))
        { }
        BAM(const Patterns &p, Bool bin =True) :
        Assoc(p, bin),
        tweights(new Matrix(sizey, sizex)),    // transpose of weights
        tthreshold(new Vector(sizey))          //
        {
                initwts(p);
        }

        ~BAM() {
                delete tweights;
                delete tthreshold;
        }
        void initwts(const Patterns&);
        void print(ostream &s = cout) const;
        void scan(istream &s = cin);
        Vector recall(const Vector&) const;
        Matrix recall(const Matrix&) const;
};
#endif
```

```
/* Non-linear associative memory NLAM class. This is part of a class of          #
# fixed wt networks. It is derived from the associative memory class.           #
# It employs a non-linear processing unit which helps reduce unwanted           #
# perturbations. During retrieval, the test pattern vector is used to           #
# compute a (matching) score vector S which passes thru the non-linear          #
# processing units to produce a binary decision vector, v. Two non-linear       #
# operators, threshold circuit and maxnet are defined. This type of             #
# network is capable of holographic retrieval on real valued inputs             #
 #Author F Che                                                                  */

#ifndef NLAM_H_
#define NLAM_H_

#include "assoc.h"

class NLAM : public Assoc  {
protected:
        Matrix *newwts;                         // Matrix of input values
        int unit_type;                          // type of non-linear processing
public:
        void initwts(const Patterns&);
        NLAM(int sz, int sy, int type =0, Bool pol =True):
        Assoc(sz, sy, pol),
        newwts( new Matrix(sz, sy)),
        unit_type(type)
        {}

        NLAM(const Patterns&, int =0, Bool =True);
        ~NLAM() {
                delete newwts;
        }
        Vector recall(const Vector&) const;     // override  recall in Assoc class
        Matrix recall(const Matrix&) const;     // override  recall in Assoc class
        void print(ostream &s =cout) const;
        void scan(istream &s =cin);
};

/* these are the non-linear processing functions */
extern Vector Maxnet(const Vector&);
extern Vector Threshold(const Vector&);

Vector (*non-linear[])(const Vector&) = { Maxnet, Threshold };
#endif
```

```
/* The hamming net selects the winner from the stored patterns which has the        #
# least hamming distance from the input vector. In this implementation, no          #
# weight computation is actually done. The stored weights are actually the input    #
# patterns. The init functions have to be overridden to for the hamming network    */
#ifndef _HAM_H
#define _HAM_H
#include "assoc.h"
class Hamming : public Assoc {
protected:

public:
        Hamming(int sz, Bool bin =False) :          // hamming net constructor
        Assoc(sz, bin) {}                           // calls Assoc constructor
        Hamming(int sz, int sy, Bool bin =False) :
        Assoc(sz, sy, bin) {}
        Hamming(const Patterns &p, Bool bin =False) ;
        ~Hamming() {}
        void initwts(const Patterns&);              // override initwts in Base
        Vector recall(const Vector&) const;
        Matrix recall(const Matrix&) const;
};

#endif

/* Hopfield networks belong to a class of feedback associative networks. They       #
# require many iterations before the stored patterns can be retrieved. Three #
# basic types of Hopfield networks can be described, the sequential Hopfield net #
# the parallel Hopfield network and finally the discrete-time continuos state #
# Hopfield model. The main difference between the sequential and parallel #
# implementations is in the fact that the diagonal weight elements are zeroed       #
# out in the sequential case.  Author F. Che                                  */
#ifndef _HOPF_H
#define _HOPF_H
#include "vector.h"
#include "matrix.h"
#include "assoc.h"

class Hopfield: public Assoc {
protected:
public:
        Hopfield(int sz):                       // constructor
        Assoc(sz) {}
        Hopfield(const Patterns &p):
        Assoc(p) { initwts(p);  }
        void initwts(const Patterns&);          // matrix of input patterns
        ~Hopfield() { delete weights;           // destructor
                delete threshold;
        }

};

#endif
```