SENSITIVITY OF NEURAL NETWORKS TO RANDOM CHANGE WITH PERTURBED WEIGHTS AND BIASES

By

MOHAMMAD A. ALRAB

Bachelor of Engineering

Yarmouk University

Irbid, Jordan

1988

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 1992

Thesis
1995
A459b

SENSITIVITY OF NEURAL NETWORKS TO RANDOM CHANGE WITH PERTURBED WEIGHTS AND BIASES

Thesis Approved:

William David Miller

Thesis Adviser

Huizhu Lu

M. Samadzadeh-H.

Thomas C. Collins

Dean of the Graduate College

ACKNOWLEDGEMENT

I wish to express sincere appreciation to Professor William Miller, my major advisor, for his encouragement and advice throughout my graduate program. Many thanks also go to Dr. Mansur Samadzadeh and Dr. Huizhu Lu for serving on my graduate committee. Their suggestions and support were very helpful throughout the study.

I am also thankful for the friendship of all my fellow graduate students who were always there to reassure me that I would make it. Special thanks goes to Feroze Khalifullah and Raja Baharuddin for their suggestions and encouragements.

My parents, Ahmad and Wesal Alrab, encouraged and supported me all the way and helped me keep the end goal constantly in sight. Thanks go to them for their prayers. My wife Muna provided moral support and was a real believer in my abilities. To my daughter Sarah who gave me the enthusiasm to approach my goal. I extend a sincere thank you to all of these people.

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION 1

II. LITERATURE REVIEW 5

Perfect Learning and Generalization. . . . 5
Monte Carlo Method 7
Random Numbers. 7
Monte Carlo Variance and Estimator. . . 8
Backpropagation Rule 10
Activation Functions. 11
Learning by Pattern or by Epoch 13
Implementation. 13

III. THE PATTERN CLASSIFICATION PROBLEM. 17

The Training Method. 17
Characterization of Input Patterns 20
Training and Testing Simulation Results. . 33

IV. MONTE CARLO SIMULATION RESULTS AND CONCLUSIONS 43

Implementation 43
Simulation Results and Discussion. 46
Conclusion 48

SELECTED BIBLIOGRAPHY 50

APPENDIX A - THE TRAINING PROGRAM. 53

APPENDIX B - THE TESTING AND MONTE CARLO PROGRAM . . 65

APPENDIX C - THE RELATIONSHIP BETWEEN WEIGHT TOLERANCE AND MISCLASSIFICATION RATE. . 80

LIST OF TABLES

Table Page

1. Input Patterns and Their Neighbors. 21

2. Partition A (zero neighbours) 22

3. Partition B (one neighbour) 23

4. Partition C (two neighbours). 23

5. Partition D (three neighbours). 24

6. Characterization of All Patterns. 24

7. Input Patterns and Their Neighbors. 25

8. Partition A (zero neighbours) 26

9. Partition B (one neighbour) 26

10. Partition C (two neighbours). 27

11. Partition D (three neighbours). 27

12. Characterization of All Patterns. 28

13. Partition A (zero neighbours) 29

14. Partition B (one neighbour) 29

15. Partition C (two neighbours). 30

16. Partition D (three neighbours). 31

17. Partition E (four neighbours) 31

18. Characterization of All Patterns. 32

19. Characterization of All Patterns. 33

20. Results for Mixtures of Interior and Border Patterns 34

21. Results Using Only Border Patterns as Training Sets. 35

Table		Page

22. Results for Mixtures of Interior and Border Patterns 35

23. Results Using Only Border Patterns as Training Sets. 37

24. Results for Mixtures of Interior and Border Patterns 38

25. Results Using Only Border Patterns as Training Sets. 41

LIST OF FIGURES

Figure Page

1. Minimum Effect on Generalization 81

2. Maximum Effect on Generalization 82

3. Classification Difficulty = 22.22 Percent. . . 83

4. Classification Difficulty = 33.33 Percent. . . 84

5. Classification Difficulty = 38.46 Percent. . . 85

6. Classification Difficulty = 42.86 Percent. . . 86

7. Various Classification Difficulties. 87

CHAPTER I

INTRODUCTION

Studies have been carried out on neural networks for a long period of time in the hope of simulating human behavior. They are called neural networks because they are composed of computational elements functioning in a way very similar to the biological neural networks [15].

Neural networks represent a more intelligent approach to information processing. These models attempt to accomplish good performance using massively parallel nets composed of many computational elements attached by links with variable weights [19]. Compared to the traditional von Neumann computer which performs a program of instructions sequentially, neural net models stand superior, because they work in parallel, because they can learn or be trained about a certain task, because they can formulate generalizations [21].

Neural networks typically consist of five principal components; computational elements, connections between units, adaptive coefficients of connections, transfer functions, and learning laws [19].

A useful neural network model, especially for classification tasks, is the multilayered feedforward model. These general models are called feedforward networks since

activations are fed from the input layers through the network toward the output layer with one or more hidden layers between the input and output units [25]. These hidden layers are not connected directly to either the input or output units. Typically in these networks units can have continuous values between 0 and 1 as determined by a sigmoidal transfer function.

A hidden layer allows the neural network to form its own internal representation of mapping input tooutput. This network is then independent of the relationships built into the input data but can determine for itself what is important in representing the mapping for the particular decision situation [16]. This provides the neural network with the flexibility to learn any type of input-output relationship.

The network is trained using data to recognize or categorize on the basis of appropriate input data. To use the network to categorize, the attributes of a particular object are presented to the network and the unit values are fed through the network, resulting in the activation at the output layer. This output activation indicates the appropriate categorization of the object [6, 23].

The method of backpropagation has become the standard process used in the training of this type of neural network [12]. Basically, the backpropagation algorithm attempts to minimize the sum of the squares of errors at the output layer during the training process. A training set is comprised of pairs of input values and the desired output

3values which the network should provide by feeding forward the input data through the network. The algorithm computes an error for each output unit proportional to the difference between the obtained network output and the desired output for a particular training case. Network connection weights are adjusted so as to minimize the sum of squared error [19].

The training of a neural network takes place in the following manner. A training set of input patterns is made available. The multilayered network is initialized with random interconnection weights. The input conditions of the training example are presented to the network and the activations are fed forward through the network, resulting in output at the output layer. This output obtained by the network is compared to the desired output for those particular input patterns. Network weights are adjusted such that the difference between the actual output and the desired output is minimized. Adjustments due to the output error are propagated backward through the network, starting at the output layer and moving back toward the input layer. The procedure is repeated over the training set until the network converges. This convergence implies that the neural network has learned the underlying characteristics of the problem and is able to produce the targeted responses given the inputs.

Computer simulations of artificial neural networks store the values of interconnection weights and unit biases in an internal representation (e.g., an array of floating-point

numbers) with accuracies of parts per million or less. In contrast, in hardware implementations of neural nets, devices used to implement weights and biases have limited accuracy, typically specified as a tolerance, such as a tolerance rate or a percent of the nominal value. Previously Stevenson and associates described an analytical study of the sensitivity of layered networks with threshold logic units [20]. They reported that the probability of an error increased as the weight perturbation ratio increased to 0.5 as a maximum limit. In the present study, Monte Carlo techniques are used to investigate the effect of random weight and bias variation on the performance of a feedforward neural network pattern classifier trained with the back propagation algorithm.

In studying the effects of random weight variation we want a classification problem which we could control and characterize precisely. To accomplish this, we attempt to recognize the presence of groups of ones in binary strings. For this problem, the input patterns can be divided into groups characterized by their distances from the class boundary. With various combinations of these groups, we construct training sets, ranging from those containing only typical patterns of each class (interior patterns) to those of border patterns.

CHAPTER II

LITERATURE REVIEW

Perfect Learning and Generalization

Perfect learning and generalization have been considered as major fields in recent research in the area of adaptive training. Ahmad and Tesauro [1] have studied neural network generalization and factors that have influence on it. They determined relationships between the size of the network, the size of the training examples, and the performance of the network. They showed that the output error, in a fixed size network, decreases exponentially with the increase in the training set size. They also showed that for a fixed performance, the size of the training set increases linearly with the size of the network. They found that the border patterns were the most important patterns among all training examples. They showed that if a certain number of random training examples is used to train a network, and if the same number of border patterns is used to train a similar network, then the latter network will generalize better than the former one.

Baum and Haussler [3] studied the relation between the size of network and the number of training patterns chosen at random distribution. They showed that if the number of

training patterns is greater or equal to $O(W/k * \log(N/k))$ where, k is a constant greater than 0 and less than or equal to 1/8, N is number of nodes in the network, W is number of input and output weights, and the network is capable to classify a fraction (1 - k/2) of the training patterns, then the network will correctly classify a fraction of (1 - k) of future test patterns.

Yu and Simmons [24] compared two measures of perfect learning in a feedforward neural network trained by specified input patterns. The first one is the sum of the squared errors. The other one is the correctness ratio which is the percentage of successfully classified patterns in the training set. They showed that the two measures are not similar and they presented the descending epsilon technique with which the backpropagation method results in a high correctness ratio.

Perugini and Engeler [17] examined the learning time for two layer backpropagation networks trained with boolean training examples to classify boolean equations.

Less work has been done on the subject of weight errors. Stevenson and associates [20] analyzed the sensitivity of feedforward layered networks of threshold logic unit elements to weight errors. They approximated and derived a function between the probability of error for a large network output and the percentage change in the weights. They reported that when the number of layers in the network and the change in the weights increases, then the probability of output error increases. They also

reported that in a network which has a large number of weights per unit and units per layer, the output error is independent of the number of weights and units in that network.

Monte Carlo Method

Random Numbers

The numbers, X_1, X_2, ..., X_n in an interval I constitutes a sequence of random numbers if X_i satisfies some distribution properties, and if these distribution properties are invariant for subsequences extracted from the sequence (X_i), for all i in the interval (1,N). These numbers can be used to simulate natural phenomenas using computer, to provide random samples to be examined rather than examiming too many existing cases, to solve complicated numerical problems, to make unbiased decisions, and to test the effectiveness of computer algorithms [13]. For practical purposes, random numbers are obtained by means of digital computers according to arithmetical algorithms, i.e., random numbers generators. Such numbers will not be genuinely random, since they are produced by some deterministic sequence of computing operations. They can be described as pseudorandom numbers.

The basic random numbers sequence is the sequence of uniform random numbers in the interval (0,1). From a sequence of uniform random numbers one may obtain random numbers with any distribution in any interval I. The most

used random numbers generators are the congruential generators including the multiplicative generators [13]:

$$X_{j+1} = AX_j \text{ (modula T)}$$

where X_1 is given, A is a constant used as the multiplier, T is a constant used as the modulus.

And the mixed generators:

$$X_{j+1} = (BX_j + C) \text{ (modula T)}$$

where X_1 is given, B is a constant used as the multiplier, C is a constant used as the increment, T is a constant used as the modulus.

The sequence of random numbers generated by these periodic relations has the diffeciency of repeating itself into cycles of infinite loops. We can achieve maximal cycle lengths by choosing proper values for the constants.

Monte Carlo Variance and Estimator

The Monte Carlo Method involves a random sampling process. Samples are drawn from the original source through sampling procedures governed by specified probability laws [11]. Statistical data are collected from the samples, and consequences concerning the original source will be available through analysis of these data. A different choice of the probability laws and different ways to draw inferences from the data lead to different Monte Carlo techniques [11]. Generally, Monte Carlo methods are designed for the study of complicated systems with many

interacting components. The behavior of the components is governed by known probability laws. It is always possible to incorporate these same laws into the Monte Carlo computational method, so that processes occurring during the simulation will be analogous to processes in the original source [18].

If we use a correctly defined Monte Carlo model to compute the sampling value X with an expected value of E, then in one simulation run we obtain for X the value X_i. Using another random numbers sequence and recomputing the value of X, we get X_j, where X_i is not equal to X_j. Through N simulation runs, the average value of X is AVG:

$$AVG = (1/N) * \sum_{i=1}^{N} X_i$$

AVG becomes concentrated about E as N increases, thus the precision of Monte carlo calculations depends on the value of N. In practice, this precision is usually estimated by the sample variance S [11]:

$$S = (1/N-1) * \sum_{i=1}^{N} (X_i - AVG)$$

The standard deviation is:

STD = sqrt(S)

The upper limit of calculation result is:

U = E + STD

The lower limit of calculation result is:

$L = E - STD$

Backpropagation Rule

In order to apply the backpropagation rule in a network, we must be able to compute the derivative of the error function with respect to any weight in the network and then change the weight according to the rule [19]:

$$\text{Delta}(W_{ij}) = \text{epsilon} * e_i * a_j$$

The weight on each line should be changed by an amount proportional to the product of the error, e, in the unit receiving input along that line, times the activation, a, of the unit sending activation along that line. The difference is in the exact determination of the "e" term. The determination of the error is a recursive process that starts with the output units. If a unit is an output unit, its error is given by

$$e_i = (t_i - a_i) * f'(net_i)$$

where $net_i = \Sigma(W_{ij} * a_j) + bias_i$, and $f'(net_i)$ is the derivative of the activation function with respect to a change in the net input to the unit. The error term for hidden units for which there is no specified target is determined recursively in terms of the error terms of the parent units and in terms of the weights of those connections between the hidden unit and its parents. It is

given by

$$e_i = \Sigma(e_j * W_{ji}) * f'(net_i)$$

whenever the unit is not an output unit.

The application of the rule then involves two phases: During the first phase the input is presented and propagated forward through the network to compute the output value for each unit. This output is then compared with the desired one, resulting in an error term for each output unit. The second phase involves a backward pass through the network during which the error term is computed for each unit in the network. This backward pass allows the recursive computation of the errors. Once these two phases are complete, we can compute for each weight, the product of the error associated with the unit it sends to times the activation of the unit it recieves from. These products can then be used to compute actual weight changes on a pattern by pattern basis, or on overall patterns.

Activation Functions

After computing the net input to each output unit, the activation of the output unit is then determined according to an activation function. Several functions are available [19]:

- Linear function. In this function, the activation of output unit is simply equal to the net input.
- Linear threshold. In this function, each of the output units is a linear threshold unit; that is, its

activation is set to 1 if its net input exceeds 0, and is set to 0 otherwise.

- Stochastic. In this function, the output is set to 1, with a probability p given by the logistic function:

$$P(O_i = 1) = 1 / (1 + e**-net_i)$$

- Continuous sigmoid. In this function, each of the output units takes on an activation that is nonlinearly related to its input according to the logistic function:

$$O_i = 1 / (1 + e ** -net_i)$$

The derivative of the backpropagation learning rule requires that the derivative of the activation function, f'(neti), exists. In most works on backpropagation, the logistic activation function is used, because it is a continuous nonlinear function. In order to apply the learning rule, we need to know the derivative of this function with respect to its net input, net_i. It is given by:

$$d(a_i)/d(net_i) = a_i * (1 - a_i)$$

Thus, for the logistic activation function, the error term, e, for an output unit is given by:

$$e_i = (t_i - a_i) * a_i * (1 - a_i)$$

and the error for a given hidden unit is given by:

$$e_i = a_i * (1 - a_i) * \Sigma(e_j*W_{ji})$$

Learning by Pattern or by Epoch

The derivation of the backpropagation supposes that we are taking the weight changes summed over all patterns. In this case, we can present all patterns and then sum the changes before adding them to the original weights. Instead, we can compute weight changes on each pattern and add them to the original weights after each pattern rather than after each epoch. When there is a very large set of patterns, the version in which weights are changed after each pattern is more satisfying.

Implementation

The program in APPENDIX A implements the backpropagation rule just described. Processing of a single pattern occurs as follows:

A pattern is read from the input file and is clamped on the input units; that is, their activations are set to 1 or 0 based on the values found in the input pattern. Next, activations are computed. For each hidden and output unit, the net input to the unit is computed and then the activation of the unit is set. The routine that performs this computation is:

```
PROCEDURE COMPUTE_OUT;
BEGIN
    loop for all hidden and output units
    initialize netinput by bias value

    loop for all hidden units
    begin
        loop for all input units
        netinput = netinput + (activation * weight)
```

```
        output = activation function of netinput
    end;
    loop for all output units
    begin
        loop for all hidden units
        netinput = netinput + (activation * weight)
        output = activation function of netinput
    end;
END;
```

Next, the error and delta terms are computed. Initially, they are set to 0 for all units. Then, error terms are calculated for each output unit. For these units, error is the difference between the desired and the obtained output of the unit. After the error has been computed for each output unit, we perform a recursive computation of error and delta terms for hidden units. The program iterates backward over the units, starting with the last output unit. The first thing it does in each pass through the loop is set delta for the current unit, which is equal to the error for the unit times the derivative of the activation function. Then, once it has delta for the current unit, the program passes this back to all predecessor units that have connections coming into the current unit. By the time a particular unit becomes the current unit, all of its parents will have already been processed, and the sum of all its error will have been accumulated, so it is ready to have its delta computed. The code for this is as follows:

```
PROCEDURE COMPUTE_ERR;
BEGIN
    loop for all hidden and output units
    initialize error by zero

    loop for all output units
    begin
        calculate difference between desired and actual
        calculate pattern squared error
        accumulate total squared error
        calculate output error
    end;
    loop for all output units
    begin
        loop for all hidden units
        begin
            backpropagate the output error
            calculate hidden error
        end;
    end;
END;
```

After computing errors and deltas, the weight change amounts are then computed from the deltas and activations. The change amounts for the bias terms are also computed. These computations occur in the following routine:

```
PROCEDURE COMPUTE_ERR_MUL_ACTV;
BEGIN
    loop for all hidden units
    begin
        loop for all input nodes
        multiply hidden error by input activation
    end;
    loop for all output units
    begin
        loop for all hidden units
        multiply output error by hidden activation
    end;
END;
```

This routine adds the weight changes caused by the present pattern into an array where they can potenially be

accumulated over patterns. These changes actually lead to changes in the original weights either after processing each pattern or after each entire epoch of processing.

For each weight, a delta weight is first calculated. The delta weight is equal to the accumulated weight changes plus a fraction of the previous delta weight, where the size of the fraction is determined by the momentum. Then, this delta weight is added into the weight, so that the weight's new value is equal to its old value plus the delta weight. The same computation is performed for all of the bias terms. The following routine performs these computations:

```
PROCEDURE CHANGE_WT;
BEGIN
    loop for all hidden units
    begin
        loop for all input units
        begin
            calculate delta of weight
            add delta of weight to original weight
        end;
    end;
    loop for all output units
    begin
        loop for all hidden units
        begin
            calculate delta of weight
            add delta of weight to original weight
        end;
    end;
    loop for all hidden and output units
    begin
        calculate delta of bias
        add delta of bias to original bias
    end;
END;
```

CHAPTER III

THE PATTERN CLASSIFICATION PROBLEM

The Training Method

Multilayered feedforward neural networks are powerful environments which map from a finite dimensional input space to the output space. One of the most desirable characteristics of such networks is their ability to learn from examples and to generalize from the training set to similar data not contained in the training examples. There are three critical factors that affect generalization in neural networks [14]: network architecture, training algorithm, and training set. Architecture determines a group of mappings from the input space to the output space. This group of mappings must be broad enough to include the correct mappings for the problem to be solved. The role of the training algorithm is to obtain this correct mapping using appropriate training examples. Training in feedforward networks can be achieved by gradually changing the weights according to a backpropagation algorithm to minimize the error in given inputs according to desired outputs in the training set. Once the network architecture and the training algorithm have been chosen, the training set will ultimately determine the mapping represented by the

network and its generalization capability. Thus, how to select a training set to accomplish maximum generalization is of central importance for any application.

The selection of certain input patterns to be trained is unlimited. We may choose typical patterns from each class to be used as a training set. The difficulty in this approach is that there are no obvious ways to define and select typical patterns of a class. We may choose patterns that are close to each other in input space even they belong to different classes. These patterns have been called border patterns [1]. Some experimental work has been done, using this approach, on boolean numbers such as the majority function [1]. They have shown that with appropriate network architecture and a backpropagation training algorithm, a training set containing the complete border patterns is sufficient to guarantee a perfect generalization.

In particular, we need to determine border and typical sets. We need to know how the network performs when trained with typical examples selected from both classes. How it performs when trained with both interior and border or incomplete border patterns. We need to know whether it is necessary to use complete border patterns to get perfect generalization.

These questions are investigated using classification in binary strings. The approach will be to partition the whole set of input patterns into groups such that patterns within groups have the same distance from the class boundary, and patterns between groups have a different

distance from the class boundary [22]. The border patterns are those groups near the boundary. On the other hand, typical patterns of a class are those groups that are in the interior of a class, or far away from the class border. By using a combination of these groups, we can form training sets of a mixture of various distances from the boundary, including those of border and typical patterns. This method of selecting the training sets facilitates a systematic method for accomplishing the required sets of input data for this work.

Investigations are performed in a series of experiments that attempt to recognize presence of clumps of ones in binary strings. The first output is desired to be 0 if there are two or more clumps of 1's in the input pattern while the second output is desired to be 1. The first output is desired to be 1 if there are less than two clumps of 1's in the input pattern while the second output is desired to be 0. The networks used are three layer feedforward networks. The network with 5 input nodes and three hidden nodes, which are fully connected to the input and output layers, represents a powerful testing environment because this architecture is successful to realize boolean functions. The output is displayed on two output units. The output function for nodes in the hidden and output layers is a sigmoid.

The network is initialized with random weights. The backpropagation algorithm with momentum is used to train the network. The learning rate and momentum used in all the

experiments are 0.5 and 0.9 respectively. The weights are updated every epoch which consists of all the patterns in the training set. Continuous cycling through the training set proceeded until the sum of squared errors reached 0.001. After training the network, it is ready for testing with test set.

Characterization of Input Patterns

The network with five inputs, with the powerful architecture, will be the major testing environment. There are 32 possible input patterns that can be clamped on the input nodes. Exactly half of the patterns shows two or more clumps of 1's and the other half shows zero or one clump. In the input space, some of these patterns are close to the border separating the two classes and some are located in the interior of each class. To determine which patterns are near the border and which ones are in the interior of a class, the nearest neighbors of each pattern are examined. If a pattern in a given class has at least one nearest neighbor that belongs to the other class, this pattern should be characterized as close to the border. Otherwise, the pattern is characterized to be in the interior of a class. Further, the distance from a pattern to the class border will be ranked according to its number of nearest neighbors in the opposite class.

We can find all the nearest neighbors of a pattern by calculating the Hamming distances from this pattern to the other patterns in the input set. Then, the nearest

neighbors of a pattern are those patterns that differ by one bit from the given pattern. With the defined ranking method, each input pattern can have either 0, 1, 2, or 3 nearest neighbors of opposite class. Accordingly, that pattern can be assigned to one of four groups, A, B, C, or D corresponding to 0, 1, 2, or 3 neighbors respectively. Table 1 shows each input pattern and its nearest neighbors in the opposite class.

TABLE 1

INPUT PATTERNS AND THEIR NEIGHBOURS

Input pattern	Nearest Neighbors in Opposite Class		
00000	-----	-----	-----
00001	00101	01001	10001
00010	01010	10010	-----
00011	01011	10011	-----
00100	00101	10100	-----
00101	00100	00001	00111
00110	10110	-----	-----
00111	00101	10111	-----
01000	01010	01001	-----
01001	01000	00001	-----
01010	01000	00010	01110
01011	00011	01111	-----
01100	01101	-----	-----
01101	01100	01111	-----
01110	01010	-----	-----
01111	01101	01011	-----
10000	10100	10010	10001
10001	10000	00001	-----
10010	10000	00010	-----
10011	00011	-----	-----
10100	10000	00100	11100
10101	-----	-----	-----
10110	00110	11110	-----
10111	00111	11111	-----
11000	11010	11001	-----
11001	11000	-----	-----

TABLE 1 (Continued)

Input pattern	Nearest Neighbors in Opposite Class		
11010	11000	11110	-----
11011	11111	-----	-----
11100	10100	11101	-----
11101	11100	11111	-----
11110	11010	10110	-----
11111	11101	11011	10111

Table 2 shows partition A which consists of input patterns that have 0 neighbors.

TABLE 2

PARTITION A (ZERO NEIGHBOURS)

CLASS 1	CLASS 2
00000	10101

Table 3 shows partition B which consists of input patterns that have 1 neighbor.

TABLE 3

PARTITION B (ONE NEIGHBOUR)

CLASS 1	CLASS 2
00110	10011
01100	11001
01110	11011

Table 4 shows partition C which consists of input patterns that have 2 neighbors.

TABLE 4

PARTITION C (TWO NEIGHBOURS)

CLASS 1	CLASS 2
00010	01001
00011	01011
00100	01101
00111	10001
01000	10010
01111	10110
11000	10111
11100	11010
11110	11101

Table 5 shows partition D which consists of input

patterns that have 3 neighbors.

TABLE 5

PARTITION D (THREE NEIGHBOURS)

CLASS 1	CLASS 2
00001	00101
10000	01010
11111	10100

Table 6 summarizes all the characterizations of input patterns. There are an equal numbers of patterns from each class in any given partition.

TABLE 6

CHARACTERIZATION OF ALL PATTERNS

Partition	Number of Patterns	Number of Neighbors
A	2	0
B	6	1
C	18	2
D	6	3

To support decisions that can be made from testing previous characterizations, more binary strings are partitioned and characterized.

Regarding four input patterns, there are 16 possible inputs that can be clamped on the input nodes. In this case the patterns in the classes are not even. In the input space, some of these patterns are close to the border separating the two classes and some are located in the interior of each class.

Table 7 shows each input pattern and its nearest neighbors in the opposite class.

TABLE 7

INPUT PATTERNS AND THEIR NEIGHBOURS

Input pattern	Nearest Neighbors in Opposite Class		
0000	----	----	----
0001	0101	1001	----
0010	1010	----	----
0011	1011	----	----
0100	0101	----	----
0101	0100	0001	0111
0110	----	----	----
0111	0101	----	----
1000	1010	1001	----
1001	1000	0001	----
1010	1000	0010	1110
1011	0011	1111	----
1100	1101	----	----
1101	1100	1111	----
1110	1010	----	----
1111	1101	1011	----

Table 8 shows partition A which consists of input patterns that have 0 neighbors.

TABLE 8

PARTITION A (ZERO NEIGHBOURS)

CLASS 1	CLASS 2
0000	----
0110	----

Table 9 shows partition B which consists of input patterns that have 1 neighbor.

TABLE 9

PARTITION B (ONE NEIGHBOUR)

CLASS 1	CLASS 2
0010	----
0011	----
0100	----
0111	----
0100	----
1110	----

Table 10 shows partition C which consists of input patterns that have 2 neighbors.

TABLE 10

PARTITION C (TWO NEIGHBOURS)

CLASS 1	CLASS 2
0001	1001
1000	1011
1111	1101

Table 11 shows partition D which consists of input patterns that have 3 neighbors.

TABLE 11

PARTITION D (THREE NEIGHBOURS)

CLASS 1	CLASS 2
----	0101
----	1010

Table 12 summarizes all the characterizations of input patterns. There are different numbers of patterns from each class in the partitions.

TABLE 12

CHARACTERIZATION OF ALL PATTERNS

Partition	Number of Patterns	Number of Neighbors
A	2	0
B	6	1
C	6	2
D	2	3

Regarding six input patterns, there are 64 possible inputs that can be clamped on the input nodes. In this case the patterns in the classes are not even. In the input space, some of these patterns are close to the border separating the two classes and some are located in the interior of each class.

Table 13 shows partition A which consists of input patterns that have 0 neighbors.

TABLE 13

PARTITION A (ZERO NEIGHBOURS)

CLASS 1	CLASS 2
000000	010101
------	100101
------	101001
------	101010
------	101011
------	101101
------	110011
------	110101

Table 14 shows partition B which consists of input patterns that have 1 neighbor.

TABLE 14

PARTITION B (ONE NEIGHBOUR)

CLASS 1	CLASS 2
------	010011
------	011001
------	011011
------	100011
------	100110
------	100111
------	110001
------	110010
------	110110
------	110111
------	111001
------	111011

Table 15 shows partition C which consists of input patterns that have 2 neighbors.

TABLE 15

PARTITION C (TWO NEIGHBOURS)

CLASS 1	CLASS 2
000110	001001
001100	001011
001110	001101
011000	010001
011100	010010
011110	010110
------	010111
------	011010
------	011101
------	100001
------	100010
------	100100
------	101100
------	101110
------	101111
------	110100
------	111010
------	111101

Table 16 shows partition D which consists of input patterns that have 3 neighbors.

TABLE 16

PARTITION D (THREE NEIGHBOURS)

CLASS 1	CLASS 2
000010	000101
000011	001010
000100	010100
000111	101000
001000	------
001111	------
010000	------
011111	------
110000	------
111000	------
111100	------
111110	------

Table 17 shows partition E which consists of input patterns that have 4 neighbors.

TABLE 17

PARTITION E (FOUR NEIGHBOURS)

CLASS 1	CLASS 2
000001	------
100000	------
111111	------

Table 18 summarizes all the characterizations of input patterns. There are different numbers of patterns from each class in the partitions.

TABLE 18

CHARACTERIZATION OF ALL PATTERNS

Partition	Number of Patterns	Number of Neighbors
A	9	0
B	12	1
C	24	2
D	16	3
E	3	4

Regarding seven input patterns, there are 128 possible inputs that can be clamped on the input nodes. In this case the patterns in the classes are not even. In the input space, some of these patterns are close to the border separating the two classes and some are located in the interior of each class.

Table 19 summarizes all the characterizations of input patterns. There are different numbers of patterns from each class in the partitions.

TABLE 19

CHARACTERIZATION OF ALL PATTERNS

Partition	Number of Patterns	Number of Neighbors
A	34	0
B	30	1
C	30	2
D	16	3
E	15	4
F	3	5

Training and Testing Simulation Results

In the experiments, the total number of possible input patterns is divided into two sets: a training set and a testing set. Training sets are formed using various combinations of different groups as defined in the previous section. Each training set is one of three basic types.

1. A Subset of border patterns.
2. Interior patterns only.
3. A Combination of interior and border patterns.

A series of experiments are performed using these training sets. In each of these experiments, the network is tested with a testing set which is the whole or part of the complement set of the corresponding training set. The

classification is considered to be correct if the outputs of the network were within 0.2 of the desired value of 1 or 0.

Table 20 shows the performance results for the experiments using the interior and mixtures of interior and border patterns as training sets for four input networks.

TABLE 20

RESULTS FOR MIXTURES OF INTERIOR AND BORDER PATTERNS

Experiment Number	Training Set	Testing Set	% Correct Ratio
1	A	B+C+D	41.23
2	A+B	C+D	52.00
3	A+C	B+D	52.00
4	A+D	B+C	10.00
5	A+B+C	D	15.50
6	A+B+D	C	52.00
7	A+C+D	B	100.00

Table 21 shows the results of experiments using subsets of border patterns as training examples for four input networks.

TABLE 21

RESULTS USING ONLY BORDER PATTERNS AS TRAINING SETS

Experiment Number	Training Set	Testing Set	% Correct Ratio
8	B	A+C+D	60.55
9	C	A+B+D	55.10
10	D	A+B+C	15.00
11	B+C	A+D	35.10
12	B+D	A+C	52.00
13	C+D	A+B	100.00
14	B+C+D	A	100.00

Table 22 shows the performance results for the experiments using the interior and mixtures of interior and border patterns as training sets for six input networks.

TABLE 22

RESULTS FOR MIXTURES OF INTERIOR AND BORDER PATTERNS

Experiment Number	Training Set	Testing Set	% Correct Ratio
1	A	B+C+D+E	45.00
2	A+B	C+D+E	55.00

TABLE 22 (Continued)

Experiment Number	Training Set	Testing Set	% Correct Ratio
3	A+C	B+D+E	55.00
4	A+D	B+C+E	10.00
5	A+E	B+C+D	60.00
6	A+B+C	D+E	14.20
7	A+B+D	C+E	40.00
8	A+B+E	C+D	60.11
9	A+C+D	B+E	70.00
10	A+C+E	B+D	60.00
11	A+D+E	B+C	100.00
12	A+B+C+D	E	60.50
13	A+B+C+E	D	50.00
14	A+B+D+E	C	100.00
15	A+C+D+E	B	100.00

Table 23 shows the results of experiments using subsets of border patterns as training examples for six input networks.

TABLE 23

RESULTS USING ONLY BORDER PATTERNS
AS TRAINING SETS

Experiment Number	Training Set	Testing Set	% Correct Ratio
16	B	A+C+D+E	60.55
17	C	A+B+D+E	55.10
18	D	A+B+C+E	17.00
19	E	A+B+C+D	12.00
20	B+C	A+D+E	37.10
21	B+D	A+C+E	55.00
22	B+E	A+C+D	57.00
23	C+D	A+B+E	50.00
24	C+E	A+B+D	70.00
25	D+E	A+B+C	100.00
26	B+C+D	A+E	75.00
27	B+C+E	A+D	50.00
28	B+D+E	A+C	100.00
29	C+D+E	A+B	100.00
30	B+C+D+E	A	100.00

Table 24 shows the performance results for the experiments using the interior and mixtures of interior and border patterns as training sets. In the first four

TABLE 24

RESULTS FOR MIXTURES OF INTERIOR AND BORDER PATTERNS

Experiment Number	Training Set	Testing Set	% Correct Ratio
1	A	B	50.00
2	A	C	38.89
3	A	D	33.33
4	A	B+C+D	40.00
5	A+B	C	66.67
6	A+B	D	0.00
7	A+B	C+D	50.00
8	A+C	B	100.00
9	A+C	D	0.00
10	A+C	B+D	50.00
11	A+D	B	0.00
12	A+D	C	11.11
13	A+D	B+C	8.33
14	A+B+C	D	16.67
15	A+B+D	C	50.00
16	A+C+D	B	100.00

experiments, the network was trained exclusively with interior patterns of both classes. The average result for the four experiments is 40.55 percent. This result suggests

that this minimum training set contains little specific information about the class boundary. Thus typical patterns are not candidates for optimal generalization. The other experiments gave more interesting results. In both experiments 8 and 9, the training set was successful to classify Group B, but it was unable to classify Group D. In both experiments 14 and 16, training sets of equal size were used but they produced two extreme performance in generalization. The first was failure while the second was very successful. It can be seen that different training sets with similar size may produce different performance in generalization. In fact, the two training sets differ by only one group; instead of B in experiment 14, D was used in experiment 16. All of experiments 3, 6, 9, and 14 were unable to classify Group D. We can observe that all the trained networks having border groups other than D in their training sets are incapable to classify Group D patterns correctly. This suggests that Group D, the closest group to the boundary, contains some vital information about the class boundary without which a perfect generalization is impossible.

Table 25 shows the results of experiments using subsets of border patterns as training examples.

Group B is the group of border patterns that are closest to the interior of a class. It does not have precise information about class boundary. On a closer look at the test data, we see, from experiments 17, 18, 19, that it is perfect classifier for A, it is an acceptable

classifier for C, but it is incapable to classify D.

Group C has the majority number of border patterns and is second closest to the class boundary. On a closer look at the test data, in experiments 21 and 22, Group C was perfect classifier for A and B. In experiment 23, the trained networks failed to classify D patterns correctly. We can see from experiment 24 that networks trained with C are only average performers.

Group D is the closest group to the border. Alone, it was incapable to classify any group in the complement test set, experiment 25, 26, 27.

Experiment 31 shows that networks trained with the training set B+C are below average performers. Similarly, experiment 34 shows that networks trained with the training set B+D are average performers.

Networks trained with the training set C+D were able to classify Group A, Group B, and the combination of these two groups in the complement test set. Addition of B to this training set in experiment 38 has no effect on the performance resulted. Similarly, addition of A to this training set in experiment 16 has no effect on the generalization. Considering all the above results, we can see that C+D turned out to be a perfect set.

We see that all the perfect sets of border patterns (experiments 16, 37 and 38) have Group D as their subset. Although, D alone is a relatively poor training set for generalization, we see that networks trained with border patterns excluding D were completely unable to classify D.

Addition of A or B to the perfect training set C+D has no effect on generalization. We also see that an arbitrary subset of border patterns except C+D, e.g., experiment 31 and 34, is not necessarily a powerful training set for generalization.

TABLE 25

RESULTS USING ONLY BORDER PATTERNS AS TRAINING SETS

Experiment Number	Training Set	Testing Set	% Correct Ratio
17	B	A	100.00
18	B	C	77.78
19	B	D	0.00
20	B	A+C+D	61.54
21	C	A	100.00
22	C	B	100.00
23	C	D	0.00
24	C	A+B+D	57.14
25	D	A	0.00
26	D	B	0.00
27	D	C	22.22
28	D	A+B+C	15.38
29	B+C	A	100.00
30	B+C	D	16.67
31	B+C	A+D	37.50

TABLE 25 (Continued)

Experiment Number	Training Set	Testing Set	% Correct Ratio
32	B+D	A	100.00
33	B+D	C	44.44
34	B+D	A+C	50.00
35	C+D	A	100.00
36	C+D	B	100.00
37	C+D	A+B	100.00
38	B+C+D	A	100.00

CHAPTER IV

MONTE CARLO SIMULATION
RESULTS AND CONCLUSIONS

Implementation

The program in APPENDIX B implements the testing procedure and then Monte Carlo calculations. Testing of a single pattern occurs as follows.

First, all input patterns are read and stored in an array structure to facilitate communication with patterns. Second, trained weights and biases are read and stored in a series of arrays. After that, a single pattern is selected to be clamped on the input units setting their activations to 1 or 0 according to the input pattern. Next, the output of the network is computed using a routine similar to that in APPENDIX A. The routine is:

```
PROCEDURE TEST_NET;
BEGIN
    for i = first_hidden to last_output  do
    begin
        netinput [i] := bias[i];
        for j = first_weight_to[i] to last_weight_to[i] do
        begin
            netinput[i] := netinput[i] +
                                (activation[j] * W[i][j]);
        end;
        activation[i] := logistic(netinput[i]);
    end;
END;
```

The order of complexity for the procedure above is

O [(nunits-ninputs)*nweights]. After computing the output of the tested pattern, its contribution to the performance is calculated using the following routine.

```
PROCEDURE COMP_PRFRMNC;
BEGIN
    if (pattern in class1) then
    pss := (1.0 - first_out) * (1.0 - first_out) +
           (0.0 - second_out) * (0.0 - second_out);
    else
    pat_sqrd_err := ((0.0 - first_out) *
                     (0.0 - first_out)) +
                    ((1.0 - second_out) *
                     (1.0 - second_out));
    tot_sqrd_err := tot_sqrd_err + pss;
    if (pattern in class1) then
    begin
        if (first_out between 0.8 and 1.2) and
           (second_out between -0.2 and 0.2) then
        correct_classf := correct_classf + 1;
        else
        mis_classf := misclassf + 1;
    end;
END;
```

The order of complexity for the procedure above is O(1). After training each network with a specified training set Monte Carlo techniques were used to define the relationship between the tolerance and the network's misclassification rate. Weight and bias tolerance rates were varied from 0 to 0.5, as a practical limit, in gradual steps with each trained network. For each tolerance rate we select 1000 independent sets of perturbed weights and biases with individual weights and biases from random number generator. Using several sets from both classes, we measure the misclassification rate with each set of perturbed weights and then compute the corresponding mean value and the standard deviation of the 1000 values. Calculations

above are implemented in the following routine.

```
PROCEDURE MONTE_CARLO;
BEGIN
    Y := 1000;
    tolerance := 0.0;
    for i:= 1 to 10 do
    begin
        tolerance := tolerance + 0.05;
        for j:= 1 to trunc(Y) do
        begin
            for k:= 1 to weight_num do
            begin
                temp :=
                       tolerance*train_weight[k]*random;
                out_weight[k] := train_weight[k] + temp;
            end;
            for k:= 1 to (hidden+out) do
            begin
                temp2 := tolerance*tbias[k]*random;
                outbias[k] := bias[k] + temp2;
            end;
            for k:= 1 to pattern_num do
            begin
                TEST_NET;
                COMP_PRFRMNC;
            end;
            expmnt_misclassf := expmnt_misclassf +
                                (mis_classf * (1.0/Y));
            sum_of_misclassf := sum_of_misclassf +
                                             mis_classf;
            sum_of_sqrd_misclassf :=
                               sum_of_sqrd_misclassf +
                               (mis_classf * mis_classf);
        end;
        variance_of_misclassf := (1.0/(Y - 1.0)) *
                                (sum_of_sqrd_misclassf -
                                (1.0/Y) *
                   (sum_of_misclassf*sum_of_midclassf));
    end;
END;
```

The order of complexity for the procedure above is

O[ntolerance[Y[weight_num+(hidden+out)+
(pattern_num*O[(nunits-ninputs*weight_num)])]]].

Simulation Results and Discussion

In all the studies two class problems were considered. After training each network with a training set, the resulting set of trained weights was taken, after the sum of total squared errors reached 0.001 or less, and then used for simulation. Monte Carlo studies were performed with several models using different test sets. All sets were taken into consideration except sets with performance less than 50 percent at zero perturbation because, this performance is considered beyond the theoretical limit, i.e., 50% in the two class problem.

Examining figures in APPENDIX C, figure 1 shows the averaged misclassification rate as a function of tolerance rate for a network trained with patterns from Group B and tested with patterns from Group A. This network has perfect performance, zero error, at zero tolerance rate. The data indicates that the random variations in the weights and biases did affect the performance of this network, and the effect increases with the tolerance level. For example, the misclassification rate increased from 0 percent with the unperturbed weights to 0.99 percent with a tolerance rate of 0.25 and then to 6.9 percent with a tolerance rate of 0.5. This shows an overall increase of 6.9 percent in the misclassification rate. This increase was the minimum among all other networks with perfect generalization.

Similarly, figure 2 shows the averaged misclassification rate as a function of tolerance rate for a

network trained with patterns from the set A+C and tested with patterns from Group B. Also, this network has perfect performance at zero tolerance rate. The data indicates that the random variations in the weights and biases did affect the performance of this network, and the effect increases with the tolerance level. For example, the misclassification rate increased from 0 percent with the unperturbed weights to 2 percent with a tolerance rate of 0.25 and then to 25.27 percent with a tolerance rate of 0.5. This shows an overall increase of 25.27 percent in the misclassification rate. On the contrary, compared to the previous network, this increase was the maximum among all other networks with perfect generalization.

Figure 3 shows the misclassification data for an imperfect network trained with patterns from Group B and tested with patterns from Group C. It has an error of 22.22 percent with the unperturbed weights. Supporting previous conclusions, the data in the figure indicates that the random variations in the weights and biases have an effect on the performance of the network, and the effect increases when weight and bias error increases. For example, the output error increased from 22.22 percent with the unperturbed weights to 25.15 percent with a tolerance rate of 0.25 and then to 28.52 percent with a tolerance rate of 0.5. Random variations contributed an overall increase of 6.3 percent to the output error.

Figure 4 shows the output error for more difficult classification. The network was trained with patterns from

the set A+B and tested with patterns from Group C. The difficulty in classification was 33.33 with correct weights. This difficulty increased to 35.08 percent with a weight error of 0.25 and then to 44.84 with a weight error of 0.5.

The data in figure 5 is very similar to that in the previous figure. The difficulty in classification is 38.46. Group B is the trained set and Groups A, C, and D are the test set. The difficulty increased to 38.53 percent with a weight error of 0.25 and then to 43.39 with a weight error of 0.5.

The difficulty of classification in figure 6 was the most among all considerable networks. The network was trained with patterns from Group C and tested with patterns in the set A+B+D. Compared to all other networks, it has the worst output error started at a value of 42.86 percent. Perturbation in weights and biases increased the error to a value close to the theoretical upper limit beyond which, the classification is impossible in two class problems.

Conclusion

A method was used to partition input binary patterns into groups based on their neighbor numbers which indicate the distance from the input pattern to the class boundary. By using various combinations of these groups, it is possible to construct a variety of training sets including interior and border sets. Supporting results in [1], it was shown that if a certain number of random training examples was used to train a network, and if the

same number of border patterns was used to train a similar network, then the latter network will generalize better than the former one. This suggests that border patterns are very recommended to be included in input examples used to train networks. Also, this approach of using border patterns facilitates systematic studies in the contribution of training sets to generalization.

Furthermore, this study showed that errors in the weights and biases in a neural network classifier affect its performance, and the magnitude of the effect increases as the magnitude of the random perturbation increases. All studies of the relationship between the weight tolerance and the failure rate, recommend that weight error should be less than 25 percent. Tests in which biases were maintained correct and weights were perturbed, showed that performances of networks were more sensitive to bias errors than weight errors. Tests in which one of the input and output weights was fixed and the other was varied, showed that errors in output weights have more influence on failure rate than errors in input weights.

SELECTED BIBLIOGRAPHY

[1] Ahmad, S. and G. Tesauro, "Scaling and Generalization in Neural Networks: A Case Study", Advances in Neural Information Processing Systems, vol. 1, Morgan Kaufman, 1989, pp. 160-168.

[2] Ahmad, S., G. Tesauro and Y. He, "Asymptotic Convergence of Backpropagation: Numerical Experiments", Advances in Neural Information Processing Systems, vol. 2, Morgan Kaufman, 1989, pp. 606-613.

[3] Baum, E. and D. Haussler,"What Size Net Gives Valid Generalization?", Advances in Neural Information Processing Systems, vol. 1, Morgan Kaufman, 1989, pp. 81-89.

[4] Brodsky, S.A and C.C. Guest, "Binary Backpropagation in Content Addressable Memory", Proceedings of the International Joint Conference on Neural Networks, vol. 3, 1990, pp. 205-210.

[5] Caviglia, D.D, M. Valle, and G.M. Bisio, "Effects of Weight Discretization on the Backpropagation Learning Method: Algorithm Design and Hardware Realization", Proceedings of the International Joint Conference on Neural Networks, vol. 2, 1990, pp. 631-637.

[6] Cheung, R., I. Lusting and A. Kornhauser, "Relative Effectiveness of Training Set Patterns for Backpropagation", Proceedings of the International Joint Conference on Neural Networks, vol. 1, June 1990, pp. 673-678.

[7] Faggin, F. and C. Mead, An Introduction to Neural and Electronic Networks, Academic Press, 1990.

[8] Fogel, D.B., "An Information Criterion for Optimal Neural Network Selection", IEEE Transactions on Neural Networks, vol. 2, No. 5, Sept 1991, pp. 490-497.

[9] Fukushima, K., "Neocognitron: A Hierarchical Neural Network Capable of Visual Pattern Recognition", Neural Networks, vol. 1, 1988, pp. 119-130.

[10] Guyon, L., L. Poujaud, L. Personnaz and G. Dreyfus, "Comparing Different Neural Network Architectures for Classifying Handwritten Digits", Proceedings of the International Joint Conference on Neural Networks, vol. 2, 1989, pp. 127-132.

[11] Hammersley, J.M. and D.C. Handscomb, Monte Carlo Methods, Spottiswoode & Co., Ltd., 1964.

[12] Hinton, G.E., "Connectionist Learning Procedures", Artificial Intelligence, vol. 40, 1989, pp. 185-234.

[13] Knuth, D.E., The Art of Computer Programming, Addison-Wesley Publishing Company, 1969.

[14] Lippman, R., "An Introduction to Computing with Neural Nets", IEEE ASSP Magazine, April 1987, pp. 4-22.

[15] Masson, E. and Y. Wang, "Introduction to Computation and Learning in Artificial Neural Networks", European Journal of Operational Research, vol. 47, 1990, pp. 1-28.

[16] Mirzai, A.R., Artificial Intelligence Concepts and Applications in Engineering, T.J. Press(Padstow, Great Britain), 1990.

[17] Perugini, N. and W. Engeler, "Neural Network Learning Time: Effects of Network and Training Set Size", Proceedings of the International Joint Conference on Neural Networks, vol. 2, June 1989, pp. 395-401.

[18] Rubinstein, R.Y., Simulation and the Monte Carlo Method, John Wiley & Sons, Inc., 1981.

[19] Rumelhart, D.E. and J.L. McClelland, Parallel Distributed Processing, MIT Press, 1986.

[20] Stevenson, M., R. Winter and B. Widrow, "Sensitivity of Feedforward Neural Networks to Weight Errors", IEEE Transactions on Neural Networks, vol. 1, March 1990, pp. 71-80.

[21] Takeda, M. and J.W. Goodman, "Neural Networks for Computation: Number Representations and Programing Complexity", Applied Optics, vol. 25, No. 18, Sept 1986, pp. 3033-3046.

[22] Weideman, W.E., M.T. Manry and H.C. Yau, "A Comparison of A Nearest Neighbor Classifier and A Neural Network for Numeric Handprint Character Recognition", Proceedings of the International Joint Conference on Neural Networks, vol. 1, 1989, pp. 117-120.

[23] Yamada, K., H. Kami, J. Tsukumo and T. Temma, "Handwritten Numeral Recognition by Multilayered Neural Network with Improved Learning Algorithm", Proceedings of the International Joint Conference on Neural Networks, vol. 2, 1989, pp. 259-266.

[24] Yu, Y. and R. Simmons, "Descending Epsilon in Back Propagation: A Technique for Better Generalization", Proceedings of the International Joint Conference on Neural Networks, vol. 3, June 1990, pp. 167-172.

[25] Yu, Y. and R.F. Simmons, "Extra Output Biased Learning" Proceedings of the International Joint Conference on Neural Networks, vol. 3, June 1990, pp. 161-166.

APPENDIX A

THE TRAINING PROGRAM

```
(*****************************************************)
(*                                                   *)
(* THIS PROGRAM IMPLEMENTS THE BACKPROPAGATION       *)
(* PROCESS. THE PROGRAM MAKES USE OF THE NETWORK     *)
(* SPECIFICATION ENTERED BY THE USER, WHICH INDICATES *)
(* THE ARCHITECTURE OF THE NETWORK. THE NETWORKS ARE *)
(* ASSUMED TO BE FEEDFORWARD NEURAL NETWORKS.        *)
(* THE USER SPECIFICATIONS INDICATE HOW MANY TOTAL   *)
(* UNITS ARE IN THE NETWORK, AND HOW MANY ARE INPUT  *)
(* UNITS, HIDDEN UNITS AND OUTPUT UNITS.             *)
(*                                                   *)
(*****************************************************)


PROGRAM TRAIN2(input,output);

const
     MAX1 = 50;
     MAX2 = 1000;

TYPE

   outunits = array [1..MAX1] of real;
           (* OUTPUT UNITS *)

   inputwts = array [1..MAX2] of real;
            (* INITIAL WTS *)

   outputwts = array [1..MAX2] of real;
             (* TRAINED WTS *)

   biases    = array [1..MAX1]  of real;
             (* INPUT BIASES *)

   outbiases = array [1..MAX1]  of real;
             (* OUT BIASES *)

   t_delta_bias = array [1..MAX1] of real;
               (* DELTA BIAS *)

   t_err_actv_b = array [1..MAX1] of real;
          (* ERROR MULT BY ACTIVATION *)

   t_w       = array [1..MAX1,1..MAX1] of real;
               (* WEIGHT ARRAY *)

   t_err_actv_w = array [1..MAX1,1..MAX1] of real;
          (* ERROR MULTIPLIED BY ACTIVATION *)

   t_delta_w = array [1..MAX1,1..MAX1] of real;
              (* DELTA OF WEIGHTS *)
```

```
   t_net      = array [1..MAX1] of real;
                  (* NET INPUTS *)

   t_error    = array [1..MAX1] of real;
              (* ERROR ARRAY *)

   allpats    = array [1..MAX2] of integer;
             (* ARRAY FOR INPUT PATTERNS *)


VAR
   i,j,k,e : integer;
   count: integer;
   curr : integer;

   pc1  : integer;(* COUNTER FOR PATTERNS OF CLASS ONE*)
   pc2  : integer;(* COUNTER FOR PATTERNS OF CLASS TWO*)
   pc   : integer;(* COUNTER FOR ALL PATTERNS          *)

   pss   : real;  (* PATTERN SUM SQUARED ERROR         *)
   tss   : real;  (* TOTAL PATTERN SUM SQUARED ERROR   *)
   str1  : string[80];(* FILE NAME OF INITIAL WEIGHTS *)
   str2  : string[80];(* FILE NAME OF INPUT PATTERNS  *)
   str3  : string[80];(* FILE NAME OF OUTPUT WEIGHTS  *)
   epoch : integer;

   ptrnfl   : text; (* FILE OF INPUT PATTERNS         *)
   inwtfl   : text; (* FILE OF INITIAL WEIGHTS        *)
   owtfl    : text; (* FILE OF OUTPUT TRAINED WEIGHTS *)

   inwts       : inputwts;  (* INITIAL WTS ARRAY      *)
   outwts      : outputwts; (* TRAINED WTS ARRAY      *)
   w           : t_w;       (* WEIGHT MATRIX          *)
   err_actv_w  : t_err_actv_w;(*ERR MULT BY ACTV FOR WT*)
   delta_w     : t_delta_w;   (* WEIGHT CHANGE MATRIX  *)

   bias        : biases; (* INITIAL BIASES ARRAY      *)
   outbias     : outbiases;(* OUT TRAINED BIASES ARRAY *)
   err_actv_b  : t_err_actv_b;(*ERR MULT BY ACTV FOR BS*)
   delta_bias  : t_delta_bias;(* BIAS CHANGE ARRAY     *)

   pats     : allpats; (* ARRAY FOR INPUT PATTERNS    *)
   o        : outunits;(* ARRAY FOR ACTIVATIONS       *)
   net      : t_net; (* ARRAY FOR NET INPUTS TO NODES *)
   error    : t_error;(* ARRAY FOR ERRORS             *)

   ninputs  : integer; (* NUMBER OF INPUT NODES       *)
   nhiddens : integer; (* NUMBER OF HIDDEN NODES      *)
   nouts    : integer; (* NUMBER OF OUTPUT NODES      *)
   nunits   : integer; (* NUMBER OF ALL NODES         *)
   epsilon  : real;
   momentum : real;

   min_tss_err : real;
```

```
(*****************************************************)
(* P R O C E D U R E       I N I T I A L I Z E 1    *)
(*****************************************************)
(*                                                   *)
(* THIS PROCEDURE IS SIMPLY TO KEEP INITIALIZING THE *)
(* ARRAYS THAT RECORD WEIGHTS AND BIASES CHANGES. IT *)
(* INITIALIZES THE ARRAYS WITH ZERO VALUES.          *)
(*                                                   *)
(*****************************************************)

PROCEDURE INITIALIZE1;

var i, j : integer;

BEGIN

    epoch := 0;
    nunits := ninputs + nhiddens + nouts;

    for i:= 1 to MAX1 do
    begin
        for j:= 1 to MAX1 do
        delta_w[i][j] := 0.0;

        delta_bias[i] := 0.0;
    end;

END; (* PROCEDURE INITIALIZE1 *)


(*****************************************************)
(* P R O C E D U R E       I N I T I A L I Z E 2    *)
(*****************************************************)
(*                                                   *)
(* THIS PROCEDURE IS SIMPLY TO KEEP INITIALIZING THE *)
(* ARRAYS THAT RECORD THE MULTIPLICATION OF ERRORS BY *)
(* ACTIVATION FOR BOTH WEIGHTS AND BIASES.           *)
(* IT INITIALIZES THE ARRAYS WITH ZERO VALUES.       *)
(*                                                   *)
(*****************************************************)

PROCEDURE INITIALIZE2;

var i, j : integer;

BEGIN

    tss := 0.0;

    for i:= 1 to MAX1 do
    begin
```

```
        for j:= 1 to MAX1 do
        err_actv_w[i][j] := 0.0;

        err_actv_b[i] := 0.0;
    end;

END; (* PROCEDURE INITIALIZE2 *)



(******************************************************)
(*   P R O C E D U R E       C O M P U T E _ O U T   *)
(******************************************************)
(*                                                    *)
(* THIS PROCEDURE IS TO CALCULATE THE NET INPUT FOR   *)
(* EACH NODE IN THE NETWORK AND THEN USE THE LOGISTIC *)
(* FUNCTION TO CALCULATE THE CORRESPONDING ACTIVATION *)
(* FOR EACH NODE.                                     *)
(*                                                    *)
(******************************************************)

PROCEDURE COMPUTE_OUT;

var  i, j : integer;

BEGIN

    curr := (count-1) * ninputs;

    for i:= 1 to nunits-ninputs do
    net[i] := outbias[i];

    for i:= 1 to nhiddens do
    begin
        for j:= 1 to ninputs do
        net[i] := net[i] + (pats[curr+j] * w[i][j]);

        o(.i.) := 1.0/ (1.0 + exp(-net(.i.)));
    end;

    for i:= nhiddens+1 to nunits-ninputs do
    begin
        for j:= 1 to nhiddens do
        net[i] := net[i] + (o[j] * w[i][j]);

        o(.i.) := 1.0/ (1.0 + exp(-net(.i.)));
    end;

END;  (* PROCEDURE COMPUTE_OUT *)
```

```
(*****************************************************)
(*   P R O C E D U R E     C O M P U T E _ E R R   *)
(*****************************************************)
(*                                                   *)
(* THIS PROCEDURE COMPUTES THE ERROR TERM FOR EACH    *)
(* OUTPUT AND HIDDEN UNIT. AFTER THE ERROR HAS BEEN   *)
(* COMPUTED FOR EACH OUTPUT UNIT, IT ITERATES BACKWARD*)
(* OVER THE UNITS PASSING THE ERROR OF CURRENT UNIT TO*)
(* ALL UNITS THAT HAVE CONNECTIONS COMING INTO THE    *)
(* CURRENT ONE.                                       *)
(*                                                   *)
(*****************************************************)

PROCEDURE COMPUTE_ERR;

var i, j : integer;

BEGIN

    for i:= 1 to nunits-ninputs do
    error[i] := 0.0;

    for i:= nhiddens+1 to nhiddens+nouts-1 do
    begin
        if (count <= pc1) then
        begin
            error[i]   := 1.0 - o[i];
            error[i+1] := 0.0 - o[i+1];
        end
        else
        begin
            error[i]   := 0.0 - o[i];
            error[i+1] := 1.0 - o[i+1];
        end;

        pss := (error[i] * error[i]) +
               (error[i+1] * error[i+1]);
        tss := tss + pss;

        error[i]   := error[i] * o(.i.) *
                      (1.0 - o(.i.));
        error[i+1] := error[i+1] * o(.i+1.) *
                      (1.0 - o(.i+1.));
    end;

    for i:= nhiddens+1 to nunits-ninputs do
    begin
        for j:= 1 to nhiddens do
        error[j] := error[j] + (error[i] * w[i][j]);
    end;
```

```
    for i:= 1 to nhiddens do
    error(.i.) := error(.i.) * o(.i.) * (1.0 - o(.i.));

END;  (* PROCEDURE COMPUTE_ERR *)
(*******************************************************)
(* P R O C E D U R E   C O M P U T E _ ERR_ MUL_ ACTV *)
(*******************************************************)
(*                                                     *)
(* THIS PROCEDURE IS TO COMPUTE THE MULTIPLICATION OF *)
(* ERROR OF THE RECIEVING NODE BY THE ACTIVATION OF   *)
(* THE SENDING NODE FOR ALL NODES.                     *)
(*                                                     *)
(*******************************************************)

PROCEDURE COMPUTE_ERR_MUL_ACTV;

var i, j : integer;

BEGIN

    curr := (count-1) * ninputs;

    for i:= 1 to nhiddens do
    begin
        for j:= 1 to ninputs do
        err_actv_w[i][j] := err_actv_w[i][j] +
                            (error[i] * pats[j+curr]);
    end;

    for i:= nhiddens+1 to nunits-ninputs do
    begin
        for j:= 1 to nhiddens do
        err_actv_w[i][j] := err_actv_w[i][j] +
                            (error[i] * o[j]);
    end;

    for i:= 1 to nunits-ninputs do
    err_actv_b[i] := err_actv_b[i] + (error[i] * 1.0);

END; (* PROCEDURE COMPUTE_ERR_MUL_ACTV *)
```

```
(*****************************************************)
(*   P R O C E D U R E       C H A N G E _ W T      *)
(*****************************************************)
(*                                                   *)
(* THIS PROCEDURE CHANGES NETWORK'S WEIGHTS TO THE   *)
(* NEW SET OF ALTERED WEIGHTS THAT PRODUCE THE FINAL *)
(* VALUES OF TRAINED WEIGHTS TO BE USED IN TESTING   *)
(* PHASE                                             *)
(*                                                   *)
(*****************************************************)

PROCEDURE CHANGE_WT;

var i, j : integer;

BEGIN

    for i:= 1 to nhiddens do
    begin
        for j:= 1 to ninputs do
        begin
            delta_w[i][j]:= (epsilon * err_actv_w[i][j])
                          + (momentum * delta_w[i][j]);
            w[i][j]      := w[i][j] + delta_w[i][j];
        end;
    end;


    for i:= nhiddens+1 to nunits-ninputs do
    begin
        for j:= 1 to nhiddens do
        begin
            delta_w[i][j]:= (epsilon * err_actv_w[i][j])
                          + (momentum * delta_w[i][j]);
            w[i][j] := w[i][j] + delta_w[i][j];
        end;
    end;


    for i:= 1 to nunits-ninputs do
    begin
        delta_bias[i] := (epsilon * err_actv_b[i])
                       + (momentum * delta_bias[i]);
        outbias[i] := outbias[i] + delta_bias[i];
    end;

END; (* PROCEDURE CHANGE_WT *)
```

```
(*****************************************************)
(*   P R O C E D U R E      R E A D _ D A T A       *)
(*****************************************************)
(*                                                   *)
(* THIS PROCEDURE IS TO READ INPUT PATTERNS FROM THE *)
(* PATTERN FILE. THEN IT WILL LOAD THE BIAS AND      *)
(* WEIGHT ARRAYS BY RANDOM VALUES GENERATED FROM     *)
(* UNIFORM RANDOM DISTRIBUTION.                      *)
(*                                                   *)
(*****************************************************)


PROCEDURE  READ_DATA;

var  i, j : integer;

BEGIN

    pc1 := 0;
    pc2 := 0;
    i   := 1;


    while not eof(ptrnfl) do
    begin
        readln(ptrnfl, e );

        case e of
          1:   pc1 := pc1 + 1;
          2:   pc2 := pc2 + 1;
        end;

        while not eoln(ptrnfl) do
        begin
            read(ptrnfl, pats(.i.));
            i := i+1;
        end;
        readln(ptrnfl);

    end;   (* EOF *)
    pc := pc1 + pc2;


    for i:= 1 to nunits-ninputs do
    begin
        bias(.i.) := random;
        outbias(.i.) := bias(.i.);
    end;

    for i:= 1 to (nouts*nhiddens + nhiddens*ninputs) do
    inwts(.i.) := random;


END; (* PROCEDURE READ_DATA *)
```

```
(*********************************************************)
(*   P R O C E D U R E          L O A D _ N E T          *)
(*********************************************************)
(*                                                       *)
(* THIS PROCEDURE IS TO LOAD THE WEIGHT MATRIX FROM      *)
(* THE INPUT WEIGHT ARRAY WHICH HAS BEEN FILLED BY       *)
(* RANDOM VALUES.                                        *)
(*                                                       *)
(*********************************************************)

PROCEDURE LOAD_NET;

var i, j, k : integer;

BEGIN

    k := 1;

    for i:= 1 to nhiddens do
    begin
        for j:= 1 to ninputs do
        begin
            w[i][j] := inwts[k];
            k := k+1;
        end;
    end;

    for i:= nhiddens+1 to nunits-ninputs do
    begin
        for j:= 1 to nhiddens do
        begin
            w[i][j] := inwts[k];
            k := k+1;
        end;
    end;


END;  (* PROCEDURE LOAD_NET *)



(*********************************************************)
(*   P R O C E D U R E          P R I N T _ T S S        *)
(*********************************************************)
(*                                                       *)
(* THIS PROCEDURE IS TO DISPLAY THE TOTAL SQUARED        *)
(* ERROR AFTER EACH EPOCH.                               *)
(*                                                       *)
(*********************************************************)

PROCEDURE PRINT_TSS;
```

```
BEGIN

    writeln('Epoch Counter              = ',epoch:8);
    writeln;
    writeln('TSS After Epoch Above      = ',tss:8:4);
    writeln;
    writeln;

END; (* PROCEDURE PRINT_TSS *)



(*******************************************************)
(*    M A I N        P R O G R A M           *)
(*******************************************************)

BEGIN

    writeln;
    writeln('ENTER NUMBER OF INPUT NODES');
    writeln;
    readln(ninputs);
    writeln;
    writeln('ENTER NUMBER OF HIDDEN NODES');
    writeln;
    readln(nhiddens);
    writeln;
    writeln('ENTER NUMBER OF OUTPUT NODES');
    writeln;
    readln(nouts);
    writeln;
    writeln('ENTER VALUE FOR EPSILON');
    writeln;
    readln(epsilon);
    writeln;
    writeln('ENTER VALUE FOR MOMENTUM');
    writeln;
    readln(momentum);
    writeln;
    writeln('ENTER MINIMUM TSS ERROR REQUIRED');
    writeln;
    readln(min_tss_err);
    writeln;
    writeln('ENTER INPUT PATTERN FILE NAME');
    writeln;
    readln(str2);
    writeln;
    assign(ptrnfl,str2);
    reset(ptrnfl);
    writeln('ENTER FILE NAME TO SAVE OUPUT WEIGHTS');
    writeln;
    readln(str3);
    writeln;
    assign(owtfl,str3);
```

```
    rewrite(owtfl);

    INITIALIZE1;
    READ_DATA;
    LOAD_NET;

    REPEAT

    BEGIN

        INITIALIZE2;

        for count:= 1 to pc do
        begin

            COMPUTE_OUT;
            COMPUTE_ERR;
            COMPUTE_ERR_MUL_ACTV;

        end;

        epoch := epoch + 1;

        PRINT_TSS;
        CHANGE_WT;

    END;
    UNTIL (tss <= min_tss_err);


    for i:= 1 to nunits-ninputs do
    write(owtfl,outbias(.i.):8:2);
    writeln(owtfl);

    for i := 1 to nhiddens do
    begin
        for j:= 1 to ninputs do
        write(owtfl,w[i][j]:8:2);
        writeln(owtfl);
    end;

    for i:= nhiddens+1 to nunits-ninputs do
    begin
        for j:= 1 to nhiddens do
        write(owtfl,w[i][j]:8:2);
        writeln(owtfl);
    end;

    writeln(owtfl);


close(ptrnfl);
close(owtfl);

END. (* MAIN PROGRAM *)
```

APPENDIX B

THE TESTING AND MONTE CARLO PROGRAM

```
(*****************************************************)
(*                                                   *)
(* THIS PROGRAM IS USED TO STUDY THE EFFECT OF RANDOM *)
(* WEIGHT AND BIAS VARIATIONS ON THE SENSITIVITY OF   *)
(* FEEDFORWARD NEURAL NETWORKS TRAINED WITH THE       *)
(* STANDARD BACKPROPAGATION RULE. AFTER TRAINING EACH *)
(* NETWORK WITH THE REQUIRED TRAINING SET, MONTE CARLO*)
(* METHOD IS USED TO DEFINE THE RELATIONSHIP BETWEEN  *)
(* THE TOLERANCE ON THE WEIGHTS AND BIASES AND THE    *)
(* NETWORK MISCLASSIFICATION RATE.                    *)
(*                                                   *)
(*****************************************************)


PROGRAM SENSITIVITY (input,output);

const
     MAX1 = 50;
     MAX2 = 1000;

TYPE

   outunits = array [1..MAX1] of real;
           (* OUTPUT UNITS *)

   inputwts = array [1..MAX2] of real;
            (* INITIAL WTS *)

   outputwts = array [1..MAX2] of real;
             (* TRAINED WTS *)

   biases    = array [1..MAX1]  of real;
             (* INPUT BIASES *)

   outbiases = array [1..MAX1]  of real;
             (* OUT BIASES *)

   t_delta_bias = array [1..MAX1] of real;
               (* DELTA BIAS *)

   t_err_actv_b = array [1..MAX1] of real;
          (* ERROR MULT BY ACTIVATION *)

   t_w       = array [1..MAX1,1..MAX1] of real;
               (* WEIGHT ARRAY *)

   t_err_actv_w = array [1..MAX1,1..MAX1] of real;
          (* ERROR MULTIPLIED BY ACTIVATION *)

   t_delta_w = array [1..MAX1,1..MAX1] of real;
              (* DELTA OF WEIGHTS *)
```

```
   t_net      = array [1..MAX1] of real;
                  (* NET INPUTS *)

   t_error    = array [1..MAX1] of real;
              (* ERROR ARRAY *)

   allpats    = array [1..MAX2] of integer;
             (* ARRAY FOR INPUT PATTERNS *)


VAR

   i,j,k,e : integer;
   count: integer;
   curr : integer;
   pc1  : integer;(* COUNTER FOR PATTERNS OF CLASS ONE*)
   pc2  : integer;(* COUNTER FOR PATTERNS OF CLASS TWO*)
   pc   : integer;(* COUNTER FOR ALL PATTERNS          *)

   pss   : real;  (* PATTERN SUM SQUARED ERROR         *)
   tss   : real;  (* TOTAL PATTERN SUM SQUARED ERROR   *)
   str1  : string[80];(* FILE NAME OF INITIAL WEIGHTS *)
   str2  : string[80];(* FILE NAME OF INPUT PATTERNS  *)
   str3  : string[80];(* FILE NAME OF OUTPUT WEIGHTS  *)
   str4  : string[80];
   str5  : string[80];

   ptrnfl    : text; (* FILE OF INPUT PATTERNS          *)
   inwtfl    : text; (* FILE OF INITIAL WEIGHTS         *)
   outfl     : text; (* FILE OF OUTPUT                  *)

   inwts       : inputwts;  (* INITIAL WTS ARRAY        *)
   outwts      : outputwts; (* TRAINED WTS ARRAY        *)
   w           : t_w;       (* WEIGHT MATRIX            *)
   err_actv_w  : t_err_actv_w;(*ERR MULT BY ACTV FOR WT*)
   delta_w     : t_delta_w;   (* WEIGHT CHANGE MATRIX  *)

   bias        : biases; (* INITIAL BIASES ARRAY        *)
   outbias     : outbiases;(* OUT TRAINED BIASES ARRAY *)
   err_actv_b  : t_err_actv_b;(*ERR MULT BY ACTV FOR BS*)
   delta_bias  : t_delta_bias;(* BIAS CHANGE ARRAY     *)

   pats      : allpats; (* ARRAY FOR INPUT PATTERNS    *)
   o         : outunits;(* ARRAY FOR ACTIVATIONS       *)
   net       : t_net; (* ARRAY FOR NET INPUTS TO NODES *)
   error     : t_error;(* ARRAY FOR ERRORS             *)

   ninputs  : integer; (* NUMBER OF INPUT NODES        *)
   nhiddens : integer; (* NUMBER OF HIDDEN NODES       *)
   nouts    : integer; (* NUMBER OF OUTPUT NODES       *)
   nunits   : integer; (* NUMBER OF ALL NODES          *)


   correct_classf, cc : real; (* COUNTER FOR CC        *)
```

```
   mis_classf, mc      : real; (* COUNTER FOR MC          *)
   y, u, u2, u3, u4    : real; (* FOR CALCULATIONS        *)
   exptss, vartss      : real;
   expmis_classf       : real; (* EXPERIMENTAL VALUES *)
   varmis_classf       : real; (* VARIANCE VALUE          *)
   stdmis_classf       : real; (* STANDARD DEVIATION  *)
   stdtss              : real;
   mtss, ptss          : real; (* MINUS AND PLUS VALUE *)
   mmis_classf         : real; (* MINUS VALUE             *)
   pmis_classf         : real; (* PLUS VALUE              *)
   r, lamda            : real;
   temp, temp2         : real;


(*******************************************************)
(*    P R O C E D U R E       T E S T _ N E T          *)
(*******************************************************)
(*                                                     *)
(* THIS PROCEDURE IS TO CALCULATE THE NET INPUT FOR    *)
(* EACH NODE IN THE NETWORK AND THEN USE THE LOGISTIC *)
(* FUNCTION TO CALCULATE THE CORRESPONDING ACTIVATION *)
(* FOR EACH NODE.                                      *)
(*                                                     *)
(*******************************************************)

PROCEDURE TEST_NET;

var  i, j : integer;

BEGIN

    curr := (count-1) * ninputs;

    for i:= 1 to nunits-ninputs do
    net[i] := outbias[i];

    for i:= 1 to nhiddens do
    begin
        for j:= 1 to ninputs do
        net[i] := net[i] + (pats[curr+j] * w[i][j]);

        o(.i.) := 1.0/ (1.0 + exp(-net(.i.)));
    end;

    for i:= nhiddens+1 to nunits-ninputs do
    begin
        for j:= 1 to nhiddens do
        net[i] := net[i] + (o[j] * w[i][j]);

        o(.i.) := 1.0/ (1.0 + exp(-net(.i.)));
    end;

END;  (* PROCEDURE TEST_NET*)
```

```
(*****************************************************)
(*    P R O C E D U R E        R E S E T _ T S S    *)
(*****************************************************)
(*                                                   *)
(* THIS PROCEDURE KEEPS INITIALIZING THE TOTAL       *)
(* SQUARED ERROR, AND THE MISCLASSIFICATION VALUES.  *)
(*                                                   *)
(*****************************************************)

PROCEDURE  RESET_TSS;

BEGIN

    tss            := 0.0;

    mis_classf     := 0;

    correct_classf := 0;

END; (* PROCEDURE RESET_TSS *)


(*****************************************************)
(*    P R O C E D U R E       I N I T _ V A R S     *)
(*****************************************************)
(*                                                   *)
(* THIS PROCEDURE IS SIMPLY TO KEEP INITIALIZING     *)
(* VALUES WHICH ARE NEEDED TO BE RESET EVERY NEW     *)
(* CYCLE OF CALCULATION.                             *)
(*                                                   *)
(*****************************************************)

PROCEDURE  INIT_VARS;

BEGIN

    nunits := ninputs + nhiddens + nouts;

    exptss        := 0.0;

    expmis_classf := 0.0;

    u             := 0.0;

    u2            := 0.0;

    u3            := 0.0;

    u4            := 0.0;

    RESET_TSS;

END; (* PROCEDURE INIT_VARS *)
```

```
(*******************************************************)
(*   P R O C E D U R E      R E A D _ D A T A         *)
(*******************************************************)
(*                                                     *)
(* THIS PROCEDURE IS TO READ INPUT PATTERNS FROM THE  *)
(* PATTERN FILE. THEN IT WILL LOAD THE BIAS AND        *)
(* WEIGHT ARRAYS BY THE TRAINED VALUES OBTAINED FROM  *)
(* THE RESULTING WEIGHTS AND BIASES PRODUCED BY        *)
(* THE BACKPROPAGATION ALGORITHM.                      *)
(*                                                     *)
(*******************************************************)

PROCEDURE  READ_DATA;

var  i, j, n : integer;

BEGIN

    pc1 := 0;
    pc2 := 0;
    n   := 1;

    while not eof(ptrnfl) do
    begin
        readln(ptrnfl, e );

        case e of
          1:   pc1 := pc1 + 1;
          2:   pc2 := pc2 + 1;
        end;

        while not eoln(ptrnfl) do
        begin
            read(ptrnfl, pats(.n.));
            n := n+1;
        end;
        readln(ptrnfl);

    end;   (* WHILE *)

    pc := pc1 + pc2;

    for i:= 1 to nunits-ninputs do
    begin
        read(inwtfl, bias(.i.));
        outbias(.i.) := bias(.i.);
    end;
    readln(inwtfl);
```

```
    n := 1;
    for i:= 1 to nhiddens do
    begin
        for j:= 1 to ninputs do
        begin
            read(inwtfl,inwts(.n.));
            n := n+1;
        end;
        readln(inwtfl);
    end;

    for i:= nhiddens+1 to nunits-ninputs do
    begin
        for j:= 1 to nhiddens do
        begin
            read(inwtfl,inwts(.n.));
            n := n+1;
        end;
        readln(inwtfl);
    end;


END; (* PROCEDURE READ_DATA *)



(******************************************************)
(*   P R O C E D U R E         L O A D _ N E T        *)
(******************************************************)
(*                                                    *)
(* THIS PROCEDURE IS TO LOAD THE WEIGHT MATRIX FROM   *)
(* THE INPUT WEIGHT ARRAY WHICH HAS BEEN FILLED BY    *)
(* THE TRAINED WEIGHTS RESULTING FROM THE             *)
(* BACKPROPAGATION PROCESS.                           *)
(*                                                    *)
(******************************************************)

PROCEDURE LOAD_NET;

var i, j, k : integer;

BEGIN

    k := 1;

    for i:= 1 to nhiddens do
    begin
        for j:= 1 to ninputs do
        begin
            w[i][j] := inwts[k];
            k := k+1;
        end;
    end;
```

```
    for i:= nhiddens+1 to nunits-ninputs do
    begin
        for j:= 1 to nhiddens do
        begin
            w[i][j] := inwts[k];
            k := k+1;
        end;
    end;


END;  (* PROCEDURE LOAD_NET *)



(******************************************************)
(*                                                    *)
(* THIS PROCEDURE CALCULATES THE SQUARED ERROR FOR    *)
(* EACH PATTERN AND THE GLOBAL SUM OF SQUARED ERROR   *)
(* FOR ALL PATTERNS. IT ALSO COMPARES THE OBTAINED    *)
(* OUTPUT WITH THE TARGET OUTPUT FOR EACH PATTERN.    *)
(* IF THE ACTUAL OUTPUT IS CLOSE TO THE DESIRED ONE   *)
(* THEN THE PATTERN IS CONSIDERED AS WELL_CALSSIFIED. *)
(* IF THE ACTUAL OUTPUT IS NOT CLOSE TO THE DESIRED   *)
(* ONE THEN THE PATTERN IS CONSIDERED AS MISCALSSIFIED*)
(*                                                    *)
(******************************************************)

PROCEDURE COMP_PRFRMNC;

var i, j : integer;

BEGIN

    for i:= nhiddens+1 to nhiddens+nouts-1 do
    begin
        if (count <= pc1) then
        pss := ((1.0 - o(.i.))*(1.0 - o(.i.))) +
               ((0.0 -o(.i+1.))*(0.0 - o(.i+1.)))

        else

        pss := ((0.0 - o(.i.))*(0.0 - o(.i.))) +
               ((1.0 -o(.i+1.))*(1.0 - o(.i+1.)));

        tss := tss + pss;  (* GLOBAL SUM OF PSS FOR
                                       A PATTERN FILE *)


                   (* THE MISCLASSIFICATION  RULE *)

        if(count <= pc1) then
```

```
        begin

            if( (o(.i.) >= 0.8) and (o(.i.) <= 1.2) and
              (o(.i+1.) <= 0.2) and (o(.i+1.) >= -0.2) )
            then
            correct_classf := correct_classf + 1

            else

            mis_classf := mis_classf + 1;

        end
        else
        begin

            if( (o(.i+1.) >= 0.8) and (o(.i+1.) <= 1.2)
            and (o(.i.) <= 0.2) and (o(.i.) >= -0.2) )
            then
            correct_classf := correct_classf + 1

            else

            mis_classf := mis_classf + 1;

        end;

    end;

END;  (* PROCEDURE COMP_PRFRMNC *)


(*****************************************************)
(* P R O C E D U R E   N O R M A L I Z E _ P R I N T *)
(*****************************************************)
(*                                                   *)
(* THIS PROCEDURE NORMALIZES MISCLASSIFICATION RATE  *)
(* TO %100 AND THEN IT PRINTS OUT RESULTS FOR TSS    *)
(* AND CORRECT CLASSIFICATION RATIO.                 *)
(*                                                   *)
(*****************************************************)

PROCEDURE NORMALIZE_PRINT;

BEGIN

    mc   := mis_classf;
    mc   := (mc/pc) * 100.0;

    cc   := correct_classf;
    cc   := (cc/pc) * 100.0;
```

```
    writeln(outfl);
    writeln(outfl);
    writeln(outfl);
      write(outfl,'      *');
writeln(outfl,'****************************************');
      write(outfl,'      *');
    writeln(outfl,'                                  *');
      write(outfl,'      *');
    writeln(outfl,'                                  *');
      write(outfl,'      *');
    writeln(outfl,'                                  *');
      write(outfl,'      *        TRAINING SET  =',
                                              str4:6);
    writeln(outfl,'              *');
      write(outfl,'      *');
    writeln(outfl,'                                  *');
      write(outfl,'      *');
    writeln(outfl,'                                  *');
      write(outfl,'      *        TESTING SET   =',
                                              str5:6);
    writeln(outfl,'              *');
      write(outfl,'      *');
    writeln(outfl,'                                  *');
      write(outfl,'      *');
    writeln(outfl,'                                  *');
      write(outfl,'      *   CORRECT CLASSIFICATION %=',
                                              cc:8:2);
    writeln(outfl,'   *');
      write(outfl,'      *');
    writeln(outfl,'                                  *');
      write(outfl,'      *');
    writeln(outfl,'                                  *');
      write(outfl,'      *');
    writeln(outfl,'                                  *');
      write(outfl,'      *');
writeln(outfl,'****************************************');
    writeln(outfl);




    writeln(outfl);
    writeln(outfl,'*************    TRAINING SET =',
                                              str4:6);
    writeln(outfl);
    writeln(outfl,'*************    TESTING SET  =',
                                              str5:6);
    writeln(outfl);
    writeln(outfl,'******* START MISSCLASSIFICATION % ='
                                             ,mc:8:2);
    writeln(outfl);
    write(outfl,'           THE MONTE CARLO ');
    writeln(outfl,'SIMULATION RESULTS');
```

```
    write(outfl,'           ');
    writeln(outfl,'__________________________________');
    writeln(outfl);
    writeln(outfl);

END;  (* PROCEDURE NORMALIZE_PRINT *)



(*****************************************************)
(* P R O C E D U R E     C H A N G E _ W E I G H T S *)
(*****************************************************)
(*                                                   *)
(* THIS PROCEDURE CHANGES NETWORK'S WEIGHTS TO THE    *)
(* NEW SET OF ALTERED WEIGHTS TO STUDY THE EFFECT OF  *)
(* WEIGHT PERTURBATION RATIO ON THE PERFORMANCE OF    *)
(* THE SPECIFIED NETWORK.                             *)
(*                                                   *)
(*****************************************************)

PROCEDURE CHANGE_WEIGHTS;

var i, j, k : integer;

BEGIN

    k := 1;

    for i:= 1 to nhiddens do
    begin
        for j:= 1 to ninputs do
        begin
            w[i][j] := outwts[k];
            k := k+1;
        end;
    end;

    for i:= nhiddens+1 to nunits-ninputs do
    begin
        for j:= 1 to nhiddens do
        begin
            w[i][j] := outwts[k];
            k := k+1;
        end;
    end;


END;  (* PROCEDURE CHANGE_WEIGHTS *)
```

```
(*****************************************************)
(*    M A I N        P R O G R A M               *)
(*****************************************************)

BEGIN

    writeln;
    writeln('ENTER NUMBER OF INPUT NODES');
    writeln;
    readln(ninputs);
    writeln;
    writeln('ENTER NUMBER OF HIDDEN NODES');
    writeln;
    readln(nhiddens);
    writeln;
    writeln('ENTER NUMBER OF OUTPUT NODES');
    writeln;
    readln(nouts);
    writeln;
    writeln('ENTER TRAINING SET');
    writeln;
    readln(str4);
    writeln;
    writeln('ENTER TESTING SET');
    writeln;
    readln(str5);
    writeln;
    writeln('ENTER INPUT WEIGHT FILE NAME');
    writeln;
    readln(str1);
    writeln;
    assign(inwtfl,str1);
    reset(inwtfl);
    writeln('ENTER INPUT PATTERN FILE NAME');
    writeln;
    readln(str2);
    writeln;
    assign(ptrnfl,str2);
    reset(ptrnfl);
    writeln('ENTER OUTPUT FILE NAME');
    writeln;
    readln(str3);
    writeln;
    assign(outfl,str3);
    rewrite(outfl);


    INIT_VARS;        (* MOD1 : NETWORK INITIALIZATION *)
    READ_DATA;
    LOAD_NET;

                  (* MOD2 : TRAINED WEIGHTS PERFORMENCE *)
    for count:= 1 to pc do
    begin
```

```
    TEST_NET;
    COMP_PRFRMNC;
end;   (* PATTERN FILE TEST *)

NORMALIZE_PRINT;


            (* MOD3 : THE MONTE CARLO METHOD *)

Y := 1000.0;
lamda := 0.00;


FOR i:= 1 to 10 DO

begin
    INIT_VARS;
    lamda := lamda + 0.05;

    writeln('lamda = ',lamda:8:3);

    FOR j:= 1 to trunc(Y) DO
    begin
        RESET_TSS;

        for k:= 1 to
             (nouts*nhiddens + nhiddens*ninputs) do
        begin

            r := random;
            temp := 4 * lamda * inwts(.k.) *
                                             (r -0.5);
            outwts(.k.) := temp + inwts(.k.);
        end;

        for k:= 1 to nunits-ninputs do
        begin

            r := random;
            temp2 := 4 * lamda * bias(.k.) *
                                             (r -0.5);
            outbias(.k.) := temp2 + bias(.k.);
        end;


        CHANGE_WEIGHTS;

        for count:= 1 to pc do
        begin
            TEST_NET;
            COMP_PRFRMNC;
        end;
```

```
    exptss         := exptss + (tss * (1.0/Y));
    u              := u + tss;
    u2             := u2 + (tss * tss);

    mc             := mis_classf;
    mc             := (mc/pc) * 100.0;

    expmis_classf := expmis_classf +
                       (mc * (1.0/Y));
    u3             := u3 + mc;
    u4             := u4 + (mc * mc);

END;


vartss          := (1.0/(Y - 1.0) *
                   (u2 - ((1.0/Y)*(u*u))));

varmis_classf   := (1.0/(Y - 1.0) *
                   (u4 - ((1.0/Y)*(u3*u3))));

stdtss          := sqrt(vartss);

stdmis_classf   := sqrt(varmis_classf);


mtss            := exptss - stdtss;

ptss            := exptss + stdtss;

mmis_classf     := expmis_classf - stdmis_classf;

pmis_classf     := expmis_classf + stdmis_classf;


writeln(outfl);
write(outfl,' THIS IS A NEW TEST WITH');
writeln(outfl,'TOLERANCE RATE =', lamda:8:2);
writeln(outfl);

writeln(outfl,'E(tss)   is         :'
                                   ,exptss:8:2);
writeln(outfl,'MNS(tss) is         :',mtss:8:2);
writeln(outfl,'PLS(tss) is         :',ptss:8:2);
writeln(outfl);

writeln(outfl,'E(mis_classf) is % :'
                            ,expmis_classf:8:2);
writeln(outfl,'MNS(mis_classf) is :'
                              ,mmis_classf:8:2);
writeln(outfl,'PLS(mis_classf) is :'
                              ,pmis_classf:8:2);
writeln(outfl);
```

```
        writeln(outfl);
        writeln(outfl);

    END;

close(inwtfl);
close(ptrnfl);
close(outfl);

END.   (* MAIN PROGRAM *)
```

APPENDIX C

THE RELATIONSHIP BETWEEN WEIGHT TOLERANCE AND MISCLASSIFICATION RATE

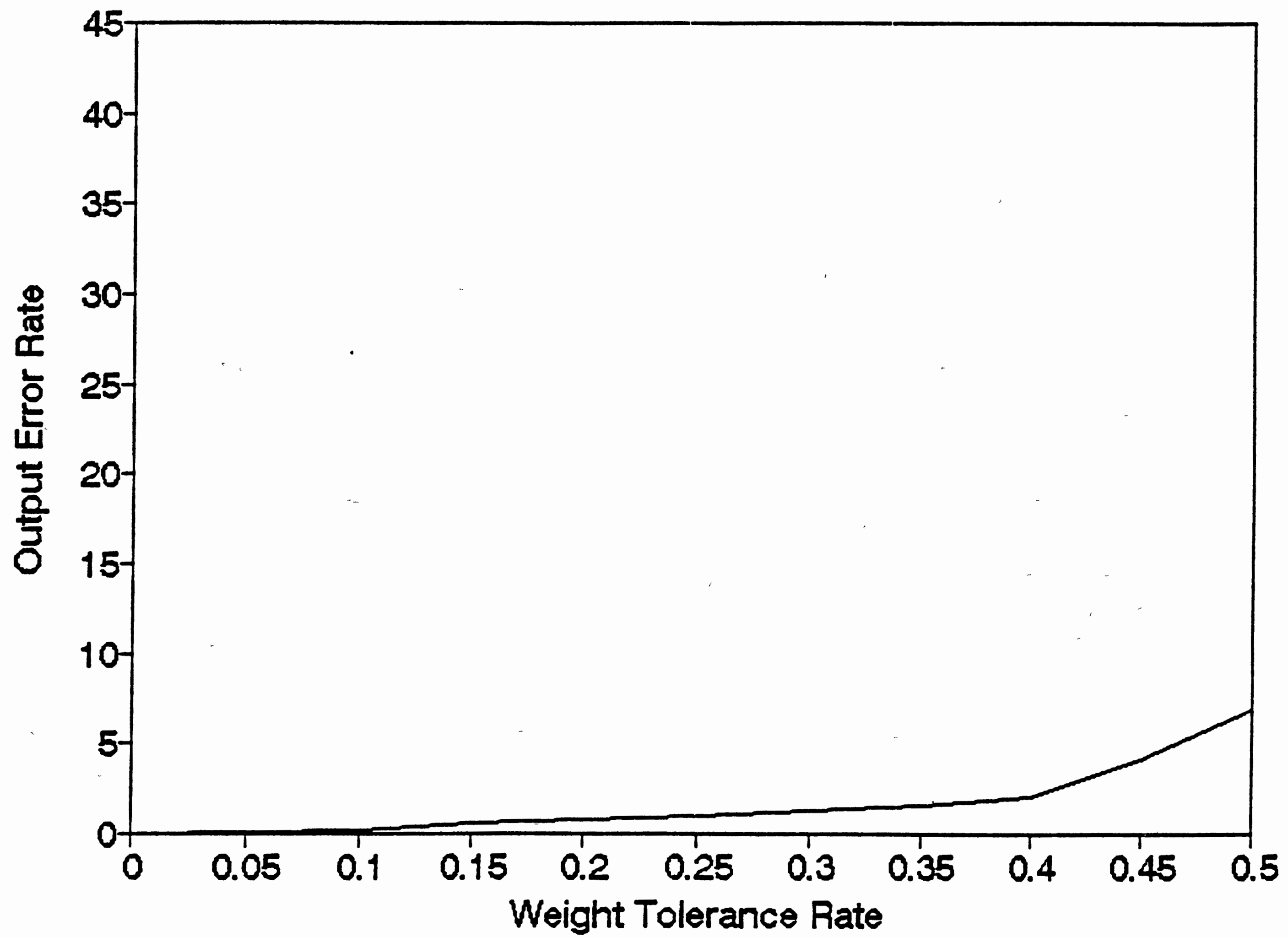

Figure 1. Minimum Effect on Generalization

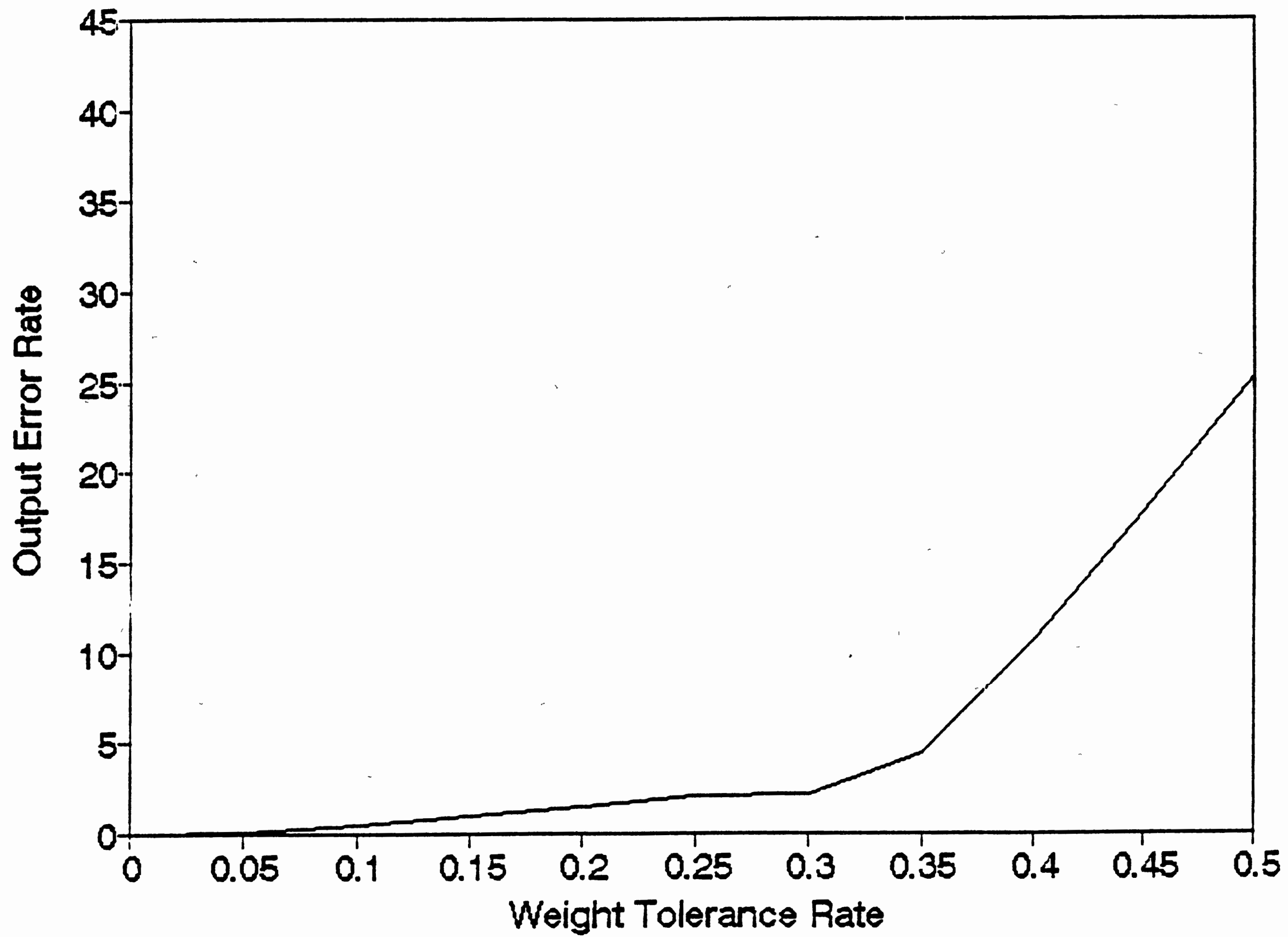

Figure 2. Maximum Effect on Generalization

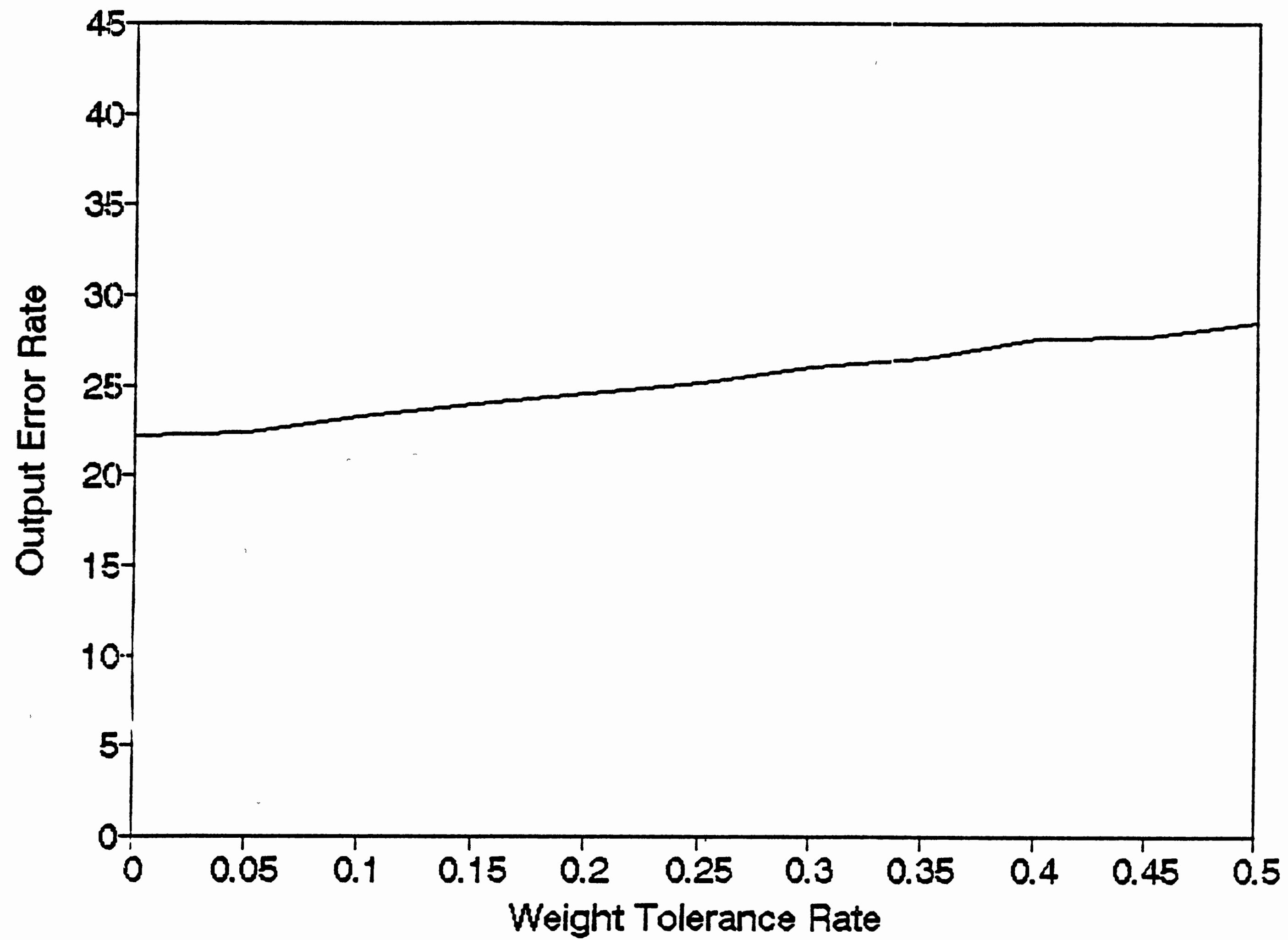

Figure 3. Classification Difficulty = 22.22 Percent

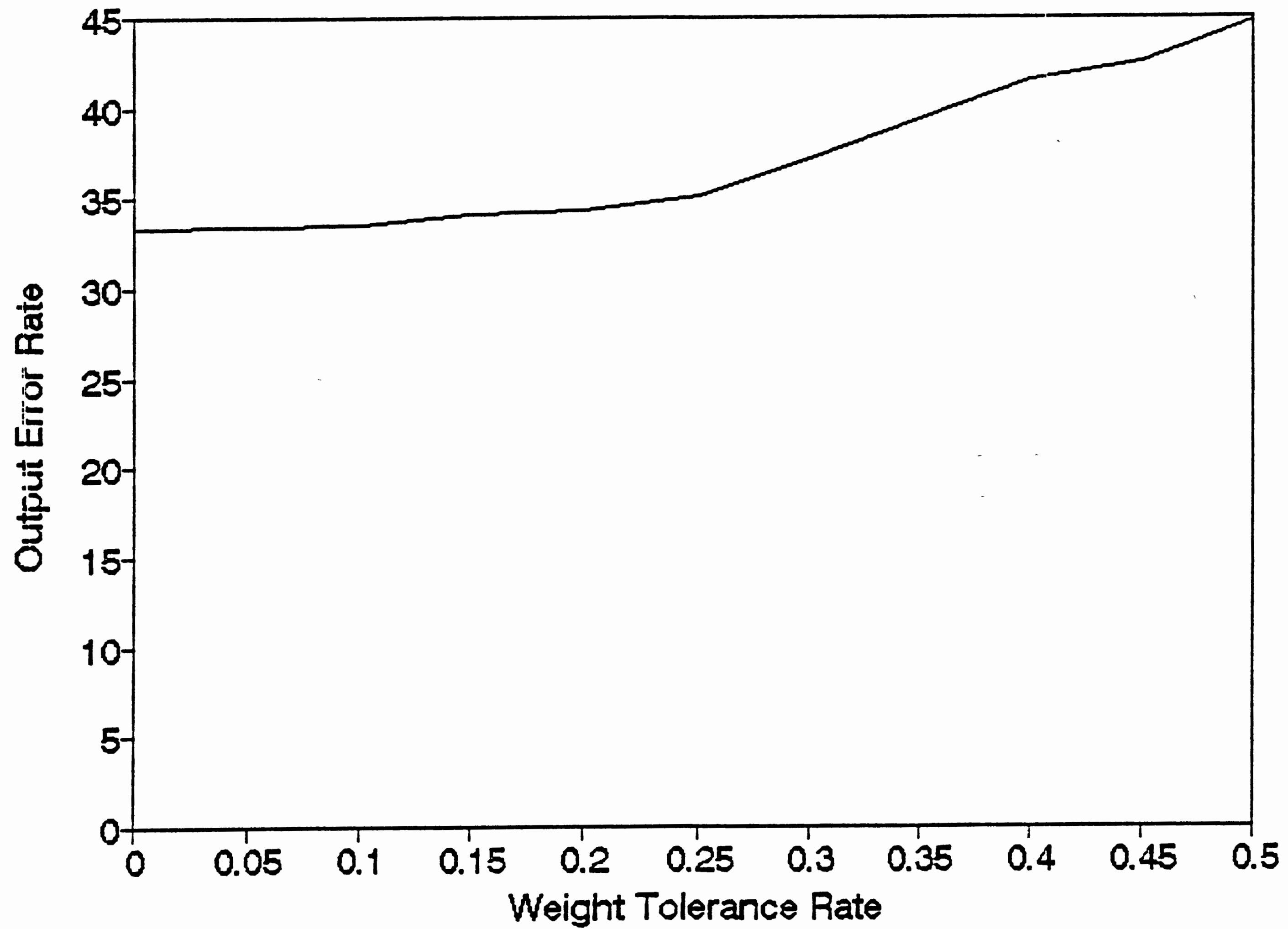

Figure 4. Classification Difficulty = 33.33 Percent

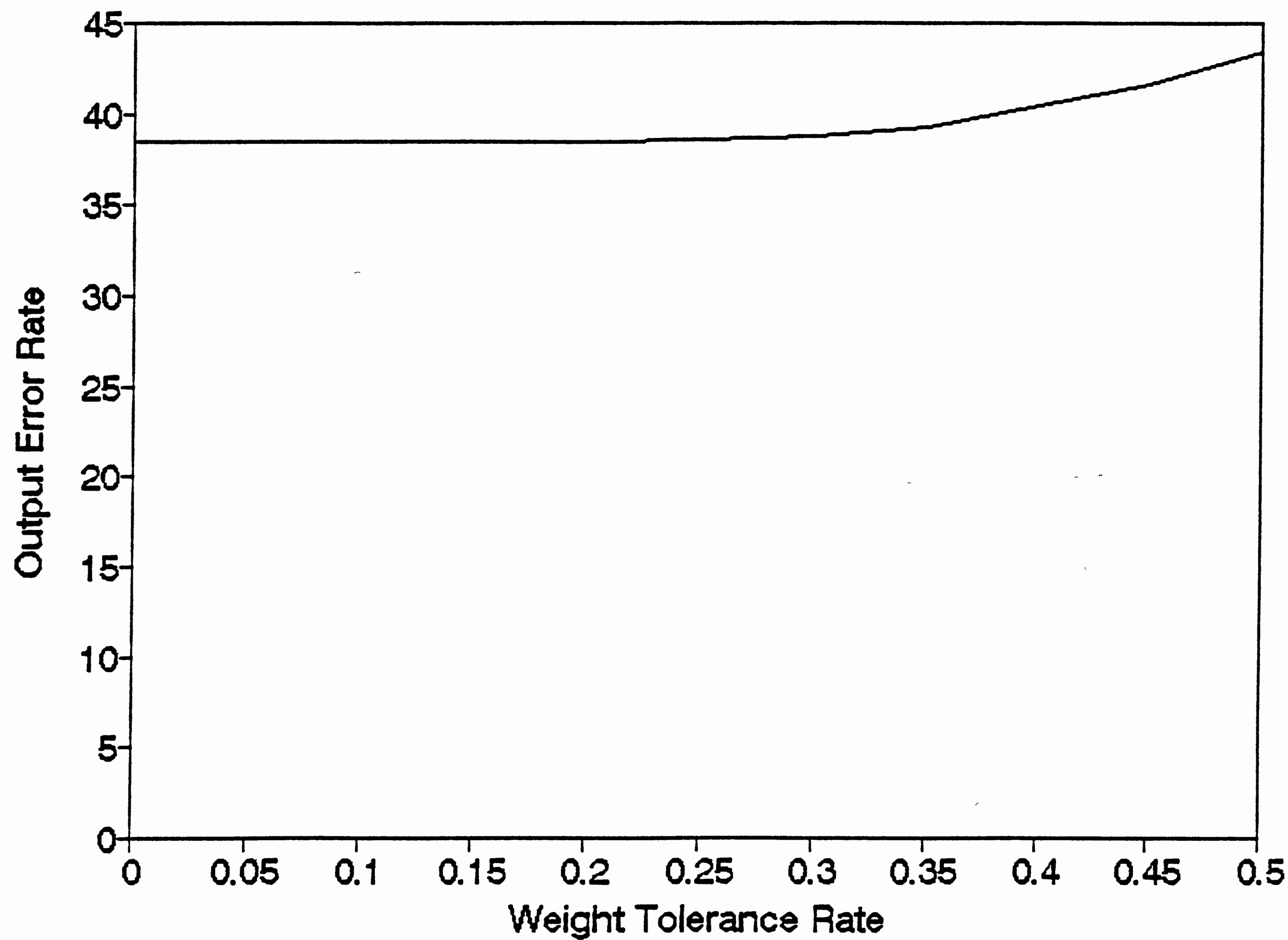

Figure 5. Classification Difficulty = 38.46 Percent

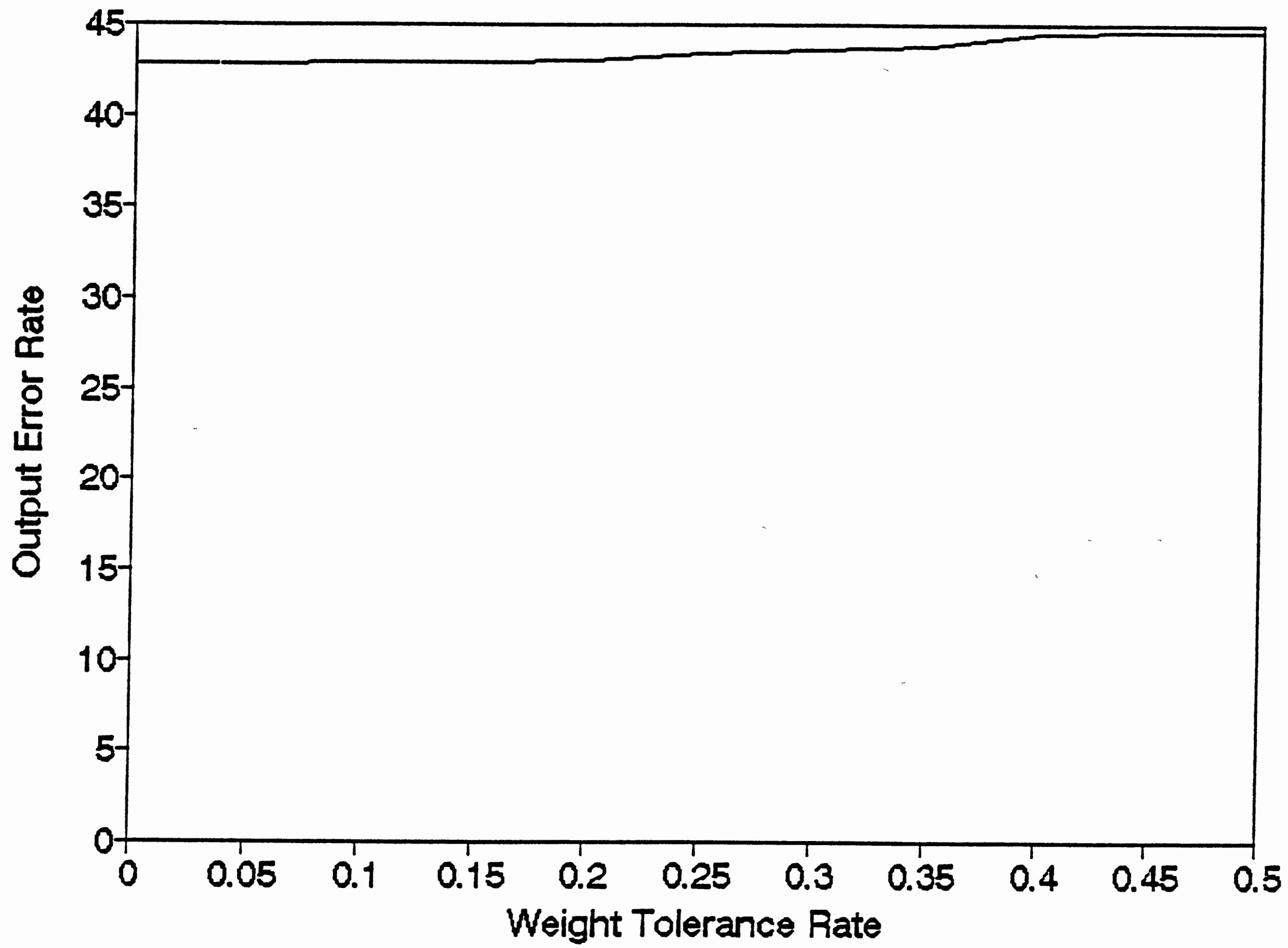

Figure 6. Classification Difficulty = 42.86 Percent

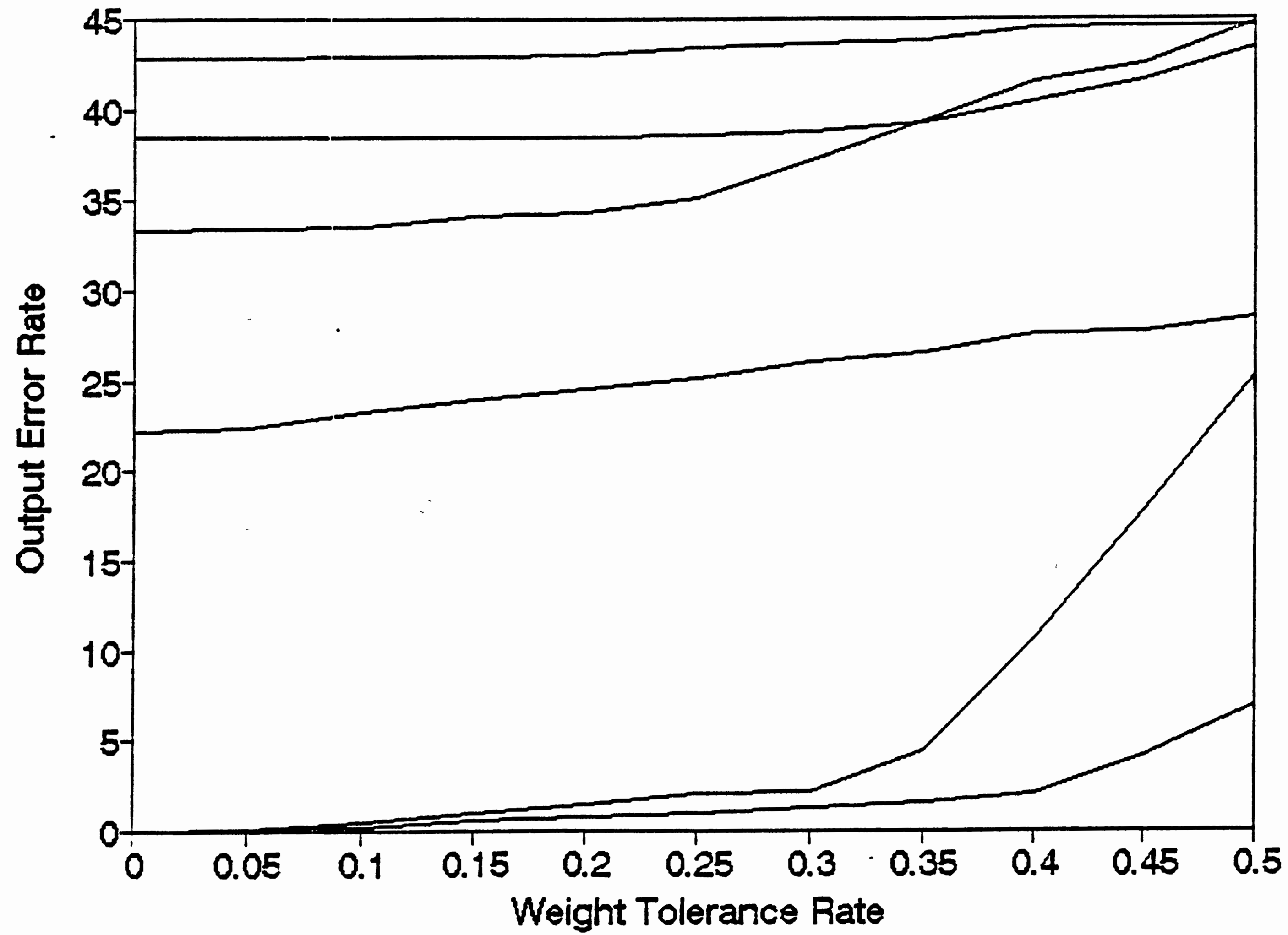

Figure 7. Various Classification Difficulties

VITA

Mohammad Ahmad Alrab

Candidate for the Degree of

Master of Science

Thesis: SENSITIVITY OF NEURAL NETWORKS TO RANDOM CHANGE WITH PERTURBED WEIGHTS AND BIASES

Major Field: Computer Science

Biographical:

Personal Data : Born in Taulkarm, West Bank in February 11, 1964, the son of Ahmad Alrab and Wesal Alrab.

Education : Graduated from JENIN High School in 1982, received a Bachelor Degree of Engineering in Electrical and Computer Engineering from Yarmouk University, Irbid, Jordan in 1988. Completed requirements for the Master of Science degree at Oklahoma State University in May, 1992.

Professional Experience: Electrical Engineer, Telecommunication Association, Irbid, Jordan.