To the Graduate Council:

I am submitting herewith a dissertation written by Zhichao Guo entitled "*Nuclear Power Plant Fault Diagnostics and Thermal Performance Studies Using Neural Networks and Genetic Algorithms*". I have examined the final copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Nuclear Engineering.

Robert E. Uhrig, Major Professor

We have read this dissertation
and recommend its acceptance:

Accepted for the Council:

Associate Vice Chancellor
and Dean of The Graduate School

Nuclear Power Plant Fault Diagnostics and Thermal Performance Studies Using Neural Networks and Genetic Algorithms

A Dissertation

Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Zhichao Guo

May 1992

Dedicated to my wife and my parents

ACKNOWLEDGMENTS

The author wishes to acknowledge his academic advisor Dr. Robert E. Uhrig for his invaluable support, encouragement and technical advice throughout the entire period of this research.

The members of the doctoral committee, Dr. Belle R.Upadhyaya, Dr. Rafael B. Perez, Dr. Lefteri H. Tsoukalas, and Dr. Tadeusz Janik are sincerely acknowledged for their constructive criticism and technical advice.

The author is most grateful to the department head Dr. Thomas W. Kerlin for providing the necessary financial foundation and encouragement which have made the completion of this work possible.

The author would also like to acknowledge the financial support of the United States Department of Energy under contract $\#DE - FG07 - 88ER12824$ entitled "Enhancing Nuclear Power Plant Performance Through the Use of Artificial Intelligence." This sponsorship is greatly appreciated.

Acknowledgment is also given to Tennessee Valley Authority (TVA) who provided the data on Sequoyah and Watts Bar nuclear power plants.

Special honor, recognition and appreciation are given to my wife An Yan for her love and support.

ABSTRACT

A new neural network architecture called *Lateral Feedback Neural Network* is introduced in this dissertation. It is to modify the widely used *backpropagation* by introducing intra-layer connections to the hidden layer of the network. The intra-layer connections follow the competition mechanism of self-excitatory and neighborhood-inhibitory. The learning algorithms are developed by using Generalized Delta Rule to adapt all inter-layer and intra-layer connections and bias terms. The benchmark tests show that the *Lateral Feedback* network has advantages in learning speed, convergence, and stability over the original *backpropagation* for all tested cases. *Lateral feedback* network is applied to fault diagnoses for *TVA's Watts Bar* nuclear power plant together with the sensitivity analysis and genetic algorithms.

The sensitivity analysis is developed for both *backpropagation* and *lateral feedback* networks. It is the analog of the sensitivity analysis for ordinary functions. It treats a successfully trained network as a mapping function and develops the first derivatives of output variables with respect to input variables through the network learning algorithms and architectures. The information extracted from the sensitivity analysis can be used to rank input variables of a network into the order of importance. In the application to *TVA's Sequoyah* nuclear power plant thermal performance study, the sensitivity analysis provides the information on what may cause the deviation in the plant performance. In the application to fault diagnoses for *TVA's Watts Bar* nuclear power plant simulator data, the sensitivity analysis is used to guide the selection of optimal input variables for a set of small modular networks to monitor different accident scenarios.

Genetic algorithm is a special search algorithm based on the mechanisms of natural selection and natural genetics. It is a very efficient and robust search

methodology which can perform optimal search in a large and complex search space where the conventional algorithms, such as calculus-based search and hill-climbing method, can not perform well. It is applied in this dissertation to neural networks to guide the search for optimal selection of input variables to simplify the network system. The goal of the search is to find an optimal combination of input variables for a neural network to reach fast training, accurate recall, and fewest possible inputs. The application to fault diagnoses for *TVA's Watts Bar* nuclear power plant simulator data shows that genetic algorithms do find the solutions for each modular network which monitors one assigned accident scenario.

Contents

1 INTRODUCTION 1

1.1 Neural Networks 1
1.2 Genetic Algorithms 2
1.3 Nuclear Power Plant Applications With Neural Networks 3
1.4 Motivations and Contributions 4
1.4.1 Motivations 4
1.4.2 Contribution I: Sensitivity Analysis 4
1.4.3 Contribution II: Use of Genetic Algorithms to Guide the Selection of Neural Network Inputs 5
1.4.4 Contribution III: Lateral Feedback Neural Network 5
1.5 Organization of the Dissertation 6

2 THEORY 8

2.1 Lateral Feedback Network 8
2.1.1 Introduction 8
2.1.2 Description of Lateral Feedback Network 10
2.1.3 Learning Algorithm 11
2.1.4 Benchmark Tests 14
2.2 Sensitivity Analysis 16
2.2.1 Introduction 16
2.2.2 Sensitivity Analysis for a Backpropagation Network 17
2.2.3 Sensitivity Analysis for the Lateral Feedback Network 20

3 TVA NUCLEAR POWER PLANT THERMAL PERFORMANCE STUDY WITH NEURAL NETWORKS 22

3.1 Introduction 22
3.2 Hybrid Network Architecture 22
3.3 Neural Modeling 23
3.4 Sensitivity Analysis 25
3.5 Results From the Sensitivity Analysis 28
3.6 More Exploration 28
3.7 Discussion and Remarks 31

4 FAULT DIAGNOSTICS FOR NUCLEAR POWER PLANTS 32
4.1 Introduction 32
4.2 Simulation Data of Watts Bar Nuclear Power Plant 33
4.3 Review of the Previous Study 33
4.4 Use of Hybrid and Modular Neural Networks for the Study 35
4.4.1 Preparation of Data for Networks 37
4.4.2 Configuration of Motherhood Network 40
4.4.3 Training and Recall Results 40
4.4.4 Modular Networks 49
4.4.5 Modular Network Configuration, Training and Recall Data . . 51
4.4.6 Recall Results of Modular Networks 52
4.4.7 Noise Tolerance of the Modular Networks 60
4.4.8 Use of Weighted Variables 68
4.4.9 Extension of Modular Networks 77

5 SELECTION OF OPTIMAL INPUT VARIABLES FOR NEURAL NETWORKS BY USING GENETIC ALGORITHMS 81
5.1 Introduction 81
5.2 Genetic Algorithms 82
5.2.1 Introduction to Genetic Algorithms 82
5.2.2 Robustness of the Genetic Algorithms 83
5.3 Genetic Algorithm: Basic Theory and Operation 84
5.3.1 Driving Force of the Genetic Algorithms 84
5.3.2 Simple Genetic Algorithm 84
5.4 Applications of Genetic Algorithms To Neural Networks 86
5.5 Using Genetic Algorithms to Guide Selection Variables 87
5.6 Fault Diagnoses With Genetic Algorithms and Neural Networks . . . 88
5.6.1 Reasons for Selecting Important Variables 88
5.6.2 Sensitivity Analysis and Genetic Algorithm Search 90
5.6.3 Network Training Data 91
5.6.4 Link Between Genetic Algorithms and Neural Networks 92
5.7 Fitness Functions 92
5.7.1 Fitness Function 1 93
5.7.2 Fitness Function 2 94
5.7.3 Fitness Function 3 98
5.8 Best Selections for All Scenarios 104
5.9 Comparison of GA Search and Sensitivity Analysis 105
5.10 Conclusion 107

6 CONCLUDING REMARKS AND RECOMMENDATIONS FOR FUTURE RESEARCH 117
6.1 Concluding Remarks 117
6.1.1 Lateral Feedback Network 117

6.1.2 Sensitivity Analysis 118
6.1.3 Application of Genetic Algorithms 118
6.1.4 Diagnostics of Nuclear Power Plant Transients 119
6.2 Recommendations for Future Research 120

BIBLIOGRAPHY **122**

APPENDICES **129**

A GENETIC ALGORITHMS: THEORY AND OPERATION **130**
A.1 Introduction 130
A.2 Basic Theory and Operation 130
A.2.1 Driven Force of Genetic Algorithms 130
A.2.2 Simple Genetic Algorithm 130
A.2.3 Similarity Templates (Schemata) 132
A.2.4 Schema Theorem and Building Block Hypothesis 133

B ACCIDENT SCENARIO DESCRIPTIONS **135**

C PLANT PROCESS VARIABLES LIST **137**

D CODE FOR EXTRACT DATA FROM RAW DATA FILES **140**
D.1 mkdata.c 140
D.2 A Sample of Index File 144
D.3 A Sample Task File For mkdata.c 146

E CODE FOR MAKING NETWORK TRAINING DATA FILES **148**
E.1 nndata.c 148
E.2 A Sample of nndata.c Task File 151

F CODE FOR MODULAR NETWORK TRAINING **152**
F.1 Main Program of subnet.c 152
F.2 Functions Used in subnet.c 152
F.3 Train Function for subnet.c 153

G CODE FOR MODULAR NETWORK RECALL **162**
G.1 Main Program of subrecall.c 162
G.2 Functions Used by subrecall.c 166

H TASK FILES FOR RUNNING *ga_nn.c* **167**
H.1 Task File for Running Main Grogram of *ga_nn.c* 167
H.2 Task File for Genetic Algorithms' Operation 167
H.3 Task File for Running the Network 167

I COMPLETE C CODE OF GENETIC ALGORITHM SEARCH FOR OPTIMAL INPUT VARIABLES FOR NEURAL NETWORKS 169

I.1 Main Code of the Program: ga_nn.c 169
I.2 Codes Related to Genetic Algorithms 172
I.2.1 Global Definitions for Genetic Algorithms: globdef.sga 172
I.2.2 Initialization of the Genetic Algorithms: initial.sga 173
I.2.3 Random Number Generator and Random Operator: random.sga 176
I.2.4 Population Statistics: stats.sga 179
I.2.5 Population Propagation: generate.sga 180
I.2.6 Reproduction, Crossover, and Mutation Operations: tripos.sga 181
I.2.7 Identify Duplicated Individuals: comp_chrome.sga 184
I.2.8 Save and Retrieve Information Necessary for Continuing Search After Interruption: ga_io.sga 184
I.2.9 Interruption Control of the Process: ctrlc_handler 186
I.3 Codes Related to the Lateral Feedback Neural Network 186
I.3.1 Main Code of the Network: net_main.c 186
I.3.2 Header Files for the Network: nnet.h 188
I.3.3 Global Definitions of the Network: globdef.nn 188
I.3.4 Network Training Function: train.nn 191
I.3.5 Network Recall Function Used Within train.nn: recall.nn . . . 197
I.3.6 Openning a File and Return File Pointer: openfile.nn 199
I.3.7 Read Data From a Task File: get_par.nn 199
I.3.8 Read Training Data: getdata.nn 200
I.3.9 Network Setup: netsetup.nn 204
I.3.10 Randomize Network Weights Initially: random_weights.nn . . 205
I.3.11 Normalization Functions: normalizer.nn 207
I.3.12 Calculation of Activations of Hidden and Output Layer: layer-out.nn 211
I.3.13 Matrix Multiplication: matrix_sum.nn 213
I.3.14 Sigmoid Transfer Function: sigmoid.nn 214
I.3.15 Change Weights After Adjusted: chweights.nn 214
I.3.16 Save or Retrieve a Network: netio.nn 215
I.3.17 Add Random Noise to the Given Data: add_noise.nn 220
I.3.18 Report GA Search Results: report.nn 221
I.4 Fitness Function for GA Search: interface.sga 222

VITA 223

List of Figures

2.1 Recurrent and Neighborhood Intra-Layer Connections 11
2.2 Three Layer Network with Intra-Layer Connections in the Hidden Layer 12
2.3 Backpropagation Neural Network With One Hidden Layer 17
2.4 Backpropagation Neural Network With Two Hidden Layers 20

3.1 Architecture of Hybrid Network Formed by a *self-organization* and a *backpropagation* 23
3.2 Comparison of plant data with neural network outputs for Unit 1 . . 26
3.3 Comparison of plant data with neural network outputs for Unit 1 . . 26
3.4 Comparison of plant data with neural network outputs for Unit 2 by using neural network trained only on Unit 1 data 27
3.5 Comparison of plant data with neural network outputs for Unit 2 by using neural network trained only on Unit 1 data 27

4.1 Outputs From Node 1 of Motherhood Network Which Responses to Total Loss of Offsite Power 41
4.2 Outputs From Node 2 of Motherhood Network Which Responses to Main Steam Line Break 42
4.3 Outputs From Node 3 of Motherhood Network Which Responses to Main Feedwater Line Break 43
4.4 Outputs From Node 4 of Motherhood Network Which Responses to Control Rod Ejection 44
4.5 Outputs From Node 5 of Motherhood Network Which Responses to Hot-leg LOCA 45
4.6 Outputs From Node 6 of Motherhood Network Which Responses to Cold- leg LOCA 46
4.7 Outputs From Node 7 of Motherhood Network Which Responses to Steam Generator Tube Break 47
4.8 Response to All 7 Scenarios From Modular Network of Total Loss of Offsite Power (5 Variables Selected for the Network) 53
4.9 Response to All 7 Scenarios From Modular Network of Main Steam Line Break (9 Variables Selected for the Network) 54

4.10 Response to All 7 Scenarios From Modular Network of Main Feedwater Line Break (7 Variables Selected for the Network) 55
4.11 Response to All 7 Scenarios From Modular Network of Control Rod Ejection (8 Variables Selected for the Network) 56
4.12 Response to All 7 Scenarios From Modular Network of Hot-leg LOCA (7 Variables Selected for the Network) 57
4.13 Response to All 7 Scenarios From Modular Network of Cold-leg LOCA (7 Variables Selected for the Network) 58
4.14 Response to All 7 Scenarios From Modular Network of Steam Generator Tube Break (5 Variables Selected for the Network) 59
4.15 Response to All 7 Scenarios From Modular Network of Total Loss of Offsite Power (5 Variables Selected for the Network and 2% Noise Used for Recall) 61
4.16 Response to All 7 Scenarios From Modular Network of Main Steam Line Break (9 Variables Selected for the Network and 2% Noise Used for Recall) 62
4.17 Response to All 7 Scenarios From Modular Network of Main Feedwater Line Break (7 Variables Selected for the Network and 2% Noise Used for Recall) 63
4.18 Response to All 7 Scenarios From Modular Network of Control Rod Ejection (8 Variables Selected for the Network and 2% Noise Used for Recall) 64
4.19 Response to All 7 Scenarios From Modular Network of Hot-leg LOCA (7 Variables Selected for the Network and 2% Noise Used for Recall) 65
4.20 Response to All 7 Scenarios From Modular Network of Cold-leg LOCA (7 Variables Selected for the Network and 2% Noise Used for Recall) 66
4.21 Response to All 7 Scenarios From Modular Network of Steam Generator Tube Break (5 Variables Selected for the Network and 2% Noise Used for Recall) 67
4.22 Response to All 7 Scenarios From Modular Network of Total Loss of Offsite Power (5 Weighted Inputs Selected for the Network) 70
4.23 Response to All 7 Scenarios From Modular Network of Main Steam Line Break (9 Weighted Inputs Selected for the Network) 71
4.24 Response to All 7 Scenarios From Modular Network of Main Feedwater Line Break (7 Weighted Inputs Selected for the Network) 72
4.25 Response to All 7 Scenarios From Modular Network of Control Rod Ejection (8 Weighted Inputs Selected for the Network) 73
4.26 Response to All 7 Scenarios From Modular Network of Hot-leg LOCA (7 Weighted Inputs Selected for the Network) 74
4.27 Response to All 7 Scenarios From Modular Network of Cold-leg LOCA (7 Weighted Inputs Selected for the Network) 75
4.28 Response to All 7 Scenarios From Modular Network of Steam Generator Tube Break (5 Weighted Inputs Selected for the Network) . . 76

4.29 Response to All 7 Scenarios From Node 1 of Modular Network Trained for Both Hot-leg and Cold-leg LOCA (8 Input Variables Selected for the Network) 79
4.30 Response to All 7 Scenarios From Node 2 of Modular Network Trained for Both Hot-leg and Cold-leg LOCA (8 Input Variables Selected for the Network) 80

5.1 Fitness Versus Number of Inputs for Ten Generation Indices 100
5.2 Fitness Versus Training Error for Ten Generation Indices 101
5.3 Response to All 7 Scenarios From Modular Network of Total Loss of Offsite Power (7 Variables Selected by GA Search for the Network) . 108
5.4 Response to All 7 Scenarios From Modular Network of Main Steam Line Break (9 Variables Selected by GA Search for the Network) . . 109
5.5 Response to All 7 Scenarios From Modular Network of Main Feedwater Line Break (6 Variables Selected by GA Search for the Network) 110
5.6 Response to All 7 Scenarios From Modular Network of Control Rod Ejection (8 Variables Selected by GA Search for the Network) 111
5.7 Response to All 7 Scenarios From Modular Network of Hot-leg LOCA (7 Variables Selected by GA Search for the Network) 112
5.8 Response to All 7 Scenarios From Modular Network of Cold-leg LOCA (7 Variables Selected by GA Search for the Network) 113
5.9 Response to All 7 Scenarios From Modular Network of Steam Generator Tube Break (5 Variables Selected by GA Search for the Network) 114

List of Tables

2.1 Summary of XOR Training Results 15

3.1 Variables Selected for Neural Networks 24
3.2 Results From the Sensitivity Study 29
3.3 Most Important Variables Selected for *Gross Heat Rate* 29
3.4 Most Important Variables Selected for *Unexpected Power Deviation* . 30

4.1 Accident Scenarios Simulated on Watts Bar Simulator 33
4.2 Desired Network Output for the Transients 34
4.3 Time required to diagnose the transients 35
4.4 Variables Selected as Inputs for the Network 38
4.5 Binary Representation of 8 Accident Scenarios 39
4.6 Clusters Formed for Each Data File 39
4.7 Time to for the Network to Diagnose Transients 49
4.8 Results of Sensitivity Study for Node 1 50
4.9 Variables Selected for Each Modular Network 51
4.10 Time required for modular networks to diagnose the transients . . . 68

5.1 Network Performance For Diagnosing the *Cold Leg LOCA (TH2)* With Different Selection of Inputs but Using the Same Number of Training Cycles 89
5.2 Accident Scenarios Simulated on Watts Bar Simulator 91
5.3 Best Inputs Selected by GA, Fitness, and Net Error 93
5.4 Selections for RD6 by Genetic Algorithms 94
5.5 Term 1 of Fitness Value of Equation 5.5 Versus Number of Inputs . . 95
5.6 Term 2 of Fitness Value of Equation 5.5 Versus Training Error . . . 96
5.7 Best Selections for TH2 With Fitness Function 2 96
5.8 Initial Generation of GA Search for TH2 With Fitness Function 2 . . 97
5.9 Initial Generation of GA Search for TH2 With Fitness Function 3 . . 102
5.10 Statistics of GA Search for TH2 With Fitness Function 3 102
5.11 Best GA Selection for TH2 With Fitness Function 3 104
5.12 Best Selection of Inputs for All Seven Scenarios 105
5.13 Selection of Inputs From GA and SA 106
5.14 Comparison of Network Training Errors for All Seven Scenarios . . . 107

CHAPTER 1

INTRODUCTION

1.1 Neural Networks

The motivation for studying neural computation is inspired by the studies in neuroscience in many different fields, such as vision, memory, and thinking. In this respect, artificial neural network structures and learning algorithms are based on our present understanding of biological nervous systems. Designing artificial neural networks to solve problems and studying real biological neural networks may also change the way we think about problems and lead to new insights and algorithmic improvements. Arbib [1] and Amit [2] have provided detailed discussion to help understand the working principles of the human brain and neurophysiological background for building up artificial neural networks. There are many different types of artificial neural networks, such as supervised learning and unsupervised learning networks distinguished by their learning algorithms, and feedforward recall and feedback recall neural networks distinguished by their recall mechanisms. These aspects are addressed in detail in many text and reference books, such as the books by Kohonen [3], Cotterill [4], Anderson and Rosenfeld [5], Rumelhart, McClelland, and PDP Research Group [6], Nadel, Cooper, Culicover, and Harnish [7], Wasserman [8], Simpson [9], Khanna [10], Caudill [11], Maren, Harston, and Pap [12], and Hertz, Krogh, and Palmer [13].

In spite of differences in structures, learning algorithms, and recall mechanisms, neural networks are essentially information processing systems which are non-algorithmic, non-digital, and intensely parallel. A neural network consists of a number of very simple and highly interconnected processors called *neurodes* or *artificial neurons*, which are the analog of the biological neural cells in the brain.

The neurodes are connected by a large number of weighted links, over which signals can pass (Caudill [11]). Neural networks are specified by the arrangement of neurodes (usually in the form of layers), transfer function of neurodes, and the learning rules that determine how the adaptive coefficients are adjusted to match the neural network model with the actual system. Since the inputs and weighted connections change over time, the network adapts and learns. Neural networks have many remarkable features, such as the ability to process noisy, sparse, and incomplete data, the high fault tolerance which enables the networks to operate properly even with some damaged neurodes or weighted connections in the networks, and their ability to respond in real-time due to their inherent parallelism. With these desired features, neural networks have gained very rich applications in pattern recognition, image processing, speech recognition, and adaptive control.

1.2 Genetic Algorithms

Genetic algorithms are search algorithms based on the mechanics of natural selection and natural genetics. The principle behind genetic algorithms is essentially Darwinian natural selection. John Holland is the founder of the field of genetic algorithms. With the publication of *"Adaptation in Neural and Artificial Systems"* [14], he integrated and elaborated two themes that had persistently recurred in his research: the ability of simple representations (bit strings) to encode complicated structures , and the power of simple transformations to improve such structures [15]. The bit strings are specially structured to represent parameters or merits for an optimal search. There are three basic genetic algorithm operators, namely, *reproduction, crossover*, and *mutation.* These three operators form the main body of the genetic algorithms. They efficiently exploit historical information to speculate on new search points with expected improved performance. Genetic algorithms are theoretically and empirically proven to provide robust search in a complex search space. The

fundamental differences between the genetic algorithm and traditional optimization methods, such as calculus-based search and hill-climbing method, can be described in four ways [16]:

1. Genetic algorithms work with a coding of the parameter set, not the parameters themselves;

2. Genetic algorithms search from a population of points, not a single point;

3. Genetic algorithms use payoff (objective function) information, not derivatives or other auxiliary knowledge;

4. Genetic algorithms use probabilistic transition rules, not deterministic rules.

Taken together, these four differences contribute to a genetic algorithm's robustness and result in advantages over other more commonly used techniques. Having been established as a valid approach for solving problems requiring efficient and effective search, genetic algorithms are now finding more widespread applications in business, scientific, and engineering fields. Appendix A gives a more detailed discussion about genetic algorithms.

1.3 Nuclear Power Plant Applications With Neural Networks

With the features of fault tolerance, inherent parallelism, and noisy or incomplete data processing, neural networks are ideally suited as candidates for fault monitoring and diagnosis, control, and risk evaluation in nuclear power plant environments. The performance of a neural network depends on its structure, learning rules, and recall mechanism, also on the input variables selected and the training pattern set. There are thousands of parameters that are monitored in a nuclear power plant, but only a limited number of them can be used as the inputs for the neural network, because it is very difficult to train a neural network if too many

input variables are involved. Therefore, proper selection of input variables is the key to the success of using neural networks for nuclear power plant applications.

1.4 Motivations and Contributions

1.4.1 Motivations

For a given complex system, neural networks can be trained to behave as an internal model of the system if the system input and output data are available. The selection of inputs for the networks is crucially important to the success of training and performance of the networks. If a network has been trained successfully, can it be studied further to explore more information about the system other than recall of stored patterns? How do we select proper inputs to reach the optimal performance for a given tasks? How can we improve the convergence and stability of the network learning? These questions are the motivations for the original contributions in sensitivity analysis, application of genetic algorithms to neural networks, and lateral feedback neural network.

1.4.2 Contribution I: Sensitivity Analysis

A successfully trained neural network with supervised learning essentially maps the input vectors in one space to the corresponding output vectors in another space. Therefore, it is possible to study a trained network as a mapping function, and to establish the relationship between output variables and input variables. This is the basic idea of sensitivity analysis as applied to neural networks. Sensitivity analysis develops the first derivatives of variables in the output layer with respect to variables in the input layer. These values of the first derivatives can be used to rank input variables in order of importance with respect to the corresponding output variables. The information extracted from the sensitivity analysis can be used for

system diagnosis and control. It is applied to the *TVA's Sequoyah* nuclear power plant thermal performance analysis in Chapter 3. The information can also be used as a guidance for the selection of input variables to build a set of small modular networks to monitor the accident scenarios as is demonstrated in Chapter 4 for fault diagnoses applied to *TVA's Watts Bar* nuclear power plant. Sensitivity analysis has been established for both the standard *backpropagation* network and the *lateral feedback* network in Chapter 2.

1.4.3 Contribution II: Use of Genetic Algorithms to Guide the Selection of Neural Network Inputs

Genetic algorithms have many advantages over the conventional optimization methods in optimal search. The robustness of the genetic algorithms makes it possible to search for optimal combinations of variables for neural networks to satisfy the criterion of simple network structure, fast training, and accurate recall. The genetic algorithms have been developed by John Holland [14] and studied further by Goldberg [16]. The main problem for different applications of genetic algorithms is to set up the proper bit string representations and create the proper fitness functions. The contribution of this dissertation is to apply genetic algorithms to neural networks to guide the selection of optimal inputs, and to establish the proper form of fitness function for the applications.

The application of the genetic algorithms to the *lateral feedback* neural network for fault diagnostics of *TVA's Watts Bar* nuclear power plant is given in Chapter 5.

1.4.4 Contribution III: Lateral Feedback Neural Network

The backpropagation neural network is probably the best known and widely-used option among the current types of neural network systems available. This

network is an outgrowth of perceptrons, with addition of *hidden layers* and use of *Generalized Delta Rule* for learning. The principal disadvantage of the backpropagation is the speed at which it converges or learns. The elementary backpropagation topology has been extended in many ways to make this algorithm converge quicker. The *lateral feedback* network introduced in this dissertation is a modified version of the *backpropagation* network. It introduces intra-layer connections in the hidden units. The intra-layer connections involve only the nearest neighborhood and self feedback. The intra-layer connections are self-excitatory and neighborhood-inhibitory. The benchmark tests show that *lateral feedback* network converges faster and is more stable than the standard *backpropagation* network in all tested cases. The learning algorithms of *lateral feedback* network are developed in Chapter 2 and applications are found in Chapters 4 and 5.

1.5 Organization of the Dissertation

The introduction of the concepts of neural networks and genetic algorithms are given in Chapter 1. The motivations of using genetic algorithms with neural networks, and developing the sensitivity analysis and lateral feedback network are also discussed in Chapter 1.

The learning algorithms of the lateral feedback neural network and the development of sensitivity analysis for both standard *backpropagation* and *lateral feedback* networks are given in Chapter 2. The basic theory of genetic algorithms is given in Appendix A.

The application of the sensitivity analysis to the thermal performance study of *TVA's Syquoyah* nuclear power plant is given in Chapter 3. The sensitivity analysis determines the important variables which strongly affect the power plant performance, and therefore, need to be monitored and controlled in order to operate the plant efficiently.

Chapter 4 gives the applications of the lateral feedback neural network and its sensitivity analysis to guide the building up of the modular networks, each one of which monitors one accident scenario.

The genetic algorithm and its application to *lateral feedback* network for search of optimal inputs for modular networks to monitor different accident scenarios are discussed in Chapter 5. Comparisons between the applications of genetic algorithms and sensitivity analysis in the selection of inputs are also provided in Chapter 5.

Concluding remarks and recommendation for future research are given in Chapter 6.

The computer program codes, task files, and the information about the accident scenarios are given in the Appendices.

CHAPTER 2

THEORY

2.1 Lateral Feedback Network

2.1.1 Introduction

The backpropagation neural network is probably the best known and widely-used option among the current types of neural network systems available. This network is an outgrowth of perceptrons, with addition of *hidden layers* and use of *Generalized Delta Rule* for learning. The principal disadvantage of the back-propagation is the speed at which it converges or learns. It is not guaranteed to find the global error minimum through training, but only the local minimum which may or may not be the global minimum. This situation sometimes leads to severe oscillations in weight adjustments during training(Simpson [9]). There are a lot of theoretical and empirical studies about its convergence, the optimization of the number of its hidden units, the improvement of its learning rate, new network connection topology, higher-order correlation, and arbitrary threshold functions. The elementary backpropagation topology has been extended in many ways to make this algorithm converge quicker. Werbos [17] introduced the dynamic feedback (back-propagation), which allowed any parameter to be adjusted by any previous set of parameters. The most general backpropagation topology possible is one that allows any neuron to connect to any other neuron. These topologies introduced into the recall dynamics are suitably called recurrent backpropagation neural networks. In some configurations, recurrent networks recirculate previous outputs back to inputs; hence, their output is determined both by their current inputs and their previous outputs. For this reason recurrent networks exhibit properties very similar to short-term memory in humans in that the state of the network outputs depends in part

upon previous inputs. These paradigms have been studied by Almeida [18], Rohwer & Forrest [19], Samad & Harper [20], Shimohara, Uchiyama & Tokunuga [21], Atiya [22], Pineda [23].

Neural networks with lateral feedback can be considered as another form of recurrent network, in which a neuron's activation depends not only on input signals from the immediate lower layer, but also on signals from the neurons in the same layer. It has been shown that the intra-layer or lateral connections can provide contrast enhancement of signals during encoding and recall (Cohen & Grossberg [24]). It may also increase the stability and convergence of the system if we consider it as a feedback system. The idea of introducing lateral connection within a layer comes intuitively from the study of human brains. If we consider a biological neural system, there are billions of neurons interconnected to form layers. Neurons within a layer should also be connected in some ways to communicate with each other within the layer. There are a number of neural network paradigms in which the lateral or intra-layer connections are utilized. The Additive Grossberg (AG) and Shunting Grossberg (SG) use the positive valued recurrent connections (self-feedback) and negative valued lateral connections to perform unsupervised learning (Grossberg [25], [26]), in which the recurrent and lateral connections are adapted by Hebbian learning laws. The idea of lateral connections is used to perform competition among neurons within a layer by the principle of self-exciting and neighbor-inhibiting. The exciting and inhibiting connections are usually preset and not adapted during learning, for example, the intra-layer connections in *ART1*, *ART2* (Carpenter & Grossberg, [27], [28], [29]), *Kohonen* (Kohonen [3]), and *Counterpropagation* (Hecht-Nielsen [30]), perform winner-take-all competitions. Kosko [31] introduced the lateral connections to $ABAM$ (Adaptive Bidirectional Associative Memory) to form competitive $ABAM$ or $CABAM$. The intra-layer connections are preset with non-adaptive self-exciting and neighbor-inhibiting connection values. Moody [32] studied the dynamic behavior of

lateral interaction networks, including additive linear network, the additive sigmoid, and non-linear shunting network coupled with Hebbian learning.

The lateral feedback neural network contemplated in this dissertation introduces intra-layer connections to the hidden layer(s) of the backpropagation network. The intra-layer connections can be modified by applying the generalized delta rule.

2.1.2 Description of Lateral Feedback Network

The network uses the generalized delta rule with the error feedback algorithm. The recurrent connections (self-feedback) and lateral feedback connections are introduced only to the hidden layer(s). The lateral connections are established only in the nearest neighbors of a neuron. There will be maximum of eight neighbors for a neuron in a 2-dimensional layer.

The recurrent signals are positive (keep the sign of the original signal) to perform self-exciting, and the lateral feedback signals are negative (invert the sign of the original signal) to exercise the neighbor-inhibiting. Therefore, a neuron in the layer receives signals from the neurons in the immediate lower layer through inter-layer connections, from itself through recurrent connection, and from its nearest neighbors through the lateral connections. All interconnections including the intra-layer connections are adapted by the generalized delta rules. Figure 2.1 shows lateral connections for neurons in a 2-D layer.

The feedback signals from and to a neuron flow through different paths with separate weights, such as the connections from A to B and from B to A are two different ways like a two way traffic lane. A neuron in the edge area has fewer lateral links than those in the center area. The maximum number of lateral connections including self feedback is 9 if a neuron is completely surrounded by other neurons. All neurons in the layer connect together by these lateral connections and information will be processed within a layer, and, of course, from layer to layer.

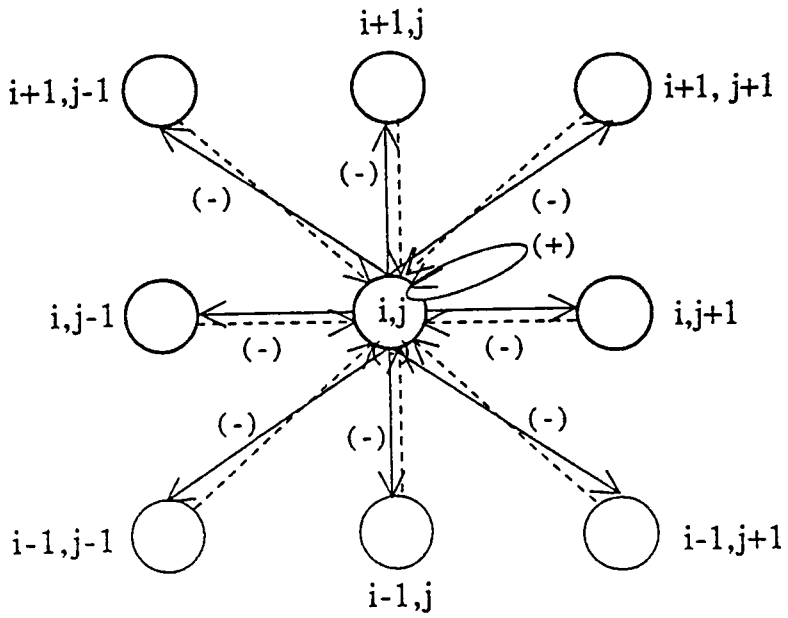

Figure 2.1: Recurrent and Neighborhood Intra-Layer Connections

2.1.3 Learning Algorithm

The generalized delta rule can be extended to include intra-layer connections. Without loss of generality, a three layer network architecture with intra-layer connections introduced only to the hidden layer is discussed below.

Define the overall error in the output layer as:

$$E = \frac{1}{2}\sum_{k}(d_k - c_k)^2 \tag{2.1}$$

where d_k and c_k are the desired and actual outputs of kth neuron in the output layer, respectively, and the summation involves all neurons in the output layer. The inter-layer connection from the ith neuron in the input layer to the jth neuron in the hidden layer is denoted as w_{ij}, and the inter-layer connection from the jth neuron in the hidden layer to the kth neuron in the output layer is denoted as v_{jk}. The

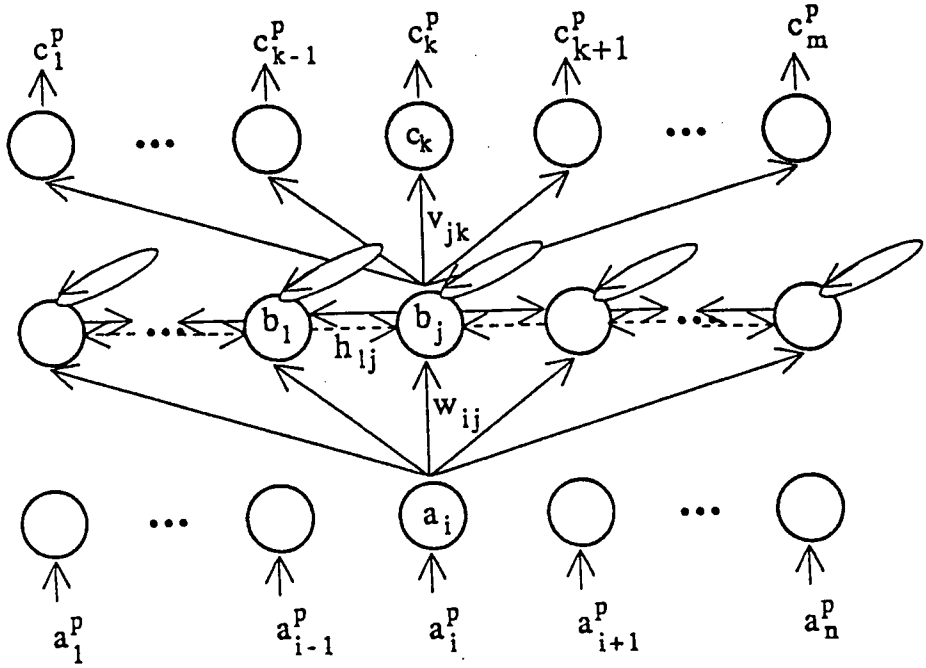

Figure 2.2: Three Layer Network with Intra-Layer Connections in the Hidden Layer

lateral connection from the lth neuron in the hidden layer to the jth neuron in the same layer is denoted by h_{lj}. Fig. 2.2 shows the connections of the network.

As an input pattern is presented to the input layer, it is processed forward to the hidden layer through inter-layer connections to create a temporal image in the hidden layer. Then, this temporal image is used to generate feedback signals through the lateral connections. Therefore, a neuron in the hidden layer receives signals from the input layer through inter-layer connections, from itself through the recurrent connection, and from its immediate neighbors through the lateral connections. All these signals are processed by the hidden units to produce the output signals, and then, feed the signals forward to the output layer. These processes are explained mathematically below:

$$b_j^* = f(\sum_i w_{ij} a_i - \theta_j) \tag{2.2}$$

$$b_j = f(\sum_i w_{ij} a_i - \sum_{l \neq j} h_{lj} b_j^* + h_{jj} b_j^* - \theta_j) \tag{2.3}$$

$$c_k = f(\sum_j v_{jk} b_j - \beta_k) \tag{2.4}$$

where b_j^* is the intermediate activation of the jth neuron in the hidden layer, b_j is the final activation of the jth neuron in the hidden layer after the intra-layer feedback, c_k is the activation of the kth neuron in the output layer, and θ and β are bias terms of neurons in the hidden layer and output layer, respectively.

The learning algorithms for the *lateral feedback* network can be developed through the chain rule as:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial c_k} \cdot \frac{\partial c_k}{\partial b_j} \cdot \frac{\partial b_j}{\partial w_{ij}} \tag{2.5}$$

$$\frac{\partial E}{\partial h_{lj}} = \frac{\partial E}{\partial c_k} \cdot \frac{\partial c_k}{\partial b_j} \cdot \frac{\partial b_j}{\partial h_{lj}} \tag{2.6}$$

$$\frac{\partial E}{\partial v_{jk}} = \frac{\partial E}{\partial c_k} \cdot \frac{\partial c_k}{\partial v_{jk}} \tag{2.7}$$

$$\frac{\partial E}{\partial \theta_j} = \frac{\partial E}{\partial c_k} \cdot \frac{\partial c_k}{\partial b_j} \cdot \frac{\partial b_j}{\partial \theta_j} \tag{2.8}$$

$$\frac{\partial E}{\partial \beta_k} = \frac{\partial E}{\partial c_k} \cdot \frac{\partial c_k}{\partial \beta_k}. \tag{2.9}$$

Note that $\partial b_j^*/\partial w_{ij}$ and $\partial b_j^*/\partial \theta_j$ are included implicitly in $\partial b_j/\partial w_{ij}$ and $\partial b_j/\partial \theta_j$ where b_j^* is the intermediate activation of the jth neuron in the hidden layer given by Equation 2.2. Hence, the learning algorithms can be obtained as:

$$w_{ij}(new) = w_{ij}(old) + \alpha_w b_j(1 - b_j)a_i[1 + h_{jj}b_j^*(1 - b_j^*)]e_j \tag{2.10}$$

$$h_{lj}(new) = h_{lj}(old) - \alpha_h b_j(1 - b_j)b_l^* e_j \quad \forall \quad l \neq j \tag{2.11}$$

$$h_{jj}(new) = h_{jj}(old) + \alpha_h b_j(1 - b_j)b_j^* e_j \tag{2.12}$$

$$v_{jk}(new) = v_{jk}(old) + \alpha_v(d_k - c_k)c_k(1 - c_k)b_j \tag{2.13}$$

$$\theta_j(new) = \theta_j(old) - \alpha_\theta b_j(1 - b_j)[1 + h_{jj}b_j^*(1 - b_j^*)]e_j \tag{2.14}$$

$$\beta_k(new) = \beta_k(old) - \alpha_\beta(d_k - c_k)c_k(1 - c_k) \tag{2.15}$$

where

$$e_j = \sum_k (d_k - c_k)c_k(1 - c_k)v_{jk} \tag{2.16}$$

and $\alpha_w, \alpha_h, \alpha_v, \alpha_\theta$, and α_β are the learning rates for the corresponding adjustments. Subscript l is the notation for all nearest neighbors of a neuron in the hidden layer.

With the learning algorithms, from Equation 2.10 to Equation 2.15, all the interconnections, including inter-layer connections, intra-layer connections, as well as bias terms, are adapted during the learning process.

2.1.4 Benchmark Tests

The network has been programmed in *C* and tested with several applications. One is XOR problem, which is a typical example often used to test neural networks. The hidden layer needs at least two neurons to map XOR problem. The

Table 2.1: Summary of XOR Training Results

Hidden Units	Iterations	Convergence	Net Type
2	10,1000	NO	BP
2	3,000	YES	MBP
4	2,500	YES	BP
4	1,500	YES	MBP
9	8,000	NO	BP
9	1,500	YES	MBP

training results show that for a standard backpropagation network (denoted as *BP*) the convergence of XOR depends heavily on the initialization of the connections. It may not converge if only 2 hidden neurons are used. On the other hand, the lateral feedback network (denoted as *MBP* for modified version of backpropagation) has converged in all tested cases. Training results for two or more hidden neurons are listed in Table 2.1. The lateral feedback network was very stable and did not show any evidence of over training for the training cycles over 10,000.

Another application was to use *TVA Sequoyah* nuclear plant thermal performance data. The task was to train the network to predict heat rate of the plant. There were 12 neurons in the input layer to receive the data from 12 selected plant process parameters, 16 in the hidden layer, and 1 in the output whose value represents the heat rate predicted by the network. The number of training patterns was 57. For the backpropagation network there was no evidence of convergence after 6,000 training cycles. For the lateral feedback network, the training began to converge after 5,000 iterations with maximum prediction error less than 5%. To test the stability of the lateral feedback network, the training was continued up to 20,000 cycles. The network continued to converge with maximum prediction error less than 1.0%.

These applications show that the lateral feedback network has advantages in training speed and stability over the conventional backpropagation network.

2.2 Sensitivity Analysis

2.2.1 Introduction

For all kinds of applications, the training process of a neural network is terminated as soon as the convergent criteria are reached. Then, the trained network is used to retrieve the stored information through the recall process. Considerable research on artificial neural networks is focused on developing new learning algorithms, exploring new network architectures, and extending new fields for network applications. Sensitivity analysis is introduced in this dissertation to study a trained network itself and obtain useful information other than what can be obtained through conventional recall.

For a continuous function, its derivatives provide very useful information in characterizing the function itself. The first derivative gives the tangential information of the function, and the second derivative gives the curvature information of the function. Sensitivity is usually provided by the first derivative of a function, because it is the ratio of a change in the function's value caused by a change in an independent variable. The larger the absolute value of the first derivative is , the more sensitive the function is with respect to the variable. The sign of the first derivative indicates whether these changes are in the same or opposite directions. The sensitivity analysis of a trained network is the analog of that for an ordinary function.

A successfully trained neural network maps an input vector $\vec{X}$ from a n-dimensional space to an output vector $\vec{Y}$ in a m-dimensional space. It can be expressed as:

$$\vec{Y} = f(\vec{X}) \tag{2.17}$$

where $\vec{Y} = (y_1, y_2, \ldots, y_m)$, and $\vec{X} = (x_1, x_2, \ldots, x_n)$.

The first derivative, $\partial y_j/\partial x_i$, measures the change in y_j while x_i is changing. Hence, it can provide information about how sensitive the output y_j is with respect to the input x_i. In other words, the larger the absolute value of $\partial y_j/\partial x_i$ is, the more important the x_i is with relation to y_j. The derivative, $\partial y_j/\partial x_i$, can be found through the structure of the trained neural network.

2.2.2 Sensitivity Analysis for a Backpropagation Network

Without loss of generality, we consider a standard backpropagation network with only one hidden layer as shown in Figure 2.3 where the weight w_{ij} is the connection from the ith neuron in the input layer to the jth neuron in the hidden layer, and v_{jk} is the connection from the jth neuron in the hidden layer to the kth neuron in the output layer.

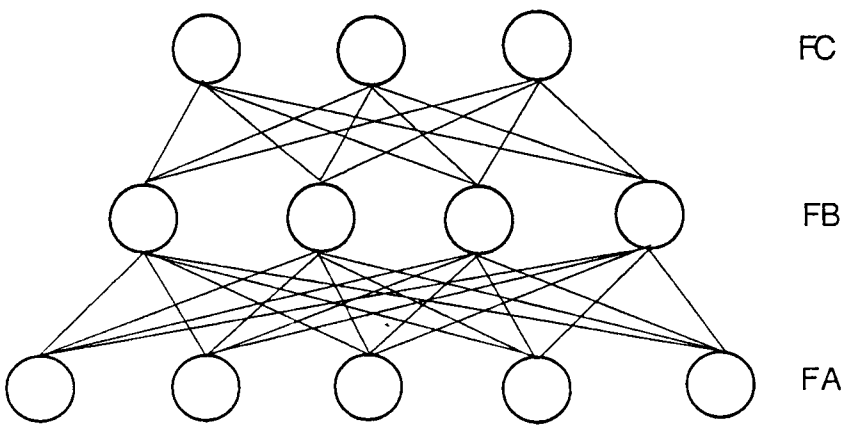

Figure 2.3: Backpropagation Neural Network With One Hidden Layer

Following its flow path we can see that when an input vector $\vec{X}$ is sent to the input layer, it is filtered forward to the hidden layer through the connections between the input layer and the hidden layer to form an output vector $\vec{B} = (b_1, b_2, \cdots, b_p)$. Then, $\vec{B}$ is filtered to the output layer through the connections between the hidden layer and the output layer to form an output vector $\vec{Y}$. This mapping process can be explained mathematically as follows:

$$b_j = f(\sum_i w_{ij} x_i + \theta_j) \tag{2.18}$$

$$y_k = f(\sum_j v_{jk} b_j + \eta_k) \tag{2.19}$$

where b_j is the output from jth neuron in the hidden layer, θ_j is the threshold value of the jth neuron in the hidden layer, η_k is the threshold value of kth neuron in the output layer, and f$(\cdot)$ is the sigmoid transfer (logistic) function:

$$f(z) = \frac{1}{1 + exp(-\beta z)} \tag{2.20}$$

where β is called the sigmoid slope.

The sigmoid function has the important property that its first derivative can be expressed in terms of the function itself, having the equation:

$$\frac{df(z)}{dz} = \beta f(z)(1 - f(z)) \tag{2.21}$$

To get $\partial y_k / \partial x_i$, simply applying the chain rule:

$$\frac{\partial y_k}{\partial x_i} = \frac{\partial y_k}{\partial b_j} \cdot \frac{\partial b_j}{\partial x_i} = \beta y_k (1 - y_k) \sum_j v_{jk} \frac{\partial b_j}{\partial x_i} \tag{2.22}$$

$$\frac{\partial b_j}{\partial x_i} = \beta b_j (1 - b_j) w_{ij} \tag{2.23}$$

which gives:

$$\frac{\partial y_k}{\partial x_i} = \beta^2 y_k(1-y_k)\sum_j v_{jk} b_j(1-b_j) w_{ij} \tag{2.24}$$

The equation 2.24 shows that the partial derivative $\partial y_k/\partial x_i$ depends not only on the information learned by the network, which is stored distributively in the connections of w_{ij} and v_{jk}, but also on the activation of neurons in both the hidden layer and the output layer, which, in turn, depends on the inputs of the network. For a trained network, all interconnections, w_{ij} and v_{jk}, are fixed, and therefore, the derivative is a function of input patterns only. If the absolute value of $\partial y_k/\partial x_i$ is averaged over all input patterns, it will give the measure of the sensitivity of the output variable y_k with respect to the input variable x_i in a global or average sense. Therefore, all input variables can be ranked in the order of sensitivity (or importance) according the values of the first derivative. It can also be studied pattern by pattern to analyze the dynamic behavior of the derivatives.

The information extracted from the sensitivity analysis can also be used to determine whether a variable selected as the network input is necessary to the network performance. Therefore, the lower ranked or less important variables can be taken away from the inputs to simplify the neural network system.

Sensitivity analysis is used in Chapter 3 for the application of *TVA* nuclear power plant thermal performance study . It is also used in Chapter 4 to guide the building up of the modular networks.

For a backpropagation with more than one hidden layers, the first derivative for the sensitivity can be developed following the same procedures as discussed before. If the network considered has two hidden layers as shown in Figure 2.4, for example, the corresponding first derivative can be obtained as:

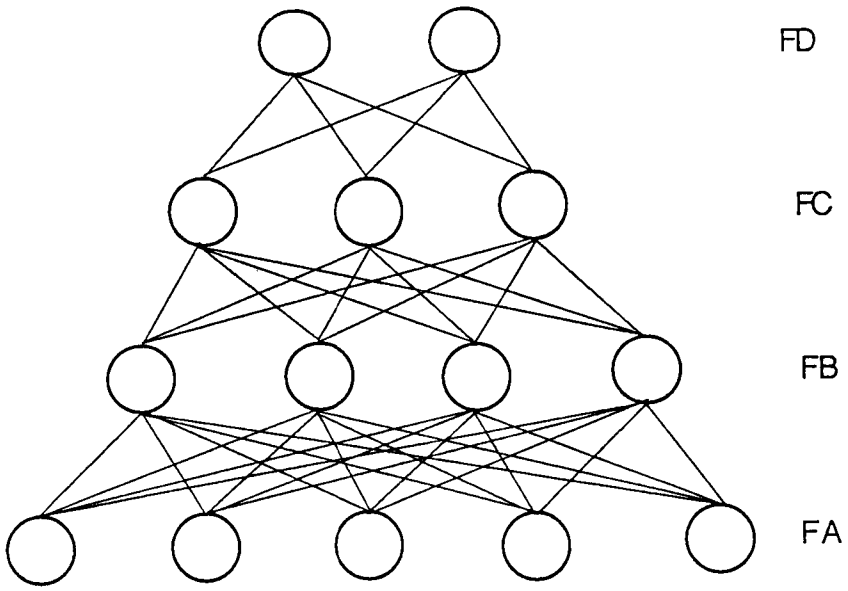

Figure 2.4: Backpropagation Neural Network With Two Hidden Layers

$$\frac{\partial y_l}{\partial x_i} = \beta^3 y_l(1-y_l)\sum_k c_k(1-c_k)u_{lk}\sum_j v_{jk}b_j(1-b_j)w_{ij} \tag{2.25}$$

where u_{lk} is the connection from the kth neuron in the second hidden layer (the third layer in the network) to the lth neuron in the output layer, and c_k is the activation of the kth neuron in the second hidden layer. The rest of the notations are the same as those shown in Figure 2.3.

2.2.3 Sensitivity Analysis for the Lateral Feedback Network

Figure 2.2 gives the configuration of the lateral feedback network schematically, and the mapping process have been explained in Section 2.1. For convenient reference, the equations for the intermediate activation of the jth hidden unit, b_j^*, the final activation of the jth hidden unit, b_j, and the activation of the kth output

unit, y_k, are listed below:

$$b_j^* = f(\sum_i w_{ij}x_i - \theta_j) \tag{2.26}$$

$$b_j = f(\sum_i w_{ij}x_i - \sum_{l \neq j} h_{lj}b_j^* + h_{jj}b_j^* - \theta_j) \tag{2.27}$$

$$y_k = f(\sum_j v_{jk}b_j - \eta_k) \tag{2.28}$$

where θ and η are bias terms of neurons in the hidden layer and output layer, respectively.

Again, using the sigmoid transfer function with the slope β, we have:

$$\frac{df(z)}{dz} = \beta f(z)(1 - f(z)) \tag{2.29}$$

To get $\partial y_k / \partial x_i$, simply applying the chain rule:

$$\frac{\partial y_k}{\partial x_i} = \frac{\partial y_k}{\partial b_j} \cdot \frac{\partial b_j}{\partial b_j^*} \cdot \frac{\partial b_j^*}{\partial x_i}$$

We have:

$$\frac{\partial y_k}{\partial x_i} = \beta^3 y_k(1-y_k) \sum_j v_{kj}b_j(1-b_j)[\frac{w_{ji}}{\beta} - \sum_{l,l \neq j} h_{jl}b_l^*(1-b_l^*)w_{li} + h_{jj}b_j^*(1-b_j^*)w_{ji}] \tag{2.30}$$

where l is the label for all nearest neighbors of the jth hidden unit.

Equation 2.30 is used in Chapter 4 to guide the selections of input variables to the modular networks, each one of which monitors one assigned accident scenario.

CHAPTER 3

TVA NUCLEAR POWER PLANT THERMAL PERFORMANCE STUDY WITH NEURAL NETWORKS

3.1 Introduction

Heat rate is a very important measure of the performance of nuclear power plants. The thermal performance data taken weekly from the *TVA's Sequoyah* nuclear power plant Unit 1 and Unit 2 include about 40 measured or calculated variables. Heat rate is one of the calculated variables. The recorded data shows that the heat rate is changing constantly and the plant is probably losing some megawatts of electric power due to heat rate variation. It is very difficult to find the causes of the heat rate deviation through traditional analytical methods because a nuclear power plant is a very complex system with thousands of variables.

In the view of the successful applications in other complicated systems, artificial neural networks are used to study heat rate of the *Sequoyah* nuclear power plant.

3.2 Hybrid Network Architecture

In order to use neural networks to study the heat rate, a proper network type has to be chosen and the network structure has to be set up. The network used in the study is a hybrid neural network which is a combination of *self-organization* and *backpropagation* neural networks. The *self-organization* network works as an organizer which rearranges the original training patterns into different classes represented by clusters. Then, the centroids of these clusters are used as the training input patterns for the *backpropagation* network. Figure 3.1 shows the hybrid network architecture schematically. The training time used by the *backpropagation*

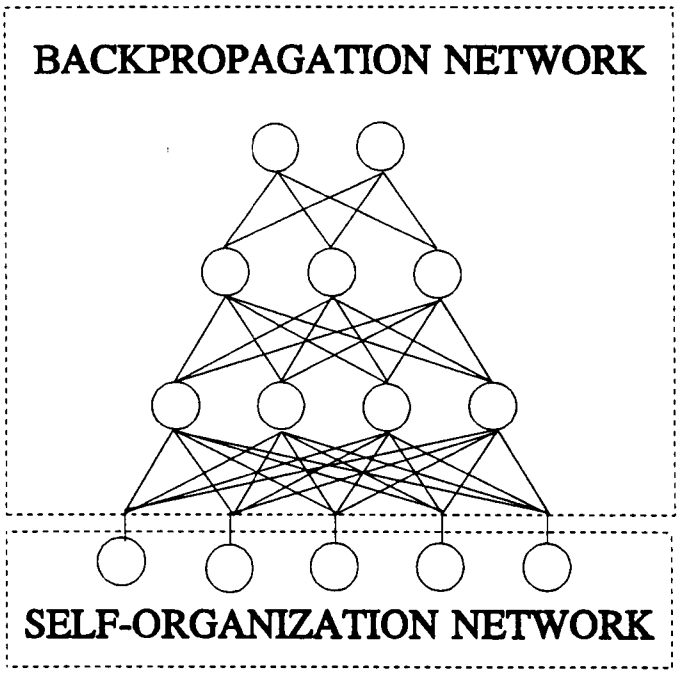

Figure 3.1: Architecture of Hybrid Network Formed by a *self-organization* and a *backpropagation*

network can be greatly reduced through this combination because the number of clusters is usually much smaller than the number of original training patterns. The application with the data from the *TVA's Sequoyah* nuclear power plant shows that the training speed, convergence, and the prediction accuracy are improved by using this special network combination.

3.3 Neural Modeling

In order to find the important variables which might strongly affect the heat rate, 24 variables were selected as the inputs which change with time, and 2 variables were chosen as the desired outputs. The other variables are almost constant, which reflect the plant design. Table 3.1 gives the list of the variables where variable 25 and 26 are desired outputs.

Table 3.1: Variables Selected for Neural Networks

	Selected Inputs
1	Reactor Power during Test (%)
2	Generator Output AT Reactor Power during Test (MWe)
3	Condenser Performance Change Based on River Temp.(MWe)
4	Other Condenser Losses (MWe) (Loss +)
5	Throttle Pressure Greater Than Design
6	Improved MSR Performance (MWe)
7	Auxiliary Steam Loads (MWe) (Loss +)
8	Other Misc. Loads (MWe) (Loss +)
9	Measured Generator Output during Test (MWe)
10	Total Unaccounted Megawatts (MWe) (Loss +)
11	Primary to Secondary Calorimetric Ratio
12	Measured Condenser Backpressure (inches HG abs.)
13	Expected Condenser Backpressure (inches HG abs.)
14	Condenser Circulating Water Inlet Temp. (deg F)
15	Final Feed Water Temperature (deg F)
16	Impulse Pressure [PT-1-73] (psig)
17	Impulse Pressure [PT-1-72] (psig)
18	Feedwater Flow (10^6 lbm/hr)
19	Avg Rx Delta Temp. (deg F)
20	Turbine Power Corrected to Design (MWe)
21	Unexpected Power Deviations (MWe) (Loss +)
22	Thermal Power Normalized RX Delta T (%)
23	Impulse Pressure Avg (psia)
24	Impulse Pressure Normalized (Frac.)
	Desired Outputs
25	Thermal Performance (INPO) (%)
26	Gross Heat Rate (Btu/Kwh)

Thermal Performance (INPO) in Table 3.1 is not thermal efficiency. It is the ratio of a reference heat rate to the measured gross heat rate corrected for river temperature, multiplied by 100 to express it in precent.

The training data was prepared from the data of the *Sequoyah* Unit 1 with 45 patterns in total. The training patterns for the *backpropagation* were reduced to 22 by using the centroids of the clusters formed by the training of the

self-organization neural network. The *backpropagation* network was set up with 24 neurons in the input layer, 10 in hidden layer, and 2 in the output layer. The network was trained successfully with system error of 0.0001, which is the average of pattern errors of all patterns. The recall was performed with the original 45 patterns from Unit 1, which were not used directly for training of the *backpropagation* network but for training of the *self-organization* network. The average error of the network prediction was about 0.06% and the largest error was 0.34%. The recall was also performed with 36 patterns from the Unit 2, a similar, but not identical plant. The data from Unit 2 were used neither for training of the *self-organization* nor for training of the *backpropagation* network. The average error of the network prediction was less than 0.32% and the largest error was 1.3%. The small prediction error for Unit 2 indicates that the thermodynamic behaviors of these two plants are similar. Figure 3.2 and 3.3 show the comparisons of the plant data with the outputs of the network for Unit 1. Figure 3.4 and 3.5 show the comparisons of the plant data from Unit 2 with the outputs of the neural network trained only by using data from Unit 1.

3.4 Sensitivity Analysis

The neural modeling network used in this application is a three layer standard *backpropagation* network, therefore, the sensitivity analysis established in Chapter 2 can be used directly to the neural modeling for further analysis to find the important input variables with respect to one or more output variables. Equation 2.24 is developed for the sensitivity analysis of a *backpropagation* with only one hidden layer. For convenience, Equation 2.24 is given below as the reference:

$$\frac{\partial y_k}{\partial x_i} = \beta^2 y_k(1 - yk) \sum_j v_{jk} b_j (1 - b_j) w_{ij} \tag{3.1}$$

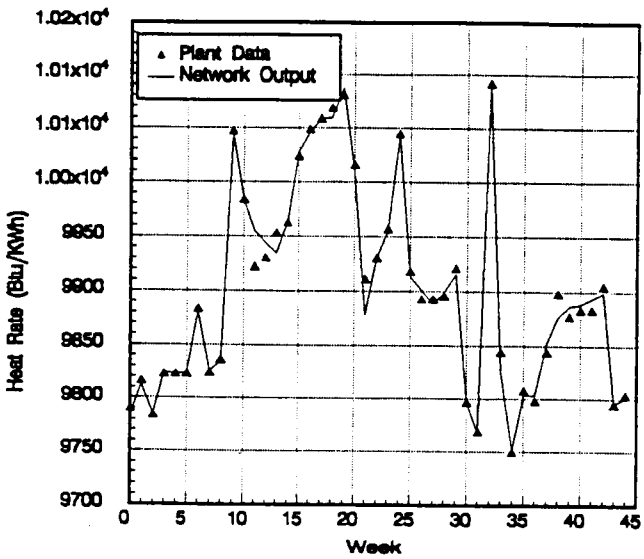

Figure 3.2: Comparison of plant data with neural network outputs for Unit 1

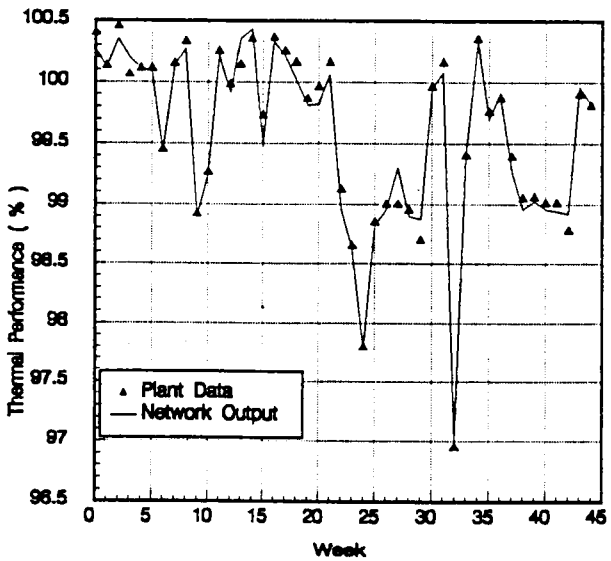

Figure 3.3: Comparison of plant data with neural network outputs for Unit 1

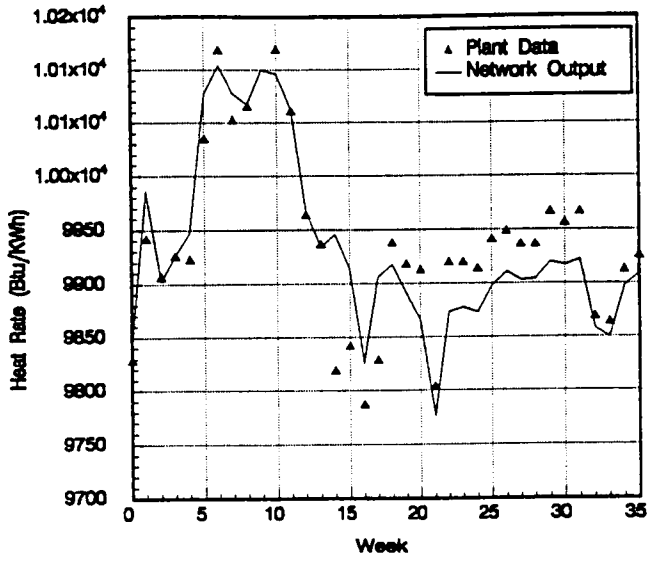

Figure 3.4: Comparison of plant data with neural network outputs for Unit 2 by using neural network trained only on Unit 1 data

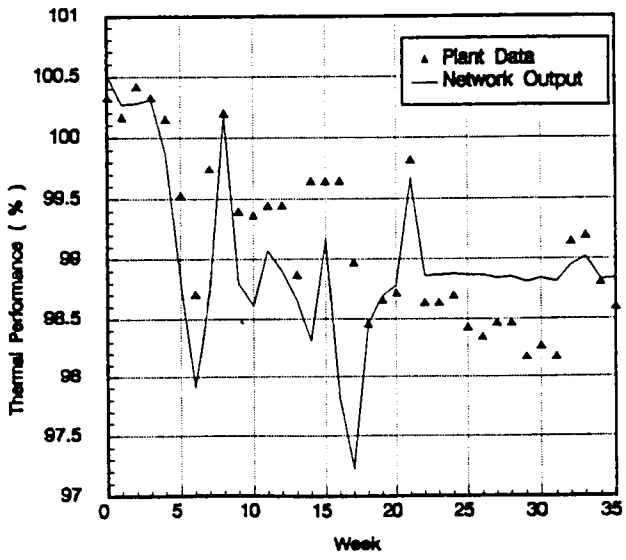

Figure 3.5: Comparison of plant data with neural network outputs for Unit 2 by using neural network trained only on Unit 1 data

Equation 3.1 shows that the partial derivative depends not only on the network connections, w_{ij} and v_{jk}, but also on the activations of neurons in both hidden layer and output layer, which in turn depend on input patterns. The importance of an input variable is measured by the sum of the absolute values of $\partial y_k/\partial x_i$ over all input patterns, which gives the measure of the importance of x_i with respect to y_k on a global or average sense.

3.5 Results From the Sensitivity Analysis

It can be seen from Figures of 3.2, 3.3, 3.4, and 3.5, that the errors are very small for the network predictions for both Unit 1 and Unit 2 even though the network was trained only by using the data of Unit 1. The small prediction error indicates that the training was successful and the trained network could be studied further for the sensitivity analysis to determine the important variables which might strongly affect the heat rate and thermal performance of the plant. The trained *backpropagation* neural network was used together with plant data from Unit 1 and Unit 2 for the sensitivity study by using equation 3.1. Table 3.2 gives the results of the sensitivity analysis, where TP is an acronym for *Thermal Performance*, and GHR is an acronym for *Gross Heat Rate*.

The results from the sensitivity study showed that there was very good agreement for identifying the important variables for both Unit 1 and Unit 2. The four most important variables selected for *Gross Heat Rate* for both Unit 1 and Unit 2 are identical and listed in Table 3.3 in the order of importance.

3.6 More Exploration

The sensitivity study can be applied further to explore more information about the system. Table 3.2 shows that variable number 21, *Unexpected Power*

Table 3.2: Results From the Sensitivity Study

Variable	Order of Importance			
	Unit 1		Unit 2	
	To TP	To GHR	To TP	To GHR
1	18	23	16	20
2	14	17	10	14
3	2	10	7	10
4	11	5	21	8
5	5	6	8	9
6	6	13	12	12
7	21	18	20	14
8	17	15	17	15
9	7	7	6	6
10	3	4	3	4
11	10	8	2	5
12	9	2	4	2
13	8	9	18	7
14	13	3	5	3
15	19	16	14	13
16	23	20	22	21
17	22	22	23	24
18	20	24	19	23
19	15	19	13	19
20	12	11	9	11
21	1	1	1	1
22	16	14	15	16
23	24	21	24	22
24	4	12	11	18

Table 3.3: Most Important Variables Selected for *Gross Heat Rate*

1	Unexpected Power Deviation (Loss +)
2	Measured Condenser Backpressure
3	Condenser Circulating Water Inlet Temperature
4	Total Unaccounted Megawatts (Loss +)

Table 3.4: Most Important Variables Selected for *Unexpected Power Deviation*

1	Total Unaccounted Megawatts (Mwe) (Loss +)
2	Other Condenser Loss (Mwe) (Loss +)
3	Auxiliary Steam Loads (Mwe) (Loss +)
4	Other Misc. Loads (MWe) (Loss +)

Deviations, is the most important variable for both *Gross Heat Rate* and *Thermal Performance.*

It is useful to carry the analysis one step further to evaluate the most important variables related to *Unexpected Power Deviation.* Another neural network was set up with *Unexpected Power Deviations* as its target variable and the 23 remaining variables in Table 3.1 (excepting *Gross Heat Rate* and *Thermal Performance*) as its input variables. The training process was the same as before, which used the *self-organization* network first to cluster the original data patterns to reduce the total training patterns for the following *backpropagation* network by using the centroids of the clusters formed. Again, the training was successful and the sensitivity study was applied to the trained network to explore more information. The results of the study are listed in Table 3.4.

These process can be carried out on and on to explore more and more detailed information about the system. The results from the sensitivity study indicate that the heat rate variation might be caused mainly by the parameters listed in Table 3.3 with *Unexpected Power Deviation* as the most important parameter. In turn, *Unexpected Power Deviation* might be caused by the parameters listed in Table 3.4. This information gives the clues to the plant personnel as to which variables are most important to the plant efficiency and which efforts to improve the efficiency will be most effective.

3.7 Discussion and Remarks

The results from the sensitivity analysis show that neural networks can be trained to work as an internal model of the system and the sensitivity analysis can be used to further study successfully trained networks. The sensitivity analysis can extract useful information which may be difficult to obtain through the traditional analytical methods. The application to heat rate and thermal performance study for *TVA's Sequoyah* nuclear power plant may provide useful information to the power plant personnel to find what may cause the deviation in the plant thermal performance and the heat rate, and then, try to control them in order to operate the plant more efficiently.

CHAPTER 4

FAULT DIAGNOSTICS FOR NUCLEAR POWER PLANTS

4.1 Introduction

A nuclear power plant is a very complex system with hundreds of subsystems integrated together to perform various functions. The operating conditions of a nuclear power plant need to be monitored to insure that the plant is operated safely and efficiently. Diagnoses of potentially unsafe plant conditions should be quick and accurate and the corrective actions must be taken as soon as possible. The diagnoses of transient or accident conditions in a nuclear power plant are sometimes difficult because the information obtained from many different sensors in the plant may be noisy, intermittent, or even incomplete.

Neural networks have some remarkable features, such as error tolerance and non-algorithmic simulation and may be used to improve the diagnostic systems. The error tolerance makes the networks suitable for processing noisy and incomplete data, and the non-algorithmic simulation makes it possible to model some complex systems, where only the data of system inputs and outputs are available. In addition, the neural computation when implemented in hardware is essentially parallel, which gives neural networks the power in processing a large number of signals in real time.

Uhrig [34] reviewed the various applications of neural networks to nuclear power plants, including fault diagnostics, reactor control, sensor validation, plant status monitoring, design of nuclear fule cycle reload, licensee event report data base access, and analysis of vibrations. He concluded that neural networks offer an interseting and productive means of addressing many of the problems that occur in nuclear power plants and have the potential to enhance the performance and safety of nuclear power plants in a cost-effective way.

Table 4.1: Accident Scenarios Simulated on Watts Bar Simulator

1	Total loss of offsite power (ED1)
2	Main feedwater line break (F23)
3	Main steam line break (MS1)
4	Control rod ejection (RD6)
5	Hot leg loss of coolant accident (TH1)
6	Cold leg loss of coolant accident (TH2)
7	Steam generator tube leak (TH5)

Because many fault conditions which need to be recognized have never actually occurred, the data from the computer simulation of the *TVA's Watts Bar* nuclear power plant are used in this dissertation.

4.2 Simulation Data of Watts Bar Nuclear Power Plant

The data obtained from *TVA's Watts Bar* simulator covers the seven accident scenarios listed in Table 4.1.

The data set for each scenario contains 81 plant process variables sampled at intervals of 0.5 or 1.0 second and the total sampling time for each scenario is 200 to 500 seconds. Within each data set the accident condition is preceded by a period of normal full power operation. The data is normalized so that a value of one is equivalent to 100% of a variable's meter reading in the control room [36]. Thus, all variables have values between zero and one. These accident scenarios are described in Appendix B and the 81 variables obtained from the simulator are listed in Appendix C.

4.3 Review of the Previous Study

Bartlett [35] used the same data set from *TVA's Watts Bar* training simulator in his doctoral dissertation. He reduced the variables from total of 81 to 27

Table 4.2: Desired Network Output for the Transients

Code	Plant Condition	Desired Output
ED1	Total loss of offsite power	1 1 1
F23	Main feedwater line break	1 1 0
MS1	Main steam line break	1 0 1
RD6	Control rod ejection	1 0 0
TH1	Hot leg loss of coolant	0 1 1
TH2	Cold leg loss of coolant	0 1 0
TH5	Steam Generator tube break	0 0 1
	Full power normal operation	0 0 0

by averaging the variables associated with each of the four reactor primary coolant loops, and then, eliminating 10 out of the remaining 37 variables to simplify the neural network system. This variable reduction effectively produced a single loop plant and reduced both the amount of redundant data and the size of the input vector to the neural network. The 27 chosen variables at a single time step were treated as one input pattern for the neural network. It was assumed that these 27 variables contain the required information to determine the plant status. Each plant condition listed in the Table 4.1 was assigned a binary code as the target values for training of the network. The binary codes and the corresponding accidents are listed in table 4.2.

The simulated condensation network was used by Bartlett [35] to distinguish the normal and accident conditions listed in Table 4.2. There were two distinct sets of training files. One contained patterns (without adding noise) directly from the simulation data, and the other one contained the noisy patterns created by adding 5% uniformly distributed pseudo-random noise to the simulation data to form the noisy training patterns. The network was trained with the final configuration which contained 27 neurons in the input layer, 19 in the first hidden layer, 12 in the second hidden layer, 5 in the third hidden layer, and 3 in the output layer.

Table 4.3: Time required to diagnose the transients

Code	Plant Condition	Diag. time(sec)		Scram Time (sec)
		0% Noise	2% noise	
ED1	Total loss of offsite power	6.0	5.5	9.0
F23	Main feedwater line break	18.5	62.0	15.5
MS1	Main steam line break	29.0	66.0	11.0
RD6	Control rod ejection	20.0	19.0	44.0
TH1	Hot leg loss of coolant	29.0	47.5	4.0
TH2	Cold leg loss of coolant	19.5	37.5	4.5
TH5	Steam Generator tube leak	86.0	113.0	379.5

The results of the application are summarized in Table 4.3. It can be seen from the table that the length of time required to diagnose the transients for the network are longer than the corresponding reactor scram time in 4 out of 7 diagnosed transients. The 2% noise degraded the network performance significantly in some case, such as the transient recognition time for main feedwater line break from 18.5 seconds for noiseless data to 62.0 seconds for data with 2% noise, even though the network was trained with 5% noise.

4.4 Use of Hybrid and Modular Neural Networks for the Study

In the study of the thermal performance of *TVA's Sequoyah* nuclear power plant, both a *self-organization* network and a *backpropagation* network were used to form a hybrid network system, which gave much better performance than a single *backpropagation*. This technique is used in the diagnostics of the accident scenarios simulated by the *Watts Bar Training Simulator*. Sensitivity analysis is also applied to guide the selection of input variables for the modular networks, each one of which responses only one accident scenario.

Each raw data file has about 500 data points for each of 81 variables. The training patterns are formed by the data points of the selected variables at one time

step. Therefore, there will be about 500 patterns, each with 81 variables, for each accident scenario, and 3,500 patterns for the total of 7 accident scenarios. It is difficult to train a neural network with such a large number of training patterns. In the previous study (Bartlett [35]), the patterns were averaged over some time intervals to reduce the total number of training patterns. A disadvantage of this approach is that the temporal information is lost, especially for the rapid transients. The *self-organization* network can be used to compress the data and reduce the number of training patterns without substantial loss of information. If data patterns are compressed by the *self-organization* network through clustering, patterns gathered in one cluster will have the same or very similar characteristics defined by the Euclidean distance, and this group of patterns can be represented by its centroid. The advantage of using *self-organization* is that the temporal information is retained after the data compression. One cluster may contain tens of or even hundreds of data patterns if they are similar, such as the data patterns in the time interval of full power operating condition and in the equilibrium stage of the transients. On the other hand, one cluster may only contain one pattern as in the case of rapid transients. By using the centroids of the formed clusters as the training patterns for the *lateral feedback* neural network, the total number of data patterns has been greatly reduced from 3,354 to 138.

Different variables may play different roles in different accident scenarios and only a few key variables are necessary for a network to distinguish one accident from another. If a single network is designed to monitor all possible transients or accident scenarios, it must have a large input layer because the important variables of each and all accident scenarios have to be considered as the input variables. One or more large hidden layers and a large output layer is also required to make it possible for the network to distinguish all scenarios. It will be very difficult or even impossible to successfully train a large neural network because the limitations of

learning ability, convergence , and stability in many frequently used neural networks. One alternative is to build a set of small-scale neural networks, called *modular networks*, each one of which responds to only one accident scenario. The size of the modular networks can be small because only the key variables for that particular accident are used as the input variables for the corresponding modular network. There will be only one neuron in the output layer whose output will be 1 or 0. 1 indicates that the accident has occurred and 0 indicates that it has not occurred. The concept of modular network will be very useful in the application to nuclear power plants where a large number of parameters and potential accidents need to be monitored.

4.4.1 Preparation of Data for Networks

The data of each simulated accident scenario is stored in one data file called raw data file, such as ed1.dat which is the data file for the accident scenario of *total loss of offsite power*. The computer code *mkdata.c* is designed to extract required data from the raw data files to form the training and recall data files for the networks. The code and a sample of task file for running the code are given in Appendix D. 22 variables are selected from the total 81 variables and listed in Table 4.4. The selection for these 22 variables is based on the author's best knowledge about the roles the variables played in the system. The 81 variables in the raw data files can be reduced to 37 if eliminating the redundant variables which are sampled in four identical loops. The further reduction of the 37 variables is to eliminate the variables related to reactor control system and radiation monitoring system. The control and shutdown bank array positions should be reflected by reactor power level. The radiation in a nuclear power plant is usually caused by the leakages in the system. If there is leakage in the RCS system, the variables in the RCS system should also reflect the changes in the system. Therefore, the variables related to the

Table 4.4: Variables Selected as Inputs for the Network

Code	Plant Process Variable
0	Nuclear Power Level
1	Unit Megawattage Output
2	Volume Control Tank Level
3	RB FL & Equip Drain Level
4	Containment Pressure
5	P. R. % Full Power AO Array
6	SG #1 Steam Flow Channel 1 Meter
7	SG #1 Main Steam HDR Pressure
8	SG #1 Feedwater Inlet Flow
9	SG #1 Feedwater Supply Pressure
10	SG #1 Aux. Feedwater Inlet Flow
11	SG #1 Level Indicator
12	SG #1 Wide Level Indicator
13	RCS Pressurizer Level
14	RCS Pressurizer Pressure
15	RCS Pressurizer Surge Line Temp.
16	RCS Loop 1 Spray Temp.
17	RCS Loop 1 Hot-leg Pressure
18	RCS Loop 1 Cold-leg Temp.
19	RCS Loop 1 Hot-leg Temp.
20	RCS Loop 1 Average Temp.
21	RCS Loop 1 Coolant Flow

control and shutdown bank, and to the radiation monitoring system are eliminated to reduce the number of variables used by the neural networks.

The data extracted from each raw data file are stored as two separate files. One contains the data under normal full power operation, and the other contains the data under transient condition, because the full power data and the transient data need to be compressed by *self-organization* separately. All normal full power data files are concatenated together to form one data file. Therefore, the total data files are eight including one for normal full power operation and seven for the accident scenarios. These 8 data files will be used as the training data files for *self-organization* to create clusters and as the recall data files for the *lateral-feedback*

Table 4.5: Binary Representation of 8 Accident Scenarios

Plant Condition	Binary Code	Data File
Normal Full Power Operation	0 0 0 0 0 0 0	full.pat
Total loss of offsite power	1 0 0 0 0 0 0	ed1t.pat
Main feedwater line break	0 1 0 0 0 0 0	f23t.pat
Main steam line break	0 0 1 0 0 0 0	ms1t.pat
Control rod ejection	0 0 0 1 0 0 0	rd6t.pat
Hot leg loss of coolant	0 0 0 0 1 0 0	th1t.pat
Cold leg loss of coolant	0 0 0 0 0 1 0	th2t.pat
Steam Generator tube leak	0 0 0 0 0 0 1	th5t.pat

Table 4.6: Clusters Formed for Each Data File

Data File	Patterns	Clusters	Eu-Distance
full.pat	255	2	0.05
ed1t.pat	459	17	0.10
f23t.pat	396	21	0.10
ms1t.pat	396	19	0.10
rd6t.pat	440	18	0.10
th1t.pat	469	25	0.10
th2t.pat	471	23	0.10
th5t.pat	468	13	0.10

to insure the training success. The data files listed in Table 4.5 are used for the training of *self-organization* and the results of the training are listed in Table 4.6.

The centroids of all clusters listed in the table are used as the training patterns for the *lateral feedback* network. The computer code *nndata.c* is designed to put all the clusters together in the form of network training data file. Appendix E gives the details about the code. The total training patterns are 138. It can be seen that by using the *self-organization* the training data patterns can be greatly reduced for the further supervised training. In this application the training patterns are reduced from original 3,354 to 138 through training of the *self-organization.*

4.4.2 Configuration of Motherhood Network

Before applying the sensitivity analysis to guide the selection of input variables for the modular networks, a motherhood network, which monitors all accident scenarios and uses all input variables listed in Table 4.4, has to be trained successfully. The network type used for the motherhood network is *lateral feedback*, whose configuration and learning algorithm have been discussed in Chapter 2. The network set up for this application had 3 layers. The input layer had 2×11 neurons to match the 22 selected variables. The hidden layer had 4×4 neurons by rule of thump. And the output layer contained 1×7 neurons to have each neuron response only one of the seven accident scenarios.

In order to train the network, all seven accident scenarios and the full power operation conditions were encoded as binary strings and listed in Table 4.5. The binary codes were used as the desired outputs for the training of the network.

The training data for the motherhood network were the centroids of the clusters forme through the training of *self-organization.* The total training patterns were 138.

4.4.3 Training and Recall Results

The **lateral feedback** network was trained for 4,500 cycles and converged with maximum error less than 0.05%, which is the relative error of a network output with respect to the corresponding target value. All 8 data files with extension *pat* listed in Table 4.5 were used as the data files for recall. Figure 4.1 through Figure 4.7 are the plotting of the recall results.

Each figure is the responses from an output neuron in the output layer which responses to one assigned accident. Node 1, for example, was trained to respond to the scenario of *Total Loss of Offsite Power.* Figure 4.1 shows that node 1 gives the output values close to 1.0 to the recall data file of *Total Loss*

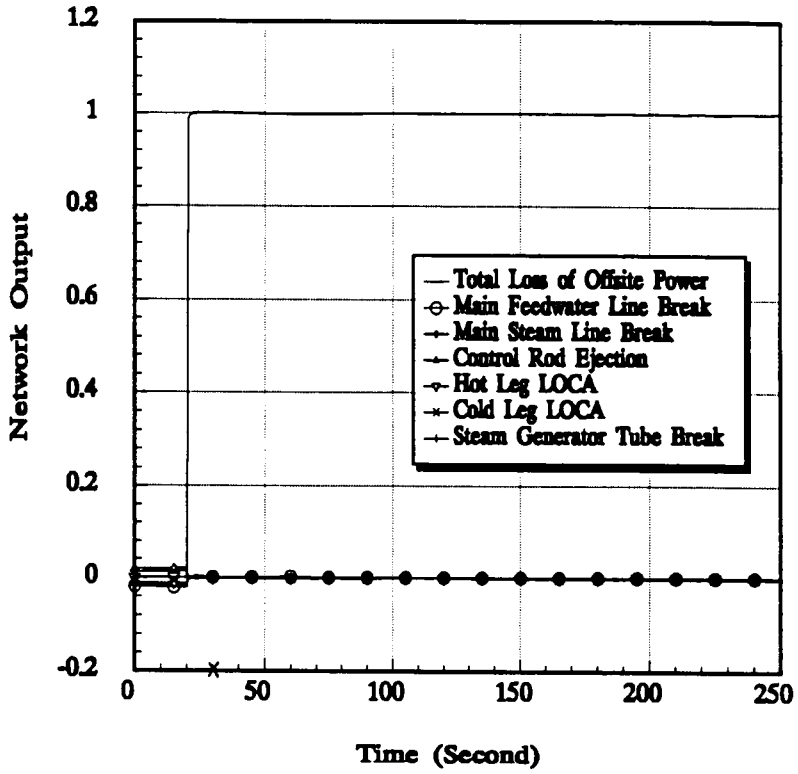

Figure 4.1: Outputs From Node 1 of Motherhood Network Which Responses to Total Loss of Offsite Power

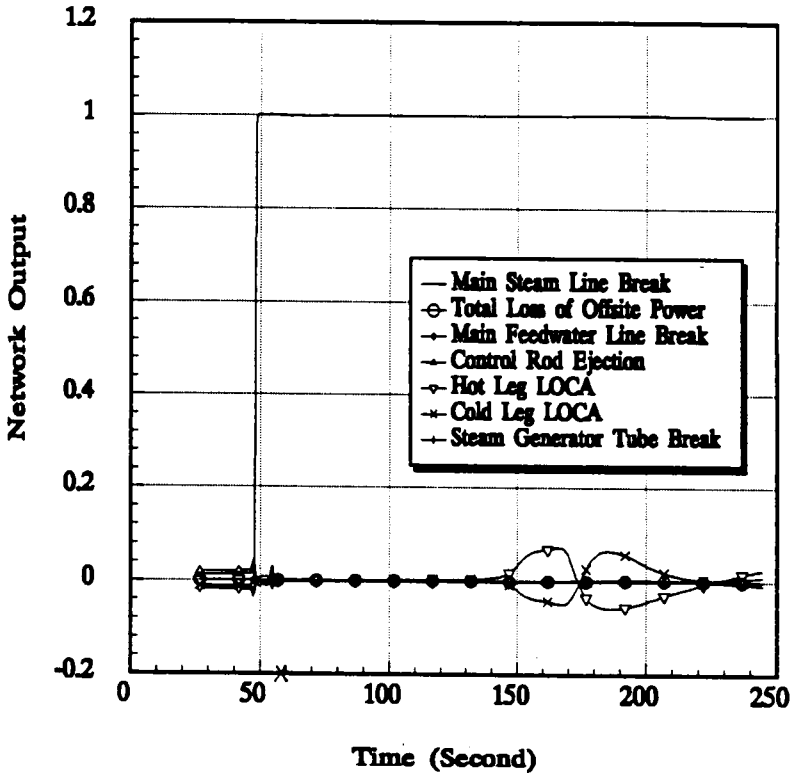

Figure 4.2: Outputs From Node 2 of Motherhood Network Which Responses to Main Steam Line Break

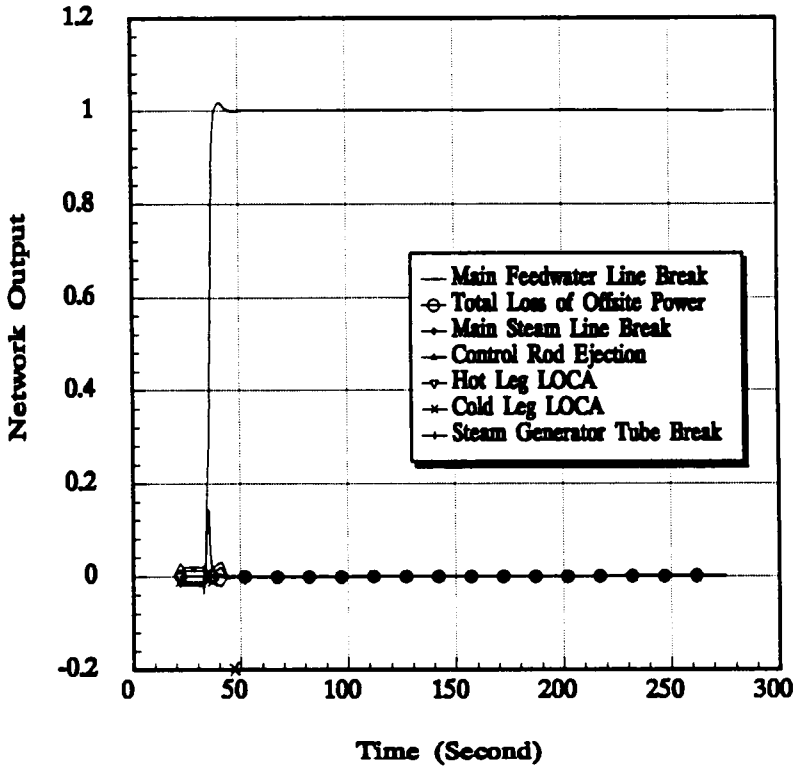

Figure 4.3: Outputs From Node 3 of Motherhood Network Which Responses to Main Feedwater Line Break

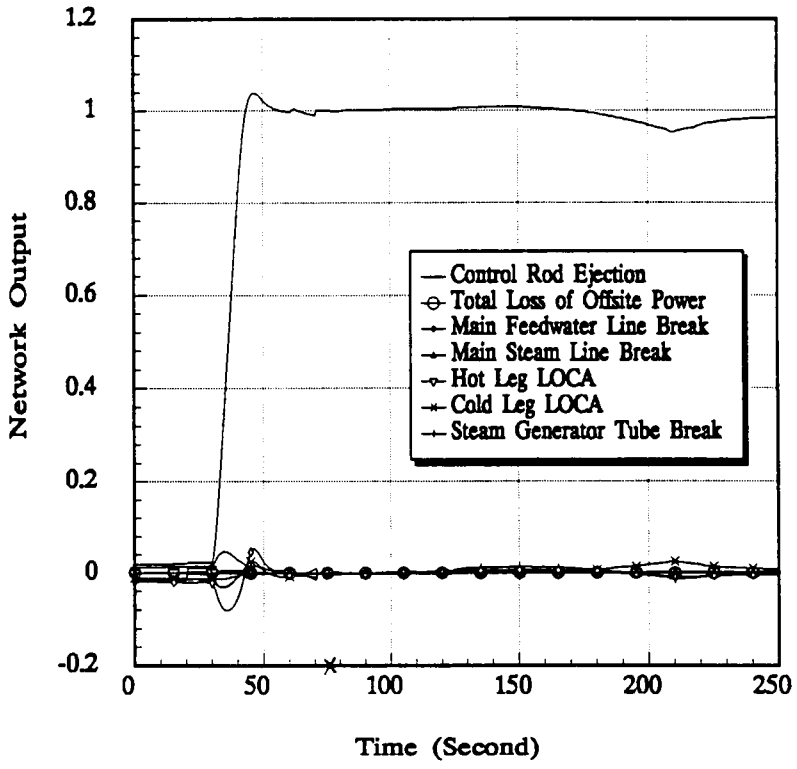

Figure 4.4: Outputs From Node 4 of Motherhood Network Which Responses to Control Rod Ejection

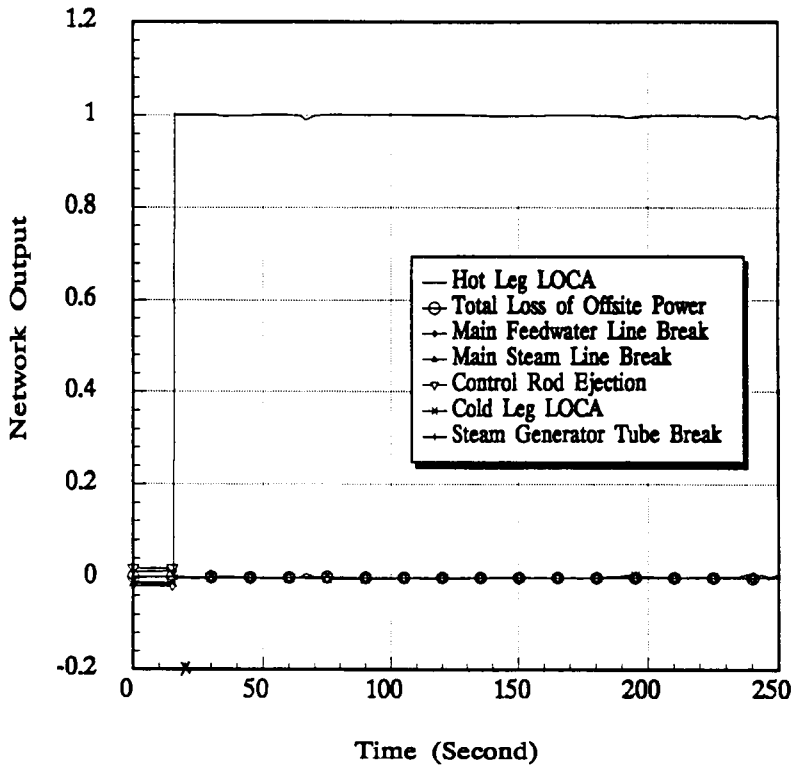

Figure 4.5: Outputs From Node 5 of Motherhood Network Which Responses to Hot-leg LOCA

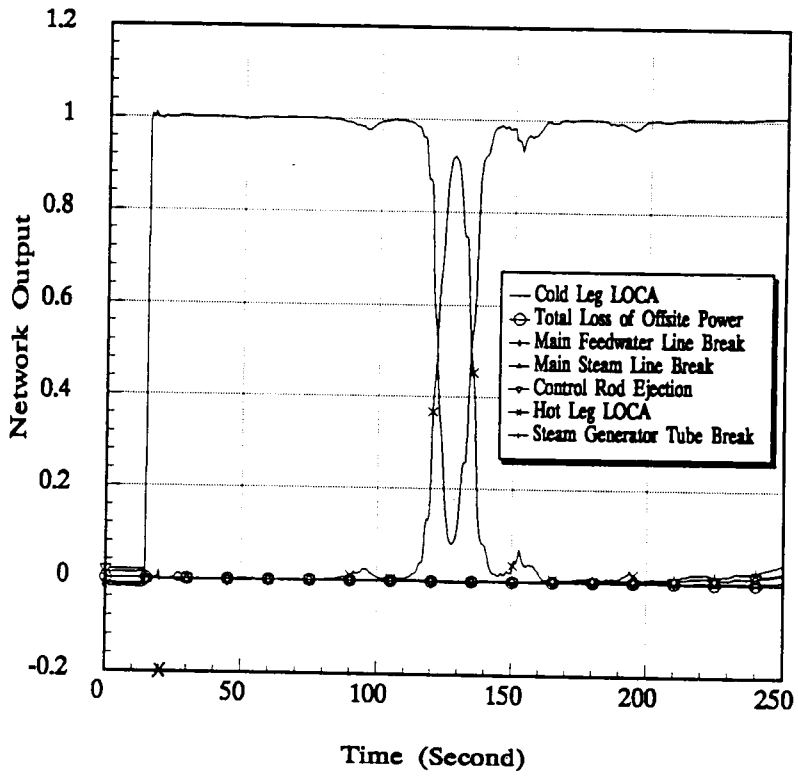

Figure 4.6: Outputs From Node 6 of Motherhood Network Which Responses to Cold- leg LOCA

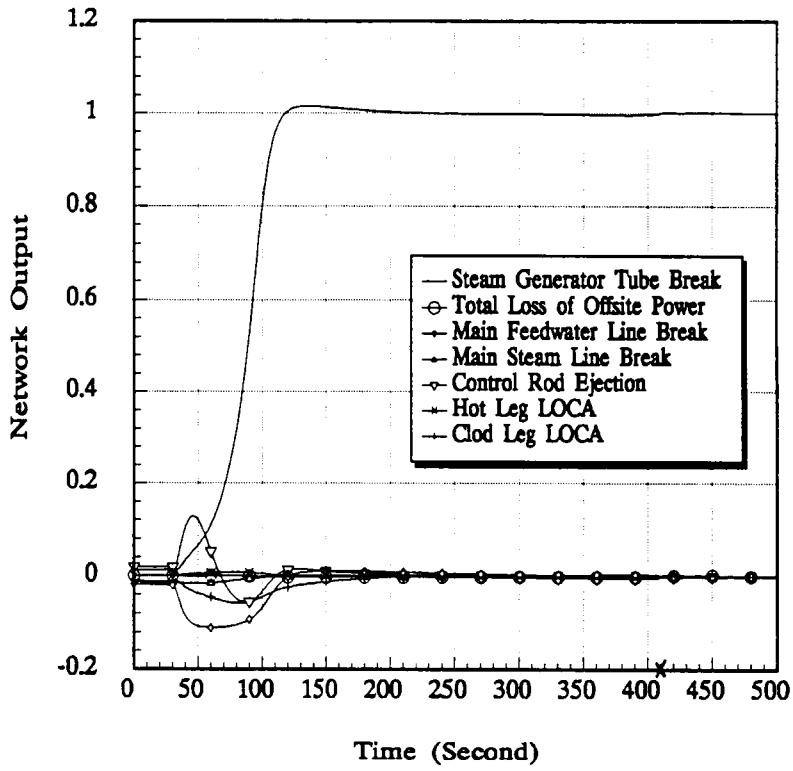

Figure 4.7: Outputs From Node 7 of Motherhood Network Which Responses to Steam Generator Tube Break

of Offsite Power, or *ed1t.pat*, and close to 0.0 to other data files related to other accident scenarios. To judge if *Total Loss of Offsite Power* is predicted incorrectly by other nodes, we can crosscheck Figures 4.2 to 4.7. It can be seen that the responses from all nodes except node 1 are close to 0.0, therefore, the motherhood network gives the correct response, which is (1 0 0 0 0 0 0), to *Total Loss of Offsite Power*. Figures 4.1 to 4.7 show that the motherhood network can distinguish all 7 accident scenarios and the preceding normal full power operation correctly except the scenario of *Cold-leg LOCA* which was confused with *Hot-leg LOCA* in the middle of the transient. The time required for the network to diagnose the transients and the corresponding reactor scram time are listed in Table 4.7. It can be seen from the table that the network can determine all the scenarios at the very beginning of the transients. For slow transients, control rod ejection and steam generator tube break, it takes longer for the network to determine the transients but the time for the diagnoses is still much shorter than the corresponding reactor scram time even though the data used for recall was not used directly for training of the motherhood network but for training of the *self-organization* to form the clusters listed in Table 4.6. Therefore, the training of the motherhood network is successful. The quick and accurate responses to all the scenarios indicates that the combination of *self-organization* and a supervised learning network, i.e. *lateral-feedback*, is suitable for the training tasks where a large amount of data need to be processed.

Figure 4.6 shows that *Cold-leg LOCA* was diagnosed instantly at the beginning of the transient but it was confused with *Hot-leg LOCA* in the middle range of the transient. The reason for this confusion might be that in the middle of the transient there was no distinctions between *Hot-leg LOCA* and *Cold-leg LOCA* when viewing the values of the selected parameters. The network gave the correct response from the very beginning up to 120 seconds which gives the operators enough time to

Table 4.7: Time to for the Network to Diagnose Transients

Plant Condition	Diag. time(sec)	Scram Time (sec)
Total loss of offsite power	0.5	9.0
Main feedwater line break	0.5	15.5
Main steam line break	0.5	11.0
Control rod ejection	10.0	44.0
Hot leg loss of coolant	0.5	4.0
Cold leg loss of coolant	0.5	4.5
Steam Generator tube leak	90.0	379.5

take necessary actions. To correct the confusion some missing key variables, which can distinguish between *Hot-leg LOCA* and *Cold leg LOCA* in the middle range of the transient, need to be added to the training input variables.

4.4.4 Modular Networks

The same diagnostic task can be performed by a set of small modular networks, each one of which monitors only one accident scenario. The performance of the neural network diagnostic system can be improved by using modular networks because they are small and can be operated parallelly , and therefore, speed up the responses to accident scenarios. The parallel operations of modular networks also improve the fault tolorence of the diagnostic system because one damaged modular network may affect only the performance of itself not the performance of other networks. To build a set of modular networks the input variables for each modular network have to be selected. The sensitivity analysis established in Chapter 2 can be applied to the previously trained motherhood network to determine the important variables for each accident scenario. The correct and fast response for all 7 accident scenarios insure that the previously trained motherhood network can be treated as the mapping function, which could, then, be studied further by the sensitivity analysis. The sensitivity study ranked the 22 input variables in the order

Table 4.8: Results of Sensitivity Study for Node 1

Input No.	$\sum \|dy/dx\|$	Rank
3	0.0042	1
1	0.0036	2
21	0.0035	3
14	0.0034	4
9	0.0025	5
7	0.0019	6
13	0.0018	7
8	0.0015	8
2	0.0013	9
17	0.0012	10
4	0.0012	11
12	0.0010	12
10	0.0009	13
11	0.0009	14
15	0.0009	15
5	0.0008	16
0	0.0008	17
19	0.0005	18
6	0.0005	19
18	0.0004	20
20	0.0004	21
16	0.0002	22

of importance for each output node which represents an assigned accident scenario. Therefore, the variables for each modular network could be selected according to their importance to the corresponding scenario. The results of sensitivity analysis for node 1, for example, which responses for the accident scenario of *Total Loss of Offsite Power*, are summarized in Table 4.8.

The first 5 variables in Table 4.8 were chosen, according to their $\sum \|\frac{dy}{dx}\|$ values, or order of importance, as the training input variables for the modular network which responses only to *Total Loss of Offsite Power*. The variables for other modular networks were selected in the same manner according to their importance measures with respect to the accident scenario monitored by the corresponding

Table 4.9: Variables Selected for Each Modular Network

Modular Network for	Input Variables Selected	# Variables
Total loss of offsite power	3 1 21 14 9	5
Main steam line break	6 7 12 9 13 8 1 14 21	9
Main feedwater line break	13 14 21 11 9 3 6	7
Control rod ejection	13 21 3 14 1 15 4 9	8
Hot leg loss of coolant	21 4 13 11 17 7 9	7
Cold leg loss of coolant	13 21 4 11 14 17 9	7
Steam Generator tube leak	13 21 14 3 9	5

modular network. The variables selected for each modular network are listed in Table 4.9.

The physical meaning of each selected variable can be retrieved from Table 4.4. For *Hot-leg LOCA* and *Cold-leg LOCA*, for example, the first three important variables are selected as 4, 13, and 21, which are, in turn, *Containment Pressure, RCS Pressurizer Level*, and *RCS Coolant Flow.* These three variables respond rapidly during the accidents and can be considered as the key parameters for the networks to determine what actually happened in the system. Because the plant process variables give similar responses for *Hot-leg LOCA* and *Cold-leg LOCA*, the key variables used to detect the accidents should be identical or similar. The sensitivity study has drawn the same conclusion and selected the virtually the same variables as the important variables for both *Hot-leg LOCA* and *Cold-leg LOCA*. It shows that the sensitivity analysis can provide useful information about the important variables for determining the system status.

4.4.5 Modular Network Configuration, Training and Recall Data

The type of neural network used for all modular networks was *lateral feedback.* All modular networks had three layers. The neurons in the input layer were determined by the number of input variables listed in Table 4.9. There were 4 neu-

rons in the hidden layer for all modular networks, and 1 neuron in the output layer to response the assigned accident scenario. The output value close to 1.0 indicates that the assigned accident has been detected and a value close to 0.0 indicates that an accident has not occur. The training data file for all modular networks was the data file formed by the clusters listed in Table 4.6. The computer code *subnet.c* was designed for the training of all modular networks. The code has the build-in function to extract data from the cluster data file (with 22 input variables) according to the variable selected for each modular network. The code will also automatically assign the target value of 1.0 to the patterns associated with the scenario monitored by the modular network, and 0.0 to the rest of the patterns. One pattern might be assigned a target value of 1.0 or 0.0 depending on which scenario is monitored by the modular networks. Appendix F gives the computer code and instructions for using it.

The recall data files used for all modular networks are the files listed in Table 4.5. These data files have not been used for training of any modular network but for training of the *self-organization*. The computer code *subrecall.c* was designed to process the recall for all the modular networks. It can extract the required data from the files in Table 4.5 (where 22 variables are contained in all files) according to the variable selected for each modular network. Appendix G gives the code and the instructions of running the code.

4.4.6 Recall Results of Modular Networks

The recall was performed for all seven modular networks. Figure 4.8 through Figure 4.14 show the recall results.

There is only one neuron in the output layer for all modular networks. The correct response for a modular network is that its output is close to 1.0 for the assigned accident and to 0.0 for the rest. The recall was performed for each modular

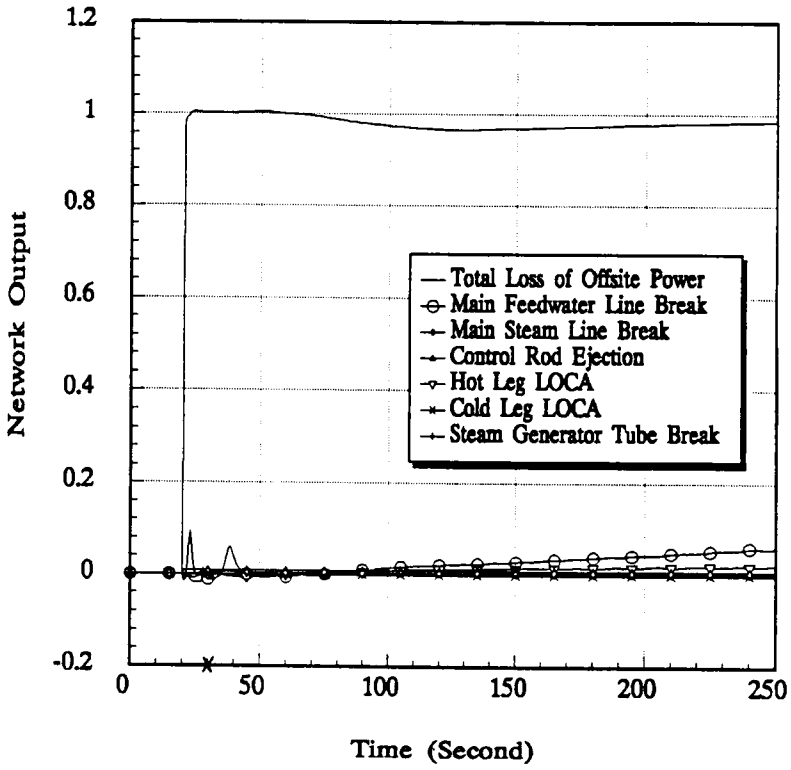

Figure 4.8: Response to All 7 Scenarios From Modular Network of Total Loss of Offsite Power (5 Variables Selected for the Network)

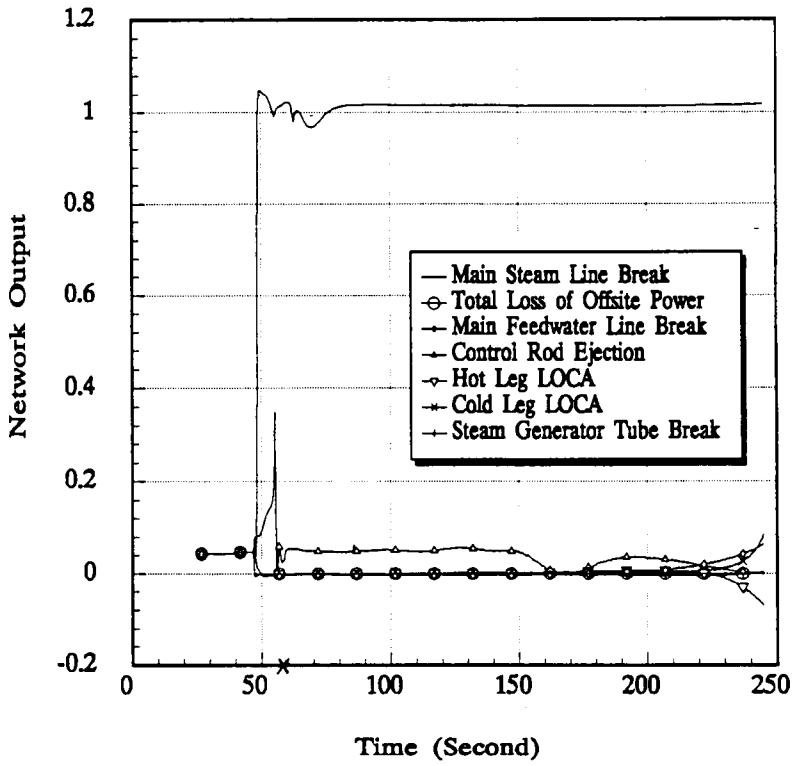

Figure 4.9: Response to All 7 Scenarios From Modular Network of Main Steam Line Break (9 Variables Selected for the Network)

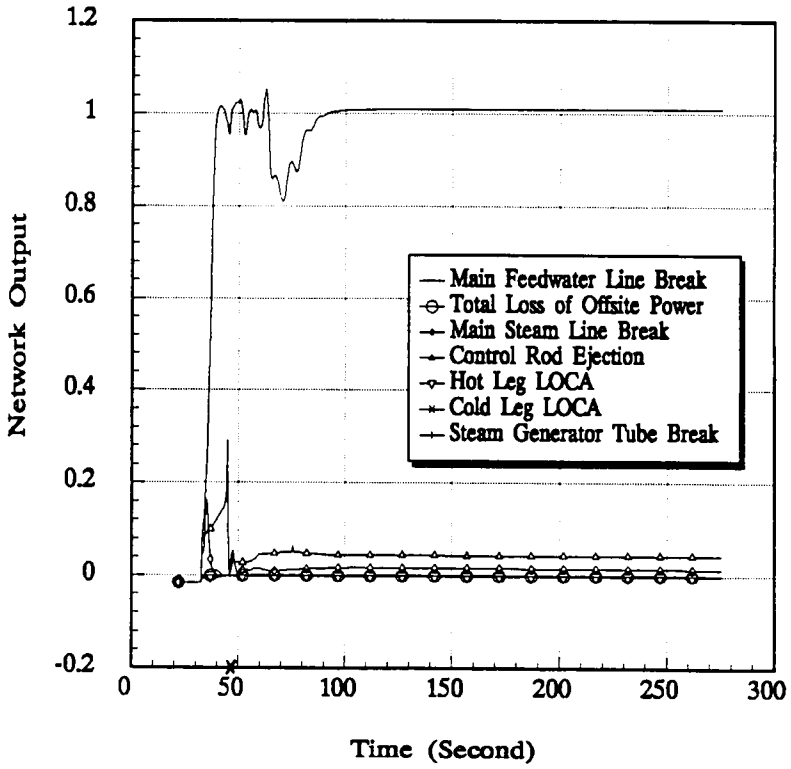

Figure 4.10: Response to All 7 Scenarios From Modular Network of Main Feedwater Line Break (7 Variables Selected for the Network)

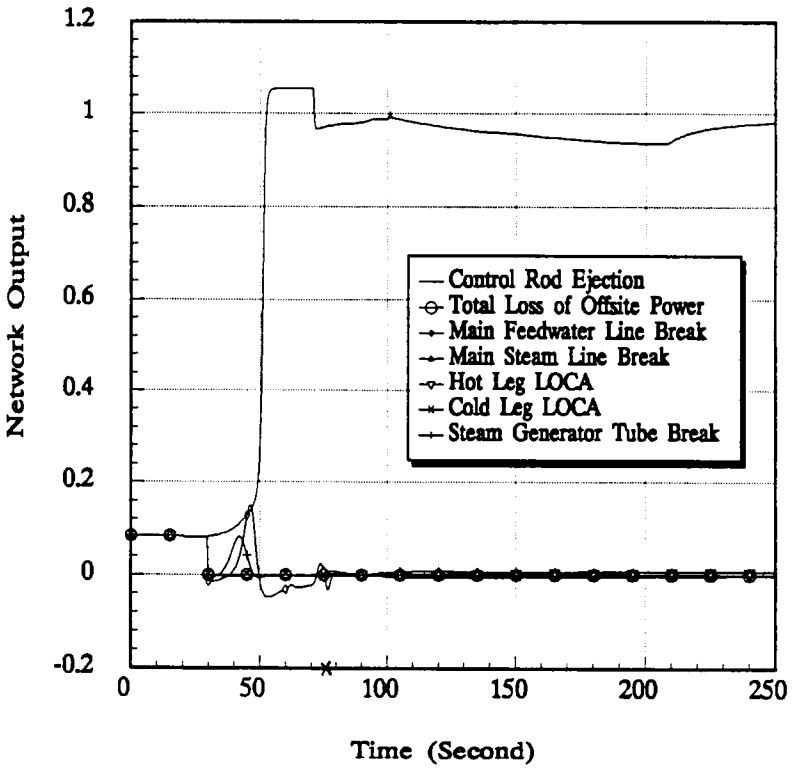

Figure 4.11: Response to All 7 Scenarios From Modular Network of Control Rod Ejection (8 Variables Selected for the Network)

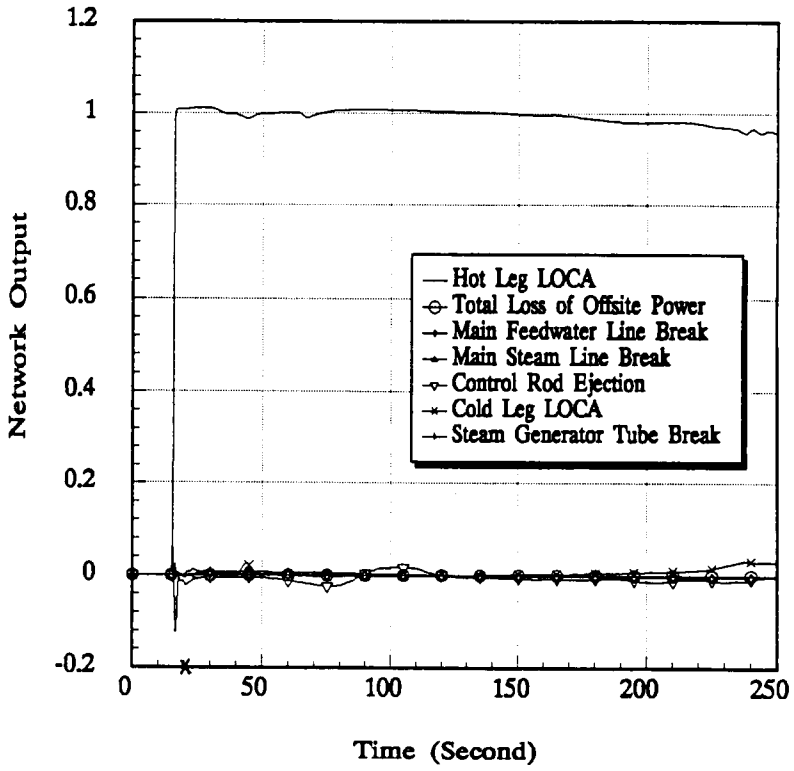

Figure 4.12: Response to All 7 Scenarios From Modular Network of Hot-leg LOCA (7 Variables Selected for the Network)

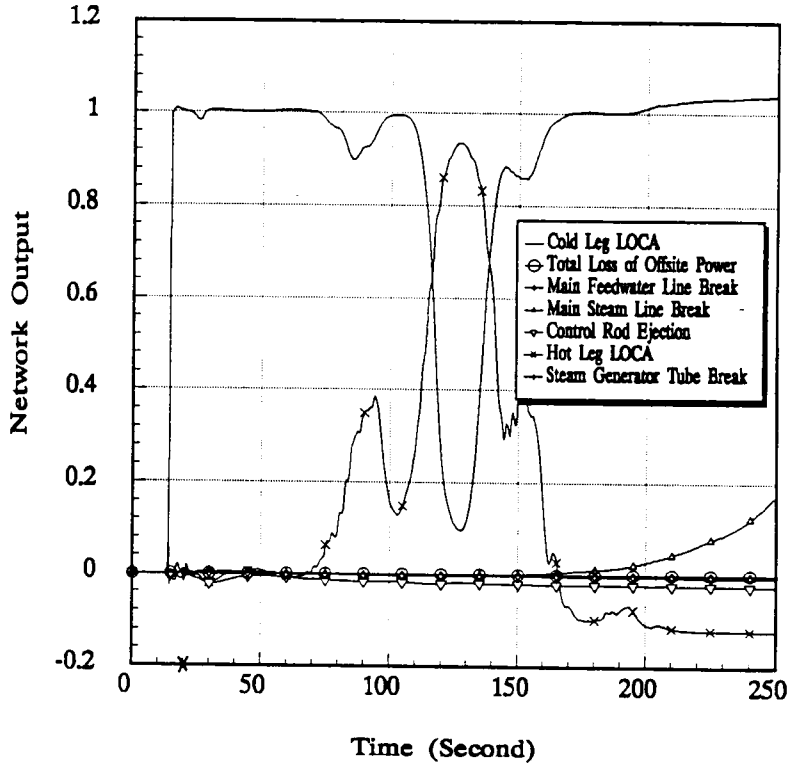

Figure 4.13: Response to All 7 Scenarios From Modular Network of Cold-leg LOCA (7 Variables Selected for the Network)

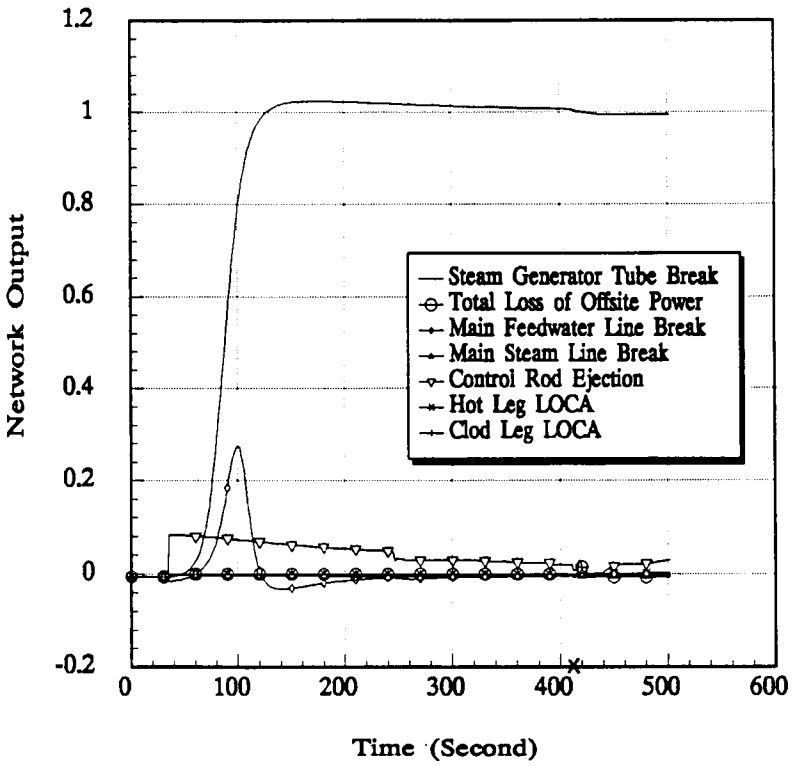

Figure 4.14: Response to All 7 Scenarios From Modular Network of Steam Generator Tube Break (5 Variables Selected for the Network)

network by using all 8 data files, listed in Table 4.6, corresponding to 7 accident scenarios and one full power operation. Therefore, each figure has 7 lines, each one of which is the response of the corresponding modular network to an accident scenario and the preceding full power operation. Figure 4.8, for example, shows that the modular network gives the output values close to 1.0 for the data file of *Total Loss of Offsite Power*, and gives the values close to 0.0 to the rest of the data files related to other accident scenarios. It can be seen that all seven modular networks could give quick and correct responses to their assigned accident scenarios. The time for all seven modular networks to determine the status of the plant was shorter than the corresponding reactor scram time marked as 'x' on the time axes. There were some false predictions at early stage of the transients from the modular network of *Main Feedwater Line Break* and *Main Steam Line Break*, but it did not cause any confusion with other scenarios. The modular network for *Cold-leg LOCA* gave the correct response immediately from the beginning of the transient up to 100 seconds. Again, it was confused with *Hot-leg LOCA* in the middle of the transients. Some methods are used to improve the response from the modular network of *Cold-leg LOCA*, such as weighted inputs which will be discussed in a later section.

4.4.7 Noise Tolerance of the Modular Networks

In order to test the noise tolerance of the modular networks, 2% uniformly distributed pseudo-random noise were introduced to the recall data files and the results from all seven modular networks are given from Figure 4.15 through Figure 4.21.

For the modular networks of *Total Loss of Offsite Power, Hot-leg LOCA*, and *Steam Generator Tube Break*, there were no significant degradation in the network performance. However, for the rest of the modular networks, the 2% noise did degrade the network performance by increased disturbance, such as the responses

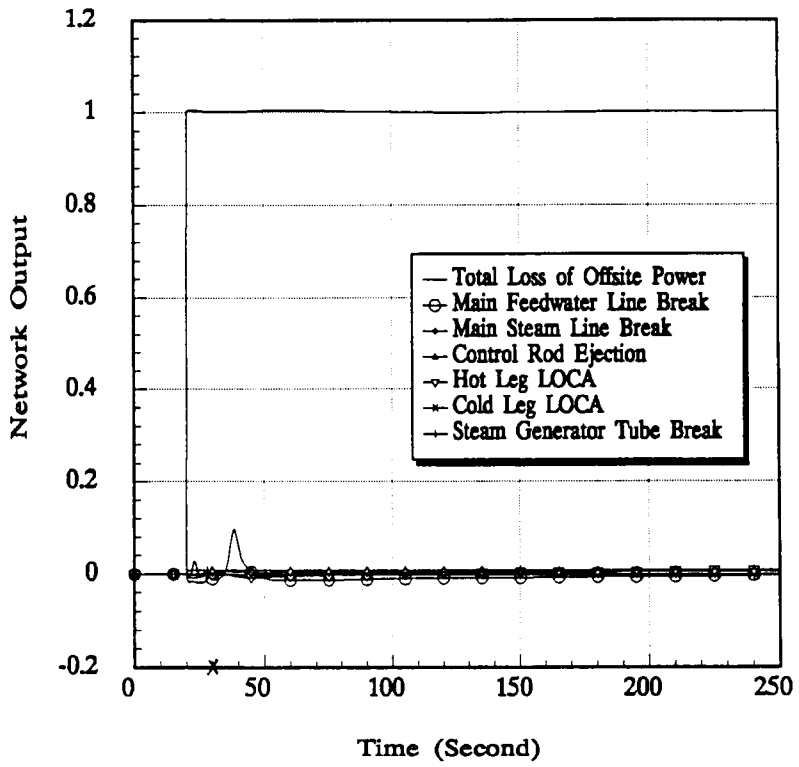

Figure 4.15: Response to All 7 Scenarios From Modular Network of Total Loss of Offsite Power (5 Variables Selected for the Network and 2% Noise Used for Recall)

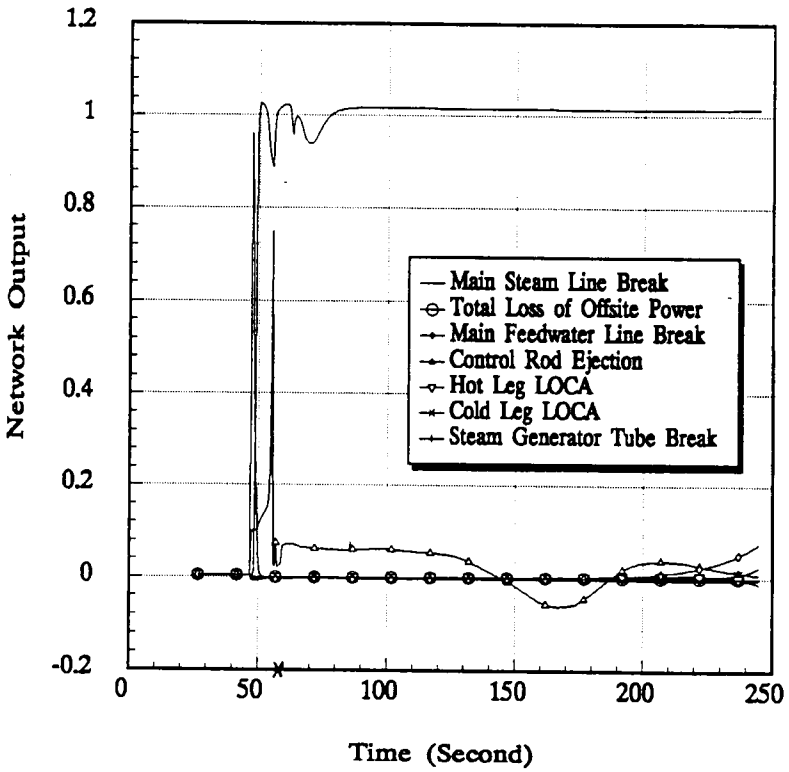

Figure 4.16: Response to All 7 Scenarios From Modular Network of Main Steam Line Break (9 Variables Selected for the Network and 2% Noise Used for Recall)

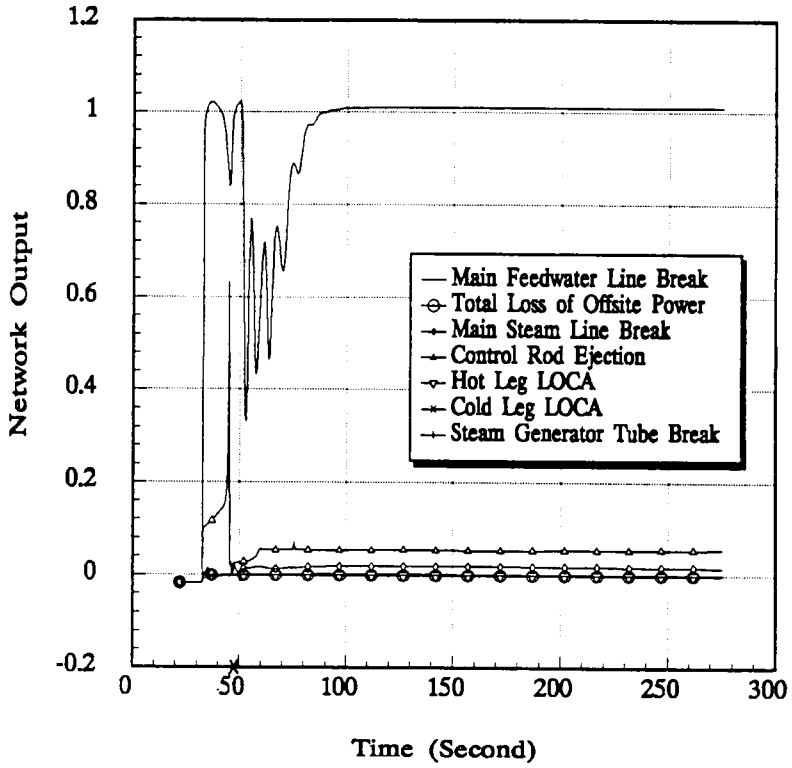

Figure 4.17: Response to All 7 Scenarios From Modular Network of Main Feedwater Line Break (7 Variables Selected for the Network and 2% Noise Used for Recall)

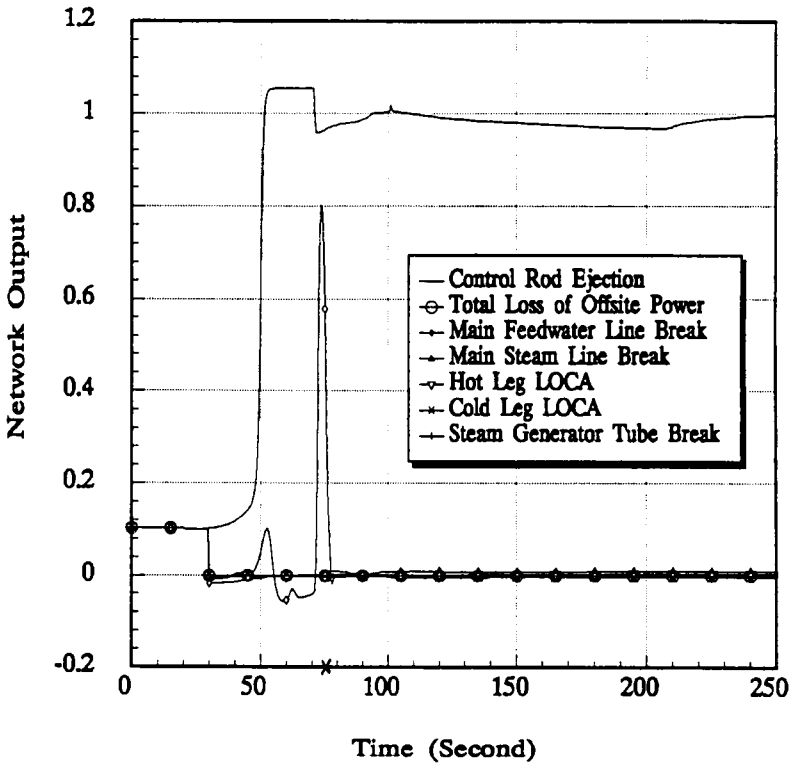

Figure 4.18: Response to All 7 Scenarios From Modular Network of Control Rod Ejection (8 Variables Selected for the Network and 2% Noise Used for Recall)

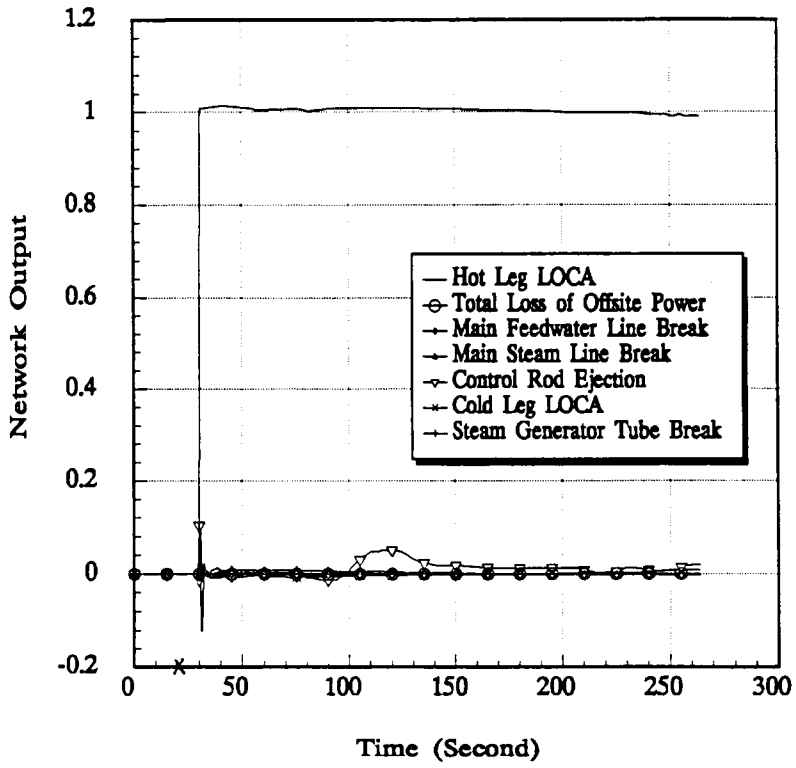

Figure 4.19: Response to All 7 Scenarios From Modular Network of Hot-leg LOCA (7 Variables Selected for the Network and 2% Noise Used for Recall)

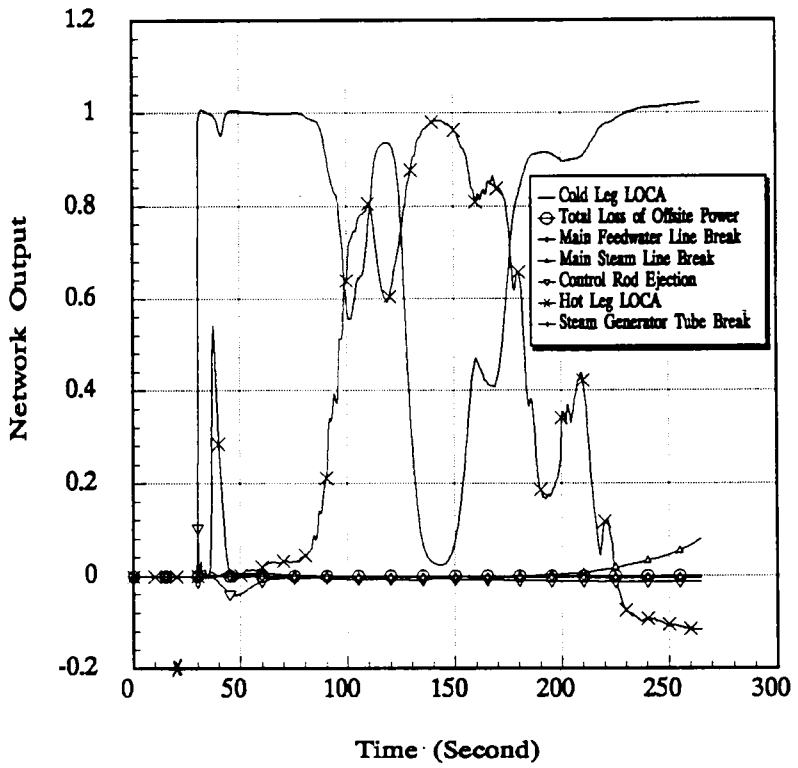

Figure 4.20: Response to All 7 Scenarios From Modular Network of Cold-leg LOCA (7 Variables Selected for the Network and 2% Noise Used for Recall)

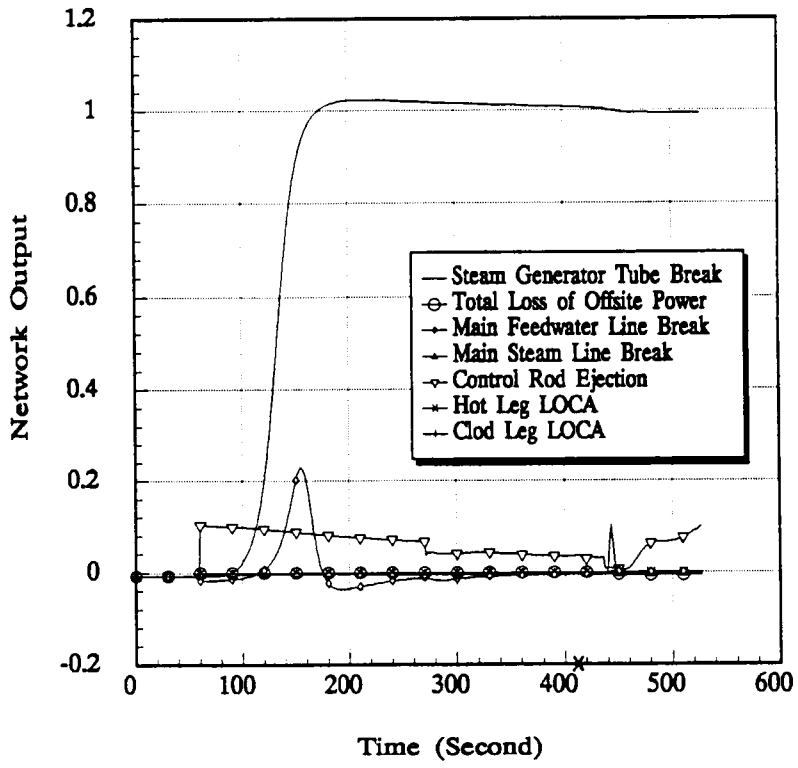

Figure 4.21: Response to All 7 Scenarios From Modular Network of Steam Generator Tube Break (5 Variables Selected for the Network and 2% Noise Used for Recall)

Table 4.10: Time required for modular networks to diagnose the transients

Code	Plant Condition	Diagnosing time(sec)			Scram time
		0% Noise	2% noise	5% noise	
ED1	Total loss of offsite power	0.5	0.5	1.0	9.0
MS1	Main steam line break	1.0	5.0	10.0	11.0
F23	Main feedwater line break	2.0	50.0	50.0	15.5
RD6	Control rod ejection	25.0	25.0	25.0	44.0
TH1	Hot leg loss of coolant	0.5	10.0	10.0	4.0
TH2	Cold leg loss of coolant	0.5	12.0	25.0	4.5
TH5	Steam Generator tube leak	30.0	50.0	70.0	379.5

from the modular networks of *Main Feedwater Line Break*, *Main Steam Line Break*, and *Cold-leg LOCA*, or generating pulses of false signals, such as the response from the modular network of *Control Rod Ejection*. The diagnosing time required for the modular networks to detect the corresponding transients are summarized in Table 4.10. It can be seen from the table that the diagnostic time is increased if there is noise presented in the recall data. For the scenarios of *Main Feedwater Line Break*, *Hot Leg LOCA*, and *Cold Leg LOCA*, 2% noise causes the diagnosing time to be increased dramatically and longer than the corresponding reactor scram time. 5% noise recall was also introduced to all modular networks, and the results are also listed in Table 4.10. It can be seen that increasing noise level is further degraded the networks' performance. Table 4.10 shows that for recall without adding noise the diagnosing time is shorter than the corresponding scram time for all 7 scenarios, and for noisy data recall the diagnosing time for three accident scenarios, *Main Feedwater Line Break*, *Hot-leg LOCA*, and *Cold-leg LOCA*, is actually longer than the corresponding scram time.

4.4.8 Use of Weighted Variables

Variables were selected for each modular network according to their importance to the accident scenario. They were treated as the equally important variables after the selection regardless of their ranks in importance. It is logical to think that a variable with higher rank should play more important role in the network performance. Therefore, the selected variables were weighed according to their rank in importance determined by $\sum \|dy/dx\|$. The approach used for weighing the inputs for this study is the method of amplitude modulation which changes the low and upper limits of normalization proportional to their $\sum \|dy/dx\|$ values, or importance. The weighing method is to increase the difference between the minimum and maximum values of an input, and therefore, make a small difference large, i.e. increasing the amplitude of a small peak in the signal, and making it easy for the network to distinguish it from other peaks. This weighing method was applied to all modular networks and the results are given in Figure 4.22 through Figure 4.28.

It can be seen that this weighing method improved the performance of some modular networks, such as modular network of *Total Loss of Offsite Power, Main Steam Line Break, Main Feedwater Line Break*, and *Steam Generator Tube Break*. For the prediction of *Main Steam Line Break* in Figure 4.23, though there was a false prediction in the final stage of the occurrence of *Cold-leg LOCA*, the overall performance of the modular network was improved by suppressing all other false predictions. Figure 4.23 indicates that the modular network will give value close to 1.0 if *Main Steam Line Break* scenario has occurred in the system, and will give value at most 0.4 only at the final stage of the occurrence of *Cold-leg LOCA*, to which the correct response from the modular network should be close to 0.0. if we crosscheck Figure 4.27, we can see that the modular network of *Cold-leg LOCA* gives the correct response with output value close to 1.0 from the middle of the transient to its end. Therefore, *Cold-leg LOCA* will not be confused with *Main Steam Line*

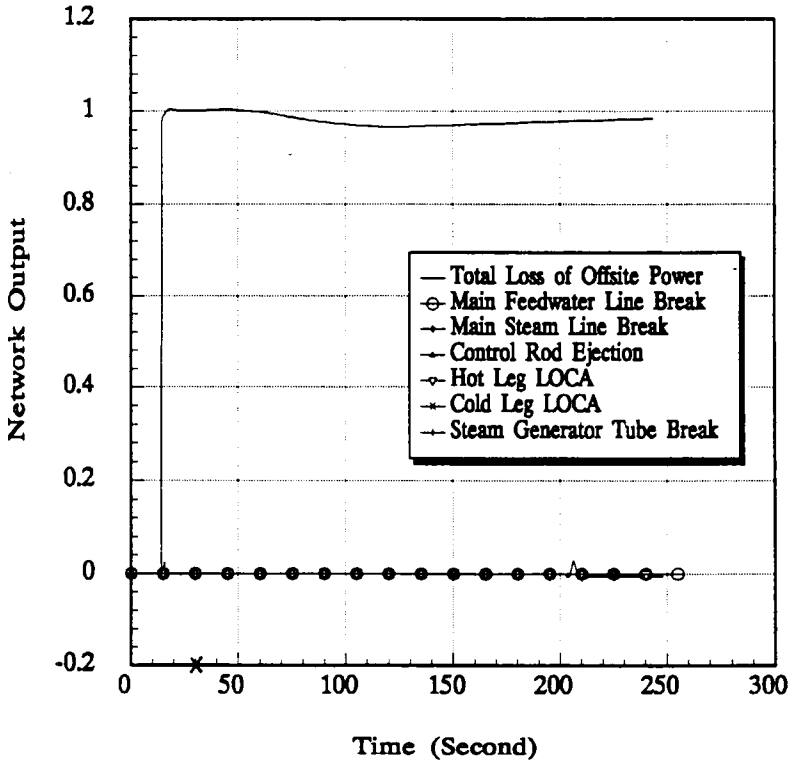

Figure 4.22: Response to All 7 Scenarios From Modular Network of Total Loss of Offsite Power (5 Weighted Inputs Selected for the Network)

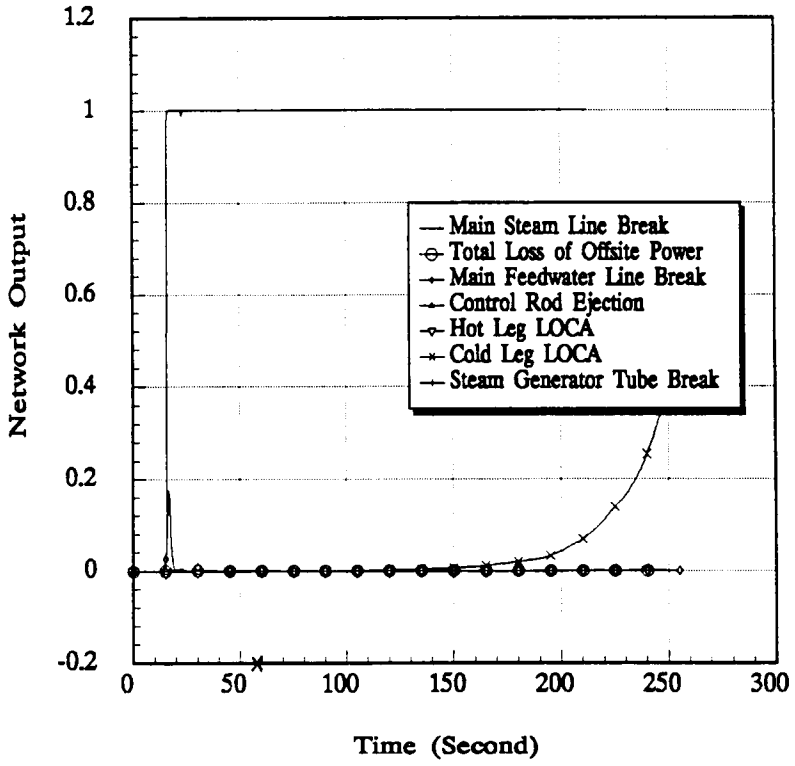

Figure 4.23: Response to All 7 Scenarios From Modular Network of Main Steam Line Break (9 Weighted Inputs Selected for the Network)

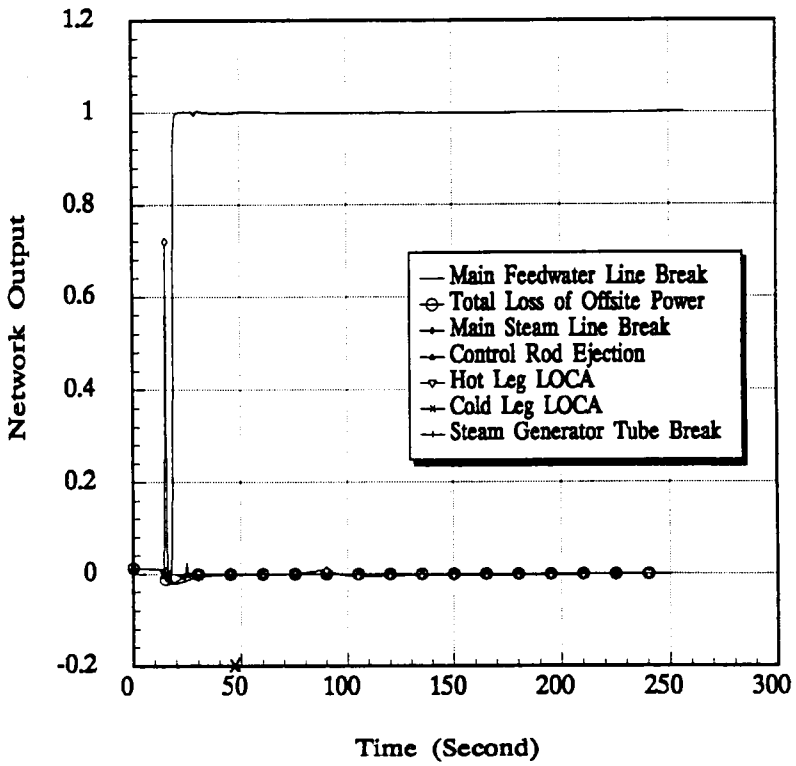

Figure 4.24: Response to All 7 Scenarios From Modular Network of Main Feedwater Line Break (7 Weighted Inputs Selected for the Network)

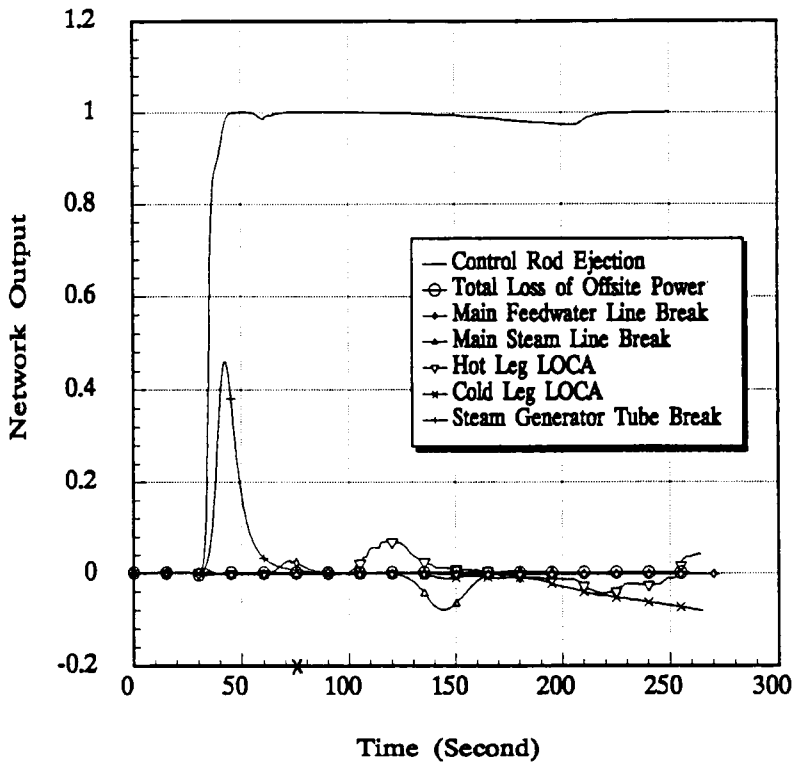

Figure 4.25: Response to All 7 Scenarios From Modular Network of Control Rod Ejection (8 Weighted Inputs Selected for the Network)

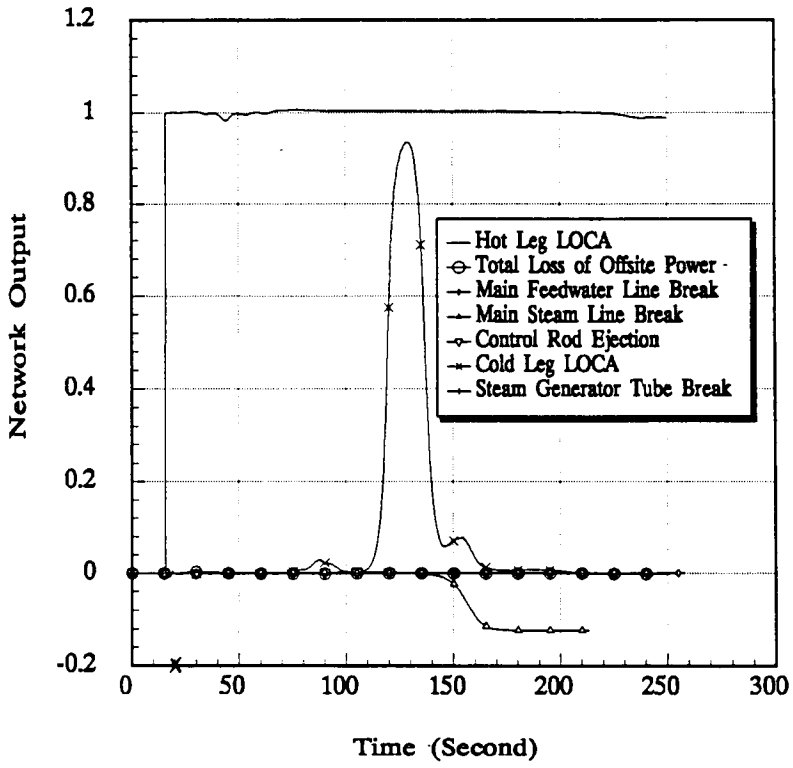

Figure 4.26: Response to All 7 Scenarios From Modular Network of Hot-leg LOCA (7 Weighted Inputs Selected for the Network)

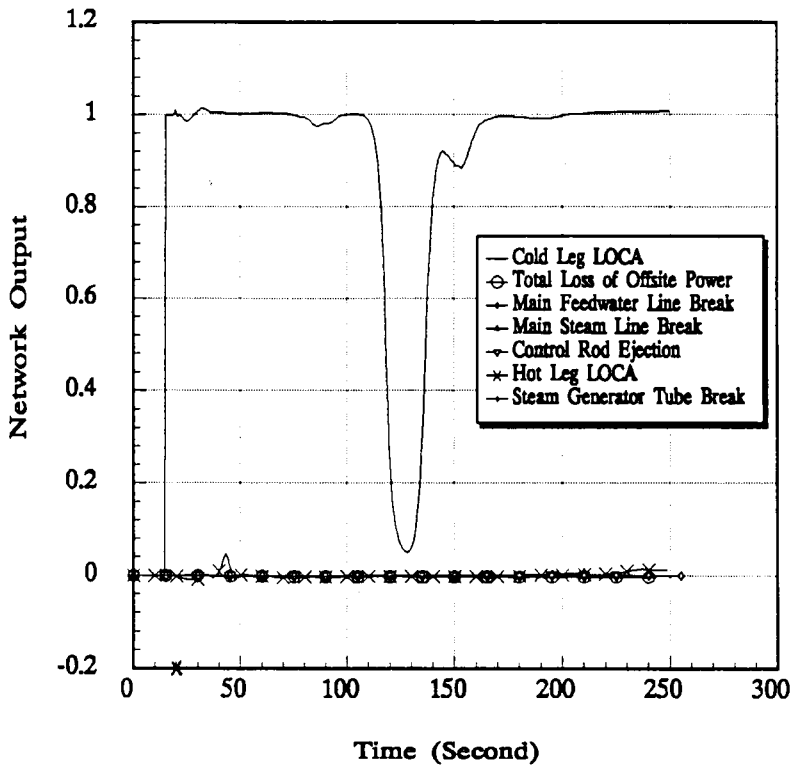

Figure 4.27: Response to All 7 Scenarios From Modular Network of Cold-leg LOCA (7 Weighted Inputs Selected for the Network)

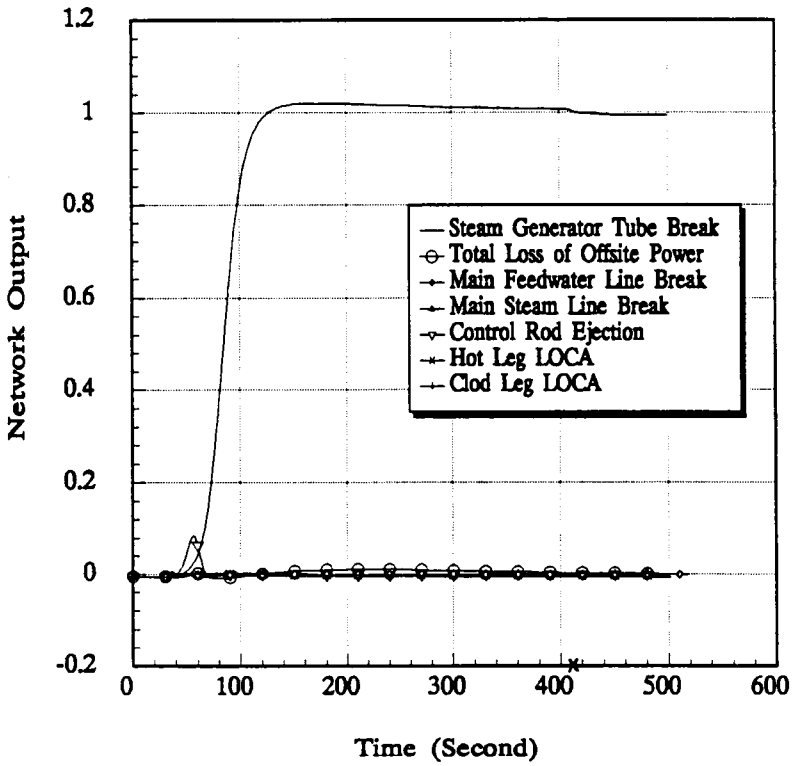

Figure 4.28: Response to All 7 Scenarios From Modular Network of Steam Generator Tube Break (5 Weighted Inputs Selected for the Network)

Break by crosschecking these two modular networks.

The weighing method, on the other hand, degraded the network performance of the modular networks of *Control Rod Ejection* by showing some false prediction signals with small amplitudes, but it will not be confused with other accident scenarios by crosschecking other modular networks.

For *Hot-leg LOCA* and *Cold-leg LOCA*, there were no significant improvement in the network performance because *Cold-leg LOCA* was still confused with *Hot-leg LOCA* in the middle of the transient.

Although the weighing method used in this chapter degraded the performance of the modular network of *Control Rod Ejection*, the overall performance of the modular networks was improved. Therefore, it can be used together with the sensitivity analysis to improve the network performance.

4.4.9 Extension of Modular Networks

A modular network can be built up to handle more than one accident scenario while still keeping its size small. The accident scenarios may be grouped together if they depend on the same or similar variables selected by the sensitivity analysis. *Hot-leg LOCA* and *Cold-leg LOCA*, for instance, have virtually the same selected variables, therefore, these two scenarios can be grouped together and monitored by a single modular network. The modular network set for this task has two output neurons instead of one in the output layer. Each one of the neurons responses to only one scenario. For example the outputs of (1, 0) or (0, 1) indicate that the *Hot-leg LOCA* or *Cold-leg LOCA* have occurred, respectively. The results of the modular network recall are given in Figures 4.29 and 4.30. It can be seen that the modular network gave quick and correct responses to all seven scenarios except in the middle range of the transient where the *Cold-leg LOCA* was mishandled as *Hot-leg LOCA*. The same confusion occurred previously for the diagnostics using

the motherhood network, even though 22 input variables were used. This confusion can only be resolved by adding more variables, which are currently not included in the raw data file, as the inputs of the network. The concept of building a small modular network to monitor more than one accident scenarios will be very useful in the applications to nuclear power plants where a large number of plant variables and potential accident scenarios need to be monitored. The sensitivity analysis can be used to guide the selection and group of plant variables for building up small and efficient modular networks.

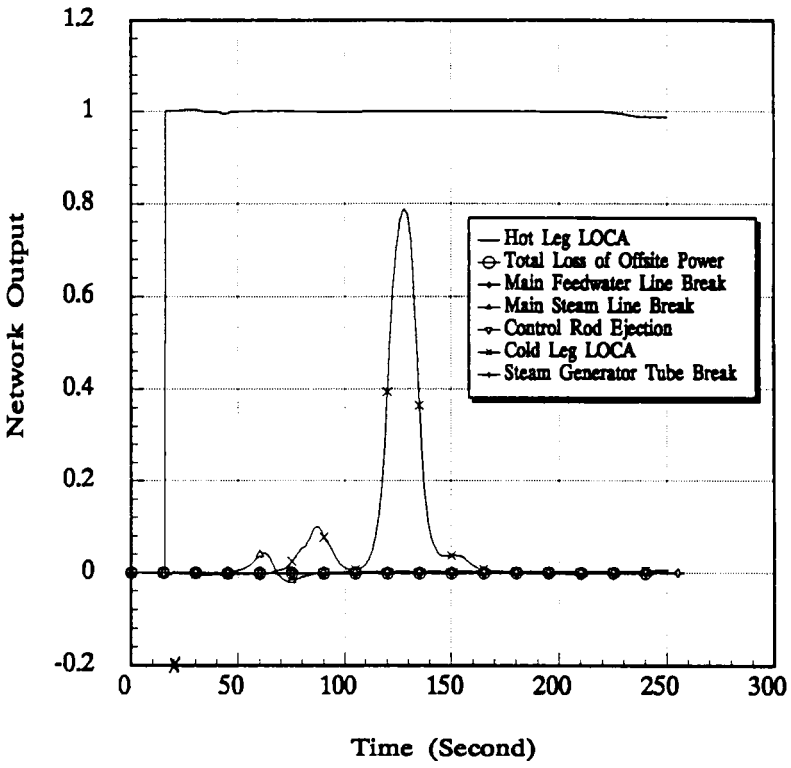

Figure 4.29: Response to All 7 Scenarios From Node 1 of Modular Network Trained for Both Hot-leg and Cold-leg LOCA (8 Input Variables Selected for the Network)

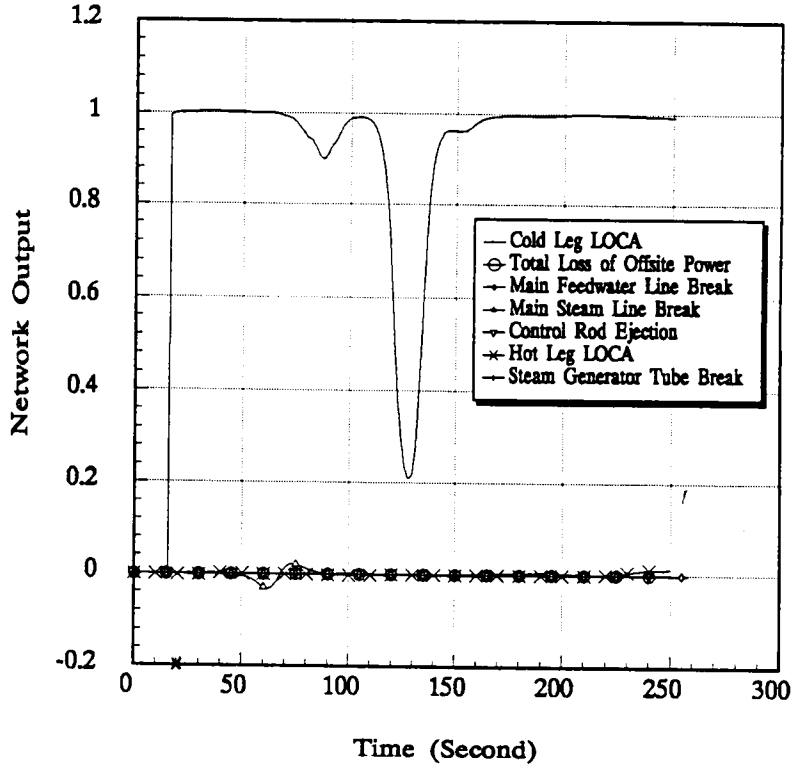

Figure 4.30: Response to All 7 Scenarios From Node 2 of Modular Network Trained for Both Hot-leg and Cold-leg LOCA (8 Input Variables Selected for the Network)

CHAPTER 5

SELECTION OF OPTIMAL INPUT VARIABLES FOR NEURAL NETWORKS BY USING GENETIC ALGORITHMS

5.1 Introduction

In the applications of neural networks to solve real world problems, the question of how to select input variables for the neural networks is often raised. In a nuclear power plant, for example, there are thousands of plant variables displayed in the control room for the operators to monitor the plant operation. The problem of selecting the appropriate variables from these thousands of variables for use in neural networks to monitor the plant operation is a major concern. We can not simply use all plant variables as the network inputs because it will result in difficulties in the network training and convergence. It would also slow down the network response even if the training was successful, because the network would be too large to give quick response. We want to select the fewest variables possible that contain necessary information for neural networks to determine the plant status. If these variables can be selected, then, we can build a small neural network which provides fast training and instant recall. The selection of input variables is often based on the person's experience and knowledge in the field where the neural networks are applied, or by trial and error iterations.

In Chapter 4 a sensitivity analysis was used to guide the selection of the input variables for individual modular network, and the results were excellent. However, problems arise where a large number of accident scenarios need to be monitored, which initially require a very large network be trained to respond to all accident scenarios before the sensitivity analysis can be applied to select variables. Again, we face the problem of training and convergence of large neural networks.

To overcome this difficulty, genetic algorithms will be used in this chapter to guide the selection of input variables to meet the criterion of fewer input variables, faster training, and higher recall accuracy.

5.2 Genetic Algorithms

5.2.1 Introduction to Genetic Algorithms

Genetic Algorithms are search algorithms based on the mechanics of natural selection and natural genetics. The principle behind genetic algorithms is essentially Darwinian natural selection. During their evolution, biological systems developed successful strategies of behavior adaption and synthesis to enhance the probability of survival and propagation. Environmental pressure requires that these strategies produce profound changes in the biological organisms. These changes are manifested in structural and functional specializations, informational organization, and internal knowledge representations [33].

John Holland is the founder of the field of genetic algorithms. With the publication of *"Adaptation in Neural and Artificial Systems"* [14], he integrated and elaborated two themes that had persistently recurred in his research: the ability of simple representations (bit strings) to encode complicated structures, and the power of simple transformations to improve such structures [15].

In genetic algorithms, specially structured bit strings are designed to represent a practical problem for optimal search. Individual bit strings are organized to form a population. Genetic algorithms guide the string population to propagate from generation to generation to improve the survival probability of the entire population. The genetic algorithms combine survival of the fittest among string structures with a structured, yet randomized information exchange to form the search algorithm. In every generation, a new set of artificial creatures (strings) is created using bits

and pieces of the fittest of the old. While randomized, genetic algorithms are not a simple random walk process. They efficiently exploit historical information to speculate on new search points with expected improved performance [16].

5.2.2 Robustness of the Genetic Algorithms

The central theme of research on genetic algorithms has been robustness, the balance between efficiency and efficacy necessary for survival in many different environments. Genetic algorithms are theoretically and empirically proven to provide robust search in complex search spaces [16]. The fundamental differences between the genetic algorithm and traditional optimization methods, such as calculus-based search and hill-climbing method, can be described in four ways [16]:

1. Genetic algorithms work with a coding of the parameter set, not the parameters themselves.

2. Genetic algorithms search from a population of points, not a single point.

3. Genetic algorithms use payoff (objective function) information, not derivatives or other auxiliary knowledge.

4. Genetic algorithms use probabilistic transition rules, not deterministic rules.

Taken together, these four differences contribute to a genetic algorithm's robustness and advantage over other more commonly used techniques. Having been established as a valid approach to problems requiring efficient and effective search, genetic algorithms are now finding more widespread applications in business, scientific, and engineering fields.

5.3 Genetic Algorithm: Basic Theory and Operation

5.3.1 Driving Force of the Genetic Algorithms

Genetic algorithms consist of a population of individual bit strings competing in a survival-of-the-fittest environment. The algorithm proceeds in steps called generations. During each generation, a new population of individuals is created from the old via application of genetic operators, and evaluated as solutions to a given problem (the environment). Due to selective pressure, the population adapts to the environment over succeeding generations, involving better solutions [16]. If the environment is a function, genetic algorithms can be used for function optimization. In this case, each individual in a population is a sample point in the function space. Classically, an individual in a genetic algorithm is represented as a bit string of some length n. Each individual thus represents one sample point in a space of size 2^n. The three basic genetic algorithm operators, namely, *reproduction*, *crossover*, and *mutation*, yield good results in many practical problems.

5.3.2 Simple Genetic Algorithm

The *reproduction* operator is an artificial version of natural selection, a Darwinian survival-of-the-fittest among string creatures. In the *reproduction* process, individual strings are copied according to their objective function values, f (biologist called this function the fitness function). Copying strings according to their fitness values means that candidates with the higher fitness values have a greater probability of contributing one or more offsprings in the next generation. The selection probability for individual i (the ith string in the population) may be defined as:

$$p_i = f_i / \sum_{i=1}^{N} f_i \tag{5.1}$$

where f_i is the fitness value (or objective function value) of *ith* individual in the population of size N.

We select the mating pool of the next generation according to the probability p_i. Once an individual has been selected for *reproduction*, it is then entered into the mating pool, a tentative new population, for further genetic operator action. If there is no overlapping between populations, meaning that the population size remains constant and a new generation entirely replaces the old (or parent) generation, the expected counter of *reproduction* of *ith* individual is:

$$n_i = N * p_i = N * f_i / \sum_{i=1}^{N} f_i = f_i / (\sum_{i=1}^{N} f_i / N) = f_i / \overline{f} \tag{5.2}$$

where $\overline{f}$ is the average fitness of the population. It gives us what we should expect: the best gets more copies, the average stays even, and the worst dies off.

After *reproduction*, simple *crossover* may proceed in two steps. First, members of the newly reproduced strings in the mating pool are mated at random. Second, each pair of strings undergoes *crossover* as follows: an integer position k along the string is selected between 1 and string length less one, $[1, l-1]$. Two new strings (offsprings) are created by swapping all characters between positions $k+1$ and l inclusively. For example, the two strings '10010110' and '11100101', split after the third bit as shown below in (a) and (b), would create two new strings of (c) and (d) after *crossover*.

Label	Old String	Label	New String
(a)	10010110	(c)	10000101
(b)	11100101	(d)	11110110

Mutation , in the simple genetic algorithms, is the occasional (with small probability) random alteration of the value of a string position. In the binary coding representation, this simply means changing a 1 to a 0 and vice versa. By itself, *mutation* is

a random walk through a string space. When used sparingly with *reproduction* and *crossover*, it is an insurance policy against premature loss of important notations. By randomly changing a bit in a string, it actually introduces new material into the knowledge base. With small probability of occurrence, *mutation* is considered as secondary mechanism of genetic algorithm adaption, or as a background operator.

The function of *crossover* is to generate rearrangements of coadapted groups of bits from high performance strings. It does not create any new "genetic material" in the knowledge base. The mechanics of *reproduction* and *crossover* are surprisingly simple, involving random number generator, string copies, and some partial string exchanges. Nonetheless, the combined emphasis of *reproduction* and the structured, though randomized, information exchange of *crossover* give genetic algorithms much of their power. How can such two simple operators result in anything useful, let alone a rapid and robust search mechanism? What is being processed by genetic algorithms and how do we know whether processing it (whatever it is) will lead to optimal results in a particular problem? Appendix A provides more detailed discussion of the theory of genetic algorithms and tries to answer the questions above.

5.4 Applications of Genetic Algorithms To Neural Networks

Genetic Algorithms have gained useful applications in many fields, such as in biology, computer science, engineering and operation research, hybrid techniques, image processing and pattern recognition, physical science, as well as social science. Glodberg [16] has reviewed many important applications of genetic algorithms, which covers the pioneer work in the 1950's to the most recent research.

Some researchers have tried to connect the genetic algorithms with neural networks in recent years. Whitely [37] used the genetic algorithms to guide his *back-propagation* based neural network in finding the "necessary" connections instead of

full connections to enhance learning speed. Whitely [38] tried to use the genetic algorithms to optimize small networks using his GENITOR II that used genetic algorithms to define the connectivity of neural nets [39]. He concluded that the GENITOR not only found smaller nets but also nets that learn much faster and much more consistently than fully connected nets. Spears [40] explored the paradigms for using both neural networks and genetic algorithms to heuristically solve boolean satisfiability(STA) problems and found that neural network paradigm might be better for smaller problems and genetic algorithms more effective on large ones. Koza [41] used genetic algorithms to guide search for the time-optimal "bang-bang" control strategy for the cart centering problem, a version of the broom balancing problem, by genetically breeding populations on control strategy programs (LISP functions) for control problems. Maricic [42] used his neural network designer – GENNET to find the most appropriate network architecture for solving a given problem. In GENNET, a genetic algorithm block is responsible for generating population architectures, which are used to create a set of stand-alone *backpropagation* networks. Interactions between genetic algorithms and neural networks generate the best network architecture. Garis [43] used the genetic algorithm to train modular neural networks by finding the proper weights for the full connections. The genetic algorithms are used to guide the design of neural control circuits, which combine the neural modules to form functional hierarchies. Muselli [44] proposed a combination method of genetic algorithms and simulated annealing to generate and choose the set of points in the network connection weight space to speed up reliability and convergence.

5.5 Using Genetic Algorithms to Guide Selection Variables

The criteria for selecting input variables are to reach the fewest possible inputs, the fastest training, and highest recall accuracy. These criteria can be built

in to a fitness function of the genetic algorithms. The fitness function of the genetic algorithms can be designed to perform different search tasks for optimization. The value of the fitness function is the quantity to guide the *reproduction* process in the genetic algorithms for creating the new generation. Usually, the fitness function is designed in the way that its values are all positive and, the higher the value of the fitness function, the better the performance of the individual bit string in the population. Higher value of the fitness function also means that the individual bit string gets the better chance to be selected for production of the next generation. By using these guidelines, the fitness function for the selection of input variables is defined as:

$$fitness = f(number\ of\ inputs\ selected,\ network\ training\ error) \quad (5.3)$$

The fewer the number of inputs selected, the higher the fitness value, and the smaller the network training error, the higher the fitness value. The training cycles are fixed for all networks; therefore, only the training error is used in the fitness function.

The detailed discussion of the fitness functions for the application to *TVA's Watts Bar* nuclear power plant will be given in later sections.

5.6 Fault Diagnoses With Genetic Algorithms and Neural Networks

5.6.1 Reasons for Selecting Important Variables

In fault diagnostics, different accident scenarios are related to different parameters which are important for modular networks to determine the plant status. If too many variables are used as network inputs, the training of networks may be very difficult to perform. If too few inputs are used, the information provided by the selected variables may not be enough for the network to determine the plant status. Table 5.1 gives the variables selected for diagnosing *Cold-leg LOCA* in the process of genetic algorithm search. It can be seen from the table that selection of different

Table 5.1: Network Performance For Diagnosing the *Cold Leg LOCA (TH2)* With Different Selection of Inputs but Using the Same Number of Training Cycles

Index	Selected Inputs	No. Inputs	Net Error (%)
1	0 4 6 7 8 9 10 11 12 14 15 19 20	13	41.748341
2	0 2 7 9 10 12 14 15 21	9	31.009270
3	0 1 2 5 6 7 8 10 12 15 16 17 21	13	24.457436
4	0 2 4 8 10 11 13 16	8	56.006687
5	0 2 4 8 10 11 13 14 15 16 18 19 21	13	0.994036
6	0 4 8 10 11 13 14 15 16 18 19 21	12	1.335877
7	1 5 6 7 8 10 12 13 14 19 20 21	12	9.034389
8	1 2 3 6 7 10 11 13 14 15 21	11	17.629047
9	1 5 6 8 10 11 13 14 15 16 18 19 20 21	14	71.486389
10	0 2 4 7 8 10 12 13 14 15 19 21	12	1.171601
11	0 1 4 5 9 10 11 14 17 19 20	11	44.673519
12	0 3 4 5 11 12 14 16 18 19 20 21	12	34.853710
13	4 5 7 10 12 14 15 21	8	32.177555
14	0 1 2 7 9 12 13 18 19 21	10	11.506926
15	0 1 2 7 9 10 11 14 16 18 19 20 21	13	33.671741
16	1 5 6 7 8 10 12 13 14 19 20 21	12	9.034389
17	2 4 5 7 8 10 11 15 16 20 21	11	27.799295
18	1 5 6 7 8 10 12 13 14 19 20 21	12	9.034389
19	4 5 7 12 13 19 21	7	1.818293
20	0 4 6 10 11 12 13 14 15 16 17 19 21	13	0.666125

inputs results in different training errors though the same number of training cycles is used. The training error does not depend on the number of inputs selected but on the selected inputs. For example, selection number 9 has 14 inputs and the network training error is 71.5% and may not converge through more training, while the selection number 19 has only 7 selected inputs, but the training error is 1.8%. Therefore, the proper selection of inputs can not only reduce the size of the network but also speed up the training process.

5.6.2 Sensitivity Analysis and Genetic Algorithm Search

The application of sensitivity analysis to guide the selection of input variables for the modular networks has been discussed in Chapter 4. In order to apply sensitivity analysis, a motherhood network, which is defined as the network to be trained to respond to all considered accident scenarios and to use all input variables, must be trained successfully. The sensitivity analysis can then be used to rank the input variables in the order of importance. The childhood or modular networks, which are defined as the individual networks that respond to only one accident scenario, can be established by selecting only the higher ranked (important) variables as their inputs. The training of modular networks is much easier because they have a small input layer due to fewer input variables selected through sensitivity study, a small hidden layer due to the small input layer and the diagnosis of only one assigned accident, and an output layer with only one neuron to respond to the assigned accident scenario.

The sensitivity analysis does provide the information about how to select the proper input variables for the modular networks to reach the goal of fast training and fast recall response. But the quality of information extracted from the sensitivity analysis depends on how good the neural model is, which in turn depends on how complete the training patterns are and how good the configuration of the network is. If the training data patterns can not provide sufficient information about the system, the motherhood network can not be trained to represent the internal model of the system, therefore, the sensitivity analysis may not provide the useful information to rank the input variables. Another difficulty of using the sensitivity analysis is that a large network, motherhood network, has to be trained before the sensitivity analysis can be applied. If there is a large number of accident scenarios to be monitored, the motherhood network will be very difficult or even impossible to train because of a large number of input variables and training patterns involved.

Table 5.2: Accident Scenarios Simulated on Watts Bar Simulator

1	Total loss of offsite power (ED1)
2	Main feedwater line break (F23)
3	Main steam line break (MS1)
4	Control rod ejection (RD6)
5	Hot leg loss of coolant accident (TH1)
6	Cold leg loss of coolant accident (TH2)
7	Steam generator tube leak (TH5)

If genetic algorithms are used to guide the selection of the input variables to build the modular networks, the motherhood network is no longer needed because the information about the importance of the variables will be explored by the operations of genetic algorithms instead of using sensitivity analysis. Therefore, the difficulties in training a large network, motherhood network, can be avoided. The robustness of the genetic algorithms makes it possible to use the network training and configuration information, i.e. a training error and the number of inputs selected, to guide the search for a near optimal combination of the inputs for an individual modular network.

5.6.3 Network Training Data

The data from *TVA's Watts Bar* nuclear power plant simulator will be used for the application to genetic algorithms and neural networks. The data contain seven accident scenarios, which are listed in Table 5.2.

The detailed discussion of the accident scenarios are given in Appendix B. The data used to train the networks are the same as those used to train all modular networks in Chapter 4, which are formed by the centroids of clusters of all raw data. The detailed discussion of the data is in Chapter 4.

5.6.4 Link Between Genetic Algorithms and Neural Networks

For a given number of training cycles, the network will have different performance, measured by the recall error of the network, for different set of input variables. The network performance and the number of variables selected as the network inputs can be used to evaluate the fitness function whose value will be used to guide the selection process in genetic algorithms. The procedure of the process is outlined below:

1. Create an initial population of individuals, or bit strings, each one of which can be interpreted as a group of selected inputs,

2. Set up and train the population of networks with the inputs selected by step 1,

3. Transfer the training error and the number of inputs selected to the fitness function to evaluate the fitness for each network,

4. Perform the operation of genetic algorithms to create a new population of the individuals,

5. Repeat steps 2 - 4 till the defined generation number is reached.

The mechanism of the genetic algorithms guarantees the convergence of the search to find a near optimal group of inputs that give superior performance with highest fitness, which in turn means the selection of fewest possible inputs to reach the smallest training error.

5.7 Fitness Functions

In the selection process of genetic algorithms, an individual bit string may or may not be selected as a parent for the next generation depending on the overall

Table 5.3: Best Inputs Selected by GA, Fitness, and Net Error

Scenario	Inputs Selected	Gen.	Inputs	Error(%)
ED1	1 2 3 11 13 14 18 21	8	9	0.76
MS1	2 7 8 11 13 14 16 20	8	8	0.79
F23	3 4 7 12 13 14 15 18	10	8	1.5
RD6	0 2 3 6 7 11 13 16 18 21	7	10	13.3
TH1	0 2 7 12 15 16 17 19 20 21	9	10	1.5
TH2	8 11 19 20	9	4	81.0
TH5	1 2 3 5 12 14 16 21	10	8	1.1

performance of an individual bit string. The fitness function may have different forms for different optimal search tasks. In this research, the fitness function needs to be defined to guide the search for the best combination of inputs for different modular networks. For the application of fault diagnostics using *TVA's Watts Bar* nuclear power station simulator data, the fitness function should be designed to select fewer variables while makes the training error smaller for a fixed number of training cycles, and to penalize the selection which chooses more than necessary inputs and/or makes training error larger. Three different fitness functions were used in the application.

5.7.1 Fitness Function 1

The first form of the fitness function has the following form:

$$Fitness = \frac{c_1 \cdot \#\ of\ \ Inputs\ \ Before\ \ Selection}{\#\ of\ \ Inputs\ \ Selected} + \frac{c_2}{Network\ \ Training\ \ error} \tag{5.4}$$

The fitness function, Equation 5.4, was used in the genetic algorithms to guide the search for the best fitted inputs. The results of the best selections are summarized in Table 5.3 where "Gen." is the abbreviation of "Generation".

It can be seen from Table 5.3 that for the scenario of *Cold-leg LOCA* the

Table 5.4: Selections for RD6 by Genetic Algorithms

Inputs Selected	Gen.	Total	Error(%)	Fitness
0 2 3 6 7 11 13 16 18 21	7	10	13.3	2.35
2 6 18 19	10	4	52.6	5.54

genetic algorithms picked 4 input variables as the network inputs but the network training error was vary large. If we examine the history of the genetic algorithm search, we can find that the same situation exists for the scenario of control rod ejection (**RD6**) as shown in Table 5.4.

The genetic algorithm search actually failed for these two cases because the network training error was too large to be accepted. The failure was caused by the definition of the fitness function, Equation 5.4. The value of the fitness in Equation 5.4 is the sum of two terms. If the search selects a small number of inputs the first term in the equation will have a large value. In the case of *RD6* and *TH2*, the selected inputs are 4 and the network training errors are very large. The overall fitness value are dominated by the first term in the equation, which gives a large value of fitness. The genetic algorithm search is guided by the fitness value solely, therefore, the designed fitness function, in some instances, will lead the search in the direction of finding the fewest inputs regardless of the network performance, such as the training error of the network.

5.7.2 Fitness Function 2

The first form of proposed fitness function was good for some cases but it led the search to the wrong direction for some other cases. The second proposed fitness function should be designed to correct the errors made by the first form. The fitness function has the following format:

Table 5.5: Term 1 of Fitness Value of Equation 5.5 Versus Number of Inputs

Inputs	4	5	6	7	8	9	10	11	12
Fitness	0.999	0.998	0.987	0.957	0.901	0.824	0.731	0.632	0.533

$$fitness = (1 - e^{-(x-1.0)^{1.5}}) \cdot e^{-0.01y^{1.5}} \tag{5.5}$$

where:

$$x = \frac{Total\ \ number\ \ of\ \ inputs\ \ before\ \ selection}{Total\ \ number\ \ of\ \ inputs\ \ after\ \ selection}$$

$$y = Network\ \ training\ \ error(\%)$$

It can be seen that the new format of the fitness function is the product of the two exponential terms. The first term reflects the awarding and penalizing of total selected inputs. Its value is exponentially decreasing with increasing number of inputs selected.

The second term reflects the awarding to a small training error and penalizing to large ones. The coefficients in the equation are determined by the consideration of the number of minimum expected inputs and the training error. The minimum number of inputs is the author's best guess. If the number of selected inputs is less than the minimum number, the information provided by the selected inputs is not enough to determine the power plant status at one time instant. The minimum expected error is the error to be considered accurate enough to determine the power plant status. Tables 5.5 and 5.6 show the values of term 1 and term 2 in equation 5.5 versus the number of inputs and the errors.

Table 5.5 shows that when the number of selected inputs is less than 7 there is no significant contribution to the fitness value, which prevents the search from choosing too few inputs to gather enough information to determine the plant status.

Table 5.6: Term 2 of Fitness Value of Equation 5.5 Versus Training Error

Error(%)	0.5	1	2	3	4	5	6	7	8	9	10	15	20
Fitness	.99	.99	.97	.95	.92	.89	.86	.83	.80	.76	.73	.56	.41

Table 5.7: Best Selections for TH2 With Fitness Function 2

Gen.	Selected Inputs	Total	Error (%)	Fitness
6	2 4 5 9 10 11 13 14 15 16 19 21	12	1.69	0.477
7	2 4 5 9 10 11 13 14 15 16 19 21	12	1.98	0.463
8	2 4 5 9 10 11 13 14 15 16 19 21	12	2.49	0.437
9	2 4 5 9 10 11 13 14 15 16 19 21	12	2.80	0.420
10	2 4 5 9 10 11 13 14 15 16 19 21	12	1.80	0.471

Table 5.6 shows that the network training error can be continuously reduced to increase the fitness, but the error less than 3% or 4% has no significant contribution to the fitness value, which means that the network prediction error less than 3% is small enough to determine the plant status and the search may concentrate on reducing number of inputs selected in order to increase the fitness value. The overall fitness value is determined as the product of the two terms. The fitness is high only if both terms have high values, therefore, the search will be guided in the direction of reducing both the number of inputs selected and the network training error.

The new fitness function was used to guide the search for *Cold-leg LOCA* again, and the results are listed in Table 5.7.

The results in Table 5.7 show that the new fitness function did improve the search and select the best fitted inputs with the training error about 2%. The variation in error was caused by the randomly initialization of the network before training. The results indicate that the best fitted selection was formed from generation 6 and kept through generation 7 to 10. Why was there no progress after the 6th generation to improve the performance of the search or to increase the values of the

Table 5.8: Initial Generation of GA Search for TH2 With Fitness Function 2

String No.	Bit String	Fitness	Paired Mates
(1)	11100111101001000010 01	0.000017	(9 10)
(2)	00000101011010110000 01	0.000000	(9 10)
(3)	00110100000011111010 11	0.000000	(15 10)
(4)	11110001000011110011 11	0.000000	(15 10)
(5)	00011001001101110110 10	0.000000	(15 10)
(6)	00100001100101101010 00	0.000000	(15 10)
(7)	01000111001011100101 00	0.000000	(10 10)
(8)	10000110011110000001 11	0.000000	(10 10)
(9)	01011001101010011100 01	0.002931	(10 10)
(10)	00101100101101111011 01	0.140920	(10 10)
(11)	10001011111111110000 10	0.000000	(10 10)
(12)	11100010001101000010 00	0.000000	(10 10)
(13)	10000110000111101011 11	0.000000	(10 15)
(14)	10101001101100011000 11	0.000062	(10 15)
(15)	10011100111100100101 10	0.053312	(10 15)
(16)	11101011010110011001 00	0.000000	(10 15)
(17)	01011100011100000111 00	0.000000	(15 10)
(18)	11011011001101110011 00	0.005805	(15 10)
(19)	11101100100110101101 01	0.000006	(10 10)
(20)	10001101000010101100 00	0.000839	(10 10)

best fitness? To answer this question it is necessary to examine the history of the genetic algorithm search. Table 5.8 shows the initial generation of bit strings representing selections of inputs, their fitness values, and string pairs chosen as parents to produce the next generation.

It can be seen from Table 5.8 that string number 10 has fitness value of 0.14 and the average fitness of the entire population is 0.011. The selection frequency of a string in the genetic algorithms is determined by the ratio of a fitness of individual string over the fitness of population average. Therefore, string number 10 will be selected as the parent for the next generation with frequency of $0.14/0.011 = 12.7 \simeq 13$. The actual frequency of selection of string number 10 was 14. The total number of strings in a population was 20, therefore, there were 14 strings out of 20 in total

to be copied directly from string number 10 to the next generation. In the genetic algorithm search strings like string number 10 in this case are called superstrings because they have much higher fitness values above the average. If the superstrings take over control of the entire population in the early stage of the propagation, the population will lose the diversity and the search will be trapped in a local minimum and never reach the global minimum. This is easy to understand that a superstring will be selected as a mate to itself because of its much higher fitness, which can be seen in Table 5.8, and the *crossover* operator will lose its power because no *crossover* operation to a pair of identical strings can produce any change in the strings. Therefore, the only remaining power of the genetic algorithms is *mutation.* The genetic algorithm search can not perform well with *mutation* only because of the very small probability of occurrence of *mutation.* This explains the results shown in Table 5.8. Therefore, the fitness function will need to be modified to prevent superstrings in the early stage of the search and to keep the diversity of population during the propagation.

5.7.3 Fitness Function 3

To prevent superstring from taking over control of the entire population and to keep the diversity of the population, the fitness function should be scaled down in the early stages to shrink the difference in fitness and be scaled up in the latter stage of the search to strengthen the difference in fitness to approach the global minimum. Therefore, the generation index, which indicates on which generation the genetic algorithms are currently working, should be included in the fitness function to guide the search. The third form of the fitness function is given below:

$$fitness = (1.0 - e^{-(x-1)^{0.15(z+1)}}) \cdot e^{-0.01y^{0.7(z+1)^{1/3}}} \tag{5.6}$$

where x and y are the ratio of total inputs over selected inputs and the network training error, respectively, given in equation 5.5. z is the generation index with values of $0, 1, \cdots, n$. The coefficients used in the function are determined in the same way as in the fitness function 2. Figures 5.1 and 5.2 show the fitness versus the number of inputs selected and the network training error for 10 generation indices.

It can be seen that the slopes of fitness versus both the number of inputs selected and the network training error are getting deeper and deeper with increasing generation indices. In the initial stages or with small generation indices the difference in fitness values of individual strings is small though the principle of rewarding and penalizing for the selection remains the same. The better fitted strings still get better chance to be selected as parents for the next generation but the frequency of selection is limited because of the limited difference in fitness values among the strings in the population. The population diversity will be maintained in the early search stages to avoid trapping in local minima (maxima). In the latter stage or with larger generation indices the small difference in the number of inputs selected and in the training error will make large difference in fitness values to assure the search of converging to a global minimum (maximum). Table 5.9 gives the initial generation of GA search for *Cold-leg LOCA* as the comparison with Table 5.8.

The maximum fitness, the population average, and the minimum fitness are 0.606211, 0.538024, and 0.470195 for Table 5.9, respectively, compared to 0.140920, 0.010195, and 0.000000 for Table 5.8. It can be seen that there no longer exists any superstring in the initial population, and the initial population is highly diversified. Though string number 10 still had the maximum fitness, it was selected only once for the next generation. Table 5.10 gives the generation statistics of GA search for *TH2* to show the improvement in GA search guided by the new form of fitness function, equation 5.6.

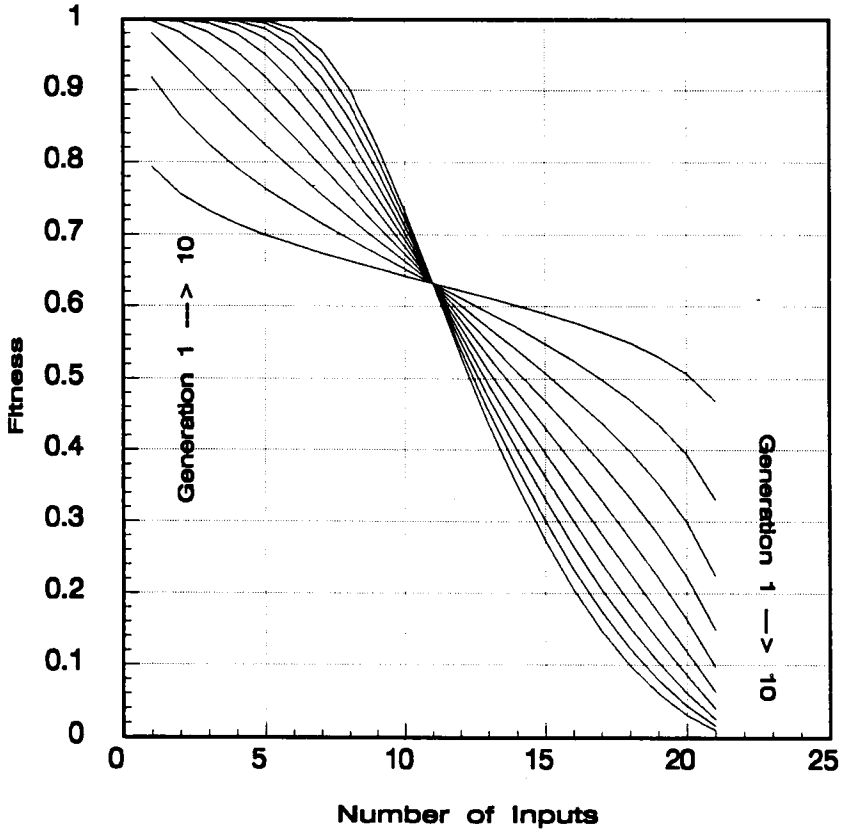

Figure 5.1: Fitness Versus Number of Inputs for Ten Generation Indices

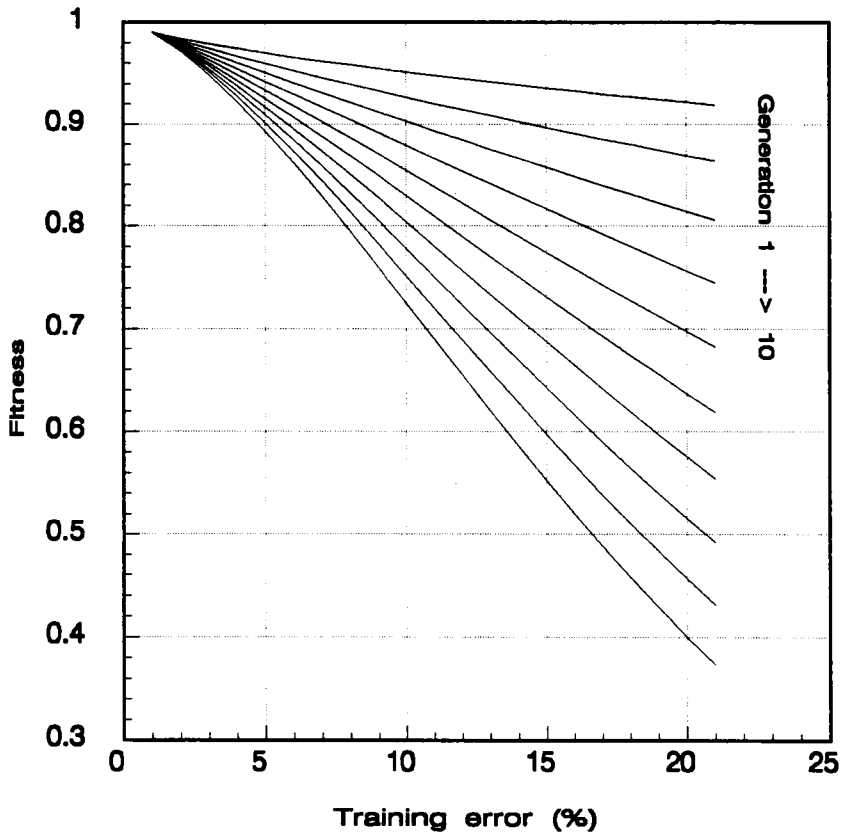

Figure 5.2: Fitness Versus Training Error for Ten Generation Indices

Table 5.9: Initial Generation of GA Search for TH2 With Fitness Function 3

String No.	Bit String	Fitness	Paired Mates
(1)	11100111100100001001	0.587846	(3 8)
(2)	00000101011010110000001	0.575030	(3 8)
(3)	00110100000011111101011	0.482089	(6 13)
(4)	11110001000011111001111	0.497037	(6 13)
(5)	00011001001101110111010	0.494399	(20 12)
(6)	00100001100101101010000	0.498036	(20 12)
(7)	01000111001011100101000	0.540717	(5 17)
(8)	10000110011110000000111	0.540112	(5 17)
(9)	01011001101010011100001	0.591227	(15 11)
(10)	00101100101101111011101	0.606211	(15 11)
(11)	10001011111111111000010	0.528007	(18 9)
(12)	11100010001101000010000	0.580011	(18 9)
(13)	10000110000111101011111	0.470195	(7 1)
(14)	10101001101100011000011	0.556465	(7 1)
(15)	10011100111100100101110	0.569233	(10 14)
(16)	11101011010110011001000	0.474706	(10 14)
(17)	01011100011100000011100	0.501510	(4 2)
(18)	11011011001101110011000	0.525357	(4 2)
(19)	11101100100110101101010	0.548741	(16 19)
(20)	10001101000010101100000	0.593558	(16 19)

Table 5.10: Statistics of GA Search for TH2 With Fitness Function 3

Generation	Max. Fit	Avg. Fit	Min. Fit	Max - MIn
0	0.606	0.538	0.471	0.135
1	0.586	0.448	0.276	0.310
2	0.566	0.378	0.158	0.408
3	0.580	0.374	0.112	0.468
4	0.747	0.395	0.018	0.726
5	0.770	0.312	0.002	0.768
6	0.872	0.353	0.020	0.852
7	0.842	0.426	0.020	0.822
8	0.917	0.479	0.000	0.917
9	0.923	0.550	0.024	0.899
10	0.946	0.636	0.000	0.946

Table 5.10 also shows that the difference between the maximum fitness and the minimum fitness in a generation was quickly increasing along with population propagation. It further shows that the minimum fitness was decreasing exponentially during the process of the population propagation. Therefore, the best fitted strings will get a better and better chance to be selected as parents for the next generation during the propagation, and the worst fitted strings will have a little and little probability of being selected and will die out during the propagation. The fitness of the population average experienced a decrease, and then, increase, which means that the search directions were very diversified in the early stage of the search, i.e. generation 1 through 3, because the maximum fitness did not increase. Then, after a short period of wondering, the genetic algorithms quickly found the better directions to search represented by increasing in maximum fitness, decreasing in minimum fitness, and increasing in the population average fitness. The entire population were converging to the global minima (maxima). These steps are what the good designed genetic algorithm search should experience. The results listed in Table 5.10 show that the fitness function 3, or Equation 5.6, does provide much better guidance for the genetic algorithm search with maximum fitness as high as 0.946. The fitness function 2, or Equation 5.5, on the other hand, only reached 0.47. The best selections for each generation in GA search for *TH2* are given in Table 5.11.

Table 5.11 shows that the best selection was actually found at generation 6 with 7 selected inputs and training error of 1.812%. The best fitted building block found during search up to generation 6 was (7 12 13 19 21) and was kept throughout the entire search followed. This best fitted building block was combined with other three building block 2, (0,4,5) in generation 7, (4,5) in generations 8 and 10, and (0,2) in generation 9, to form the best selections. It can be seen that the best choice was actually found in the 6th generation, and lost partially in the 7th generation and recovered in the 8th generation, and then, lost partially again in the 9th generation

Table 5.11: Best GA Selection for TH2 With Fitness Function 3

Generation	Best Selection	No. Inputs	Error(%)
0	2 4 5 8 10 11 13 14 15 16 18 19 21	13	0.893
1	0 2 4 8 10 11 13 14 15 16 18 19 21	13	0.994
2	0 2 4 8 10 11 13 14 15 16 18 19 21	13	0.994
3	1 2 3 4 6 7 10 11 13 14 15 21	12	1.921
4	4 5 7 12 13 18 19 21	8	3.479
5	4 5 7 12 13 18 19 21	8	3.479
6	4 5 7 12 13 19 21	7	1.812
7	0 4 5 7 12 13 19 21	8	1.597
8	4 5 7 12 13 19 21	7	1.812
9	0 2 7 12 13 19 21	7	2.317
10	4 5 7 12 13 19 21	7	1.812

and recovered again in the 10th generation. It illustrates the inherent stability and convergence of genetic algorithm search. Interestingly, the trade-off between the number of inputs selected and the network training error is not always true. The best selection in generations 4 and 5 was formed by adding an additional input, number 18, into the global optimal selection, (4 5 7 12 13 19 21), but the resultant training error was actually higher instead of lower as expected in general. Therefore, the best combination of inputs may be extracted from the given variables to build smaller but higher qualified network than that by using all input variables.

5.8 Best Selections for All Scenarios

The fitness function 3, or equation 5.6, were used to guide the search for the best selections of input variables for all 7 monitored accident scenarios. The network structure was fixed with 4 neurons in the hidden layer and 1 neuron in the output layer. The neurons in the input layer are determined by the number of inputs selected by the genetic algorithms. The network used is the lateral feedback neural network discussed in Chapter 2. But any supervised networks can be used together

Table 5.12: Best Selection of Inputs for All Seven Scenarios

Scenario	Best Selection	No. Inputs	Error(%)	Generation
ED1	1 3 4 5 12 13 18	7	0.90	10
MS1	4 6 8 10 11 12 15 16 19	9	2.42	6 to 10
F23	3 5 8 11 13 20	6	1.79	10
RD6	0 3 4 7 10 13 16 17	8	4.60	8 to 10
TH1	4 5 9 10 11 14 21	7	0.82	9
TH2	4 5 7 12 13 19 21	7	1.82	6, 8, 10
TH5	3 4 7 11 17	5	2.02	9 to 10

with genetic algorithms for reaching the best performance. The training data file is the same as that used in Chapter 4 for motherhood network training, sensitivity analysis, and modular network training. The data file is formed by 138 cluster centroids obtained by applying *self-organization* to all raw data files. Table 4.6 gives the information about the clusters formed for each raw data file. The computer code for this application contains both genetic algorithms and lateral feedback neural network, and requires three task files to operate. The first file is to supply names for output files, GA task file, and network task file. The second file is the GA task file which provides data needed to run the genetic algorithms. The third file is the network task file which supplies data to define network structure, learning rates, training cycles, names of training data file and output file, etc. Appendix H gives sample task files. Appendix I gives the code for genetic algorithms and associated lateral feedback network. The best selections for all 7 scenarios are given in Table 5.12.

5.9 Comparison of GA Search and Sensitivity Analysis

The sensitivity analysis (SA) was used to guide the selection of inputs for modular networks in Chapter 4. To determine which method, sensitivity analysis or GA search, can provide a better solution in choosing proper inputs , the same

Table 5.13: Selection of Inputs From GA and SA

Scenario	Best Selection		Identical
	Genetic Algorithms	Sensitivity Analysis	Selection
ED1	1 3 4 5 12 13 18	3 1 21 14 9 13	1 3 13
MS1	4 6 8 10 11 12 15 16 19	6 7 12 9 13 8 1 14 21	6 8 12
F23	3 5 8 11 13 20	13 14 21 11 9 3	3 11 13
RD6	0 3 4 7 10 13 16 17	13 21 3 14 1 15 4 9	3 4 13
TH1	4 5 9 10 11 14 21	21 4 13 11 17 7 9	4 9 11 21
TH2	4 5 7 12 13 19 21	13 21 4 11 14 17 9	4 13 21
TH5	3 4 7 11 17	13 21 14 3 9	3

number of inputs as those selected by GA search were chosen according to the order of importance ranked by the sensitivity analysis for all accident scenarios. The inputs chosen by the sensitivity analysis for each scenario are given in Table 5.13, in the order of importance, and the selections from GA search are also listed in the same table for comparison. It can be seen from Table 5.13 that all of the most important variables ranked by sensitivity analysis for each scenario were selected by GA search except for the case of *TH5*. For all scenarios, except *TH5*, there were 3 or 4 of the same variables chosen for each case by both GA search and sensitivity analysis as the selected inputs for the corresponding modular networks. This indicates that the sensitivity analysis and GA search have a good degree of agreement in exploring the same information, selecting important variables, though through different ways. To further compare the information explored by GA search and by the sensitivity analysis, seven modular networks were set up with their inputs selected by the sensitivity analysis as given in Table 5.13, and trained with the identical network structure, learning algorithm, learning rate, random seed, training cycles, and training data, as those used for GA search. The training results together with GA search results are given in Table 5.14.

Table 5.14 shows that the network training errors are very close to each

Table 5.14: Comparison of Network Training Errors for All Seven Scenarios

Scenario	No. Inputs	Error_GA(%)	Error_SA(%)
ED1	7	0.90	1.45
MS1	9	2.42	5.81
F23	6	1.79	1.87
RD6	8	4.60	3.62
TH1	7	0.82	1.74
TH2	7	1.82	1.54
TH5	5	2.02	1.13
Average Error		2.05	2.45

other by using inputs selected by either GA search or by the sensitivity analysis. The GA search has better overall performance, but the difference is small.

Figure 5.3 to Figure 5.9 show the performance of all seven modular networks with inputs selected by genetic algorithm search. These figures can be compared directly with Figure 4.8 to Figure 4.14, which show the performance of the modular networks with inputs selected by using sensitivity analysis. It can be seen that the performance of modular networks of *Total Loss of Offsite Power, Main Feedwater Line Break, Cold-leg LOCA*, and *Steam Generator Tube Break* was improved by using the inputs selected by genetic algorithms. For the modular network of *Cold-leg LOCA*, though there was still a big performance loss in the middle range of the transient it was no longer confused with *Hot-leg LOCA* in the entire transient process. For the modular networks of *Main Steam Line Break, Control Rod Ejection*, and *Hot-leg LOCA*, the performance was about the same for using inputs selected by both sensitivity analysis and genetic algorithms.

5.10 Conclusion

In this chapter, the genetic algorithms were investigated as a special search method for selecting best fitted inputs to neural networks. The performance of the

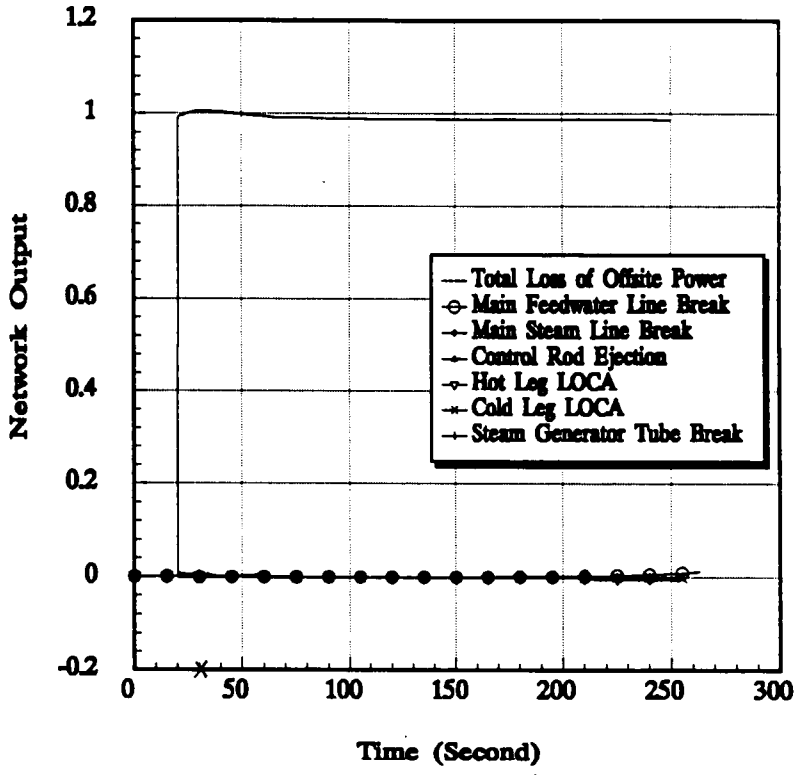

Figure 5.3: Response to All 7 Scenarios From Modular Network of Total Loss of Offsite Power (7 Variables Selected by GA Search for the Network)

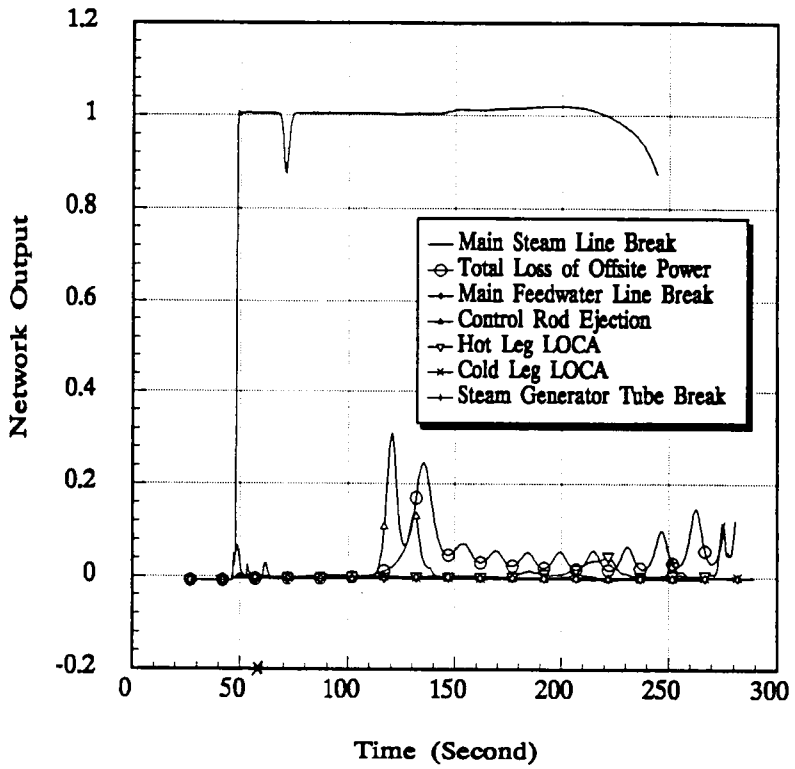

Figure 5.4: Response to All 7 Scenarios From Modular Network of Main Steam Line Break (9 Variables Selected by GA Search for the Network)

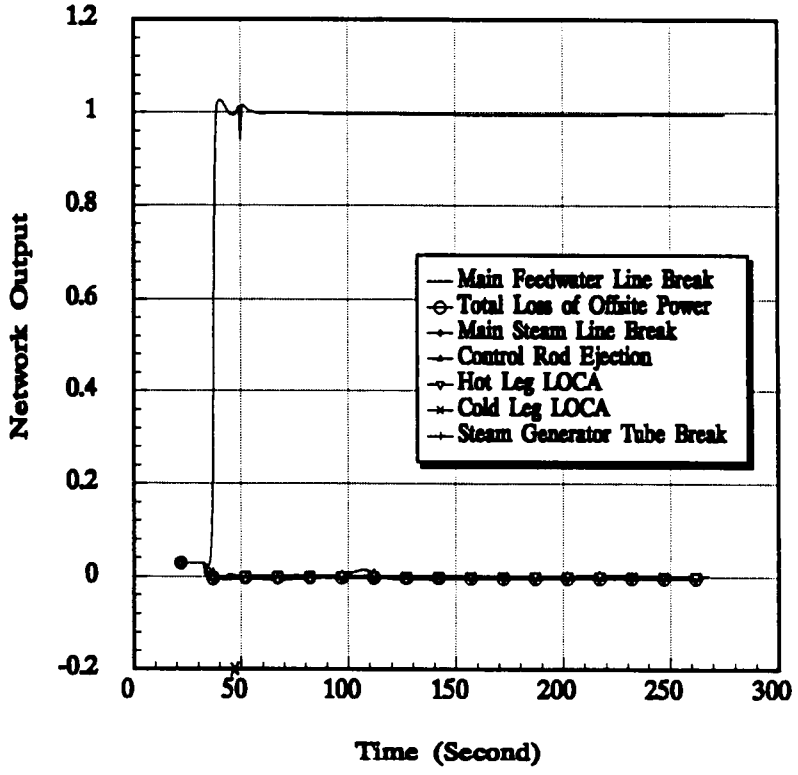

Figure 5.5: Response to All 7 Scenarios From Modular Network of Main Feedwater Line Break (6 Variables Selected by GA Search for the Network)

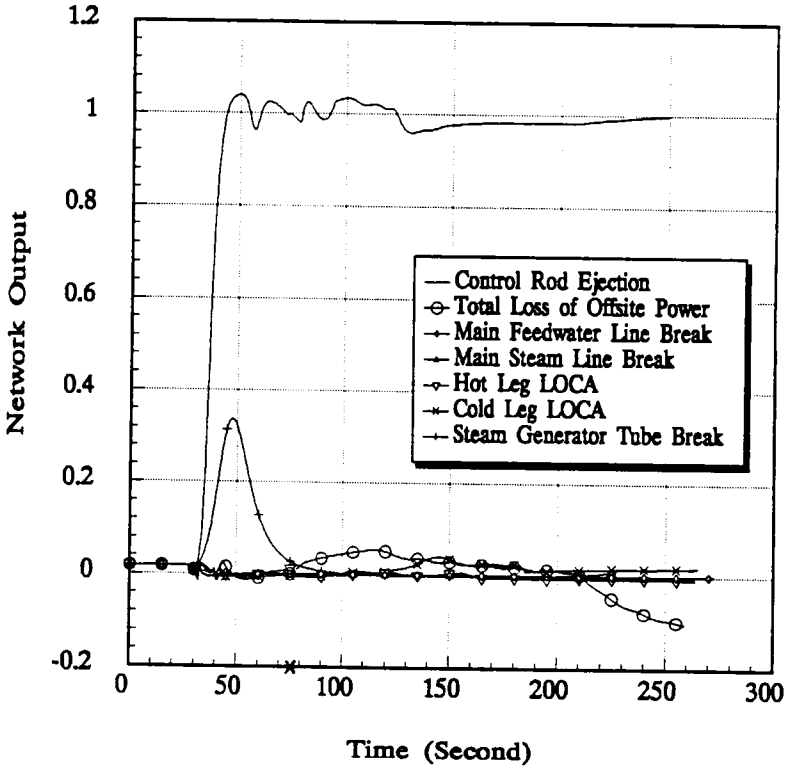

Figure 5.6: Response to All 7 Scenarios From Modular Network of Control Rod Ejection (8 Variables Selected by GA Search for the Network)

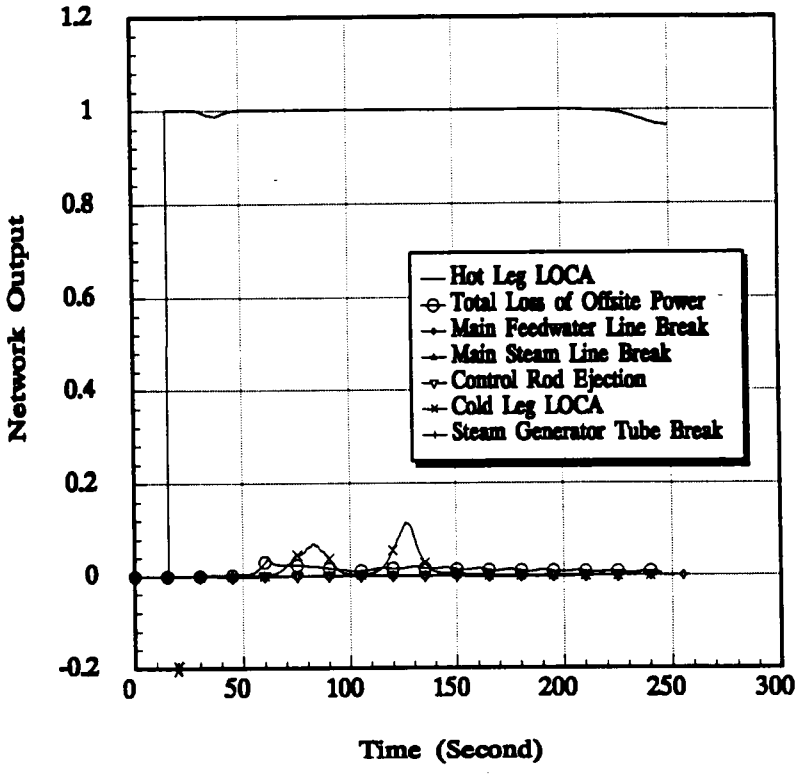

Figure 5.7: Response to All 7 Scenarios From Modular Network of Hot-leg LOCA (7 Variables Selected by GA Search for the Network)

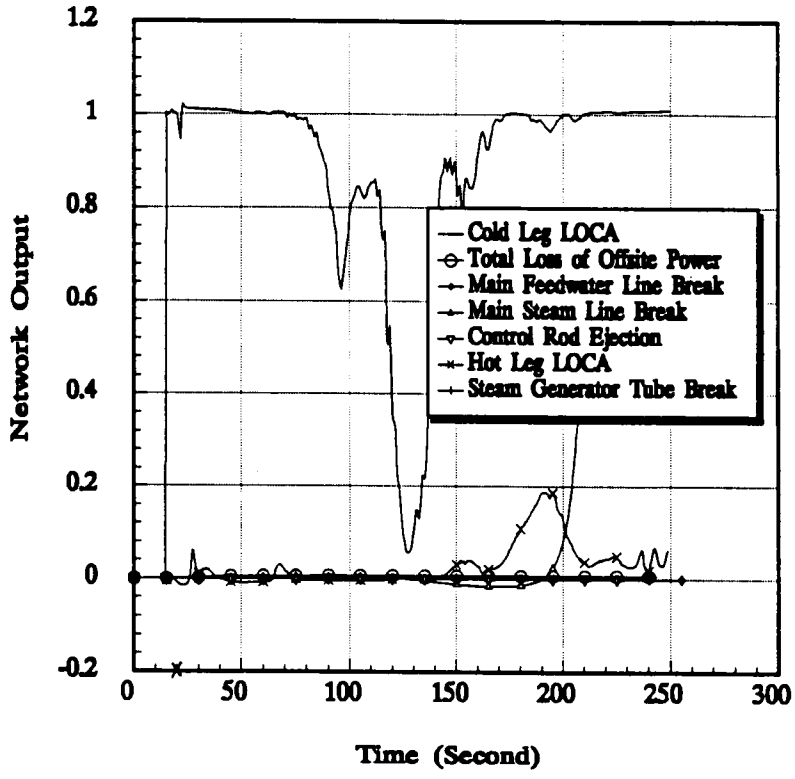

Figure 5.8: Response to All 7 Scenarios From Modular Network of Cold-leg LOCA (7 Variables Selected by GA Search for the Network)

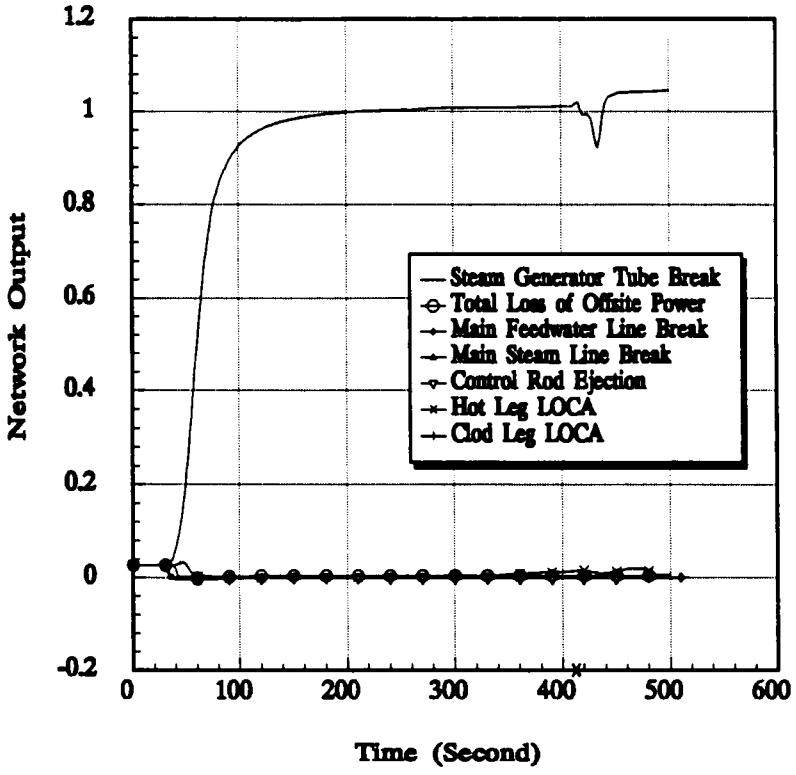

Figure 5.9: Response to All 7 Scenarios From Modular Network of Steam Generator Tube Break (5 Variables Selected by GA Search for the Network)

genetic algorithm search depends strongly on the design of the fitness function for a given task. Three different types of fitness functions were used in the GA search, and two of them have difficulties in guiding the search in some situations. The third form of the fitness function, Equation 5.6, which is a function of number of inputs selected, network training error, and the generation indices, gave the desired performance in guiding the GA search.

The application to the task of fault diagnoses for *Watts Bar* nuclear power plant shows that the genetic algorithms can be used to search for input variables for neural networks to reach higher performance in different diagnosing tasks. The input variables selected by GA search make the modular networks small in size, fast in training, and accurate in recall, which are the criteria in GA search.

The comparison between the GA search and the sensitivity analysis for selecting important variables shows a good degree of agreement. The training errors of the modular networks set up by using the same number of inputs selected by GA search and sensitivity analysis, respectively, are both small and close to each other, though the GA search seems to perform better globally. The comparison of network performance shows that the genetic algorithm search for network inputs can provide better results.

Both the sensitivity analysis and genetic algorithms can be used for choosing proper inputs for neural networks. The advantage of using the sensitivity analysis is that it can draw the conclusion through one cycle; therefore, it is quick and inexpensive. The disadvantage of using sensitivity analysis is that a motherhood network has to be trained successfully in advance before performing the sensitivity analysis. The information extracted from the sensitivity analysis strongly depends on how successfully the motherhood network is trained. Therefore, special effort must be made in training. If the training data base is large and a lot of variables and situations need to be considered, the motherhood network may fail to train

because of its large size. The genetic algorithm search, on the other hand, does not require a motherhood network to be trained first. It directly works with the modular networks and finds the best fitted input variables for them. The disadvantage of using GA search is that it is expensive and takes a lot of computation to find the solution. Therefore, if the motherhood network can be trained easily, the sensitivity analysis is preferred. Otherwise, the genetic algorithms should be used.

CHAPTER 6

CONCLUDING REMARKS AND RECOMMENDATIONS FOR FUTURE RESEARCH

6.1 Concluding Remarks

6.1.1 Lateral Feedback Network

The *lateral feedback* neural network has been introduced in this dissertation, which is a modification of the widely used *backpropagation.* It introduces the intra-layer connections to the hidden layer. It has been shown by other researchers that the intra-layer or lateral connections can provide contrast enhancement to signals during encoding and recall (Cohen & Grossberg [24]). This enhancement increases the stability and the speed of convergence of the *lateral feedback* neural network.

The *lateral feedback* network introduces the intra-layer connections to the hidden units with self-excitatory and neighborhood-inhibitory mechanisms. The neighborhood is defined in the network architecture as the nearest neighbors surrounding a hidden neurode. The learning algorithms for the network has been developed by extending the generalized delta rule to this special network architecture. The benchmark tests and the applications to *TVA's Watts Bar* Nuclear Power Plant simulator data show that the *lateral feedback* network has advantages in the speed of learning, convergence, and stability over the conventional *backpropagation.* These successful applications also indicate that the lateral feedback network can be a serviceable tool for nuclear power plant applications in the fields of fault diagnosis and advanced control.

6.1.2 Sensitivity Analysis

The sensitivity analysis has been utilized for both *backpropagation* and *lateral feedback* networks. The sensitivity analysis is not new for general mathematical functions, which is the analysis of the first derivatives, but introducing it to the neural networks is a contribution of this dissertation. It is particularly useful to analyze a complicated system where mathematical modeling of the system is very difficult to develop. If the necessary information is provided by the available system input and output data, the neural network model can be built up and the sensitivity analysis can be applied to extract useful information about the system, such as the identification of the most important variables to control in order to improve the system performance. This is demonstrated in Chapter 3 for *TVA's Sequoyah* nuclear power plant thermal performance study. It can also be used to guide the selection of input variables for a set of small modular networks to monitor plant operating status and diagnose the possible accidents. This is demonstrated in Chapter 4 with application to fault diagnostics for *TVA's Watts Bar* Nuclear Power Plant. In addition, the concept of modular network is very useful if the system is very large and complex where a lot of process variables and accident scenarios need to be monitored. Also, it could be used to identify degrees of a particular transient.

6.1.3 Application of Genetic Algorithms

A genetic algorithm is a very efficient and robust search methodology that can perform optimal search in a large and complex search space where conventional algorithms, such as calculus-based search and hill-climbing methods, can not perform well. Though genetic algorithm has been developed by its founder John Holland [14] and following researchers (i.e. Goldbery [16]), the contribution of this dissertation is to use it with neural networks to guide the selection of proper input variables to simplify the network systems.

Both sensitivity analysis and genetic algorithms can be used to guide the selection of proper input variables for small modular networks, and the results from both methods are competitive. The limitation in using the sensitivity analysis is that a motherhood network has to be built up before the analysis can be applied. If a system is large and complex and if there are many process variables and scenarios to be monitored, the corresponding motherhood network will be very large and may fail in training. The genetic algorithm, on the other hand, does not require any kind of motherhood network being trained first. It directly gives the solution to the given task through the optimal search. The disadvantage of using genetic algorithm is that it involves a lot of computation to perform the optimal search. Therefore, it may be expensive in operation.

6.1.4 Diagnostics of Nuclear Power Plant Transients

The diagnostics of nuclear power plant transients was performed using modular networks. Because there is usually a large number of plant process variables and accident scenarios that need to be monitored to ensure the plant safety and high performance, it is impossible to use a single neural network to monitor the entire plant system. Therefore, building up modular networks to monitor the plant system is necessary. The crucial point to success in building up the modular networks is to choose the proper inputs for different modular networks to reach simple network structure, fast training, and accurate recall. Two methodologies were introduced in this dissertation to help choosing network inputs. One is to use sensitivity analysis, and the other is to use genetic algorithms. Both methods were used to guide the selection of inputs for modular networks. The results show that the modular networks built by using inputs selected by genetic algorithms give better performance.

All accident scenarios except *Cold-leg LOCA* can be monitored efficiently

by the corresponding modular networks. The modular networks can give quick and accurate diagnosis for all simulated accident scenarios. The time used for diagnoses is shorter than the corresponding reactor scram time. The noisy recall data degraded the performance of some modular networks and made the diagnosing time longer than the corresponding reactor scram time. The confusion between *Clod-leg LOCA* and *Hot-leg LOCA* in the middle of the transient was encountered even with the motherhood network that used 22 plant variables. Therefore, more plant process variables, which are currently not included in the raw data file of the scenarios, are necessary to overcome this confusion.

6.2 Recommendations for Future Research

Lateral feedback network shows the advantages in the speed of training, convergence, and stability over the conventional *backpropagation* network. Further research may extend the intra-layer connections to networks with more than one hidden layers and/or to an output layer as well. The definition of neighborhood may also be studied further to explore the relation of network performance and the intra-layer connection architectures.

The concept of the sensitivity analysis may be extended to other supervised learning network paradigms. The higher order of derivatives may also be developed to extract more information about the system simulated by neural networks.

Genetic algorithm can be used to explore the optimal architecture for neural networks, such as the optimal number of hidden layers and the number of units in the hidden layer, choice of transfer function parameters, partial connections, and learning rates.

In the application part of this dissertation, there is some confusion between the diagnosis of *Hot-leg LOCA* and *Cold-leg LOCA*. More plant process parameters need to be sampled in order to distinguish between *Hot-leg LOCA* and *Cold-leg*

LOCA. To approach the actual plant application, many more accident scenarios have to be monitored, which requires that more data be available from the plant to train the networks. Moreover, the neural network diagnostic system could be coupled with the plant's safety parameter display system to help the operators monitor the operating conditions of the whole power plant. The noise tolerance of the neural networks could be improved by adding noisy data and using additional input variables for training. Finally, the nuclear power plant thermal performance studies with sensitivity analysis methodology should be continued with carefully selected network inputs, which should be basic plant process variables monitored during the plant operation, and the results should be compared with these obtained from other analyses.

BIBLIOGRAPHY

Bibliography

[1] Arbib, A. M., "The Metaphorical Brain 2 Neural Networks and Beyond," John Wiley & Sons, New York, 1989.

[2] Amit, J. D., "Modeling brain function: The world of attractor neural networks," Cambridge University Press, New York, 1989.

[3] Kohonen, T., "Self-Organization and Associative Memory," New York: Springer-Verlag, 1984.

[4] Cotterill, R. M. J., "Computer simulation in brain science," Cambridge University Press, New York, 1988.

[5] Anderson J. A. and Rosenfeld E. "Neurocomputing: Foundation of Research," MIT Press, Cambridge, Massachusetts, 1988.

[6] Rumelhart D., McClelland J. L., and PDP Research Group "Parallel Distributed Processing: Volume 1: Foundation; Volumn 2: Psychological and Biological Models," MIT Press, Cambridge, Massachusetts, 1988.

[7] Nadel, L., Cooper, L. A., Culicover, P., and Harnish, R. M., "Neural Connections, Mental Computation," MIT Press, Cambridge, Massachusetts, 1989.

[8] Wasserman, P. D., "Neural Computing — Theory and Practice," Van Nostrand Reinhold, New York, 1989.

[9] Simpson, Patrick k., "Artificial Neural Systems," Pergamon Press, New York, 1989.

[10] Khanna, Tarun, "Foundations of Neural Networks," Addison-Wesley, New York, 1990.

[11] Caudill, M., Mahaffey, K., and Butler, C., "Understanding Neural networks: Computer Explorations," MIT Press, Cambridge, Massachusetts, 1990.

[12] Maren, A., Harston, C. and Pap, R., "Handbook of Neural Computing Applications," Academic Press, San Diego, California, 1990.

[13] Hertz, J., Krogh, A. and Palmer, R. G., "Introduction to the theory of neural computation," Addison-Wesley, Redwood City, California, 1991.

[14] Holland, John H, "Adaptation in Neural and Artificial Systems," Ann Arbor, Mich.: University of Michigan Press, 1975.

[15] Davis, Lawrence, "Genetic Algorithm and Simulated Annealing," MOrgan Kaufmann Publishers, Inc., Los Altos, California, 1987.

[16] Glodberg, David E, "Genetic Algorithms in Search, Optimization, and Machine Learning," Addison-Wesley, New York, 1989.

[17] Werbos, P., "Beyond regression: New tools for prediction and analysis in the behavioral science," Ph. D Dissertation, Harvard University, 1974.

[18] Almeida, L., "A learning rule for asynchronous perceptrons with feedback in a combinational environment," Proceeding of the IEEE First International Conference on Neural Networks: Vol. 2, pp. 609-618. San Diego:IEEE, 1987.

[19] Rohwer, R. & Forrest, B., "Training time-dependence in neural networks," Proceedings of the IEEE International Conference on Neural Networks: Vol. 2, pp. 701-708. San Diego:IEEE, 1987.

[20] Samad, T. & Harper, P., "Associative memory storage using a variant of the generalized delta rule," Proceedings of the IEEE International Conference on Neural Networks: Vol. 3, pp. 173-184. San Diego:IEEE, 1987.

[21] Shimohara, K., Uchiyama, T. & Tokunuga, Y., "Backpropagation networks for event-driven temporal sequence processing," Proceedings of the IEEE International Conference on Neural Networks: Vol. 1, pp. 665-672. San Diego:IEEE, 1988.

[22] Atiya, A., "Learning on a general network," In D. Anderson (Ed.),Proceedings of the 1987 IEEE Conference on Neural Information Processing System – Natural and Synthetic, pp. 22-30. New York:American Institute of Physics, 1988.

[23] Pineda, F., Generalization of backpropagation to recurrent and higher order neural networks," In D. Anderson (Ed.), Proceedings of the 1987 IEEE International Conference on Neural Information Processing Systems – Natural and Synthetic, pp. 602-611. New York: American Institute of Physics, 1988.

[24] Cohen, M. & Grossberg, S., "Absolute stability of global pattern formation and parallel memory storage by competitive neural networks," IEEE Transactions on Systems, Man, and Cybernetics, SMC-13, pp. 815-825, 1983.

[25] Grossberg, S. "Some nonlinear networks capable of learning a spatial pattern of arbitrary complexity," Proceedings of National Academy of Science: Vol. 59, pp.368-372, 1968.

[26] Grossberg, S., "Contour enhancement, short term memory, and constancies in reverberating networks," Studies in Applied Mathematics: Vol. 52, pp. 217-257, 1973.

[27] Carpenter, G. & Grossberg, S., "Absolute stable learning of recognition codes by self-organizing neural network," In J. Denker(Ed.), AIP Conference Proceedings 151: Neural Networks for Computing, pp. 77-85. New York: American Institute of Physics, 1986.

[28] Carpenter, G. & Grossberg, S., "ART2: Self-organization of stable category recognition codes for analog input patterns," Proceedings of the IEEE First International Conference on Neural networks: Vol. 2, pp 727-736. San Diego:IEEE, 1987.

[29] Carpenter, G. & Grossberg, S., "ART2: Self-organization of stable category recognition codes for analog input patterns," Applied Optics: Vol. 26, pp. 4919-4930, 1987.

[30] Hetch-Nielsen, R., "Counterpropagation networks," Proceedings of the IEEE First International Conference on Neural Networks: Vol. 2, pp. 19-32. San Diego:IEEE, 1987.

[31] Kosko, B., "Competitive adaptive bidirectional associative memory," Proceedings of the IEEE First International Conference on Neural Networks: Vol. 2, pp. 759-766. San Diego:IEEE, 1987.

[32] Moody, J., "Dynamics of lateral interaction networks," INNC-90-Paris: Vol. 2, pp. 1001-1004. Paris, France, July, 1990.

[33] "An introduction to Genetic Algorithms," AI Expert, March 1990, pp. 49-53.

[34] Uhrig, R. E.,"Potential Application of Neural Networks to the Operation of Nuclear Power Plants," Nuclear Safety: Vol. 32 No. 1, pp. 68-79, 1991.

[35] Bartlett, E. B., "Nuclear Power Plant Status Diagnostics Using Simulated Condensation: An Auto-Adaptive Computer Learning Technique," Ph.D Dessertation, Department of Nuclear Engineering, University of Tennessee, 1990.

[36] "Watts Bar Nuclear Power Station, Watts Bar Control Room Instrumentation Report," Rev C, Soddy-Daisy Tennessee, 1989.

[37] Whitely, D. and Bogart, C., "The Evolution of Connectivity: Pruning Neural networks Using Genetic Algorithms", IJCNN-90-Wash-DC: Vol. 1, pp. 134-137. Washington D.C Jan. 1990.

[38] Whitely, D. and Starkwerther T., "Optimizing Small Neural Networks Using a Distributed Genetic Algorithm," IJCNN-90-Wash-DC: Vol. 1, pp. 206-209. Washington DC, Jan. 1990.

[39] Whitely, D. and Bogart, C., "Optimizing Neural Networks using Faster, More Accurate Genetic Search," Proceedings of the Third International Conference on Genetic Algorithms. Morgan Kaufmann, 1989.

[40] Spears, William M. and De Jong, Kenneth A., "Using Neural Networks and Genetic Algorithms as Heuristic for NP-Complete Problem," IJCNN-90-Wash-DC: Vol. 1, pp. 118-121. Washington DC, Jan. 1990.

[41] Koza, John R. and Keane, Martin A., "Cart Centering and Broom Balancing by Genetically Breeding Populations of Control Strategy Programs," IJCNN-90-Wash-DC: Vol. 1, pp. 198-201. Washington DC, Jan. 1990.

[42] Maricic, Borut and Nikolov, Zoran, "GENNET - System for Computer Aided Neural Network Design Using Genetic Algorithms," IJCNN-90-Wash-DC: Vol. 1, pp. 102-105. Washington DC, Jan. 1990.

[43] Hugo de Garis, "Modular Neural Evolution for Darwin Machine," IJCNN-90-Wash-DC: Vol 1, pp. 194-197. Washington DC, Jan. 1990.

[44] Muselli, Marco and Ridella, Sandro, "Supervised Learning Using a Genetic Algorithm," INNC-90-Paris: Vol. 2, pp. 790. Paris, France, July, 1990.

[45] N-NET User's Manual, AI WARE Incorporated, Cleveland, OH, 1989.

APPENDICES

Appendix A

GENETIC ALGORITHMS: THEORY AND OPERATION

A.1 Introduction

Genetic Algorithms are search algorithms based on the mechanics of natural selection and natural genetics. The principle behind genetic algorithms is essentially Darwinian natural selection.

In the genetic algorithms, specially structured bit strings are designed to represent a set of parameters for a practical problem in the optimal search. Individuals, which are in the form of bit strings, are organized to form a population. The genetic algorithms combine survival of the fittest among string structures with a structured yet randomized information exchange to form the search algorithm which efficiently exploit historical information to speculate on new search points with expected improved performance. The genetic algorithms are theoretically and empirically proven to provide robust search in complex search spaces. Having been established as a valid approach to problems requiring efficient and effective search, genetic algorithms are now finding more widespread applications in business, scientific, and engineering fields.

A.2 Basic Theory and Operation

A.2.1 Driven Force of Genetic Algorithms

Genetic algorithms consist of a population of individuals competing on a survival-to-the-fittest in an environment. The algorithm proceeds in steps called generations. During each generation, a new population of individuals is created from the old via application of genetic operators, and evaluated as solutions to a given problem (the environment). Due to selective pressure, the population adapts to the environment over succeeding generations, involving better solutions [16]. If the environment is a function, genetic algorithms can be used for function optimization. In this case, each individual in a population is a sample point in the search space. Classically, an individual in a genetic algorithm is represented as a bit string of some length n. Each individual thus represents one sample point in a space of size 2^n. The three basic genetic algorithm operators forms the simple genetic algorithms, which are *reproduction*, *crossover*, and *mutation*, and they yield good solutions to many practical problems.

A.2.2 Simple Genetic Algorithm

The *reproduction* operator is an artificial version of nature selection, a Darwinian survival of the fittest among string creatures. In the *reproduction* process,

individual strings are copied according to their objective function values, f (biologist called this function the fitness function). Copying strings according to their fitness values means that candidates with the higher fitness values have a greater probability of contributing one or more offsprings in the next generation. The selection probability for individual i (the ith string in the population) may be defined as:

$$p_i = f_i / \sum_{i=1}^{N} f_i \tag{A.1}$$

where f_i is the fitness value (or objective function value) of ith individual in the population of size N.

We select the mating pool of the next generation according the probability p_i. Once an individual has been selected for *reproduction*, it is then entered into the mating pool, a tentative new population, for further genetic operator action. If there is no overlapping between populations, which means that the populations size stays constant and a new generation entirely replaces the old (or parent) generation, the expected counter of *reproduction* of ith individual is:

$$n_i = N * p_i = N * f_i / \sum_{i=1}^{N} f_i = f_i / (\sum_{i=1}^{N} f_i / N) = f_i / \overline{f} \tag{A.2}$$

where $\overline{f}$ is the average fitness of the population.

It gives us what we should expect: the best produces more copies, the average stays even, and the worst dies off.

After *reproduction*, simple *crossover* may proceed in two steps. First, members of the newly reproduced strings in the mating pool are mated at random. Second, each pair of strings undergoes *crossover* as follows: an integer position k along the string is selected between 1 and string length less one $[1, l-1]$. Two new strings (offsprings) are created by swapping all characters between positions $k+1$ and l inclusively. For example, the two strings '10010110' and '11100101', split after the third bit as shown below in (a) and (b), would create two new strings of (c) and (d) after *crossover*.

Label	Old String	Label	New String
(a)	100<u>10110</u>	(c)	100<u>00101</u>
(b)	111<u>00101</u>	(d)	111<u>10110</u>

Mutation , in the simple genetic algorithms, is the occasional (with small probability) random alteration of the value of a string position. In the binary coding representation, this simply means changing a 1 to a 0 and vise versa. By itself, *mutation* is a random walk through a string space. When used sparingly with *reproduction* and *crossover*, it is an insurance policy against premature loss of

important notations. With small probability of occurrence, *mutation* is considered as secondary mechanism of genetic algorithm adaption, or as a background operator.

The function of *crossover* is to generate rearrangements of coadapted groups of bits from high performance strings. It dose not create any new "genetic material" in the knowledge base. *Mutation*, on the other hand, randomly changes a bit in a string, thereby introducing new material into the knowledge base.

The mechanics of *reproduction* and *crossover* are surprisingly simple, involving a random number generator, string copies, and some partial string exchanges. Nonetheless, the combined emphasis of *reproduction* and the structured, though randomized, information exchange of *crossover* give genetic algorithms much of their power. How can such two simple operators result in anything useful, let along a rapid and robust search mechanism? What is being processed by genetic algorithms and how do we know whether processing it (whatever it is) will lead to optimal results in a particular problem? The next section will explore this in more detail.

A.2.3 Similarity Templates (Schemata)

The fundamental question we may ask is that in a search process given only fitness value, what information is contained in the population of strings and their fitness function values to help guide a directed search for improvement? The study of two things may answer this question. One is to seek the similarities among strings in the population, and the other one is to look for causal relationship between these similarities and high fitness. How do we measure the similarities among the strings?

Holland [14] introduced *schema* as a similarity template describing a subset of strings with similarities at certain string positions. If we use * as the *don't care* symbol and add it to the binary alphabet{0,1} to form ternary alphabet {0,1,*}. For the position by position comparison among strings, a 1 matched a 1, a 0 matches a 0, and a * matches either. For example, the schema *0000 matches two strings, namely {10000, 00000}, *111* matches a subset of four members {01110, 01111, 11110, 1111}, and 0*1** matches any of eight strings of length 5 that begin with 0 and have 1 in the third position. It is clear that the idea of a schema give us a powerful and compact way to talk about all well-defined similarities among finite-length strings. It should be emphasized that * is only a metasymbol (a symbol about other symbols); it is not explicitly processed by the genetic algorithms. It is simply a notional device that allows description of all possibilities among strings of particular length and alphabet. In general, for alphabet of cardinary (number of alphabet characters) k, there are $(K+1)^l$ schemata. But for a fixed string of length l, it contains 2^l schemata because each position may take on its actual value or a don't care symbol. As a result, a population of size n contains somewhere between 2^l to $n \cdot 2^l$ schemata depending upon the population diversity. It is clear now that a wealth of information about important similarities is indeed contained in even

moderately sized population, which help guide the genetic search.

How do the important schemata grow up or decay out in the simple genetic algorithm search?

The effect of *reproduction* is easy to determine; since highly fitted strings have higher probability of selection. On average, we are given an ever increasing number of samples to the observed best similarity patterns; however, *reproduction* along samples no new points in the search space, because it only copies some strings from the old generation, which have been visited in search before.

Crossover leaves a schema unscathed if it dose not cut the schema, but it may disrupt a schema when it does. For example, consider two schemata 1***0 and **11*. The first is likely to be disrupted by *crossover* while the second is relatively unlikely to be destroyed. As a result, schemata with short defining length are left along by *crossover* and reproduced at a good sampling rate by *reproduction* operator.

Mutation, with low rate at normal, does not disrupt a particular schema very frequently. It has little effect on the important schemata.

In conclusion, highly fit, short-defining-length schemata (called building blocks) are propagated generation to generation by given exponentially increasing samples to the observed best; all this goes in parallel with no special bookkeeping or special memory other than the population of n strings.

A.2.4 Schema Theorem and Building Block Hypothesis

The mathematic foundations of genetic algorithms can be found from Goldberg [16]. To explain the idea of schema, two terminologies, called order and defining length, need to be defined.

The *order* of schema H, denoted by $o(H)$, is simply the number of fixed positions (in binary alphabet, the number of 1's and 0's) presented in the template. For example, $o(011 * 1 * *) = 4$.

The *defining length* of a schema H, denoted by $\delta(H)$, is the distance between the first and last specific string position. For example, the schema 011*1** has defining length $\delta = 4$ because the last specific position is 5 and the first specific position is 1, and the distance between them is $\delta(H) = 5 - 1 = 4$.

During *reproduction*, a string is copied according to its fitness. More precisely, a string A_i gets selected with probability $p_i = f_i/\bar{f}$, where $\bar{f}$ is the fitness of the population average. The schemata are processed the same way. Therefore, schemata with fitness values above the population average will receive an increasing number of samples in the next generation, while schemata with fitness values below the population average will receive a decreasing number of samples. This expected behavior is carried out in parallel with every schema H contained in a particular population A. In other words, all the schemata in a population grow or decay according to their schema averages under the operation of *reproduction* alone. It is shown [16] that *reproduction* allocates exponentially increasing (decreasing) number of trails to above- (below-) average schemata. To some extent, it is curious that *reproduction*

can allocate exponentially increasing and decreasing number of schemata to further generation in parallel. But there will be no new points searched if we only copy old structures without change. This where *crossover* steps in.

Crossover is a structured and yet randomized information exchange between strings. It creates new strings with minimum of disruption to the allocation strategy dictated by *reproduction* along. This results in exponentially increasing (or decreasing) proportions of schemata in a population on many of the schemata contained in the population.

With both *crossover* and *reproduction* (the effect of *mutation* is simply ignored because of its small probability of occurrence), It is clearly that those schemata with both above-average observed performance, low order, and short defining length are going to be sampled at an exponentially increasing rate in subsequent generations. This conclusion is so important that it is given a special name: the *Schema Theorem.*

It was mentioned previously that we have somewhere between 2^l and $n \cdot 2^l$ schemata in a string population with length l and size n. But not all of these are processed with high probability because *crossover* destroys those with relatively high orders and long defining lengths. Goldberg [16] indicated that despite the process of only n strings each generation, a genetic algorithm processes something link n^3 schemata. This result is so important, Holland [14] has given it a special name, *implicit parallelism.* Even through for each generation we perform computation proportional to the size of the population, we get useful processing of something like n^3 schemata in parallel with no special bookkeeping or memory other than the population itself.

It is very clear now that short, low-order, and highly fitted schemata are sampled, recombined, and resampled to form strings of potentially higher fitness. In a way, by working with these particular schemata (the building blocks), we have reduced the complexity of our problem; instead of building high-performance strings by trying every conceivable combination, we construct better and better strings from the best partial solution of past sampling.

Because highly fit schemata of short defining length and low- order play such an important rule in the action of genetic algorithms, we have already given them a special name: *Building blocks.* The genetic algorithms discover new solutions by speculating on many combinations of building blocks within the current population to form better strings.

There are some other advanced genetic algorithm operators, such as *Dominance, Inversion, Intrachromosomal duplication, Delation, Translocation*, and *Segregation.* These advanced operators add a lot of power to the genetic algorithms to deal with very complicated problems.

Appendix B

ACCIDENT SCENARIO DESCRIPTIONS

The following are the descriptions for each of seven accident scenarios simulated by **TVA's Watts Bar** simulator that are used in this dissertation.

1. Total Loss of Offsite Power:
   Power is unavailable from the 500 Kv yard. Power is lost to all shutdown boards. A reactor-turbine trip is initiated. All diesel generators automatically start and energize their respective 6.9 kv boards. The black-out sequence is initiated. The loss of all reactor coolant pumps initiates reactor coolant system natural circulation. Station power and support are available using the diesel generators.

2. Main Steam Line Break Inside Containment:
   Main steam line breaks inside containment caused by the pipe failure upstream of flow element. Steam flow and steam generator pressure decreases and level increases. Reactor coolant system average temperature increases, resulting in pressurizer pressure and level decrease while reactor power increases. Containment pressure, temperature and humidity increase.

3. Main Feedwater Line Break Inside Containment:
   Steam generator inlet pipe breaks. Steam generator pressure decreases. Mass lost from steam generator blows out into reactor containment with increasing containment pressure, humidity and sump level. Loss of steam generator water level will cause feedwater isolation. High containment pressure will cause initiation of safety injection, containment isolation and containment spray. Steam generator blowdown to containment together with safety injection leads to a solid plant condition.

4. Control Rod Ejection:
   Reactor core control assembly F06 housing assembly fails. Reactor coolant pressure ejects reactor core control assembly F06 from the core. The rod position indicator will indicate that the rod has been dropped. Reactor power will respond to the available positive reactivity.

5. Hot Leg Loss Of Coolant Accident:
   Hot leg breaks caused by pipe failure at reactor vessel nozzle. Reactor coolant system depressurizes and flushes into containment volume. Containment radiation, temperature, pressure, humidity, and sump level increase. Reactor trips and safety injection initiates, high containment pressure isolates containment. Reactor pumps cavitate, charging, safety injection, residual heat remove pumps and cold leg accumulators will reflood the core. Ice condenser doors open to help condense the steam.

6. Cold Leg Loss Of Coolant Accident:
   Cold leg breaks caused by the pipe failure at reactor vessel nozzle. Results are similar to hot leg loss of coolant accident except that the core will heat up at a faster rate due to decreasing in coolant flow.

7. Steam Generator tube Break:
   Steam generator tube ruptures at top of bundle. Pressurizer level and pressure decrease. Steam generator radiation, level and pressure increase.

Appendix C

PLANT PROCESS VARIABLES LIST

Below is an ordered list of 81 plant process variables in raw data files generated by the **TVA Watts Bar** nuclear power station simulator.

RCS Power Control

1. CMD ZAOGI1	Shutdown Bank A Array Pos. Ind.	Stps
2. CMD ZAOGI2	Shutdown Bank B Array Pos. Ind.	Stps
3. CMD ZAOGI3	Shutdown Bank C Array Pos. Ind.	Stps
4. CMD ZAOGI4	Shutdown Bank D Array Pos. Ind.	Stps
5. CMD ZAOLI6	Control Bank A Array Pos. Ind.	Stps
6. CMD ZAOLI7	Control Bank B Array Pos. Ind.	Stps
7. CMD ZAOLI8	Control Bank C Array Pos. Ind.	Stps
8. CMD ZAOLI9	Control Bank D Array Pos. Ind.	Stps
9. CMD ZAOLI3225	Unclear Power Level	0 - 100%
10. CMD ZAOEI5716A	Unit Megawattage Output	0 - 1500 MW
11. CMD ZAOLI62129	Vol. Control Tank Level	0 - 100%
12. CMD ZAOLI77410	RB FL & Equip. Drain Level	0 - 100%
13. CMD ZAOPI30310	Containment Pressure	-5, 60 PSI
14. CMD ZAONI303	P.R. % Full Power AO Array	0 - 120%
15. CMD ZAONI41C	N41 % Flux Diff. AO	-30, 30%
16. CMD ZAONI42C	N42 % Flux Diff. AO	-30, 30%
17. CMD ZAONI43C	N43 % Flux Diff. AO	-30, 30%
18. CMD ZAONI44C	N44 % Flux Diff. AO	-30, 30%

Steam Flow Indicators

19. CMD ZAOFI13A	SG #1 Steam Flow Ch. 1 Mtr	PPH
20. CMD ZAOFI110A	SG #2 Steam Flow Ch. 1 Mtr	PPH
21. CMD ZAOFI121A	SG #3 Steam Flow Ch. 1 Mtr	PPH
22. CMD ZAOFI128A	SG #4 Steam Flow Ch. 1 Mtr	PPH

Steam Pressure

23. CMD ZAOPI11C	SG #1 Main Steam HDR Press	0 - 1300 PSIG
24. CMD ZAOPI112	SG #2 Main Steam HDR Press	0 - 1300 PSIG
25. CMD ZAOPI119C	SG #3 Main Steam HDR Press	0 - 1300 PSIG
26. CMD ZAOPI126C	SG #4 Main Steam HDR Press	0 - 1300 PSIG

Feedwater

27. CMD ZAOFI335A	SG #1 Feedwater Inlet Flow	PPH
28. CMD ZAOFI348A	SG #2 Feedwater Inlet Flow	PPH
29. CMD ZAOFI390A	SG #3 Feedwater Inlet Flow	PPH
30. CMD ZAOFI3103A	SG #4 Feedwater Inlet Flow	PPH
31. CMD ZAOPI337	SG #1 FW Supply Press	0 - 1300 PSIG
32. CMD ZAOPI350	SG #2 FW Supply Press	0 - 1300 PSIG
33. CMD ZAOPI392	SG #3 FW Supply Press	0 - 1300 PSIG
34. CMD ZAOPI3150	SG #4 FW Supply Press	0 - 1300 PSIG
35. CMD ZAOFI3163A	SG #1 Aux. FW Inlet Flow	0 - 150 GPM
36. CMD ZAOFI3155A	SG #2 Aux. FW Inlet Flow	0 - 150 GPM
37. CMD ZAOFI3147A	SG #3 Aux. FW Inlet Flow	0 - 150 GPM
38. CMD ZAOFI3170A	SG #4 Aux. FW Inlet Flow	0 - 150 GPM

SG Level Indicator

39. CMD ZAOLI338	SG #1 Level Indicator	0 - 100%
40. CMD ZAOLI343B	SG #1 Level Wide Level Ind	0 - 100%
41. CMD ZAOLI351	SG #2 Level Indicator	0 - 100%
42. CMD ZAOLI356B	SG #2 Level Wide Level Ind	0 - 100%
43. CMD ZAOLI393	SG #3 Level Indicator	0 - 100%
44. CMD ZAOLI398B	SG #3 Level Wide Level Ind	0 - 100%
45. CMD ZAOLI3106	SG #4 Level Indicator	0 - 100%
46. CMD ZAOLI3111B	SG #4 Level Wide Level Ind	0 - 100%

RCS Pressurizer

47. CMD ZAOLI68320	RCS Pressurizer Level	0 - 100%
48. CMD ZAOLI68321	RCS Pressurizer Level (CC)	0 - 100%
49. CMD ZAOPI68337C	RCS PZR Pressure	1700 - 25000 PSIG
50. CMD ZAOPI68342C	RCS PZR Wide Range Press	1700 - 25000 PSIG
51. CMD ZAOTI68317	RCS PZR Surge Line Temp.	100 - 700 DEGF
52. CMD ZAOTI68317	RCS Loop #1 Spray Temp.	100 - 700 DEGF
53. CMD ZAOTI68316	RCS Loop #2 Spray Temp.	100 - 700 DEGF
54. CMD ZAOLI68300	RCS PRT Level	0 - 100%
55. CMD ZAOPI68311C	RCS PRT Pressure	0 - 10 PSI

Coolant Loop Data

56. CMD ZAOPI6863	RCS Loop #1 Hot-leg Press	0 -3000 PSIG
57. CMD ZAOPI6864	RCS Loop #3 Hot-leg Press	0 -3000 PSIG
58. CMD ZAOTI6818	RCS Loop #1 Cold-leg Temp.	0 -700 DEGF
59. CMD ZAOTI681	RCS Loop #1 Hot-leg Temp.	0 -700 DEGF
60. CMD ZAOTI682E	RCS Loop #1 Ave. Temp.	530 - 630 DEGF
61. CMD ZAOFI686A	RCS Loop #1 Coolant Flow	0 - 110%
62. CMD ZAOTI6841	RCS Loop #2 Cold-leg Temp.	0 -700 DEGF
63. CMD ZAOTI6824	RCS Loop #2 Hot-leg Temp.	0 -700 DEGF
64. CMD ZAOTI6825E	RCS Loop #2 Ave. Temp.	530 - 630 DEGF
65. CMD ZAOFI6829A	RCS Loop #2 Coolant Flow	0 - 110%
66. CMD ZAOTI6860	RCS Loop #3 Cold-leg Temp.	0 -700 DEGF
67. CMD ZAOTI6843	RCS Loop #3 Hot-leg Temp.	0 -700 DEGF
68. CMD ZAOTI6844E	RCS Loop #3 Ave. Temp.	530 - 630 DEGF
69. CMD ZAOFI6848A	RCS Loop #3 Coolant Flow	0 - 110%
70. CMD ZAOTI6883	RCS Loop #4 Cold-leg Temp.	0 -700 DEGF
71. CMD ZAOTI6865	RCS Loop #4 Hot-leg Temp.	0 -700 DEGF
72. CMD ZAOTI6867E	RCS Loop #4 Ave. Temp.	530 - 630 DEGF
73. CMD ZAOTI6867A	RCS Lp #4 Delta Temp.% Full PWR	0-150%
74. CMD ZAOFI6871A	RCS Loop #4 Coolant Flow	0 - 110%

Radiation Monitoring

75. CMD ZAO1RI90120A	SG BLDN Liquid Monitor	1 - 1E7 CPM
76. CMD ZAO1RI90123A	SG CC Liq Effluent Monitor	1 - 1E7 CPM
77. CMD ZAO1RI90275	RC Drain Tank Rad. Monitor	1 - 1Ee mR/hr
78. CMD ZAO1RI90277	RC Equip Drain Sump Rad. Mon	1 - 1Ee mR/hr
79. CMD ZAO1RR90104	RCS LTDWN Liquid Rad. Mon	1 - 1E7 CPM
80. CMD ZAO1RI90106A	CNTMT Bldg Lower Comp Rad Det	1E1 - 1E7 CPM
81. CMD ZAO1RI90112A	CNTMT Bldg Upper Comp Rad Det	1E1 - 1E7 CPM

Appendix D

CODE FOR EXTRACT DATA FROM RAW DATA FILES

D.1 mkdata.c

```
/* This program is designed to extract data from the raw data files, ed1.dat, f23,dat, ...,
to form the format required by the use of neural networks. The variables are selected
according to the index files, i.e. index1.dat given below after the program. In the index
files, there are two columns. The first column is simply the alphabetical order of the 81
plant variables. The second column is the binary value of 1 or 0. A variable will be
selected by the program if the corresponding value in the second column is 1, or vise
versa. To run the program, a task file is required to process all the raw data files listed
in the task file. The index file and output files are also defined in the task file. One
sample task file is given in the end of this appdenix. */
#include<stdio.h>
#include<math.h>
#include "task_io.h"
#define LENGTH 20
#define TRUE 1
#define FALSE -1
main()
{
int i, j, k, l, m, n; /* loop variables */
char fname[LENGTH]; /* file name for file I/O */
int index[81][2]; /* index for select variables for the data bank */
FILE *fptask, *fpin, *fpout1, *fpout2; /* input and output file pointers */

int flag = TRUE, id; /* id is the pattern number indicator */
char ch;
int varid[81]; /* id # of selected variables */
int totvar, totpat; /* total selected variables and total patterns */
int position; /* position of target with nonzero value */
int f_tot = 0, t_tot = 0; /* total patterns in F status and T status */

int tot_files; /* total number of files to be precessed */
int f_allpat = 0; /* total f_data patterns from all f_data files */

struct unit {
char status;
float time;
float var[81];
}
; struct unit data[550]; /* each element contains the information of F ot T, time, and 81

variable values at the time moment */
```

```
printf("Input task file name > ");
scanf("%s", fname);
printf("\n\n");

if((fptask = fopen(fname, "r")) == NULL)
{
printf("Can't open file %s, the excution is halted.\n", fname);
exit(1);
}

tot_files = getint(fptask); /* total number of files listed in the task file */

printf("Total number of files to be precessed is %d\n\n", tot_files);

/* open the index data file and read the index data into the program. The index data
has value of either 1 or 0. if the value is 1 which means that the corresponding variable
will be selected for network use, and vice versa. */

getstring(fptask,fname);
printf("The file name of the index data is %s.\n\n", fname);
if((fpin = fopen(fname, "r")) == NULL)
{
printf("Can't open file %s, the excution is halted.\n", fname);
exit(1);
}

for(i=0; i<81; i++)
{
fscanf(fpin, "%d%d", &index[i][0], &index[i][1]);
printf("variable # %d index = %d\n", index[i][0], index[i][1]);
}
fclose(fpin);

k = 0;
for(i=0; i<81; i++)
if(index[i][1] == 1)
varid[k++] = i;
totvar = k;
printf("total selected variables are %d:\n", totvar);
for(i=0; i<totvar; i++)
printf("variable %d\n", varid[i]);

/* loop for processing all data files listed in the task file */
for(n = 0; n < tot_files; n++)
{
id = 0; /* initialize the id value for each data file */
t_tot = 0; /* initialize t_tot for each data file */
f_tot = 0; /* initialize f_tot for each data file */
```

```
getstring(fptask,fname);
printf("File name of the data file %d is %s.\n\n" , n+1, fname);

if((fpin = fopen(fname, "r")) == NULL)
{
printf("Can't open file %s, the excution is halted.\n", fname);
exit(1);
}

while(flag)
{
fscanf(fpin, "%c", &data[id].status);
while( data[id].status != 'F' && data[id].status != 'T')
fscanf(fpin, "%c", &data[id].status);

/* printf("status = %c\n", data[id].status); */
fscanf(fpin, "%f", &data[id].time);
if(data[id].time < 0.0)
goto OUT_LOOP;

if(data[id].status == 'F')
f_tot += 1;
else
if(data[id].status == 'T' && data[id].time > 0.0)
t_tot += 1;

/* printf("Time = %f\n", data[id].time); */
for(i=0; i<81; i++)
fscanf(fpin, "%f", &data[id].var[i]);

id += 1;
/* printf("id = %d\n", id); */
} /* end of while(flag) loop */

OUT_LOOP: fclose(fpin);
f_allpat += f_tot;

printf("Total patterns of F data are %d\n", f_tot);
printf("Total patterns of T data are %d\n", t_tot);

/* Open a file for data output */
position = getint(fptask);
printf("Position where 1.0 will be put is at %d\n\n", position + 1);

getstring(fptask,fname);
printf("File name for output file of F data is %s\n\n" , fname);

if((fpout1 = fopen(fname, "a")) == NULL)
{
printf("Can't open file %s, the excution is halted.\n", fname);
```

```
exit(1);
}

getstring(fptask,fname);
printf("File name for output file of T data is %s.\n\n ", fname);

if((fpout2 = fopen(fname, "w")) == NULL)
{
printf("Can't open file %s, the excution is halted.\n", fname);
exit(1);
}

/* prepare data file for network training according to index[] */

if(n==0)
{
fprintf(fpout1, "%d\n", f_tot);
fprintf(fpout1, "%d\n", totvar);
fprintf(fpout1, "7 \n\n");
} /* all f_data are written in one file and title lines are written just once. */

for(i=0; i<f_tot; i++)
{
for(j=0; j<totvar; j++)
{
fprintf(fpout1, "%f ", data[i].var[varid[j]]);
if((j+1) % 5 == 0)
fprintf(fpout1, "\n");
}
fprintf(fpout1, "\n");
/* if(data[i].status == 'F') */

for(j=0; j<7; j++)
fprintf(fpout1, "0.0 ");
fprintf(fpout1, "\n\n");
} /* end of i loop for F data */

fclose(fpout1);
/* else
if(data[i].status == 'T') */

fprintf(fpout2, "%d\n", t_tot);
fprintf(fpout2, "%d\n", totvar);
fprintf(fpout2, "7 \n\n");
for(i=f_tot; i<t_tot+f_tot; i++)
{
for(j=0; j<totvar; j++)
{
```

```
fprintf(fpout2, "%f ", data[i].var[varid[j]]);
if((j+1) % 5 == 0)
fprintf(fpout2, "\n");
}
fprintf(fpout2, "\n");
for(j=0; j<7; j++)
if(j == position)
fprintf(fpout2, "1.0 ");
else
fprintf(fpout2, "0.0 ");
fprintf(fpout2, "\n\n");
} /* end of i loop for T data */

fclose(fpout2);

} /* end of n loop for all data files to be processed by task file */

getstring(fptask,fname);
printf("File name for output file of F data is %s\n\n" , fname);

if((fpout1 = fopen(fname, "a")) == NULL)
{
printf("Can't open file %s, the excution is halted.\n", fname);
exit(1);
}

fprintf(fpout1, "\n\n\n Total number of f_data patterns are %d\n", f_allpat);
fclose(fpout1);

fclose(fptask);

} /* end of main */
```

D.2 A Sample of Index File

The index file is used to guide the program secelcting required variables for the raw data file where 81 plant variables are included.

```
1 0
2 0
3 0
4 0
5 0
6 0
7 0
8 0
9 1
```

10 1
11 1
12 1
13 1
14 1
15 0
16 0
17 0
18 0
19 1
20 0
21 0
22 0
23 1
24 0
25 0
26 0
27 1
28 0
29 0
30 0
31 1
32 0
33 0
34 0
35 1
36 0
37 0
38 0
39 1
40 1
41 0
42 0
43 0
44 0
45 0
46 0
47 1
48 0
49 1
50 0
51 1
52 1

53 0
54 0
55 0
56 1
57 0
58 1
59 1
60 1
61 1
62 0
63 0
64 0
65 0
66 0
67 0
68 0
69 0
70 0
71 0
72 0
73 0
74 0
75 0
76 0
77 0
78 0
79 0
80 0
81 0

D.3 A Sample Task File For mkdata.c

All raw data files need to be processed by makdata.c to form the data files required by the training and recall of the neural networks.

```
7           Number of data files to be processed
index2.dat  the selection of the variables from loop 2

ed1.dat     first data file to be processed
0           position where 1.0 instead of 0.0 will be put
fall2.pat   append output file for data with F(full power) status
ed1t2.pat   output file for data with F(full power) status

f23.dat     second data file to be processed
1           position where 1.0 instead of 0.0 will be put
fall2.pat   append output file for data with F(full power) status
f23t2.pat   output file for data with F(full power) status
ms1.dat     thrid data file to be processed
2           position where 1.0 instead of 0.0 will be put
fall2.pat   append output file for data with F(full power) status
ms1t2.pat   output file for data with F(full power) status

rd6.dat     forth data file to be processed
2           position where 1.0 instead of 0.0 will be put
fall2.pat   append output file for data with F(full power) status
rd6t2.pat   output file for data with F(full power) status

th1.dat     fifth data file to be processed
4           position where 1.0 instead of 0.0 will be put
fall2.pat   append output file for data with F(full power) status
th1t2.pat   output file for data with F(full power) status

th2.dat     sixth data file to be processed
5           position where 1.0 instead of 0.0 will be put
fall2.pat   append output file for data with F(full power) status
th2t2.pat   output file for data with F(full power) status

th5.dat     seventh data file to be processed
6           position where 1.0 instead of 0.0 will be put
fall2.pat   append output file for data with F(full power) status
th5t2.pat   output file for data with F(full power) status

fall2.pat   reopen the file to write total # of f_data patterns in it
```

Appendix E

CODE FOR MAKING NETWORK TRAINING DATA FILES

E.1 nndata.c

```
/* nndata.c is designed to put all formed cluster centroids together to form the netwrok
training file. It needs a task file to list all cluster files to be processed and define the
name of output file. One sample of the task file is given after the program. */

#include<stdio.h>
#include<math.h>
#include "task_io.h"
#define LENGTH 20
#define MAXPAT 500
#define MAXNODE 40

main()
{
int i, j, k, l, m, n; /* loop variables */
char fname[LENGTH]; /* file name for file I/O */
FILE *fptask, *fpin, *fpout; /* input and output file pointers */
char dummy, string[30]; /* dummy char and string */
int file; /* loop variable for input files */
int inputs, outputs; /* input and output variables in the file */
int tot_files; /* total number of files to be precessed */
int tot_pat; /* total patterns from all files */
int clusters; /* clusters in each data file */
int ip; /* loop variable for patterns */

float xin[MAXPAT][MAXNODE]; /* training inputs */
float tar[MAXPAT][MAXNODE]; /* training tragets */

printf("Input task file name > ");
scanf("%s", fname);
printf("\n\n");

if((fptask = fopen(fname, "r")) == NULL)
{
printf("Can't open file %s, the excution is halted.\n", fname);
exit(1);
}

tot_files = getint(fptask); /* total number of files listed in the task file */

printf("Total number of files to be precessed is %d\n \n", tot_files);

/* loop for processing all data files listed in the task file */
```

```
tot_pat = 0;
/* initialize total patterns indicator */
for(file = 0; file < tot_files; file++)
{
getstring(fptask,fname);
printf("File name of the data file %d is %s.\n \n", file+1, fname);

if((fpin = fopen(fname, "r")) == NULL)
{
printf("Can't open file %s, the excution is halted.\n", fname);
exit(1);
}

fscanf(fpin, "%s", string);
while((strcmp(string, "clusters:") != 0))
fscanf(fpin, "%s",string);

fscanf(fpin, "%d", &clusters);
printf("Clusters in file %d is %d.\n\n", file+1, clusters);

fscanf(fpin, "%s", string);
while((strcmp(string,"inputs:") !=0 ))
fscanf(fpin, "%s",string);

fscanf(fpin, "%d", &inputs);
printf("Inputs in file %d is %d.\n\n", file+1, inputs);

fscanf(fpin, "%s", string);
while((strcmp(string,"outputs:") !=0 ))
fscanf(fpin, "%s",string);

fscanf(fpin, "%d", &outputs);
printf("Outputs in file %d is %d.\n\n", file+1, outputs);

for(ip = 0;ip < clusters;ip++)
{
fscanf(fpin, "%s", string);
while((strcmp(string,"Centroid") != 0 ))
fscanf(fpin, "%s", string);
fscanf(fpin, "%c", &dummy);
while(dummy != '\n') fscanf(fpin, "%c", &dummy);

for(i=0; i<inputs; i++)
fscanf(fpin, "%f", &xin[tot_pat][i]);

fscanf(fpin, "%s", string);
while((strcmp(string,"Target") != 0 ))
fscanf(fpin, "%s", string);
fscanf(fpin, "%c", &dummy);
```

```
while(dummy != '\n')
fscanf(fpin, "%c", &dummy);

for(i=0; i<outputs; i++)
fscanf(fpin, "%f", &tar[tot_pat][i]);
/* printf("tar[%d][%d] = %f\n", tot_pat, i, tar[tot_pat][i]);
/ tot_pat += 1;

/* counter for total patterns of all files */ } /* end of ip loop for all clusters of one file */


fclose(fpin); /* close fpin for next use */

} /* end of for(file ...) loop for all files in the task list */

printf("Total patterns from (all) %d data file(s) are %d\n", tot_files, tot_pat);

/* Open a file for data output */

getstring(fptask,fname);
printf("File name for output file with NN format is %s\n\n", fname);

if((fpout = fopen(fname, "w")) == NULL)
{
printf("Can't open file %s, the excution is halted.\n", fname);
exit(1);
}

printf("Total patterns are %d, inputs = %d outputs = %d\n", tot_pat, inputs, outputs);
fprintf(fpout, "%d\n", tot_pat);
fprintf(fpout, "%d\n", inputs);
fprintf(fpout, "%d\n", outputs);

for(ip = 0; ip < tot_pat; ip++)
{
for(i=0; i < inputs; i++)
{
fprintf(fpout, "%f ", xin[ip][i]);
if((i+1) % 8 == 0)
fprintf(fpout, "\n");
}
fprintf(fpout, "\n");

for(i=0; i < outputs; i++) {
fprintf(fpout, "%f ", tar[ip][i]);
if((i+1) % 8 == 0) fprintf(fpout, "\n");
}
fprintf(fpout, "\n\n\n");

} /* end of ip loop for all patterns */
```

```
fclose(fpout);

fclose(fptask);

} /* end of main */
```

E.2 A Sample of nndata.c Task File

The nndata.c requires a task file to operate. The task file lists all the files which are the centroids of all formed clusters to be processed and the name of the output file which used as the training file for all single and modular networks.

8	Number of files to be processed
f_all4.clu	first file to be processed
ed1t4.clu	second file to be processed
ms1t4.clu	forth file to be processed
f23t4.clu	third file to be processed
rd6t4.clu	fifth file to be processed
th1t4.clu	sixth file to be processed
th2t4.clu	seventh file to be processed
th5t4.clu	eightth file to be processed
wb4_nn.dat	output file with NN formate

Appendix F

CODE FOR MODULAR NETWORK TRAINING

F.1 Main Program of subnet.c

The progam is the modification of lateral-feedback neural network. The modification allows the program to select variables required by an individual modular network from the data files made by mkdata.c. The functions used by the program will be discussed in latter sections.

```
#include "subnet.h"
#include "train2.sub"

main()
{
extern void task_select();
task_select(); /* select and perform a task: training, recall, or retraining */
} /* finished main() */
```

F.2 Functions Used in subnet.c

/* The following is the file of **subnet.h** which defines all functions used in subnet.c. Some functions in the list are the same as these used in **lateral-feedback** except **getdat3.sub**, **normalizer2.nn** and **get_weights.nn. getdat3.sub** is the function to extract the required data from the data files made by **mkdata.c** according to the selection of the variables for an individual modular network. **normalizer2.nn** is the function for the data normlization by the method of amplitude modulaion. **get_weights.sub** is the function to obtain weighing data from the sensitivity analysis for the the amplitude modulation in **normalizer2.n** */

```
#include<stdio.h>
#include<math.h>
#include<signal.h>
#include "globdef.nn"
#include "netsetup.nn"
#include "openfile.nn"
#include "getdat3.sub"
#include "normalizer2.nn"
#include "randomizer.nn"
#include "sigmoid.nn"
#include "matrix_sum.nn"
#include "layerout.nn"
#include "errorcal.nn"
#include "chweights.nn"
#include "netio2.nn"
```

```
#include "task_select.nn"
#include "recall2.sub"
#include "get_weight.sub"
#include "ctrlc_handler"
```

F.3 Train Function for subnet.c

```
/* The following function is the training function for subnet_main.c The change from
net_train.nn is to read the weighing data from the output of rankpe.c to moddify the
selected inputs in the process of normalization. The network I/O function has also been
changed from netio.nn to netio2.nn to save data of veriable selection index and the
corresponding weighing factors. */

void training(flag) int flag; {
extern float sum(), rnum(), sigmoid();
extern void random();
extern void random2();
extern FILE *fileopen();
extern void datain();
extern void netsave();
extern void setup();
extern float energyf();
extern void chwts();
extern void chseta();
extern void normf();
extern void netin();
extern void randomh();
extern void hidout();
extern void forward();
extern void normf();
extern void denorm();
extern void typeout();
extern void getw();
extern void ctrlc_handler(); /* task over control-c interruption and allow to save net
before quit */

int oldcount = 0; /* the # of training cycles for a trained net */
float energy1 = 0, energy2 = 0; /* old f new energy */
float wtemp[MAXNODE][MAXNODE][MAXNODE][MAXNODE];
float vtemp[MAXNODE][MAXNODE][MAXNODE][MAXNODE];
float setahtemp[MAXNODE][MAXNODE];
float setaotemp[MAXNODE][MAXNODE];
float htemp[MAXNODE][MAXNODE][9];
float hiderr[MAXNODE][MAXNODE];
float dw[MAXNODE][MAXNODE];
float dv[MAXNODE][MAXNODE];
```

```
float dntar, dnout; /* denormalized target and output values for error analysis */
int totvar = 0; /* total number of variables after selection */
int job; /* flag for TRAIN or RETRAIN = RECALL */
float temp = 0.0; int target_no; /* target in the big net which responses one scenario is
the one used to bulid the corresponding subnet */
char ch; /* character dummy */

if (flag == NEW)
{
printf("Here will open the index file for selecting inputs.\n");
fpin = fileopen(OLD);

fscanf(fpin, "%d", &tot_var);
for(i=0; i<tot_var; i++)
fscanf(fpin, "%d%d", &var_index[i][0], &var_index[i][1]);

fclose(fpin);

for(i=0; i<tot_var; i++)
if(var_index[i][1] == 1)
totvar += 1;

printf("Total selected variables are %d\n", totvar);
setup(); /* setup the network configuration */
job = TRAIN; /* flag for getting weighing data for inputs */
}
else
{
job = RETRAIN;
printf("Here will retraive a trained network for more training.\n");
netin(); /* retraive a trained network including network cinfig., lower and upper
bounders for normalization, weights, threshold values, etc. */
printf("The network has been trainined for %d times.\n", count);
printf("The definition of the network is:\n");
printf("%d row(s) & %d column(s) in the input layer.\n",inrow,incol);
printf("%d row(s) & %d column(s) in the hidden layer.\n",hidrow,hidcol);
printf("%d row(s) & %d column(s) in the output layer.\n",outrow,outcol);
printf("The sigmoid slopee used in for original training is %f\n",sslope);
}

/* get weighind data from the data file if job=TRAIN, or from netin() if job =
RETRAIN for retraining */

if ( job == TRAIN) /* for RECALL, weight will be got from netin() and */
{
/* can be passed to this function */
printf("Do you want to weigh your input data? Y or N.\n");
while((answer = getchar()) != 'Y' && answer != 'y' && answer != 'N' && answer != 'n')
```

```
;
if(answer == 'Y' —— answer == 'y')
{
printf("Here will open weight data file.\n");
fpin = fileopen(OLD);

/* get the target number for which the subnet will be built to response the scenario
associated with this target */
printf("Which target selected for the subnet >");
scanf("%d", &target_no);
printf("\n\n");

/* get weighing data for the target numbered target_no */
getw(fpin, target_no, weight);
fclose(fpin);
}
else
{
for(i=0; i<tot_var; i++)
weight[i] = 1.0; /* no weighing occurd */
} /* end of if(answer == 'Y' ...) block */
while ((ch = getchar()) != '\n') /* clear the remaining */
; /* input line */
} /* end of if (job == TRAIN) block */

printf("The weighing data are listed below:\n\n");
for(i=0; i<tot_var; i++)
printf("For input #%d %f\n", i, weight[i]);
printf("\nHere will open the input training file.\n");
fpin = fileopen(OLD); /* open a input training file */
datain(fpin,var_index); /* read the input data into xin[][][] f tar[][][] */
fclose(fpin);

/* Normalization is ignored here for wattsbar data */
if (flag == NEW)
{
normf(xin, tar, minin, maxin, minout, maxout, weight, NEW);
/* normalize the data column by column by their own min. and max. values , and low
and upper limit are modified by weight to make the inputs have different importance to
the network */
printf("The maximum and minimum input values are listed below:\n");

typeout(inrow, incol, minin, maxin);
printf("The maximum and minimum target values are listed below:\n");
typeout(outrow, outcol, minout, maxout);
}
else
```

```
{
printf("The maximum and minimum input values are listed below:\n");

typeout(inrow, incol, minin, maxin);
printf("The maximum and minimum target values are listed below:\n");
typeout(outrow, outcol, minout, maxout);
normf(xin, tar, minin, maxin, minout, maxout, weight, OLD);
/* normalize the input and target values by the old values used to train the network
before and weight data is from netin(). */
}

printf("\nThe training file is listed below after normalization:\n");
for (ip = 0; ip < npats; ip++)
{
for(i = 0; i < inrow; i++)
for(j = 0; j < incol; j++)
printf("%f ",xin[ip][i][j]);
printf("\n");

for(i = 0; i < outrow; i++)
for(j = 0; j < outcol; j++)
printf("%f ",tar[ip][i][j]);
printf("\n\n");
}

if(flag == NEW)
{
random(w, hidrow,hidcol,inrow,incol); /* randomize the connections */
random(v, outrow,outcol,hidrow,hidcol); /* betweem 1th and 2nd layer */
/* and the 2nd to the 3nd */
randomh(h, hidrow, hidcol); /* randomize lateral connections */
random2(setah, hidrow, hidcol); /* randomize thresholds of PEs */
random2(setao, outrow, outcol); /* in 2nd f 3nd layers */
}

/* initialize the momentum terms all equal 0.0 */
for(m = 0; m < hidrow; m++)
for(n = 0; n < hidcol; n++)
{
dsetahold[m][n] = 0.0;
for(k = 0; k < inrow; k++)
for(l = 0; l < incol; l++)
dwold[m][n][k][l] = 0.0;

for(k=0; k < 9; k++)
dhold[m][n][k] = 0.0;
}
```

```
for(m = 0; m < outrow; m++)
for(n = 0; n < outcol; n++)
{
dsetaoold[m][n] = 0.0;
for(k = 0; k < hidrow; k++)
for(l = 0; l < hidcol; l++)
dvold[m][n][k][l] = 0.0;
}

printf("Here a file will be open to save the middle results of training.\n");
fpout = fileopen(NEW); /* open a file to show the histry of training */

more = YES;
first = YES;
oldcount = count; /* # of training cycles of the old network */
count = 1; /* the # of cycles for more training */
answer = 'n'; /* initial setting of ch answer */

/* take over the control-c interruption and save net before quit */
if(signal(SIGINT, ctrlc_handler) == SIG_ERR) {
perror("signal failed");
exit(0);
}

while(more)
{
if(first == NO)
{
printf("The old values of alpha, beat, and alpham are %f %f %f\n", alpha, beta,
alpham);
printf("Do you want to moidfy these values? Y or N\n");
while((answer = getchar()) == '\n' —— answer == ' ')
;
if(answer == 'Y' —— answer == 'y')
{
printf("Input new values for alpha, beta, and alpham > ");
scanf("%f%f%f", &alpha, &beta, &alpham);
printf("\n\n");
}
}
else
{
printf("input 3 float values for learn rates of alpha, beta & alpham > ");
scanf("%f %f %f", &alpha,&beta, &alpham);
printf("\n");
}
```

```
if(first == NO)
printf("How many more iterations do you want ?\n");
else
printf("How many iterations to perform ?\n");
scanf("%d", &times);

printf("input a number to jump when disply and write out to a file.\n");
scanf("%d",&jump);

while (times--) /* Begin times interation */
{
if(count%jump == 1)
{
printf("Iterations = %d\n\n",count);
fprintf(fpout,"Iteration = %d\n\n",count);
printf("PATTREN OUTPUT TARGET ERROR(%%)\n");
printf("----- ---- ---- ----\n");
fprintf(fpout,"PATTREN OUTPUT TARGET ERROR(%%)\n");
fprintf(fpout,"----- ---- ---- ----\n");
}

for(ip = 0; ip < npats; ip++) /* begin loop calculation for each pattern */
{
forward(ip, w, v, h, setah, setao); /* cal. outputs forward from input layer to output
layer */

/* middle results output to the screen and a disk file */
for(i = 0; i < outrow; i++)
{
for(j = 0; j < outcol; j++)
{
if(count%jump == 1)
{
/* denormalize tar[] and out[] values for error analysis */
temp = (float)fabs((double)(maxout[i][j]-minout[i][j]));
if( temp < 1.0e-5) {
dntar = tar[ip][i][j] * maxout[i][j];
dnout = out[i][j] * maxout[i][j];
}
else
{
dntar = (tar[ip][i][j] - LOW_YLIMIT) * (maxout[i][j] - minout[i][j]) / (UP_YLIMIT -
LOW_YLIMIT) + minout[i][j];
dnout = (out[i][j] - LOW_YLIMIT) * (maxout[i][j] - minout[i][j]) / (UP_YLIMIT -
LOW_YLIMIT) + minout[i][j];
}
```

```
if(fabs(dntar) >= .0001)
temp = fabs((dntar - dnout) / dntar)*100.;
else
if(fabs(dnout) >= .0001)
temp = fabs((dntar - dnout) / dnout)*100.;
else
temp = fabs(dntar - dnout) * 100.;
printf(" %d %f %f %f\n", ip+1, dnout, dntar, temp);
fprintf(fpout," %d %f %f %f\n", ip+1, dnout, dntar, temp);
} /* end of if block */
} /* end of j loop */
if(count%jump == 1)
{
fprintf(fpout,"\n");
printf("\n");
}
} /* end or i loop */
if (count%jump == 1)
{
fprintf(fpout,"\n\n");
printf("\n\n");
}

/* energy1 = energyf(out, tar[ip]);
energy before weight adjustmants */

/* adjust weights between 1st & 2nd layer */
for(m = 0; m < hidrow; m++)
for(n = 0; n < hidcol; n++)
{
total = 0.;
for(i = 0; i < outrow; i++)
for(j = 0; j < outcol; j++)
total += (tar[ip][i][j] - out[i][j]) * out[i][j] * (1. - out[i][j]) * v[i][j][m][n];
total /= sslope; /* counting sigmoid slope */
total *= yhid[m][n] * (1. - yhid[m][n]) / sslope;
hiderr[m][n] = total;
dw[m][n] = total*(1.+ h[m][n][4]*yhidtemp[m][n]* (1.- yhidtemp[m][n]) / sslope);
setahtemp[m][n] = setah[m][n] - alpha * dw[m][n] + alpham*dsetahold[m][n];
} /* end of n_loop, m_loop, l_loop and k_loop */

for(k = 0; k < hidrow; k++)
for(l = 0; l < hidcol; l++)
for(m = 0; m < inrow; m++)
for(n = 0; n < incol; n++)
wtemp[k][l][m][n] = w[k][l][m][n] + alpha * dw[k][l]* xin[ip][m][n] + alpham *
dwold[k][l][m][n];
```

```
/* adjust lateral connections of 2nd layer PEs */
for(i=0; i<hidrow; i++)
for(j=0; j<hidcol; j++)
{
n = 0;
for(l=i-1; l<i+2; l++)
for(m=j-1; m<j+2; m++)
{
if((l>=0) && (m>=0) && (m<hidcol) && (l<hidrow)) yfb[n] = yhidtemp[l][m];
else
yfb[n] = 0.0;
n += 1;
} for(k=0; k<9; k++)
{
if(h[i][j][k] != 0.0)
{
if(k != 4)
htemp[i][j][k] = h[i][j][k] - beta*hiderr[i][j]*yfb[k] + alpham*dhold[i][j][k];
else
htemp[i][j][k] = h[i][j][k] + beta*hiderr[i][j]*yfb[k] + alpham*dhold[i][j][k];
}
} /* end of k-loop */
} /* end of j-loop and i-loop */

/* adjust weights between the 2nd layer and the 3nd layer */
for(i = 0; i < outrow; i++)
for(j = 0; j < outcol; j++)
{
dv[i][j] = out[i][j] / sslope *(1.- out[i][j])* (tar[ip][i][j]-out[i][j]);
setaotemp[i][j] = setao[i][j] - alpha * dv[i][j] + alpham*dsetaoold[i][j];
}

for(i = 0; i < outrow; i++)
for(j = 0; j < outcol; j++)
for(k = 0; k < hidrow; k++)
for(l = 0; l < hidcol; l++)
vtemp[i][j][k][l] = v[i][j][k][l] + alpha*dv[i][j]*yhid[k][l] + alpham*dvold[i][j][k][l];

/* calculate the outputs after the weight adjustments
forward(ip, wtemp, vtemp, htemp, setahtemp, setaotemp);
energy2 = energyf(out, tar[ip]); total error */
/* of the system after the weight adjustments */

/* if ( energy2 < energy1)
{ */
chwts(wtemp, vtemp, htemp); /* accept the weight adjustments */
chseta(setahtemp, setaotemp); /* accept the threshold adjustments */
```

```
/* dwold[][], etc. are also updated by these two functions */
/* } */
} /* end of the loop for the variable of ip (input pattern) */

if(count%jump == 1)
{
printf("\n");
printf("\n");
}
count++;
if( count%100 == 0 )
{
printf("The iteration has been performed %d times\n\n", count);
energy1 = 0.0; /* initialize system error */
for(ip = 0; ip < npats; ip++)
{
forward(ip, w, v, h, setah, setao);
energy1 += energyf(out, tar[ip]); /* total error of the system after the weight
adjustments */
}
printf("System error is %f\n",energy1/npats);
printf("\nThe training can be interrupted by hitting ctrl_c \n");
printf("and the netwrok will be saved before quit.\n");
}
} /* end times iteration */

printf("Do you want more inerations ?\nanswer y or n.\n");
while((answer = getchar()) == '\n' —— answer == ' ')
;
if(answer == 'y' —— answer == 'Y')
{
first = NO;
more = YES;
}
else
more = NO;
} /* end while loop for more iterations */
fclose(fpout); /* close the file for outputing training history */
printf("Here will open a file to save the trained network.\n");
count += oldcount; /* the total training cycles for the network */
netsave(); /* save trained network */
printf("Training is finished. The total training times is %d\n",count);
/* for(i=0; i<50; i++)
printf("%c",BEEP); Beep when finished */
} /* end of training() */
```

Appendix G

CODE FOR MODULAR NETWORK RECALL

G.1 Main Program of subrecall.c

```
/* This is the independent file for subnet recall. It uses getdat3.sub
to get recall data according the index data and weighing data from
netio2.sub. The recall process is controlled by a task file which
gives the information of which data files are used for recall and
where the recall results will go. */

#include"recal2.h"

main()
{
extern float sum(), rnum(), sigmoid();
extern void datain();
extern void subnetin();
extern void hidout();

float error = 0.0;
float time, initime, time_step;
FILE *fptask; /* to open task file */
int file, files; /* loop variables for files to processed and the
total number of files */
char fname[LENGTH]; /* io file names */
float noise; /* noise level on % to the original signal */

printf("Please response the following qusetion for your recall process.\n");

printf("Name of the recall task file > ");
scanf("%s", fname);

if((fptask = fopen(fname, "r")) == NULL)
{
printf("\n\n Can't open file %s and the program is halted.\n", fname);
exit(1);
}

files = getint(fptask);
printf("Number of files to be processed are %d.\n", files);

getstring(fptask, fname); /* read the trained network from the task file */
printf("The trainde network is %s.\n", fname);
if((fpin = fopen(fname, "r")) == NULL)
{
```

```
printf("Can't open network %s and the program is halted.\n",fname);
exit(1);
}
subnetin(fpin); /* read in the trained network */
printf("inrow = %d incol = %d\n",inrow,incol);
printf("hidrow = %d hidcol = %d\n",hidrow,hidcol);
printf("outrow = %d outcol = %d\n",outrow,outcol);
printf("tot_var = %d before selecting.\n", tot_var);
fclose(fpin);

for(file = 0; file < files; file++)
{
getstring(fptask, fname);
printf("The recall data is %s.\n", fname);
if((fpin = fopen(fname, "r")) == NULL)
{
printf("Can't open data file %s and program is halted.\n",fname);
exit(1);
}

datain(fpin,var_index);
/* read the recall data into xin[][][] */
fclose(fpin);

/*

printf("\n");
printf("The recall data file is listed below:\n\n");
for(i=0; i<npats; i++)
{
printf("PATTREN_%d:\n",i+1);
for(j=0; j<inrow; j++)
{
for(k=0; k<incol; k++)
printf("%f ",xin[i][j][k]);
printf("\n");
}
if(flag == TRAIN)
{
for(j=0; j<outrow; j++)
{
for(k=0; k<outcol; k++)
printf("%f ", tar[i][j][k]);
printf("\n");
}
} end of if block
printf("\n\n");
```

```
} end of i_loop */

/* add random noise to the recall data if required */

noise = getfloat(fptask); /* get the noise level from task file */
printf("Noise level used is %f%%.\n", noise);
if( noise > 0.0 )
noise_add(noise, xin);

/* get initial time and time step for the output file */

initime = getfloat(fptask);
time_step = getfloat(fptask);
printf("Initial time and time step are %f and %f\n.", initime,time_step);

/* calculate the recall outputs and write them out */

getstring(fptask, fname);
printf("The output file is %s.\n", fname);
if((fpout = fopen(fname, "w")) == NULL)
{
printf("Can't open network %s and the program is halted.\n",fname);
exit(1);
}

fprintf(fpout,"The network was trained for %d iterations.\n\n",count);

fprintf(fpout, "Desired output: ");
for(i=0; i<outrow; i++)
{
for(j=0; j<outcol; j++)
fprintf(fpout, "%.1f ", tar[1][i][j]);
fprintf(fpout, "\n");
}
fprintf(fpout, "\n");

/* normalize the inputs and outputs to recall information from the net.
There is no normalization for wattsbar application */

normf(xin, tar, minin, maxin, minout, maxout,weight, OLD);

time = initime;
for(ip = 0; ip < npats; ip++) /* begin loop calculation for each pattern */
{
/*
printf("\n");
printf("Recall result,target, and error for pattern %d are:\n", ip+1);
fprintf(fpout,"\n");
fprintf(fpout,"Recall result, target, and error for pattern %d is:\n",
```

```
ip+1);
/

forward(ip, w, v, h, setah, setao); /* calculate outputs forward from */
/* input layer to output layer */

/* denormalize the network outputs and target values for output. */

denorm(xin[ip], tar[ip], minin, maxin, minout, maxout, weight, NEW);
denorm(xin[ip], out, minin, maxin, minout, maxout,weight, OLD);

/*
for(i = 0; i < outrow; i++)
{
for(j = 0; j < outcol; j++)
{
if(fabs(tar[ip][i][j]) >= .0001)
error = fabs(tar[ip][i][j] - out[i][j]) / tar[ip][i][j]*100.;
else
error = tar[ip][i][j] - out[i][j];

printf("%f %f %f%%\n",out[i][j],tar[ip][i][j],error);
fprintf(fpout,"%f %f %f%%\n",out[i][j],tar[ip][i][j],error);
}
}

printf("\n");
fprintf(fpout,"\n");

*/

for(i=0; i<outrow; i++)
{
fprintf(fpout, "%3.2f ", time);
for(j=0; j<outcol; j++)
fprintf(fpout, "%.3f ", out[i][j]);
fprintf(fpout, "\n");
}

time += time_step;

} /* end of loop ip= */

fclose(fpout);

} /* end of file loop for all data files to be processed */

fclose(fptask);

printf("The recall is finished.\n");
/* for(i=0; i<50; i++)
```

```
printf("%c",BEEP); */
} /* end of main — recall() */
```

G.2 Functions Used by subrecall.c

The following is **recall.h** which includes all functions used by **subrecall.c**

```
#include<stdio.h>
#include<math.h>
#include "globdef.nn"
#include "getdat3.sub"
#include "normalizer2.nn"
#include "sigmoid.nn"
#include "matrix_sum.nn"
#include "layerout.nn"
#include "netio2.sub"
#include "task_io.h"
#include "add_noise.nn"
```

Appendix H

TASK FILES FOR RUNNING *ga_nn.c*

H.1 Task File for Running Main Grogram of *ga_nn.c*

The following is the task file required to run the program of *ga_nn.c*

f23out6.ga	output file for genetic algorithm results
f23out6.net	output file for network training results
sga.task	task file for genetic algorithms
net_gann.task	task file for training the network

H.2 Task File for Genetic Algorithms' Operation

The File is a sample task file for the genetic algorithms' operation, which supplies data necessary to run the code.

1.0	coefficient used in the objective function
4.0	coefficient used in the objective function
20	population size
22	length of chromosome = total input variables before selection
10	maximum generation to propagate
0.6	probability of crossvore
0.01	probability of mutation
0.12345	random seed

H.3 Task File for Running the Network

The following is a sample task file for running neural network, which supplies data necessary to run the code.

2	number of rows in the hidden layer
2	number of columns in the hidden layer
1	number of rows in the output layer
1	number of columns in the output layer
0.5	sogmoid slope in the sigmoid trainfer function
wb_nn.dat	training data file
0.3	learning rate for inter layer connection adjustment
0.3	learning rate for intra layer connection adjustment
0.6	momentum term coefficient
100	number of iterations to perform
0.0	initial time for the begining of the transient
0.5	time step in second for data recording
nnet.out	output of the training results

Appendix I

COMPLETE C CODE OF GENETIC ALGORITHM SEARCH FOR OPTIMAL INPUT VARIABLES FOR NEURAL NETWORKS

I.1 Main Code of the Program: ga_nn.c

```
/* This is the main program for running genetic algorithms guided neural networks */

#include<stdio.h>
#include<math.h>
#include "globdef.nn"
#include "globdef.sga"
#include "sga.h"
#include "nnet_main.c"

main()
{
int pop, old_new, first_time, new_file;
float new_net_err[MAXPOP];    /* old_net_err[] & old_net_inputs[] are */
int new_net_inputs[MAXPOP];    /* defined globally in globdef.sga */
FILE *task;
char task_file[LENGTH], ga_task[LENGTH], net_task[LENGTH];
char ga_report[LENGTH], net_report[LENGTH];
char task_flag, gadata[LENGTH];

printf("Choose the task type from below:\n");
printf("Begin a new task — B Continue the job — C\n");
printf("What to perform > ");
task_flag = getchar();
while(task_flag != 'B' && task_flag != 'b' && task_flag != 'C' &&
task_flag != 'c')
task_flag = getchar();

printf("\nInput the task file name > ");
scanf("%s", task_file);
printf("The task file nema is %s.\n", task_file);

task = fileopen(task_file, OLD);


getstring(task, ga_report);

if(task_flag == 'B' —— task_flag == 'b')
```

```
gaout = fileopen(ga_report, NEW);
else
gaout = fileopen(ga_report, APPEND);


getstring(task, net_report);

if(task_flag == 'B' —— task_flag == 'b')
netout = fileopen(net_report,NEW);
else
netout = fileopen(net_report,APPEND);

getstring(task, ga_task);
getstring(task, net_task);

fclose(task);


gen = 0;

initialize(gaout, ga_task, task_flag);    /* initialize entire GA system */

if(task_flag == 'C' —— task_flag == 'c')
{
printf("Name of the GA data file saved before > ");
scanf("%s", gadata);
getdat_ga(gadata);
statistics(oldpop);    /* calculate max., min., and average fitness */
goto CONTINUE;
}

printf("Pass initialize() call\n");

for(pop = 1; pop <= popsize; pop++)  /* initialize new population */
newpop[pop] = oldpop[pop];

first_time = YES;    /* flag for new beginning of the process */
new_file = YES;    /* flag to write global information for net output */

do
{
/* take control_c action under control by ctrlc_handler() */

if(signal(SIGINT, ctrlc_handler) == SIG_ERR)
{
```

```
perror("signal failed");
exit(0);
}


fprintf(netout,
"————— The data records for generation # %d: ————————\n",gen);
fprintf(netout,
" ========================================\n");
for(pop = 1; pop <= popsize; pop++)
{
if(first_time == NO)
{
for(i = 1; i <= popsize; i++)
{
old_new = comp_chrom(newpop[pop].chrom, oldpop[i].chrom);
if(old_new == OLD)
{
new_net_err[pop] = old_net_err[i];
new_net_inputs[pop] = old_net_inputs[i];
if(new_net_inputs[pop] > MIN_NODE)
newpop[pop].fitness = objfun(new_net_inputs[pop],
new_net_err[pop],gen);
else
newpop[pop].fitness = 0.0;
   /* to prevent NULL sized input */
goto SKIP_TRAIN;
}
}
}    /* end of if(first_time == NO) */

new_net_err[pop] = nnet(newpop[pop].chrom, net_task);
new_net_inputs[pop] = totvar;

if(new_net_inputs[pop] > MIN_NODE)
newpop[pop].fitness = objfun(new_net_inputs[pop],
new_net_err[pop],gen);
else
newpop[pop].fitness = 0.0;     /* to prevent NULL sized input */

SKIP_TRAIN: ;
record(netout, newpop[pop], new_net_err[pop],
new_net_inputs[pop], pop, new_file);
new_file = NO;
}
```

```
statistics(newpop);    /* calculate max., min., and average fitness */

for(pop = 1; pop <= popsize; pop++)
{
oldpop[pop] = newpop[pop];
old_net_err[pop] = new_net_err[pop];
old_net_inputs[pop] = new_net_inputs[pop];
}

CONTINUE: ;
generation();
report(gaout, first_time);

first_time = NO;
savedat_ga();    /* save the GA operation data for continuing next time */

} while(gen++ < maxgen);


fclose(netout);
fclose(gaout);

}    /* end of main() */
```

I.2 Codes Related to Genetic Algorithms

I.2.1 Global Definitions for Genetic Algorithms: globdef.sga

```
/* globdef.sga is the file, which contains all the global variables */
/* definition used in the simple genetic algorithm (SGA.C) program */

#define TRUE 1
#define FALSE 0
#define MAXPOP 100
#define MAXSTRING 50
#define MIN_NODE 3

typedef float real;
typedef int boolean;    /* boolean variabnle is essentially an int */
typedef int allel;    /* allel = bit positon */

struct individual {
```

```
boolean chrom[MAXSTRING];    /* Genotype = bit String */
float x;    /* Phenotype = unsigned intrger */
float fitness;    /* objective function value */
int parent1, parent2, xsite;
  /* Parents and cross site */
};


struct individual oldpop[MAXPOP], newpop[MAXPOP];
  /* Two non-overlapping populations */


int popsize, lchrom, gen, maxgen;    /* integer global variables */
float pcross, pmutation, sumfitness;    /* real global variables */
int nmutation, ncross;    /* integer statistics */
float avg, max, min;    /* real statistics */
char fname[12];    /* output file naem */
int jrand;    /* global variable for random.sga */
float oldrand[60];    /* array of 55 random #s in random.sga */
int jcross;    /* crossvore site for each function call */
int maxchrom[MAXSTRING], minchrom[MAXSTRING];
  /* arraies to store max. & min. Fitness strings */
float random_seed;    /* seed for random number generator */
float old_net_err[MAXPOP];    /* net training error */
int old_net_inputs[MAXPOP];    /* number of inputs for each net */
float c1, c2;    /* constant used in the fitness function */
FILE *gaout, *netout;    /* output files opened in ga_nn.c */
```

I.2.2 Initialization of the Genetic Algorithms: initial.sga

```
  /* initial.sga contains initdata, initpop, initreport, initialize */

void initdata(gatask)
FILE *gatask;
{

printf("-----------------------\n");
printf(" A Simple Genetic Algorithm - SGA \n");
printf(" (c) Zhichao Guo October 1991 \n");
printf(" All Rights Reserved \n");
printf("-----------------------\n");
```

```
printf("********** SGA Data Entry and Initialization **********\n\n");

c1 = getfloat(gatask);
c2 = getfloat(gatask);
printf("Constants used in objfun() are c1 = %f c2 = %f\n", c1, c2);

popsize = getint(gatask);
printf("The population size = %d\n", popsize);

lchrom = getint(gatask);
printf("The chromosome length = %d\n", lchrom);

maxgen = getint(gatask);
printf("The maximum generations = %d\n", maxgen);

pcross = getfloat(gatask);
printf("The defined crossover porbability =%f\n", pcross);

pmutation = getfloat(gatask);
printf("The defined mutation probability = %f\n", pmutation);

random_seed = getfloat(gatask);
printf("Random seed = %f\n", random_seed);
   /* initialize random number generator */
randomize(random_seed);
nmutation = 0;
ncross = 0;
}



void initreport(fp)
FILE *fp;
{


fprintf(fp, "----------------------------\n");
fprintf(fp, " A Simple Genetic Algorithm - SGA \n");
fprintf(fp, " (c) Zhichao Guo October 1991 \n");
fprintf(fp, " All Rights Reserved \n");
fprintf(fp, "----------------------------\n");

fprintf(fp, "********** SGA Data Entry and Initialization **********\n\n");
```

```
fprintf(fp, "Entered population size is %d\n", popsize);
fprintf(fp, "Entered chromosome length is %d\n", lchrom);
fprintf(fp, "Entered maximum generations is %d\n", maxgen);
fprintf(fp, "Entered crossover porbability is %f\n", pcross);
fprintf(fp, "Entered mutation probability is %f\n\n", pmutation);

   /*
fprintf(fp, "Initial Generation Statistics\n");
fprintf(fp, "==============================\n\n");

fprintf(fp, "Initial population maximum fitness = %f\n", max);
fprintf(fp, "Initial population average fitness = %f\n", avg);
fprintf(fp, "Initial population minimum fitness = %f\n", min);
fprintf(fp, "Initial population sum of fitness = %f\n", sumfitness);

/
}




void initpop()    /* initialize a population at random */
{

int j, j1;

for(j = 1; j <= popsize; j++)
{
for(j1 = 1; j1 <= lchrom; j1++)
oldpop[j].chrom[j1] = flip(0.5);    /* a fair coin toss */

   /* oldpop[j].fitness = objfun(net_inputs[j],net_error[j], c1, c2);
oldpop[j].parent1 = 0;
oldpop[j].parent2 = 0;
oldpop[j].xsite = 0; initialize print out variables */
}

}
```

```
void initialize(fout, ga_task, flag)    /* initialization coordinator */
FILE *fout;
char ga_task[], flag;
{
FILE *gatask;

gatask = fileopen(ga_task, OLD);

initdata(gatask);
initreport(fout);
if(flag == 'B' —— flag == 'b')
initpop();
   /* statistics(oldpop); */
}
```

I.2.3 Random Number Generator and Random Operator: random.sga

```
/* random.sga includes four functions, which are: */
/* advance_random: retreives a new bath of pseudorandom umbers */
/* random: rreturns asingle pseudoradom real value */
/* between 0.0 and 1.0 */
/* warmup_random: initialize random number generator */
/* flip: returns result of a simulated biased */
/* coin toss ( TRUE = head ) */
/* rnd: returns an integer edlected uniformly */
/* psuedorandomly between upper and lower limits */
/* randomize: queries terminal for user defined random */
/* seed and initialize 'random' */

void advance_random()    /* create next bath of 55 random numbers */
{
int j1;
float new_random;

for(j1 = 1; j1 <= 24; j1++)
{
new_random = oldrand[j1] - oldrand[j1+31];
if ( new_random < 0.0 ) new_random += 1.0;
oldrand[j1] = new_random;
}

for(j1=25; j1 <= 55; j1++)
{
```

```
new_random = oldrand[j1] - oldrand[j1-24];
if ( new_random < 0.0 ) new_random += 1.0;
oldrand[j1] = new_random;
}
}



void warmup_random(random_seed)
float random_seed;
{
int j1, ii;
float new_random, prev_random;

oldrand[55] = random_seed;
new_random = 1.0e-9;
prev_random = random_seed;

for(j1=1; j1<55; j1++)
{
ii = (21*j1) % 55;
oldrand[ii] = new_random;
new_random = prev_random - new_random;
if(new_random < 0.0)
new_random += 1.0;
prev_random = oldrand[ii];
}

advance_random();
advance_random();
advance_random();

jrand = 0;


}




float random()    /* fetch a single random number between 0.0 and 1.0 */
```

```
{    /* Subtractive method. See Knuth, D. (1969) for detail */

jrand += 1;

if(jrand > 55)
{
jrand = 1;
advance_random();
}

return(oldrand[jrand]);

}




boolean flip(probability)    /* flip a biased coin. TEUE if head */
float probability;
{
boolean flag;

if(probability == 1.0)
flag = TRUE;
else
flag = (random() <= probability);

return(flag);
}




int rnd(low, high)    /* pick up a random integer between low and high */
int low, high;
{
int i;

if( low >= high )
i = low;
else
{
```

```
i = (int)(random() * (high - low +1 ) + low);
if ( i > high ) i = high;
}
return(i);
}
```

```
void randomize(random_seed)
float random_seed;    /* get seed number for 'random' and start it up */
{
warmup_random(random_seed);
}
```

I.2.4 Population Statistics: stats.sga

```
/* Stats.sga: Calculate the population statistics */
void statistics(pop)
struct individual pop[];    /* structure 'individual' defined globly */
{
int j, k;

sumfitness = pop[1].fitness;    /* initialize global variables */
min = pop[1].fitness;
max = pop[1].fitness;

for(k=1; k<=lchrom; k++)
{
maxchrom[k] = pop[1].chrom[k];
minchrom[k] = pop[1].chrom[k];
}

for(j=2; j <= popsize; j++)
{
sumfitness += pop[j].fitness;    /* accumulate fitness sum */

if(pop[j].fitness > max)
{
max = pop[j].fitness;    /* new max */
```

```
for(k=1; k<=lchrom; k++)
maxchrom[k] = pop[j].chrom[k];
}
else
if(pop[j].fitness < min)
{
min = pop[j].fitness;    /* new min */
for(k=1; k<=lchrom; k++)
minchrom[k] = pop[j].chrom[k];
}
}    /* end of j loop */

avg = sumfitness / popsize;    /* average fitness */

}
```

I.2.5 Population Propagation: generate.sga

```
/* generate.sga: contains generation(). */
void generation()    /* create a new generation through select, crossover */
   /* and mutation. It assume a even number of popsize */
{
int j, k, mate1, mate2, mating_pool[MAXPOP];
int nremainder;

preselect(oldpop, mating_pool);    /* to fill up mating pool for the
new generation */
printf("Pass preselect()\n");

nremainder = popsize;
j = 1;

do {    /* select, crossover, and mutation until new population is filled */
mate1 = selection(mating_pool, nremainder);    /* pick pair of mates */
nremainder -= 1;
   /* # of remainders reduced by 1 after calling selection() */
mate2 = selection(mating_pool, nremainder);
nremainder -= 1;

   /* printf("Number of remainders are %d.\n", *nremainder);
while(mate2 == mate1) no homosexuality allowed
```

```
mate2 = selection(oldpop); */

   /* crossover and mutation — mutation is embedded within crossover */

crossover(oldpop[mate1].chrom, oldpop[mate2].chrom, newpop[j].chrom,
newpop[j+1].chrom);


   /* record parentage data on both children */

for(k=0; k<2; k++)
{
newpop[j].parent1 = mate1;
newpop[j].parent2 = mate2;
newpop[j++].xsite = jcross;
}

} while ( j < popsize);
}
```

I.2.6 Reproduction, Crossover, and Mutation Operations: tripos.sga

```
/* tripos.sga contains three operators: Reproduction(selection) */
/* Crossover(crossover), and Mutation(mutation). */

void preselect(pop, mating_pool)    /* select mates for next generation */
int *mating_pool;
struct individual *pop;
{
int j, jassign, k;
float expected, fraction[MAXPOP];

j = 0;
k = 0;

do {    /* Assign whole number */

expected = pop[++j].fitness / avg;
jassign = (int)expected;
fraction[j] = expected - jassign;
while(jassign--)
mating_pool[++k] = j;
```

```
} while(j < popsize);

j = 0;

while( k < popsize)    /* assign fractional parts */
{
if(j++ > popsize)
j = 1;

if(fraction[j] > 0.0)
{
if(flip(fraction[j]))
{
mating_pool[++k] = j;
fraction[j] -= 1.0;
}
}
}    /* end of while(k < popsize) */

}    /* end of preselect() */



int selection(mates, nremain)
int *mates, nremain;
{
extern int rnd();
int jpick, select;
jpick = rnd(1, nremain);    /* pick a mate random in total # of remainder */
select = mates[jpick];
mates[jpick] = mates[nremain];
return(select);

}



int mutation(alleleval)    /* mutate an allele w/mutation, count */
int alleleval;    /* number of nutations. */
{
int mutate, mut;
```

```
mutate = flip(pmutation);    /* flip tha biased coin */
if(mutate)
{
nmutation += 1;
if(alleleval)
mut = FALSE;
else    /* reverse a allelaval */
mut = TRUE;
}
else
mut = alleleval;    /* no reverse occured */

return(mut);
}



/* Cross 2 parent strings, place in 2 child strings */
void crossover(parent1, parent2, child1, child2)
int *parent1, *parent2, *child1, *child2;
{
int j;

if(flip(pcross))    /* do crossover with pcross */
{
jcross = rnd(1, lchrom - 1);    /* cross between 1 and l - 1 */
ncross += 1;    /* increment crossover counter */
}
else    /* otherwise set cross site to force mutation */
jcross = lchrom;

for(j = 1; j <= jcross; j++)    /* 1st, exchange 1 to 1 and 2 to 2 */
{
child1[j] = mutation(parent1[j]);
child2[j] = mutation(parent2[j]);
}

if(jcross != lchrom)    /* skip if cross site is lchrom — no cross */
{
for(j=jcross+1; j <= lchrom; j++)
{
child1[j] = mutation(parent2[j]);
child2[j] = mutation(parent1[j]);
```

```
}
}
}
```

I.2.7 Identify Duplicated Individuals: comp_chrome.sga

```
int comp_chrom(string1, string2)
int string1[], string2[];
{
int flag = OLD, i;

for(i=1; i<=lchrom; i++)
{
if(string1[i] != string2[i])
{
flag = NEW;
break;
}
}

return(flag);
}
```

I.2.8 Save and Retrieve Information Necessary for Continuing Search After Interruption: ga_io.sga

```
void savedat_ga()
{
int i, j;
char gadata[LENGTH];
FILE *ga;

printf("The GA process data will be saved automatically before quit.\n");
   /* printf("Name chosen for the file > ");
scanf("%s", gadata);
printf("\n\n");

ga = fileopen(gadata, NEW); */
```

```
if((ga=fopen("temp_dat.ga", "a")) == NULL)
{
printf("Can't open temp_dat.ga, the program halted.\n");
exit(1);
}

printf("The GA data file was saved as temp_dat.ga for further process.\n");
fprintf(ga, "%d ", gen);     /* generation index */
fprintf(ga, "%d\n", lchrom);     /* length of chromesome */
for(i = 1; i<= popsize; i++)
{
for(j=1; j<=lchrom; j++)
fprintf(ga,"%d ", oldpop[i].chrom[j]);
fprintf(ga, "%f ", oldpop[i].fitness);
fprintf(ga, "%d ", old_net_inputs[i]);
fprintf(ga, "%f\n", old_net_err[i]);
}

fclose(ga);

}



void getdat_ga(gadata)
char gadata[];
{
int i, j;
FILE *ga;

ga = fileopen(gadata, OLD);

fscanf(ga, "%d%d", &gen, &lchrom);

for(i=1; i<= popsize; i++)
{
for(j=1; j<=lchrom; j++)
fscanf(ga, "%d", &oldpop[i].chrom[j]);
fscanf(ga, "%f%d%f", &oldpop[i].fitness, &old_net_inputs[i],
&old_net_err[i]);
}

fclose(ga);
}
```

I.2.9 Interruption Control of the Process: ctrlc_handler

```
/* This function takes control of interruption signal, ctrlc, and allows
user to save the information necessary for continuing next time */
void ctrlc_handler(sig)
int sig;
{
int c;

signal(SIGINT, SIG_IGN);

while((c = getchar()) != '\n')
;

printf("\nInterrupted. Quit? Y or N > ");
c = getchar();

if(c == 'Y' —— c == 'y')     /* if ctrl-c interrupted, */
{
printf("\nThe output files will be saved here before quit.\n");
savedat_ga();
fclose(netout);
fclose(gaout);
exit(0);
}

back_again = 1;
signal(SIGINT, ctrlc_handler);
}
```

I.3 Codes Related to the Lateral Feedback Neural Network

I.3.1 Main Code of the Network: net_main.c

```
/*************************************************************************/
hspace1mm
```

```
/* This is the main code of the lateral feedback neural network */
/* connected to the operations of the genetic algorithms. */
/* The modification is made in the hidden layer units. Every neuron */
/* in the hidden layer uses information not only from forwarding infor-*/
/* mation from the input layer but also from feedback signals of its */
/* physical neighbors and itself. All the feedbacks considered here */
/* are negative feedback except the self feedback, which is the posii- */
/* tive feedback to the neuron its self. Momentum terms for all the */
/* adjustable terms including weights and threshold values, are intro-*/
/* duced to the program to speed up the learning process. Sigmoid slop*/
/* is also used in the sigmoid transfer function to improve to coding */
/* ability. The normalization is changed to the way of normalize the */
/* data file column by column for both input and output varibales */
/* instead of using one maximum value to normalize the entire data file*/
/* including both inputs and outputs. The normalization range for */
/* input is [-1, 1], and for output is [0.1 0.9]. */
/* */
/*************************************************************************/
hspace1mm


#include "nnet.h"
#include "train.nn"
#include "recall.nn"

float nnet(index, task_file)
int index[];
char task_file[];
{
float avg_error;
FILE *task;

printf("The task file is: %s\n ", task_file);
task = fileopen(task_file, OLD);

avg_error = training(task, index);

printf("The pattern average error is %f%%\n", avg_error);

fclose(task);

return(avg_error);

}    /* finished nnet() */
```

I.3.2 Header Files for the Network: nnet.h

```
#include "netsetup.nn"
#include "openfile.nn"
#include "getdata.nn"
#include "normalizer.nn"
#include "randomizer.nn"
#include "sigmoid.nn"
#include "matrix_sum.nn"
#include "layerout.nn"
#include "errorcal.nn"
#include "chweights.nn"
#include "netio.nn"
#include "get_par.nn"
```

I.3.3 Global Definitions of the Network: globdef.nn

```
#define MAXPAT 600    /* Maximum # of patterns in a training file */
#define MAXNODE 20    /* for functions used for all layers */
#define MAX_IN_ROW 3    /* maximum # rows in the input layer */
#define MAX_IN_COL 19    /* maximum # rows in the input layer */
#define MAX_HID_ROW 6    /* maximum # rows in the hidden layer */
#define MAX_HID_COL 6    /* maximum # rows in the hidden layer */
#define MAX_OUT_ROW 3    /* maximum # rows in the output layer */
#define MAX_OUT_COL 3    /* maximum # rows in the output layer */
#define LENGTH 20    /* The max name length of a file */
#define SCALE 2147483646.0    /* Used for random number generastor */
#define YES 1
#define NO 0
#define NEW 1    /* set a new open file to "w" status */
#define OLD 0    /* set a lod file to "r" status */
#define APPEND -1    /* flag for opening file */
#define TRAIN 1    /* set training flag */
#define RECALL 0    /* set recall flag */
#define RETRAIN 0    /* set retrain flag */
#define UP_XLIMIT 1.0    /* normalization region of inputs is [-1, 1] */
#define LOW_XLIMIT -1.0
#define UP_YLIMIT 0.9    /* normalization region of outputs is [.1 .9] */
#define LOW_YLIMIT 0.1
#define SEED 12345    /* seed for random number generator */
#define CYCLE 10    /* Maximum value for iteration training loop */
#define NRUN 100    /* Number of iterations for a set of training */
  /* pattern. After each set, check convergence. */
```

```
   /* if not converge, process one more uset of */
   /* training till convergence. */

int inrow, incol;    /* #PEs in rows and columns of input layer */
int hidrow, hidcol;    /* #PEs in rows and columns of hidden layer */
int outrow, outcol;    /* #PEs in rows and columns of output layer */
float sslope;    /* sigmoid slope used in the sigomid function */
   /* f(x) = 1. / (1. + exp(-x/sslpoe)) */

int i, j, k, l, m, n;    /* loop variables */
int ip, ir, ic;    /* loop variables */
char filename[LENGTH];    /* to store a input file name */
FILE *fpin, *fpout;    /* File pointer points to a input or output file */
FILE *net;    /* File pointer for saving weights */
int npats;    /* number of patterns in the input file */
int times, count = 1 ;    /* iteration times for training */
int flag;
int jump;    /* interval for display and write out result */
int more, first;    /* determine more iteration */
float total = 0;    /* tempary space to store a summation */
float target, alpha, beta, alpham;
char answer;    /* control variable for more iterations */
   /* float max; the max. target value used to normize all target values */
int maxp = 0;    /* the pattern with max. target value */

float xin[MAXPAT][MAX_IN_ROW][MAX_IN_COL];    /* store input datd */

float yhid[MAX_HID_ROW][MAX_HID_COL];    /* store the activations of */
   /* PEs in the 2nd layer */
float w[MAX_HID_ROW][MAX_HID_COL][MAX_IN_ROW][MAX_IN_COL];
   /* store weights between 1st and 2nd layers */
float v[MAX_OUT_ROW][MAX_OUT_COL][MAX_HID_ROW][MAX_HID_COL];
   /* store connections within 2nd layer */
float tar[MAXPAT][MAX_OUT_ROW][MAX_OUT_COL];
   /* store targets in input file */
float out[MAX_OUT_ROW][MAX_OUT_COL];
   /* store the activation of PEs in 2nd layer */
float setah[MAX_HID_ROW][MAX_HID_COL];
   /* threshold values of the 2nd layer */
float setao[MAX_OUT_ROW][MAX_OUT_COL];
   /* threshold values of the 3nd layer */
float h[MAX_HID_ROW][MAX_HID_COL][9];
   /* lateral connections of 2nd layer */
float tempsum[MAX_HID_ROW][MAX_HID_COL];    /* tempearl activation of PEs in
*/
```

```
  /* 2nd layer before and after the */
float yhidtemp[MAX_HID_ROW][MAX_HID_COL];    /* transfer function */

float yfb[9];    /* feedback signals to a PE in the hidden layer */

float minin[MAX_IN_ROW][MAX_IN_COL];    /* min. values of inputs */
float maxin[MAX_IN_ROW][MAX_IN_COL];    /* max. values of inputs */
float minout[MAX_OUT_ROW][MAX_OUT_COL];    /* min. values of outputs */
float maxout[MAX_OUT_ROW][MAX_OUT_COL];    /* max. values of outputs */

float dwold[MAX_HID_ROW][MAX_HID_COL][MAX_IN_ROW][MAX_IN_COL];
float dvold[MAX_OUT_ROW][MAX_OUT_COL][MAX_HID_ROW][MAX_HID_COL];
float dsetahold[MAX_HID_ROW][MAX_HID_COL];
float dsetaoold[MAX_OUT_ROW][MAX_OUT_COL];
float dhold[MAX_HID_ROW][MAX_HID_COL][9];

float weight[MAX_IN_ROW*MAX_IN_COL];    /* weighing data to modify input data */
int var_index[MAX_IN_ROW*MAX_IN_COL][2];   /* Guide to select input variables */
int tot_var;    /* total number of input variables before selection */
int totvar;    /* total number of input variables after selection */

float sum();    /* sum two 2_D matrix term by term product */
float rnum();    /* random number generator */
void random4();    /* randomize a 4D array */
void random2();    /* randomize a 2D array */
FILE *fileopen();    /* open a file and return the file pointer */
float sigmoid();    /* sigmoid transfer function */
float training();    /* training function for the training process */
float recall();    /* recall function for the recall process */
void datain();    /* read a data file for training or recall */
void netsave();    /* save a trained network */
void netin();    /* retraive a trained network */
void setup();    /* to set the structure of the network */
void chwts();    /* replace old weights by new ones */
float energyf();    /* calculate the error of the 2nd layer */
void chseta();    /* replace old threshold values by new values */
void normf();    /* normize the data file by its max. value */
void randomh();    /* randomize the lateral connections of 2nd layer */
void hidout();    /* outputs of PEs in 2nd layer after lateral feedback */
float maxerf();    /* find the max. error in output units */
float merrorf();    /* find the max. error in outputs of all patterns */
void forward();    /* calculate output forward from input to output layer*/
void denorm();    /* denormalize the data file */
void typeout();    /* print something on the screen */
void getw();    /* get weighing data to modify the input data */
```

```
void ctrlc_handler();    /* task over the control-c interruption */
float getfloat();
int getint();
void getstring();
void record();    /* record for ga_nn on writting net training information */

int back_again = 0;    /* used with ctrlc_handler() */
```

I.3.4 Network Training Function: train.nn

```
/* The following function is the training function for net_main.c
The change from nettrain.nn is to invoke a index file for the selecting of
input variables because the subnet only use part of the inputs as its
training inputs. The nettrain.nn will use all the inputs as its inputs
without any selection */

float training(task, index)
FILE *task;
int index[];
{
float energy1 = 0, energy2 = 0;    /* old f new energy */
float wtemp[MAX_HID_ROW][MAX_HID_COL][MAX_IN_ROW][MAX_IN_COL];
float vtemp[MAX_OUT_ROW][MAX_OUT_COL][MAX_HID_ROW][MAX_HID_COL];
float setahtemp[MAX_HID_ROW][MAX_HID_COL];
float setaotemp[MAX_OUT_ROW][MAX_OUT_COL];
float htemp[MAX_HID_ROW][MAX_HID_COL][9];
float hiderr[MAX_HID_ROW][MAX_HID_COL];
float dw[MAX_HID_ROW][MAX_HID_COL];
float dv[MAX_OUT_ROW][MAX_OUT_COL];
float dntar, dnout;    /* denormalized target and output values for error
analysis */
float temp = 0.0;
float net_error = 0.0;    /* return training error to main() */
char fname[LENGTH];
int iterations;    /* total number of iterations to perform */

totvar = 0;    /* total number of variables after selection */


/* count the total number of input variables selected */

tot_var = lchrom;    /* total variables before the selection */

printf("The index file is listed below:\n\n");
```

```
for(i=0; i<tot_var; i++)
{
var_index[i][0] = i;
var_index[i][1] = index[i+1];
if(var_index[i][1] == 1)
totvar += 1;
}

printf("Total selected variables are %d\n", totvar);


setup(task, totvar);    /* setup the network configuration */
printf("The network configuration is:\n");
printf("\n inrow = %d incol = %d", inrow, incol);
printf("\n hidrow = %d hidcol = %d", hidrow, hidcol);
printf("\n outrow = %d outcol = %d\n", outrow, outcol);

getstring(task, fname);    /* get training file name from task file */
printf("input training file is %s.\n", fname);
fpin = fileopen(fname, OLD);    /* open a input training file */
datain(fpin,var_index);
  /* read the input data into xin[][][] & tar[][][] */


normf(xin, tar, minin, maxin, minout, maxout, NEW);
  /* normalize the data column by column by their own
min. and max. values */
printf("The maximum and minimum input values are listed below:\n");

typeout(inrow, incol, minin, maxin);

printf("The maximum and minimum target values are listed below:\n");

typeout(outrow, outcol, minout, maxout);

srand(SEED);
printf("The random seed used is %d\n", SEED);

rand_w();    /* randomize connection weights between 1st and 2nd layer */
rand_v();    /* randomize connection weights between 2nd and 3rd layer */
rand_setah();    /* randomize bias terms for the hidden layer */
rand_setao();    /* randomize bias terms for the output layer */
randomh(h, hidrow, hidcol);    /* randomize lateral connections */
```

```
/* initialize the momentum terms all equal 0.0 */

for(m = 0; m < hidrow; m++)
for(n = 0; n < hidcol; n++)
{
dsetahold[m][n] = 0.0;
for(k = 0; k < inrow; k++)
for(l = 0; l < incol; l++)
dwold[m][n][k][l] = 0.0;

for(k=0; k < 9; k++)
dhold[m][n][k] = 0.0;
}


for(m = 0; m < outrow; m++)
for(n = 0; n < outcol; n++)
{
dsetaoold[m][n] = 0.0;
for(k = 0; k < hidrow; k++)
for(l = 0; l < hidcol; l++)
dvold[m][n][k][l] = 0.0;
}


count = 1;    /* the # of cycles for more training */


/* get training parameters from the task file */

alpha = getfloat(task);
beta = getfloat(task);
alpham = getfloat(task);

printf("learn rates of alpha, beta & alpham : %f %f %f\n ",
alpha, beta, alpham);


/* get iterations required and display jumps from the task file */

times = getint(task);
printf("iterations to perform is %d.\n", times);

iterations = times;
```

```
while (iterations--)    /* Begin times interation */
{

for(ip = 0; ip < npats; ip++)
   /* begin loop calculation for each pattern */
{

forward(ip, w, v, h, setah, setao);    /* cal. outputs forward from
input layer to output layer */

/* adjust weights between 1st & 2nd layer */


for(m = 0; m < hidrow; m++)
for(n = 0; n < hidcol; n++)
{
total = 0.;
for(i = 0; i < outrow; i++)
for(j = 0; j < outcol; j++)
total += (tar[ip][i][j] - out[i][j]) * out[i][j] *
(1. - out[i][j]) * v[i][j][m][n];

total /= sslope;    /* counting sigmoid slope */

total *= yhid[m][n] * (1. - yhid[m][n]) / sslope;
hiderr[m][n] = total;
dw[m][n] = total*(1.+ h[m][n][4]*yhidtemp[m][n]*
(1.- yhidtemp[m][n]) / sslope);

setahtemp[m][n] = setah[m][n] - alpha * dw[m][n] +
alpham*dsetahold[m][n];

}    /* end of n_loop, m_loop, l_loop and k_loop */


for(k = 0; k < hidrow; k++)
for(l = 0; l < hidcol; l++)
for(m = 0; m < inrow; m++)
for(n = 0; n < incol; n++)
wtemp[k][l][m][n] = w[k][l][m][n] + alpha *
dw[k][l]* xin[ip][m][n] + alpham * dwold[k][l][m][n];

/* adjust lateral connections of 2nd layer PEs */
```

```
for(i=0; i<hidrow; i++)
for(j=0; j<hidcol; j++)
{
n = 0;
for(l=i-1; l<i+2; l++)
for(m=j-1; m<j+2; m++)
{
if((l>=0) && (m>=0) && (m<hidcol) && (l<hidrow))
yfb[n] = yhidtemp[l][m];
else
yfb[n] = 0.0;
n += 1;
}
for(k=0; k<9; k++)
{
if(h[i][j][k] != 0.0)
{
if(k != 4)
htemp[i][j][k] = h[i][j][k] - beta*hiderr[i][j]*yfb[k]
+ alpham*dhold[i][j][k];
else
htemp[i][j][k] = h[i][j][k] + beta*hiderr[i][j]*yfb[k]
+ alpham*dhold[i][j][k];
}
}    /* end of k-loop */
}    /* end of j-loop and i-loop */

/* adjust weights between the 2nd layer and the 3nd layer */

for(i = 0; i < outrow; i++)
for(j = 0; j < outcol; j++)
{
dv[i][j] = out[i][j] / sslope *(1.- out[i][j])*
(tar[ip][i][j]-out[i][j]);
setaotemp[i][j] = setao[i][j] - alpha * dv[i][j] +
alpham*dsetaoold[i][j];
}


for(i = 0; i < outrow; i++)
for(j = 0; j < outcol; j++)
for(k = 0; k < hidrow; k++)
for(l = 0; l < hidcol; l++)
vtemp[i][j][k][l] = v[i][j][k][l] +
alpha*dv[i][j]*yhid[k][l] + alpham*dvold[i][j][k][l];
```

```
/* calculate the outputs after the weight adjustments

forward(ip, wtemp, vtemp, htemp, setahtemp, setaotemp);

energy2 = energyf(out, tar[ip]); total error */
   /* of the system after the weight adjustments */
chwts(wtemp, vtemp, htemp);    /* accept the weight adjustments */
chseta(setahtemp, setaotemp);  /* accept the threshold adjustments */



}    /* end of the loop for the variable of ip (input pattern) */

count++;

if( count%100 == 0 )
{
printf("The iteration has been performed %d times\n\n", count);
printf("The training can be interrupted by hitting ctrl-c\n");
printf("and the network will be saved before quit.\n");
}


}    /* end times iteration */


net_error = recall(task);
   /* perform recall after times iterations and return recall error
which will be used by sga.c to perform genetic algorithm
operations */


printf("Training is finished. The total training times is %d\n",count);

return (net_error);

}    /* end of training() */
```

I.3.5 Network Recall Function Used Within train.nn: recall.nn

```
   /* This is the recall function for net_main.c used within train.nn. */

float recall(task)
FILE *task;
{
float error = 0.0, err[MAXPAT][MAX_OUT_ROW][MAX_OUT_COL];
float time, initime, time_step;
char fname[LENGTH];

   /* get initial time and time step for the output file */

initime = getfloat(task);
time_step = getfloat(task);

printf("Initial time and time step are %f and %f\n", initime,time_step);

   /* calculate the recall results and write them out */

getstring(task, fname);
printf("save the recall results to %s.\n", fname);

fpout = fileopen(fname, NEW);     /* open a file to save recall results */

fprintf(fpout,"The network was trained for %d iterations.\n\n",count);
fprintf(fpout,"The network configuration is (%dx%d)(%dx%d)(%dx%d)\n",
inrow, incol, hidrow, hidcol, outrow, outcol);
fprintf(fpout, "The learning rate of alpha = %f\n", alpha);
fprintf(fpout, "The learning rate of beta = %f\n", beta);
fprintf(fpout, "The learning rate of alpham= %f\n", alpham);
fprintf(fpout, "Total number of recall patterns are %d\n", npats);

fprintf(fpout, "Desired output: ");
for(i=0; i<outrow; i++)
{
for(j=0; j<outcol; j++)
fprintf(fpout, "%.1f ", tar[1][i][j]);
fprintf(fpout, "\n");
}
fprintf(fpout, "\n");

fprintf(fpout, "Pattern Target Output Error(%%)\n\n");

for(ip = 0; ip < npats; ip++)     /* begin loop calculation for each pattern */
```

```
{
forward(ip, w, v, h, setah, setao);    /* calculate outputs forward from */
   /* input layer to output layer */

   /* The normalized target values are .9 or .1 insterd of 1.0 and 0.0 after
denormalization. Target value of 0.0 will cause problems in the error
calculation. The denormalization process can be performed after the error
calculation. */


for(i = 0; i < outrow; i++)
for(j = 0; j < outcol; j++)
{
err[ip][i][j] = fabs(tar[ip][i][j] - out[i][j]) /
tar[ip][i][j]*100.;
error += err[ip][i][j];
}


   /* denormalization for output */

denorm(xin[ip], tar[ip], minin, maxin, minout, maxout);
denorm(xin[ip], out, minin, maxin, minout, maxout);

for(i=0; i<outrow; i++)
{
fprintf(fpout,"%d ", ip);
for(j=0; j<outcol; j++)
fprintf(fpout, "%.3f %.3f %.3f", tar[ip][i][j],
out[i][j], err[ip][i][j]);
fprintf(fpout, "\n");
}

}    /* end of loop ip= */

fprintf(fpout, "\nThe pattern average recall error is %f\n",
error/(float)npats);
fclose(fpout);

printf("The recall is finished.\n");

return(error/(float)npats);    /* return pattern average error to the training
program and will be used later by sga.c to
perform genetic algorithm operations */
```

```
}    /* end of recall() */
```

I.3.6 Openning a File and Return File Pointer: openfile.nn

```
   /* open a file and return the file pointer */
FILE *fileopen(filename,flag)
char filename[];
int flag;
{
FILE *fp;    /* file pointer */


if(flag == NEW)
fp = fopen(filename,"w");    /* open a new file */
else
if(flag == OLD)
fp = fopen(filename,"r");    /* open the input file */
else
fp = fopen(filename, "a");


if(fp == NULL)
{
printf("Can't open %s\n", filename);
exit(1);
}
return (fp);
}
```

I.3.7 Read Data From a Task File: get_par.nn

```
   /* This function performs reading data from a data file, in which each line
contains a data and the discriptipn of the data */

int getint(fp)
FILE *fp;
{
int data;
char dummy;

fscanf(fp, "%d", &data);
```

```
fscanf(fp, "%c", &dummy);
while(dummy != '\n')
fscanf(fp, "%c", &dummy);

return(data);
}



void getstring(fp, string)
FILE *fp;
char string[];
{
char dummy;

fscanf(fp, "%s", string);
fscanf(fp, "%c", &dummy);
while(dummy != '\n')
fscanf(fp, "%c", &dummy);

}



float getfloat(fp)
FILE *fp;
{
float data;
char dummy;

fscanf(fp, "%f", &data);
fscanf(fp, "%c", &dummy);
while(dummy != '\n')
fscanf(fp, "%c", &dummy);

return(data);
}
```

I.3.8 Read Training Data: getdata.nn

```
/* function for reading a input data file for training or recall */
```

```
void datain(fp, index)
FILE *fp;    /* points to a opened input file */
int index[][2];    /* index for selecting variables */
{
int ip, ir, ic, or, oc;    /* loop varibales */
int inr, inc, outr, outc;    /* store the pattern dim. from data file */
char type[LENGTH];    /* store Train or Recall in the 1st line of the file */
int item;    /* # of items in the head fo the data file */
int temp;
float temp2[MAX_IN_ROW * MAX_IN_COL];
int low_p[MAX_IN_ROW * MAX_IN_COL];
int up_p[MAX_IN_ROW * MAX_IN_COL];
  /* beginning and ending of patterns with target value
1.0, and beyond p1 and p1 all target values are 0.0 */


if(fscanf(fp, "%s", type) == 0)
{
printf("The 1st line of the file must type Train or Recall.\n\n");
exit(1);
}

if(type[0] == 't' —— type[0] == 'T')
{
flag = TRAIN;
item = 5;
}
else
{
flag = RECALL;
item = 3;
}


for(i = 0; i < item; i++)    /* read in the # for the patternsd rows */
{   /* and columns in the input file */
if(fscanf(fp, "%d", &temp) == 0 )
{
printf("The integers representing # of patterns, nodes in row\n");
printf("and in column must be put in the begining of the file.\n");
exit(1);
}
switch(i)
{
```

```
case 0: npats = temp;
break;
case 1: inr = temp;
break;
case 2: inc = temp;
break;
case 3: outr = temp;
break;
case 4: outc = temp;
break;
}
}    /* end of i loop */

if(flag == TRAIN)
{
for(i=0; i<outrow*outcol; i++)
{
if(fscanf(fp, "%d%d", &low_p[i], &up_p[i]) == 0)
{
printf("Lower and upper bounders for %d scenarios have to be given\n",
outrow*outcol);
exit(1);
}
}    /* end of i loop */
}    /* end of if(flag == TRAIN) */


printf("number of pattern in the file is %d\n",npats);

for( ip = 0; ip < npats; ip++)    /* read into the training data */
{
for( ir = 0; ir < inr; ir++)    /* from the input file */
for( ic = 0; ic <inc; ic++)
if(fscanf(fp,"%f", &xin[ip][ir][ic]) == EOF)
{
printf("End of File is reached in the file: %s\n", filename);
exit(2);
}
if(flag == TRAIN)
{
for( or = 0; or < outr; or++)    /* from the input file */
for( oc = 0; oc < outc; oc++)
if(fscanf(fp,"%f", &tar[ip][or][oc]) == EOF)
{
printf("End of File is reached in the file: %s\n", filename);
```

```
exit(2);
}
}    /* end of (flag == TRAIN) */

i = 0;

for(ir = 0; ir < inr; ir++)
for(ic = 0; ic < inc; ic++)
{
if(index[ir*inc+ic][1] == 1)
temp2[i++] = xin[ip][ir][ic];
xin[ip][ir][ic] = 0.0;
}


for(ir = 0; ir < inrow; ir++)
for(ic = 0; ic < incol; ic++)
xin[ip][ir][ic] = temp2[ir*incol+ic];


for(ir = 0; ir < outrow; ir++)
for(ic = 0; ic < outcol; ic++)
tar[ip][ir][ic] = 0.0;    /* initialize targets */

}    /* end of ip loop */


   /* The following program segment will assign 1 0 ... 0 as the target value
for scenario # 1 bounded by low_p[0] and up_p[0], 0 1 0 ... 0 to scenario
# 2 bounded by low_p[1] and up_p[1], so on and so forth. For the patterns
not covered by low_p[] and up_p[] have been assigned 0 0 ... 0 as their
target values by the above target initialization process. */

if(flag == TRAIN)
{
for(i=0; i<outrow; i++)
for(j=0; j<outcol; j++)
{
for(ip = low_p[i*outcol+j]; ip<up_p[i*outcol+j]; ip++)
{
for(k=0; k<outrow; k++)
for(l=0; l<outcol; l++)
{
if(k == i && l == j)
tar[ip][k][l] = 1.0;
```

```
else
tar[ip][k][l] = 0.0;
}
}    /* end of ip loop */

}    /* end of i, j loop */
}    /* end of (flag == TRAIN) */


if(fclose(fp)!= 0 )    /* close input training file */
printf("Error in closing input training file.\n");

}    /* end of datain() */
```

I.3.9 Network Setup: netsetup.nn

```
  /* This function performs setting up the 2-D sturcture of the network according
to inputs selected by the genetic algorithms and supplied data */

void setup(task,tot_inputs)
FILE *task;
int tot_inputs;
{
  /* set up the input layer for total number of neurons = tot_inputs */
if(tot_inputs % 3 == 0)
{
inrow = 3;
incol = tot_inputs / 3;
}
else
if(tot_inputs % 2 == 0)
{
inrow = 2;
incol = tot_inputs / 2;
}
else
{
inrow = 1;
incol = tot_inputs;
}


  /* get hiddern, output layer dimensions and sigmoid slop from the task file */
```

```
hidrow = getint(task);
hidcol = getint(task);
outrow = getint(task);
outcol = getint(task);
sslope = getfloat(task);

}    /* end of setup() */
```

I.3.10 Randomize Network Weights Initially: random_weights.nn

```
   /* These functions perform random number generating, and randomizing
connection of the network initially. */

   /* random number generator */
float rnum()
{
   /* extern int rand(); */
float num = 0.0;
num = ((float)rand()) / (SCALE - 1.0);
return (num);
}




   /* randomize the connections between 1st and 2nd layer */

void rand_w()
{
for(i=0; i<hidrow; i++)
for(j=0; j<hidcol; j++)
for(k=0; k<inrow; k++)
for(l=0; l<incol; l++)
w[i][j][k][l] = rnum();
}




   /* randomize the connections between 2nd and 3rd layer */
```

```
void rand_v()
{
for(i=0; i<outrow; i++)
for(j=0; j<outcol; j++)
for(k=0; k<hidrow; k++)
for(l=0; l<hidcol; l++)
v[i][j][k][l] = rnum();
}
```

```
   /* randomize the threshold values of 2nd */

void rand_setah()
{
for(k=0; k<hidrow; k++)
for(l=0; l<hidcol; l++)
setah[k][l] = rnum();
}
```

```
   /* randomize the threshold values of 3nd */

void rand_setao()
{
for(k=0; k<outrow; k++)
for(l=0; l<outcol; l++)
setao[k][l] = rnum();
}
```

```
   /* randomize the lateral connections of 2nd layer PEs */

void randomh(hc, row, col)
float hc[][MAX_HID_COL][9];
int row, col;
{
extern float rnum();
```

```
for(i=0; i<row; i++)
for(j=0; j<col; j++)
{
k=0;
for(l=i-1; l<i+2; l++)
for(m=j-1; m<j+2; m++)
{
if((l>=0) && (m>=0) && (m<col) && (l<row))
hc[i][j][k] = rnum();
else
hc[i][j][k] = 0.0;

k += 1;
}
}
}
```

I.3.11 Normalization Functions: normalizer.nn

```
   /* NORMALIZE.NN: contains normf(), normold(), & denorm() functions */
   /* for normalization and denormalizations of inputs and outputs */
   /* The formula used for normalization is: */
   /* y = (x - x_min)*(Upper_limit-Lower_limit)/(x_max - x_min) + */
   /* Lower_limit. */
   /* where: */
   /* lower_limit = -1.0 for input, and 0.1 for output */
   /* upper_limit = 1.0 for input, anf 0.9 for output */
   /* */
   /* _min and _max are min. and max. values of inputs or outputs. */


   /* Scaning the training input file and finding the min. and max.
*/
   /* values in for each input and target. */

void normf(x, y, minx, maxx, miny, maxy, flag)
float x[][MAX_IN_ROW][MAX_IN_COL];    /* network input */
float y[][MAX_OUT_ROW][MAX_OUT_COL];    /* network target */
float minx[][MAX_IN_COL], maxx[][MAX_IN_COL], miny[][MAX_OUT_COL],
maxy[][MAX_OUT_COL];
int flag;
{
```

```
int k, i;
float delta;


if(flag == NEW)
{

   /* find maximum and minimum values for each input and output variable */
for(i=0; i<inrow; i++)
for(j=0; j<incol; j++)
{
maxx[i][j] = x[0][i][j];
minx[i][j] = x[0][i][j];
}

for(i=0; i<outrow; i++)
for(j=0; j<outcol; j++)
{
maxy[i][j] = y[0][i][j];
miny[i][j] = y[0][i][j];
}

for(i=0; i<inrow; i++)
for(j=0; j<incol; j++)
for(k=0; k<npats; k++)
{
if(x[k][i][j] > maxx[i][j])
maxx[i][j] = x[k][i][j];
else
if(x[k][i][j] < minx[i][j])
minx[i][j] = x[k][i][j];
}



for(i=0; i<outrow; i++)
for(j=0; j<outcol; j++)
for(k=0; k<npats; k++)
{
if(y[k][i][j] > maxy[i][j])
maxy[i][j] = y[k][i][j];
else
if(y[k][i][j] < miny[i][j])
miny[i][j] = y[k][i][j];
```

```
}

}    /* end of if(flag == NEW) */

   /* Normalization of input and target values */

for(i=0; i<inrow; i++)
for(j=0; j<incol; j++)
for(k=0; k<npats; k++)
{
delta = maxx[i][j] - minx[i][j];
if(fabs(delta) < 1.0e-5)
{
if(fabs(maxx[i][j]) > 1.0e-5)
x[k][i][j] = x[k][i][j] / maxx[i][j];
}
else
x[k][i][j]= (x[k][i][j] - minx[i][j])*
(UP_XLIMIT - LOW_XLIMIT) / delta + LOW_XLIMIT;
}


for(i=0; i<outrow; i++)
for(j=0; j<outcol; j++)
for(k=0; k<npats; k++)
{
delta = maxy[i][j] - miny[i][j];
if(fabs(delta) < 1.0e-5)
{
if(fabs(maxy[i][j]) > 1.0e-5)
y[k][i][j] = y[k][i][j] / maxy[i][j];
}
else
y[k][i][j]= (y[k][i][j] - miny[i][j])*
(UP_YLIMIT - LOW_YLIMIT) / delta + LOW_YLIMIT;
}


}    /* end of normf() */
```

```
   /* denormalize the normalized file back to its original values */

void denorm(x, y, minx, maxx, miny, maxy)
float x[][MAX_IN_COL], y[][MAX_OUT_COL];
float minx[][MAX_IN_COL], maxx[][MAX_IN_COL],
miny[][MAX_OUT_COL], maxy[][MAX_OUT_COL];
{
int j, i;
float delta;

   /* Denormalization of input and target values */

for(i=0; i<inrow; i++)
for(j=0; j<incol; j++)
{
delta = maxx[i][i] - minx[i][j];
if(fabs(delta) < 1.0e-5)
{
if(fabs(maxx[i][j]) > 1.0e-5)
x[i][j] = x[i][j] * maxx[i][j];
}
else
x[i][j] = (x[i][j] - LOW_XLIMIT)* delta
/ (UP_XLIMIT - LOW_XLIMIT) + minx[i][j];
}

for(i=0; i<outrow; i++)
for(j=0; j<outcol; j++)
{
delta = maxy[i][i] - miny[i][j];
if(fabs(delta) < 1.0e-5)
{
if(fabs(maxy[i][j]) > 1.0e-5)
y[i][j] = y[i][j] * maxy[i][j];
}
else
y[i][j] = (y[i][j] - LOW_YLIMIT)* delta
/ (UP_YLIMIT - LOW_YLIMIT) + miny[i][j];
}

}
```

```
void typeout(row, col, min, max)
int row, col;
float min[][MAXNODE], max[][MAXNODE];
{
int i, j;

for(i=0; i<row; i++)
{
for(j=0; j<col; j++)
printf("%f ", min[i][j]);
printf("\n");

for(j=0; j<col; j++)
printf("%f ", max[i][j]);
printf("\n\n");
}
}
```

I.3.12 Calculation of Activations of Hidden and Output Layer: layerout.nn

```
  /* calculate the activations of 2nd layer PEs after lateral feedback */

void hidout(sumt, yhidt, h, yhid, row, col)
float (*sumt)[MAX_HID_COL], (*yhidt)[MAX_HID_COL];
float (*h)[MAX_HID_COL][9], (*yhid)[MAX_HID_COL];
int row, col;
{
extern float sigmoid();

for(i=0; i<row; i++)
for(j=0; j<col; j++)
{
n = 0;
for(l=i-1; l<i+2; l++)
for(m=j-1; m<j+2; m++)
{
if((l>=0) && (m>=0) && (m<col) && (l<row))
yfb[n] = yhidt[l][m];
else
yfb[n] = 0.0;
```

```
n += 1;
}

total = 0.0;
for(k=0; k<9; k++)
{
if(k != 4)
total += h[i][j][k] * yfb[k];
else
total -= h[i][j][k] * yfb[k];
}    /* end of k-loop */

yhid[i][j] = sigmoid(sumt[i][j] - total);
}    /* end of j-loop */
}    /* end of hidout() */




void forward(ip, w1, v1, h1, seta1, seta2)
int ip;
float w1[][MAX_HID_COL][MAX_IN_ROW][MAX_IN_COL];
float v1[][MAX_OUT_COL][MAX_HID_ROW][MAX_HID_COL];
float h1[][MAX_HID_COL][9];
float seta1[][MAX_HID_COL];
float seta2[][MAX_OUT_COL];
{
extern float sum();
extern float sigmoid();
extern void hidout();

for(i = 0; i < hidrow; i++)    /* Calculate activations of PEs */
for(j = 0; j < hidcol; j++)    /* in the 2nd layer */
{
tempsum[i][j]=sum_hid(xin[ip],w1[i][j],inrow,incol) - seta1[i][j];
yhidtemp[i][j] = sigmoid(tempsum[i][j]);
}


  /* calculate the activations of PEs in 2nd layer after feedback */

hidout(tempsum, yhidtemp, h1, yhid, hidrow, hidcol);
```

```
for(i = 0; i < outrow; i++)
for(j = 0; j < outcol; j++)
{
total = sum_out(yhid, v1[i][j], hidrow, hidcol) - seta2[i][j];
out[i][j] = sigmoid(total);
}
}
```

I.3.13 Matrix Multiplication: matrix_sum.nn

```
  /* summerize two 2_D matrix term by term product for hidden layer outputs */
float sum_hid(pta, ptb, nr, nc)
float (*pta)[MAX_IN_COL], (*ptb)[MAX_IN_COL];
int nr, nc;
{
int i, j;
float total = 0.0;

for(i = 0; i < nr; i++)
for(j = 0; j < nc; j++)
total += pta[i][j]*ptb[i][j];

return (total);
}
```

```
  /* summerize two 2_D matrix term by term product for output layer outputs */
float sum_out(pta, ptb, nr, nc)
float (*pta)[MAX_HID_COL], (*ptb)[MAX_HID_COL];
int nr, nc;
{
int i, j;
float total = 0.0;

for(i = 0; i < nr; i++)
for(j = 0; j < nc; j++)
total += pta[i][j]*ptb[i][j];

return (total);
}
```

I.3.14 Sigmoid Transfer Function: sigmoid.nn

```
   /* sigmoid transfer function */
float sigmoid(x)
float x;
{
x = 1./(1. + (float)exp(-x/sslope));
return x;
}
```

I.3.15 Change Weights After Adjusted: chweights.nn

```
   /* using new weights wt and vt to replace old ones w and v */

void chwts(wt, vt, ht)
float (*wt)[MAX_HID_COL][MAX_IN_ROW][MAX_IN_COL];
float (*vt)[MAX_OUT_COL][MAX_HID_ROW][MAX_HID_COL];
float (*ht)[MAX_HID_COL][9];

{
for(i = 0; i < hidrow; i++)
for(j = 0; j < hidcol; j++)
for(k = 0; k < inrow; k++)
for(l = 0; l < incol; l++)
{
dwold[i][j][k][l] = wt[i][j][k][l] - w[i][j][k][l];
w[i][j][k][l] = wt[i][j][k][l];
}


for(i = 0; i < outrow; i++)
for(j = 0; j < outcol; j++)
for(k = 0; k < hidrow; k++)
for(l = 0; l < hidcol; l++)
{
dvold[i][j][k][l] = vt[i][j][k][l] - v[i][j][k][l];
v[i][j][k][l] = vt[i][j][k][l];
}
```

```
for(i=0; i<hidrow; i++)
for(j=0; j<hidcol; j++)
for(k=0; k<9; k++)
{
dhold[i][j][k] = ht[i][j][k] - h[i][j][k];
h[i][j][k] = ht[i][j][k];
}

}
```

```
   /* using new threshold values to replace old ones */

void chseta(shtemp, sotemp)
float (*shtemp)[MAX_HID_COL];
float (*sotemp)[MAX_OUT_COL];

{
for(i = 0; i < hidrow; i++)
for(j = 0; j < hidcol; j++)
{
dsetahold[i][j] = shtemp[i][j] - setah[i][j];
setah[i][j] = shtemp[i][j];
}


for(i = 0; i < outrow; i++)
for(j = 0; j < outcol; j++)
{
dsetaoold[i][j] = sotemp[i][j] - setao[i][j];
setao[i][j] = sotemp[i][j];
}
}
```

I.3.16 Save or Retrieve a Network: netio.nn

```
   /* function for saving a network */

void netsave(netname)
char netname[];
{
```

```
#define IO fprintf
extern FILE *fileopen();

net = fileopen(netname, NEW);     /* open a file to save weights */


IO(net,"%d\n", count);

IO(net,"%d %d\n", inrow, incol);
IO(net,"%d %d\n", hidrow, hidcol);
IO(net,"%d %d\n", outrow, outcol);
IO(net,"%f\n", sslope);

   /* save the min. and max. values of each input and output variable, which
are used to normalize the training data file */

IO(net, "\n\n");
for(i=0; i<inrow; i++)
{
for(j=0; j<incol; j++)
IO(net, "%f ", minin[i][j]);
IO(net, "\n");

for(j=0; j<incol; j++)
IO(net, "%f ", maxin[i][j]);
IO(net, "\n\n");
}

IO(net, "\n\n");
for(i=0; i<outrow; i++)
{
for(j=0; j<outcol; j++)
IO(net, "%f ", minout[i][j]);
IO(net, "\n");

for(j=0; j<outcol; j++)
IO(net, "%f ", maxout[i][j]);
IO(net, "\n\n");
}

IO(net, "\n\n");

   /* save weights between 1st f 2nd layers connections */

for(i=0;i<hidrow;i++)
```

```
for(j=0;j<hidcol;j++)
{
for(k=0;k<inrow;k++)
{
for(l=0;l<incol;l++)
IO(net,"%f ", w[i][j][k][l]);
IO(net,"\n");
}
IO(net,"\n\n");
}

   /* save weights between 2nd f 3nd layers connections */

for(i=0;i<outrow;i++)
for(j=0;j<outcol;j++)
{
for(k=0;k<hidrow;k++)
{
for(l=0;l<hidcol;l++)
IO(net,"%f ", v[i][j][k][l]);
IO(net,"\n");
}
IO(net,"\n\n");
}

   /* save weights of lateral connections of PEs in 2nd layer */

for(i=0; i<hidrow; i++)
for(j=0; j<hidcol; j++)
{
for(k=0; k<9; k++)
{
IO(net, "%f ", h[i][j][k]);
if(k == 4)
IO(net,"\n");
}
IO(net, "\n\n");
}

   /* save threshold values of 2nd layer PEs */

for(i=0; i<hidrow; i++)
for(j=0; j<hidcol; j++)
```

```
IO(net, "%f\n", setah[i][j]);
IO(net, "\n\n");


   /* save threshold values of 3nd layer PEs */

for(i=0; i<outrow; i++)
for(j=0; j<outcol; j++)
IO(net, "%f\n", setao[i][j]);


   /* save index data for the selection of the training variables */

IO(net, "\n\n\n%d\n", tot_var);
for(i=0; i<tot_var; i++)
IO(net, "\n%d %d\n", var_index[i][0], var_index[i][1]);

fclose(net);

}    /* end of netsave() */




   /* function for retrieving a network */

void netin(netname)
char netname[];
{
#define IO2 fscanf
extern FILE *fileopen();

net = fileopen(netname, OLD);     /* open a trained network */


IO2(net,"%d",&count);

IO2(net,"%d%d", &inrow, &incol);
IO2(net,"%d%d", &hidrow,&hidcol);
IO2(net,"%d%d", &outrow,&outcol);
IO2(net,"%f", &sslope);

   /* retraive the min. and max. values of each input and output variable, which
are used to normalize the training data file */
```

```
for(i=0; i<inrow; i++)
{
for(j=0; j<incol; j++)
IO2(net, "%f", &minin[i][j]);

for(j=0; j<incol; j++)
IO2(net, "%f", &maxin[i][j]);
}

for(i=0; i<outrow; i++)
{
for(j=0; j<outcol; j++)
IO2(net, "%f", &minout[i][j]);

for(j=0; j<outcol; j++)
IO2(net, "%f", &maxout[i][j]);
}


   /* read weights between 1st f 2nd layer */

for(i=0;i<hidrow;i++)
for(j=0;j<hidcol;j++)
for(k=0;k<inrow;k++)
for(l=0;l<incol;l++)
IO2(net,"%f", &w[i][j][k][l]);


   /* read weights between 2nd f 3nd layer */

for(i=0;i<outrow;i++)
for(j=0;j<outcol;j++)
for(k=0;k<hidrow;k++)
for(l=0;l<hidcol;l++)
IO2(net,"%f", &v[i][j][k][l]);


   /* read weights of lateral connections of PEs in 2nd layer */

for(i=0; i<hidrow; i++)
for(j=0; j<hidcol; j++)
for(k=0; k<9; k++)
IO2(net, "%f", &h[i][j][k]);
```

```
   /* read threshold values ofthe 2nd layer units */
for(i=0; i<hidrow; i++)
for(j=0; j<hidcol; j++)
IO2(net, "%f", &setah[i][j]);


   /* read threshold values of the 3nd layer units */
for(i=0; i<outrow; i++)
for(j=0; j<outcol; j++)
IO2(net, "%f", &setao[i][j]);


   /* read index data for the selection of the training variables */

IO(net, "%d", &tot_var);
for(i=0; i<tot_var; i++)
IO(net, "%d%d", &var_index[i][0], &var_index[i][1]);

fclose(net);

}    /* end of netin() */
```

I.3.17 Add Random Noise to the Given Data: add_noise.nn

```
   /* This function performs adding random noise to the input data */
void noise_add(level, x)
float level, x[][MAX_IN_ROW][MAX_IN_COL];
{
#define UPPER 1.0
#define LOWER -1.0

int i, j, k;    /* loop variables */
float rnum;

for(i=0; i<npats; i++)
for(j=0; j<inrow; j++)
for(k=0; k<incol; k++)
{
rnum = (UPPER - LOWER)*(float)rand()/SCALE + LOWER;
x[i][j][k] *= (1.0 + level/100.0);
}
}    /* end of add_noise() */
```

I.3.18 Report GA Search Results: report.nn

```
  /* This function reports the variables selected by GA, training error,
and fitness value, for each network throughout the whole process. */

void record(fp, pop, error, size, id, flag)
FILE *fp;
struct individual pop;
float error;
int size, id, flag;
{

int i;


if(flag == NEW)
{
fprintf(fp,
"This file contains the results of using the new form of the objective\n");
fprintf(fp,
"function which is the function of net training error, number of input\n");
fprintf(fp,
"variables selected, and the generation index. The parameters used for\n");
fprintf(fp,
"training are listed below:\n\n");
fprintf(fp, " Size of the hidden layer is %d x %d\n", hidrow, hidcol);
fprintf(fp, " The learning rate alpha = %f\n", alpha);
fprintf(fp, " The learning rate beta = %f\n", beta);
fprintf(fp, " The learning rate alpham = %f\n", alpham);
fprintf(fp, " The total training cycle = %d\n\n", times);
}    /* end of if(flag == NEW) */



fprintf(fp, "For network No. %d the selected inputs are:\n", id);
printf("For network No. %d the selected inputs are:\n", id);
for(i = 0; i < tot_var; i++)
if(pop.chrom[i+1] == 1)
{
fprintf(fp, "%d ", i);
printf("%d ", i);
}
```

```
fprintf(fp, "\n\nTotal number of inputs is %d\n", size);
fprintf(fp, "pattern average error is %f\n", error);
fprintf(fp, "The fitness function value is %f\n\n\n", pop.fitness);
printf("\n\nTotal number of inputs is %d\n", size);
printf("pattern average error is %f\n", error);
printf("The fitness function value is %f\n\n\n", pop.fitness);
}
```

I.4 Fitness Function for GA Search: interface.sga

```
   /* interfac.sga: contains objfun, deccode */
   /* change this for different problems */

float objfun(inputs, error, gen)
int inputs, gen;
float error;
{
float fit, r, x, y, z;

if(inputs == 0) inputs = 1;

x = (float)lchrom / (float)inputs;
x -= 1;
y = exp(0.15 * (float)(gen+1) * log(x));
   /* z = exp(0.2 * (float)(gen+1) * log(error)); */
r = exp(1./3. * log((float)gen + 1.0));
z = exp(0.7 * r * log(error));
fit = (1.0 - exp(-y)) * exp(-0.01 * z);

return(fit);

}
```

VITA

Zhichao Guo was born in Pixian County, Sichuan Province of The People's Republic of China on June 9, 1957. In spring of 1978, he attended Shanghai Jiaotong University (Shanghai, China) where he obtained the Bachelor of Science degree in January 1982 and the Master of Science degree in July 1984, respectively, both majored in Nuclear Engineering. Also in July 1984 , he began to work as a research associate at the same university. He spent three years of research and teaching in the field of fluid flow and heat transfer, especially in two-phase flow and boiling heat transfer. In September 1987, he enrolled at the University of Tennessee at Knoxville to pursue the Doctor of Philosophy degree and continue his study of nuclear engineering. Also in September 1987, he began working as a graduate research assistant at Nuclear Engineering Department, the University of Tennessee.