Neural Networks and Their Application for Structural Self-Diagnosis

by

Tung-Ju Hsieh

Bachelor of Science in Civil Engineering
June 1997
National Chiao Tung University
Hsinchu, Taiwan

Submitted to the Department of Civil and Environmental Engineering in partial fulfillment of the requirements for the degree of

Master of Science in Civil and Environmental Engineering
at the
Massachusetts Institute of Technology

February 2001

Signature of Author..... .....
Department of Civil and Environmental Engineering
January 10, 2001

Certified by..... .....
Jerome J. Connor
Professor, Department of Civil and Environmental Engineering
Thesis Supervisor

Accepted by..... .....
Oral Buyukozturk
Chairman, Department of Committee on Graduate Studies

Neural Networks and Their Application for Structural Self-Diagnosis

by

Tung-Ju Hsieh

Submitted to the Department of Civil and Environmental Engineering on January 1, 2001 in partial fulfillment of the requirements for the degree of Master of Science in Civil and Environmental Engineering.

ABSTRACT

The objective of this study is to design a neural network based Java program to estimate the location and size of cracks in a beam. This study also explores the potential of artificial neural networks for structural self-diagnosis. A neural network architecture for structural self-diagnosis is formulated to achieve this objective. There are three components of the architecture: the physical system model, data preprocessing units, and neural networks that are trained to predict the location and the magnitude of the damage. Important design issues include choosing variables to be observed, the architecture of the neural networks, and the training algorithm. These design issues are examined for the case of a cantilever beam. First, a single-point damage diagnosis system model is developed and evaluated. The approach is then extended to handle a two-point damage. Numerical modeling of the cantilever beam vertical displacements is calculated with SAP-2000. The cantilever beam is divided to 21 equal length sections, and damage is introduced by reducing the moment of inertia at a specific damage location. The results of this work can be used for the inspection of structural elements such as cantilever beams. The proposed method solves efficiently the inverse problem of estimating damage size and location from the beam displacement information for this restricted scenario. The neural network also performs adequately for data contaminated by measurement errors.

Thesis Supervisor: Professor Jerome J. Connor
Title: Professor of Civil and Environmental Engineering

ACKNOWLEDGEMENTS

I would like to sincerely thank Professor Jerome J. Connor, my thesis advisor, for his encouragement and guidance. This work would not have been accomplished without his encouragement and suggestion.

Meanwhile, wholeheartedly thanks for Yi-Mei Maria Chow, Yi-San Lai, Tai-Lin Tung, and Ji-Yong Wang for their help and encouragement during my graduate study at the Institute. Thanks to all those friends at MIT who shared their experience with me.

I would like to express my gratitude to Professor Shih-Lin Hung for his recommendation and encouragement my graduate study in this country.

I want to like to thank my aunt Han-Yee Chen, who encouraged me and gave me advises toward my future.

I would also like to thank my aunt Sue H Lin, whose phone calls and e-mails brightened my hardest days at the Institute.

Finally, I would like to send my deepest thanks to my family. This thesis is dedicated to my parents, whose love and support have been the main source of my strength throughout my life.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1
Introduction

1.1 Neural Network-Based Structural Damage Diagnosis

Structural identification has become increasingly an important research topic in connection with damage assessment and safety evaluation of existing structures. Structural identification is a process for constructing a mathematical description of a physical system when both the input and the corresponding output are known. In structural applications, the input is usually a forcing function and the output could be the displacement, or any other structural response such as strain or vibration signals. The primary objective of structural self-diagnosis is to estimate a set of behavioral parameters from the measured response of the real structure due to a known disturbance. The principle of structural self-diagnosis is based on the fact that when a structure experiences various degrees of damage, certain characteristics undergo changes. To identify those changes, a sequence of tests is conducted during inspection, and the resulting parameters are measured. This is an inverse problem and can be dealt with by standard system identification techniques. The application of artificial neural networks is demonstrated as an efficient tool for structural identification, especially when the measured data is limited or imprecise. Recent developments in this area have been made possible by rapid advances in computer technology for data acquisition, signal processing, and analysis. Pattern-matching techniques using neural networks have drawn considerable attention in the field of damage assessment and structural identification.

Structural self-diagnosis techniques may be classified as global or local. Global methods attempt to simultaneously assess the condition of the whole structure, whereas local methods focus non-destructive evaluation tools on specific structural components. However, the universe of damage detection scenarios likely to be encountered in realistic applications to all candidate physical systems is very broad and encompassing. Structural self-diagnosis techniques that rely on non-parametric system identification approaches, for which a priori information about the nature of the model is not needed, have a significant advantage when dealing with real-world situations where the selection of a suitable class of parametric models to be used for identification purpose is quite often a demanding task.

Figure 1.1 shows the schematic diagram of the neural network based approach of this thesis for the damage detection methodology. The overall procedure is divided into two parts: the training stage and the detection stage. In the training stage, as depicted in the Figure 1.1, a neural network is trained by the data obtained from the undamaged system using an appropriate training method. In the detection stage, as shown in Figure 1.2, the trained network is fed input data that is the same input to the system. Then, the output from the network and the output from the system are compared to each other. If the network has been well trained, and if the system characteristics have not changed, both the system and the network will have matching outputs. On the other hand, if the system has changed, the output from the system will not correspond any more to the output of the trained network. Consequently, the network will yield an output error. Therefore, the deviation between the output from the system and the output from the network provides a quantitative measure of the changes in the physical system relative to its undamaged condition.

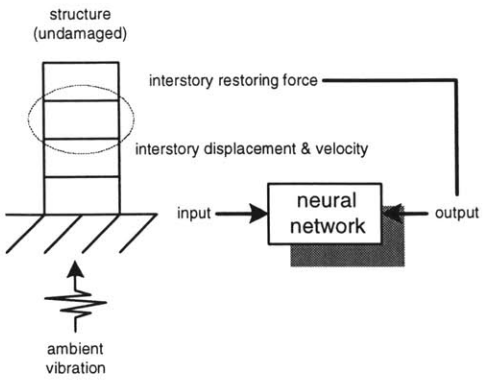


Figure 1.1 Schematic diagram of damage detection using neural networks: Training of Neural Network.

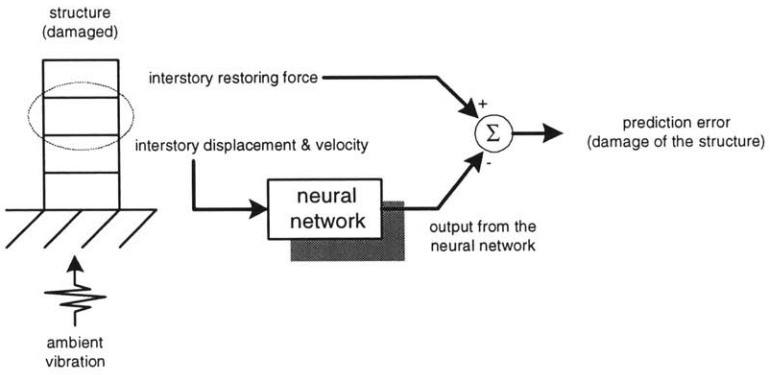


Figure 1.2 Schematic diagram of damage detection using neural networks: Damage Detection by the Neural Network.

1.2 Objects and Scope of the Research

The objective of this study is to design a neural network based Java program to estimate the location and size of cracks in a beam. This study also explores the potential of artificial neural networks for structure self-diagnosis. A neural network architecture for structural self-diagnosis is formulated to achieve this objective. There are three main components of the architecture: the physical system model, data preprocessing units, and neural networks that are trained to predict the location and the magnitude of the damage. Important design issues include choosing variables to be observed, the architecture of the

neural networks, and the training algorithm. These design issues are examined for the case of a cantilever beam.

The main part of this thesis work is to develop a neural network based Java program to evaluate the crack size and location in a cantilever steel beam. This Java program has three main functions. Firstly, it can learn from the environment. Secondly, it could performance the analysis of the training result. Finally, it can do graphical display of the crack size and location in a cantilever beam.

A multi-layer feedforward neural network is used in this thesis research. The back-propagation learning model is appropriate for pattern classification. First, a single-point damage diagnosis system model is developed and evaluated. The approach is then extended to handle a two-point damage. Numerical modeling of the cantilever beam vertical displacements is calculated with SAP-2000. The cantilever beam is first divided into 21 equal length sections, and then damage is introduced by reducing the moment of inertia at a specific damage location. The proposed method solves efficiently the inverse problem of estimating damage size and location from the cantilever beam displacement information. The neural network also performs adequately for data contaminated by measurement errors.

1.3 Organization

Chapter 2 contains an introduction to the basic concepts and definitions of artificial neural networks. It introduces several different kinds of neural network architectures and training paradigms.

Chapter 3 gives a more detail description of the neural network learning algorithms. The back-propagation algorithm and conjugate-gradient algorithm are discussed in this chapter. The last part of this chapter has a discussion in the optimum design of a neural network.

Chapter 4 discusses the design and implementation of a Java based Neural Networks system. There are two key software issues; data structure and object-oriented programming. The architecture of the neural network-based Java program is illustrated. The class hierarchy and member functions are also demonstrated.

Chapter 5 is concerned with the application of neural networks to structural damage self-diagnosis. A neural network architecture is presented. Two cases are studied; single-point damage and two-point damage. The neural network-based Java problem is used to evaluate these two cases. The neural network-based program performs adequately for data contaminated by measurement errors.

Chapter 2

Foundations of Artificial Neural Networks

2.1 Background History of Artificial Neural Networks

The modern era of neural networks began with the pioneering work of McCulloch and Pitts (1943). In their classic paper, McCulloch and Pitts describe a logical calculus of neural networks that united the studies of neurophysiology and mathematical logic. They showed that a network so constituted would, in principle, compute any computable function. It is generally agreed that the disciplines of neural networks and of artificial intelligence were born.

In 1948, Wiener wrote a book *Cybernetics*, describing some important concepts for control, communications, and statistical signal processing. Wiener appears to grasp the physical significance of statistical mechanics in the context of the subject matter, but it was left to Hopfield to bring the linkage between statistical mechanics and learning systems to full fruition.

The next major development in neural networks came in 1949 with the publication of Hebb's book *The Organization of Behavior*, in which an explicit statement of a physiological learning rule for *synaptic modification* was presented for the first time. Hebb's book has been a source of inspiration for the development of computational models of learning and adaptive systems. In 1967, Minsky's book, *Computational Finite and Infinite Machines*, was published. This clearly written book extended the 1943 results

of McCulloch and Pitts and put them in the context of automata theory and the theory of computation. In 1958, a new approach to the pattern recognition problem was introduced by Rosenblatt in his work on the *perceptron*, a novel method of supervised learning. In 1969, Minsky and Papert used mathematics to demonstrate that there are fundamental limits on what single-layer perceptrons can compute. They stated that there was noreason to assume that any of the limitations of single-layer perceptrons could be overcome in the multilayer version. And then the development of neural network had a lag for more than 10 years.

In 1982, Hopfield used the idea of an energy function to formulate a new way of understanding the computation performed by recurrent networks with symmetric synaptic connections. This particular class of neural networks with feedback attracted a great deal of attention in the 1980s. Moreover, Hopfield showed that he had the insight from the spin-glass model in statistical mechanics to examine the special case of recurrent networks with symmetric connectivity, thereby guaranteeing their convergence to a stable condition.

In 1986, the development of the back-propagation algorithm was reported by Rumelhart, Hilton, and Williams. In the same year, the celebrated book, Parallel Distributed Processing: Explorations in the Microstructures of Cognition, was published. This book has been a major influence in the use of back-propagation learning, which has emerged as the most popular learning algorithm for the training of multilayer perceptrons. IN 1988, Broomhead and Lowe described a procedure for the design of layered feedforward networks using radial basis functions, which provide an alternative to multilayer perceptrons. In the early 1990s, Vapnik invented a computationally powerful class of supervised learning networks, called support vector machines, for solving pattern recognition, regression, and density estimation problems.

Neural networks have certainly come a long way from the early days of McCulloch and Pitts. They have established themselves as an interdisciplinary subject with deep roots in the neurosciences, psychology, mathematics, the physical sciences, and engineering.

2.2 Definition of Artificial Neural Networks

A neural network is a massive parallel-distributed processor made up of simple processing units, which has a natural propensity for storing experiential knowledge and making it available for use. It resembles the brain in two respects: First, knowledge is acquired by the network from its environment through a learning process. Second, inter-neuron connection strengths, known as synaptic weights, are used to store the acquired knowledge. The modification of synaptic weights provides the traditional method for the design of neural networks. Such an approach is the closest to linear adaptive filter theory, which is already well established and successfully applied in many diverse fields. Neural networks are also referred to in literature as neurocomputers, connectionist networks, parallel-distributed processors, etc.

2.3 Models of a Neuron

A *neuron* is an information-processing unit that is fundamental to the operation of a neural network. Figure 2.1 shows the model of a neuron, which forms the basis for designing neural networks. In mathematical terms, we may describe a neuron k by writing the following pair of equations:

$$u_k = \sum_{j=1}^{m} w_{kj} x_j \tag{2.1}$$

and

$$y_k = \varphi(u_k + b_k) \tag{2.2}$$

where x_m is the input signal; w_{km} is the synaptic weights of neuron k; u_k is the linear combiner output signal of the neuron. The use of bias b_k has the effect of applying a transformation to the output u_k of the linear combiner in the model of Figure 2.1, as shown by

$$v_k = u_k + b_k \tag{2.3}$$

The bias b_k is an external parameter of artificial neuron k. We may account for its presence as in Equation (2.2). Equivalently, we may formulate the combination of Equations (2.1) to (2.3) as follows:

$$v_k = \sum_{j=0}^{m} w_{kj} x_j \tag{2.4}$$

and

$$y_k = \varphi(v_k) \tag{2.5}$$

In Equation (2.4) we have added a new synapse. Its input is $x_0 = +1$, and its weight is $w_{k0} = b_k$. The activation function, denoted by $\varphi(v)$, defines the output of a neuron in terms of the induced local field v.

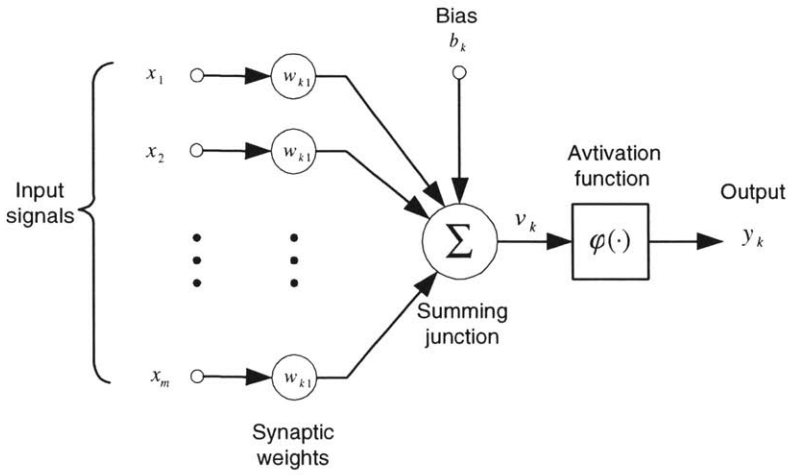


Figure 2.1 Nonlinear model of a neuron.

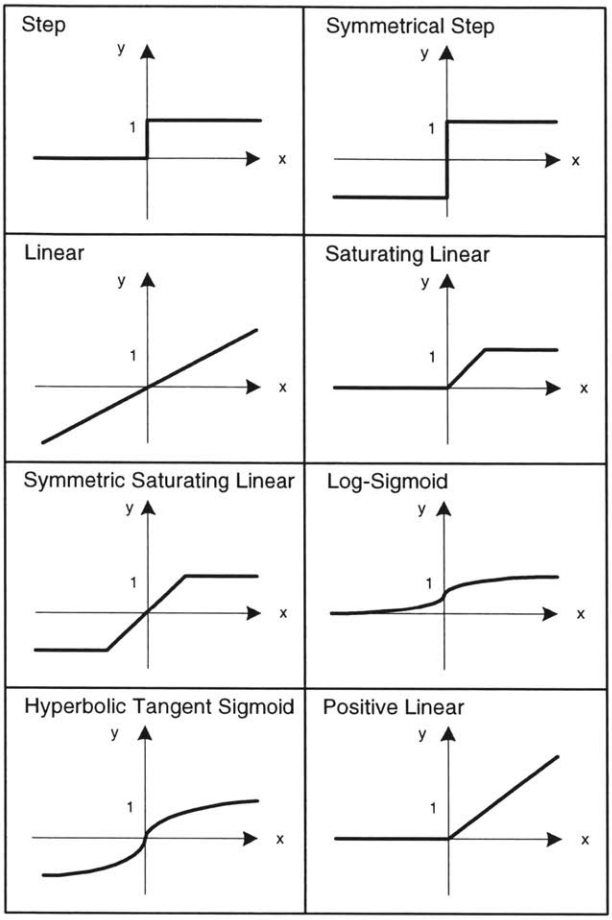


Figure 2.2 Common Activation Functions

2.4 Types of Neural Network Architectures

The manner in which the neurons of a neural network are structured is intimately linked with the learning algorithm used to train the network. We may therefore speak of learning algorithms used in the design of neural networks as being structured. In general, there are three fundamentally different classes of network architectures:

Single-Layer Feedforward Networks

In a layered neural network the neurons are organized in the form of layers. In the simplest form of a layered network, we have an input layer of source nodes that project onto an output layer of neurons. Figure 2.3 shows the architecture of a single-layered neural network.

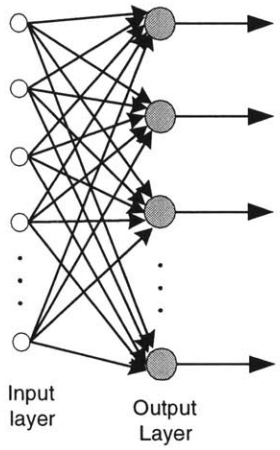


Figure 2.3 Feedforward network with a single layer on neurons.

Multilayer Feedforward Networks

Multilayer feedforward neural networks have one or more hidden layers, whose computation nodes are correspondingly called hidden neurons or hidden units. The function of hidden neurons is to intervene between the external input and the network output in some useful manner. By adding one or more hidden layers, the network is enabled to extract higher-order statistics. The ability if hidden neurons to extract higher-order statistics is particularly valuable when the size if the input layer is large. The output signals of the second layer are used as inputs to the third layer of the network, and so on for the rest of the network. Figure 2.4 illustrates the layout of a multilayer feedforward neural network for the case of a single hidden layer.

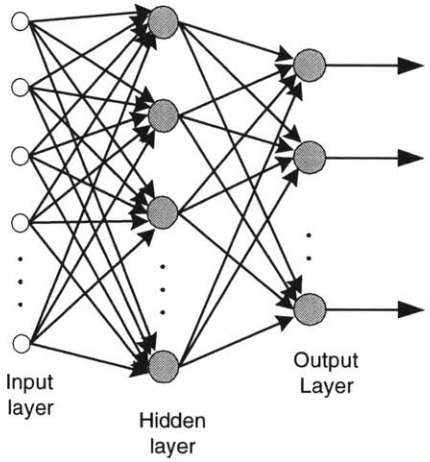


Figure 2.4 Feedforward network with one hidden layer and one output layer.

Recurrent Network

A recurrent neural network distinguishes itself from a feedforward neural network in that it has at least one feedback loop. A recurrent network may consist of a single layer of neurons with each neuron feeding its output signal back to the inputs of all the other neurons. Figure 2.5 shows a recurrent network. In the structure depicted in this figure there are no self-feedback loops in the network; self-feedback refers to a situation where the output of a neuron is fed back into its own input. The presence of feedback loop has a profound impact on the learning capability of the network and on its performance. Moreover, the feedback loops involve the use of particular branches composed of unit-delay elements. Which result in a nonlinear dynamical behavior, assuming that the neural network contains nonlinear units.

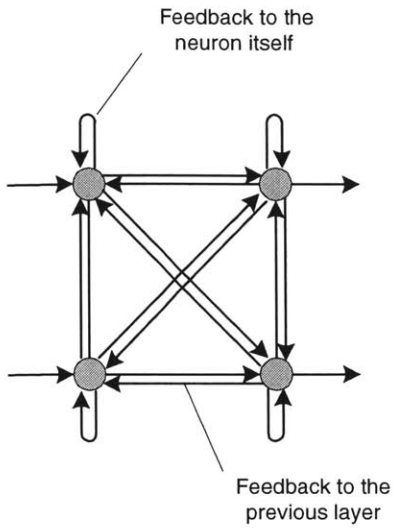


Figure 2.5 Recurrent Neural Networks.

2.5 Neural Network Learning Algorithms

Learning is one of the important features of artificial neural networks. A neural network learns from its environment through an interactive process of adjustments to its weights and bias values. Theoretically, the network becomes more knowledgeable after each iteration of the training process. During the learning process, the following events occur in sequence: First, the neural network is stimulated by the environment inputs. Second, the neural network changes its parameters as a result of its environmental stimulation. Finally, the neural network responds in a new way to the environment because of the changes that have occurred in its internal structure.

A learning algorithm is a prescribed set of well-defined rules for the solution of a learning problem. Learning algorithms differ from each other in the way in which the adjustment to a synaptic weight of a neuron is formulated. Another factor to be considered is the manner in which a neural network made up of a set of inter-connected neurons, relates to its environment.

2.5.1 Supervised Learning

Figure 2.6 shows the block diagram that illustrates unsupervised learning. The knowledge was being represented by a set of input-output examples. The teacher is able to provide the neural network with a desired response for that training vector. The desired response represents the optimum action to be performed by the neural network. The network parameters are adjusted under the combined influence of the training vector and the error signal. This adjustment is carried out iteratively in a step-by-step fashion with the aim of eventually making the neural network emulate the teacher. The form of supervised learning is the error-correction learning. It is a closed-loop feedback system, but the unknown environment is not in the loop. As a performance measure for the system we may think in terms of the mean-square error or the sum of squared errors over the training sample, defined as a function of the free parameters of the system. This function may be visualized as a multidimensional error-performance surface. The true error surface is averaged over all possible input-output examples. Any given operation of the system under the teacher's supervision id represented as a point as a point on the error surface. For the system to improve performance over time and therefore learn from the teacher, the operating point has to move down successively toward a minimum point of the error surface. Nevertheless, given an algorithm designed to minimize the cost function, an adequate set of input-output examples, and enough time permitted to do the training, a supervised learning system is usually able to perform such tasks as pattern classification and function approximation.

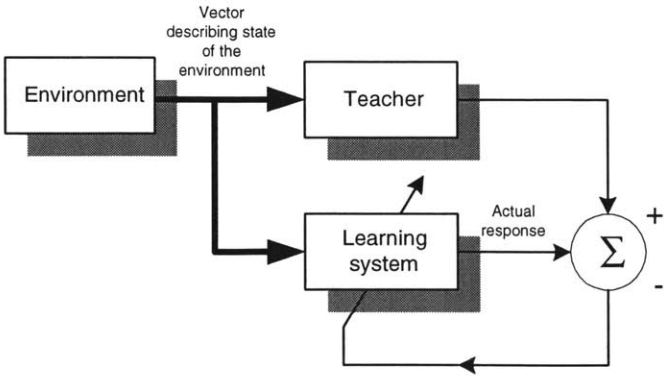


Figure 2.6 Block diagram of learning with a teacher.

2.5.2 Unsupervised Learning

There is no teacher to oversee the learning process when it comes to unsupervised learning. There are no labeled examples of the function to be learned by the network. Figure 2.7 shows the block diagram of one form of a reinforcement learning system built around a critic that converts a primary reinforcement signal received from the environment into a higher quality reinforcement signal called the heuristic reinforcement signal, both of which are scalar input. Delayed-reinforcement learning is very appealing provides the basis for the system to interact with its environment, thereby developing the ability to learn to perform a prescribed task solely on the basis of the outcomes of its experience that result form the interaction.

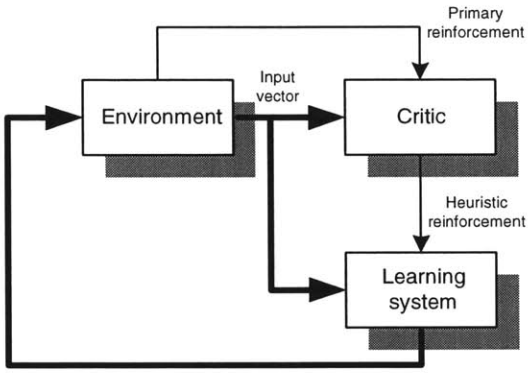


Figure 2.7 Block diagram of reinforcement learning.

In unsupervised learning there is no external teacher to oversee the learning process. Rather, provision is made for a task-independent measure of the quality of representation that the network is required to learn, and the free parameters of the network are optimized with respect to that measure. Once the network has become tuned to the statistical regularities of the input data, it develops the ability to form internal representations for encoding features of the input and thereby to create new classes automatically.

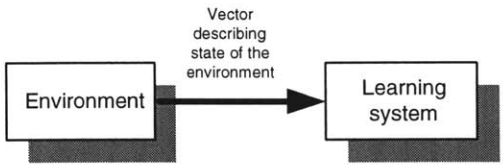


Figure 2.8 Block diagram of unsupervised learning.

2.6 Application of Neural Networks in Structural Engineering

Artificial Neural Networks have been applied in many fields. Here are some lists of the neural network applications: aerospace, automotives, banking, defense, electronics, entertainment, financial, insurance, manufacturing, medical, exploration in oil and gas, robotics, speech recognition, market securities, telecommunications, and transportation. There are also some applications in the field of structural engineering. Neural networks can be trained based on observed information. The pattern recognition problem demonstrates a solution, which would otherwise be difficult to code in a conventional program. Neural networks have been applied to the field of structural engineering. These applications have demonstrated that neural networks may be successfully applied to solve many structural engineering problems. These networks are capable of simulating learning of the type of knowledge used by structural engineers. While in appears that neural networks may be built to solve almost any problem in which sufficient training data exit, their use should be limited to problems that are presently difficult or time-consuming to solve. For example, many finite-element solutions fall into the time-consuming category. Several problems in which algorithmic solutions are difficult to develop are summarized here: Combining software with traditional rule-based expert systems to give an expert system the power to treat new problems including automatic learning. Studying the use if neural networks in solving civil engineering optimization problems. Using programs with pattern recognition capabilities to help identify code and design inadequacies. Past acceptable designs would be used for training purposes. Continued study involving training methods to reduce required training time and improve developed system accuracy.

Chapter 3

Design and Training of Neural Networks

3.1 Back-Propagation Algorithm

3.1.1 History of Back-Propagation Algorithm

The back-propagation algorithm is the most commonly used method for training multi-layer feedforward neural networks. D.E. Rumelhart, G.E. Hilton, and R.J. Williams popularized the back-propagation algorithm in 1986. The book, *Parallel Distributed Processing: Explorations in the Microstructures of Cognition*, edited by Rumelhart and McClelland, was published. This book has been a major influence in the use of back-propagation for the training of multi-layer perceptrons. After the back-propagation algorithm was discovered by D.E. Rumelhart, G.E. Hilton, and R.J. Willaims in mid-1980s, it was found that the algorithm had been mentioned by Werbos. Werbos's Ph.D. thesis at Harvard University in 1974 was the first documented description of efficient reverse-mode gradient computation that was applied to neural network models. The basic idea of back-propagation can be traced further back to the book, *Applied Optimal Control*, edited by Bryson and Ho in 1969. In Section 2.2, a derivation of back propagation using a Lagrangian formalism is mentioned. However, most of the credit for the back-propagation algorithm was given to D.E. Rumelhart, G.E. Hilton, and R.J. Williams for proposing its use for machine learning.

3.1.2 Introduction of Multilayer Perceptrons

In general, a network consists one input layer, one or more hidden layers of computation nodes, and an output layer of computation nodes. The input signal propagates through the network in a forward direction. This kind of neural network is commonly known as multilayer perceptrons (MLPs).

Mulrilayer perceptrons have been applied successfully to solve problems by training them in a supervised manner with error-correction learning rule. Back-propagation learning consists two passes through the different layers of the network: a forward pass and a backward pass. In the forward pass, an input vector is applied to the sensory nodes, and its effect propagates through the network layer by layer. Finally, a set of output is produced. The weights of the networks are fixed during the forward pass. However, the weights are adjusted with an error-correction rule during the backward pass. The response of the network is subtracted from a target response to produce an error signal. This error signal is then propagated backward through the network against the direction of synaptic connections. The synaptic weights are adjusted to make the actual response of the network move closer to the desired response in a statistical sense.

The development of the back-propagation algorithm represents a landmark that it provides an efficient method for the training of multilayer perceptrons. Although the back-propagation algorithm is not the optimal solution for all solvable problems, it has put to rest the pessimism about learning in multilayer machines.

3.1.3 Preliminaries of Multilayer Neural Networks

Figure 3.1 shows the architectural of a multilayer perceptron with one input layer, two hidden layers, and an output layer. A neuron in any layer of the network is connected to all the neurons in the previous layer. Figure 3.2 shows a portion of the multilayer perceptron. There are two kinds of signals propagate in the network.

Function Signals

It is an input signal that comes in at the input end of the network propagates forward through the network, and produces an output signal at the output end of the network. It is presumed to perform a useful function at the output of the network. At each neuron of the network through which a function signal passes, the signal calculated as a function of the inputs and associated weights applied to that neuron.

Error Signals

An error signal originates at an output neuron of the network, and propagates backward through network. Its computation by every neuron of the network involves an error-dependent function.

Each hidden and output neuron of a multilayer perceptron is designed to perform two computations: The computation of the function signal appearing at the output of a neuron, which is expressed as a continuous nonlinear function of the input signal and synaptic weights associated with that neuron. The computation of an estimate of the gradient vector, which is needed for the backward pass through the network.

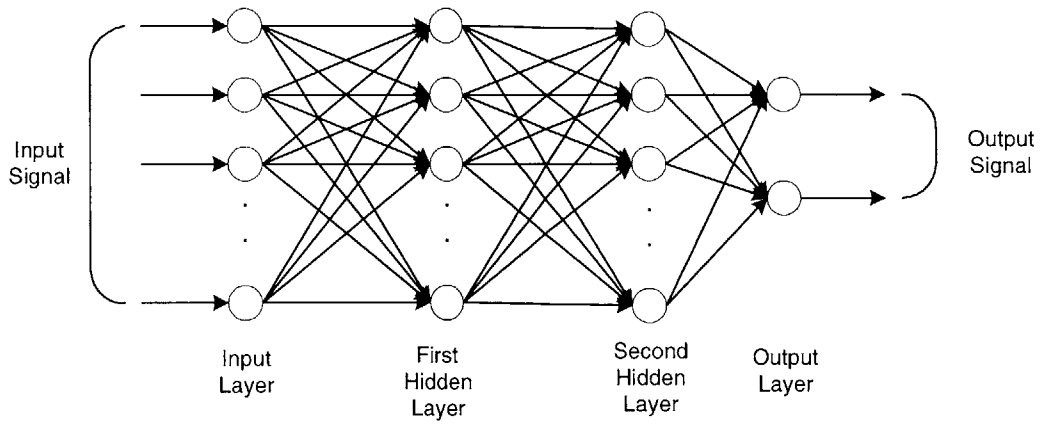


Figure 3.1 Architectural graph of a multilayer perceptron with two hidden layers.

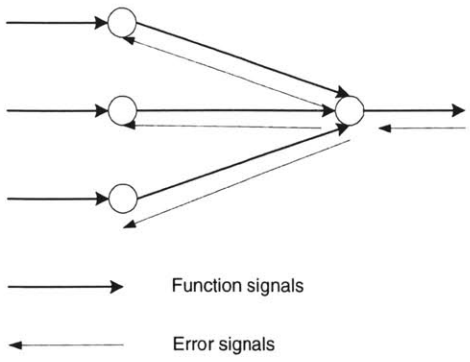


Figure 3.2 Illustration of the directions of two basic signal flows in a multilayer perceptron: forward propagation of function signals and back-propagation of error signals

3.1.4 Concept of Back-Propagation Algorithm

The back-propagation algorithm is an error-correcting learning procedure that generalizes the delta rule to multi-layer feedforward neural networks with hidden units between the input and output units. The feedforward net with back-propagation of error has been found to be an effective learning procedure for classification problems. (Rumelhart, Hilton, and Williams, 1986). Standard back-propagation algorithm is a gradient descent algorithm, as is the Widrow-Hoff learning rule. Properly trained back-propagation networks will give reasonable response answers when the inputs have never seen. A new input will lead to an output similar to the correct output for input vectors used in training that are similar to the new input. This property makes it possible to train a network on a representative set of input/target pairs and get good results without training the network on all input/target pairs.

The purpose of back-propagation is to adjust the network weights so the network produces the desired output in response to every input pattern in a predetermined set of training patterns. It is a supervised algorithm, for every input pattern, there is an externally corresponding specified correct output, which acts as a target for the network to imitate. The difference between the output value and the desired target value is called as an error. It is necessary to minimize the errors. The Learning with a teacher model

must decide which patterns to include in the training set and specify the correct output for each. It is an off-line algorithm in the sense that training and normal operation occur at different times. In the usual case, training could be considered part of the producing process wherein the network is trained once for a particular function, then frozen and put into operation. No further learning occurs after the initial phase.

In order to train a back-propagation neural network, it is necessary to have a set of input patterns and corresponding desired output, and an error function that measures the cost of differences between network output and the desired values. This is the basic step to implement a back-propagation neural network.

1. Present a training pattern and propagate it through the network to obtain the desired outputs.
2. Compare the network outputs with the desired target values and then calculate the error.
3. Calculate the derivatives $\partial E / \partial w_{ij}$ of the error with respect to the weights.
4. Adjust the weights to minimize the error.
5. Repeat the above procedure until the error is acceptably small or the limit of iteration is reached.

3.1.5 Explanation of the Back-Propagation Algorithm

The error at the output of neuron j at the presentation of the nth training example is defined by

$$e_j(n) = d_j(n) - y_j(n) \tag{3.1}$$

Define the instantaneous value of the error energy for neuron j as $\frac{1}{2}e_j^2(n)$. The instantaneous value $\xi(n)$ of the total error energy is obtained by summing $\frac{1}{2}e_j^2(n)$ over all neurons in the output layer; these are neurons for which error signals can be calculated directly. The total error energy can be written as

$$\xi(n) = \frac{1}{2}\sum_{j \in C} e_j^2(n) \tag{3.2}$$

where the set C includes all the neurons in the output layer of the network. Let N denote the total number of training set. The average squared error energy can be written as

$$\xi_{av} = \frac{1}{N}\sum_{n=1}^{N} \xi(n) \tag{3.3}$$

The instantaneous error energy $\xi(n)$, and the average error energy ξ_{av}, is a function of all the free parameters of the network. For a given training set, ξ_{av} represents the cost function as a measure of learning performance. The objective of the learning process is to adjust the free parameters of the network to minimize ξ_{av}. Consider a method of training in which the weights are updated on a pattern-by-pattern basis until one complete presentation of the entire training set has been dealt with. The adjustments to the weights are made in accordance with the respective errors computed for each pattern presented to the network.

The average of these individual weight changes over the training set is an estimate of the true change that would result from modifying the weights based on minimizing the cost function ξ_{av} over the entire training set.

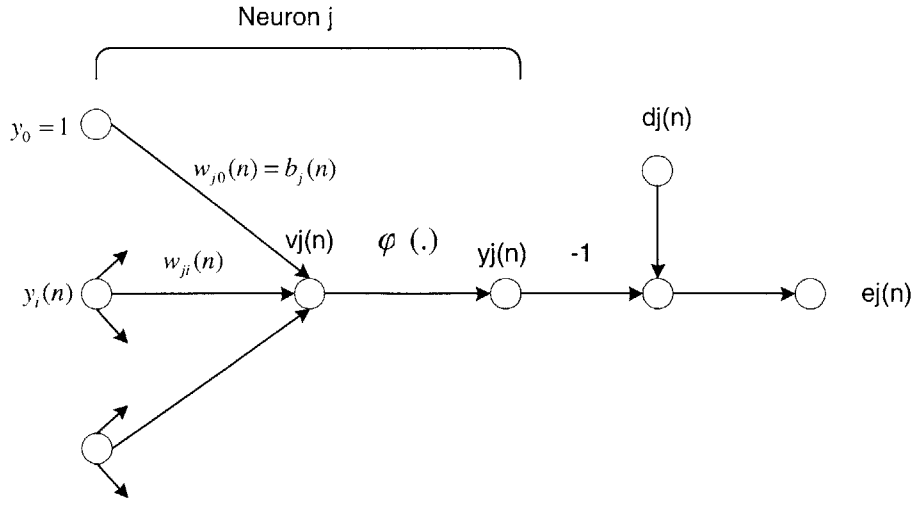


Figure 3.3 Signal-flow graph highlighting the details of output neuron j.

From Figure 3.3, neuron j being fed by a set of function signals produced by a layer of neurons to its left. The induced local field $v_j(n)$ produced at the input of the activation function associated with neuron j is therefore

$$v_j(n) = \sum_{i=0}^{m} w_{ji}(n) y_i(n) \tag{3.4}$$

where m is the total number of inputs fed to neuron j. The synaptic weight w_{j0} equals the bias b_j applied to neuron j. The function signal $y_j(n)$ appearing at the output of neuron j at iteration n is

$$y_j(n) = \varphi_j(v_j(n)) \tag{3.5}$$

The back-propagation algorithm applies a correction $\Delta w_{ji}(n)$ to the synaptic weight $\Delta w_{ji}(n)$, which is proportional to the partial derivation $\partial \xi(n) / \partial w_{ji}(n)$. This gradient can be expressed as:

$$\frac{\partial \xi(n)}{\partial w_{ji}(n)} = \frac{\partial \xi(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)} \tag{3.6}$$

The partial derivative $\partial \xi(n) / \partial w_{ji}(n)$ determines the direction of search in weight space for the synaptic weight w_{ji}. Differentiating both sides of Equation (3.2) with respect to $e_j(n)$

$$\frac{\partial \xi(n)}{\partial e_j(n)} = e_j(n) \tag{3.7}$$

Differentiating both sides of Equation (3.1) with respect to $y_j(n)$,

$$\frac{\partial e_j(n)}{\partial y_j(n)} = -1 \tag{3.8}$$

Differentiating Equation (3.5) with respect to $v_j(n)$

$$\frac{\partial y_j(n)}{\partial v_j(n)} = \varphi_j'(v_j(n)) \tag{3.9}$$

Finally, differentiating Equation (3.4) with respect to $w_{ji}(n)$ yields

$$\frac{\partial v_j(n)}{\partial w_{ji}(n)} = y_i(n) \tag{3.10}$$

Substitute Equations (3.7) to (3.10) in (3.6) yields

$$\frac{\partial \xi(n)}{\partial w_{ji}(n)} = -e_j(n)\varphi_j'(v_j(n))y_i(n) \tag{3.11}$$

The correction $\Delta w_{ji}(n)$ applied to $w_{ji}(n)$ is defined by the delta rule:

$$\Delta w_{ji}(n) = -\eta \frac{\partial \xi(n)}{\partial w_{ji}(n)} \tag{3.12}$$

Where η is the learning-rate parameter of the back-propagation algorithm. The use of the minus sign in Equation (3.12) accounts for gradient descent in weight space $\xi(n)$. Substitute Equation (3.11) in (3.12) yields

$$\Delta w_{ji}(n) = \eta \delta_j(n) y_j(n) \tag{3.13}$$

where the local gradient $\delta_j(n)$ is defined by

$$\delta_j(n) = -\frac{\partial \xi(n)}{\partial v_j(n)} = -\frac{\partial \xi(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} = e_j(n)\varphi_j'(v_j(n)) \tag{3.14}$$

The local gradient points to required changes in synaptic weights. According to Equation (3.14), the local gradient $\delta(n)$ for output neuron j is equal to the product of the corresponding error signal $e_j(n)$ for that neuron and the derivative $\varphi_j'(v_j(n))$ of the associated activation function.

The key factor involved in the calculation of the weight adjustment $\Delta w_{ji}(n)$ is the error signal $e_j(n)$ at the output of neuron j. There are two cases, depending on where in the network neuron j is located. In the first case, neuron j is an output node. Each output node of the network is supplied with a desired response of its own, making it a straightforward matter to calculate the associated error signal. In the second case, neuron j is a hidden node. Even though hidden neurons are not directly accessible, they share responsibility for any error made at the output of the network. The question is to know how to penalize or reward hidden neurons for their share of the responsibility. This problem is the credit-assignment problem. It is solved in an elegant fashion by back-propagation the error signals through the network.

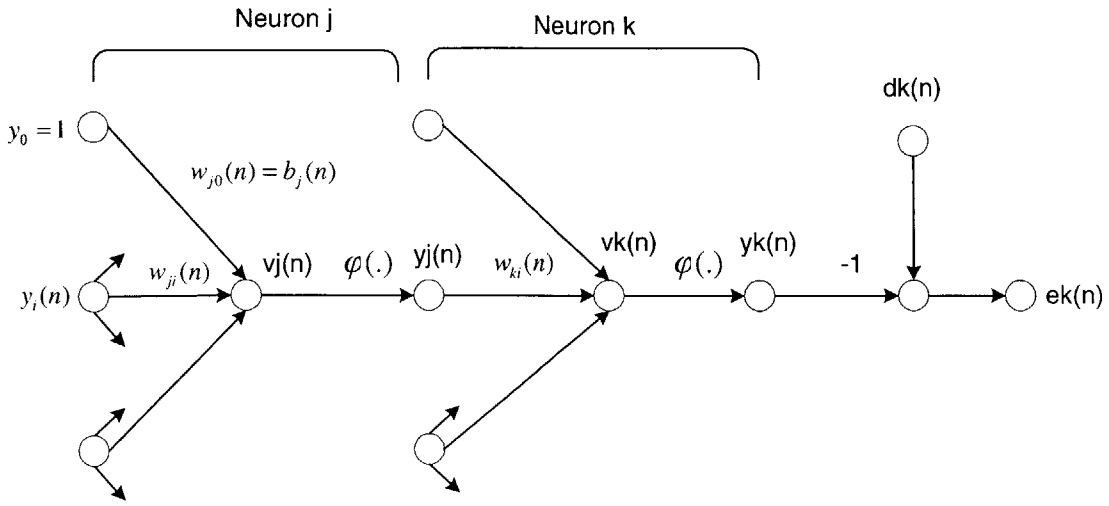


Figure 3.4 Signal-flow graph highlighting the details of output neuron *k* connected to hidden neuron *j*.

Neuron j is an Output Node

When neuron j is located in the output layer of the network, which means j = 0. It is supplied with a desired response of its own. The error signal $e_j(n)$ associated with this neuron is $e_j(n) = d_j(n) - y_j(n)$. Having determined $e_j(n)$, it is a straightforward matter to compute the local gradient $\delta_j(n)$ using Equation (3.14). The expression of the local gradient for neuron x in layer 0 is $\delta_x^0 = e_x^0 . f^{0'}(n_x^0)$.

Neuron j Is a Hidden Node

When neuron j is located in a hidden layer of the network, there is no specified desired response for that neuron. The error signal for a hidden neuron would have to be determined recursively in terms of the error signals of all the neurons to which that hidden neuron is directly connected; this is where the development of the back-propagation algorithm gets complicated. Consider the situation depicted in Fig. 3.4, which depicts neuron j as a hidden node of the network. According to Equation (3.14), it may redefine the local gradient $\delta_j(n)$ for hidden neuron j as

$$\delta_j(n) = -\frac{\partial \xi(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} = -\frac{\partial \xi(n)}{\partial y_j(n)} \varphi_j'(v_j(n)) \tag{3.15}$$

Use Equation (3.9) to calculate the partial derivative $\partial \xi(n) / \partial y_j(n)$,

$$\xi(n) = \frac{1}{2} \sum_{k \in C} e_k^2(n) \tag{3.16}$$

Equation (3.2) with index k used in place of index j. Differentiating Equation (3.16) with respect to the function signal $y_j(n)$

$$\frac{\partial \xi(n)}{\partial y_j(n)} = \sum_k e_k \frac{\partial e_k(n)}{\partial y_j(n)} \tag{3.17}$$

Use the chain rule for the partial derivative $\partial e_k(n) / \partial y_j(n)$, and rewrite Equation (3.17) in the equivalent form

$$\frac{\partial \xi(n)}{\partial y_j 9n)} = \sum_k e_k(n) \frac{\partial e_k(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial y_j(n)} \tag{3.18}$$

From Figure 4.4, it can be found that

$$e_k(n) = d_k(n) - y_k(n) = d_k(n) - \varphi_k(v_k(n)) \tag{3.19}$$

Therefore

$$\frac{\partial e_k(n)}{\partial v_k(n)} = -\varphi_k'(v_k(n)) \tag{3.20}$$

From Figure 3.4 that for neuron k the induced local field is

$$v_k(n) = \sum_{j=0}^{m} w_{kj}(n) y_j(n) \tag{3.21}$$

where m is the total number of inputs applied to neuron k. The synaptic weight $w_{k0}(n)$ is equal to the bias $b_k(n)$ applied to neuron k, and the corresponding input is fixed at the value +1. Differentiating Equation (3.21) with respect to $y_j(n)$ yields

$$\frac{\partial v_k(n)}{\partial y_j(n)} = w_{kj}(n) \tag{3.22}$$

Using Equations (3.20) and (3.22) in (3.18) we get the desired partial derivative:

$$\frac{\partial \xi(n)}{\partial y_j(n)} = -\sum_k e_k(n)\varphi_k'(v_k(n))w_{kj}(n) \tag{3.22}$$

Using Equations (3.20) and (3.22) in (3.18) the desired partial derivative:

$$\frac{\partial \xi(n)}{\partial y_j(n)} = -\sum_k e_k(n)\varphi_k'(v_k(n))w_{kj}(n) = -\sum_k \delta_k(n)w_{kj}(n) \tag{3.23}$$

The definition of the local gradient $\delta_k(n)$ given in Equation (3.14) with the index k substituted for j. Using Equations (3.20) and (3.22) in (3.18) we get the desired partial derivative:

$$\delta_j(n) = \varphi_j'(v_j(n))\sum_k \delta_k(n)w_{kj}(n) \tag{3.24}$$

Figure 4.5 shows the signal-flow graph representation of Equation (3.24), assuming that the output layer consists of m_L neurons.

The factor $\varphi_j'(v_j(n))$ involved in the computation of the local gradient $\delta_j(n)$ in Equation (3.24) depends solely on the activation function associated with hidden neuron j. The remaining factor involved in this computation, namely the summation over k, depends on two sets of terms. The first set of terms, the $\delta_k(n)$, requires knowledge of the error signals $e_k(n)$, for all neurons that lie in the layer to the immediate right of hidden neuron j, and that are directly connected to neuron j: see Figure 3.4. The second set of terms, the $w_{kj}(n)$, consists of the synaptic weights associated with these connections.

This is the relation for back-propagation algorithm. Firstly, the correction $\Delta w_{ji}(n)$ applied to the synaptic weight connecting neuron I to neuron j is defined by the delta rule:

$$(\Delta w_{ji}(n)) = (\eta)\cdot(\delta_j(n))\cdot(y_i(n)) \tag{3.25}$$

Which weight correction = learning-rate parameter times local gradient times input signal of neuron *j*. Secondly, the local gradient $\delta_j(n)$ depends on whether neuron *j* is an output node or a hidden node:

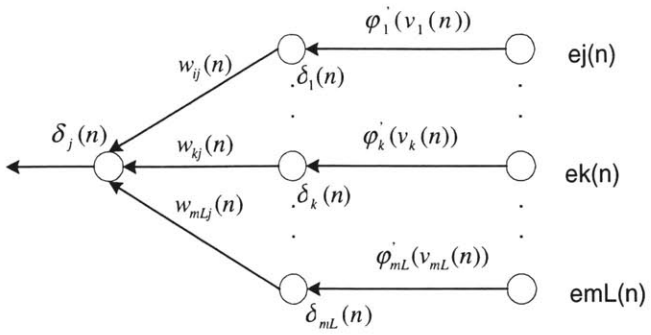


Figure 3.5 Signal-flow graph of a part of the adjoint system pertaining to back-propagation of error signals

Neuron *j* is an Output Node:

$\delta_j(n)$ equals the product of the derivative $\varphi_j'(v_j(n))$ and the error signal $e_j(n)$, both of which are associated with neuron *j*.

Neuron *j* is a Hidden Node:

$\delta_j(n)$ equals the product of the associated derivative $\varphi_j'(v_j(n))$ and the weighted sum of the δs computed for the neurons in the next hidden or output layer that are connected to neuron *j*.

The application of the back-propagation algorithm consists two distinguish passes of computation, forward pass and backward pass.

Forward Pass

In the forward pass, the weights remain unaltered throughout the network. The function signals appearing at the output of neuron *j* is

$$y_j(n) = \varphi(v_j(n)) \tag{3.26}$$

where $v_j(n)$ is the induced local field of neuron *j*

$$v_j(n) = \sum_{i=0}^{m} w_{ji}(n) y_i(n) \tag{3.27}$$

where m is the total number of inputs applied to neuron j, and $w_{ji}(n)$ is the synaptic weight connecting neuron i to neuron j, and $y_i(n)$ is the input signal of neuron j. If neuron j is in the first hidden layer

$$y_i(n) = x_i(n) \tag{3.28}$$

where $x_i(n)$ is the ith element of the input vector. If neuron j is the output layer of the network

$$y_i(n) = o_i(n) \tag{3.29}$$

where $o_i(n)$ is the jth element of the input vector. This output is compared with the desired response $d_j(n)$, obtaining the error signal $e_j(n)$ for the jth output neuron. Thus the forward pass begins at the first hidden layer by presenting it with the input vector, and terminates at the output layer by computing the error signal for each neuron of this layer.

Backward Pass

Backward pass, starts at the output layer by passing the error signals through the network, and recursively computing the local gradient δ for each neuron. This process permits the synaptic weights of the network to undergo changes in accordance with the delta rule of Equation (3.25). First, use Equation (3.25) to compute the changes to the weights of all the connections feeding into the output layer. Second, use Equation (3.24) to compute δ for all neurons in the penultimate layer. This recursive computation is continued, layer by layer, by propagating the changes to all synaptic weights in the network.

Learning Rate

The back-propagation algorithm provides an approximation in weight space computed by the method of steepest descent. From one iteration, the smaller the learning-rate η, the smaller the changes to the synaptic weights will be, and the smoother will be in weight space. However it will have a slower rate of learning. If the learning-rate η is too large, the large changes in the synaptic weights may become unstable.

It was assumed that the learning-rate parameter is a constant. In fact, the learning-rate parameter should be connection-dependent. In the application of the back-propagation algorithm, the synaptic weights may be adjustable, or weights can be fixed during the adaptation.

3.1.6 Procedure of the Back-Propagation Algorithm

The algorithm cycles through the training sample as follows:

1. Initialization.

Pick the synaptic weights and thresholds from a uniform distribution whose mean is zero and whose variance is chosen to make the standard deviation of the induced local fields of the neurons lie at the transition between the linear and saturated parts of the sigmoid activation function.

2. Presentations of Training Examples.

Present the network with training examples. For each example set, ordered it, perform the sequence of forward and backward computations respectively.

3. Forward Computation.

With the input vector $x(n)$ applied to the input layer of sensory nodes and the desired response vector $d(n)$ presented to the output layer of computation nodes. Compute the induced local fields and function signals of the network by proceeding forward through the network. The induced local field $v_j^{(l)}(n)$ for neuron j in layer l is

$$v_j^{(l)}(n) = \sum_{i=0}^{m_0} w_{ji}^{(l)}(n) y_i^{(l-1)}(n) \tag{3.30}$$

where $y_i^{(l-1)}(n)$ is the output signal of neuron i in the previous layer l-1 at iteration n and $w_{ji}^{(l)}(n)$ is the synaptic weight of neuron j in layer l that is fed from neuron i in layer l-1.

For $i=0$, $y_0^{(l-1)}(n) = +1$ and $w_{j0}^{(l)}(n) = b_j^{(l)}(n)$ is the bias applied to neuron j in layer l. Use a sigmoid function, the output signal of neuron j in layer l is

$$y_j^{(l)} = \varphi_j(v_j(n)) \tag{3.31}$$

If neuron j is in the first hidden layer

$$y_j^{(0)}(n) = x_j(n) \tag{3.32}$$

where $x_j(n)$ is the jth element of the input vector $x(n)$. If neuron j is in the output layer

$$y_j^{(L)} = o_j(n) \tag{3.33}$$

Compute the error signal

$$e_j(n) = d_j(n) - o_j(n) \tag{3.34}$$

where $d_j(n)$ is the jth element of the desired response vector $d(n)$.

4. Backward Computation.

Compute the δ of the network, for neuron j in output layer L

$$\delta_j^{(l)}(n) = e_j^{(L)}(n)\varphi_j'v_j^{(L)}(n)) \tag{3.35}$$

For neuron j in hidden layer l

$$\delta_j^{(l)}(n) = \varphi_j'(v_j^{(l)}(n))\sum_k \delta_k^{(l+1)}(n)w_{kj}^{(l+1)}(n) \tag{3.36}$$

where the prime in $\varphi_j'(\cdot)$ denotes differentiation with respect to the argument. Adjust the synaptic weights of the network in layer l according to the generalized delta rule:

$$w_{ji}^{(l)}(n+1) = w_{ji}^{(l)}(n) + \alpha[w_{ji}^{(l)}(n-1)] + \eta\delta_j^{(l)}(n)y_i^{(l-1)}(n) \tag{3.37}$$

where η is the learning-rate parameter and α is the momentum constant.

5. Iteration

Iterate the forward and backward computations under previous two procedures by presenting new epochs of training examples to the network until the stopping criterion is met.

3.1.7 Improve the Performance of Back-Propagation Algorithm

There are some methods that will significantly improve the back-propagation algorithm's performance:

1. Sequential or Batch Update

The sequential mode of back-propagation learning is computationally faster than the batch mode. Especially when the training data set is large and highly redundant.

2. Maximizing Information Content

Every training example should be chosen on the basis that its information content is the largest possible for the task at hand. There are two ways to approach. First, use an example that results in the largest training error. Second, use of an example that is radically different from all those previous used. One technique is to randomize the order in which the examples are presented to the multilayer perceptron from one epoch to the next.

3. Activation Function

A multilayer perceptron trained with back-propagation algorithm can learn faster when the sigmoid activation function built into the neuron model of the network is antisymmetric than when it is nonsymmetric. A popular example of an antisymmetric activation function is a sigmoid nonlinearity in the form of a hyperbolic tangent, $\varphi(v) = 1.7159\tanh(0.6667v)$, (LeCun, 1989, 1993).

4. Target Values

The target value should be chosen within the range of the sigmoid activation function. The desired response d_j for neuron j in the output layer of the multilayer perceptron should be offset by some amount away from the limiting value of the sigmoid activation function, depending on whether the limiting value.

5. Normalizing the Inputs

Each input variable should be preprocessed so that its mean value is close to zero. There are two measures can accelerate the back-propagation learning process. The input variables contained in the training set should be uncorrelated. The decorrelated input variables should be scaled so that their covariances are approximately equal, ensuring that the different synaptic weights in the network learn at approximately the same speed.

6. Initialization

It is important to choose good initial values of the synaptic weights and thresholds of the network. It is desirable for the uniform distribution, from which the synaptic weights are selected, to have a mean of zero and a variance equal to the reciprocal of the number of synaptic connections of a neuron.

7. Learning from Hints

Learning from a set of training examples deals with an unknown input-output mapping function. The process of learning from examples may be generalized to include learning from hints, which is achieved by allowing prior information that we may about the function to be included in the learning process.

8. Learning Rates

All neurons in the multilayer perceptron should ideally learn at the same rate. The last layers usually have larger local gradients than the layers at the front end of the network. The learning-rate should be assigned a smaller value in the last layers than in the front layers. Neurons with many inputs should have a similar learning time for all neurons in the network. For a given neuron, the learning rate should be inversely proportional to the square root of synaptic connections made to that neuron.

3.2 Conjugate Gradient Learning Algorithm

The back-propagation learning algorithm is widely used for training multilayer neural networks. However, back-propagation has a slow rate of learning. Some more effective neural network learning algorithms have to be developed. These learning algorithms improve the convergence rate and reduce the total number of iterations and the execution time. Kollias and Anastassiou (1989) developed an adaptive least squares learning algorithm for multilayer neural networks. Douglas and Meng (1991) developed an adaptive linearized least squares learning algorithm for training of multilayer feedforward neural networks. These two algorithms achieved better convergence rate than the momentum back-propagation learning algorithm by using second order derivatives of the error function with respect to the network weights. However, in both algorithms the Hessian matrix containing the second order derivatives of the network weights, requiring a large amount of memory storage and additional computations. These two algorithms are efficient only when the input data set is small.

The conjugate gradient method, originally proposed by Fletcher and Reeves (1964), has been recognized as one practical method for solving large optimization problems, because it does not require any large matrix storage and its iteration cost is relatively low.

3.2.1 Concept of Conjugate Gradient Algorithm

The basic back-propagation algorithm adjusts the weights (step size) in the steepest descent direction. This is the direction in which the performance function is decreasing most rapidly. This causes a problem, although the function decreases most rapidly along the negative of the gradient, this does not necessarily produce the fastest convergence. However, in the conjugate gradient algorithms a search is performed along conjugate directions, which produces generally faster convergence than steepest descent directions.

In back-propagation algorithm, a learning rate is used to determine the length of the weight update. In conjugate gradient algorithms the weight update size is adjusted at each iteration. A search is made along the conjugate gradient direction to determine the step size, which will minimize the performance function along that line.

3.2.2 Explanation of the Conjugate Gradient Algorithm

The conjugate gradient method belongs to a class of second-order optimization methods known collectively as conjugate direction methods. Considering the minimization quadratic function

$$f(x) = \frac{1}{2} x^T A x - b^T x + c \tag{3.38}$$

where x is a W-by-1 vector, A is a W-by-W matrix, b is a W-by-1 vector, and c is a scalar. Minimization of the quadratic function $f(x)$ is achieved by assigning to x the unique value

$$x^* = A^{-1} b \tag{3.39}$$

Thus minimizing $f(x)$ and solving the linear system of equations $Ax^* = b$ are equivalent problems. Given the matrix A, a set of nonzero vectors *s(0), s(1),...s(W-1)* is A-conjugate if the following condition is satisfied:

$$s^T(n) A s(j) = 0 \tag{3.40}$$

for all n and j such that $n \neq j$. If A is equal to the identity matrix, conjugacy is equivalent to the usual notion of orthogonality.

3.2.3 Procedure of the Conjugate Gradient Algorithm

Initialization

Choose the initial value $w(0)$ using a procedure similar to that described for the back-propagation algorithm.

Computation

1. For $w(0)$, use back-propagation to compute the gradient vector $g(0)$.
2. Set $s(0) = r(0) - g(0)$.
3. At time step n, use a line search to find $\eta(n)$ that minimize $\xi_{av}(\eta)$ sufficiently, representing the cost function ξ_{av} expressed as a function of η for fixed values of w and s.
4. Test to determine if the Euclidean norm of the residual $r(n)$ has fallen below a specified value $\|r(0)\|$.
5. Update the weight vector: $w(n+1) = w(n) + \eta(n)s(n)$
6. For $w(n+1)$, use back-propagation to compute the updated gradient vector $g(n+1)$.
7. Set $r(n+1) = -g(n+1)$
8. Use the Polak-Ribiere method to calculate $\beta(n+1)$:

$$\beta(n+1) = \max\left\{\frac{r^T(n+1)(r(n+1) - r(n))}{r^T(n)r(n)}, 0\right\} \tag{3.41}$$

9. Update the direction vector:

$$s(n+1) = r(n+1) + \beta(n+1)s(n) \tag{3.42}$$

10. Set $n = n+1$, and go back to step 3.

Stopping Criterion

Terminate the algorithm when the following condition is satisfied:

$$\|r(n)\| \le \varepsilon\|r(0)\| \tag{3.43}$$

where ε is a prescribed small number.

3.2.4 An Adaptive Conjugate Gradient Algorithm

The conjugate gradient method is an effective modification of the steepest descent method proposed by Fletcher and Reeves (1964) and modified by Polak and Ribiere(1969). In 1986, Powell showed that the unrestarted Polak-Ribiere method with exact line search may fail to converge for non-convex problems, and proposed a more robust algorithm to ensure convergence. By using the Powell's modified conjugate gradient algorithm for minimizing the system error in neural networks with the inexact line search algorithm, here is an adaptive conjugate gradient neural network-learning algorithm.

Inexact Line Search Algorithm

The step size determination has a great effect on the efficiency of the mathematical optimization algorithm. An inexact line search algorithm can determine the search step size λ within a small percentage of its true value. In order to ensure that the selected step size λ is not too large, it has to satisfy the following condition:

$$E(W^{(k)} + \lambda d^{(k)}) \leq E(W^{(k)}) + \beta\lambda(\nabla E(W^{(k)})^T d^{(k)}) \tag{3.44}$$

$\beta \in (0,1), \lambda > 0$, In order to ensure that the selected step size λ is not too small, it has to satisfy the following condition:

$$\nabla E(W^{(k)} + \lambda d^{(k)})^T d^{(k)} \geq \theta(\nabla E(W^{(k)})^T d^{(k)}) \tag{3.45}$$

$\theta \in (\beta,1), \lambda > 0$, the condition $1 > \theta > \beta > 0$, guarantees the above two condition can be satisfied simultaneously. However, the above two conditions do not guarantee that descent directions are always generated. The following condition ensures that the selected step size satisfies the descent condition (Nocedal, 1990).

$$\nabla E(W^{(k)} + \lambda d^{(k)})^T d^{(k+1)} < 0 \tag{3.46}$$

The acceptable step size, λ, is located in a region that satisfies the three conditions. This inexact line search algorithm is based on the above three conditions and backtracking by successive parabolic and cubic interpolations.

3.2.5 Procedure of the Adaptive Conjugate Gradient Algorithm

There are L decision variables.

1. Generate initial weight vector $W \in R^L$ randomly. Set the iteration counter $n=1$, set the convergence parameter $\varepsilon = 10^{-6}$, maximum number of iterations, and the minimum system error. Set the initial search direction $d^{(0)} = \{0\}$. Set STOP1=0. Set the acceptable minimum and maximum step size as minlen and maxlen. Set β, δ.

2. For $k=1$ *to* p, perform the following for the kth training instance:

   2.1 Perform feedforward procedure of the neural network

$$\text{Set } O_k^{(1)} = \begin{bmatrix} X_k \\ 1 \end{bmatrix}$$

   For $i=1$ *to* $N-1$, calculate the output vector in $(i+1)$st layer:

$$O_k^{(i+1)} = \begin{bmatrix} G(W^{(i)} \cdot O_k^{(i)}) \\ 1 \end{bmatrix}$$

$$G(W^{(i)} \cdot O_k^{(i)}) = \begin{bmatrix} \dfrac{1}{1+e^{-\sum_{j=1}^{n_i+1}(w_{1,j}^{(i)} \cdot o_{kj}^{(i)})}} \\ \dfrac{1}{1+e^{-\sum_{j=1}^{n_i+1}(w_{2,j}^{(i)} \cdot o_{kj}^{(i)})}} \\ \vdots \\ \dfrac{1}{1+e^{-\sum_{j=1}^{n_i+1}(w_{n_{i+1},j}^{(i)} \cdot o_{kj}^{(i)})}} \end{bmatrix}$$

   2.2 Calculate the system error for the kth training instance:

$$E_k(X_k, W) = \sum_{m=1}^{n_N}(y_{km} - o_{km})^2$$

   2.3 Calculate the deltas in the output layer for the kth training instance:

$$\delta_{kr}^{(N)} = (y_{kr} - o_{kr}^{(N)})(1 - o_{kr}^{(N)})o_{kr}^{(N)}, r = 1,2,...,n_N$$

   2.4 For $r=N-1$ *down to* 1, calculate the deltas in the hidden layers:

$$\delta_{kr}^{(N)} = (y_{kr} - o_{kr}^{(N)})(1 - o_{kr}^{(N)})o_{kr}^{(N)}, q = 1,2,...,n_r$$

2.5 For *i=1 to N-1*, calculate the gradient vector for the *k*th training instance:

$$\nabla E_k(W^{(n)}) = \frac{\partial E(W)}{\partial w_{q,r}^{(i)}} = \delta_{kq}^{(i+1)} o_{kr}^{(i)}, q = 1,2,...,n(i+1).and.r = 1,2,...,n_i + 1$$

3. Calculate the total system error:

$$E(X_k, W) = \frac{1}{2p}\sum_{k=1}^{p} E_k(X_k, W)$$

If $E(X_k, W) <$ miner, set STOP1=1 and go to step 19, Otherwise, go to next step.

4. Calculate the gradient vector of the total neural network system error:

$$\nabla E(W^{(n)}) = \sum_{k=1}^{p} \nabla E_k(W^{(n)})$$

Assign the search direction as $d^{(n)} = -\nabla E(W^{(n)})$. If $\left|\nabla E(W^{(n)}) < \varepsilon\right|$, set STOP1=1 and go to step 19. In this case, $W^{(n)}$ is the optimum solution. Otherwise, continue.

5. Ste *iter=iter+1*. If *iter>L*, set iter=0. If *iter=1*, set $\alpha_n = 0$ and go to the next step. Otherwise, calculate the new conjugate direction as:

$$d^{(n)} = -\nabla E(W^{(n)}) + \alpha_n d^{(n-1)}$$

where $\alpha_n = \max\left\{0, \frac{\nabla E(W^{(n)})^T v^{(n-1)}}{\left|\nabla E(W^{(n-1)})\right|^2}\right\}$

and $v^{(n-1)} = \nabla E(W^{(n)}) - \nabla E(W^{(n-1)})$

6. Perform the inexact line search algorithm to calculate λ. Set the stopping criterion STOP2=0. Initialize λ=1.

7. Calculate $E(W^{(n)} + \lambda d^{(n)})$ and $E(W^{(n)}) + \beta\lambda(\nabla E(W^{(n)})^T d^{(n)})$. If $E(W^{(n)} + \lambda d^{(n)}) \leq E(W^{(n)}) + \beta\lambda(\nabla E(W^{(n)})^T d^{(n)})$, go to the next step. Otherwise, go to step 15.

8. Calculate $\nabla E(W^{(n)} + \lambda d^{(n)})^T d^{(n)})$ and $\theta(\nabla E(W^{(n)})^T d^{(n)})$. If $\nabla E(W^{(n)} + \lambda d^{(n)})^T d^{(n)} < \theta(\nabla E(W^{(n)})^T d^{(n)})$ go to the next step. Otherwise, go to step 13.

9. If $\lambda = 1$, go to step 10. Otherwise, go to step 11.

10. Set $\lambda = \min(2\lambda, maxlen)$.
    Calculate the new search direction $d^{(n+1)}$.
    Calculate $\nabla E(W^{(n)} + \lambda d^{(n)})^T d^{(n+1)}$.
    If $(\nabla E(W^{(n)} + \lambda d^{(n)})^T d^{(n+1)}) < 0$
    Calculate $\nabla E(W^{(n)} + \lambda d^{(n)})^T d^{(n)}$ and $\theta(\nabla E(W^{(n)})^T d^{(n)}$.
    If $(\nabla E(W^{(n)} + \lambda d^{(n)})^T d^{(n+1)}) \geq 0$
    Or $(\nabla E(W^{(n)} + \lambda d^{(n)})^T d^{(n)}) \geq \theta(\nabla E(W^{(n)})^T d^{(n)})$, or $1 > maxlen$, go to step 11. Otherwise, go to step 10.

11. If $\lambda < 1$ or ($\lambda > 1$ and $(\nabla E(W^{(n)} + \lambda d^{(n)})^T d^{(n+1)}) \geq 0$), go to step 12. Otherwise, go to step 17.

12. Perform backtracking using parabolic interpolation to find a new λ.
    Calculate $\nabla E(W^{(n)} + \lambda d^{(n)})^T d^{(n)}$ and $\theta(\nabla E(W^{(n)})^T d^{(n)}$.
    Calculate the new search direction $d^{(n+1)}$
    Calculate $\nabla E(W^{(n)} + \lambda d^{(n)})^T d^{(n+1)}$

If both conditions, $(\nabla E(W^{(n)} + \lambda d^{(n)})^T d^{(n)}) \geq \theta(\nabla E(W^{(n)})^T d^{(n)})$ and $\nabla E(W^{(n)} + \lambda d^{(n)})^T d^{(n+1)} < 0$ hold simultaneously, set STOP2=1 and go to step 17. Otherwise, go to the beginning of this step.

13. Calculate the new search direction $d^{(n+1)}$.

    Calculate $\nabla E(W^{(n)} + \lambda d^{(n)})^T d^{(n+1)}$.

    If $\nabla E(W^{(n)} + \lambda d^{(n)})^T d^{(n+1)} \geq 0$, go to step 14. Otherwise, set STOP2=1 and go to step 17.

14. Perform backtracking using parabolic interpolation to find a new λ.

    Calculate the new search direction $d^{(n+1)}$.

    Calculate $\nabla E(W^{(n)} + \lambda d^{(n)})^T d^{(n+1)}$.

    $\nabla E(W^{(n)} + \lambda d^{(n)})^T d^{(n+1)} < 0$, set STOP2=1 and go to step 17. Otherwise, go to the beginning of this step.

15. If $\lambda <$*minlen*, then set $\lambda =$*0*, set STOP2=1 and go to step 17. Otherwise, go to the next step.

16. If $\lambda =$*1*, perform backtracking using parabolic interpolation to find a new λ. Otherwise, perform backtracking using cubic interpolation to find a new λ. Go to the next step.

17. If STOP2=1, set $\lambda_n = 1$, stop the iterations of inexact line search algorithm, and go to the next step. Otherwise, go to step 7.

18. Update the weight vector as

    $W^{(n+1)} = W^{(n)} + \lambda_n d^{(n)}$

    Set $n=n+1$. If $n>i$, set STOP1=1 and go to next step.

19. If STOP1=1, stop the iteration of the adaptive conjugate gradient learning algorithm; $W^{(n)}$ is the optimum weight vector. Otherwise, go to step 2.

The algorithm is restarted every L iterations by setting $\alpha_k = 0$.

The performance of the algorithm was evaluated as following. The problem of arbitrary trial-and-error selection of the learning ratio λ and momentum ratio α encountered in the back-propagation algorithm is circumvented in the new adaptive algorithm. Instead of constant learning and momentum ratios, the step size in the inexact line search is adapted during the learning process through a mathematical approach. The adaptive algorithm provides a more solid mathematical foundation for neural network learning. Also, this algorithm converges much faster than the back-propagation algorithm.

3.3 Optimum Design of Neural Networks

Barai and Pandey (1994) proposed a conclusion that issues will affect the design performance of a neural network. Figure 3.6 is their proposed neural network design paradigm. Selecting an optimal neural network architecture depends on the application domain. The successful application of neural networks to a specific problem depends mainly on two factors, *representation* and *learning*. Choosing a topology (input/output units, number of hidden units per layer, number of hidden layers, etc) and training parameters (learning parameters μ, momentum parameter α, error tolerance, etc) are very much context-dependent and usually arrived at by trial-and-error. There are some issues will effect the performance of a neural network.

1. Choosing Input/Output nodes

Every training example will decide the number of input nodes, and the corresponding desired output parameter gives the number of nodes in the output layer.

2. Training Patterns

It is very important to present a good training set in network learning and the decision is very critical. If a small percentage of the resulting generalization may be poor, while in the opposite case it is likely that higher oscillation would make it impossible to reach a state of global minima.

3. Normalization of the Training Set

The input patterns must be normalized before being given to the network. This gives an advantage over the size of the network.

4. Number of Hidden Layers in the Network

It has been mentioned that two to three layers are sufficient for most problems. However, the optimal number of layers will dependent on different applications. It is suggested that multilayer networks with linear neurons are equivalent to two-layer networks. Hence, the various weight matrices can be combined into a single matrix, which serves the same purpose as a multilayer network with linear neurons.

5. Number of Neurons in the Hidden Layers

How many hidden neurons should be used in a layer is arbitrary, and has been usually decided by trail-and-error. It is good enough to use the average of the number of input and output neurons. Another possibility is to make the hidden layer of the same size as either the input or the output layer. The fewer hidden neurons the fewer connections, and hence less training capacity. Generally the hidden layer should not be the smallest layer in the network, nor should it be the largest.

6. Choosing Training Parameters

The training parameters are arrived at by investigating the application domain. Though these parameters have generally been frozen in several investigations, it would be desirable to carry out a further study of these parameters in order to see their influence in the context of the application.

7. Choosing the Activation Function

There are several types of activation functions, linear, linear threshold, step, sigmoid, and Gaussian activation functions. With the exception of the linear activation functions, all these functions introduce a nonlinear in the network dynamics by bounding the output values within fixed ranges. The sigmoid function (S-shaped semi-linear or squashing function) has been recommended in most of the back-propagation applications. In a sigmoid function the output is a continuous, monotonic function of the input. The function itself and its derivatives are continuous everywhere.

8. Choosing the Average System Error

The acceptable average system error depends upon the amount of accuracy required for training and testing the network. The acceptable error plays an important role in determining the number of training cycles, and finally it has an impact on the training time. The best way to choose the average system error is to start with a large value of the average system error and watch the performance of the network. Then, depending upon the accuracy required from the network, reduce the value of average system error. The initial large value of average system error also helps in determining the possibility of the network's convergence for a small value of the average system error.

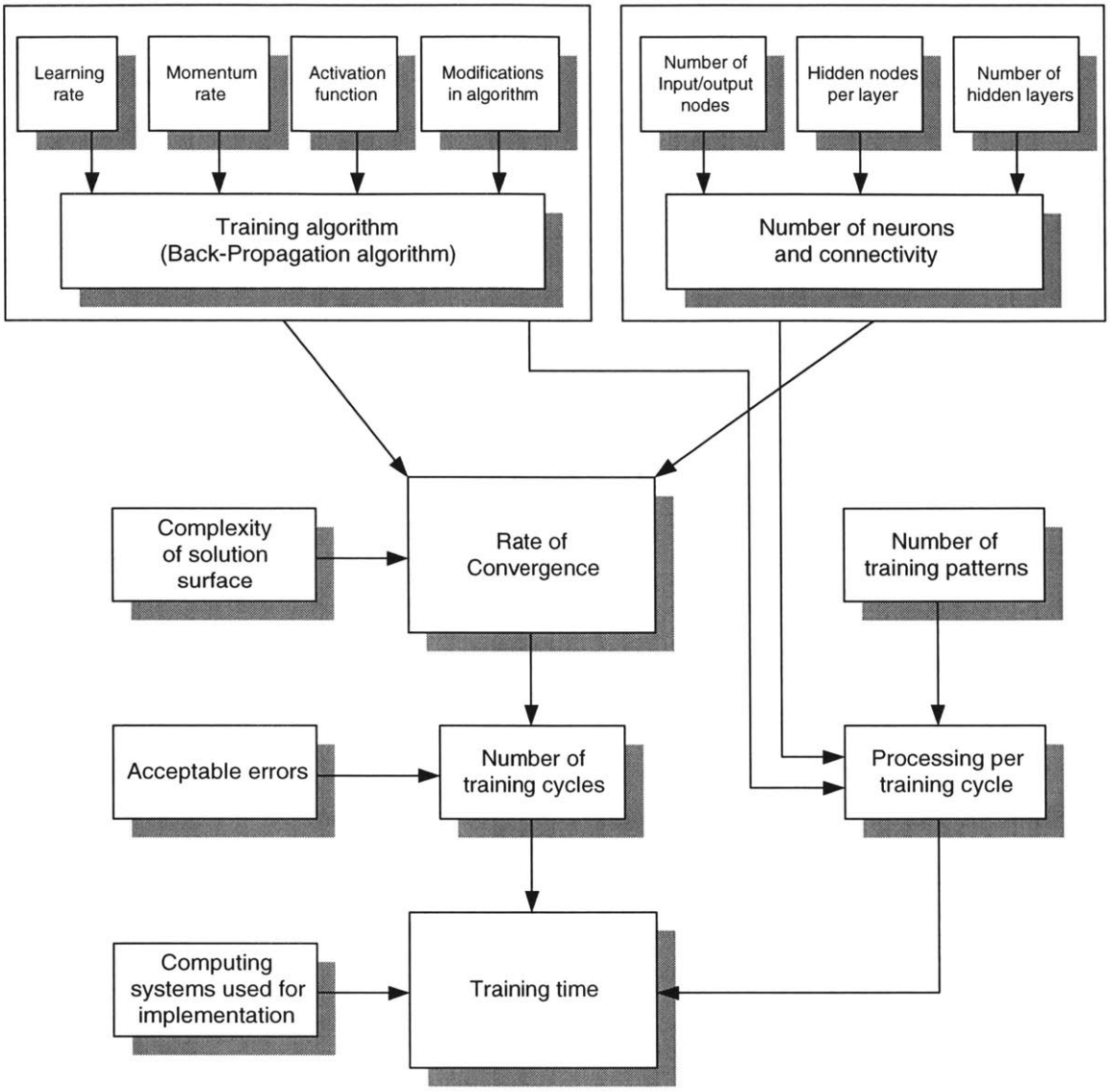


Figure 3.6 Neural-network training paradigm.

Chapter 4

Architecture of a Structural Self-Diagnosis Java Program

4.1 Object-Oriented Software Design by using Java

4.1.1 Procedural Programming Approach

The conventional software development method is called procedure programming. This kind of software system is treated as a set of data representing information and procedures that manipulate the data. The process to solve the problems is formulated using a sequence of commands. Data and procedures are tailored to fit each other but they are independent entities. The programmers have to define the command procedures to data and ensure that the procedure will work correctly on the used data types. As a result, designing program in this way ends up with complexity, mess, and difficulty to maintain software. Moreover, there is another disadvantage, this kind of software cannot be reused by other programs.

4.1.2 Object-Oriented Programming Approach

In the conventional procedure programming design, computer code and data are treated separately. However, in an object-oriented programming model, there is no separation of code and data. Therefore, the programming code and data are integrated in a single entity, which is called object. In a conventional programming design, the programmers have to

specify explicitly the inputs and outputs for the program. In contrast, in an object-orient program, the programmer is concerned about the functions each object is expected to perform.

An object-oriented programming model can be developed for the evaluation of performance of structure damages. It can be implemented in the programming language Java or C++. Java and C++ provide the object-oriented programming concepts such as inheritance, encapsulation, and polymorphism.

The main concepts in an object-oriented programming model are "objects", "classes", "encapsulation", "inheritance", and "polymorphism".

Classes

Classes are the blueprints that are used to create objects. A class defines the attributes and behaviors that each object creates from the class will possess. Creating new classes involves a two-part process: First, define the attributes that objects created from the class will use to store the state. Second, define the messages to let the objects understand. For each message, create a procedure, called a *method* that implements these steps. Class is a user-defined data type in Java and C++. The class construct provides the basic foundation for object-oriented programming. Each class may consist of not only primitive data types such as integer, short, long, float, or double, but also user-defined data types such as array, or inner class. Classes act as templates for objects with a particular set of properties and methods. All instances of a given class have the same properties and methods, but the value of each property may vary.

Objects

In the conventional procedure programming, procedures are used to build structured programs. In contrast, in object-oriented programming, objects are used to build object-oriented programs. An object-oriented program is a collection of objects that are organized for, and cooperate toward, the accomplishment of some goal. Each object not only contains data but also has a set of defined behaviors and an individual identity.

Objects are instances of a class. An object is a self-contained entity and consists of properties and methods.

Encapsulation

Encapsulation is a concept that will let programmer create well- designed classes. Encapsulation is the process of packaging the program, dividing each of its classes into two distinct parts: the interface and the implementation. The objects are made of attributes and methods. Some of these attributes and methods are publicly available, visible from outside the object: These are the interface. Other attributes and methods are reserved for the private use of the object itself: These are the implementation. Separating the interface from the implementation is the most fundamental design decision when the programmers design an object-oriented program. The private data of an object is not accessible to any other object in the program. This concept is the key to object-oriented programming. Unlike the conventional programming design concepts, where data and procedures operating on the data have to be declared and represented separately, object-oriented programming encapsulations the specific data with the procedures operating on the data. Thus, based on this concept, the programmer can represent the real world by objects, design operating procedures in each object, and describe the behavior of each object. The procedures attached to a class are called methods in Java. Methods may be declared in public in a class; they can operate on the data declared private in a class. When a specific method is called, the object class searches its public methods and executes the operation using the object's data as input. Since methods are mostly declared in public in a class, they can be accessed by other object classes.

Inheritance

Encapsulation is necessary for creating robust classes that can be maintained and changed easily. Inheritance is concerned with a group of classes and their relationships. Classes can be created in a hierarchy with inheritance property. In Java, object classes can be organized into a hierarchical taxonomy by using the two features of "base class" and "derived class". A derived class inherits all the properties and methods of its base class.

In Java, a derived class can only inherit on base class. However, a derived class can implement several *interfaces*.

Polymorphism

Polymorphism works together with encapsulation and inheritance to simplify the flow of control in an object-oriented program. When a message is sent to an object, that object must have a method defined to response to that message. When classes are connected in an inheritance hierarchy, all the subclasses of a parent class automatically inherit their parent's interface. Anything that a superclass object can do, a subclass object can also do. Although a subclass object responds to the same messages that a superclass object does, the message need not trigger the same behavior. It simply needs to be understood. Each subclass can rely on the superclass to define the appropriate response or define a new, specialized response. Therefore, each of the subclass is able to respond differently-polymorphically or according to its nature, so to speak-to the same message. Another important concept is late biding. Normally, when a compiler for a non- object-oriented language comes across a method invocation, it determines exactly what target code should be called and builds machine language to represent that call. In object-oriented language, this is not possible since the proper code to invoke is determined based upon the class of the object being used to make the call, not the type of the variable. Instead, code is generated that will allow the decision to be made at runtime. Java's Virtual Machine has been designed from the start to support an object-oriented programming system, so there are machine-level instructions for making methods calls. The compiler only needs to prepare the argument list and produce one method invocation instruction; the job of identifying and calling the proper target code is performed by the Virtual Machine. If the Virtual Machine is to be able to decide what actual code should be invoked by a particular method call, it must be able to determine the class of the object upon which the call is based. Unlike traditional languages or runtime environments, every time the Java system allocates memory, it marks that memory with the type of the data that it has been allocated to hold. This means that given any object, and without regard to the type associated with the reference variable acting as a handle to that object, the runtime system can determine the real class of that object by inspection.

4.1.3 Object Orient Analysis and Design

There are three components corresponding to analysis, design, and implementation/ programming. Analysis deals with the problem domain and design with the solution domain. Object oriented analysis focuses on problem domain objects and object oriented design on solution objects. Figure 4.1 shows the relationship between analysis and design. The problem and solution domain representations are different and smaller than the real world problem or in the case of engineering applications the mathematical model. Moreover the solution domain includes everything in the problem domain plus any additional constructs required by the solution.

Analysis involves problem definition and modeling. Object orient analysis models the problem domain by identifying and specifying a set of semantic objects that interact and behave according to system requirements. Problem domain objects represent things or concepts used in describing the problem rather then its solution. They are called mantic objects because they have meaning in the problem domain. During analysis the focus is on representing the problem and identifying abstractions. The semantic classes may then be extended if useful abstractions are discovered.

Design focuses on solution specification and modeling object oriented design transforms the problem representation into a solution representation. The solution domain includes, but is not only limited to, the semantic objects. During design the emphasis is on defining a solution. Object oriented design models the solution domain, which includes semantic classes with possible additions and interface, application and utility objects identified during the design process. Interface objects are associated with user interface. They are not directly part of the problem domain. They represent the user's view of the semantic objects. In an object-oriented environment such objects are mainly part of graphical user interface. Application objects can be thought of as the control mechanisms for the system. They are mainly objects that start the application and perhaps control the sequencing of high-level functions. These categories are also called design components. Object-oriented design should be still language-independent. It precedes physical design.

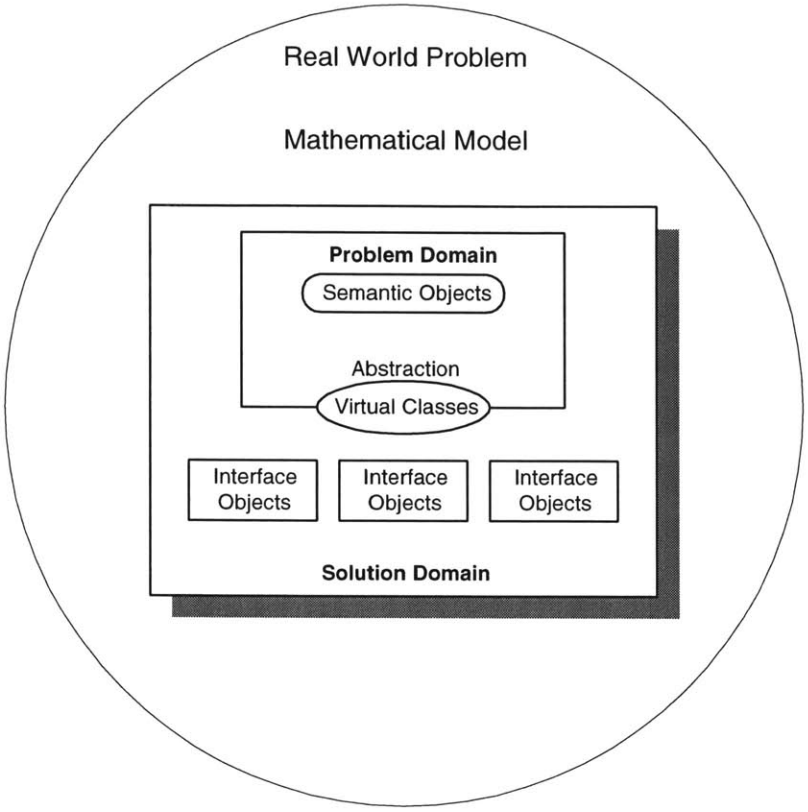


Figure 4.1 Problem and solution domain objects and classes.

4.2 Data Structures in Java

4.2.1 Reference in Java

All the types in Java are *reference* except primitive types. Reference types include strings, arrays, and file streams. A *reference* in Java is a variable that stores the memory address where an object resides. A reference will always store the memory address where some object is residing, unless it is not currently referencing any object. In this case, it will store the *null reference*, null. Java does not allow references to primitive variables. There are two broad categories of operations that can be applied to reference variables. One allows us to examine or manipulate the reference value. The other category of operations applies to the object being referenced.

4.2.2 Array and Linked Lists in Java

An *array* is the basic mechanism for storing a collection of identically typed entities. In Java the array is not a primitive type. Instead, it behaves very much like an object. A *linked list* is a sequence of elements arranged one after another, with each element connected to the next element by a link. A programming practice is to place each element together with the link to the next element, resulting in a component called a *node*. There are certain operations that are better performed by arrays and others where linked lists are preferable. Arrays are better at random access. Linked lists are better at additions or removals at a cursor. Doubly linked lists are better for a two-way cursor.

4.2.3 Binary Trees in Java

A *binary tree* is a finite set of nodes. The set might be empty, which is called empty tree. It follows these rules. First, there is one special node, called the *root*. Each node may be associated with up to two other different nodes, called its *left child* and its *right child*. Second, Each node, except the root, has exactly one parent; the root has no parent. Third, If starting at a node and move to the node's parent, then move again to that node's parent, and keep moving upward to each node's parent, it will eventually reach the root. Here is the Java code fragment of a binary tree node.

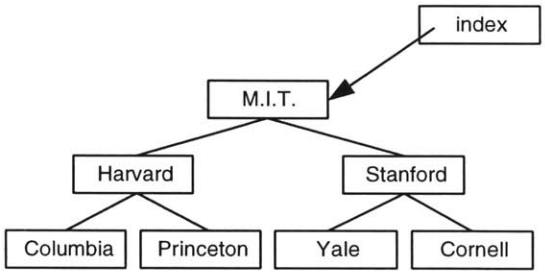
```
public class BbinaryTreeNode{
   private Object data;
   private BinaryTreeNode left;
   private BinaryTreeNode right;
}
```

Here is a constructor for the BinaryTreeNode class. It has three arguments, which are the initial values for the node's data and link variables:

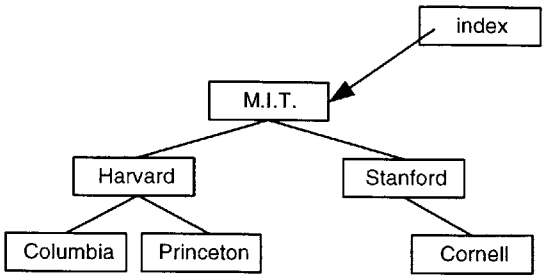
```
public BbinaryTreeNode(
   Object initialData;
   BinaryTreeNode initialLeft;
   BinaryTreeNode initialRight;
)
{
   data = initialData;
   left = initialLeft;
   right = initialRight;
}
```

4.2.4 Binary Trees Operations in Java

Here is a binary tree, the node index refer to the root node of this tree.



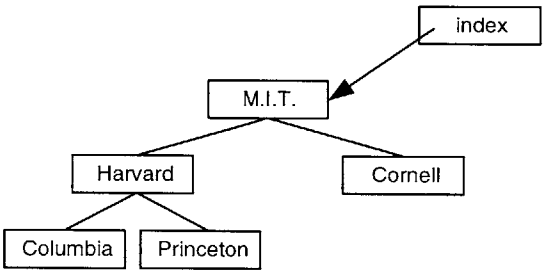
If we remove the leftmost node in index's right subtree, which is the node containing "Yale". Right now, index.getRight() is a reference to the "Stanford" node, therefore, index.getRight().removeLeftmost() will remove "Yale" and have a return value that is the reference to the root of the new smaller subtree. We have to set index's right link to the new smaller tree, so the complete statement to remove the leftmost node in index's right subtree is index.setRight(index.getRight().removeLeftmost());. The resulting tree will look like this:



Notice that the sets index's right link to the root of the new smaller subtree. This is important because in some cases the root of the subtree might now be null or it could be a different node. For example, we want to remove the leftmost node of index's right subtree a second time. When we activate:

```
index.setRight(index.getRight().removeLeftmost());
```

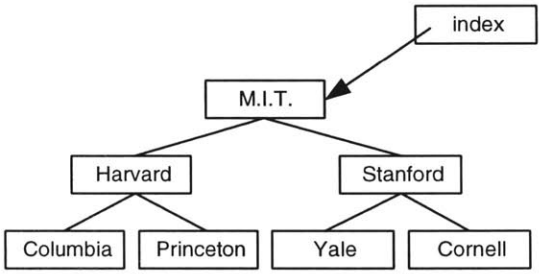
The leftmost node of the right subtree now contains "Stanford" itself, so the activation of index.getRight().removeLeftmost() will remove the "Stanford" node and return a reference to the new smaller subtree, which contains only "Cornell". The right link of index is set to refer to the new smaller tree, so we end up with this situation:



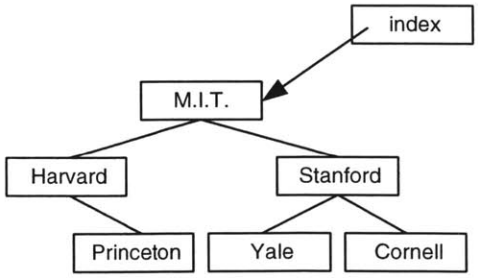
The implementation of removeLeftmost has a simple case when the node that activates removeLeftmost has no left child. This is the situation that we just saw, where the "Stanford" node had no left child. In this case, the node that we simply return a reference to the rest of the tree, which is on the right side. Thus, the implementation begins like this:

```
public BbinaryTreeNode removeLeftmost()
{
   if (left == null)
      return right;
   ...
}
```

This code returns a reference to the node that contains "Cornell". However, in the case that is a left child, we want to remove the leftmost node from the left subtree. For example, suppose that the root of this tree activates removeLeftmost:



We must remove the leftmost node from the tree. Since the root has a left child, we can accomplish our task by removing the leftmost node form the left subtree. This is a smaller version of the very problem that we are trying to solve, and we can solve this smaller problem by a recursive call of left.removeleftmost(). The recursive call will remove the leftmost node from the left subtree. This new smaller left subtree may have a different root than the original left subtree, so we need to remove the leftmost node from the left subtree and set the left link to this new smaller tree is: left = left.removeLeftmost(); This is a recursive call because we are using removeLeftmost to solve a smaller version of the removeLeftmost problem. After the recursive call, the tree looks like this:



In this recursive solution, there is one last task. We must return a reference to the entire tree. This entire tree is now smaller than the tree that we began with. The root to this smaller tree is the very node that activated removeLeftmost in the first place. In Java, the keyword "this" is always a reference to the object that activated the method. So, removeLeftmost uses "this" as shown in the following complete implementation.

```
public BbinaryTreeNode removeLeftmost()
{
   if (left == null)
      return right;
   else{
      left = left.removeLeftmost();
      return this;
   }
}
```

Notice the final line in the recursive case "return this" which returns a reference to the original node that activated the method.

4.3 Architecture of Cantilever Beam Damage Self-Diagnosis Java Program

Figure 4.2 shows the running procedure of the structural self-diagnosis Java program. There are two main performances in this program, neural network learning and neural network analysis. This program consists two neural network learning algorithms, back-propagation learning algorithm and conjugate gradient learning algorithm. When learning, the program takes the training data set from the input file, and then it performs the training procedure for the neural network. Finally, this program produces two output files; they are weights of neurons and neural network architecture data. When analysis, the program takes the analysis data set form the input file, and it produces the analysis result to an output file, which contains the system errors.

There are three classes in this structural self-diagnosis Java program; they are class Ann, class DiaplayListener, and class AnimationPanel. Class Ann consists all the neural network performance functions. This class also contains the main function. Class DisplayLister consists all the events handling functions. Class AnimationPanel consists all the display routine functions. Figure 4.3 shows the relationship between the class and the functions. Figure 4.4 shows the screen shot of the structural self-diagnosis Java program.

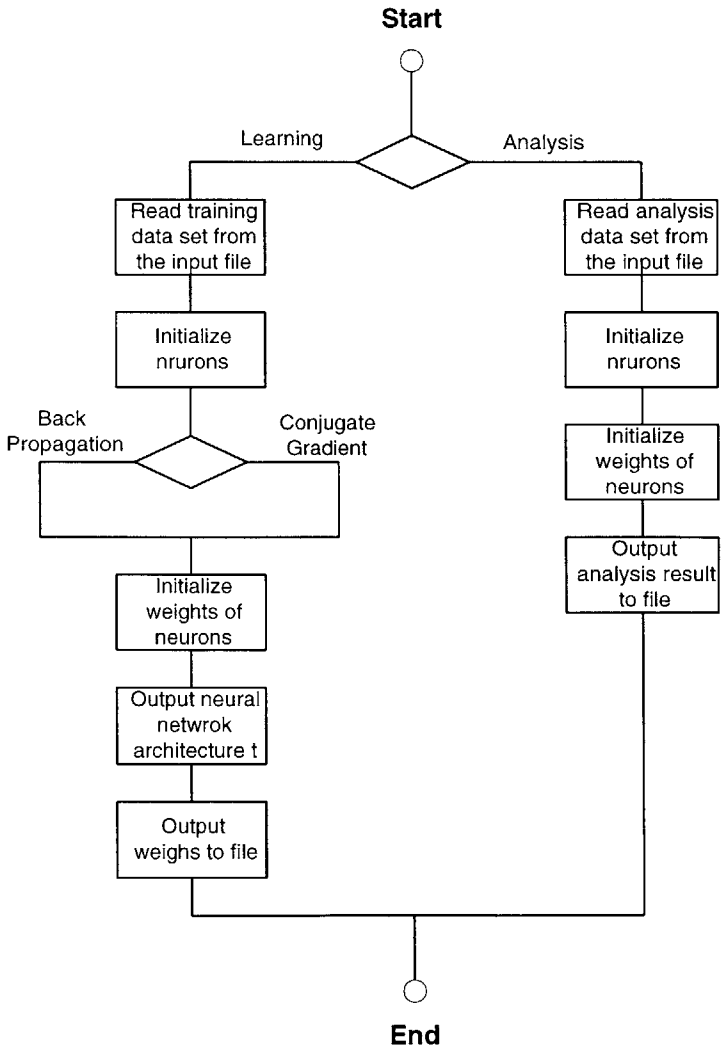


Figure 4.2 Flow chart of the Structural Self-Diagnosis Java Program

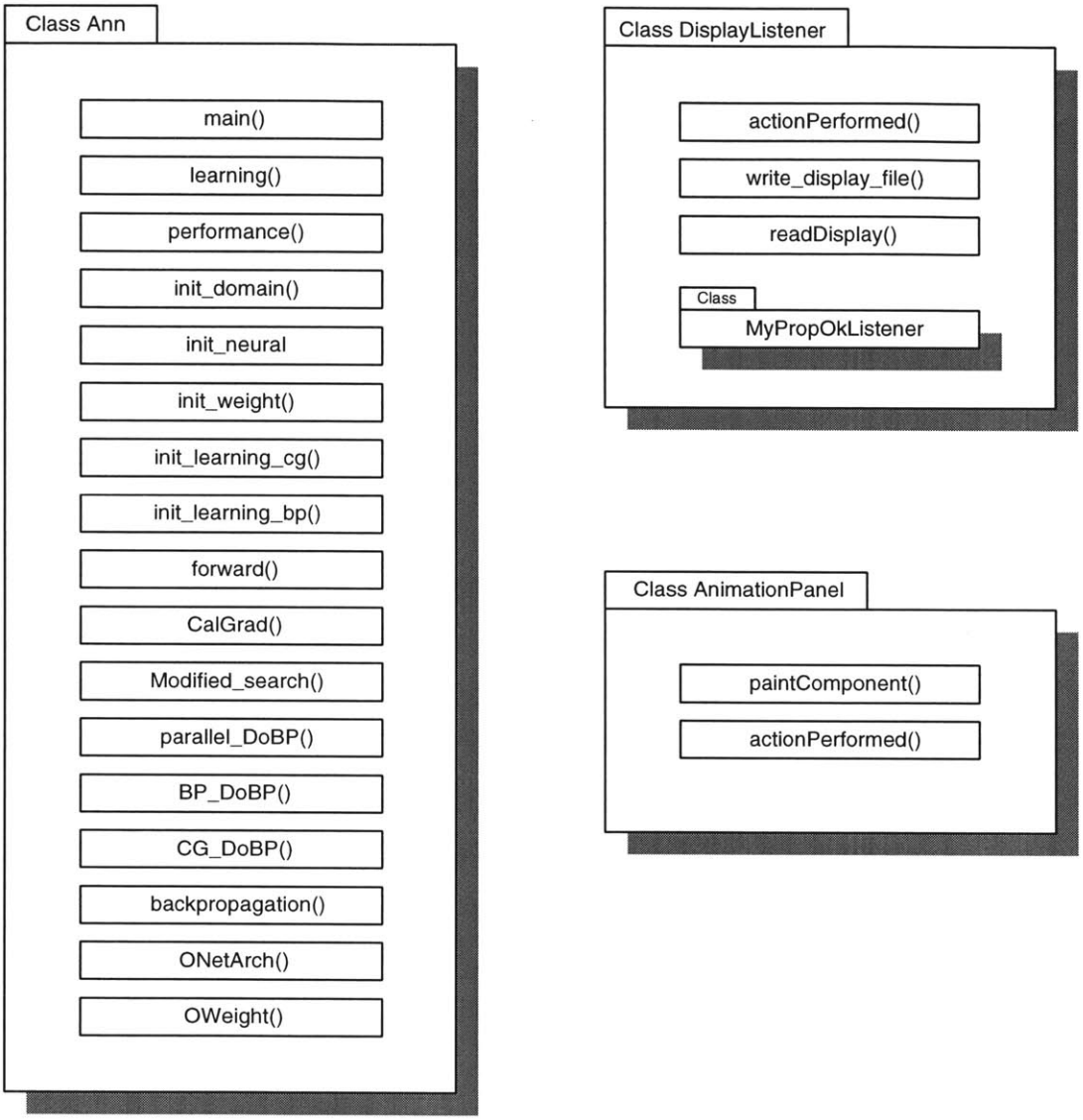


Figure 4.3 Class Relationship of the Structural Self-Diagnosis Java Program

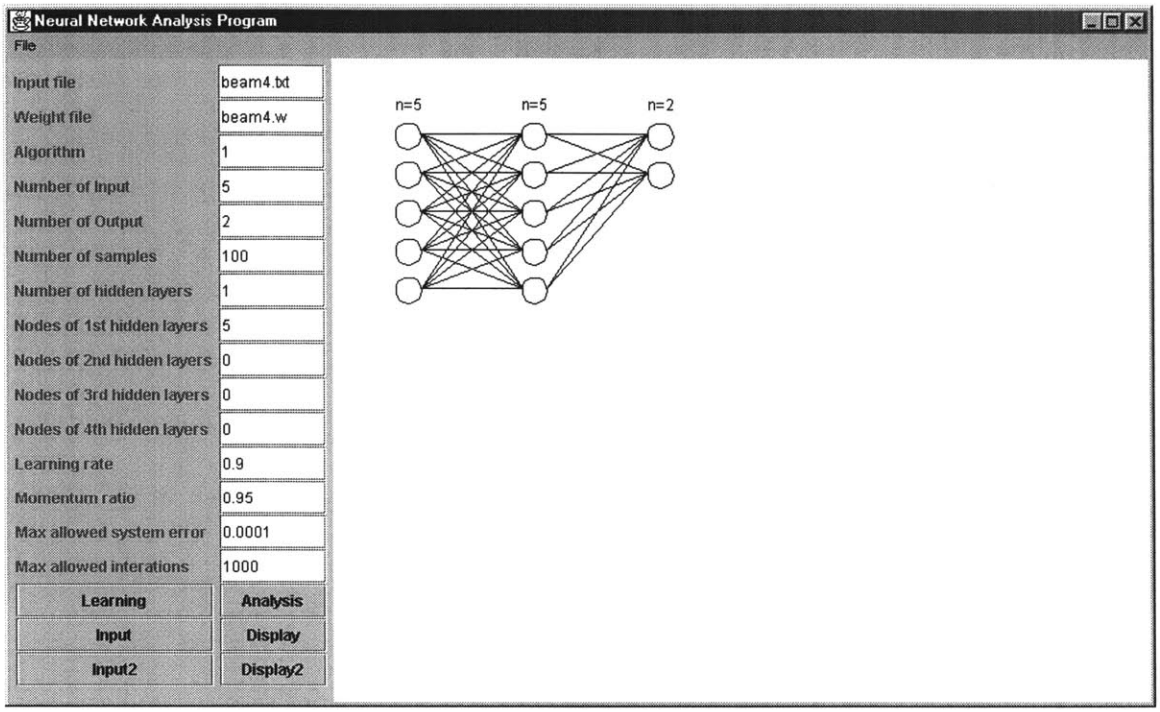


Figure 4.4 Screen Shot of the Structural Self-Diagnosis Java Program

Chapter 5

Case Study: Crack Self-Diagnosis of a Cantilever Beam

5.1 Neural Network Based Inverse Analyses

5.1.1 Introduction of Neural Network Based Inverse Analyses

Computational mechanics has been developed dramatically. The use of distributed systems and parallel computers have made it possible to analyze large-scale three-dimensional structural problems over millions degree of freedom. In the field of structural vibration analyses, it is difficult to perform accurate and reliable vibration analyses of a whole complex structural system because of poor modeling techniques. In the field of vibration analyses, "structural self-diagnosis" means a process to build a mathematical model which well-describes vibration characteristics of actual mechanical and structural components. This is a typical inverse problem. This modeling process significantly influences numerical accuracy as well as computation time. Various structure identification methods can be roughly classified into the following three categories: theoretical methods, experimental methods, and hybrid theoretical and experimental methods.

For the pas two decades, the neural networks analyses have been developed and applied to the field of structural self-identification. Among various neural network architectures, the multilayer neural networks have the following advantages:

1. It can automatically construct a non-linear mapping function from multiple input data to multiple output data within the network in a distributed manner through a training process with many training patterns.
2. The trained network has a generalization feature, that is, a kind of interpolation, such that the well trained network estimates appropriate output data even for untrained patterns.
3. The trained network operates quickly in an application process. Computational resources required for operating the trained network may be equivalent to only that of a personal computer.

The neural networks have been applied in various structural engineering problems. One of the applications is to develop an inverse analysis approach using the multilayer neural networks and the computational mechanics, and applied the approach to several inverse problems. The present inverse analysis approach basically consists of the following three subprocesses. First, parametrically varying model parameters of a system, their corresponding responses of the system are calculated through computational mechanics simulations, each of which is an ordinary direct analysis. Each data pair of model parameters vs. system responses is called training pattern. Second, a neural network is trained using a number of training patterns. Here the system responses are given to the input units of the network, while the model parameters to be identified are shown to the network as teacher data. Finally, some system responses measured are given to the trained network, which immediately outputs appropriate model parameters even for untrained patterns.

5.1.2 Fundamental Principle of Neural Network Based Inverse Analysis

The neural network based inverse analysis approach can be defined as follows:

Stage 1: Prepare a number of training patterns that are parameters like cantilever beam vibration frequencies and tip displacement.

Stage 2: Train the neural network using the training patterns. When the back-propagation algorithm is used, the network topology is determined a priori through trial-and-error.

Stage 3: Input a set of measured data to the input units of the well-trained network. Then the network immediately estimates their corresponding crack parameters.

Figure 5.1 illustrates the analysis procedure. The most important and time-consuming optimization process is invoked only once in the training process of the network at the second stage, and the algorithm of training process is independent of a physical problem to be solved. As the result, the present approach can be easily applied to any kinds of inverse problems if a sufficient number of training patterns are available through computational mechanics simulations. Another key feature of the present approach is that once a training process finishes, one can solve quickly the inverse problems for various combinations of crack parameters, which exist within a data space over all the training patterns. The neural network approach looks like a database of finite element solutions with an inverse analysis capability.

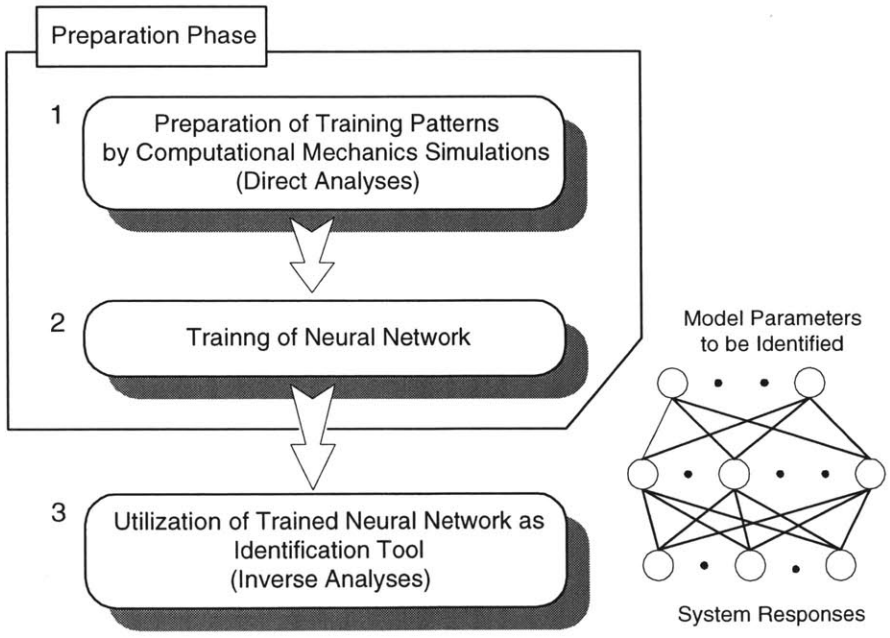


Figure 5.1 Flow of the inverse analysis approach

5.2 Neural Network Systems for Structural Damage Self Diagnosis

Peetathawatchai (1996) proposed a neural network based diagnosis system in his Ph.D. thesis. Figure 5.2 shows the architecture of a basic neural network based diagnosis system. There are four major components: the numerical model, data processing unit, a neural network for detecting the damage location (NNET1), and a neural network for detecting the damage extent (NNET2). Network NNET1 can define the location of the damage in the structure. Network NNET2 can define the extent of damage at the given damage location. The training procedure for NNET1 is shown on Figure 5.3. For each training cycle, the time response of the model corresponding to each damage condition is determined via simulation and is passed to the data preprocessing unit, which transforms the time response data into a normalized input pattern. Figure 5.4 shows the training procedure for NNET2. The output of NNET2 is used to predict the extent of damage at the given damage location.

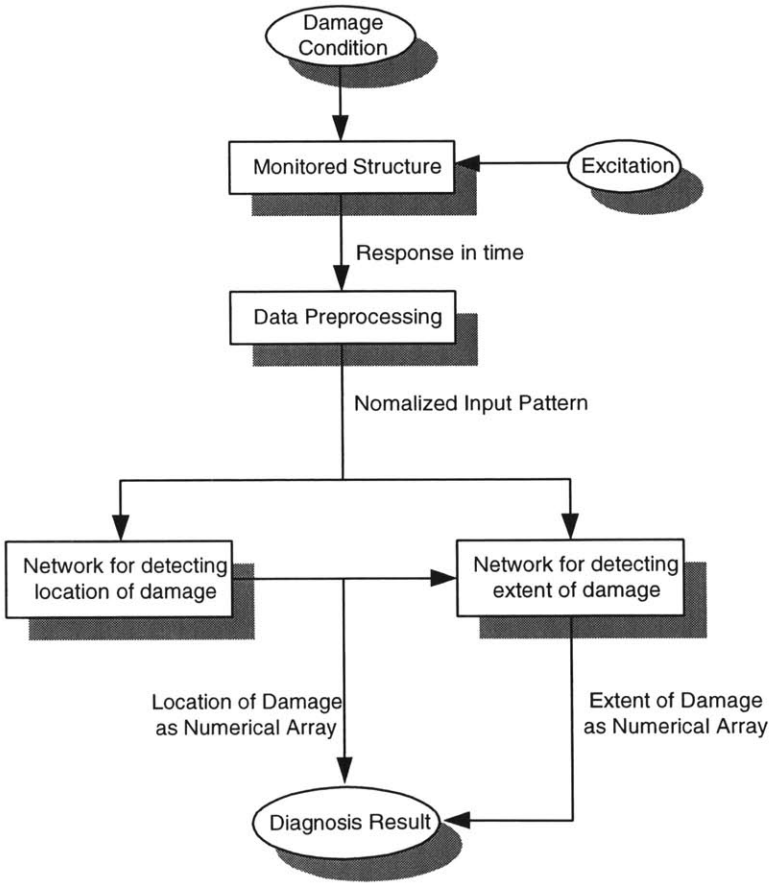


Figure 5.2 Neural Network Based Diagnosis System.

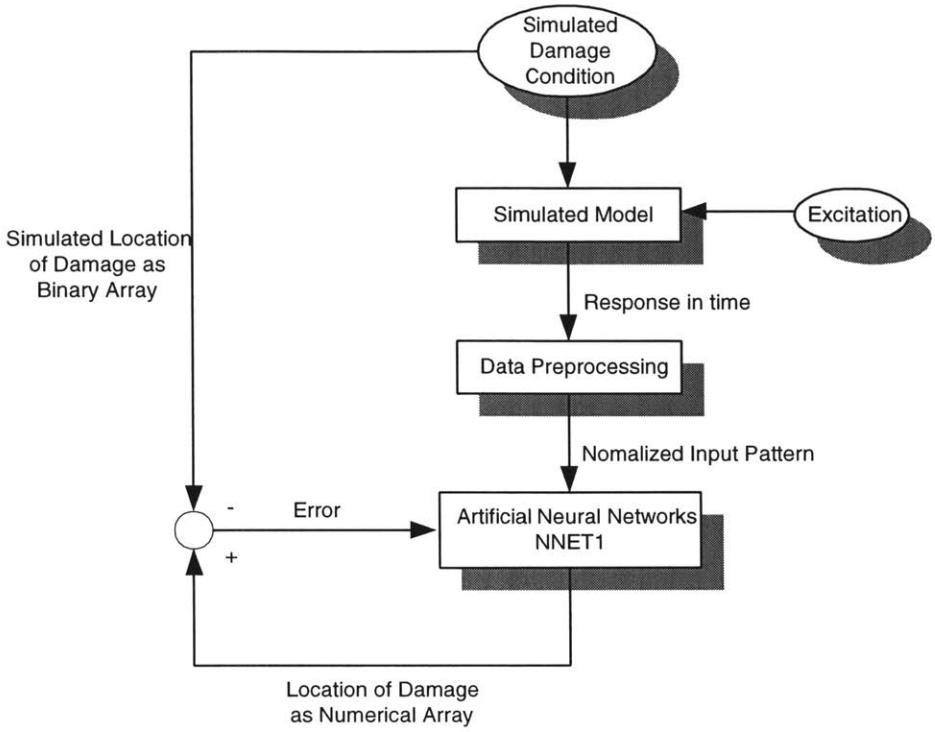


Figure 5.3 The Training Process of Neural Network for Detecting Location of Damage

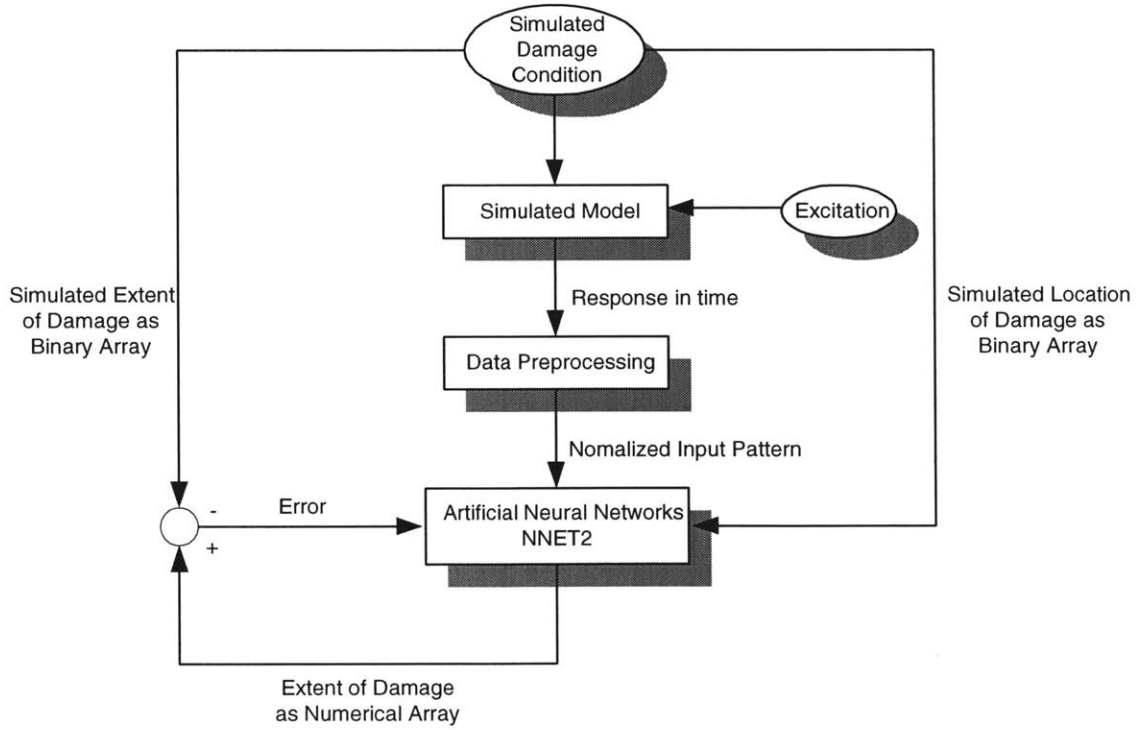


Figure 5.4 The Training Process of Neural Network for Recognizing of Damage

5.3 Formulation of Cantilever Beam Tip Displacement

The displacement models have been produced using a linear model of a cracked cantilever beam based on previously published studies (Rizos, Aspragathos, and Dimarogonas, 1990; Ostachowiez and Krawezuk, 1991). A crack is modeled as a linear rotational spring, as shown in Figure 5.2. The equivalent torsional stiffness at the crack location is calculated using stress intensity factors for an open single-sided crack (Anifantis and Dimarogonas, 1983; Ostachowiez and Krawezuk, 1991).

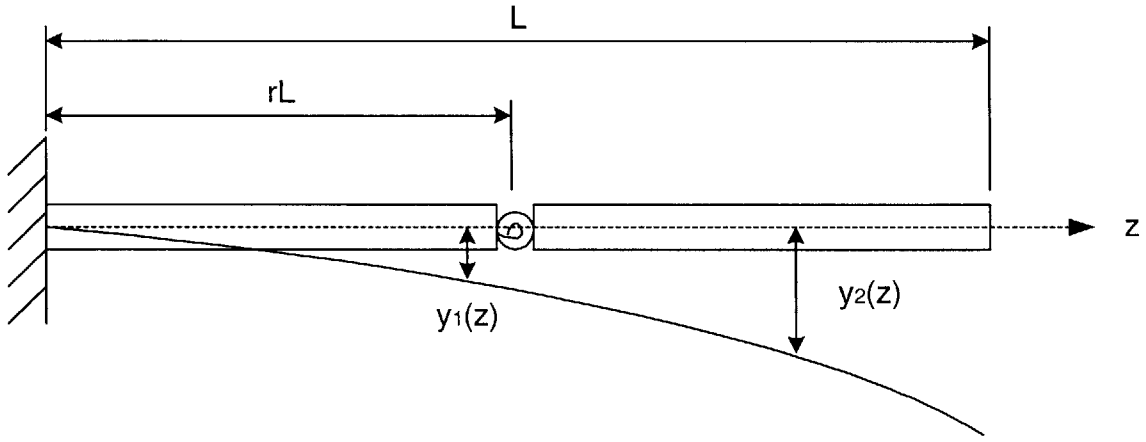


Figure 5.5 Model of a cracked beam.

The displacement functions for both parts of the beam can be expressed as:

$$y_1(z) = a_1 \cosh(kz) + a_2 \sinh(kz) + a_3 \cos(kz) + a_4 \sin(kz) \tag{5.1}$$

where $y_1(z)$ is the displacement function from the root to the crack.

$$y_2(z) = b_1 \cosh(kz) + b_2 \sinh(kz) + b_3 \cos(kz) + b_4 \sin(kz) \tag{5.2}$$

where $y_2(z)$ is the displacement function from the crack to the tip.

where

$$k = L\left(\frac{w_n^2 \rho A}{EI}\right)^{1/4} \tag{5.3}$$

The following boundary conditions describe the investigated beam:

$$y_1(0) = 0 \tag{5.4}$$

$$\frac{dy_1(0)}{dz} = 0 \tag{5.5}$$

$$y_1(r) = y_2(r) \tag{5.6}$$

$$\frac{dy_1(r)}{dz} = \frac{dy_2(r)}{dz} - q\frac{d^2 y_2(r)}{dz^2} \tag{5.7}$$

$$\frac{d^2 y_1(r)}{dz^2} = \frac{d^2 y_2(r)}{dz^2} \tag{5.8}$$

$$\frac{d^3 y_1(r)}{dz^3} = \frac{d^3 y_2(r)}{dz^3} \tag{5.9}$$

$$\frac{d^2 y_2(1)}{dz^2} = 0 \tag{5.10}$$

$$\frac{d^3 y_2(1)}{dz^3} = 0 \tag{5.11}$$

where

$$q = \frac{6H}{L(\gamma^2(72.21 - 117.1\gamma + 420.7\gamma^2 - 585.5\gamma^3 + 854.2\gamma^4 - 829.3\gamma^5 + \gamma^6))} \tag{5.12}$$

and

$$\gamma = \frac{d}{H} \tag{5.13}$$

The fundamental frequencies and tip displacements corresponding to a given combination of a nondimensional crack location r and crack size d can be obtained from equation (5.4)-(5.11). The fundamental frequencies are found by equating the determinant of the coefficient matrix to zero and solving the resulting characteristic equation for k. Tip

displacements are obtained as follows. Equations (5.11) is modified to account for the tip driving force of magnitude F

$$\frac{d^3 y_2(1)}{dz^3} = \frac{FL^3}{EI} \tag{5.14}$$

and Equation (5.3) is modified to account for the excitation frequency ω

$$k = L\left(\frac{\omega^2 \rho A}{EI}\right)^{1/4} \tag{5.15}$$

The system of Equations (5.4)-(5.11) is solved for coefficients a and b. Tip displacements are determined by computing the value of y_2 at $z=L$.

5.4 Problem Statement of Single/Double Cracks Cantilever Beam

5.4.1 Definition of Single Crack Cantilever Beam

Assuming that the cantilever beam is modeled as an assemblage of equal section quality elements, and the crack means changing the moment of inertia of specific element. The moment of inertia is reduced in the crack area.

In the case of single point crack, the cantilever beam is 21 feet long, and it is divided to 21 equal length sections. Each section has the same length. The 10^{th} element from the left is damaged. The moment of inertia, I, of this element is reduced to different levels. Figure 5.3 shows the location of the damaged element. Figure 5.4 illustrates the different reduction of *moment of inertia* in this damaged element.

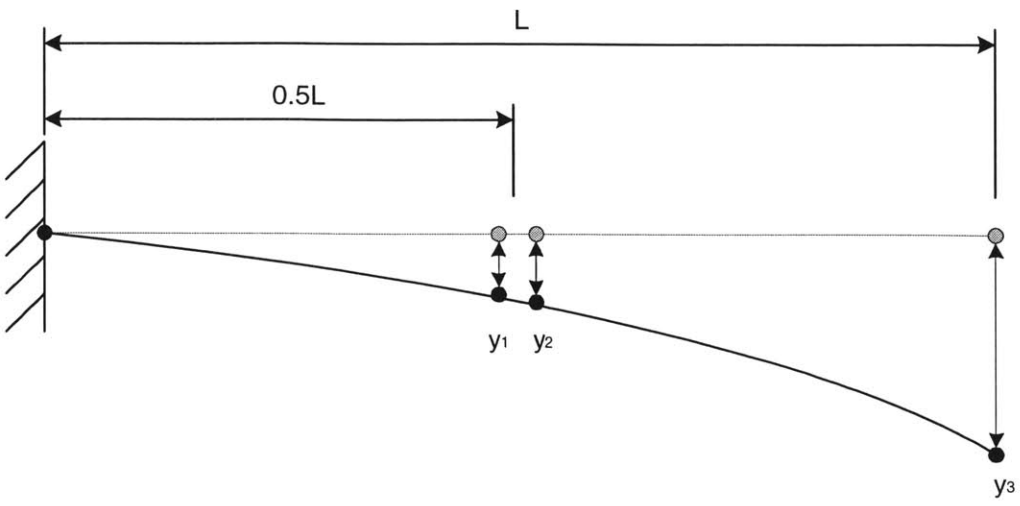


Figure 5.6 Single Crack Cantilever Beam

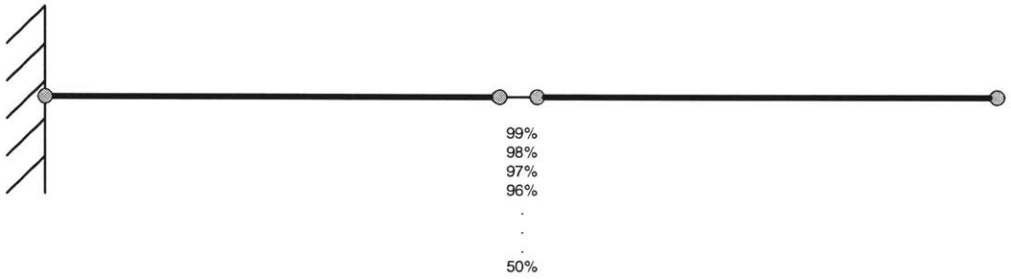


Figure 5.7 Single Crack Cantilever Beam Reduction of Moment of Inertia

5.4.2 Definition of Double Cracks Cantilever Beam

By definition, double-point damage refers to the case where cracks occur at two different locations. As described before, the cantilever beam is modeled as an assemblage of equal section quality elements, and the crack is simulated by the change of the mechanical properties of specific elements.

In the case of the double-point damage, the cantilever beam is 21 feet long, and it is divided to 21 equal length sections. Each section has the same length. There are two damaged elements in the cantilever beam. The second element from the left is damaged, and the 10^{th} element from the left is also damaged. The moment of inertia, I, of these two elements is reduced to different levels. Figure 5.3 shows the location of the damaged element. Figure 5.4 illustrates the reduction of *moment of inertia* in the damaged elements.

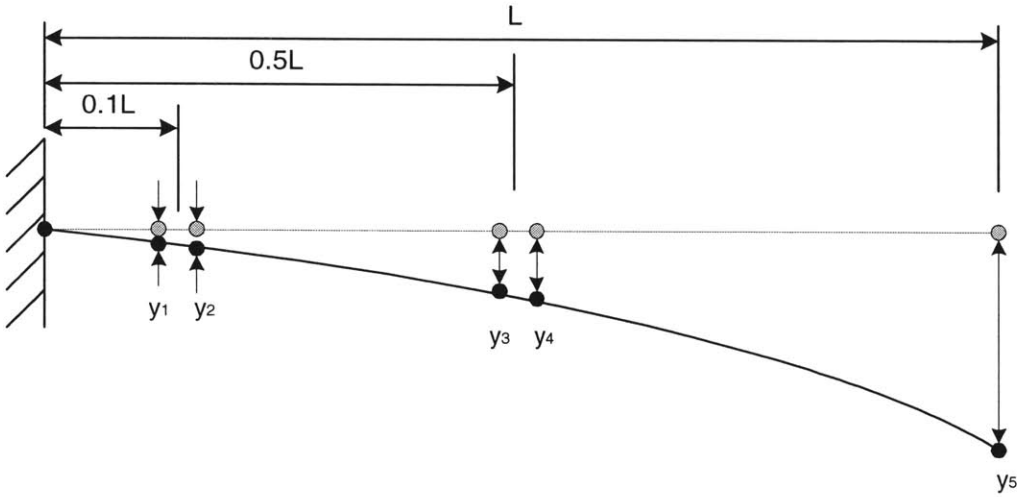


Figure 5.8 Double Cracks Cantilever Beam

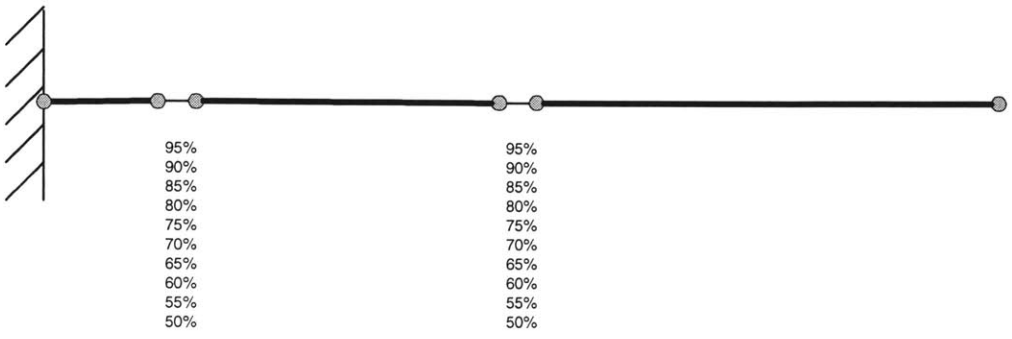


Figure 5.9 Double Cracks Cantilever Beam Reduction of Moment of Inertia

5.5 Neural Network Design for Crack Self-Diagnosis Cantilever Beam

5.5.1 Neural Network Architecture of Single Crack Cantilever Beam

Fifty different reductions of *moment of inertia* are considered. Reductions range from one percent to fifty percent. The damaged cantilever beam has been evaluated by using SAP-2000 nonlinear analysis program. The following parameters have been assumed: elasticity modulus $E = 2.9E^{6} ib/in^{2}$, cantilever beam length $L = 21\ feet$, cross section height $H = 18\ inch$, cross section width $B = 10$ inch, steel yield stress $f_{y} = 36ksi$, and specific density $\rho = 2.830E^{-0.4} ib/in^{3}$.

Damage section displacements and tip displacement associated with the reduction percentage are used to train the feedforward back-propagation neural network with five neurons in the single hidden layer. That network has been designed as a function approximator. This neural network has three inputs in input-layer. Two of them are the displacements of the damaged section. The last input in the input-layer is the tip displacement of the cantilever beam. The output layer with one linear neuron produces the estimations of the reduction in the damaged section. The inputs to this neural network have been scaled between 0.1 and 0.9 to avoid saturation of log-sigmoid activation functions. The adaptive conjugate gradient algorithm was used in this neural network. A schematic of this neural network is shown in Figure 5.7.

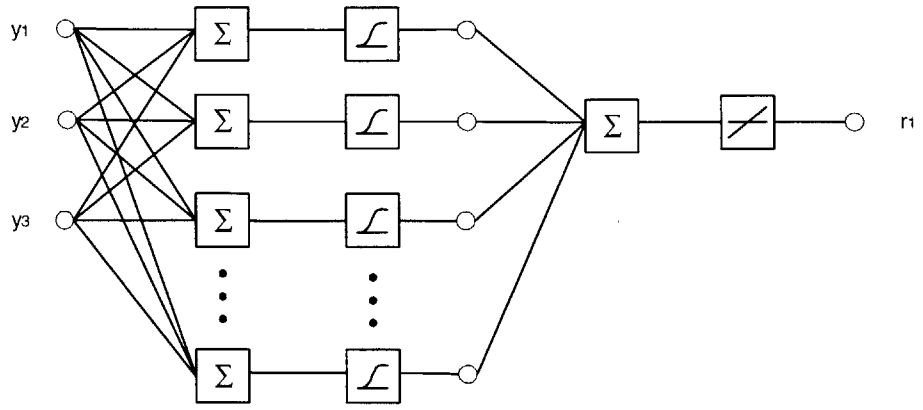


Figure 5.10 Single Crack Neural Network Architecture.

5.5.2 Neural Network Architecture of Double Cracks Cantilever Beam

There are 100 different combinations of moment of inertia reductions of each damaged section. Reductions of moment of inertia range from 95 percent reduction to 50 percent reduction. The damaged cantilever beam has been evaluated by using SAP-2000 nonlinear analysis program. The following parameters have been assumed: elasticity modulus $E = 2.9E^{6} ib/in^{2}$, cantilever beam length $L = 21\ feet$, cross section height $H = 18\ inch$, cross section width $B = 10\ inch$, steel yield stress $f_y = 36ksi$, and specific density $\rho = 2.830E^{-0.4} ib/in^{3}$.

Four displacements of damaged elements and one tip displacement associated with the crack reduction percentage are used to train the feedforward back-propagation neural network with five neurons in the single hidden layer. This neural network has been designed as a function approximator. This neural network has five inputs. Four of them are displacements of damaged element. The last one is the tip displacement. There are two outputs in the output layer. These are the estimations of the reduction in the damaged elements. The inputs to the network have been scaled between 0.1 and 0.9 to avoid saturation of log-sigmoid activation functions. The adaptive conjugate gradient algorithm

was used in this neural network. The schematic of this neural network is shown in Figure 5.8.

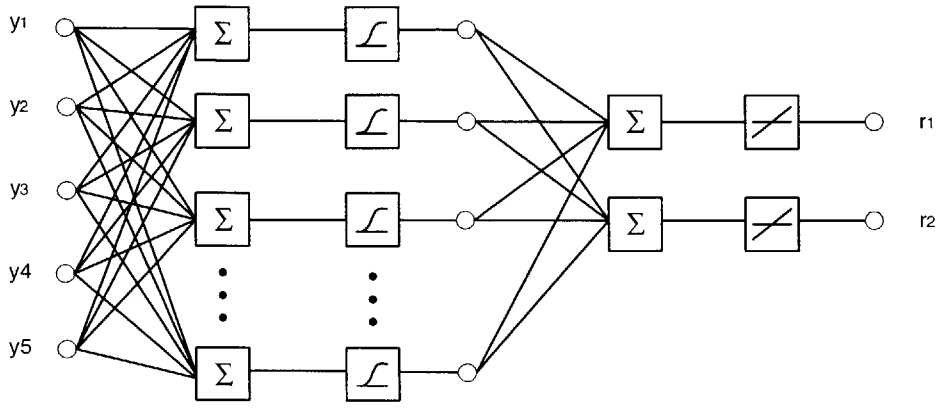


Figure 5.11 Double Cracks Neural Network Architecture.

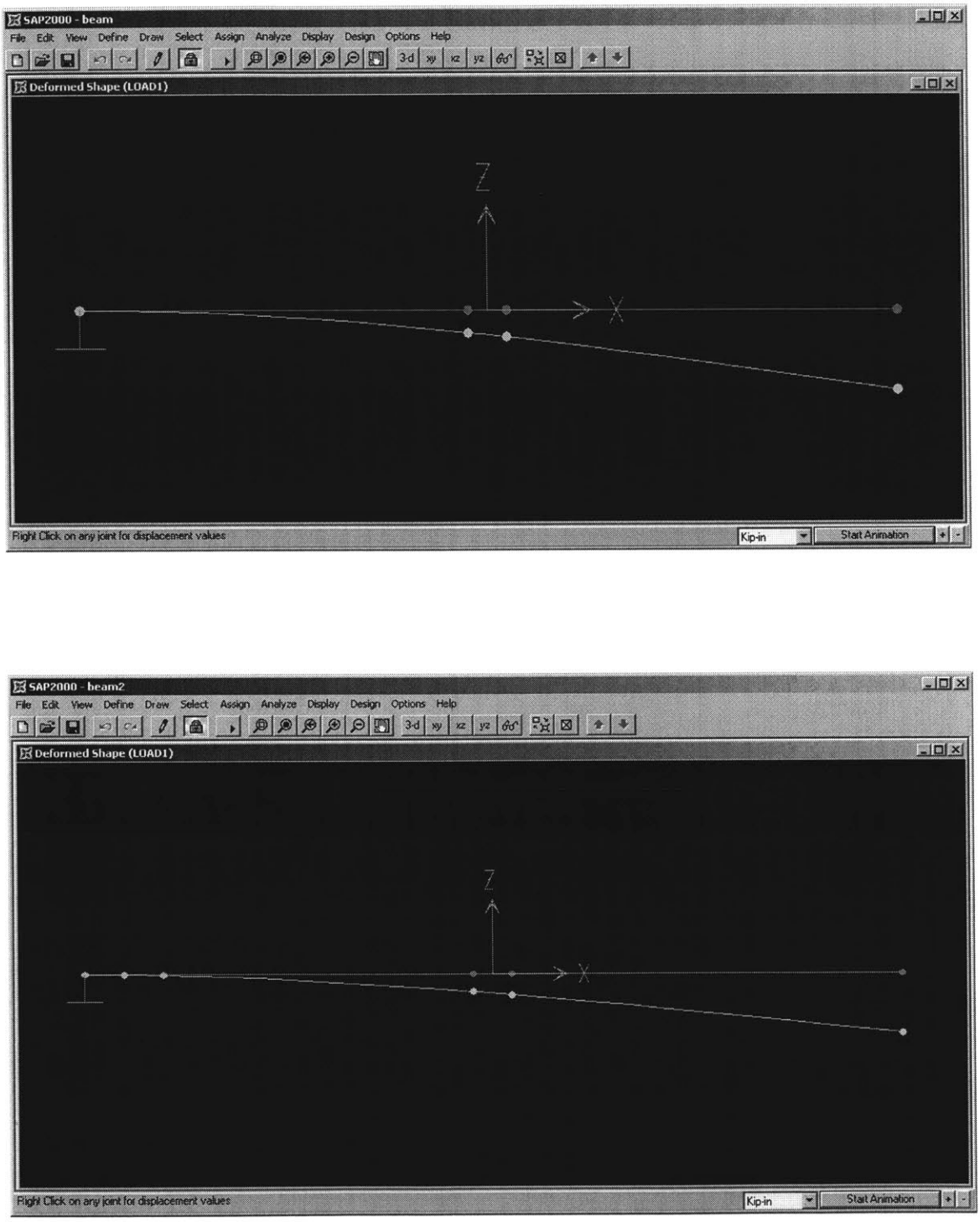


Figure 5.12 Screen Shot of SAP-2000 Nonlinear Analysis Program.

5.6 Discussions and Summary

In the single damage case, the beam tip displacement and the displacements in damaged element corresponding to the 50 different crack reduction have been created and used to train the neural network. The role of the neural network is to find the coordinates of the intersection of crack characteristics corresponding to a given cantilever beam displacement. Figure 5.13 shows the sensitivity of network solution to measurement errors in input data. The training of this feedforward back-propagation neural network with 3 neurons in the single hidden layer was discontinued after 20 iterations when the sum of squared errors reached 0.0098. The conjugate gradient algorithm was used to train the neural network. Results presented in Figure 5.13 were obtained for network input consisting of exact displacement data.

Next, in the double damage case, the beam tip displacement and the displacements in the damaged elements corresponding to the 100 different crack reduction combinations have been created and used to train the neural network. The training of this feedforward back-propagation neural network with 5 neurons in the single hidden layer was discontinued after 498 iterations when the sum of squared errors reached 0.0099. The conjugate gradient algorithm was used to train the neural network. The performance of this network is shown in Figure 5.14.

This work shows the feasibility of applying a neural network to predict the damage location and the reduction of moment of inertia in a damaged element. It has been proved that that the neural network performs adequately for data contaminated by measurement errors.

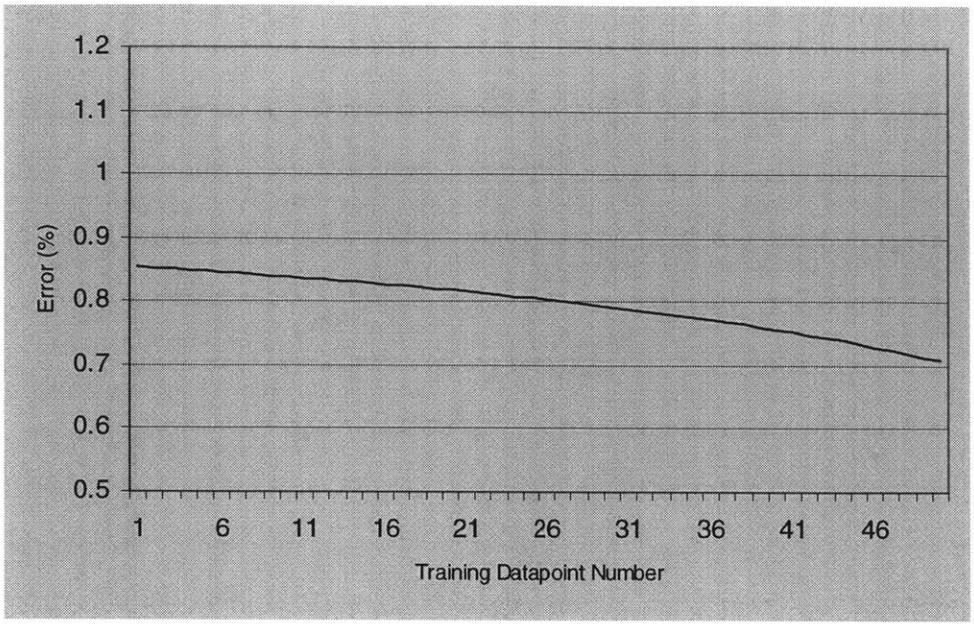


Figure 5.13 Percentage Errors for Neural Network Solution.

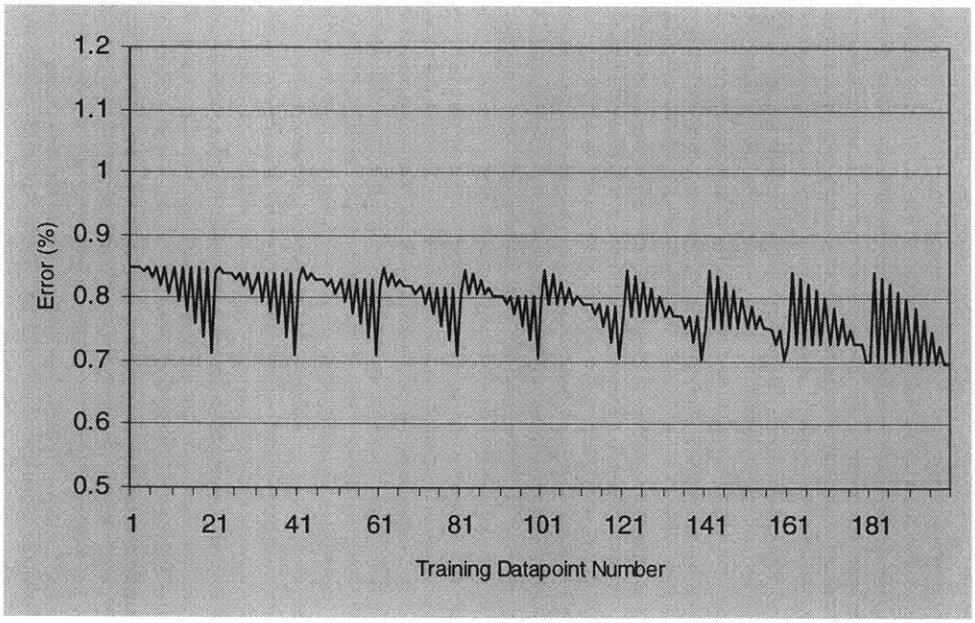


Figure 5.14 Percentage Errors for Neural Network Solution.

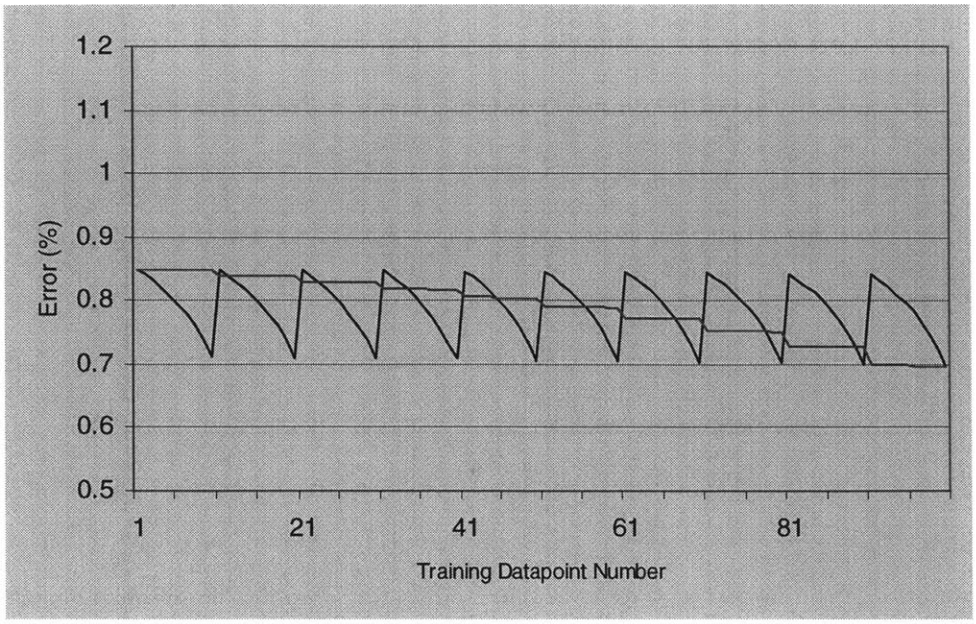


Figure 5.15 Percentage Errors for Neural Network Solution.

REFERENCES

1. Nabil Kartam, Ian Flood, James H. Garrett, *"Artificial Neural Networks for Civil Engineers: Fundamentals and Applications"*, ASCE, New York, 1997.

2. Tomas Hrycej, *"Neurocontrol: Towards an Industrial Control Methodology"*, John Wiley & Sons, Inc., New York, 1997.

3. Rumelhart, McClelland, *"Parallel Distributed Processing Volume 1: Foundations"*, The MIT Press, Cambridge, 1986.

4. Simon Haykin, *"Neural Networks: A Comprehensive Foundation"*, Prentice Hall, New Jersey, 1999.

5. B. H. V. Topping, A.I. Khan, *"Neural Networks and Combinatorial Optimization in Civil and Structural Engineering"*, Civil-Comp Press, Edinburgh, 1993.

6. B. H. V. Topping, *"Developments in Neural Networks and Evolutionary Computing in Civil and Structural Engineering"*, Civil-Comp Press, Edinburgh, 1995.

7. Reed, Marks, *"Neural Smithing"*, The MIT Press, Cambridge, 1999.

8. Hojjat Adeli, Shin-Lin Hung, *"Machine Learning: Neural Networks, Genetic Algorithms, and Fuzzy Systems"*, John Wiley & Sons, Inc., New York, 1995.

9. Stephen Gilbert, Bill McCarty, *"Object-Oriented Design In Java"*, White Group Press, Corte Madera, CA, 1998.

10. Mark Allen Weiss, *"Data Structures and Problem Solving Using Java"*, Addison-Wesley, Reading, MA, 1998.

11. Howard Demuth, Mark Beale, *"Neural Network Toolbox: For Use with MATLAB"*, The Math Works Inc., Natick, MA, 1998 .

12. Kawiecki G, "Application of neural networks to detect detection in cantilever beams with linearized damage behavior", *Journal of Intelligent Material Systems and Structures*, 10(10), 797-801,1999.

13. C.B. Yun, E.Y. Bahng, "Substructural identification using neural networks", *Computers and Structures*, 77, 41-52, 2000.

14. M.I. Friswell, J.E.T. Penny, S.D. Garvey, "A combined genetic and eigensensitivity algorithm for the location of damage in structures", *Computers and Structures,* 69, 547-556, 1998.

15. C.C. Chang, T.Y.P. Chang, Y.G.Xu, "Adaptive neural networks for model updating of structures", *Smart Mater. Struct.*, 9, 59-68, 2000.

16. Mitsuru Nakamura, Sami F. Masri, A.G. Chassiakos, T.K. Caughey, "Amethod for non-parametric damage detection through the use of neural networks", *Earthquake engineering and structural dynamics*, 27, 997-1010, 1998.

17. Tshilidzi Marwala, "Damage identification using committee of neural networks", *Journal of Engineering Mechanics*, 216(1), 43-50, 2000.

18. S.V. Barai, P.C. Pandey, "Performance of the generalized delta rule in structural damage detection", *Engng Applic. Artif. Intell.*, 8(2), 211-221, 1995.

19. r. Ceravolo, A. De Stefano, D. Sabia, "Hierarchical use of neural techniques in structural damage recognition", *Smart Mater. Struct.*, 4, 270-280, 1995.

20. S. Yoshimura, A. Matsuda. G. Yagawa, "New regularization by transformation for neural network based inverse analyses and its application to structure identification", *International Journal for Numerical Methods in Engineering*, 39, 3953-3968, 1996.

21. K. Worden, A.D. Ball, G.R. Tomlinson, "Fault location in a framework structure using neural networks", *Smart Mater. Struct.*, 2, 189-200, 1993.

22. J.N. Kudva, N Munir, P.W. Tan, "Damage detection in smart structures using neural networks and finite-element analyses", *Smart Mater. Struct.*, 1, 108-112, 1992.

23. Z.P. Szewczyk, Prabhat Hajela, "Damage detection in structures based on feature-sensitive neural networks", *Journal of Computing in Civil Engineering*, 8(2), 163-178, 1994.

24. R.D. Vanluchene, Roufei Sun, "Neural networks in structural Engineering", *Microcomputers in Civil Engineering*, 5, 207-215, 1990.

25. Hojjat Adeli, H.S. Park, "Counterpropagation neural networks in structural engineering", *Journal Struct. Engng.*, 121(8), 1205-1212, 1995.

26. H. Adeli, C. Yeh, "Perceptron learning in engineering design", *Microcomputers in Civil Engineering*, 4, 247-256, 1989.

27. C. Peetathawatchai, "The applicability of neural network systems for structural damage diagnosis", Ph.D. *Thesis*, Department of Civil and Environmental Engineering, Massachusetts Institute of Technology.

28. Danny C.C. Poo, Derek B.K. Kiong, *"Object-oriented programming"*, Springer, New York, 1998.

29. Michael Main, *"Data structures & other objects using Java"*, Addison-Wesley, Reading, MA, 1999.

30. David Gries, F.B. Schneider, "Data structures and Algorithms", Springer, New York, 1997.

31. R.R. Gajewski, "An object oriented approach to finite element programming", *Artificial Intelligence and Object Oriented Approaches for Structural Engineering*, Civil-Comp Press, Edinburgh, UK, 1994.

Appendices

Appendix A: Neural Network Analysis Java Program Codes

```
// Ann.java
import java.io.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;

public class Ann
{
      static boolean isDisplay_res;
      static boolean isInput2;
      static boolean isNetwork;
      static boolean isDisplay;
      static boolean isDisplay2;
      static double displacement;
      static double display_reduction;  // middle one
      static double display_reduction2; // left one

      // Labels
      static JLabel input_file_Label;
      static JLabel weight_file_Label;
      static JLabel algo_Label;
      static JLabel NIA_Label;
      static JLabel NOA_Label;
      static JLabel NSAMP_Label;
      static JLabel NHL_Label;
      static JLabel NUH1_Label;
      static JLabel NUH2_Label;
      static JLabel NUH3_Label;
      static JLabel NUH4_Label;
      static JLabel NUH5_Label;
      static JLabel LR_Init_Label;
      static JLabel MR_Label;
      static JLabel MSE_Label;
      static JLabel MaxIter_Label;

      // Text/fields
      static JTextField input_file_Field;
      static JTextField weight_file_Field;
      static JTextField algo_Field;
      static JTextField NIA_Field;
      static JTextField NOA_Field;
      static JTextField NSAMP_Field;
      static JTextField NHL_Field;
      static JTextField NUH1_Field;
      static JTextField NUH2_Field;
      static JTextField NUH3_Field;
      static JTextField NUH4_Field;
      static JTextField NUH5_Field;
      static JTextField LR_Init_Field;
      static JTextField MR_Field;
      static JTextField MSE_Field;
      static JTextField MaxIter_Field;
```

```
      static final int RAND_RANGE =  32;
      static final int tasking    =  1;
      static final int PRERR      =  1;
      static final int MAXHL      =  5;
      static final int MAXOA      =  50;
      static final int MAXIA      =  50;
      static final int MAXSAMP    =  1000;
      static final int MAXUNT     =  50;
      static final int MAXW       =  (MAXHL+1) * MAXOA * MAXIA;
      static final double Delta   =  0.01;
      static final double bt      =  0.2;
      static final double G1      =  1.618;
      static final double G2      =  0.382;
      static final double G3      =  0.618;
      static final double e       =  0.0001;
      static final double gama    =  0.2;
      static final double Eplson  =  0.000001;
      static final double minNSE  =  0.0000001;
      static final double alph    =  0.0001;
      static final double beta    =  0.9;
      static final double M       =  5;
      static final double fmax    =  5678;

     /* Variables for learning domain */
      static double[][] Sample  = new double[MAXSAMP][MAXIA];  // Learning
samples
      static double[][] Output  = new double[MAXSAMP][MAXOA];  // Output (oij)
      static double[][] Desired = new double[MAXSAMP][MAXOA];  // Desired
output (tij)

     /* Variables for cell of neural */
      static int  check;                          // jan
      static int[]  WL  = new int[MAXHL+2];
      static double[] Weight  = new double[MAXW]; // Weights associated with
links
      static double[] Weightp = new double[MAXW];
      static double ak,bk;                        // ak and bk in conjugate
gradient method
      static double[] d = new double[MAXW];       // dk in conjugate gradient
method
      static double[] g1 = new double[MAXW];
      static double[] g0 = new double[MAXW];      // gk in conjugate gradient
method
      static double[] H = new double[MAXW];

     /* Variables for BackPropagation */
      static int IsNew;                           // Is a new iteration
      static int IsShort;
      static int IsOld;                           // Is an old iteration
      static double NSE;                          // Normalized system error
      static double MPE;                          // Max allowed pattern error
      static double LR_Final,LR_Change_Rate;

      static double[][] Oij = new double[MAXHL+2][MAXOA]; //  Input layer
iutput calculation
      static double[][] Del = new double[MAXHL+2][MAXOA]; // Error pointer
      static int L1,L2,NoIter,again, Nokey;
      static int task;
      static int save_err,plot,save_w,save_arc,save_res;
      static int from, to;
      static double ao, MiNSE=10;
```

```
      static String fpt, f2, inf, farc, fw, ferr, fres, buffer;
      static int step;     // 1     2        3        4

      /* Input file parameters */
      static char select = 'l';          // l,L for learning, a, A for analysis
      static int algo;                   // 0 - BP, 1 - CG
      static int NIA;                    // Number of input attributes
      static int NOA;                    // Number of output attributes
      static int NSAMP;                  // Number of learning samples
      static int NHL;                    // Number of the hiddern layers
      static int[] NUH = new int[MAXHL+2];
      static int ans = 1;                // 1:Initialize the weights ; else input
from file
      static double LR_Init;
      static double MR;
      static double MSE;                 // Max allowed system error
      static double MaxIter;             // Max allowed interations


static void init_domain()
{
  System.out.println("Training instance File for learning/analysis : " + fpt );
  System.out.println("Number of training instances -- " + NSAMP);
  System.out.println("Number of input/output nodes -- " + NIA + " " + NOA);

  try{
     readSD(fpt);
     System.out.println("readSD fpt = " + fpt );
  }
  catch (IOException e) {
  }
}

static void readSD(String fileName) throws IOException
{
  Reader fileReader = new FileReader(fileName);
  BufferedReader bufferedReader = new BufferedReader(fileReader);
  String nextLine;

  for (int i=0; i<NSAMP; i++){
        nextLine = bufferedReader.readLine();
        StringTokenizer tokenizer = new StringTokenizer(nextLine);

        for (int j=0; j<NIA; j++){
            Sample[i][j] = Double.parseDouble(tokenizer.nextToken());
        }
        for (int j=0; j<NOA; j++){
            Desired[i][j] = Double.parseDouble(tokenizer.nextToken());
        }
  }
  bufferedReader.close();
  fileReader.close();
}

static void init_neural()
{
  int i,j;
  if(task==0)   /* learning */
  {
      System.out.println("Number of hidden layers ( Max is 5 ) -- " + NHL );

      for(i=0; i<NHL; i++)   /* NV */
      {
```

```
          System.out.println("Number of nodes in hidden layer -- " + (i+1) + "
" + NUH[i+1]  );
      }
  }
  if(task==1)  /* analysis */
  {
      System.out.println("The *.top file for analysis -- " + fpt );
      System.out.println("Number of hidden layers ( Max is 5 ) -- " + NHL );

      for(i=0; i<NHL; i++)    /* NV */
      {
         System.out.println("Number of nodes in hidden layer -- " + (i+1) + " "
+ NUH[i+1]  );
      }
  }

  NUH[0]=NIA;
  NUH[NHL+1]=NOA;
  NUH[NHL+2]=0;

  L1=0;
  for(i=0; i<NHL+2; i++)    /* V */
    {
      WL[i]=(NUH[i]+1)*NUH[i+1];
      L1+= WL[i];
    }
  for(j=0; j<NHL+1; j++){    /* V */
    Oij[j][NUH[j]]=1.0;
  }
}

static void init_weight()
{
  int i;
  double temp;

  if(task==0)   // learning
    {
      if(ans==1)   /* Initialize the weights */
         {
        for(i=0; i<L1; i++)    /* NV */
              Weight[i]= Math.random() * Math.pow(2,14);    /* 0 -- 2**15-1 */

        for(i=0; i<L1; i++)
             {
            Weight[i]=2*(Weight[i]/(Math.pow(2.0,(double)(RAND_RANGE-1))))-1.0;
               Weightp[i]=Weight[i];
             }
         }
      else    /* Input the weights from file */
      {
      }
    }
  else if(task==1)   // analysis
  {
      System.out.println("Load the weights from file :" + fw );
      try{
        readW(fw);
      }
      catch (IOException e){
      }
  }
}
```

```
static void readW(String fileName) throws IOException
{
      Reader fileReader = new FileReader(fileName);
      BufferedReader bufferedReader = new BufferedReader(fileReader);
      String nextLine;
      nextLine = bufferedReader.readLine();

      for(int i=0; i<L1; i++) {
           StringTokenizer tokenizer = new StringTokenizer(nextLine);
           Weightp[i] = Weight[i]  = Double.parseDouble(tokenizer.nextToken());
      }
      bufferedReader.close();
      //fileReader.close();
}

static void init_learning_CG() /* 203 */
{
  if(task==0)      /* learning */
    {
      MSE=0.01;
      MPE=0.001;
      System.out.println("Maximum total system error -- "    + MSE );
      System.out.println("Max pattern error -- default -- " + 0.001 );
      System.out.println("Maximum number of iteration -- "  + MaxIter);
    }
}

static void init_learning_BP() /* 203 */
{
  if(task==0)      /* learning */
  {
      System.out.println("Learning rates -- "    + LR_Init );
      System.out.println("Momentum ratio -- "    + MR );
      System.out.println("Maximum total system error -- default -- " + MSE );
      System.out.println("Maximum number of iteration -- "  + MaxIter);
  }
}

static void forward(int i)
{
  int p,q,r,more,morew;
  double[] net = new double[MAXOA*MAXIA];

  /* the input layer */
  for(p=0; p<NIA; p++)    /* V */
    Oij[0][p]=Sample[i][p];

  /* the hidden layers  */
  morew=0;
  for(p=1; p<NHL+2; p++)
  {
      for(q=0; q<NUH[p]; q++)    /* V */
       net[q]=0.0;
      for(r=0; r<NUH[p-1]+1; r++)
      {
         for(q=0; q<NUH[p]; q++)    /* V */
           {
             more= q*(NUH[p-1]+1)+r+morew;
             net[q] += Weight[more]*Oij[p-1][r];
           }
      }
      morew += WL[p-1];
```

```
      for(q=0; q<NUH[p]; q++)    /* V */
       Oij[p][q]=1/(1+ Math.exp((double)-net[q]));
  }
}

static double System_error(double[] W, double[] delw, double a)
{
  int i,p,q,r,more,morew;
  double[] net = new double[MAXOA*MAXIA];
  double xnse,xNSE;
  double[] weight = new double[MAXW];

  NSE=0;
  for(i=0; i<L1; i++)   /* V */
    weight[i] = W[i] + a*( delw[i] );

  xNSE=0;
  for(i=0; i<NSAMP; i++)    /* Tasked */
    {
      /* the input layer */
      for(p=0; p<NIA; p++)    /* V */
             Oij[0][p]=Sample[i][p];

      /* the hidden layers  */
         morew=0;
      for(p=1; p<NHL+2; p++)
       {
                for(q=0; q<NUH[p]; q++)    /* V */
                     net[q]=0.0;
                for(r=0; r<NUH[p-1]+1; r++)
                     {
                       for(q=0; q<NUH[p]; q++)    /* V */
                            {
                              more= q*(NUH[p-1]+1)+r+morew;
                              net[q] += weight[more]*Oij[p-1][r];
                            }
                     }
                morew += WL[p-1];
                for(q=0; q<NUH[p]; q++)    /* V */
                     {
                       if(net[q] > fmax)
                            Oij[p][q]= 5000.;
                       else if(net[q] < -fmax)
                            Oij[p][q]= -5000.;
                       else
                            Oij[p][q]=1/(1+ Math.exp(-net[q]));
                     }
       }
      for(p=0; p<NUH[NHL+1]; p++)  /* V */
             {
               xnse=Desired[i][p]-Oij[NHL+1][p];
               xNSE += (xnse*xnse);
             }
       }
  NSE += xNSE;
  NSE /= (2*NSAMP);
  return(NSE);
}

static double CalGrad(double[] W, double[] d, double a)
{
  int i,j,p,q,r,more,morew;
  double[] net = new double[MAXOA*MAXIA];
```

```
double Oj,newslope;
double[] xDelW = new double[MAXW];

for(j=0; j<L1; j++)  /* V */
  {
    Weight[j] = W[j] + a*( d[j] );
    g1[j]=0.0;
  }

  for(j=0; j<L1; j++)   /* V */
    xDelW[j]=0.;

for(i=0; i<NSAMP; i++)     /** tasked  **/
  {
    /** Forward process **/
    for(p=0; p<NIA; p++)          /* V */
      Oij[0][p]=Sample[i][p];
    morew=0;
    for(p=1; p<NHL+2; p++)
      {
        for(q=0; q<NUH[p]; q++)    /* V */
          net[q]=0.0;
        for(r=0; r<NUH[p-1]+1; r++)
          {
            for(q=0; q<NUH[p]; q++)    /* V */
              {
                more= q*(NUH[p-1]+1)+r+morew;
                net[q] += Weight[more]*Oij[p-1][r];
              }
          }
        morew += WL[p-1];
        for(q=0; q<NUH[p]; q++)    /* V */
         {
           if(net[q] > fmax)
            Oij[p][q]= 5000.;
           else if(net[q] < -fmax)
            Oij[p][q]= -5000.;
           else
            Oij[p][q]=1/(1+ Math.exp(-net[q]));
         }
      }
    /** Calculate system error **/

    for(p=0; p<NUH[NHL+1]; p++)  /* V */
      {
        Oj= Oij[NHL+1][p];
        Del[NHL+1][p]= (Desired[i][p]-Oj)*(1-Oj)*Oj;
      }

    for(p=NHL+1; p>=1; p--)
      {
        morew=0;
        for(q=0; q<p-1; q++)
          morew += WL[q];
        for(q=0; q<(NUH[p-1]+1); q++)
          {
            Del[p-1][q]=0.0;

            for(r=0; r<NUH[p]; r++)  /* NV */
              {
                more=(NUH[p-1]+1)*r+q+morew;
                xDelW[more] += (Del[p][r]*Oij[p-1][q]);
                Del[p-1][q] += (Del[p][r]*Weight[more]);
```

```
                }
              Del[p-1][q] *= ((1-Oij[p-1][q])*Oij[p-1][q]);
            }
        }
    }

  /** Update weight & total system error **/
  for(j=0; j<L1; j++)     /* V */
    g1[j] += xDelW[j];

  newslope=0;
  for(j=0; j<L1; j++)
    newslope -= g1[j]*d[j];
  return (newslope);
}

static void Modified_search_1(double[] W, double[] delw)
{
  int i,k,cnt,lmt,goon,ok;
  double fa0,fa1,fa2,dt;
  double a0,a1,a2;
  double p,q,r,t,ee,eps,fx,fv,fu,fw,u,v,w,x,c,a,b,d=0,tol,m,t2;

  cnt=0;
  lmt=0;
  goon=1;
  dt=Delta;
  k=1;
  a0=0;
  a1=dt;
  a2=a1+G1*dt;
  fa0=System_error(W,delw,a0);
  fa1=System_error(W,delw,a1);

  if(fa1 > fa0)
    {
      while((fa1 >fa0) && (lmt <=5))
       {
         dt *= 0.6;
         a1=dt;
         fa1=System_error(W,delw,a1);
         lmt++;
          cnt++;
       }

      a2=2*a1;
      fa2=System_error(W,delw,a2);

      if((fa0<fa1)&&(fa0<fa2))   {
         ao=-0.0011;
         NSE=System_error(W,delw,ao);
         goon=0;
         }
      if((fa0<fa1)&&(fa2<fa0))  {
         ao=a2;
         NSE=System_error(W,delw,ao);
         goon=0;
         }
    }
  if(fa0>fa1)
    {
      fa2=System_error(W,delw,a2);
      dt*=100;
```

```
      while((!((fa0>=fa1) && (fa2>fa1)) && k<=20))
       {
         k++;
         a0=a1; a1=a2; fa0=fa1; fa1=fa2;
         a2+= Math.pow(G1,k)*dt;
         fa2=System_error(W,delw,a2);
         cnt++;
       }
      if(k==21)  {
         ao=a2; /*   here!! error!!   */
         NSE=System_error(W,delw,ao);
         goon=0;
         }
    }
  if(goon==1)
    {
      t=.000001;
      eps=e;
      a=a0;  b=a2;
      c=0.381966;
      v=w=x=a+c*(b-a);
      ee=0;
      fv=fw=fx=System_error(W,delw,x);
      m=0.5*(a+b);
      tol=eps*x+t;
      t2=2*tol;
      ok=1;
      while(( Math.abs(x-m) > (t2-0.5*(b-a))) && ok==1)
       {
         p=q=r=0;
         if( Math.abs(ee) > tol)
           {
             r=(x-w)*(fx-fv);
             q=(x-v)*(fx-fw);
             p=(x-v)*q-(x-w)*r;
             q=2*(q-r);
              if (q > 0)
                  p = -p;
              else
                  q = -q;
             r=ee; ee=d;
           }
         if(( Math.abs(p) < Math.abs(0.5*q*r)) && (p < q*(a-x)) && (p < q*(b-
x)))
           {
             d=p/q;
             u=x+d;
             if(((u-a) < t2) || ((b-u) < t2))
              d=((x < m) ? tol : (-tol));
           }
         else
           {
             ee=((x < m) ? (b-x) : (a-x));
             d=c*ee;
           }
         if(Math.abs(d) >= tol)
           u=x+d;
         else {
            if (d > 0)
                u = x+tol;
            else
                u = x-tol;
```

```
          }

         fu=System_error(W,delw,u);
         if(fu <= fx)
         {
              if (u < x)
                 b = x;
              else
                 a = x;

             v=w;  fv=fw;
             w=x;  fw=fx;
             x=u;  fx=fu;
         }
         else
          {
              if ( u < x )
                 a = u;
              else
                 b=u;

             if((fu <= fw) || (w ==x))
              {
                v=w;  fv=fw;
                w=u;  fw=fu;
              }
             else if((fu <= fv) || (v==x) || (v==w))
              {
                v=u;  fv=fu;
              }
         }

         m=0.5*(a+b);
         tol=eps*Math.abs(x)+t;
         t2=2*tol;
          cnt++;
       }
          if(x<=0.001)  {
             ao=0.001;
             NSE=System_error(W,delw,ao);
             }
          else  {
             ao=x;
             NSE=System_error(W,delw,ao);
             }
      }
if(NSE<MiNSE)  {
   x=(MiNSE-NSE)*1000;
   x=x*NSAMP*2;
   if(x>0.001)  {
      for(i=0;i<L1;i++)
         Weightp[i]=Weight[i];
      MiNSE=NSE;
      Nokey=0;
      }
   }
}

static void parallel_DoBP(int from ,int to)  /*310*/
{
  int i,j,p,q,r,more,morew;
  int noksam;    /* jan */
  double ei;      /* jan */
```

```
double[] net = new double[MAXOA*MAXIA];
double Oj;
double[] xDelW = new double[MAXW];
double xNSE,xnse;

for(j=0; j<L1; j++)  /* V */
  g1[j]=0.0;
NSE=0.;
for(j=0; j<L1; j++)   /* V */
  xDelW[j]=0.;
xNSE=0.;
check=0;    /* jan */
noksam=0;   /* jan */
for(i=from; i<to; i++)     /** tasked  **/
  {
      /** Forward process **/
    for(p=0; p<NIA; p++)         /* V */
          Oij[0][p]=Sample[i][p];
    morew=0;
    for(p=1; p<NHL+2; p++)
          {
            for(q=0; q<NUH[p]; q++)    /* V */
                net[q]=0.0;
            for(r=0; r<NUH[p-1]+1; r++)
                {
                  for(q=0; q<NUH[p]; q++)    /* V */
                      {
                        more= q*(NUH[p-1]+1)+r+morew;
                        net[q] += Weight[more]*Oij[p-1][r];
                      }
                }
            morew += WL[p-1];
            for(q=0; q<NUH[p]; q++)    /* V */
                {
                  if(net[q] > fmax)
                      Oij[p][q]= 5000.;
                  else if(net[q] < -fmax)
                      Oij[p][q]= -5000.;
                  else
                      Oij[p][q]=1/(1+ Math.exp((double)-net[q]));
                }
          }
    /** Calculate system error **/
    ei=0;    /* jan */
    for(p=0; p<NUH[NHL+1]; p++)  /* V */
          {
            xnse=Math.abs(Desired[i][p]-Oij[NHL+1][p]);
            xNSE += (xnse*xnse);
        ei=ei+(xnse*xnse);    /* jan */
            Oj= Oij[NHL+1][p];
            Del[NHL+1][p]= (Desired[i][p]-Oj)*(1-Oj)*Oj;
          }
    if(ei>MPE*MPE)              /* jan */
       noksam=noksam+1;   /* jan */
    for(p=NHL+1; p>=1; p--)
          {
            morew=0;
            for(q=0; q<p-1; q++)
                morew += WL[q];
            for(q=0; q<(NUH[p-1]+1); q++)
                {
                  Del[p-1][q]=0.0;
                  for(r=0; r<NUH[p]; r++)  /* NV */
```

```
                        {
                          more=(NUH[p-1]+1)*r+q+morew;
                          xDelW[more] += (Del[p][r]*Oij[p-1][q]);
                          Del[p-1][q] += (Del[p][r]*Weight[more]);
                        }
                    Del[p-1][q] *= ((1-Oij[p-1][q])*Oij[p-1][q]);
                  }
              }
        }
  if(noksam>0)    /* jan */
      check=1;    /* jan */
  /** Update weight & total system error **/
  for(j=0; j<L1; j++)    /* V */
    gl[j] += xDelW[j];
  NSE += xNSE;
}

static void BP_DoBP(int from ,int to)      /*310*/
{
  int i,j,p,q,r,more,morew;
  int noksam;    /* jan */
  double ei;      /* jan */
  double[] net = new double[MAXOA*MAXIA];
  double Oj;
  double[] xDelW = new double[MAXW];
  double[] DelW = new double[MAXW];
  double xnse,LR;

  LR=LR_Init;

  for(j=0; j<L1; j++)  /* V */
    DelW[j]=0.0;

  do
  {
        for(j=0; j<L1; j++)   /* V */
          xDelW[j]=0.;
        NSE=0.;
        check=0;    /* jan */
        noksam=0;   /* jan */
        for(i=from; i<to; i++)     /** tasked  **/
        {
        /** Forward process **/
             for(p=0; p<NIA; p++)          /* V */
               Oij[0][p]=Sample[i][p];
             morew=0;
             for(p=1; p<NHL+2; p++)
             {
               for(q=0; q<NUH[p]; q++)    /* V */
                    net[q]=0.0;
               for(r=0; r<NUH[p-1]+1; r++)
               {
                    for(q=0; q<NUH[p]; q++)    /* V */
                    {
                      more= q*(NUH[p-1]+1)+r+morew;
                      net[q] += Weight[more]*Oij[p-1][r];
                    }
               }
               morew += WL[p-1];
               for(q=0; q<NUH[p]; q++)    /* V */
               {
                    if(net[q] > fmax)
                      Oij[p][q]= 5000.;
```

```
                else if(net[q] < -fmax)
                  Oij[p][q]= -5000.;
                else
                  Oij[p][q]=1/(1+ Math.exp((double)-net[q]));
              }
            }
    /** Calculate system error **/
            ei=0;    /* jan */
            for(p=0; p<NUH[NHL+1]; p++)  /* V */
            {
              xnse=Math.abs(Desired[i][p]-Oij[NHL+1][p]);
              NSE += (xnse*xnse);
                 ei=ei+(xnse*xnse);    /* jan */
              Oj= Oij[NHL+1][p];
              Del[NHL+1][p]= (Desired[i][p]-Oj)*(1-Oj)*Oj;
            }

      if(ei>MPE*MPE)             /* jan */
        noksam=noksam+1;  /* jan */

      for(p=NHL+1; p>=1; p--)
            {
              morew=0;
              for(q=0; q<p-1; q++)
                   morew += WL[q];
              for(q=0; q<(NUH[p-1]+1); q++)
              {
                Del[p-1][q]=0.0;
                   for(r=0; r<NUH[p]; r++)  /* NV */
                   {
                     more=(NUH[p-1]+1)*r+q+morew;
                     xDelW[more] += (Del[p][r]*Oij[p-1][q]);
                     Del[p-1][q] += (Del[p][r]*Weight[more]);
                   }
                   Del[p-1][q] *= ((1-Oij[p-1][q])*Oij[p-1][q]);
              }
            }
     }
  if(noksam>0)    /* jan */
    check=1;    /* jan */

/** Update weight & total system error **/
  for(j=0; j<L1; j++)
  {
     Weight[j] += (LR*xDelW[j]+MR*DelW[j]);
     DelW[j]=xDelW[j];
  }

     NoIter++;
      NSE /= (2*NSAMP);
     if((plot==1) && ((NoIter % 10)==0)){
      }

     if(save_err==1){
         try {
             writeErr("beam.err");
         }
         catch(IOException e){
         }
      }

     if((NoIter >= MaxIter) && (NSE > MSE))
     {
```

```
            MaxIter = 2 * MaxIter;
            System.out.println(" MaxIter *= 2; ");
        }
    }while((NoIter <= MaxIter) && (NSE >= MSE));
}

static void writeErr(String fileName) throws IOException {
     Writer writer = new FileWriter(fileName);
     PrintWriter printWriter = new PrintWriter(writer);
     printWriter.print( NoIter + " " + NSE + " ");
     printWriter.flush();
     printWriter.close();
}

static void CG_DoBP(int from ,int to)    /*310*/
{
  int j,iter,Stop;
  double temp,temp1,temp2;

  NoIter=0;
  for(j=0; j<L1; j++)
    g0[j]=0.0;

  parallel_DoBP(from,to);
  if((plot==1) && ((NoIter % 10)==0)){
     System.out.println(" NoIter= " + NoIter + " NSE= " + NSE );
  }
  if(save_err==1){
  }
  do
    {
      temp=0;
      parallel_DoBP(from,to);
      Modified_search_1(Weight,g1);
      for(j=0; j<L1; j++)    /* V */
       {
                d[j]=g1[j]; /* -dW */
                   g0[j]=g1[j];
                Weight[j] += ao*d[j];
                temp += g1[j]*g1[j];
       }
      NoIter++;
      iter=0; Stop=0;
      do{
              parallel_DoBP(from,to);
              temp1=0;
              temp2=0;
              for(j=0; j<L1; j++)  /* V */
              {
                temp1 += g1[j]*g1[j];
                   temp2 += g1[j]*g0[j];
                g0[j]=g1[j];
              }
              if(Math.sqrt(temp1) > Eplson)
              {
                     bk= (temp1-temp2)/temp;
                     bk=((bk > 0) ? bk : 0);
                     temp=temp1;
                     for(j-0; j<L1; j++)  /* V */
                               d[j]= g1[j]+bk*d[j];
                     Modified_search_1(Weight,d);
                     for(j=0; j<L1; j++)  /* V */
                     Weight[j] += ao*d[j];
```

```
            }
            else Stop=1;
            NoIter++;
            iter++;
        if((plot==1) && ((NoIter % 10)==0))
             System.out.println(" NoIter= " + NoIter + " NSE= " + NSE );

    }while((iter <= L1) && (NoIter <= MaxIter) && (NSE >= MSE) && (Stop !=1));

  }while((NoIter <= MaxIter) && (NSE >= MSE) && (Stop !=1));
}

static int backpropagation(int from,int to)
{
  int i,j,p;
  double rel_error;
  NoIter=0;  /*400*/
  step = 1;

  switch(algo)
  {
  case 0:
       System.out.println("Backpropagation (BP) Learning Algorithm. \n");
       BP_DoBP(from,to);
        break;
  case 1:
       CG_DoBP(from,to);
        break;
  case 2:
       CG_DoBP(from,to);
       break;
  default:
       break;
  }
  System.out.println("finish!");
  System.out.println("Total number of iteration is " + NoIter);
  System.out.println("Normalized system error is " + NSE);

  for(i=from; i<to; i++)
  {
      forward(i);
      for(p=0; p<NUH[NHL+1]; p++)
           Output[i][p]=Oij[NHL+1][p];
  }

  for(i=0; i<NSAMP; i++)
  {
      for(j=0; j<NOA; j++)
      {
          rel_error=Math.abs((Output[i][j]-Desired[i][j])/Desired[i][j])*100;
      }
  }
  return(step);
}

static void ONetArch(int save_arc)
{
  if(save_arc == 1){
     try {
         writeArc("beam.arc");
     }
     catch(IOException e){
     }
```

```
  }
}

static void writeArc(String fileName) throws IOException {
     Writer writer = new FileWriter(fileName);
     PrintWriter printWriter = new PrintWriter(writer);

     printWriter.print( NHL + "\n");
     for(int i=1; i<NHL+1; i++){
        printWriter.print( NUH[i] + " ");
     }

     printWriter.flush();
     printWriter.close();
}

static void OWeight(int save_w)
{
  if(save_w == 1){
     try {
         //writeWeight("beam.w");
         writeWeight(fw);
     }
     catch(IOException e){
     }
  }
}

static void writeWeight(String fileName) throws IOException {
     Writer writer = new FileWriter(fileName);
     PrintWriter printWriter = new PrintWriter(writer);

     for(int i=0; i<L1; i++){
        printWriter.print( Weightp[i] + " ");
     }

     printWriter.flush();
     printWriter.close();
}

static void learning()
{
  init_domain();
  init_neural();
  if(algo==0)
       init_learning_BP();
  else if((algo==1) || (algo==2))
       init_learning_CG();

  init_weight();

  /* save system error as a file */
  save_err=1;
  ferr = "beam.err";
  System.out.println("Save system error in file: beam.err ");

  /* show system error on line */
  plot=1;
  System.out.println("On line show the learning process.");

  /* save arc as a file */
  save_arc=1;
  farc = "beam.arc";
```

```
  System.out.println("Save the topology of the network in file: beam.arc");

  /* asve weights as a file */
  save_w=1;
  System.out.println("Save the weights of the network in file: " + fw );
  step=backpropagation(0,NSAMP);
  if(step==2)
  System.out.println("Max number of iteration reached -- failure to learn");
  ONetArch(save_arc);
  OWeight(save_w);
}

static void performance()
{
  int i,j,p,save;
  double rel_error;

  init_domain();
  init_neural();
  init_weight();

  /* save computed results as a file */
  save_res=1;
  if ( isDisplay_res == false ){
     fres = "beam.res";
  } else if (isDisplay_res == true){
     fres = "display.res";
  }
  System.out.println("Display the analysis results!");

  for(i=0; i<NSAMP; i++)
  {
      forward(i);
      for(p=0; p<NUH[NHL+1]; p++)
         Output[i][p]=Oij[NHL+1][p];
  }
  for(i=0; i<NSAMP; i++)
  {
      for(j=0; j<NOA; j++)
      {
          rel_error=Math.abs((Output[i][j]-Desired[i][j])/Desired[i][j])*100;
          System.out.println("Output " + (j+1) + " Computed= " + Output[i][j] +
                             " Desired= " + Desired[i][j] + " Error= " +
rel_error);
      }
  }

  if(save_res==1){
     try {
        writeRes(fres);
     }
     catch(IOException e){
     }
  }
}

static void writeRes(String fileName) throws IOException {
     Writer writer = new FileWriter(fileName);
     PrintWriter printWriter = new PrintWriter(writer);

     for(int i=0; i<NSAMP; i++){
         for(int j=0; j<NOA; j++){
             printWriter.print( Desired[i][j] + " " + Output[i][j] + "\n");
```

```
        }
    }
    printWriter.flush();
    printWriter.close();
}

static void read_data()
{
   algo    = (int) new Double(algo_Field.getText()).doubleValue();
   NIA     = (int) new Double(NIA_Field.getText()).doubleValue();
   NOA     = (int) new Double(NOA_Field.getText()).doubleValue();
   NSAMP   = (int) new Double(NSAMP_Field.getText()).doubleValue();
   NHL     = (int) new Double(NHL_Field.getText()).doubleValue();
   NUH[1]  = (int) new Double(NUH1_Field.getText()).doubleValue();
   NUH[2]  = (int) new Double(NUH2_Field.getText()).doubleValue();
   NUH[3]  = (int) new Double(NUH3_Field.getText()).doubleValue();
   NUH[4]  = (int) new Double(NUH4_Field.getText()).doubleValue();
   NUH[5]  = (int) new Double(NUH5_Field.getText()).doubleValue();
   LR_Init = new Double(LR_Init_Field.getText()).doubleValue();
   MR      = new Double(MR_Field.getText()).doubleValue();
   MSE     = new Double(MSE_Field.getText()).doubleValue();
   MaxIter = new Double(MaxIter_Field.getText()).doubleValue();

   fpt = input_file_Field.getText();
   fw  = weight_file_Field.getText();
}

public static void main(String[] args)
{
        isDisplay_res = false;
        isInput2 = false;
        isNetwork = true;
        isDisplay = false;

        AnimationPanel myAnimationPanel = new AnimationPanel();
        myAnimationPanel.setBackground(Color.white);

        // Create a window and set its layout manager to be BorderLayout.
        JFrame frame = new JFrame("Neural Network Analysis Program");
        frame.setSize(900,530);
        Container cf = frame.getContentPane();
        cf.setLayout(new BorderLayout());

        // Create two panels and set its layout manager to be FlowLayout.
        JPanel left_panel = new JPanel();
        left_panel.setLayout(new FlowLayout());
        JPanel north_panel_1 = new JPanel();
        north_panel_1.setLayout(new GridLayout(0,1));
        JPanel north_panel_2 = new JPanel();
        north_panel_2.setLayout(new GridLayout(0,1));

        // Create two analysis buttons
        JButton learningButton = new JButton("Learning");
        JButton analysisButton = new JButton("Analysis");
        JButton inputButton = new JButton("Input");
        JButton displayButton = new JButton("Display");
        JButton input2Button = new JButton("Input2");
        JButton display2Button = new JButton("Display2");

        // Add actionlistener to displayButton
        DisplayListener displayListener = new DisplayListener("Display");
        inputButton.addActionListener(displayListener);
        displayButton.addActionListener(myAnimationPanel);
```

```
input2Button.addActionListener(displayListener);
display2Button.addActionListener(myAnimationPanel);

// Add actionlistener to buttons
learningButton.addActionListener(myAnimationPanel);
analysisButton.addActionListener(myAnimationPanel);

// Add Labels
input_file_Label = new JLabel("Input file");
weight_file_Label = new JLabel("Weight file");
algo_Label = new JLabel("Algorithm");
NIA_Label = new JLabel("Number of Input");
NOA_Label = new JLabel("Number of Output");
NSAMP_Label = new JLabel("Number of samples");
NHL_Label = new JLabel("Number of hidden layers");
NUH1_Label = new JLabel("Nodes of 1st hidden layers");
NUH2_Label = new JLabel("Nodes of 2nd hidden layers");
NUH3_Label = new JLabel("Nodes of 3rd hidden layers");
NUH4_Label = new JLabel("Nodes of 4th hidden layers");
NUH5_Label = new JLabel("Nodes of 5th hidden layers");
LR_Init_Label = new JLabel("Learning rate");
MR_Label = new JLabel("Momentum ratio");
MSE_Label = new JLabel("Max allowed system error");
MaxIter_Label = new JLabel("Max allowed interations");

// Add TextFields
input_file_Field = new JTextField("beam3.txt", 4);
weight_file_Field = new JTextField("beam3.w", 4);
algo_Field = new JTextField("1", 4);
NIA_Field = new JTextField("3", 4);
NOA_Field = new JTextField("1", 4);
NSAMP_Field = new JTextField("50", 4);
NHL_Field = new JTextField("1", 4);
NUH1_Field = new JTextField("5", 4);
NUH2_Field = new JTextField("0", 4);
NUH3_Field = new JTextField("0", 4);
NUH4_Field = new JTextField("0", 4);
NUH5_Field = new JTextField("0", 4);
LR_Init_Field = new JTextField("0.9", 4);
MR_Field = new JTextField("0.95", 4);
MSE_Field = new JTextField("0.0001", 4);
MaxIter_Field = new JTextField("1000", 4);


// Add buttons to north_panel
north_panel_1.add(input_file_Label);
north_panel_2.add(input_file_Field);
north_panel_1.add(weight_file_Label);
north_panel_2.add(weight_file_Field);
north_panel_1.add(algo_Label);
north_panel_2.add(algo_Field);
north_panel_1.add(NIA_Label);
north_panel_2.add(NIA_Field);
north_panel_1.add(NOA_Label);
north_panel_2.add(NOA_Field);
north_panel_1.add(NSAMP_Label);
north_panel_2.add(NSAMP_Field);
north_panel_1.add(NHL_Label);
north_panel_2.add(NHL_Field);
north_panel_1.add(NUH1_Label);
north_panel_2.add(NUH1_Field);
north_panel_1.add(NUH2_Label);
```

```
        north_panel_2.add(NUH2_Field);
        north_panel_1.add(NUH3_Label);
        north_panel_2.add(NUH3_Field);
        north_panel_1.add(NUH4_Label);
        north_panel_2.add(NUH4_Field);
        north_panel_1.add(LR_Init_Label);
        north_panel_2.add(LR_Init_Field);
        north_panel_1.add(MR_Label);
        north_panel_2.add(MR_Field);
        north_panel_1.add(MSE_Label);
        north_panel_2.add(MSE_Field);
        north_panel_1.add(MaxIter_Label);
        north_panel_2.add(MaxIter_Field);
        north_panel_1.add(learningButton);
        north_panel_2.add(analysisButton);
        north_panel_1.add(inputButton);
        north_panel_2.add(displayButton);
        north_panel_1.add(input2Button);
        north_panel_2.add(display2Button);

        // Add to left_panel
        left_panel.add(north_panel_1);
        left_panel.add(north_panel_2);

        System.out.println("Artificial Neural Network (ANN) Learning Model)");

        //Create MenuBar for the window frame
        MenuBar menuBar = new MenuBar();
        Menu menuFile = new Menu();
        menuFile.setLabel("File");
        MenuItem menuItemOpen = new MenuItem();
         menuItemOpen.setLabel("Open Learning File");
         menuFile.add(menuItemOpen);
        menuBar.add(menuFile);
        frame.setMenuBar(menuBar);

        // Add panels to the frame
        cf.add(left_panel, BorderLayout.WEST);
        cf.add(myAnimationPanel, BorderLayout.CENTER);

        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        // Make the frame visible after adding the components to it.
        frame.setVisible(true);
} // end of main()
} // end of public class ANN

class DisplayListener extends JFrame implements ActionListener
{
    JTextField massField;
    JDialog dialog = new JDialog(this, "Input Dialog");

    public DisplayListener(String title) {
         super(title);
    }

   public void actionPerformed(ActionEvent e)
    {
        if (e.getActionCommand().equals("Input")) {
              Ann.isInput2 = false;
```

```
        // create three text fields
        massField = new JTextField("0.39904", 10);

        //Layout the text fields in a panel.
        JPanel fieldPane = new JPanel();
        fieldPane.setLayout(new GridLayout(0, 1));
        fieldPane.add(massField);

        //Create three labels.
        JLabel massLabel = new JLabel("End displacement = ");

        //Tell accessibility tools about label/textfield pairs.
        massLabel.setLabelFor(massField);

        //Layout the labels in a panel.
        JPanel labelPane = new JPanel();
        labelPane.setLayout(new GridLayout(0, 1));
        labelPane.add(massLabel);

        //create a ok button
         JButton p_ok_button = new JButton("OK");
         p_ok_button.setPreferredSize(new Dimension(50,40));
         MyPropOkListener myPropOkListener = new MyPropOkListener();
         p_ok_button.addActionListener(myPropOkListener);

        // add panels to properties dialog
        dialog.getContentPane().add(labelPane, BorderLayout.WEST);
        dialog.getContentPane().add(fieldPane, BorderLayout.CENTER);
        dialog.getContentPane().add(p_ok_button, BorderLayout.SOUTH);

        dialog.pack();
        dialog.setVisible(true);
    }
    else if (e.getActionCommand().equals("Input2")) {
        Ann.isInput2 = true;
        // create three text fields
        if (Ann.isInput2 == false){
            massField = new JTextField("0.41227", 10);
        }
        else if (Ann.isInput2 == true){
            massField = new JTextField("0.46168", 10);
        }

        //Layout the text fields in a panel.
        JPanel fieldPane = new JPanel();
        fieldPane.setLayout(new GridLayout(0, 1));
        fieldPane.add(massField);

        //Create three labels.
        JLabel massLabel = new JLabel("End displacement = ");

        //Tell accessibility tools about label/textfield pairs.
        massLabel.setLabelFor(massField);

        //Layout the labels in a panel.
        JPanel labelPane = new JPanel();
        labelPane.setLayout(new GridLayout(0, 1));
        labelPane.add(massLabel);

         //create a ok button
         JButton p_ok_button = new JButton("OK");
         p_ok_button.setPreferredSize(new Dimension(50,40));
         MyPropOkListener myPropOkListener = new MyPropOkListener();
```

```
            p_ok_button.addActionListener(myPropOkListener);

            // add panels to properties dialog
            dialog.getContentPane().add(labelPane, BorderLayout.WEST);
            dialog.getContentPane().add(fieldPane, BorderLayout.CENTER);
            dialog.getContentPane().add(p_ok_button, BorderLayout.SOUTH);
            dialog.pack();
            dialog.setVisible(true);
        }
    }

    class MyPropOkListener implements ActionListener
    {
        public void actionPerformed(ActionEvent e) {
              if (e.getActionCommand().equals("OK")) {
                  Ann.isDisplay_res = true;
                  Ann.read_data();
                  Ann.displacement = Double.parseDouble( massField.getText() );
                  try {
                      write_display_file("beam.dis", Ann.displacement);
                  }
                  catch(IOException e1){
                  }
                  Ann.NSAMP = 1;
                  Ann.fpt = "beam.dis";
                  Ann.task=1;
                  Ann.performance();

                  // read analysis result
                  try{
                      readDisplay("display.res");
                  }
                  catch (IOException e2){
                  }
                  dialog.hide();
              }
         }
    }

    static void write_display_file(String fileName, double displacement) throws
IOException{
       Writer writer = new FileWriter(fileName);
       PrintWriter printWriter = new PrintWriter(writer);
       if (Ann.isInput2 == false){
          printWriter.print(".11504 .13707 " + displacement + " 0.50");
       }
       else if (Ann.isInput2 == true){
          printWriter.print(".00143 .00674 .13659 .16115 " + displacement + "
0.50 0.50");
       }
       printWriter.flush();
       printWriter.close();
    }

   static void readDisplay(String fileName) throws IOException
   {
      Reader fileReader = new FileReader(fileName);
      BufferedReader bufferedReader = new BufferedReader(fileReader);
      String nextLine;

      nextLine = bufferedReader.readLine();
      StringTokenizer tokenizer = new StringTokenizer(nextLine);
```

```
      // 2 times, because the result is the second double value in the first
line
      Ann.display_reduction = Double.parseDouble(tokenizer.nextToken());
      Ann.display_reduction = Double.parseDouble(tokenizer.nextToken());

      if (Ann.isInput2 == true) {
          nextLine = bufferedReader.readLine();
          tokenizer = new StringTokenizer(nextLine);
          // 2 times, because the result is the second double value in the
first line
          Ann.display_reduction2 = Double.parseDouble(tokenizer.nextToken());
          Ann.display_reduction2 = Double.parseDouble(tokenizer.nextToken());
      }

      System.out.println("Ann.display_reduction  = " + Ann.display_reduction );
      System.out.println("Ann.display_reduction2 = " + Ann.display_reduction2);
      bufferedReader.close();
  }
}// class DisplayListener


// AnimationPanel.java
import javax.swing.*;
import java.text.*;
import java.awt.*;
import java.awt.event.*;

class AnimationPanel extends JPanel implements ActionListener
{
    AnimationPanel() {
       setBackground(Color.white);
    }

    public void paintComponent(Graphics g) {
      super.paintComponent(g);
      g.setColor(Color.black);

      if (Ann.isNetwork == true) {
        //NIA
        int x=50, y=50;
        for (int i=0; i<Ann.NIA; i++){
             g.drawArc(x, y+(i*30), 20, 20, 0, 360);
             for (int j=0; j<Ann.NUH[1]; j++){
                 g.drawLine(x+20, y+(i*30)+8, x+100, y+(i*30) + 30*(j-i)+8 );
             }
        }

        //NHL
        for (int i=0; i<Ann.NHL; i++){
            x = 50 + 100*(i+1); y = 50;
            for (int j=0; j<Ann.NUH[i+1] ; j++){
                 g.drawArc(x, y+(j*30), 20, 20, 0, 360);
                 int repeat;
                 if (Ann.NUH[i+2] == 0)
                    repeat= Ann.NOA;
                 else
                    repeat = Ann.NUH[i+2];
                 for (int k=0; k<repeat; k++){
                      g.drawLine(x+20, y+(j*30)+8, x+100, y+(j*30)+30*(k-j)+8);
                 }
            }
        }
```

```
    // NOA
    x = 50 + 100*(Ann.NHL+1);
    y = 50;
    for (int i=0; i<Ann.NOA; i++){
         g.drawArc(x, y+(i*30), 20, 20, 0, 360);
    }

    g.drawString("n=" + Ann.NIA,  50,  40 );
    for (int i=0; i<Ann.NHL ; i++){
         g.drawString("n=" + Ann.NUH[i+1], 50 + 100*(i+1),  40 );
    }
    g.drawString("n=" + Ann.NOA, 50 + 100*(Ann.NHL+1),  40 );
}
else if (Ann.isDisplay == true) {
    g.setColor(Color.black);
    g.drawLine(100,190,100,210);

    // draw blue displacement line
    double scale = 0.000001;
    g.setColor(Color.blue);
    for (int i=0; i<420; i++){
         g.drawLine(100+i,   (int)(200 + i*i*i * scale),
                    100+i+1, (int)(200 + (i+1)*(i+1)*(i+1) * scale) );
    }

    // draw black line
    g.setColor(Color.black);
    g.drawLine(100,200,300,200);
    g.setColor(Color.red);
    g.drawLine(300,200,320,200);
    g.setColor(Color.black);
    g.drawLine(320,200,520,200);

    // display crack reduction
    g.setColor(Color.red);
    DecimalFormat df = new DecimalFormat("0.##");
    g.drawString( df.format(Ann.display_reduction), 300, 190);
}
else if (Ann.isDisplay2 == true) {
    g.setColor(Color.black);
    g.drawLine(100,190,100,210);

    // draw blue displacement line
    double scale = 0.000001;
    g.setColor(Color.blue);
    for (int i=0; i<420; i++){
         g.drawLine(100+i,   (int)(200 + i*i*i * scale),
                    100+i+1, (int)(200 + (i+1)*(i+1)*(i+1) * scale) );
    }

    // draw black/red line
    g.setColor(Color.black);
    g.drawLine(100,200,120,200);
    g.setColor(Color.red);
    g.drawLine(120,200,140,200);
    g.setColor(Color.black);
    g.drawLine(140,200,300,200);
    g.setColor(Color.red);
    g.drawLine(300,200,320,200);
    g.setColor(Color.black);
    g.drawLine(320,200,520,200);

    // display crack reduction
```

```
            g.setColor(Color.red);
            DecimalFormat df = new DecimalFormat("0.##");
            g.drawString( df.format(Ann.display_reduction),  300, 190);
            g.drawString( df.format(Ann.display_reduction2), 120, 190);
        }
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getActionCommand().equals("Learning")) {
            Ann.isDisplay_res = false;
            Ann.isNetwork = true;
            Ann.isDisplay = false;
            Ann.isDisplay2 = false;
            Ann.read_data();
            Ann.task=0;
            Ann.learning();
            System.out.println("Learning...");
            repaint();
        }
        else if (e.getActionCommand().equals("Analysis")) {
            Ann.isDisplay_res = false;
            Ann.isNetwork = true;
            Ann.isDisplay = false;
            Ann.isDisplay2 = false;
            Ann.read_data();
            Ann.task=1;
            Ann.performance();
            System.out.println("Analysis...");
            repaint();
        }
        else if (e.getActionCommand().equals("Display")) {
            Ann.isNetwork = false;
            Ann.isDisplay = true;
            Ann.isDisplay2 = false;
            System.out.println("Display...");
            repaint();
        }
        else if (e.getActionCommand().equals("Display2")) {
            Ann.isNetwork = false;
            Ann.isDisplay = false;
            Ann.isDisplay2 = true;
            System.out.println("Display2...");
            repaint();
        }
        System.out.println("Finished...");
    }
} //class AnimationPanel
```

// beam3.txt

```
.11504 .13641 .39845 .99
.11504 .13641 .39859 .98
.11504 .13642 .39874 .97
.11504 .13643 .39889 .96
.11504 .13644 .39904 .95
.11504 .13644 .39920 .94
.11504 .13645 .39936 .93
.11504 .13646 .39952 .92
.11504 .13647 .39969 .91
.11504 .13647 .39986 .90
.11504 .13648 .40003 .89
.11504 .13649 .40021 .88
```

.11504 .13650 .40039 .87
.11504 .13651 .40058 .86
.11504 .13652 .40077 .85
.11504 .13653 .40096 .84
.11504 .13654 .40116 .83
.11504 .13655 .40137 .82
.11504 .13656 .40158 .81
.11504 .13657 .40180 .80
.11504 .13658 .40202 .79
.11504 .13659 .40224 .78
.11504 .13660 .40248 .77
.11504 .13661 .40271 .76
.11504 .13662 .40296 .75
.11504 .13664 .40321 .74
.11504 .13665 .40347 .73
.11504 .13666 .40373 .72
.11504 .13668 .40401 .71
.11504 .13669 .40429 .70
.11504 .13670 .40458 .69
.11504 .13672 .40488 .68
.11504 .13673 .40518 .67
.11504 .13675 .40550 .66
.11504 .13676 .40582 .65
.11504 .13678 .40616 .64
.11504 .13680 .40651 .63
.11504 .13681 .40686 .62
.11504 .13683 .40723 .61
.11504 .13685 .40761 .60
.11504 .13687 .40801 .59
.11504 .13689 .40842 .58
.11504 .13691 .40884 .57
.11504 .13693 .40928 .56
.11504 .13695 .40973 .55
.11504 .13697 .41020 .54
.11504 .13700 .41069 .53
.11504 .13702 .41119 .52
.11504 .13705 .41172 .51
.11504 .13707 .41227 .50

// beam4.txt
.00143 .00552 .11617 .13770 .40164 .95 .95
.00143 .00552 .11617 .13774 .40246 .95 .90
.00143 .00552 .11617 .13779 .40337 .95 .85
.00143 .00552 .11617 .13784 .40440 .95 .80
.00143 .00552 .11617 .13789 .40556 .95 .75
.00143 .00552 .11617 .13796 .40689 .95 .70
.00143 .00552 .11617 .13803 .40842 .95 .65
.00143 .00552 .11617 .13812 .41021 .95 .60
.00143 .00552 .11617 .13822 .41233 .95 .55
.00143 .00552 .11617 .13834 .41487 .95 .50
.00143 .00560 .11743 .13911 .40453 .90 .95
.00143 .00560 .11743 .13915 .40535 .90 .90
.00143 .00560 .11743 .13919 .40626 .90 .85
.00143 .00560 .11743 .13924 .40729 .90 .80
.00143 .00560 .11743 .13930 .40845 .90 .75
.00143 .00560 .11743 .13936 .40978 .90 .70
.00143 .00560 .11743 .13944 .41131 .90 .65
.00143 .00560 .11743 .13953 .41310 .90 .60
.00143 .00560 .11743 .13963 .41522 .90 .55
.00143 .00560 .11743 .13975 .41776 .90 .50
.00143 .00568 .11884 .14068 .40776 .85 .95
.00143 .00568 .11884 .14072 .40858 .85 .90

.00143 .00568 .11884 .14077 .40949 .85 .85
.00143 .00568 .11884 .14082 .41052 .85 .80
.00143 .00568 .11884 .14087 .41168 .85 .75
.00143 .00568 .11884 .14094 .41301 .85 .70
.00143 .00568 .11884 .14101 .41454 .85 .65
.00143 .00568 .11884 .14110 .41633 .85 .60
.00143 .00568 .11884 .14120 .41845 .85 .55
.00143 .00568 .11884 .14132 .42099 .85 .50
.00143 .00578 .12043 .14245 .41139 .80 .95
.00143 .00578 .12043 .14249 .41221 .80 .90
.00143 .00578 .12043 .14254 .41312 .80 .85
.00143 .00578 .12043 .14259 .41415 .80 .80
.00143 .00578 .12043 .14264 .41531 .80 .75
.00143 .00578 .12043 .14271 .41664 .80 .70
.00143 .00578 .12043 .14278 .41818 .80 .65
.00143 .00578 .12043 .14287 .41997 .80 .60
.00143 .00578 .12043 .14297 .42208 .80 .55
.00143 .00578 .12043 .14309 .42462 .80 .50
.00143 .00588 .12222 .14446 .41551 .75 .95
.00143 .00588 .12222 .14450 .41633 .75 .90
.00143 .00588 .12222 .14454 .41724 .75 .85
.00143 .00588 .12222 .14459 .41827 .75 .80
.00143 .00588 .12222 .14465 .41943 .75 .75
.00143 .00588 .12222 .14472 .42076 .75 .70
.00143 .00588 .12222 .14479 .42229 .75 .65
.00143 .00588 .12222 .14488 .42408 .75 .60
.00143 .00588 .12222 .14498 .42620 .75 .55
.00143 .00588 .12222 .14510 .42874 .75 .50
.00143 .00600 .12427 .14675 .42022 .70 .95
.00143 .00600 .12427 .14679 .42103 .70 .90
.00143 .00600 .12427 .14684 .42195 .70 .85
.00143 .00600 .12427 .14689 .42297 .70 .80
.00143 .00600 .12427 .14694 .42414 .70 .75
.00143 .00600 .12427 .14701 .42547 .70 .70
.00143 .00600 .12427 .14708 .42700 .70 .65
.00143 .00600 .12427 .14717 .42879 .70 .60
.00143 .00600 .12427 .14727 .43091 .70 .55
.00143 .00600 .12427 .14739 .43344 .70 .50
.00143 .00615 .12664 .14940 .42565 .65 .95
.00143 .00615 .12664 .14944 .42646 .65 .90
.00143 .00615 .12664 .14948 .42738 .65 .85
.00143 .00615 .12664 .14953 .42840 .65 .80
.00143 .00615 .12664 .14959 .42957 .65 .75
.00143 .00615 .12664 .14965 .43090 .65 .70
.00143 .00615 .12664 .14973 .43243 .65 .65
.00143 .00615 .12664 .14981 .43422 .65 .60
.00143 .00615 .12664 .14992 .43633 .65 .55
.00143 .00615 .12664 .15004 .43887 .65 .50
.00143 .00631 .12940 .15249 .43198 .60 .95
.00143 .00631 .12940 .15253 .43280 .60 .90
.00143 .00631 .12940 .15257 .43371 .60 .85
.00143 .00631 .12940 .15262 .43474 .60 .80
.00143 .00631 .12940 .15268 .43590 .60 .75
.00143 .00631 .12940 .15274 .43723 .60 .70
.00143 .00631 .12940 .15282 .43876 .60 .65
.00143 .00631 .12940 .15290 .44055 .60 .60
.00143 .00631 .12940 .15300 .44267 .60 .55
.00143 .00631 .12940 .15313 .44521 .60 .50
.00143 .00650 .13267 .15614 .43947 .55 .95
.00143 .00650 .13267 .15617 .44028 .55 .90
.00143 .00650 .13267 .15622 .44120 .55 .85
.00143 .00650 .13267 .15627 .44222 .55 .80

.00143 .00650 .13267 .15633 .44339 .55 .75
.00143 .00650 .13267 .15639 .44472 .55 .70
.00143 .00650 .13267 .15646 .44625 .55 .65
.00143 .00650 .13267 .15655 .44804 .55 .60
.00143 .00650 .13267 .15665 .45016 .55 .55
.00143 .00650 .13267 .15678 .45269 .55 .50
.00143 .00674 .13659 .16051 .44845 .50 .95
.00143 .00674 .13659 .16055 .44927 .50 .90
.00143 .00674 .13659 .16060 .45018 .50 .85
.00143 .00674 .13659 .16065 .45121 .50 .80
.00143 .00674 .13659 .16070 .45237 .50 .75
.00143 .00674 .13659 .16077 .45370 .50 .70
.00143 .00674 .13659 .16084 .45523 .50 .65
.00143 .00674 .13659 .16093 .45702 .50 .60
.00143 .00674 .13659 .16103 .45914 .50 .55
.00143 .00674 .13659 .16115 .46168 .50 .50