TWO-DIMENSIONAL TARGET DETECTION UNDER NOISY CONDITIONS WITH NEURAL NETWORKS

by

TZU-CHIEN CHIN, B.S., M.S.

A THESIS

IN

ELECTRICAL ENGINEERING

Submitted to the Graduate Faculty
of Texas Tech University in
Partial Fulfillment of
the Requirements for
the Degree of

MASTER OF SCIENCE

IN

ELECTRICAL ENGINEERING

Approved

Accepted

May, 1993

ACKNOWLEDGEMENTS

I would like to send my sincere thanks and appreciations to Dr. Micheal Parten for his help and advice on this study. I would also like to thank Dr. William J.B. Oldham and Dr. Sunanda Mitra for their attendance on my thesis committee.

I would like to give my gratitude to my fellow graduate student, Mr. Mustafa Ulutas for his help. The period in Texas Tech University is merry and unforgettable. Finally, I like to thank my wife Pi-Sha who has accompanied with me to study and to live a life.

CONTENTS

ACKNOWLEDGEMENTS ii

ABSTRACT v

LIST OF TABLES vi

LIST OF FIGURES vii

CHAPTER

1. INTRODUCTION 1
   1.1 Introduction to Target Detection 1
   1.2 Introduction to Neural networks(NNWs) in Target Detection 2
   1.3 Simple Target Formation and the Proper Invariant Structure 4

2. THEORY OF SOME NEURAL NETWORKS 5
   2.1 Different Structural Neural networks(NNWs) 5
      2.1.1 Layer Neural Network 8
      2.1.2 High-Order Neural Network(HNNW) 15
      2.1.3 Invariant High-Order Neural Network(IHNNW) . 18
   2.2 Proof of Translation and Rotation Invariance 21
   2.3 Back-Propagation Learning 25

3. SIMULATION METHOD 34
   3.1 Signal Generation 36
   3.2 Noise Generation 40
   3.3 Exhaustive Pattern Training 45
   3.4 Window Output and Threshold 48
   3.5 Display 50

4. SIMULATION RESULTS AND DISCUSSION 53
   4.1 Capability of Learning 53
   4.2 Convergent Time of NNWs Trained with Different Levels of Noise 58
   4.3 6X6 Scene Outputs of NNWs Trained with Different Levels of Noise 59
   4.4 Experimental Noise Effect 62
   4.5 Size Effect and Incomplete Effect 63
   4.6 16X16 Scene Outputs of NNWs Trained with Different Levels of Noise 65
   4.7 Comparison of Network Output and Display 69
   4.8 The Effect of Different Types of Noise 75

5. CONCLUSION 79

REFERENCES 82

APPENDIX A: GTC.C Simulation Program 84
APPENDIX B: IHNW.C Simulation Program 96
APPENDIX C: PWN.C Simulation Program 124

ABSTRACT

This thesis developed a method to detect a two-dimensional target in a noisy environment by using a second-order neural network that has both translation and 90-degree rotation invariances.

The target had a 5X2 constant size, but could be anywhere in a 16X16 picture scene. In addition, the background contained an experimentally generated uncorrelated noise. Due to the sizes involved, the target loses its basic shape for any rotation other than 90 degrees. Therefore, only 90-degree rotation and translation invariance were employed. Back-propagation learning with the least mean square algorithm was used to train the NNW. The output function was a sigmoid.

Four variations of the second-order NNW were examined. Networks with product terms only, with product and square terms, with product and linear terms, and with product, square, and linear terms were used. Only the NNW with all three terms learned and tested well under noisy conditions for the rectangular target detection. The noise levels used were 0, 0.6, 0.7, and 0.8.

The training scene size was limited by the size of the target and chosen to be 6X6. A 6X6 moving window was used to cover the 16X16 scene. This approach reduces the size of the network and improves the convergence during training. However, the partial target problem makes the selection of threshold values for target identification critical.

The NNW, trained with experimental uncorrelated noise with noise levels of 0.6, 0.7, and 0.8, had above 80% accuracy when tested with the 6X6 and 16X16 samples of corresponding levels of noise.

The results have shown that the special neural network architecture can be used to detect a two-dimensional rectangular target in a large, noisy scene.

LIST OF TABLES

4.1 Training parameters of target and TC detection 55

4.2 Training results of NNW for TC patterns 56

4.3 Training results of different NNWs
at a noise level of 0 57

4.4 Training results of different NNWs
at a noise level of 0.6 57

4.5 Training results of different levels of noise 58

4.6 Statistical results of 6X6 scene inputs 60

4.7 Statistical results of 16X16 scene inputs 66

4.8 Training results of different types of noise 75

4.9 Outputs of NNWs trained by 3 types of noise 78

LIST OF FIGURES

2.1 Artificial neuron with activation function 5

2.2 Activation functions 7

2.3 Sigmoid function, with bias and shape modification .. 8

2.4 Single layer neural network 9

2.5 Multi-layer Neural Network 11

2.6 The role of hidden units in multi-layer NNWs 12

2.7 Second-order neural network 17

2.8 Illustration of the possibility of the learning procedure being trapped at non-optimum values 33

3.1 4X4 TC scenes without noise (a) a scene of T (b) a scene of C 37

3.2 6X6 scenes without noise (a) horizontal target scene (b) vertical target scene (c)right-slant target scene (d) left-slant target scene 39

3.3 Flow chart for GTC.C 42

3.4 16X16 scenes at a noise level of 0.6 (a) horizontal target scene (b) vertical target scene (c)right-slant target scene (d) left-slant target scene 43

3.5 Flow chart for NNW learning 47

3.6 Flow chart for output generation 50

3.7 Flow chart for graphical presentation of data 52

4.1 A 16X16 scene at a noise level of 0.8 (1) 71

4.2 A 16X16 scene at a noise level of 0.8 (2) 72

4.3 A 16X16 scene at a noise level of 0.8 (3) 73

4.4 A 16X16 scene at a noise level of 0.8 (4) 74

CHAPTER 1

INTRODUCTION

1.1 Introduction to Target Detection

Target detection, or pattern recognition, is a specific field of study within the field of image processing. The objective is to detect or recognize a target from one image or a sequence of images (visual image, infrared spectral bands, radio wave image, etc.). If the background signal is very similar to the target signal or the noise is too intense, target detection is a very difficult task. Statistical pattern recognition is a typical approach used in target detection. Statistical procedures are used to compute discriminant function parameters. However, this method requires a large representative training set, and is not very robust. In the discriminant function method of target detection, the objective is to find a mathematical function to determine class membership. Unfortunately, the discriminant function method is difficult to apply to target features. Other methods such as segmentation, feature extraction, and expert systems can only solve part of the target detection problem. These systems either produce high false alarm rates or do not provide a solution when the target varies slightly or the background varies in an unknown manner [1].

Target detection methods should be robust to the target signal and environmental variations. Neural networks (NNWs)

may provide an improved method for target detection. Although NNWs still cannot solve the problem of target detection satisfactorily, this technology does provide a number of tools which could form the basis for a potentially effective approach to target detection problems. Artificial neural networks exhibit characteristics that other methods lack. NNWs can learn from experience, generalize from previous examples to new ones, and abstract essential characteristics from inputs containing irrelevant data. The network can also modify its behavior in response to its environment. A neural network can be insensitive to minor variations of its inputs. This ability to see through noise and distortion is vital to target detection in a real world environment. For example, a system trained to recognize printed letters can still recognize letters which are corrupted up to 40% by noise [2]. A neural network followed by an associative rule can recognize 80% of a series of handwritten characters [3].

1.2. Introduction to Neural Networks(NNWs) in Target Detection

Although very few products using neural network technology are currently available, a lot of applications for neural networks have been proposed, such as pattern recognition, knowledge data bases for stochastic information, optimization computation, robotic control, decision making and many others [4]. A considerable research effort has been directed to the potential applications where human use is

inefficient and conventional computation is cumbersome or inadequate. Potential applications for neural networks include image processing, vision, speech recognition, fuzzy knowledge processing, data sensor fusion, and coordination and control of robot motion. Some very encouraging results using neural network technology exist commercially and other applications are in an experimental stage [3]. This thesis will concentrate on the application of neural networks to target detection.

A major problem in target detection is that real data, both with and without an object present, is quite noisy. One of the objectives of this study is to detect a target under low signal-to-noise ratio conditions.

In 2-dimensional target detection, traditional 3-layer and high-order NNW without invariances are nearly impossible to use because the number of weights explodes as the size of the scene increases. When employing some invariances in a high-order NNW, the number of weights and connections can be dramatically reduced, and the network can be practically applied. The 3 kinds of invariances are translation, rotation, and scale invariance. NNWs with any two of the three invariances will be of second-order. A NNW with all three invariances will have to be of third-order and will have significantly more weights and connections than a second-order network. This study concentrates on the second- order NNW with translation and 90-degree rotation invariance.

1.3 Simple Target Formation and the Proper Invariant Structure

The problem to be examined is a 5X2 pixel target in a 16X16 pixel scene [5]. The size of the target is constant, but the target can be in a horizontal, vertical, right slant, or left slant position. The scene includes noise that is modeled by an experimental noise [6].

Because the size of the target is constant, only translation and rotation invariances are employed. However, due to pixel resolution limit, the distance of any two pixels may change after rotating the line, which is formed by connecting the two pixels, with a small angle. Only the distance which rotates a multiple of 90 degrees will have the same value of distance. Therefore, only 90-degree rotation invariance and translation invariance are employed. The proof of translation and 90-degree rotation invariances is in section 2.2.

CHAPTER 2

THEORY OF SOME NEURAL NETWORKS

2.1 Different Structural Neural Networks(NNWs)

There are many kinds of neural networks. Most neural networks are composed of the same basic type of artificial neurons.

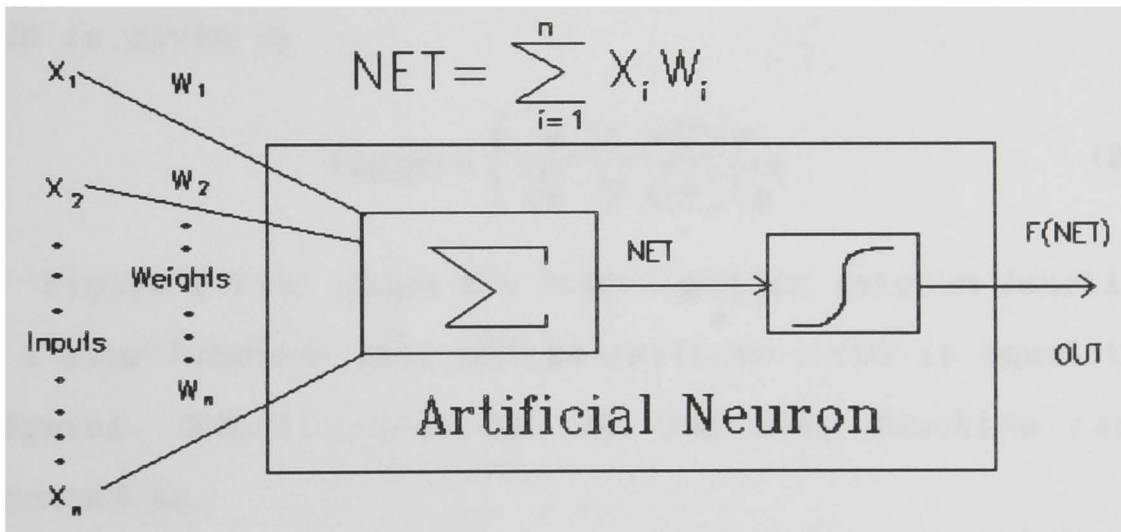


Figure 2.1 Artificial neuron with activation function

The structure of an artificial neuron is shown in Figure 2.1. In this model, a set of inputs x_1, x_2, ... , x_n is applied to the neuron. Each input may represent the output of another neuron. Each input is multiplied by a corresponding weight, w_1, w_2, ..., w_n, and all the weighted inputs are then summed. The summation block adds all the weighted inputs algebraically

and produces the output NET. This can be expressed in vector notation as

$$NET = \vec{X} \cdot \vec{W} \ . \tag{2.1}$$

The NET signal is further processed by an activation function F (also called a threshold function or a squashing function) to produce the neuron's output signal, OUT.

$$OUT = F(NET) \ . \tag{2.2}$$

The activation function, F, maps NET to a specified range. Three commonly used functions are the ramp, step and sigmoid functions [7]. Figure 2.2(a) shows a ramp function which is given by

$$F(NET) = \begin{cases} \alpha \ if \ NET \geq \alpha \\ NET \ if \ |NET| < \alpha \\ -\alpha \ if \ NET \leq -\alpha \end{cases} . \tag{2.3}$$

Figure 2.2(b) shows the step function (signum function). For a step function when NET is positive , OUT is equal to α, otherwise, OUT is equal to $-\alpha$. The step function can be expressed as

$$F(NET) = OUT = \begin{cases} \alpha \ if \ NET > 0 \\ -\alpha \ otherwise \end{cases} . \tag{2.4}$$

The sigmoid (S-shape) function is the most pervasive activation function. It provides a graded and nonlinear response. A common sigmoid function is the logistic function shown in Equation 2.5. The saturation levels are 0 and 1.

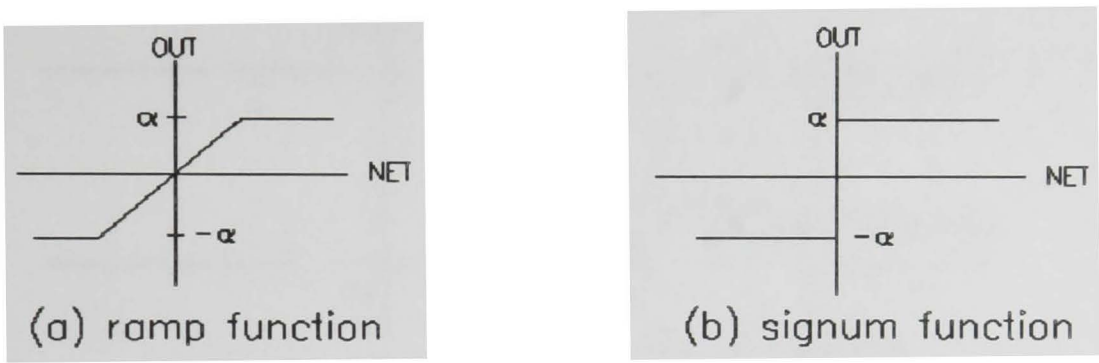


Figure 2.2 Activation functions

$$OUT = \frac{1}{1 + e^{-NET}} \quad . \tag{2.5}$$

Another sigmoid function shown in equation 2.6 is the hyperbolic tangent. Its saturation levels are -1 and 1.

$$OUT = \tanh(NET) \quad . \tag{2.6}$$

Another variation of the sigmoid activation function is shown in Figure 2.3 and in mathematical form in Equation 2.7.

$$OUT_j = \frac{1}{1 + e^{(-NET_j + \theta_j)/\theta_0}} \quad , \tag{2.7}$$

where θ_j serves as a bias.

The effect of a θ_j is to shift the activation function along the horizontal axis. The effect of θ_0 is to modify the

shape of sigmoid function. A low value of θ_0 tends to make the sigmoid similar to the step function, whereas a high value of θ_0 results in a more gently varying function.

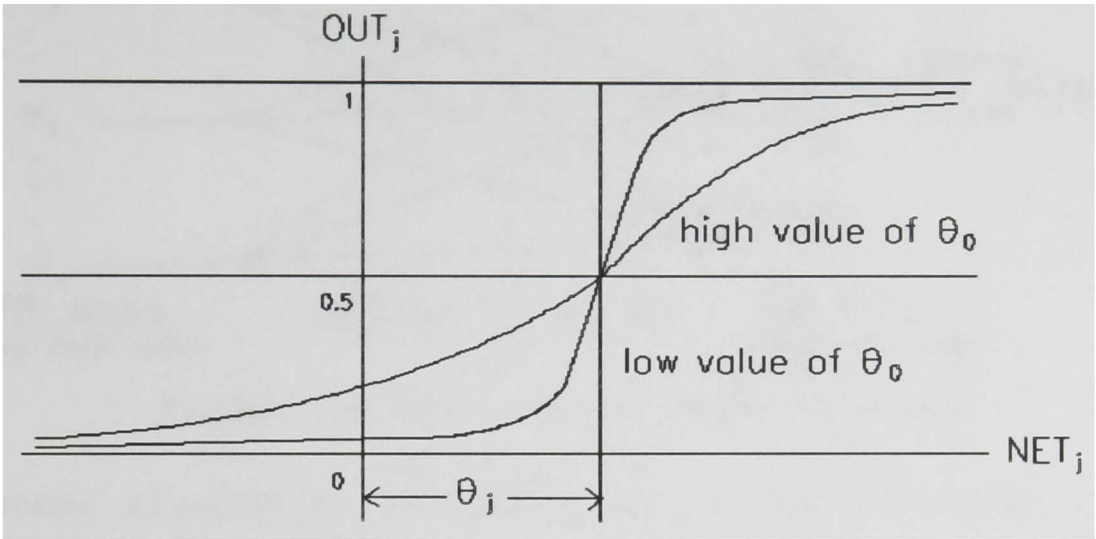


Figure 2.3 Sigmoid function, with bias and shape modification

2.1.1 Layer Neural Network

A single-layer perceptron is also known as a linear discriminate or single-layer neural network and can be represented schematically in the form of an array of multipliers and summing junctions, as shown in Figure 2.4.

This network has the ability to recognize simple patterns. The connection weights and the bias (threshold) in a perceptron can be fixed or adapted by using a number of

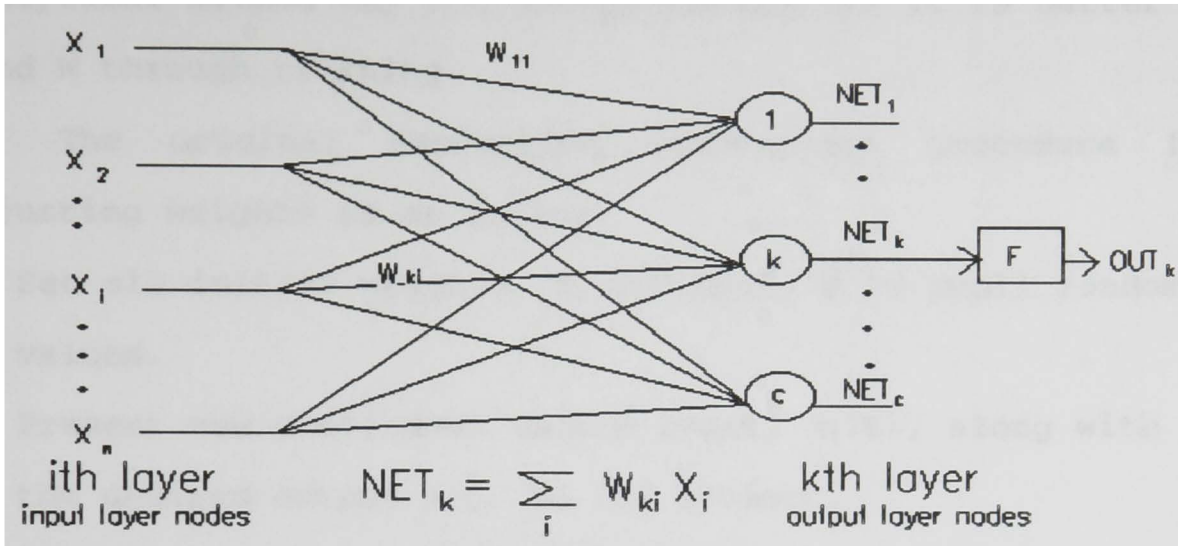


Figure 2.4 Single layer neural network

different algorithms. The objective of the algorithm is to find W in Equation 2.1.

The weights, W, can be found by the following method[8]. Multiplying both side of Equation 2.1 by the transpose of matrix X gives equations below :

$$\vec{X}^t\vec{X}\cdot\vec{W} = \vec{X}^t\cdot\vec{b} \tag{2.8}$$

$$\vec{W} = (\vec{X}^t\vec{X})^{-1}\,\vec{X}^t\cdot\vec{b} \tag{2.9}$$

$$\vec{W} = X^*\,\vec{b}\ , \tag{2.10}$$

where X^* is a pseudo inverse and can be obtained as the limiting value of the process in Equation 2.11.

In practice, however, nonsensical values are obtained if

$$X^* = \lim_{\epsilon \to 0} (\vec{X}^t \vec{X} + \epsilon I)^{-1} \vec{X}^t \qquad (2.11)$$

the determinant of X^t X is extremely small. So analytically, it is not easy to find W. Also practically the above analytical method may not be applicable, so it is better to find W through training.

The original perceptron convergence procedure for adjusting weights is as follow:

1. Set all initial weights, W_i and bias, θ to small random values.
2. Present new continuous valued input, $x_i(t)$, along with the desired output b(t) to the network.
3. Calculate actual network output from

$$OUT(t) = F\left(\sum_{i=0}^{N-1} w_i(t)\, x_i(t) - \theta\right) \qquad (2.12)$$

   where N is the number of inputs to the perceptron, i is the i^{th} input and t is the t^{th} iteration.
4. Adapt the weights by

$$w_i(t+1) = w_i(t) + \eta\,[b_i(t) - OUT_i(t)] \cdot x_i(t) \qquad (2.13)$$

   where η is a positive gain term ranging from 0 to 1 and b(t) is the desired correct output for the current input.

Equation 2.13 can be written as

$$\Delta w_i = \eta \delta_i\, x_i \qquad (2.14)$$

where δ_i is equal to $(b_i - OUT_i)$, the difference between the desired output, b_i and the actual output OUT_i produced by the

network. The algorithm above is also known as the delta training rule. The shortcomings of the single-layer perceptron is that it is incapable of solving a non-linear separable problem, such as the EXCLUSIVE-OR problem [4].

Multi-layer neural networks, however, can solve non-linear problems. The architecture of a multi-layer neural network is shown in Figure 2.5.

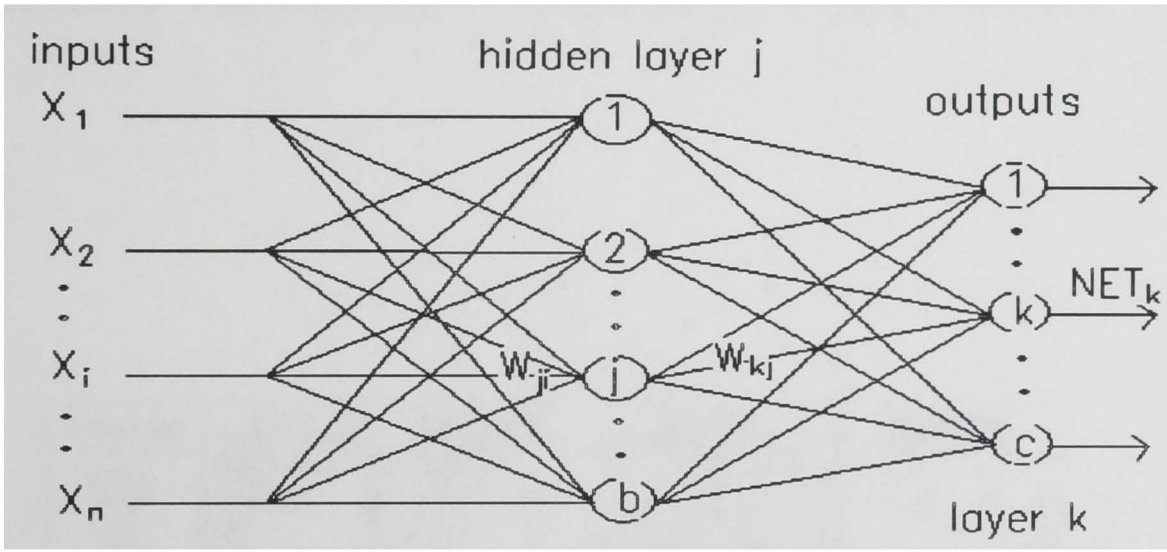


Figure 2.5 Multi-layer neural network

Multi-layer perceptrons are feed forward nets with one or more hidden layers of nodes (a layer of nodes between the input and output nodes). The capabilities of multi-layer perceptrons stem from the nonlinear characteristics used within nodes. If the nodes were linear elements, then a multi-layer perceptron could be replaced by a single-layer

perceptron.

A two-layer perceptron with linear activation function can be represented mathematically as

$$\vec{X}\vec{W}_1\vec{W}_2 = \vec{b} \ , \qquad (2.15)$$

which can be written as

$$\vec{X}\vec{W} = \vec{b} \qquad (2.16)$$

where b is the desired output vector. Thus a two-layer perceptron can be replaced by a single-layer perceptron if the perceptive elements are linear. The capability of perceptrons with one-, two-, and three-layers that use non-linear activation functions is illustrated in Figure 2.6 [9].

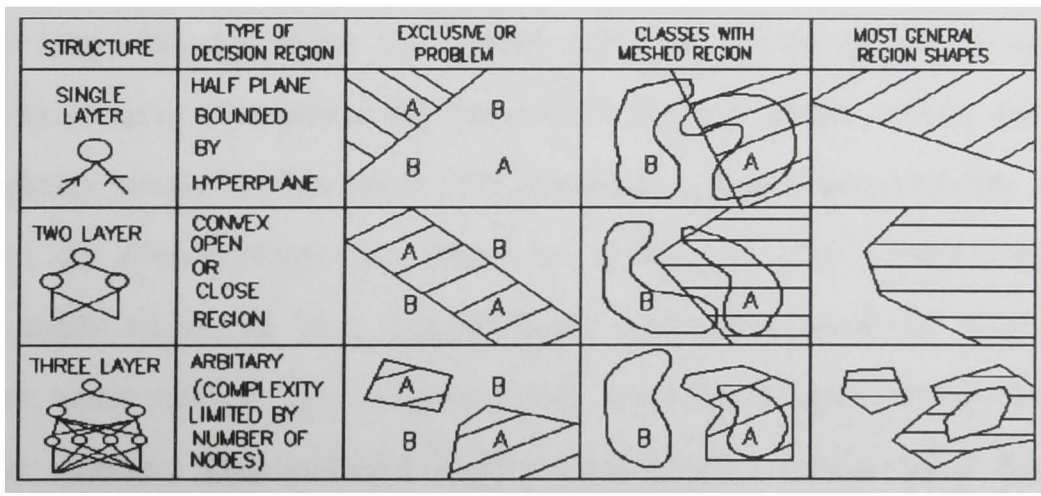


Figure 2.6 The role of hidden units in multi-layer NNWs

A single-layer perceptron forms half-plane decision

regions. A two-layer perceptron can form a, possibly unbounded, convex region in the space spanned by the inputs. Each node in the first layer behaves like a single-layer perceptron, each of which divides the input space into half-plane regions. Thus convex regions are formed from interconnections of the half-plane regions formed by each node in the first layer. So the convex regions have, at most, as many sides as the number of nodes in the first-layer. The above analysis provides some insight into the problem of selecting the number of nodes to use in a two-layer perceptron. The number of nodes must be large enough to form a decision region that is as complex as is required by a given problem. But for a two-layer perceptron, no number of nodes can separate the meshed class region as shown in Figure 2.6 for a three-layer perceptron.

A three-layer perceptron can form arbitrarily complex decision regions and can separate the meshed classes also. This can be proven by construction. The proof depends on partitioning the desired decision region into small hypercubes (squares when there are two inputs). Each hypercube requires 2n (n is the number of inputs) nodes in the first-layer, one for each side of the hypercube, and one node in the second-layer that takes the logical AND of the output from the first-layer nodes. The second-layer nodes will fire only for inputs within each hypercube. Hypercubes are assigned to the proper decision regions by connecting the output of each second-layer

node only to the output node corresponding to the decision region that node's hypercube is in. The third-layer nodes (output nodes) then perform a logical OR operation on the inputs from the second-layer nodes. If all inputs lie within a decision region, at least one node in the second-layer fires which triggers the firing of the output node. On the other hand, if any inputs lie outside the decision region, the second-layer nodes will not fire, nor will the output fire. This construction procedure can be generalized for arbitrarily shaped convex regions instead of small hypercubes and can show that the network is capable of generating the disconnected and non-convex regions. This analysis demonstrates that no more than three layers are required in perceptron-like feed forward nets because a three-layer net can generate arbitrarily complex decision regions. It also provides some insight into the problem of selecting the number of nodes to use in three-layer perceptrons. The number of nodes in the second-layer must be grater than one when decision regions are disconnected or meshed and cannot be formed from one convex area. The number of second-layer nodes required in the worst case is equal to the disconnected regions in the input distributions. The number of nodes in the first-layer must be sufficient to provide three or more edges for each convex area generated by every second-layer node. The number of nodes in the first-layer should be at least 3 times as many nodes as those in the second-layer. Similar arguments can be applied to multi-output

three-layer perceptron.

2.1.2 High-Order Neural Network(HNNW)

Early research has shown that single-layer first-order neural networks can only produce linear discrimination boundaries between pattern classes. Also it has been shown that non-linear discrimination surfaces could be derived either by multi-layer first-order or by single-layer high-order neural networks [10]. The term "first-order" indicates that the input values are directly applied to the inputs of the neural network. The term "high-order" refers to the creation of non-linear combination of input values (e.g., products of any two original input values, etc.), prior to application at the network input nodes. Most real word target detection problems require non-linear discrimination boundary surfaces. Therefore, single-layer high-order or multi-layer first-order NNW are better than single-layer first-order NNW.

In solving contiguity problems, like the two-three clump problem (recognizing the pattern of 2 to 3 groups of 1s in a string of 0s), a multi-layer NNW does not perform as well as a high-order neural network [11]. For this problem, the multi-layer NNW requires thousands of presentations in order to learn the training set and the accuracy is only slightly greater than 50%. Also in solving the TC problem (recognizing the translational and rotational version of the letters T and C), a multi-layer perceptron requires over 5000 presentations

of an exhaustive training set. A high-order neural network can solve the above problems with much higher accuracy (100% accuracy for a training set containing 1/10 of all possible patterns in the TC problem) and a shorter training time [11]. HNNW can be made to incorporate translation, scale and/or rotation invariance by using the different combinations of the input values. With these invariances, not only the performance for special problems is improved, but also the network architecture is simplified.

The reason a multi-layer NNW can solve the non-linear separable problem is that the hidden layers produce non-linearities. These non-linearities also can be produced by nodes in HNNW. The output of a high-order unit can be represented as:

$$\begin{aligned} OUT_j &= F(T_0 + T_1 + T_2 + \ldots\ldots) \\ &= F(\theta_0 + \sum_{i=1}^{n} w_{ji} x_i + \sum_{k=1}^{n} \sum_{l=1}^{k} w_{jkl} x_k x_l + \ldots\ldots) \end{aligned} \tag{2.17}$$

where the subscript j indicates the j^{th} output node, i indicates the i^{th} input, k and l indicate the different combination of inputs in the product terms. The high-order weights capture the high-order correlations.

The zeroth-order term T_0 is an adjustable threshold or bias, denoted by θ_0. T_1 is the first-order term (or linear term) which is the weighted sum of inputs. T_2 is the second-order term which is the linear weighted sum over the second-order products of inputs. A unit which includes terms up to n,

including degree k, will be called a k^{th}-order unit. It has been shown that a second-order neural network can incorporate translational and scale invariance in the network architecture while a third-order neural network can include rotational invariance as well [12]. The architecture of a second-order neural network is shown in Figure 2.7. The j^{th} node output is represented by:

$$OUT_j = F(\theta_j + \sum_{i=1}^{n} w_{ji} x_i + \sum_{k=1}^{n} \sum_{l=1}^{k} w_{jkl} x_k x_l) \quad . \tag{2.18}$$

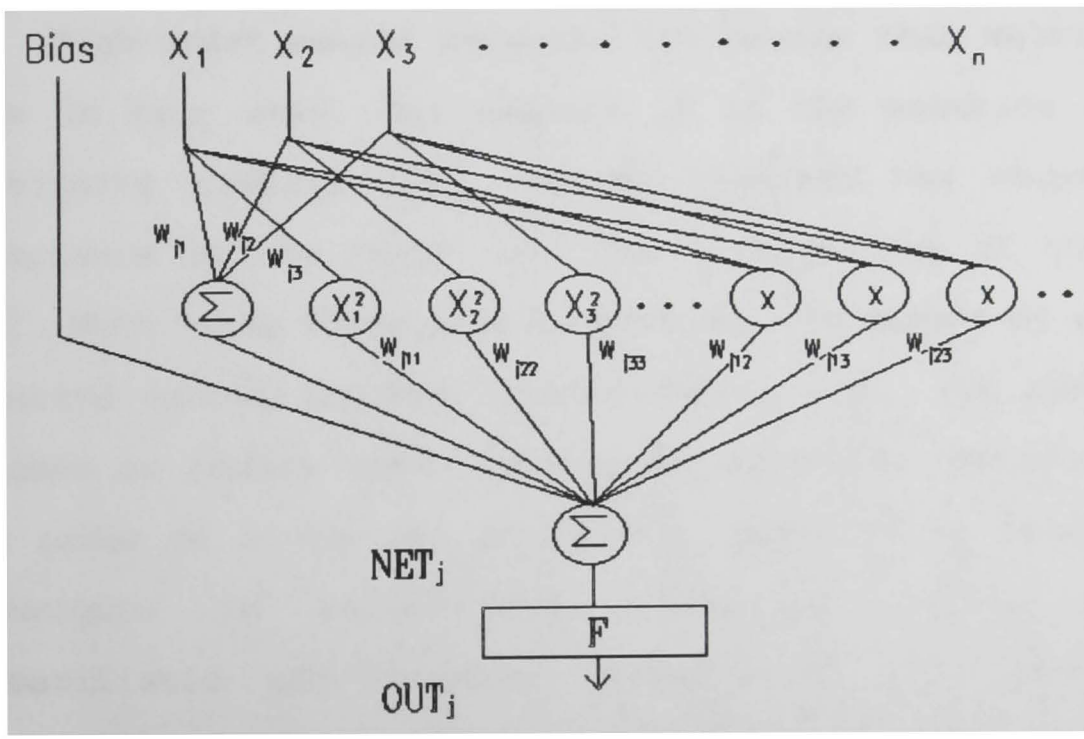


Figure 2.7 Second order neural network

As seen in Equation 2.17 and Figure 2.7, a network with

n inputs and one output, using r^{th}-order terms, requires C^n_r number of weights (interconnections). C^n_r is of the order of n^r. So for a high-order NNW, the number of interconnections can be extremely large and can easily become impracticable as the input size grows. For example, for a 6X6 pixel input field, a 3rd-order network requires C^{36}_3= 7,140 product weights. For a 16X16 pixel input field, a 3rd-order network requires C^{256}_3= 2,763,520 product weights while a 128x128 pixel field requires $7.3x10^{11}$ interconnections. This combinational explosion is the greatest drawback of HNNW.

2.1.3 Invariant High-Order Neural Network(IHNNW)

High-order neural networks are better than multi-layer NNWs in many ways. One example is in the solution of the contiguity problem [13]. Recent research has shown that invariance can be built into the architecture of the HNNW [12]. With these invariance properties, the number of weights required can be reduced tremendously , e.g., the number of weights in second-order translation invariant network is in the order of n for the input size equal to n. Some other techniques in connectivity, such as local, sample, probabilistic and regional connectivity [14] have been proposed to further reduce the number of weights required in the networks. Solving the TC problem by regional connectivity, can reduce the number of weights by a factor of 200,000 [14].

The theory of one dimensional translation invariant neural network comes from C. Lee Giles and Tom Maxwell [11]. The order of this kind IHNNW is 2. For the product term, i.e., X_kX_1, the weights are determined by the rule that the products of any two input pixel values will have the same weight if the distance between the two product pixels is the same. This can be represented by

$$\begin{aligned} OUT &= F(NET) \\ &= F(\theta_0 + T_1 + T_2) \\ &= F(\theta_0 + \sum_{i=1}^{n} w_i x_i + \sum_{d=1}^{n} w_d \sum_{k=1}^{n} x_k x_{k+d}) \end{aligned} \tag{2.19}$$

where θ_0 is the bias term. T_1 is the linear term and w_i (i=1...n) have the same value. But while considering the learning convergency especially when there is noise, w_i (i=1...n) needs to be different [15]. T_2 is the product term and square term. d is the distance between two pixels X_k and X_{k+d}. If d=0, it means a square term. Now the square term is inclusive of T_2.

The original theory of two-dimensional translation and scale invariant neural networks is developed by C. L. Giles, R. D. Griffin, and T. Maxwell [16]. The network is second-order IHNNWs. For the product term, the networks take the invariance of the tangent of any two pixels. In other words, if the tangent of $p_1(x_1,y_1)$ and $p_2(x_2,y_2)$ is the same as that of $p_3(x_3,y_3)$ and $p_4(x_4,y_4)$, the product value $X_{p1}X_{p2}$ and $X_{p3}X_{p4}$ will have the same weight. The OUT, NET and T_1 are the same as above. Only T_2 is different and, in this case, is generally

$$T_2 = \sum_{i(\tan(t))} W_{i(\tan(t))} \sum_{pk,pl} X_{pk}X_{pl}$$
$$\tan(t) = \frac{y_l - y_k}{x_l - x_k} \quad for\ all\ 1 \le l,k \le n \tag{2.20}$$

Here, tan(t) is the tangent of pixels p_k and p_l , and i(tan(t)) denotes the i^{th} tangent.

The theory of two-dimensional rotation and scale invariant neural networks was also developed by Giles, Griffin, and Maxwell [16]. The network is 2nd-order IHNNW. The networks use polar coordinates. Let r1 and t1 be the radius and angle of p1, i.e., p1(r1,t1). The invariance is the ratio of radii and the difference of angles, e.g., considering four points, p1(r1,t1), p2(r2,t2), p3(r3,t3), and p4(r4,t4). If r1/r2 = r3/r4 and t1-t2 = t3-t4, then $X_{p1}X_{p2}$ and $X_{p3}X_{p4}$ will have the same weight. So T_2 is

$$T_2 = \sum_{i(dr),j(dt)} W_{i(dr),j(dt)} \sum_{pk,pl} X_{pk}X_{pl}$$
$$dr = \frac{r_k}{r_l},\ dt = t_k - t_l,\ for\ all\ 1 \le k,l \le n\ . \tag{2.21}$$

Here, (r_k, t_k) and (r_l, t_l) are the polar coordinates of pixels p_k and p_l. dr is the ratio of the radii of pixels p_k and p_l and dt is the difference of pixels p_k and p_l. i(dr) denotes the i^{th} dr and j(dt) denotes the j^{th} dt.

A third-order NNW can provide two-dimensional translation, scale and rotation invariances. The detail theory comes from Lilly Spirkovska and Max B. Reid [14]. The invariance comes from the three inner angles of a triangle formed by three pixels. The weight W_{ijkl} of pixels p_j, p_k and p_l

can be expressed as

$$w_i = w_{ijkl} = w_{i\alpha\beta\gamma} = w_{i\beta\gamma\alpha} = w_{i\gamma\beta\alpha} \qquad (2.22)$$

where α, β, γ are the three angles in the triangle formed by p_j, p_k and p_l. T_3 is given by

$$T_3 = \sum_i w_i \sum_{j,k,l} x_j x_k x_l \quad for\ all\ 1 \le j,k,l \le n\ . \qquad (2.23)$$

Although network does not have T_1 and T_2, the number of weights is large. As a result, connectivity strategies are used to further reduce the number of weights.

2.2 Proof of Translation and Rotation Invariance

The two dimensional target detection problem of interest here requires translation and rotation invariance. In this case, there is only one simple small target appearing in the scene that is corrupted by noise. The shape of the target will be somewhat blurred by the noise but the basic shape is unchanged. Since the basic size of the target is constant, scale invariance is not necessary. Therefore, the network should be a second-order IHNNW with translation and rotation invariance.

To derive the translation and rotation invariances, use the zeroth-order term T_0, the first-order term T_1, and the second-order term T_2 in Equation (2.17) of HNNW. Let the weights of T_1 be $w^1[p]$ and the weights of T_2 be $w^2[p]$. p denotes the pixels.

For 2 dimensional translation invariance:

$$\begin{aligned} w^1[p;x_1,y_1] &= w^1[p] \\ w^2[p;x_1,y_1;x_2,y_2] &= w^2[p;x_1-x_2;y_1-y_2] \end{aligned} \tag{2.24}$$

Equation 2.24 means that if both the x axis distance and y axis distance of $p_1(x_1,y_1)$ and $p_2(x_2,y_2)$ are the same as those of $p_3(x_3,y_3)$ and $p_4(x_4,y_4)$, i.e. $|x_1-x_2|=|x_3-x_4|$ and $|y_1-y_2|=|y_3-y_4|$, then the product of values of p_1 and p_2 should have the same weight as the product of p_3 and p_4.

For 2-dimensional rotational invariance, with the center of the target at the origin (0,0), when the target rotates an angle t degrees with fixed axes from $p_1(x_1,y_1)$ to $p_1(x_{1'},y_{1'})$, the weights need to be the same. Thus,

$$\begin{aligned} x_{1'} &= x_1\cos(t) - y_1\sin(t) \\ y_{1'} &= x_1\sin(t) + y_1\cos(t) \\ w^1[p;x_1,y_1] &= w^1[p;x_{1'},y_{1'}] \ . \end{aligned} \tag{2.25}$$

With the same condition for $p_2(x_2,y_2)$,

$$\begin{aligned} x_{2'} &= x_2\cos(t) - y_2\sin(t) \\ y_{2'} &= x_2\sin(t) + y_2\cos(t) \\ w^1[p;x_2,y_2] &= w^1[p;x_{2'},y_{2'}] \\ w^2[p;x_1,y_1;x_2,y_2] &= w^2[p;x_{1'},y_{1'};x_{2'},y_{2'}] \ . \end{aligned} \tag{2.26}$$

Because the distance squared of the left hand side is the same as that of the right hand side for both cases above, the distance squared is the invariant quantity. The reason for choosing distance squared other than distance itself is because in the integer cartesian coordinates all the distances squared are integers but some of the distances are not. This

will simplify the simulation program. The weights of the second-order term can be written as

$$w^2[p;x_1,y_1;x_2,y_2] = w^2[p;(x_1-x_2)^2+(y_1-y_2)^2] \quad . \qquad (2.27)$$

The invariance of the distance squared can be shown by translating the pixels p_1, p_2 an amount x_0 in the x axis and y_0 in the y axis and then rotating the two pixels by an angle t. The new locations are given by

$$\begin{aligned} x_{1'} &= (x_1-x_0)\cos(t) - (y_1-y_0)\sin(t) \\ y_{1'} &= (x_1-x_0)\sin(t) + (y_1-y_0)\cos(t) \\ x_{2'} &= (x_2-x_0)\cos(t) - (y_2-y_0)\sin(t) \\ y_{2'} &= (x_2-x_0)\sin(t) + (y_2-y_0)\cos(t) \quad . \end{aligned} \qquad (2.28)$$

The distance is given by

$$(x_1-x_2)^2+(y_1-y_2)^2 = (x_{1'}-x_{2'})^2+(y_{1'}-y_{2'})^2 \quad . \qquad (2.29)$$

So, the invariance of distance squared holds and the second-order terms of equal distance should have the same weights as

$$w^2[p;(x_1-x_2)^2+(y_1-y_2)^2] = w^2[p;(x_{1'}-x_{2'})^2+(y_{1'}-y_{2'})^2] \quad . (2.30)$$

Also by first rotating the pixels p_1 and p_2 an angle t and then translating the pixels x_0 in the x axis and y_0 in the y axis, gives

$$\begin{aligned} x_{1''} &= x_1\cos(t) - y_1\sin(t) - x_0 \\ y_{1''} &= x_1\sin(t) + y_1\cos(t) - y_0 \\ x_{2''} &= x_2\cos(t) - y_2\sin(t) - x_0 \\ y_{2''} &= x_2\sin(t) + y_2\cos(t) - y_0 \quad . \end{aligned} \qquad (2.31)$$

The corresponding distance is given by

$$8(x_1-x_2)^2+(y_1-y_2)^2 = (x_{1''}-x_{2''})^2+(y_{1''}-y_{2''})^2 \quad . \qquad (2.32)$$

Again the invariance of distance squared still holds and the

equality of weights is given by

$$w^2[p;(x_1-x_2)^2+(y_1-y_2)^2]=w^2[p;(x_{1''}-x_{2''})^2+(y_{1''}-y_{2''})^2]\ . \quad (2.33)$$

So, the invariant quantity "distance squared" can be used in the translation and rotation IHNNW. However, the unit distance for different angles on a screen with discrete pixels are different because of the limitation of resolution of the image. Therefore, only the 90-degree rotation and translation invariance will be employed.

$$\begin{aligned} T_2 &= \sum_{i(ds)} w_{i(ds)} \sum_{k=1,l=1}^{k=n,l=n} x_k(x_{k1},y_{k1})\, x_l(x_{l2},y_{l2}) \\ ds &= (x_{k1}-x_{l2})^2+(y_{k1}-y_{l2})^2 \\ &\textit{for all } k,l \quad 1\le k,l\le n \\ &\textit{with the same distance squared, } ds\ . \end{aligned} \quad (2.34)$$

Here, ds denotes the distance squared of pixels x_k and x_l and i(ds) denotes the i^{th} ds.

If k=l then ds=0, which means a square term, otherwise it will be a product term.

$$\begin{aligned} T_2 &= T_s + T_p \\ T_s &= w_0 \sum_{i=1}^{n} x_i x_i \ : \ ds=0,\ \textit{square term} \\ T_p &= \sum_{i(ds\neq 0)} w_{i(ds)} \sum_{k=1,l=1,k\neq l}^{k=n,l=n} x_k(x_{k1},y_{k1})\, x_l(x_{l2},y_{l2}) \ . \\ &\quad : ds\neq 0,\ \textit{product term} \end{aligned} \quad (2.35)$$

Although the invariances come from the second-order term, T_2 which contains product term and square term, the linear term with different weights w_i (i=1...n) and bias have significant contribution to the convergence of the IHNNW especially when there is noise. The following variations of

the neural network are interesting:

(1) product terms only

$$OUT = F(NET) = F(\theta_0+T_p);$$

(2) linear and product terms only

$$OUT = F(NET) = F(\theta_0+T_1+T_p);$$

(3) product and square terms only

$$OUT = F(NET) = F(\theta_0+T_p+T_s);$$

(4) linear, square, and product terms

$$OUT = F(NET) = F(\theta_0+T_1+T_s+T_p).$$

Here θ_0 is bias and T_1 is linear term with different weights coming from noise as shown in Equation 2.19 [15].

Their ability to converge will be discussed in section 4.1.

2.3 Back-Propagation Learning

Training of a multi-layer neural network with back-propagation was well described by Kwan [15]. Below will discuss the back-propagation learning for the IHNNW discussed in section 2.2.

In the training phase of a network, a set of patterns X_p, p= 1..m, are presented sequentially and the weights, W, of the network and also the bias in the nodes are adjusted. The differences between the desired outputs (targets), b_p, and the real outputs, OUT_p, need to be minimized by repeating the adjusting process. After repeating a certain number of times, the set of outputs, OUT_p, should be very near to the set of

desired outputs (targets), b_p. Thus weights and bias found for the minimum error are the trained weights and bias.

To reduce the differences between a set of real outputs and a set of desired outputs, a specific error criteria must be used, the most common method is the least mean square (LMS) method. For each pattern, the square of the error is

$$E_p = \frac{1}{2}\sum_{k=1}^{c} (b_k - OUT_k)^2 \qquad (2.36)$$

where k ranges from 1 to the number of nodes, c, in the output layer and the factor 1/2 is inserted for mathematical differential convenience.

The average error for all the input patterns is

$$E = \frac{1}{2p}\sum_{p=1}^{m}\sum_{k=1}^{c} (b_{pk} - OUT_{pk})^2 \qquad (2.37)$$

where m is the number of patterns presented to the network. E is the average error of m patterns. The capability of convergency for all the input patterns is unknown, but the network can always converge for one input pattern. So, the network can be designed so that it will compute its output by the corrected weights and bias of the previous input pattern. If the network can converge, the outputs will move closer to the desired outputs and the total errors will decrease. If the network cannot converge, there will exist some outputs which will deviate more from the desired outputs and the total errors will not decrease. The error of one pattern can be

denoted by E, as

$$E = \frac{1}{2}\sum_{k=1}^{c} (b_k - OUT_k)^2 \ . \tag{2.38}$$

To reduce the error E, the incremental changes of weights, Δw_{kj}, should be proportional to $-\partial E/\partial w_{kj}$, that is

$$\Delta w_{kj} = -\eta \frac{\partial E}{\partial w_{kj}} \ , \tag{2.39}$$

where η is a proportionality constant called the gain factor or the learning rate. The nonlinear output of node k is then

$$\begin{aligned} OUT_k &= F(NET_k) \\ NET_k &= \sum_{j=1}^{b} w_{kj} OUT_j + Bias_k \ . \end{aligned} \tag{2.40}$$

Using chain rule differentiation,

$$\frac{\partial E}{\partial w_{kj}} = \frac{\partial E}{\partial NET_k}\frac{\partial NET_k}{\partial w_{kj}} = \frac{\partial E}{\partial NET_k} OUT_j \tag{2.41}$$

since

$$\frac{\partial NET_k}{\partial w_{kj}} = OUT_j \tag{2.42}$$

if δ_k is defined as

$$\delta_k = -\frac{\partial E}{\partial NET_k} \tag{2.43}$$

then

$$\Delta w_{kj} = \eta \delta_k OUT_j \ . \tag{2.44}$$

δ_k can be rewritten as

$$\delta_k = -\frac{\partial E}{\partial OUT_k}\frac{\partial OUT_k}{\partial NET_k} \quad . \tag{2.45}$$

Because

$$\frac{\partial E}{\partial OUT_k} = -(b_k - OUT_k) \tag{2.46}$$

and

$$\frac{\partial OUT_k}{\partial NET_k} = F'(NET_k) \tag{2.47}$$

δ_k can be expressed as

$$\delta_k = (b_k - OUT_k)\, F'(NET_k) \quad . \tag{2.48}$$

So, for the output layer node k, Δw_{kj} can be expressed as

$$\Delta w_{kj} = \eta\,(b_k - OUT_k)\, F'(NET_k)\, OUT_j \quad . \tag{2.49}$$

For the invariant high-order neural network, only one layer is needed. So, let j layer be the input layer and k layer be the output nodes.

In this thesis, the error is taken as

$$E' = \frac{1}{2c}\sum_{k=1}^{c} (b_k - OUT_k)^2 \quad . \tag{2.50}$$

This means that the error is averaged over the output nodes. setting

$$\eta' = \frac{\eta}{c} \tag{2.51}$$

gives

$$\Delta w_{kj} = -\eta \frac{\partial E'}{\partial w_{kj}} = -\eta' \frac{\partial E}{\partial w_{kj}} \quad . \tag{2.52}$$

The constant c can be absorbed into η', the learning rate. So, when running the program, a special learning rate is used to compensate for the effect of the constant c.

In particular, if the activation function is a sigmoid function

$$OUT_k = \frac{1}{1 + e^{(-NET_k + \theta_k)/\theta_0}} \tag{2.53}$$

setting

$$\begin{aligned} \theta'_k &= \theta_k / \theta_0 \\ NET'_k &= NET_k / \theta_0 \\ w'_{kj} &= w_{kj} / \theta_0 \\ Bias'_k &= Bias_k / \theta_0 \end{aligned} \tag{2.54}$$

then

$$\begin{aligned} \frac{\partial OUT_k}{\partial NET'_k} &= OUT_k(1-OUT_k) \\ \delta_k &= (b_k - OUT_k)\, OUT_k(1-OUT_k) \\ \Delta w_{kj} &= \eta' \delta_k OUT_j \end{aligned} \tag{2.55}$$

where OUT_j is the input layer at j^{th} input .

Now the independent variables are changed from θ_k, w_{kj}, $Bias_k$ and NET_k to θ_k', w_{kj}', $Bias_k'$, and NET_k'. The independent variable θ_0 now is inserted into other variables. So, all the corrective quantities $\Delta w_{kj}'$, $\Delta\theta_k'$, and $\Delta Bias_k'$ are needed to correct w_{kj}, θ_k, and $Bias_k$.

Using the same procedure, $\Delta\theta_k'$ and $\Delta Bias_k'$ are found.

$$\begin{aligned}\Delta\theta_k' &= -\eta' \frac{\partial E}{\partial \theta_k'} \\ \frac{\partial E}{\partial \theta_k'} &= \frac{\partial E}{\partial OUT_k}\frac{\partial OUT_k}{\partial \theta_k'} \\ &= -(b_k - OUT_k)\,[-OUT_k(1-OUT_k)]\ . \\ &= \delta_k\end{aligned} \tag{2.56}$$

Thus, $\Delta\theta_k' = -\eta'\delta_k$.

$$\begin{aligned}\Delta Bias_k' &= -\eta' \frac{\partial E}{\partial Bias_k'} \\ \frac{\partial E}{\partial Bias_k'} &= \frac{\partial E}{\partial OUT_k}\frac{\partial OUT_k}{\partial NET_k}\frac{\partial NET_k}{\partial Bias_k'} \\ &= -(b_k - OUT_k)\,OUT_k(1-OUT_k)\cdot 1\ . \\ &= -\delta_k\end{aligned} \tag{2.57}$$

Thus, $\Delta Bias_k' = \eta'\delta_k$.

For this sigmoid function, -NET' will be added to θ_k' and -$Bias_k'$ will be added to θ_k'. Thus if the Bias' increases $\eta'\delta_k$, the addition of -$Bias_k'$ and θ_k' will decrease $2\eta'\delta_k$.

It is important to note that, for this kind of activation function, a node cannot have an output value of 1 or 0 without infinitely large positive or negative weights. Therefore, in the learning phase, the value 0.9 and 0.1 are used for specifying binary target output values.

In learning w_{kj}', Bias', and θ_k', a set of patterns X_p, p=1...m, are applied sequentially. These parameters will be corrected for every input pattern. So, they will be corrected m times for the first iteration of patterns. Also, they will be corrected m times for every iteration thereafter. The number of iterations and the magnitude of average system error

can be set by the user.

Because the learning is processed pattern by pattern (step by step), if there are two patterns which are not compatible, then the former will be forgotten after the latter has been learned. And the difference of the outputs by the two patterns will increase while the number of iterations increase. But, if the two patterns are compatible, the difference of the two outputs by these patterns will decrease as the number of iterations increase.

The back-propagation algorithm has been tested with a number of deterministic problems and has performed well in most cases [9]. However, there are some issues that need to be addressed when such an algorithm is applied. There is no known best choice of a value for η'. As might be expected, a large η' corresponds to rapid learning but might also results in oscillations. There was suggestion to add a momentum term as below [15]:

$$\Delta w'_{kj}(i+1) = \eta' \delta_k OUT_j + \alpha \Delta w'_{kj}(i) \ . \qquad (2.58)$$

The quantity i+1 is used to indicate the $i+1^{th}$ step and α is a proportionality constant. The second term above is used to specify that the change in w_{kj}' at the $i+1^{th}$ step should be somewhat similar to the change undertaken at the i^{th} step. In this way, some inertia is built in, and momentum in the rate of change is conserved to some degree. A finite α tends to dampen the oscillation of system error but can also serve to slow the rate of learning. The $Bias_k'$ and θ_k' terms can be

adjusted by the same way as below:

$$\begin{aligned} \Delta Bias_k'(i+1) &= \eta'\delta_k + \alpha\Delta Bias_k'(i) \\ \Delta\theta_k'(i+1) &= -\eta'\delta_k - \alpha\Delta\theta_k'(i) \quad . \end{aligned} \tag{2.59}$$

By assuming θ_k to be a value between 0 and 1, θ_0 can be obtained and the shape of sigmoid function is known. The $Bias_k'$ and θ_k' terms can be combined into one term, thus the number of variables will be reduced and the speed of convergency will increase, but the information about the shape of the sigmoid function would be unknown.

One drawback of the back-propagation algorithm concerns the question of whether the system might get trapped in some local minimum or at some stationary point, or perhaps oscillate between such points, as shown in Figure 2.8. Under such circumstances, the system error remains large regardless of how many iterations are carried out. Suggestions to improve performance and reduce the occurrence of local minima include allowing extra hidden units, lowering the gain term η', and making many training runs starting with different sets of random weights [9]. However, there is no fixed method to eliminate the system from being trapped in local minimum or stationary points. All suggestions are empirical rather than extracted from theoretical ground.

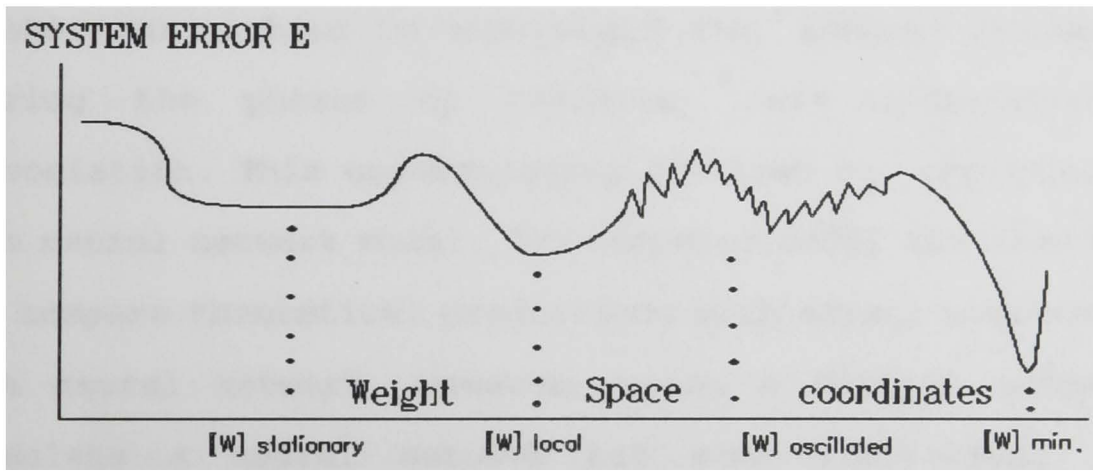


Figure 2.8 Illustration of the possibility of the learning procedure being trapped at non-optimum values

CHAPTER 3

SIMULATION METHOD

Although neural network technology is spreading to almost every branch of information science, the number of successful applications is still limited. Implementation of neural networks is a topic of considerable research and investigation [3].

Neural network technology is still a new technology. Most of the theories and proposed applications are the consequence of simulations. Simulation of a neural network model is a vital part of its implementation. Simulation can be used to test the accuracy of the theory involved in developing the neural network model. Simulation involves the conversion of a neural network model into a computer model. Computer models enable researchers to understand what happens in the model during the phases of training, self organization and association. This understanding can lead to improvements in the neural network model. The computer model can also be used to compare theoretical predictions with actual performance of the neural network. However, using a digital computer to simulate a neural network has some limitations. First, conventional computers are serial machines while neural networks exhibit massive parallelism. Second, signals in the human nervous system are essentially analog, while simulating them on a digital machine provides an inherent mismatch and

could provide results which may differ from the actual situation.

Neural network models are represented by mathematical formulas. The training algorithm is also expressed in mathematical form. Simulation of neural network models involves the writing of programs to implement the mathematical formulas and algorithms. The application of the simulated model usually involves four stages: artificial data generation, training of the network with artificial data, testing with artificial data, and testing with real data when possible. The most important and crucial step is the training of the network. Training involves the selection of parameters, such as the learning rate, the momentum rate, the error tolerance and others. For most of these parameters, there is no theoretical basis for the choice of a particular value. A trial-and-error approach is normally used in determining the parameters. Unfortunately, there is no guarantee of convergence for a given set of training vectors. If the network converges in training, the network architecture and weights are saved for future testing. If the accuracy of the testing is not high enough, the tolerance of the error can be reduced and the network trained again. By repeating the procedure, the accuracy may improve but not necessarily to a satisfactory level. If the network does not converge during the training phase, the network parameters can be adjusted to try to help the network converge. But if the network outputs

are far from the desired outputs, the network may not converge at all.

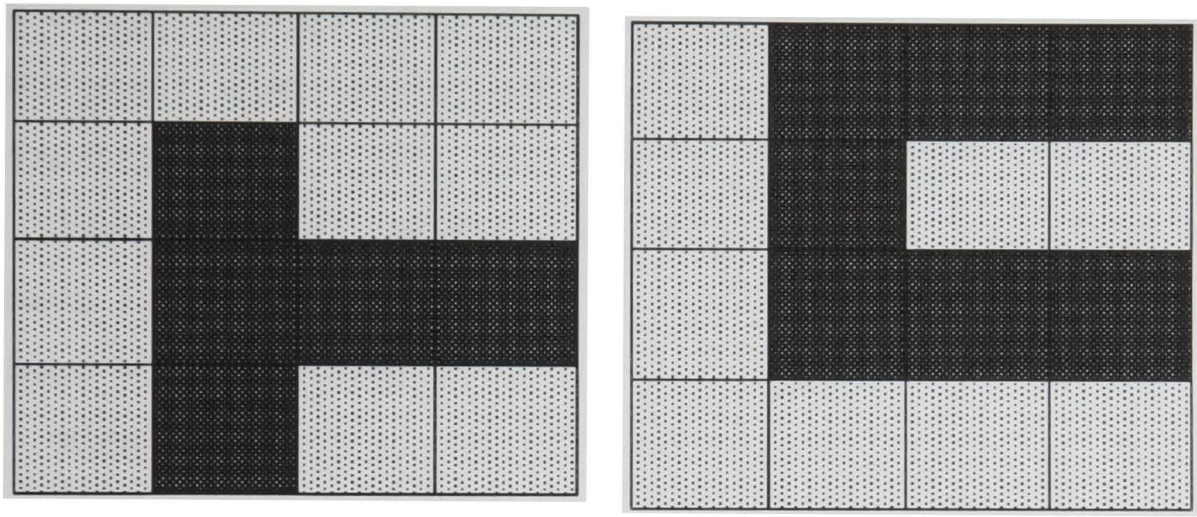
The simulation programs for this thesis are written in TURBO C 2.0 and are listed in Appendix A and are discussed in chapter 3. The structures of the programs come from Yee-Man Kwan [15] but the details are different. All the programs were run on a 486-33MHZ personal computer.

The main issue of this study is to solve the target detection problem defined in section 1.3. The translation and 90-degree rotation invariance IHNNW, stated in section 2.2, has the ability to solve this problem with the lowest cost and good effectiveness. There are 4 variations of this IHNNW (mentioned in section 2.2) and are tested through simulation and shown in section 4.1. A common test for pattern recognition invariance involves detecting the letters "T" and "C" [12]. The simulation results are also shown in section 4.1.

3.1 Signal Generation

In order to implement the theory of the 90-degree rotation and translation IHNNW, the patterns of the letters T and C are detected. Using a 3X3 pixel scene, with a letter T or C inserted, a plate is formed. By pasting the plate on a 4X4 pixel learning scene, a training scene is formed. For each letter plate, there are four directions top, left, down, and right. Also for each letter plate there are four possible

positions to paste it in a learning scene. So, for each letter plate there are 4X4 = 16 ways to paste it. For both T and C plates, there are 32 ways to paste both letters. By appending 8 blank learning scenes, a training set of 40 learning scenes is formed. Scenes of T and C are shown in Figure 3.1.



(a) T scene (b) C scene

Figure 3.1 4X4 TC scenes without noise

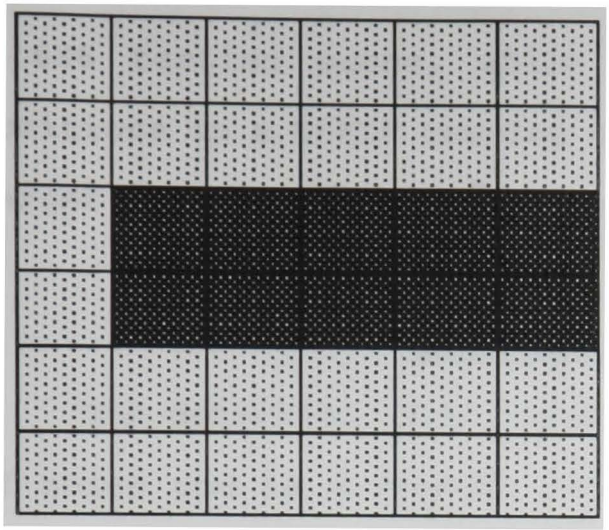
There are 3 possible features in a 4X4 pixel testing scene but only one feature exists. One is a letter T, another is a letter C, and the other is blank. The testing scenes can be divided into two groups. One has a T or C and the other does not have a T or C. Choosing 10 scenes as a set with a T or C in each scene, there are four scenes of letter T in four directions and four scenes of letter C in four directions. The ninth scene has a letter T and the tenth scene has a letter C. Because there are four possible positions in each scene, a

random function is used to determine the position. The number of scenes in a testing set is determined by the user but the number needs to be a multiple of ten. The TC detection problem is a simple and necessary way of testing the translation and 90-degree rotation IHNNW.

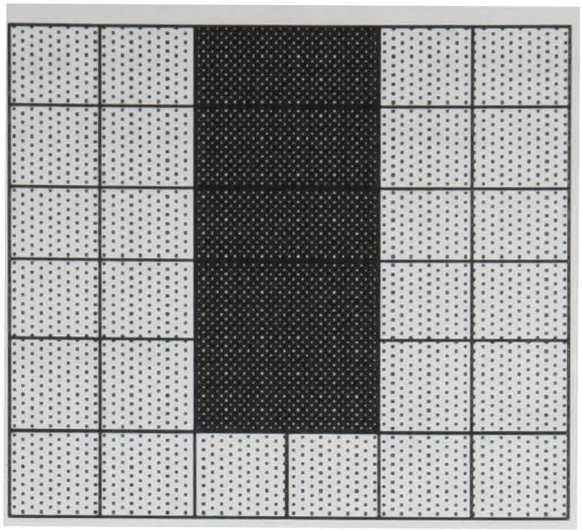
The specific problem of interest involves a submarine-like target in a noisy background. The target is simulated by a 5X2 pixel plate with four possible directions, vertical, horizontal, right-slant, and left-slant. They are similar to the target patterns studied by Jon P. Davis and William A. Schmidt [5]. The target pasted on a 6X6 pixel learning scene without noise is shown in Figure 3.2.

There are five features for the problem of target detection. Besides the four features shown in Figure 3.2, the fifth feature is a blank. Forty learning scenes with a target in each one and ten learning scenes without any target form a training set. The two slant-target patterns are repeated five times to make 20 scenes as is the number of vertical and horizontal target scenes. So the error coming from the slant-target scenes will have the same contribution as the error coming from the vertical and horizontal target scenes.

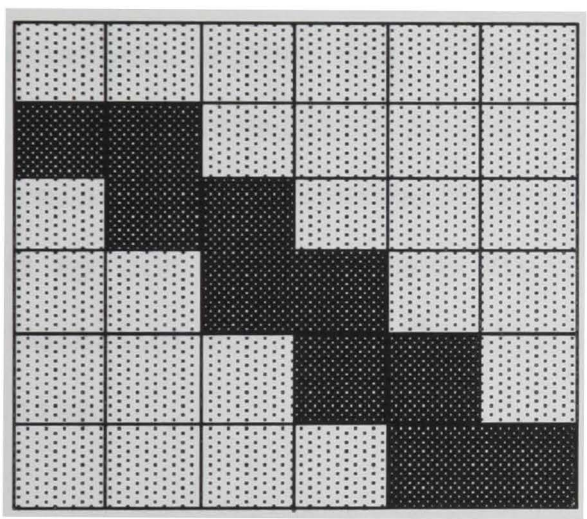
Two kinds of testing scenes are used. One is 6X6 pixel testing scene and the other is 16X16 pixel testing scene. Each of the scenes can be divided into two groups. One group has a target and the other does not have a target. For the group with a target in each scene with 10 scenes in a set, each set



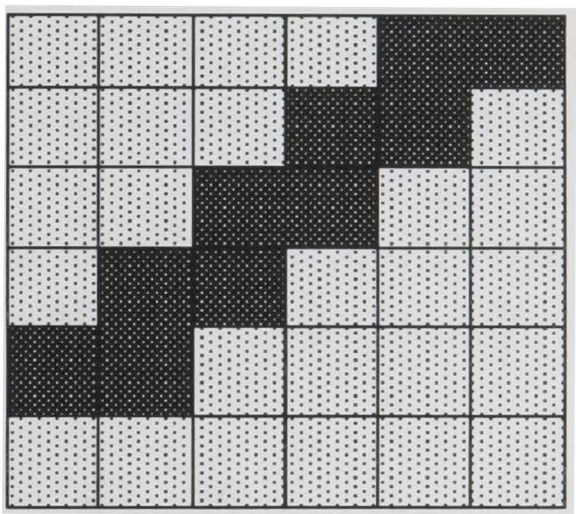
(a) a horizontal target



(b) a vertical target



(c) a right-slant target



(d) a left-slant target

Figure 3.2 6X6 scenes without noise

has 2 horizontal target scenes, 2 vertical target scenes, 3 right-slant target scenes, and 3 left-slant target scenes. The number of testing scenes should be a multiple of 10. The position of the target is determined by a random function. For

a slant target scene, there are only two possible positions in a 6X6 testing scene, while there are 12X11=132 possible positions in a 16X16 testing scene. The value of a pixel in a target is 1 and that of a blank is 0.

3.2 Noise Generation

The detection of a rectangular target under high levels of noise and in different positions is the primary issue of this thesis. The TC detection problem is used only as a verification of the 90-degree rotation and translation IHNNW.

There are four kinds of noise considered in this discussion. First the noise can be divided into experimental noise and normalized noise. Each type can be separated into correlated and uncorrelated noise. The experimental noise was used by Jon P. Davis and William A. Schmidt [6] and is shown in Equation 3.1. This formula is derived by experimental data fitting.

$$noise = 0.25 + sign \cdot AMP \cdot [1 - e^{-(\frac{x}{0.5})^2}] \qquad (3.1)$$

where the sign is +/- with equal probability and x is uniformly distributed in (0,1). The noise level is controlled by the variable AMP which is between 0 and 1. The noise is from -0.75 to 1.25 and the addition of the signal and noise is in the range (-0.75,2.25). If the addition of the signal and noise is less than 0, it is assigned to 0. Therefore the final range is (0,2.25). When the noise level is 0.5, the lower

bound of the "with target" pixels corrupted with noise is the same as the upper bound of the "without target" pixels corrupted with noise. This value is 0.75.

The normalized noise is designed to make the addition of the signal and noise between 0 and 1 so that whatever the signal is corrupted by noise or not, the total value is always in the range (0,1). The normalized noise is given by

$$noise = \begin{cases} -\frac{AMP}{2} + sign \cdot \frac{AMP}{2} \cdot [1 - e^{-(\frac{x}{0.5})^2}] \\ \quad , \text{if signal is } 1. \\ \frac{AMP}{2} + sign \cdot \frac{AMP}{2} \cdot [1 - e^{-(\frac{x}{0.5})^2}] \\ \quad , \text{if signal is } 0. \end{cases} \tag{3.2}$$

where sign, x, and AMP are the same as for the experimental noise. Under this noise, the corrupted signal will be in the range (0,1).

Both of the above noise types are uncorrelated noise when they are applied independently to every pixel in the scene. There are many ways to correlate the noise. Sometimes the adjacent pixels will have the same noise [6]. Correlated noise is formed from uncorrelated noise by making the noise in two horizontal adjacent pixels the same. The most left pixel of a horizontal line will have its own noise value or the same noise value as the second left pixel has with equal probability.

Sections 3.1 and 3.2 have described the important part of the data generated program, GTC.C which is in Appendix A. The flow chart of the program GTC.C is shown in Figure 3.3. The

program first asks for some parameters and generates samples for TC or the rectangular target detection. The program can add noise to the samples at the user's request. Finally, the program saves the data samples in a file. Some 16X16 scenes with experimental, uncorrelated noise with a level of 0.6 are shown in Figure 3.4.

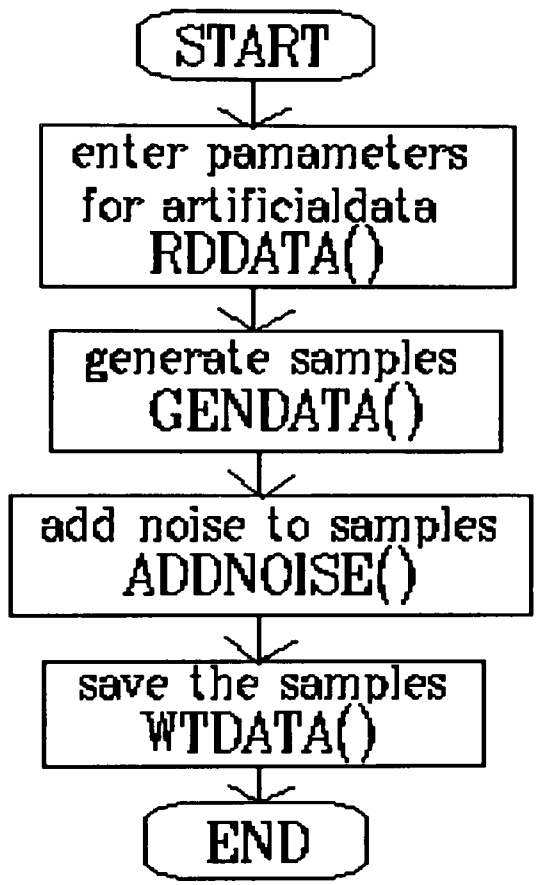
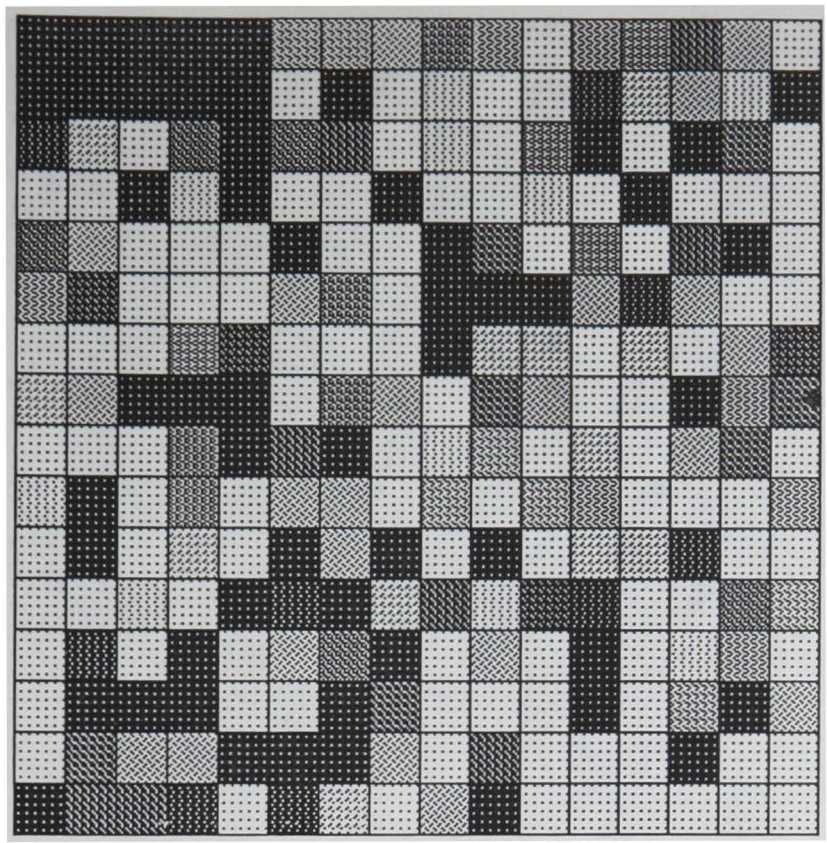
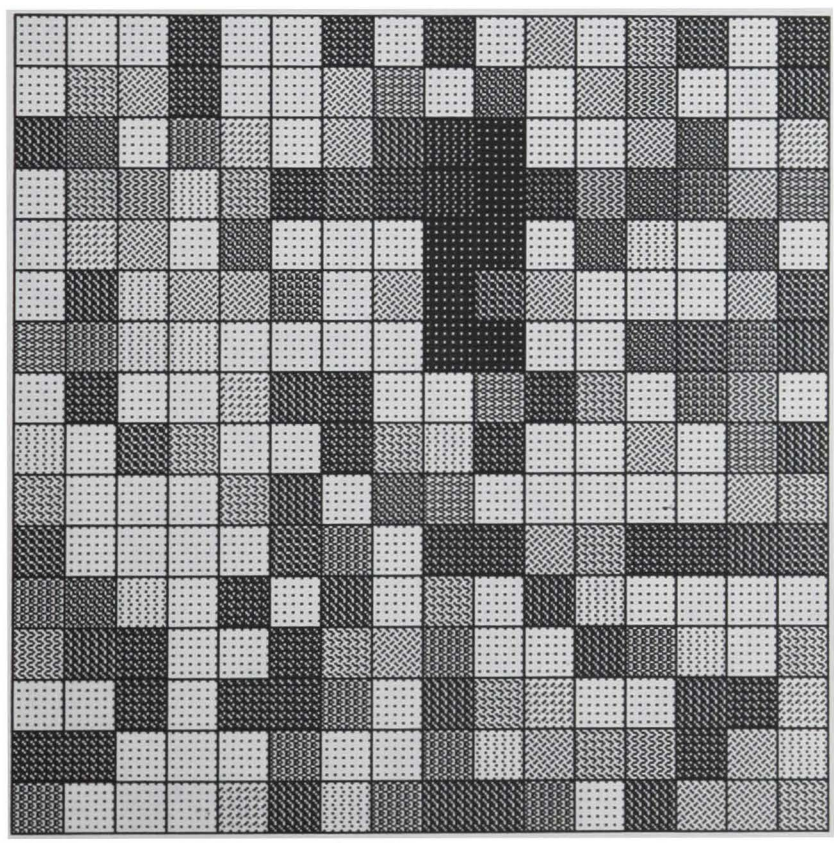


Figure 3.3 Flow chart for GTC.C

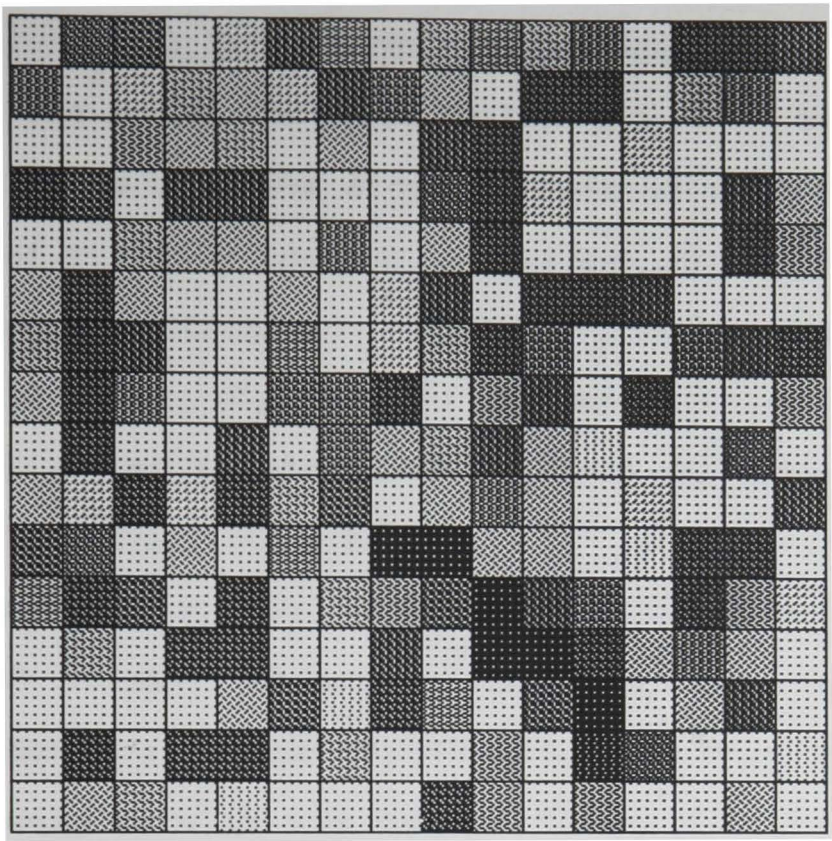


(a) horizontal target scene

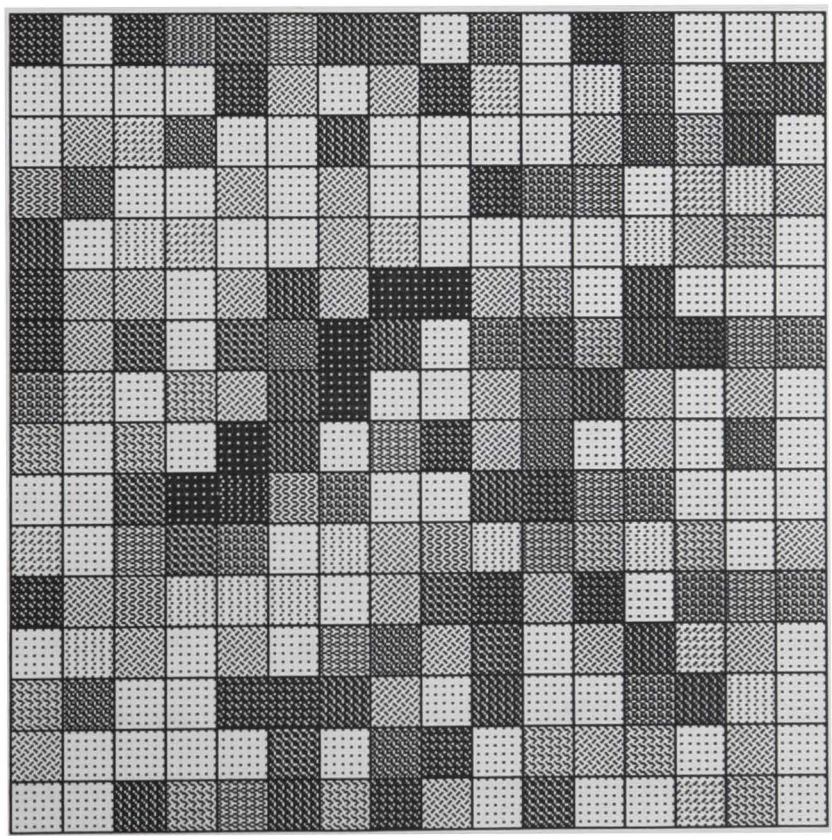


(b) vertical target scene

Figure 3.4 16X16 scenes at a noise level of 0.6



(c)right-slant target scene



(d) left-slant target scene

Figure 3.4 continued

3.3 Exhaustive Pattern Learning

NNWs can learn new patterns but they also can forget old patterns if the old ones are not repeatedly applied. Because the size of training scene is small(4X4 for TC detection and 6X6 for target detection), an exhaustive pattern set can be used for training. This means that all the possible signals with no noise can be collected to train the network.

For TC detection, there are 40 learning scenes with 8 blank scenes in them. The size of the letters is 3X3. When the letter "T" rotates in north, west, south, and east, four training samples are generated. Also in the same way, four training samples with the letter "C" in four directions are obtained. Thus, there are 8 samples in a set. There are 4 positions to paste a 3X3 plate in a 4X4 scene with no rotation. Therefore, 4 sets (8X4=32 samples) are needed. With 8 blank samples, there are a total of 40 samples to do an exhaustive pattern training.

For the rectangular target detection, there are 50 learning scenes with 14 blank scenes in them. Because the rectangular target size is 5X2 and the training scene size is 6X6, there are 10 possible positions for pasting a vertical target and also 10 for a horizontal target. There are only 2 possible positions for pasting a right-slant target and also 2 for a left-slant target (reference Figure 3.4). To balance the number of different training scenes, 8 right-slant target samples and 8 left-slant target ones are used. With another 14

blank scene samples, a training set with 50 samples is formed. Eight of the fourteen blank scene samples are mixed with the slant target scene samples. Therefore, the network can learn smoothly.

The detail description of the signal patterns is in section 3.1. When the noise is applied to the learning scenes, it is impossible to do exhaustive pattern training. That is because the random noise makes the values of pixels any real number in a certain range. So, the number of possible values is nearly infinitive (the number depends on the resolution of the computer).

Because there are 3 features (T, C, and blank) in TC detection, the network needs 2 output nodes, node 0 and node 1. For a learning scene with a T (no matter what multiple of 90 degrees it rotates), the desired output of node 0 is 0.9 and that of node 1 is 0.1. For a learning scene with a C (no matter what multiple of 90 degrees it rotates), the desired output of node 1 is 0.9 and that of node 0 is 0.1. For a blank learning scene, both desired outputs are 0.1.

For the rectangular target detection, there are 5 features and 3 output nodes, node 0, node 1, and node 2. If there is a horizontal or vertical 5X2 pixel target, the desired output of node 0 is 0.9 and those of the other two nodes are both 0.1. This is because of the 90-degree rotation invariance. If there is a right-slant 5X2 pixel target, the desired output of node 1 will be 0.9 and those the others are

both 0.1. If there is a left-slant 5X2 pixel target, the desired output of node 2 is 0.9 and that of the others is both 0.1. When the two slant features were combined into one output node, the training result of no noise case was not 100% correct. The reason that the right and left-slant features cannot be combined by 90-degree rotation invariance is because they are not a 90-degree rotation pair as shown in Figure 3.2. If there is no target, all the desired outputs will be 0.1.

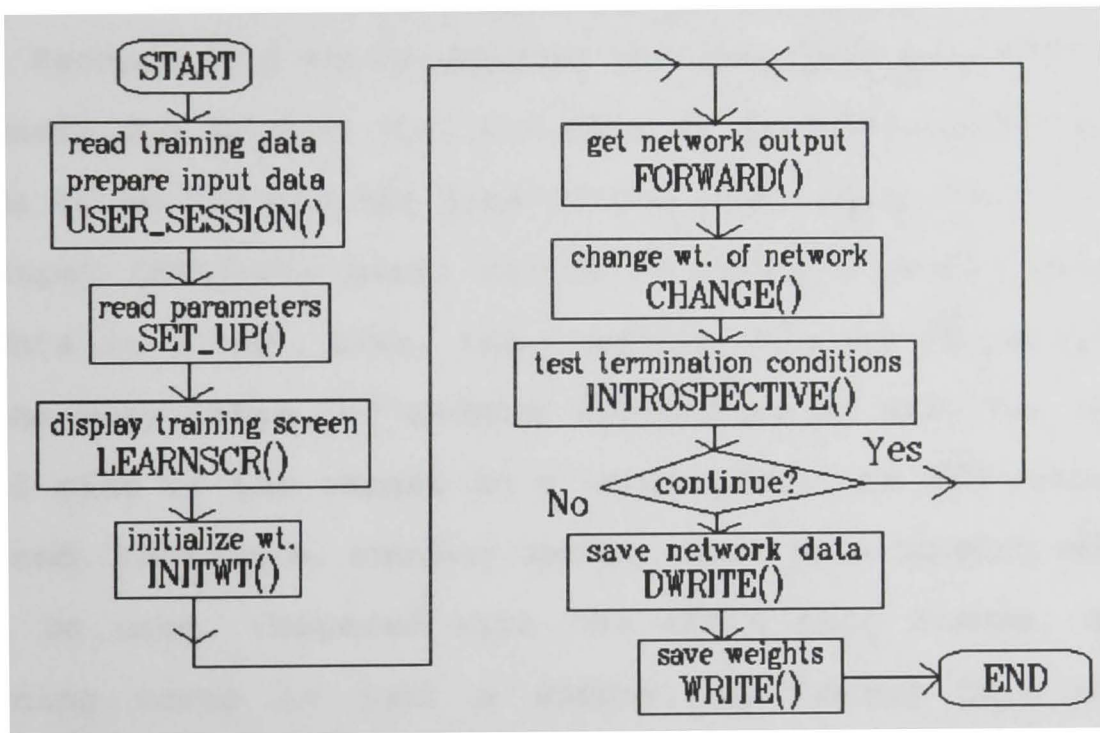


Figure 3.5 Flow chart for NNW learning

The flow chart of the learning function in the program, IHNW.C which is in Appendix B, is shown in Figure 3.5. In the function of user_session(), the input values will be

rearranged to get linear, product, and square terms which are shown in Equation (2.34) and (2.35). There are four kinds of neural networks, as described in section 2.2, used in this study.

The core part of training function is the function rumelhart(), which has a function forward() and a function change(). The former calculates the output values of each output node and the latter corrects the weights and bias of each output node.

3.4 Window Output and Threshold

Because this thesis studies the same problem in the study of Davis and Schmidt [5], the size of the rectangular target needs to be 5X2 and the size of the scene needs to be 16X16. To input 16X16=256 pixel values requires a large number of weights in a NNW. Also, the training set and training time become very large. Of greater importance is that due to the small size of the target in a large scene the NNW cannot be trained. Therefore, another method, such as a "moving window" must be used. Compared with the 16X16 test scenes, a 6X6 learning scene is just a window. By moving this window horizontally one column each time and vertically two rows each time (totally 11X6 = 66 times), a 16X16 test scene can be completely covered. Refer to Figure 3.2 (c) and (d). Because the slant targets occupy six columns, the window needs to move one column at a time. Also because the slants target occupy

five rows, the window can move two rows at a time. There are 3 output nodes. Each output node value of a 16X16 test sample is the greatest output node value of the corresponding output nodes of 11X6=66 window outputs.

Because the real output values are not the same as the desired output value, a threshold is usually set between 0.1 and 0.9. For TC detection, if the value of node 0 is greater or equal to the threshold, then the network will indicate that there is a T. A similar relation holds for node 1. If both the values are smaller than the threshold, the network will indicate there is nothing in the scene. The thresholds can be set by the user. For the rectangular target detection, the process is similar. Because rotation is limited to 90 degrees, the shape of the target needs to be very simple and have 90-degree symmetry. If it does not, then the extra output nodes are needed to represent the unsymmetrical shapes. Output node 0 represents a horizontal or vertical target, node 1 represents a right-slant target, and node 2 represents a left-slant target. This is because when the right-slant target is rotated by a multiple of 90 degrees, it does not look like the left-slant target shown in Figure 3.2.

The core part of the output function is the function output_generation(). It calls the function forward(). The flow chart is shown in Figure 3.6. The output function is built in the program IHNW.C in Appendix B.

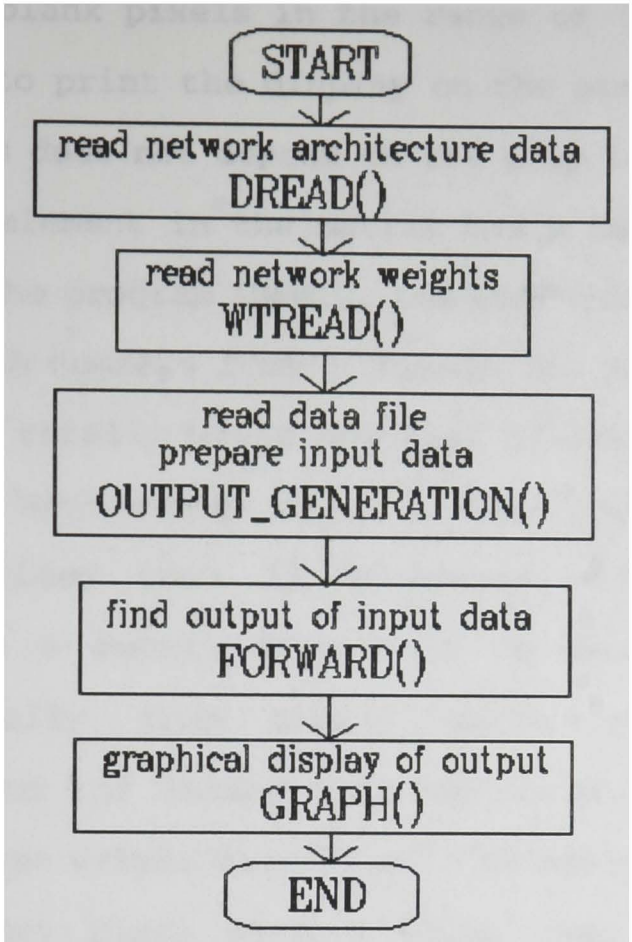


Figure 3.6 Flow chart for output generation

3.5 Display

The display of patterns (TC or Rectangular Target) offers the user a chance to compare the results of the network and the display shown on the screen. The user can determine if there is a target by watching the display on the screen.

Without noise, the signal pixels have a value of 1 and the blank pixels have a value of 0. With experimental noise, the signal pixels have values in the range (0,2.25) and the blank pixels have values in the range (0,1.25). With normalized noise, the signal pixels have values in the range

(0,1) and the blank pixels in the range of (0,1).

In order to print the display on the screen, the display of pixel values does not depend on the gray level but on a 8X8 matrix. Every element in the matrix has a value 1 (black) or 0 (white). In the program IHNW.C, the function graph() assigns 17 matrices with numbers from 0 through 16. The display of the No.0 matrix is totally white and that of the No.16 matrix is totally black. With a step of 1/15, the range (0,1) of pixel values is divided into 15 intervals and each interval corresponds to a matrix from No.1 to No.15. The display changes gradually from almost white to almost black corresponding to the change from matrix No.1 to No.15. The pixels which have values above 1 will be assigned to the No.15 matrix. Thus any pixel with a value over .93334 will be represented by matrix No.15.

The flow chart of the graphical presentation of data is shown in Figure 3.7. The core part of the presentation is the function graph() which defines the filling pattern for different levels of values and displays the entire screen and output values. The display function is built into the program IHNW.C in Appendix B.

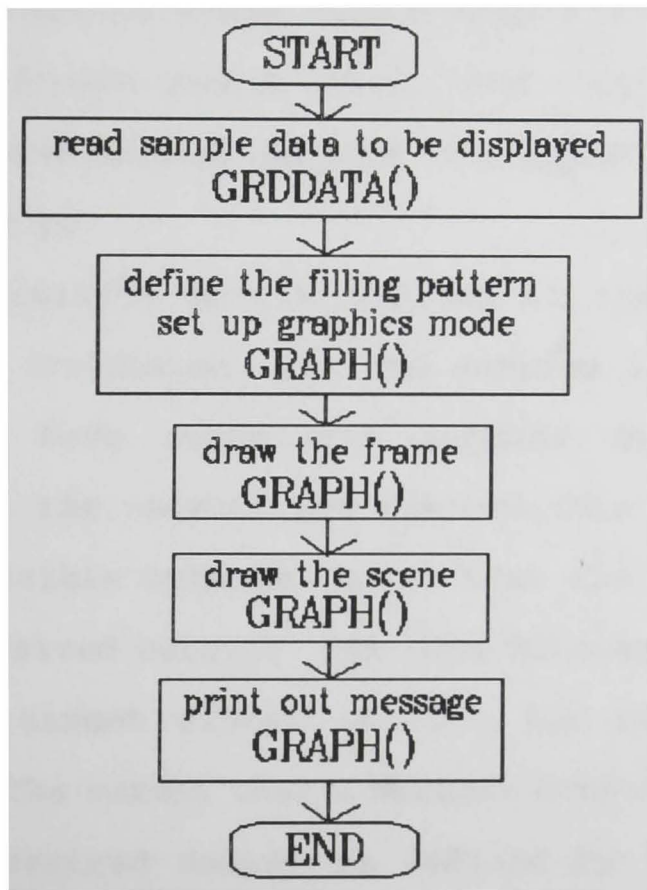


Figure 3.7 Flow chart for graphical presentation of data

CHAPTER 4

SIMULATION RESULTS AND DISCUSSION

4.1 Capability of Learning

During the learning phase, a set of pixel values is applied to the network. The network rearranges the pixel values to form a set of inputs and forwards them to the output nodes. The differences of the outputs and the desired outputs will be used to correct the weights and bias in each output node. After a number of iterations, the output will approach the desired output in every output node, if the network can learn. If the network cannot learn, some outputs will deviate gradually from the desired outputs. The capability of learning is defined here by:

> After a training set is applied to the network for a number of iterations, all the samples in this training set will have compatible outputs by the network. Therefore, the network has learned this training set.

The "compatible outputs" means that the network outputs approach the desired outputs. The desired outputs are 0.9 for pixels with a target signal and 0.1 for pixels without a target signal. The extent that a network output approaches the corresponding desired output is defined by the "threshold" value that is set by the user.

The IHNNW with translation and 90-degree rotation invariances discussed in section 2.2 is used here. The

following variations of the neural network are discussed :

(1) product terms only,

(2) linear and product terms only,

(3) product and square terms only, and

(4) linear, square, and product terms.

With the method in section 3.3, the training of TC and rectangular target detection uses the parameters shown in Table 4.1.

For the rectangular target detection problem, a 6X6 pixel training scene is used. The number of output nodes is 3. There are 50 samples in a training set. The details of the training set are in section 3.3. There are some better ways to select the learning rate and momentum rate. In this study, their values are chosen only by a few experiments. Because the learning rate is chosen as small as 0.1, the momentum rate can be set to 0. Training stops when the normalized system error is 0.001 or less and the normalized individual error is 0.0001 or less. The initial weights are all set to 0 and the output function is a sigmoid.

For the TC detection problem, the training scene is 4X4. The number of output nodes is 2. There are 40 samples in the training set. Section 3.3 describes the details of the training set. Other parameters are the same as in the rectangular target detection case.

Table 4.1 Training parameters of target and TC detection

	target	TC
No. of pixels in input scenes	6X6=36	4X4=16
No. of output nodes	3	2
No. of input samples	50	40
learning rate	0.1	0.1
momentum rate	0.0	0.0
normalized system error	0.001	0.001
normalized individual error	0.0001	0.0001
initial weights	0	0
output function	sigmoid	sigmoid

The following tables indicate the number of iterations (called counts) to obtain the minimum error, the total number of iterations for training, the minimum total normalized error (normalized by the number of output nodes), and the total training time (in seconds).

In the following tables, "P" denotes the NNW with only product terms. "S" denotes the NNW with product and square terms. "L" denotes the NNW with product and linear terms. "I" denotes the NNW with product, square, and linear terms. Because this thesis originates from the study of Davis and Schmidt [5], the same kind of noise, experimental uncorrelated noise is used. The letter "u" denotes experimental uncorrelated noise and the number after "u" denotes the level of noise. Only Table 4.2 is related to TC detection, the other tables in Chapter 4 are all related to the rectangular target

detection. All the training and testing were done by a 486-33MHZ personal computer.

To learn the difficult pattern, letters "T" and "C," the NNW with all three terms and with no noise was employed. Table 4.2 shows the training results of the NNW for TC patterns. The error is still larger than 0.001 after 10,000 iterations, but the outputs are compatible to the desired outputs. This means that the outputs are similar to the desired outputs. If proper threshold values are set, the NNW can obtain 100% accuracy with the TC detection problem. Therefore, it can learn well with the difficult case of TC detection under no noise condition.

Table 4.2 Training results of NNW for TC patterns

NNW Type	Iu0
min_err CNTs	10,000
total CNTs	10,000
min_err	0.001670
total time(sec)	1,382

The following paragraphs discuss the NNWs with the rectangular target detection problem. Table 4.3 shows the training results for different NNWs with a zero noise level. The "Pu0" and "Su0" cases did not converge. After 1000 iterations, both minimum errors were greater than 0.001. Some of the outputs of these networks were not compatible with the desired outputs. This means that some desired output is 0.1 (0.9) and the corresponding network output deviates from 0.1

(0.9). Thus, the networks were not able to learn the training set. The NNWs of "Lu0" and "Iu0" learned well. Their errors were both less than 0.001 and the outputs were compatible with the desired outputs.

Table 4.3 Training results of different NNWs at a noise level of 0

Types of NNW	Pu0	Su0	Lu0	Iu0
CNTs of min_err	142	32	74	100
total CNTs	1000	1000	74	100
min_err	0.048442	0.034465	0.000993	0.000999
total time(sec)	82	84	20	52

Table 4.4 shows the training results for the different NNWs with a noise level 0.6. Again "Pu6" and "Su6" did not converge to the desired outputs. "Lu6" did not quite converge after 5000 iterations. "Iu6" did converge, but required over 3000 iterations.

Table 4.4 Training results of different NNWs at a noise level of 0.6

Types of NNW	Pu6	Su6	Lu6	Iu6
CNTs of min_err	5000	296	5000	3032
total CNTs	5000	1000	5000	3032
min_err	0.017719	0.020570	0.004618	0.001
total time(sec)	418	84	1383	818

The NNW with product, linear, and square terms was the only one of the four types that could be trained under a

variety of noise levels. The following discussion will concentrate on this network.

4.2 Convergent Time of NNWs Trained with Different Levels of Noise

The NNW with product, linear, and square terms was trained with experimental uncorrelated noise at levels of 0, 0.6, 0.7, 0.8, and 0.9. The number of iterations (called counts) to obtain the minimum error, the total number of iterations for training, the minimum total normalized error (normalized by the number of output nodes), and the total training time (in second) of the network trained with the different levels of noise are shown in Table 4.5.

In Table 4.5, "I" denotes the NNW with product, linear, and square terms. The letter "u" denotes experimental uncorrelated noise and the number after "u" denotes the level of the noise.

Table 4.5 Training results of different levels of noise

NNW Type	Iu0	Iu6	Iu7	Iu8	Iu9
min_err CNTs	100	3,033	1,592	1,273	5,000
Total CNTs	100	3,033	1,592	1,273	5,000
min_err	0.000999	0.001	0.001	0.001	0.001564
time(sec)	52	818	430	344	1,382

Without noise, the time to converge is the smallest. With a noise level of 0.9, the convergence time is the largest and the error is still more than 0.001. Therefore, the NNW "Iu9"

does not converge well. With noise levels of 0.6, 0.7, and 0.8, the convergence time does not increase as the noise level increases. In this case, the convergence time decreases as the noise level increases.

4.3 6X6 Scene Outputs of NNWs Trained with Different Levels of Noise

The NNW with all three terms and trained with experimental uncorrelated noise with noise levels of 0, 0.6, 0.7, and 0.8 was then tested. The statistical testing results are shown in Table 4.6. The table entries indicate the percent of correct identifications. The threshold values were 0.5 for all 3 output nodes.

In "W6u0," every test sample has a target in a 6X6 scene with experimental uncorrelated noise level of 0. There are 20 samples in a set and 5 sets in the sample file, "W6u0." The other 7 test sample files have the same structures. The letter, "W" means there is a target in the scene, while "O" means there is no target in the scene. The number, "6" means the scene size is 6X6, whereas "u" denotes experimental uncorrelated noise and the number after "u" denotes the level of the noise.

In the "without target" test sample files, the signals only contain the noise at some level. In the "with target" test sample files, there is only one target in a sample in the order of horizontal, vertical, right-slant, and left-slant. However, the position of the target in a scene is random. The

Table 4.6 Statistical results of 6X6 scene inputs

NNW	Iu0	Iu6	Iu7	Iu8	NNW	Iu0	Iu6	Iu7	Iu8
W6u0	100	85	80	75	O6u0	100	100	100	100
	100	85	75	60		100	100	100	100
	100	70	75	75		100	100	100	100
	100	60	80	75		100	100	100	100
	100	70	80	65		100	100	100	100
mean	100	74	78	70	mean	100	100	100	100
std	0	9.7	2.4	6.3	std	0	0	0	0
W6u6	100	90	75	75	O6u6	30	90	90	95
	100	85	80	65		25	80	95	100
	100	80	85	80		20	80	80	85
	100	95	85	70		20	80	95	80
	100	85	70	60		10	70	90	95
mean	100	87	79	70	mean	21	80	90	91
std	0	5.1	5.8	7.1	std	6.6	6.3	5.5	7.3
W6u7	100	85	95	65	O6u7	5	95	90	90
	100	95	95	08		15	60	90	100
	100	95	95	80		25	85	85	90
	100	95	75	75		30	80	80	100
	100	80	90	70		15	65	75	100
mean	100	90	90	74	mean	18	77	84	96
std	0	6.3	7.7	5.8	std	8.7	12.9	5.8	4.9
W6u8	100	85	80	75	O6u8	0	55	75	80
	100	90	100	80		0	80	70	85
	100	90	90	90		5	50	70	80
	100	90	90	70		5	65	75	80
	100	80	80	70		5	70	70	90
mean	100	87	88	77	mean	3	64	72	83
std	0	4	7.5	7.5	std	2.4	10.7	2.4	4

"with target" signals then were corrupted by noise at some level.

The NNW "Iu0" trained without noise, is 100% accurate for both the "with target" and the "without target" cases if the noise level of the test samples is 0. However, the accuracy is very poor for the case of the "without target" and with some level of noise.

For the NNW "Iu6" trained with a noise level of 0.6, the accuracy (mean) of the "without target" case increases as the noise level decreases but the accuracy (mean) of the "with target" case decreases as the noise level decreases. For the test sample files , "W6u6" and "O6u6," with a noise level of 0.6, both the accuracies (means) of the "with target" and the "without target" cases are above 80%.

The NNW "Iu7" trained with a noise level of 0.7 has similar results. The accuracy (mean) of the "without target" case increases as the noise level decreases, but, the accuracy (mean) of the "with target" case decreases as the noise level decreases. For the test sample files, "W6u7" and "O6u7", with a noise level of 0.7 , both the accuracies (means) of the "with target" and the "without target" cases are above 80%.

The NNW "Iu8" trained with a noise level of 0.8, has the same increasing and decreasing accuracy conditions as "Iu7." For the test sample files, "W6u8" and "O6u8" with a noise level of 0.8, the accuracies only approaching 80%.

The effect of training the NNW with different levels of

noise is shown on the horizontal lines in Table 4.6. The results of the sample files, "W6u6" and "O6u6," for a training noise level 0.6 is an example. The accuracy (mean) of "W6u6" decreases as the noise level of the trained NNW increases but the accuracy (mean) of "O6u6" increases. Meanwhile, the outputs of both "Wxu8" and "Oxu8" decrease as the noise level of the trained NNW increases. This means that if the NNW is trained with higher levels of noise, it will output lower values.

4.4 Experimental Noise Effect

In section 3.2, the range of pixel values for the addition of signal and noise is depicted for both normalized and experimental noise. If the normalized noise is used, the range of pixel values will still in (0,1) no matter what the level of noise is. But if the experimental noise is used, the range of pixel values will be in the range (0.15, 1.35) for a noise level 0.1 and in the range (0, 2.15) for a noise level 0.9. Thus, the range is (0, 1.85) for a noise level 0.6, (0, 1.95) for a noise level 0.7, and (0, 2.05) for a noise level 0.8.

With experimental noise, the highest limit in the range of the corrupted signal increases as the noise level increases. When the NNW is trained by higher noise added data, it will recognize higher pixel values as normal ones. If a sample with a lower noise level is used (the highest limit

will be lower), the NNW will output lower values. Most of the values will be smaller than the threshold value. Thus, if the sample has a target, the NNW will think there is no target. If the sample does not have a target, the NNW will more likely indicate there is no target. Therefore, the NNW trained with higher noise levels is less accurate at detecting a target when there is a target than the NNW trained at lower noise levels. Similarly, the NNW is better at detection the "without target" case, when it is trained with higher noise levels.

In above, all the networks trained and tested with the same level of noise have very good accuracy. However, when they were tested by samples with different levels of noise, the results are not good enough. This is because the training is not sufficient. If the network is trained by the training sets of noise levels from 0 to 0.8, the network should have good accuracy with the test samples of noise levels from 0 to 0.8.

4.5 Size Effect and Incomplete Effect

As mentioned previously, the size of "T" and "C" is 3X3 and that of the rectangular target is 5X2. The smallest window for pasting a "T" or "C" is 3X3 and that for pasting a rectangular target is 6X6 because of some slant target patterns.

Without noise, the special IHNNW in this study can be trained for a 4X4 window pasting with T or C. However, the

network does not converge for 5X5 and greater sized windows. Without noise, the network with a 6X6 window pasting with a rectangular target can be trained but an 8X8 or greater sized window cannot. It seems that the ratio of the number of signal pixels to the number of window pixels has to be greater or equal to a certain value, approximate 5/16 in this case. That the ratio of the size of the target and the scene needs to be greater than some value makes a "size effect."

The reason for this effect is because the network is very simple. For target detection, the number of product weights is 19. When it is added by the 36 linear weights and square weight, the number of weights for a output node is 56. The total number of weights for 3 output nodes is 168. For TC detection, the number of product weights is 9. The number of linear weights is 16. The number of weights for a output node is 26. The total number of weights for 2 output nodes is 52. Those numbers are very small when compared with the 3rd-order network described in Equation 2.23.

For rectangular target detection, the window size is limited to 6X6 by the target size but the test sample is a 16X16 pixel scene. Therefore, the window is applied sequentially over the scene of a test sample as mentioned in section 3.4. This moving window causes an "incomplete effect."

Usually when the window moves past a target, it will generate several window outputs with incomplete target inputs. Because such incomplete target patterns were not taught, the

window may classify them to any output node. If they are classified to the output node that indicates a target, the classification is right. If they are classified to the output node that indicates no target, the target can still be recognized by a moving window that covers the entire target. In this study, the incomplete targets are usually classified to output node 0 by the network itself. This causes some trouble when an incomplete target is formed by noise. Therefore, a proper threshold value for output node 0 is necessary to distinguish a complete target from an incomplete target.

4.6 16X16 Scene Outputs of NNWs Trained with Different Levels of Noise

Because this thesis originates from the study of Davis and Schmidt [5], the NNW needs to detect a 5X2 rectangular target in a 16X16 scene with experimental uncorrelated noise of a level at least 0.7. The NNW with all three terms and trained with experimental uncorrelated noise with noise levels of 0, 0.6, 0.7, and 0.8 was then tested. The statistical testing results are shown in Table 4.7. The table entries indicate the percent of correct identifications. The threshold values are all 0.8 for the test samples without noise. The threshold values for the test samples with noise levels above or equal to 0.6 are 0.995, 0.8, and 0.8 for the 3 output nodes. The threshold values are chosen by directly looking at the output values obtained by entering the training set.

Table 4.7 Statistical results of 16X16 scene inputs

NNW	Iu0	Iu6	Iu7	Iu8	NNW	Iu0	Iu6	Iu7	Iu8
Wxu0	100	95	79	77	Oxu0	100	100	100	100
	100	97	81	76		100	100	100	100
	100	93	74	77		100	100	100	100
	100	95	79	78		100	100	100	100
	100	95	77	72		100	100	100	100
mean	100	95	78	76	mean	100	100	100	100
std	0	1.3	2.4	2.1	std	0	0	0	0
Wxu6	100	94	79	61	Oxu6	100	96	96	99
	100	89	76	67		100	97	99	100
	100	92	76	73		100	97	98	99
	100	85	74	64		100	94	98	99
	98	88	68	61		100	97	98	99
mean	100	90	75	65	mean	100	96	98	99
std	0.8	3.1	3.7	4.5	std	0	1.2	1.0	0.4
Wxu7	100	96	89	64	Oxu7	88	72	86	96
	100	95	79	65		88	73	93	98
	99	97	83	62		88	77	88	97
	100	97	80	69		89	74	89	98
	100	96	82	78		85	71	83	95
mean	100	96	83	68	mean	88	73	88	97
std	0.4	0.7	3.5	5.7	std	1.4	2.1	3.3	1.2
Wxu8	100	97	95	81	Oxu8	23	32	63	80
	100	98	90	81		41	42	64	88
	100	96	92	83		26	33	48	84
	100	97	94	81		34	39	59	86
	100	99	95	83		33	33	55	80
mean	100	97	93	82	mean	31	36	58	84
std	0	1.0	1.9	1.0	std	6.3	4.0	5.8	3.2

Because some training samples are "with target" and others are not, the proper threshold values can be chosen by their network outputs. From the incomplete effect of section 4.5, the incomplete targets which are formed by noise are usually classified to output node 0. Therefore, to distinguish a complete target and an incomplete target, a high threshold as 0.995 for output node 0 is necessary.

There are 3 output nodes. Each output node value of a 16X16 test sample is the greatest output node value of the corresponding output nodes of 11X6=66 window outputs obtained by the moving window technique (reference to section 3.4). Therefore, the 3 outputs may come from different window outputs. For a "with target" window, usually there is only one output value is high and the other two values are low. For the 66 windows in a test sample, any one of the 3 outputs comes from the highest value of the 66 window outputs. Therefore, the 3 outputs may all be high values. If any one of the 3 output node values is greater than or equal to its corresponding threshold value, the NNW will indicate that there is a target.

In "WXu0," every test sample has a target in each 16X16 scene with experimental uncorrelated noise of level 0. There are 100 samples in a set and 5 sets in "WXu0." The other 7 test sample files have the same structures. The letter, "W" means there is a target in the scene, while "O" means there is no target in the scene. The letter "X" means the scene size is

16X16. Whereas "u" denotes experimental uncorrelated noise and the number after "u" denotes the level of the noise.

In the "without target" test sample files, the signals only contain the noise at some level. In the "with target" test sample files, there is only one target in a sample in the order of horizontal, vertical, right-slant, and left-slant. However, the position of the target in a scene is random. The "with target" signals were corrupted by noise at some level.

The NNW "Iu0" trained without noise is 100% accurate for both the "with target" and the "without target" cases if the noise level of the test samples is 0. However, "Iu0" does very poorly for the case of "without target" and with some level of noise.

For the NNW "Iu6" trained with a noise level of 0.6, the accuracy (mean) of the "without target" case increases as the noise level decreases but the accuracy (mean) of the "with target" case decreases as the noise level decreases. For the test sample files , "Wxu6" and "Oxu6," with a noise level of 0.6, both the accuracies of the "with target" and the "without target" cases are nearly 90%.

The NNW "Iu7" trained with a noise level of 0.7, has similar results. The accuracy of the "without target" case increases as the noise level decreases but the accuracy of the "with target" case decreases as the noise level decreases. For the test sample files, "Wxu7" and "Oxu7," with a noise level of 0.7, both the accuracies of the "with target" and the

"without target" cases are above 80%.

The NNW "Iu8" trained with a noise level of 0.8, has the same increasing and decreasing accuracy conditions. For the test sample files, "Wxu8" and "Oxu8," with a noise level of 0.8 the accuracies are both above 80%.

The effect of training the NNW with different levels of noise is shown on the horizontal lines in Table 4.7. The results of the sample files, "WXu8" and "OXu8," for a training noise level 0.8 is an example. The accuracy of "Wxu6" decreases as the noise level of the trained NNW increases but the accuracy of "Oxu6" increases. Meanwhile, the outputs of both "Wxu8" and "Oxu8" decrease as the noise level of the trained NNW increases. This indicates that if the NNW is trained with higher levels of noise, it will output lower values. This characteristic comes from the effect of experimental noise and is explained in section 4.4.

4.7 Comparison of Network Output and Display

This section shows the screen display of targets in noise along with the NNW output values. An examination of these displays can provide a good qualitative evaluation of the NNW.

There are 3 output nodes. Each output node value of a 16X16 test sample is the greatest output node value of the 11X6=66 window outputs obtained by the moving window technique. Therefore, the 3 outputs may come from different window outputs. For the 66 windows in a test sample, any one

of the 3 outputs comes from the highest value of the 66 window outputs. Therefore, the 3 outputs may all be high values. If any one of the 3 output node values is greater than or equal to its corresponding threshold value, the NNW will indicate that there is a target.

The first output node (node 0) represents both the horizontal and vertical targets. The second node (node 1) stands for right-slant targets. The third node (node 2) is for left-slant targets. If the output value is very close to 1 or 0, this means that the NNW can easily detect if there is a target or not. If the output values are around the threshold values, this means that it is difficult for the NNW to tell if there is a target or not.

With all three terms in the NNW and the signal corrupted by experimental uncorrelated noise at a level of 0.8, the samples are tested by the NNW trained with noise level 0.8. Displays with and without targets are individually shown in Figure 4.1 through Figure 4.4. The background is bright and the target is dark. The first horizontal line shows the network outputs of 3 output nodes from the 3 highest output node values of the 66 window outputs. The second horizontal line shows the desired outputs of 3 target nodes.

The usual test results of "with target" samples are similar to Figure 4.1 and those of "without target" samples are like Figure 4.2. There are some unusual cases shown in Figure 4.3 and Figure 4.4.

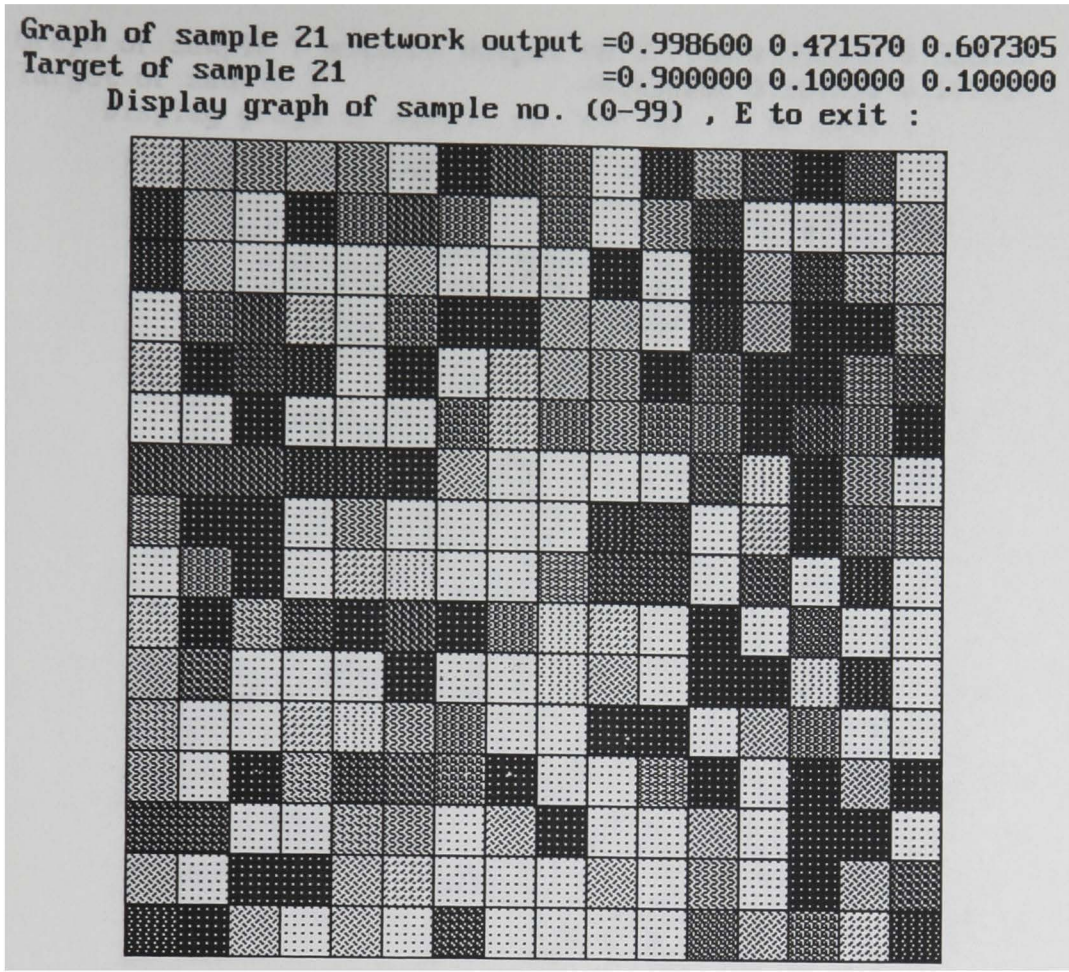


Figure 4.1 A 16X16 scene at a noise level of 0.8 (1)

In Figure 4.1, there seems to be a horizontal target and a vertical target in the right-upper of the scene. The value of the NNW output node 0 is above 0.995 and the value of other nodes are below 0.8. This means that the NNW indicates there is a horizontal or vertical target. In fact, the target signal shows that the value of the target node 0 is 0.9 and the other target nodes are only 0.1. There is only a vertical or horizontal target in the scene. The NNW can catch the target in the scene.

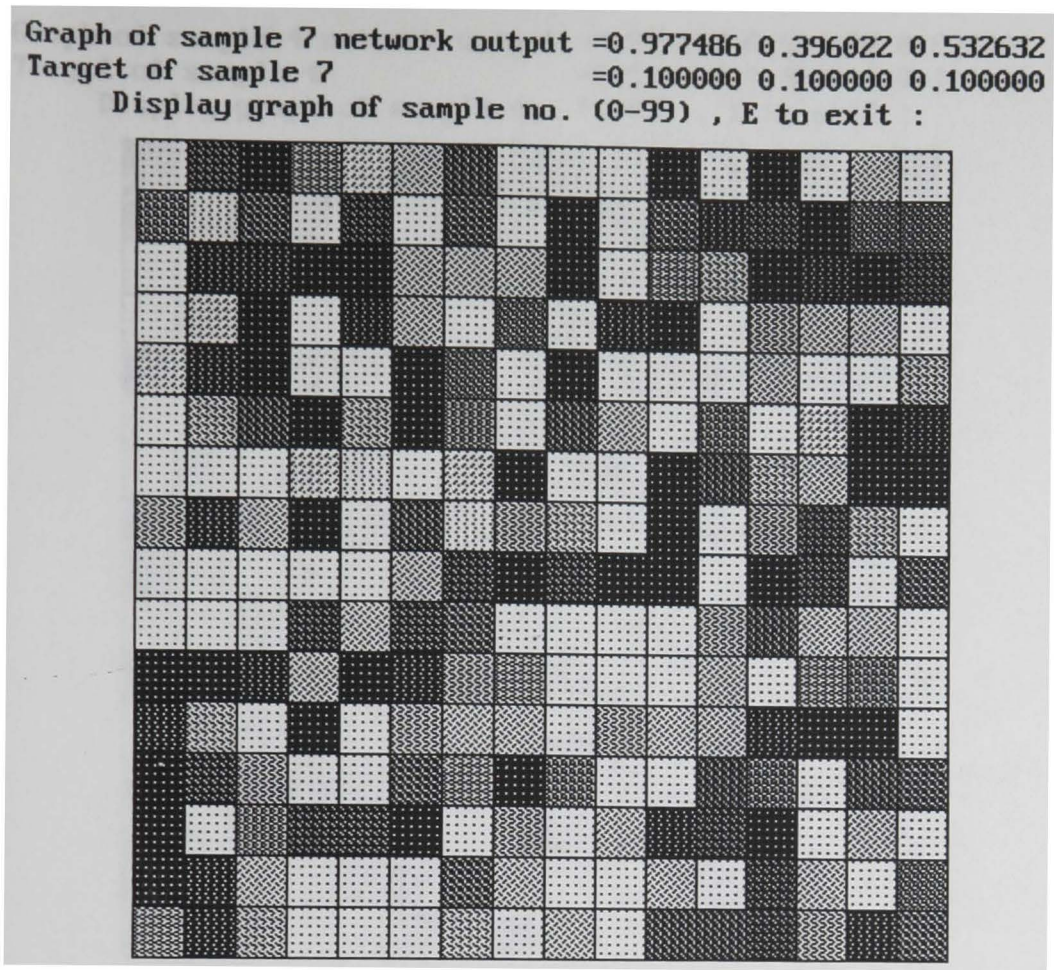


Figure 4.2 A 16X16 scene at a noise level of 0.8 (2)

Watching the right-upper corner in the Figure 4.2, there seems to be a horizontal target but nobody can be sure. The value of the NNW output node 0 is somewhat below 0.995 which means that the NNW shows there is no horizontal or vertical target. The other two output node values are far below 0.8, so no right-slant or left-slant target exists. In fact, because the values of all the target nodes are 0.1, there is no target in the scene. The NNW can detect there is no target in the scene.

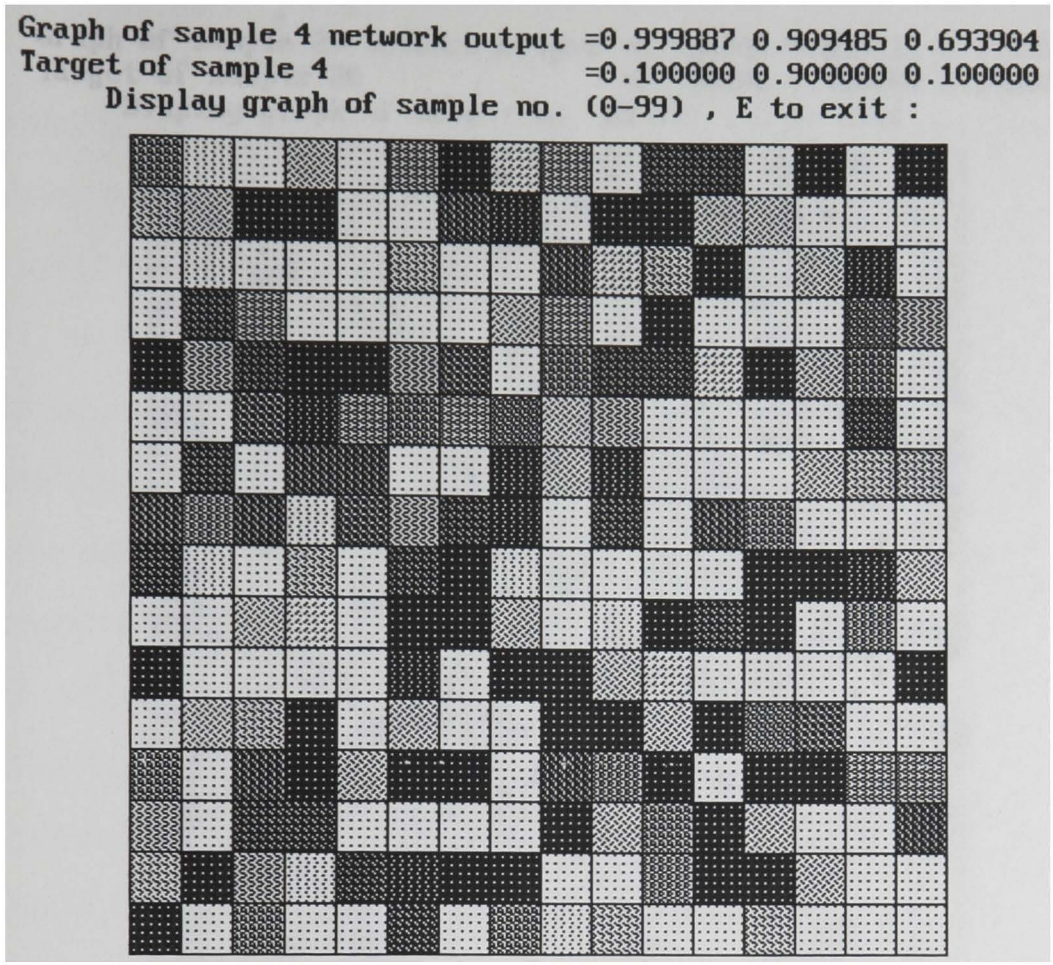


Figure 4.3 A 16X16 scene at a noise level of 0.8 (3)

In Figure 4.3, there looks like a horizontal target in the left-upper region and a right-slant target in the right-lower region. The value of the NNW output node 0 is above 0.995. This means that the NNW indicates there is a horizontal or vertical target. Because the value of node 1 is above 0.8, the NNW indicates there is a right-slant target. In fact, the original signal shows that the value of the target node 1 is 0.9 and the other target nodes are only 0.1. There is only a right-slant target in the scene. The NNW can detect the target in the scene.

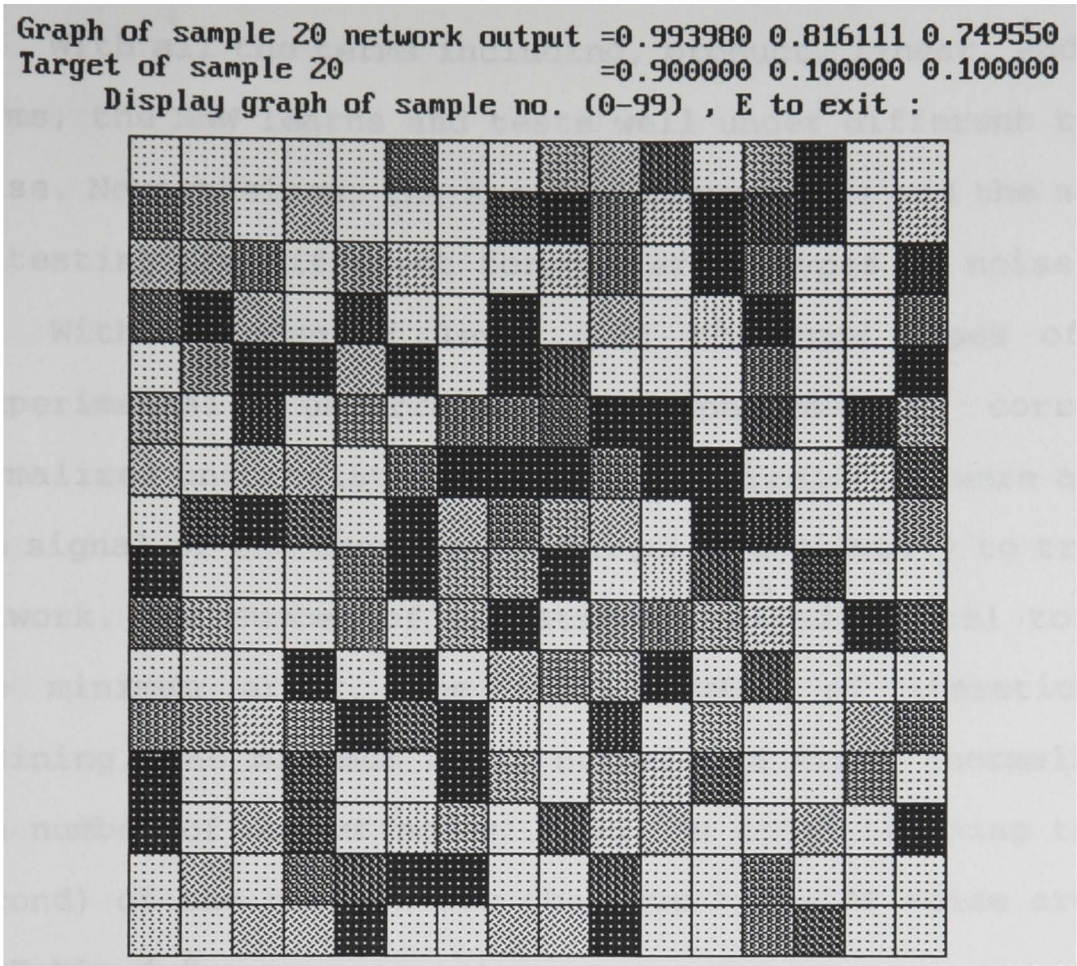


Figure 4.4 A 16X16 scene at a noise level of 0.8 (4)

In Figure 4.4, there seems to be a horizontal target and a right-slant target in the center of the scene. The value of the NNW output node 0 is just somewhat below 0.995 and this means that the NNW indicates there is no horizontal or vertical target. The value of node 1 is somewhat above 0.8, so the NNW indicates there is a right-slant target. In fact, the original signal shows that the value of the target node 0 is 0.9 and the other target nodes are only 0.1. There is only a horizontal or vertical target in the scene. The NNW can still indicate there is a target in the scene.

4.8 The Effect of Different Types of Noise

With all the terms including, product, linear, and square terms, the NNW learns and tests well under different types of noise. Nevertheless, the speed of convergence and the accuracy of testing are different for different types of noise.

With a noise of level 0.8, the four types of noise (experimental uncorrelated, experimental correlated, normalized uncorrelated, normalized correlated) were added to the signal of the rectangular target individually to train the network. The number of iterations (called counts) to obtain the minimum error, the total number of iterations for training, the minimum total normalized error (normalized by the number of output nodes), and the total training time (in second) of the network for the four types of noise are shown in Table 4.8.

Table 4.8 Training results of different types of noise

NNW Type	Iu8	In8	Ic8	Io8
min_err CNTs	1,273	3,178	3,907	5,000
total CNTs	1,273	3,178	3,907	5,000
min_err	0.001	0.001	0.001	0.001546
time(sec)	344	855	1,056	1,345

In Table 4.8, "I" denotes the NNW with product, linear, and square terms. The letter "u" denotes experimental uncorrelated noise and "n" denotes normalized uncorrelated noise. The letter "c" denotes experimental correlated noise and "o" denotes normalized correlated noise.

In Table 4.8, the convergence time for the uncorrelated noise cases is less than that of the correlated noise cases. For the signal corrupted with correlated noise, two horizontal adjacent pixels have the same noise. This seems to make the training time longer for the network. Whether the noise is uncorrelated or correlated, the convergence time of the experimental noise cases is less than that of the normalized noise cases. This is probably because the corrupted signal range of the normalized noise is in the range of (0,1). The experimental noise is larger and in the range of (0,2.25). The network seems to learn more easily with a larger range of signals. The convergent time for experimental uncorrelated noise is the smallest.

The total normalized error for the normalized correlated case still exceeds 0.001 when the number of iterations reaches 5000. The network seems to need more time to train for this case. Whether the noise is correlated or not, the normalized system error oscillates if the noise is experimental noise, and does not oscillate if the noise type is normalized noise. This indicates the normalized noise case is more difficult to learn.

The NNW with product, square, and linear terms was trained with a noise level of 0.8 by normalized uncorrelated, experimental correlated, and normalized correlated noise, individually. These cases are denoted by "In8," "Ic8," and "Io8," respectively. The test samples with a noise level of

0.8 and a scene size of 6X6 are generated by program GTC.C. The results are shown in Table 4.9.

Each trained network was tested with six different 100 sample test sets. The output threshold values were set to 0.5 for all 3 output nodes. In Table 4.9, "W" means there is a target in the scene and "O" means there is no target in the scene. The notation, "6" means the scene size is 6X6 and "8" means the noise level is 0.8. The letters, "n," "c," and "o" denote the noise type. The table entries indicate the percent of correct identifications.

The accuracies for both the "W" and "O" cases should approach or exceed 80%, for the network to be acceptable. Table 4.9 shows that none of the three cases is acceptable. Therefore, the NNW performs poorly in detecting targets under these 3 kinds of noise. However, the NNW is very accurate under experimental uncorrelated noise as shown in sections 4.3 and 4.6.

Table 4.9 Outputs of NNWs trained by 3 types of noise

NNW Type	In8	Ic8	Io8
W6n8	82	36	66
O6n8	25	77	25
Average	53.5	56.5	45.5
W6c8	78	76	100
O6c8	9	60	11
Average	43.5	68	55.5
W6o8	84	18	75
O6o8	16	94	33
Average	50	56	55

CHAPTER 5

CONCLUSION

This thesis developed a method to detect a two-dimensional target in a noisy environment by using a special kind of neural network architecture. The target detection problem originates from the study of Davis and Schmidt [5]. A second-order neural network is used that has both translation and 90-degree rotation invariances.

The target had a 5X2 constant size, but could be anywhere in a 16X16 picture scene. In addition, the background contained a substantial amount of a particular form of noise. Due to the sizes involved, the target loses its basic shape for any rotation other than 90 degrees. Therefore, only 90-degree rotation and translation invariance were employed. Back-propagation learning with the least mean square algorithm was used to train the NNW. The output function was a sigmoid.

Four variations of the second-order NNW were examined. Networks with product terms only, with product and square terms, with product and linear terms, and with product, square, and linear terms were used. Only the NNW with all three terms learned and tested well under noisy conditions for the rectangular target detection. The noise used was an experimental uncorrelated noise [5]. The noise levels used were 0, 0.6, 0.7, and 0.8.

The training scene size was limited by the size of the

target and chosen to be 6X6. A 6X6 moving window was used to cover the 16X16 scene. This approach reduces the size of the network and improves the convergence during training.

Both 6X6 scene samples and 16X16 scene samples corrupted with experimental uncorrelated noise of levels 0, 0.6, 0.7, and 0.8, were used to test the NNW trained with a corresponding noise level. The NNW, trained with experimental uncorrelated noise with noise levels of 0.6, 0.7, and 0.8, had above 80% accuracy when tested with the samples of corresponding levels of noise. This performance is better than the method presented by Davis and Schmidt [5]. In their paper, the noise level was only up to 0.7.

The NNW with all three terms was trained by four kinds of noise with level 0.8: experimental uncorrelated, normalized uncorrelated, experimental correlated, normalized correlated. The two types of uncorrelated noise had a small convergence time than that of the two types of correlated noise. The two types of experimental noise converged sooner than the two types of normalized noise. The convergence time for the experimental uncorrelated noise was the smallest. The accuracy of the NNW to recognize signals corrupted by noise was also highest for the experimental uncorrelated noise.

The results have shown that a special neural network architecture can be used to detect a two-dimensional rectangular target in a larger, noisy scene. The linear terms in the network, while not invariant, seem to contribute

substantially to reducing the effect of the noise. The moving window approach seems to work well in this case. However, the partial target problem makes the selection of threshold values for target identification critical.

The specific effect of the linear terms requires further study. A more complete comparison of prefiltering of the data before target detection would also be beneficial. Some unusual test results, Like Figure 4.3 and 4.4, show the NNW may classify the wrong features. This is because the NNW was not trained by "incomplete target" patterns. The ability to recognize different features can be enhanced by training with "incomplete target" samples.

In the future, the moving window can be replaced by parallel processing all the windows simultaneously. Thus, the speed to detect the target can increase rapidly. The 16X16 scene size can be extended to any size, e.g., 256X256, and the speed does not decrease. Therefore, the target can be detected in real time.

REFERENCES

[1] Michael W. Roth, "Survey of Neural Network Technology for Automatic Target Recognition," IEEE Transactions on Neural Networks, Vol.1, No.1, March 1990.

[2] Philip D. Wasserman, Tom Schwartz, "Neural Networks-Part 2," IEEE Expert, Spring 1988.

[3] Richard K. Miller, Neural Networks-Implementing Associative Memory Model in Neurocomputer, (chapter 3), The Fairmont Press, Inc. 1990.

[4] Teuvo Kohonen, "An Introduction to Neural Computing," Neural Networks, Vol.1, pp.3-16, 1988.

[5] Jon P. Davis and William A. Schmidt, "Use of Partial Templates with High-order Neural Nets for 2-D Target Detection," Naval Air Development Center, Warminster, PA 18974, Code 5012.

[6] Jon P. Davis and William A. Schmidt, "Local Correlations as a Translationally Invariant Feature Space for Target Detection," Naval Air Development Center, Warminster, PA 18974, Code 5012.

[7] Patric K. Simpson, Artificial Neural System: Foundations, Paradigms, Applications, and Implementations, Pergamon Press, Elmsfoid, NY, 1990.

[8] Yoh-Han Pao, Adaptive Pattern Recognition and Neural Networks, Addison-Wesley Publishing Company, Inc., Reading, MA, 1989.

[9] Richard P. Lippmann, "An Introduction to Computing with Neural Nets," IEEE ASSP Magazine, April, 1987.

[10] M. Minsky, S. Papert, Perceptions: An Introduction to Computational Geometry, MIT Press, Cambridge, MA, 1969.

[11] C. Lee Giles and Tom Maxwell, "Learning, Invariance, and Generalization in High-order Neural Networks," Applied Optics, Vol.26, No.23, Dec., 1987.

[12] Max B. Reid, Lilly Spirkovska, Ellen Ochoa, "Rapid Training of High-order Neural Networks for Invariant Pattern Recognition," Proceedings, Joint International Conference on Neural Network, Washington D.C. (June 1988).

[13] Tom Maxwell, C. Lee Giles, YC. Lee, "Generalization in Neural Networks: The Continuity Problem," In Proceedings, IEEE International Conference on Neural Network, San Diego (June 1987).

[14] Lilly Spirkovska and Max B. Reid, "Connectivity Strategies for Higher-order Neural Networks Applied to Pattern Recognition," Proceedings of the International Joint Conference on Neural Network, San Diego (June 1990).

[15] Yee-Man Kwan, "Target Detection Under Low Signal to Noise Ratio Conditions with Neural Networks," a thesis of Department of Electrical Engineering, Texas Tech University, May, 1991.

[16] C.L. Giles, R.D. Griffin, T. Maxwell, "Encoding Geometric Invariances in High-order Neural Networks," American Institute of Physics, 1988, pp.301-309.

APPENDIX A: GTC.C Simulation Program

```
/* GTC.C */
/* Generate 2D samples for neural network                          */
/* ! Because this program uses far pointer in the huge             */
/* memory, you have to leave Turbo C before running it.            */
/* targets attached at the end of each sample                      */
/* 0) samples for TC       1) samples for Target detect            */
/* samples can be added with noise level from 0 - 1                */
/* Two noise each one has two types                                */
/* 0) experimental noise   1) normalized noise                     */
/* Two types of noise                                              */
/* 0) un-correlated noise  1) correlated noise                     */

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <alloc.h>

#define  sam_no_max    100     /* Max. no. of samples       */
#define  sam_size       16     /* edge of a scene           */
#define  NMXOUT          3     /* Max. No. of output nodes  */

int samples, npts, nout, detect, xsize, ysize;
/* samples - no. of samples in the sample file.                    */
/* xnpts - no. of points in x edge of a scene  npts                */
/* ynpts - no. of points in y edge of a scene  npts                */
/* noiamp - noise amplitude 0-1                                    */
/* nout - no. of output nodes                                      */
/* detect - 1 for TC detect, 0 for Target detect                   */
/* xsize - the length of target                                    */
/* ysize - the height of target                                    */

float noiamp,(huge *sample)[sam_size][sam_size];
float target[sam_no_max][NMXOUT];
char pass[3];
FILE *fp1, *fp2;

main()
{
  char cont[2];
  sample = farcalloc(sizeof(*sample),sam_no_max);
   do {
     rddata();           /* input sample parameters          */
     gendata();          /* generate data                    */
     addnoise();         /* add noise to samples generated   */
     wtdata();           /* save samples to disk file        */
     printf("\nGenerate another set of data ? (y/n) ");
     scanf ("%s",&cont);
      }while(cont[0]=='y' || cont[0]=='Y');
```

```
}

 rddata()
 {
   clrscr();
   printf("   TC and Target Artificial data generation \n");
   printf("\nNo. of samples (for testing use a multiple of\
\10)");
   printf("\nfor TC learning npts==4 =>40 ");
   printf("\nfor Target learning npts=6 =>50");
   printf("\n<= %d) : ",sam_no_max);
   scanf ("%d",&samples);
   printf("\nNo. of points in edge of a scene(4 only TC)( 6\
\8 12 16 ):");
   scanf ("%d",&npts);
   printf("\nNo. of output node (2 for TC, 3 for Target):\
\");
   scanf ("%d",&nout);
   if(nout==2) detect=1;
   else detect=0;
   if(!detect){
      printf("\nInput xsize ( default 5) : ");   xsize=5;
      scanf ("%d",&xsize);
      printf("\nInput ysize ( default 2) : ");   ysize=2;
      scanf ("%d",&ysize);
   }
 }

 gendata()
 {
     int i,j,k,x0,y0,p,c,m,n,q;
     float temp;

  printf("\nEnter 0 for training data, 1 for testing data\
\with objective");
  printf("\n, 2 for testing data without objective : ");
  scanf ("%d",&c);
  for(i=0; i<samples; i++)
    for(j=0; j<npts; j++)
      for(k=0; k<npts; k++)
     sample[i][j][k]=0.0;

  if(detect){
   if(c==2){
     for(i=0;i<samples;i++)
       for(j=0;j<nout;j++)
      target[i][j]=0.1;
   } /* c==2 */
   else if(c==1){
    for(i=0;i<samples;i+=10){
/* initial all samples are 0 , and all targets are 0-0.1 */
    /* Do not use randomize(); */
```

```
    /* TC problem */
    /*  T        */
/* left-up T 0<=x0<=npts-3, 0<=y0<=npts-3  */    /*  X**  */
    temp=rand();  x0= temp*(npts-3)/32768+.5;    /*   *   */
    temp=rand();  y0= temp*(npts-3)/32768+.5;    /*   *   */
    sample[i][y0][x0]=1;  sample[i][y0][x0+1]=1;
    sample[i][y0][x0+2]=1;
    sample[i][y0+1][x0+1]=1;  sample[i][y0+2][x0+1]=1;
    target[i][0]=.9;     target[i][1]=.1;
/* left-up |- 0<=x0<=npts-3, 0<=y0<=npts-3  */   /*  X    */
    temp=rand();  x0= temp*(npts-3)/32768+.5;    /*  ***  */
    temp=rand();  y0= temp*(npts-3)/32768+.5;    /*  *    */
    sample[i+1][y0][x0]=1; sample[i+1][y0+1][x0]=1;
    sample[i+1][y0+1][x0+1]=1;
    sample[i+1][y0+1][x0+2]=1; sample[i+1][y0+2][x0]=1;
    target[i+1][0]=.9;   target[i+1][1]=.1;
/* right-down JL 2<=x0<=npts-1, 2<=y0<=npts-1 */ /*   *   */
    temp=rand();  x0= temp*(npts-3)/32768+2+.5;  /*   *   */
    temp=rand();  y0= temp*(npts-3)/32768+2+.5;  /*  **X  */
    sample[i+2][y0][x0]=1;  sample[i+2][y0][x0-1]=1;
    sample[i+2][y0][x0-2]=1;
    sample[i+2][y0-1][x0-1]=1;  sample[i+2][y0-2][x0-1]=1;
    target[i+2][0]=.9;   target[i+2][1]=.1;
/* right-down -| 2<=x0<=npts-1, 2<=y0<=npts-1 */ /*    *  */
    temp=rand();  x0= temp*(npts-3)/32768+2+.5;  /*  ***  */
    temp=rand();  y0= temp*(npts-3)/32768+2+.5;  /*    X  */
    sample[i+3][y0][x0]=1;  sample[i+3][y0-1][x0]=1;
    sample[i+3][y0-2][x0]=1;
    sample[i+3][y0-1][x0-1]=1;  sample[i+3][y0-1][x0-2]=1;
    target[i+3][0]=.9;   target[i+3][1]=.1;
    /* end of T */
    /* C        */
    /* left-up C 0<=x0<=npts-3, 0<=y0<=npta-3 */
    temp=rand();  x0= temp*(npts-3)/32768+.5;
    temp=rand();  y0= temp*(npts-3)/32768+.5;
    for(j=y0;j<y0+3 ;j++)
      for(k=x0;k<x0+3 ;k++)
      {sample[i+4][j][k]=1;   sample[i+5][j][k]=1;
       sample[i+6][j][k]=1;   sample[i+7][j][k]=1;
      }
    sample[i+4][y0+1][x0+1]=0;  sample[i+5][y0+1][x0+1]=0;
    sample[i+6][y0+1][x0+1]=0;  sample[i+7][y0+1][x0+1]=0;
    sample[i+4][y0][x0+1]=0;          /*  * *  */
    target[i+4][0]=.1;                /*  * *  */
    target[i+4][1]=.9; /* up */       /*  ***  */
    sample[i+5][y0+1][x0+2]=0;                  /*  ***  */
    target[i+5][0]=.1;                          /*  *    */
    target[i+5][1]=.9;/* right*/                /*  ***  */
    sample[i+6][y0+2][x0+1]=0;        /*  ***  */
    target[i+6][0]=.1;                /*  * *  */
    target[i+6][1]=.9;/* down */      /*  * *  */
```

```
    sample[i+7][y0+1][x0]=0;                              /*  ***  */
    target[i+7][0]=.1;                                    /*    *  */
    target[i+7][1]=.9; /* left */                         /*  ***  */

/* left-up T 0<=x0<=npts-3, 0<=y0<=npts-3  */            /*  X**  */
    temp=rand();  x0= temp*(npts-3)/32768+.5;             /*   *   */
    temp=rand();  y0= temp*(npts-3)/32768+.5;             /*   *   */
    sample[i+8][y0][x0]=1;  sample[i+8][y0][x0+1]=1;
    sample[i+8][y0][x0+2]=1;
    sample[i+8][y0+1][x0+1]=1;  sample[i+8][y0+2][x0+1]=1;
    target[i+8][0]=.9;     target[i+8][1]=.1;
/* left-up |- 0<=x0<=npts-3, 0<=y0<=npts-3  */           /*  X    */
    temp=rand();  x0= temp*(npts-3)/32768+.5;             /*  ***  */
    temp=rand();  y0= temp*(npts-3)/32768+.5;             /*  *    */
    sample[i+9][y0][x0]=1; sample[i+9][y0+1][x0]=1;
    sample[i+9][y0+1][x0+1]=1;
    sample[i+9][y0+1][x0+2]=1; sample[i+9][y0+2][x0]=1;
    target[i+9][0]=.9;   target[i+9][1]=.1;
    }/* i */
   }/* c==1 */
   else{ /* c==0 */
    i=0;
    for(j=0; j<npts-2; j++)
      for(k=0; k<npts-2; k++,i+=9){
    target[i][0]=.1;     target[i][1]=.1;
      /* sample order sensitive */
    x0=j;      y0=k;
/* left-up T 0<=x0<=npts-3, 0<=y0<=npts-3  */            /*  X**  */
                                                          /*   *   */
                                                          /*   *   */
    sample[i+1][y0][x0]=1;  sample[i+1][y0][x0+1]=1;
    sample[i+1][y0][x0+2]=1;
    sample[i+1][y0+1][x0+1]=1;  sample[i+1][y0+2][x0+1]=1;
    target[i+1][0]=.9;     target[i+1][1]=.1;
/* left-up |- 0<=x0<=npts-3, 0<=y0<=npts-3  */           /*  X    */
                                                          /*  ***  */
                                                          /*  *    */
    sample[i+2][y0][x0]=1; sample[i+2][y0+1][x0]=1;
    sample[i+2][y0+1][x0+1]=1;
    sample[i+2][y0+1][x0+2]=1; sample[i+2][y0+2][x0]=1;
    target[i+2][0]=.9;   target[i+2][1]=.1;
/* right-down JL 0<=x0<=npts-3, 0<=y0<=npts-3 */         /*  X*   */
                                                          /*   *   */
                                                          /*  ***  */
    sample[i+3][y0][x0+1]=1;    sample[i+3][y0+1][x0+1]=1;
    sample[i+3][y0+2][x0]=1;    sample[i+3][y0+2][x0+1]=1;
    sample[i+3][y0+2][x0+2]=1;
    target[i+3][0]=.9;   target[i+3][1]=.1;
/* right-down -| 0<=x0<=npts-3, 0<=y0<=npts-3 */         /*  X *  */
                                                          /*  ***  */
                                                          /*    *  */
```

```
  sample[i+4][y0][x0+2]=1;   sample[i+4][y0+1][x0]=1;
  sample[i+4][y0+1][x0+1]=1; sample[i+4][y0+1][x0+2]=1;
  sample[i+4][y0+2][x0+2]=1;
  target[i+4][0]=.9;   target[i+4][1]=.1;
  /* C         */
  sample[i+5][y0][x0]=1;  sample[i+5][y0][x0+2]=1;
  sample[i+5][y0+1][x0]=1;  sample[i+5][y0+1][x0+2]=1;
  sample[i+5][y0+2][x0]=1;  sample[i+5][y0+2][x0+1]=1;
  sample[i+5][y0+2][x0+2]=1;                    /*  X *   */
  target[i+5][0]=.1;                            /*  * *   */
  target[i+5][1]=.9;     /* up */               /*  ***   */

  sample[i+6][y0][x0]=1;  sample[i+6][y0][x0+1]=1;
  sample[i+6][y0][x0+2]=1;  sample[i+6][y0+1][x0]=1;
  sample[i+6][y0+2][x0]=1;  sample[i+6][y0+2][x0+1]=1;
  sample[i+6][y0+2][x0+2]=1;                    /*  X**   */
  target[i+6][0]=.1;                            /*  *     */
  target[i+6][1]=.9;     /* right */            /*  ***   */

  sample[i+7][y0][x0]=1;  sample[i+7][y0][x0+1]=1;
  sample[i+7][y0][x0+2]=1;  sample[i+7][y0+1][x0]=1;
  sample[i+7][y0+1][x0+2]=1;  sample[i+7][y0+2][x0]=1;
  sample[i+7][y0+2][x0+2]=1;                    /*  X**   */
  target[i+7][0]=.1;                            /*  * *   */
  target[i+7][1]=.9;     /* down */             /*  * *   */

  sample[i+8][y0][x0]=1;  sample[i+8][y0][x0+1]=1;
  sample[i+8][y0][x0+2]=1;  sample[i+8][y0+1][x0+2]=1;
  sample[i+8][y0+2][x0]=1;  sample[i+8][y0+2][x0+1]=1;
  sample[i+8][y0+2][x0+2]=1;                    /*  X**   */
  target[i+8][0]=.1;                            /*    *   */
  target[i+8][1]=.9;      /* left */            /*  ***   */
  }/* npts==4 => 32 with TC(4k) + 8 without TC(1K) = 5K */
  for(; i<samples; i++){
     target[i][0]=.1;    target[i][1]=.1;
  }
 }/* c==0 */
}/* detect */
else{ /* if target detect problem */
 if(c==2){
   for(i=0;i<samples;i++)
     for(j=0;j<nout;j++)
        target[i][j]=0.1;}/* c==2 */
 else if(c==1){
 for(i=0; i<samples; i+=10){
              /*  X****   0<=x0<=npts-xsize    */
              /*  *****   0<=y0<=npts-ysize    */
   temp=rand();   x0= temp*(npts-xsize)/32768+.5;
   temp=rand();   y0= temp*(npts-ysize)/32768+.5;
   for(j=y0; j< y0+ysize; j++)
     for(k=x0; k< x0+xsize; k++)
        sample[i][j][k]=1.0;
```

```
      target[i][0]=.9;          target[i][1]=.1;
      target[i][2]=.1;

                /*  X****   0<=x0<=npts-xsize    */
                /*  *****   0<=y0<=npts-ysize    */
      temp=rand();   x0= temp*(npts-xsize)/32768+.5;
      temp=rand();   y0= temp*(npts-ysize)/32768+.5;
      for(j=y0; j< y0+ysize; j++)
        for(k=x0; k< x0+xsize; k++)
       sample[i+1][j][k]=1.0;
      target[i+1][0]=.9;          target[i+1][1]=.1;
      target[i+1][2]=.1;

      temp=rand();   x0= temp*(npts-ysize)/32768+.5;
      temp=rand();   y0= temp*(npts-xsize)/32768+.5;
      for(j=y0; j< y0+xsize; j++)
        for(k=x0; k< x0+ysize; k++)
          sample[i+2][j][k]=1.0;  /* X*                        */
                                  /* **  0<=x0<=npts-ysize */
      target[i+2][0]=.9;          /* **                        */
      target[i+2][1]=.1;          /* **  0<=y0<=npts-xsize */
      target[i+2][2]=.1;          /* **                        */

      temp=rand();   x0= temp*(npts-ysize)/32768+.5;
      temp=rand();   y0= temp*(npts-xsize)/32768+.5;
      for(j=y0; j< y0+xsize; j++)
        for(k=x0; k< x0+ysize; k++)
          sample[i+3][j][k]=1.0;  /* X*                        */
                                  /* **  0<=x0<=npts-ysize */
      target[i+3][0]=.9;          /* **                        */
      target[i+3][1]=.1;          /* **  0<=y0<=npts-xsize */
      target[i+3][2]=.1;          /* **                        */


      temp=rand();   x0= temp*(npts-xsize-ysize+1)/32768+.5;
      temp=rand();   y0= temp*(npts-xsize)/32768+.5;
      for(p=0; p<xsize; p++)
       for(k=x0+p,j=y0+p;k<x0+p+ysize;k++)
         sample[i+4][j][k]=1.0;
                        /* X*                                */
                        /*  **   0<=x0<npts-xsize-ysize+2  */
      target[i+4][0]=.1;/*   **                             */
      target[i+4][1]=.9;/*    **                            */
      target[i+4][2]=.1;/*     **  0<=y0<npts-xsize+1       */


/* 90 degrees rotation of above
      temp=rand();   x0= temp*(npts-xsize)/32768+xsize-1+.5;
      temp=rand();   y0= temp*(npts-xsize-ysize+1)/32768+.5;
      for(p=0; p<xsize; p++)
        for(k=x0-p, j=y0+p; j<y0+p+ysize;j++)
          sample[i+5][j][k]=1.0;
```

```
    xsize-2<x0<=npts-1                              X
     target[i+5][0]=.1;                            **
     target[i+5][1]=.9;                           **
     target[i+5][2]=.1;                          **
   0<=y0<npts-xsize-ysize+2                     **
                                                *
*/

     temp=rand();
     x0= temp*(npts-xsize-ysize+1)/32768+xsize+ysize-2+.5;
     temp=rand();   y0= temp*(npts-xsize)/32768+.5;
     for(p=0; p<xsize; p++)
       for(k=x0-ysize-p+1, j=y0+p; k<x0-p+1; k++)
         sample[i+5][j][k]=1.0;
                       /*     *X xsize+ysize-2<=x0<=npts-1*/
     target[i+5][0]=.1;/*    **                           */
     target[i+5][1]=.1;/*   **                            */
     target[i+5][2]=.9;/*  **   0<=y0<npts-xsize+1        */
                       /* **                              */


/* 90 degrees rotation of above
     temp=rand();   x0= temp*(npts-xsize)/32768+.5;
     temp=rand();   y0= temp*(npts-xsize-ysize+1)/32768+.5;
     for(p=0; p<xsize; p++)
       for(k=x0+p,j=y0+p;j<y0+p+ysize;j++)
         sample[i+7][j][k]=1.0;              X
   0<=x0<npts-xsize+1                       **
     target[i+7][0]=.1;                      **
     target[i+7][1]=.1;                       **
     target[i+7][2]=.9;                        **
   0<=y0<npts-xsize-ysize+2                     *
*/

     temp=rand();   x0= temp*(npts-xsize-ysize+1)/32768+.5;
     temp=rand();   y0= temp*(npts-xsize)/32768+.5;
     for(p=0; p<xsize; p++)
      for(k=x0+p,j=y0+p;k<x0+p+ysize;k++)
        sample[i+6][j][k]=1.0;
                       /* X*                              */
                       /*  **   0<=x0<npts-xsize-ysize+2  */
     target[i+6][0]=.1;/*   **                           */
     target[i+6][1]=.9;/*    **                          */
     target[i+6][2]=.1;/*     ** 0<=y0<npts-xsize+1       */


     temp=rand();
     x0= temp*(npts-xsize-ysize+1)/32768+xsize+ysize-2+.5;
     temp=rand();   y0= temp*(npts-xsize)/32768+.5;
     for(p=0; p<xsize; p++)
       for(k=x0-ysize-p+1, j=y0+p; k<x0-p+1; k++)
         sample[i+7][j][k]=1.0;
```

```
                        /*      *X xsize+ysize-2<=x0<=npts-1*/
    target[i+7][0]=.1;/*     **                              */
    target[i+7][1]=.1;/*    **                               */
    target[i+7][2]=.9;/*   **    0<=y0<npts-xsize+1          */
                        /* **                                */

    temp=rand();    x0= temp*(npts-xsize-ysize+1)/32768+.5;
    temp=rand();    y0= temp*(npts-xsize)/32768+.5;
    for(p=0; p<xsize; p++)
     for(k=x0+p,j=y0+p;k<x0+p+ysize;k++)
       sample[i+8][j][k]=1.0;
                        /* X*                               */
                        /*  **    0<=x0<npts-xsize-ysize+2  */
    target[i+8][0]=.1;/*    **                              */
    target[i+8][1]=.9;/*     **                             */
    target[i+8][2]=.1;/*      ** 0<=y0<npts-xsize+1         */

    temp=rand();
    x0= temp*(npts-xsize-ysize+1)/32768+xsize+ysize-2+.5;
    temp=rand();    y0= temp*(npts-xsize)/32768+.5;
    for(p=0; p<xsize; p++)
      for(k=x0-ysize-p+1, j=y0+p; k<x0-p+1; k++)
        sample[i+9][j][k]=1.0;
                        /*      *X xsize+ysize-2<=x0<=npts-1*/
    target[i+9][0]=.1;/*     **                              */
    target[i+9][1]=.1;/*    **                               */
    target[i+9][2]=.9;/*   **    0<=y0<npts-xsize+1          */
    }/* i */            /* **                               */
    }/* c==1 */

   else{ /* c==0 */
   i=0;
   for(q=0; q<5; q++)
    for (m=0; m<npts-xsize-ysize+2; m++)
     for (n=0; n<npts-xsize+1; n++, i+=3){
          /* sample number sensitive */
     target[i][0]=.1;  target[i][1]=.1;   target[i][2]=.1;
          /*order sensitive*/


     x0=m;    y0=n;
     for(p=0; p<xsize; p++)
      for(k=x0+p,j=y0+p;k<x0+p+ysize;k++)
        sample[i+1][j][k]=1.0;
                        /* X*                               */
                        /*  **    0<=x0<npts-xsize-ysize+2  */
     target[i+1][0]=.1;/*    **                             */
     target[i+1][1]=.9;/*     **                            */
     target[i+1][2]=.1;/*      **  0<=y0<npts-xsize+1       */


/* 90 degrees rotation of above
```

```
      x0=n;  y0=m;
      x0=x0+xsize-1;
      for(p=0; p<xsize; p++)
        for(k=x0-p, j=y0+p; j<y0+p+ysize;j++)
          sample[i+2][j][k]=1.0;
    xsize-1<=x0<=npts-1                          X
      target[i+2][0]=.1;                         **
      target[i+2][1]=.1;                        **
      target[i+2][2]=.9;                       **
    0<=y0<npts-xsize-ysize+2                  **
                                              *                */
      x0=m;   y0=n;
      x0=x0+xsize+ysize-2;
      for(p=0; p<xsize; p++)
        for(k=x0-ysize-p+1, j=y0+p; k<x0-p+1; k++)
          sample[i+2][j][k]=1.0;
                        /*      *X xsize+ysize-2<=x0<=npts-1*/
      target[i+2][0]=.1;/*     **                           */
      target[i+2][1]=.1;/*    **                            */
      target[i+2][2]=.9;/*   **   0<=y0<npts-xsize+1       */
                        /* **                               */

/* 90 degrees rotation of above
      x0=n;   y0=m;
      for(p=0; p<xsize; p++)
        for(k=x0+p,j=y0+p;j<y0+p+ysize;j++)
       sample[i+4][j][k]=1.0;                  X
    0<=x0<npts-xsize+1                         **
      target[i+4][0]=.1;                        **
      target[i+4][1]=.9;                         **
      target[i+4][2]=.1;                          **
    0<=y0<npts-xsize-ysize+2                       *          */
      }/* \ / */

    for(m=0; m<npts-xsize+1; m++)
      for(n=0; n<npts-ysize+1; n++, i+=2){
      x0=m;   y0=n;
                 /*  X****   0<=x0<=npts-xsize    */
                 /*  *****   0<=y0<=npts-ysize    */
      for(j=y0; j< y0+ysize; j++)
        for(k=x0; k< x0+xsize; k++)
       sample[i][j][k]=1.0;
      target[i][0]=.9;          target[i][1]=.1;
      target[i][2]=.1;
      x0=n;   y0=m;
      for(j=y0; j< y0+xsize; j++)
        for(k=x0; k< x0+ysize; k++)
          sample[i+1][j][k]=1.0;  /* X*                      */
                                  /* **  0<=x0<=npts-ysize */
      target[i+1][0]=.9;          /* **                      */
      target[i+1][1]=.1;          /* **  0<=y0<=npts-xsize */
      target[i+1][2]=.1;          /* **                      */
```

```
     }/* ||   =   */
/* npts==6 =>40 with target + 10 without target = 50 ->15k*/
   for(; i<samples; i++){
      target[i][0]=.1;   target[i][1]=.1;   target[i][2]=.1;
   }
   }/* c==0 */
  }/* detect */
 }

 addnoise()
 {
    int i,y,x,sign,noisea,noisetype,k;
    float random_noise,temp;
    char noise[2];
    randomize();
    printf("\nAdd noise to the sample (y/n) ? ");
    scanf ("%s",&noise);
    if (noise[0]=='y'|| noise[0]=='Y'){
      printf("\n0 - experimental noise, 1 - normalize noise\
\: ");
      scanf ("%d",&noisea);
      printf("\nType of noise : 0 - uncorrelated,   1 -\
\correlated : ");
      scanf ("%d",&noisetype);
      printf("\nInput the noise amplitude (0-1) : ");
      scanf("%f",&noiamp);
      for (i=0; i<samples; i++)
      {
        if(noisetype==0){
        for (y=0; y<npts; y++)
          for (x=0; x<npts; x++)
          {
          if ( ( (float) rand()/RAND_MAX ) > 0.5)
             sign = 1;
          else
             sign =-1;
        /*      random_noise = 1.0;
          while (random_noise > noiamp)                    */
            random_noise = (float) rand()/RAND_MAX;
          temp=1-exp(-(random_noise/.5)*(random_noise)/.5);
          if(noisea==1) { temp=-1*noiamp*temp;}
          else  { temp=.25+sign*noiamp*temp;  }
          temp+=sample[i][y][x];
/*                  if(temp>1)  temp=1.0;      */
          if(temp<0){
            if(noisea==1)  temp=-1*temp;
            else  temp=0.0;
          }
          sample[i][y][x]=temp;
          }/* for y , for x */
        }
        else{
```

```
        for (y=0; y<npts; y++){
          if ( ( (float) rand()/RAND_MAX ) > 0.5){
            k=1;
            for(x=0; x<npts; x+=npts-1){
          if ( ( (float) rand()/RAND_MAX ) > 0.5)
             sign = 1;
          else
             sign =-1;
        /*      random_noise = 1.0;
          while (random_noise > noiamp)                    */
            random_noise = (float) rand()/RAND_MAX;
          temp=1-exp(-(random_noise/.5)*(random_noise)/.5);
          if(noisea==1) { temp=-1*noiamp*temp;}
          else  { temp=.25+sign*noiamp*temp;  }
          sample[i][y][x]+=temp;
          if(sample[i][y][x]<0){
            if(noisea==1)  sample[i][y][x] *= -1;
            else  sample[i][y][x]=0.0;
          }
            }/* for x */
          } /* if => k=1  */
          else      k=0;
          for (x=k; x<npts-k; x+=2)
          {
          if ( ( (float) rand()/RAND_MAX ) > 0.5)
             sign = 1;
          else
             sign =-1;
        /*      random_noise = 1.0;
          while (random_noise > noiamp)                    */
            random_noise = (float) rand()/RAND_MAX;
          temp=1-exp(-(random_noise/.5)*(random_noise)/.5);
          if(noisea==1) { temp=-1*noiamp*temp;}
          else  { temp=.25+sign*noiamp*temp;  }
          sample[i][y][x]+=temp;

          if(sample[i][y][x]<0){
            if(noisea==1)  sample[i][y][x] *= -1;
            else  sample[i][y][x]=0.0;
          }
          sample[i][y][x+1]+=temp;
          if(sample[i][y][x+1]<0){
            if(noisea==1)  sample[i][y][x+1] *= -1;
            else  sample[i][y][x+1]=0.0;
          }
          }/* for x */
        }/* for y */
        }/* noisetype */
      }/* i */
    }
}
```

```
wtdata()
{
  char filename[32];
  int i,y,x, k;
  printf("\nInput file name at most 7 characters (without\
\extension): ");
  scanf ("%s",&filename);
  strcat(filename,".dat");
  printf("\nInclude target at the end of each sample\
\vector");
  if ((fp1 = fopen(filename,"w+")) ==NULL)
    {
    printf("\nCannot open data file ");
    exit(0);
    }
  for (i=0; i<samples; i++)
    {
    k=0;
    for(y=0;y<npts;y++)
    for (x=0; x<npts; x++) {
      if (k==8) {
        k=0;
        fprintf(fp1,"\n");
      }
      fprintf(fp1,"%f ",sample[i][y][x]);
        k++;
    }
    fprintf(fp1,"\n");
    for(k=0; k<nout; k++)
      fprintf(fp1,"%f ",target[i][k]);
    fprintf(fp1,"\n");
    }
  rewind(fp1);
  if ((fclose(fp1)) !=0)
    printf("\nFile cannot be closed %s",filename);
}
```

APPENDIX B: IHNW.C Simulation Program

```
/*****************    Revision 2-2-93      ****************/
/**  ihnw.c
     Program Description:
     This program contains IHNNWs
     !  Because it uses far pointer in the huge memory,
     you have to leave Turbol C before running this program.

 *   1. 90 degrees Rotation and Translation invariant
     High order ( order = 2 )
     Back propagation neural network
     for 2 Dimensional Target Detect

     This program allows a user to build a generalized
     delta rule net for supervised learning. User can
     specify the number of input square root and output
     units.

     After the net is built, learning takes place in the
     net with a given set of training samples. User
     specifies values of the learning rate eta, the
     momentum rate alpha, maximum tolerance errors and
     maximum number of iterations.

     After learning, all the information relevant to the
     structure of the net, including weights and
     thresholds are stored in files.

     Outputs can be generated for new patterns by reading
     from file and by reconstructing the net.

     Training set samples and additional samples for
     processing are stored in files.

     The network can be configured to contain :
     a) product terms only,
     b) product and square terms only
     c) linear and product terms only, and
     d) linear, square and product terms

     The trained network data is saved in
     a) xxd.dat - holds network structure data
     b) xxw.dat - holds network weight data
     c) **o.dat - output generated by the trained network
                  xx.dat from input sample file **.dat
     d) xxe.dat - error file by training xx.dat
       where xx is the file name of training samples
*/
```

```
#include <stdio.h>
#include <math.h>
#include <graphics.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <alloc.h>

/* define constants used throughout functions      */
#define  NMXOATTR    3 /* max. no. of output features     */
#define  NMXINP    100 /* max. no. of input samples, 200K */
#define  NMXIATTR    6 /* max. no. of pts. in the edge of a
                          learning scene */
#define  NMXSCENE   16 /* max. no. of pts. in the edge of a
                          testing scene */
#define  SEXIT       3 /* exit successfully               */
#define  RESTRT      2 /* restart                         */
#define  FEXIT       1 /* exit in failure                 */
#define  CONTNE      0 /* continue calculation            */
#define  NMXPR_WT   19 /* max. no. of pr_wt by 6X6 scene for
                          1 output*/

/* 6K memory for product values  */
/*  Data base: declarations of variables    */
float eta;        /** learning rate              **/
float alpha;      /** momentum rate              **/
float err_curr;   /** normalized system error    **/
float maxe;       /** max allowed system error   **/
float maxep;      /** max allowed pattern error  **/
int weint;       /** initial weight values as 0 or random **/
int yfunc;    /* operation on output sum - choice of three */
int weights; /* number of terms for the higher order net  */
float bias[NMXOATTR], lin_wt[NMXOATTR][NMXIATTR][NMXIATTR];
float sq_wt[NMXOATTR];
float sq[NMXINP];
struct pwt{                          /* bias, linear weight, */
       int dq;             /* square weight, product weight */
       float wt[NMXOATTR];
             /* dq:square of distance          wt:weight */
       } pr_wt[NMXPR_WT];
float (huge *pr_val)[NMXPR_WT];
      /* val: sum of input product of the same distance */
float theta[NMXOATTR]; /* for sigmoid use */
float err[NMXOATTR];
float del_bias[NMXOATTR];
float delin_wt[NMXOATTR][NMXIATTR][NMXIATTR];
float delsq_wt[NMXOATTR], delpr_wt[NMXOATTR][NMXPR_WT];
float deltheta[NMXOATTR];
float target[NMXINP][NMXOATTR];
float (huge *input)[NMXSCENE][NMXSCENE];
float ep[NMXINP];
float output[NMXINP][NMXOATTR];
```

```
float out[NMXOATTR];
int nscene; /* no. of pts. in the edge of a testing scene */
int xstart, ystart;
    /* x, y starting point of a block in a testing scene */
int   ninput, ninattr, noutattr, nprwt;
int   result, cnt, cnt_num;
int   nsnew, nsample;     /* nsample <= ninput */
char  task_name[20];
FILE *fp1, *fp2, *fp3, *fopen();
int   fplot10;   /* for showing training set data */
unsigned int  ts,tl,y,tk;
                /* ts=start time  tk=last training time
                 * tl=loop time   y=y pos on screen */
int   min_iter;  /* iteration at which min err occurs */
float minerr;    /* holds mininmum error */
int   invtype;   /* invtype holds invariant network type :
              0 - product terms only
              1 - linear and product terms
              2 - product and square terms
              3 - linear, square, and product term      */
int  relay; /* 1 for relay training, 0 for first training */
int  exist;
       /*1 for existing target or TC, 0 for trivial data */

/************************  MAIN  ************************/

main()
{
    char select[2], cont[10], showgraph[2];

    pr_val = farcalloc(sizeof(*pr_val),NMXINP);
    input = farcalloc(sizeof(*input),NMXINP);
    strcpy(task_name, "*********");
    do {
     clrscr();
     printf("\n\n ** Select \n\n       L (earning) ");
     printf("\n            O  (utput  generation  and  graph\
\display)");
     printf("\n       D (isplay sample set graphically)");
     printf("\n\n       option : ");
     do {
         scanf("%s", select);
         switch(select[0]) {
            case 'o':
            case 'O':
            output_generation ();
            result=1;            /* display of output */
            printf("\n\n See samples graphically (y/n) ? ");
            scanf ("%s",showgraph);
            if (showgraph[0]=='y' || showgraph[0]=='Y')
                graph();
            break;
```

```
            case 'l':
            case 'L':
            learning();
            break;
            case 'd':
            case 'D':
            result=0;              /* temp variable */
                              /* for target display */
            strcpy(showgraph,"Y");
            do{
               grddata();        /* read data for graphing */
               graph();          /* display graph          */
               printf("\n See another sample set ? (Y/N) :\
\");
               scanf ("%s",showgraph);
            }while (showgraph[0]=='y'|| showgraph[0]=='Y');
            break;
            default:
            printf("\nanswer learning or output generation\
\");
            break;
         }
      } while ((select[0]!='o')&&(select[0]!='O')
            && (select[0]!='l')&&(select[0]!='L')
            && (select[0]!='d')&&(select[0]!='D'));
      printf("\nDo you want to continue? (Y/N) : ");
      scanf("%s",cont);
    } while ((cont[0]=='y') || (cont[0]=='Y'));

    printf("\nIt is all finished.. ");
    printf("\n Good bye  ");
}



/***************** main body of learning *****************/

learning()
{
   int result;

   user_session();
   set_up();
   do {
     if(!relay)  initwt();
     result = rumelhart(0,ninput);
   } while (result == RESTRT);
   if (result == FEXIT){
       printf("\n Max number of iterations reached,");
       printf("\n but failed to decrese system");
       printf("\n error sufficiently");
   }
```

```
      dwrite(task_name);
      wtwrite(task_name);
}


learnscr()
{
   char netype[20],ofn[20];  /* ofn - output funtion name */
   char invmsg[25];          /* invariant net type message */
   int i;

   clrscr();
      strcpy(netype,"Invariant");
      if (invtype == 0) strcpy(invmsg,"product terms only");
      if (invtype == 1) strcpy(invmsg,"linear & product\
\terms");
      if (invtype == 2) strcpy(invmsg,"square & product\
\terms");
      if (invtype == 3) strcpy(invmsg,"lin, sq. & product\
\terms");
      printf("\t      Learning with invariant network\
\(%s)",invmsg);

   printf("\n\n Sample file name                    : %s\
\",task_name);
   printf("\n No. of points in the edge of a scene :\
\%d",ninattr);
   printf("\n No. of output units                  :\
\%d",noutattr);
   printf("\n No. of input sample set              :\
\%d",ninput);
   printf("\n\n Learning rate          : %f",eta);
   printf("\n Momentum rate          : %f",alpha);
   printf("\n System error is set to : %f",maxe);
   printf("\n Individual error       : %f\n",maxep);
   if (weint==0)         printf("\n Initial weights : 0.0");
   else  printf("\n Initial weights : random");

   if (yfunc==0) strcpy(ofn,"sgn");
   if (yfunc==1) strcpy(ofn,"tanh");
   if (yfunc==2) strcpy(ofn,"sigmoid");
   printf("\n Output function : %s \n\n",ofn);
   ts=time();
   if(!relay){
   cnt=0;
   err_curr=0.0;
   minerr=1;
   tk=0;
   }
   y=wherey();
   gotoxy(2,y);    printf("Iteration : %d",cnt);
   gotoxy(21,y);   printf("error : %f",err_curr);
```

```
   gotoxy(42,y);   printf("min. err : %f",minerr);
   gotoxy(67,y);   printf("time : %d",tk);
   printf("\n Min. error occurs at iteration : ");
   gotoxy(35,y+1); printf("%d",min_iter);
   printf("\n No. of iterations set   : %d    Processing,\
\wait!",cnt_num);
}


/********** initialize weights with random numbers
            between -0.5 and +0.5 ********************/
initwt()
{
    int i,j,k;

    if (invtype == 0) weights = (1+nprwt)*noutattr;
    if (invtype == 1)
       weights = (1+nprwt+ninattr*ninattr)*noutattr;
    if (invtype == 2) weights = (2+nprwt)*noutattr;
    if (invtype == 3)
       weights = (2+nprwt+ninattr*ninattr)*noutattr;
    if (yfunc==2){
      for (i=0; i<noutattr; i++){
       theta[i] = 0.5;   bias[i] = 1.0;
      }
    }
    if (weint) {
       for (i=0; i<noutattr; i++) {
      bias[i] = rand()/32768 + 0.5;
      if(invtype == 2 || invtype == 3){
      sq_wt[i] = rand()/pow(2.0,15.0) - 0.5;
      }
       }
       if(invtype == 1 || invtype == 3){
      for (i=0; i<noutattr ; i++)
        for (j=0; j<ninattr ; j++)
          for (k=0; k<ninattr ; k++)
            lin_wt[i][j][k] = rand()/pow(2.0,15.0) - 0.5;
       }
       for (i=0; i<nprwt ; i++)
      for (j=0; j<noutattr ; j++)
        pr_wt[i].wt[j] = rand()/pow(2.0,15.0) - 0.5;
    }
}


     /************** specify architecture of net and
        values of learning parameters ****************/
set_up()
{
  int i;
  char ans[2];
```

```
  eta = 0.1;
  printf("Learning rate eta (default = 0.1)?: ");
  scanf("%f",&eta);

  alpha = 0.01;
  printf("Momentum rate alpha (default = 0.00)?: ");
  scanf("%f",&alpha);

  maxe = 0.001;
  maxep = 0.0001; /* error per sample */
  printf("\nMax total error (default = 0.001)?: ");
  scanf("%f",&maxe);
  printf("Max individual error (default = 0.0001)?: ");
  scanf("%f",&maxep);

  if(relay){
    printf("Now cnt is %d, enter a number grater than\
\%d\n",cnt,cnt);
  }
  printf("Max number of iterations cnt_num (default = 1000)\
\> cnt?: ");
  scanf("%d", &cnt_num);

  if(!relay){
  weint=0;
  printf("Initial weights (0 or 1 for random)?: ");
  scanf("%d", &weint);

  printf("Output fn   y = f(x),Sgn() = 0   Tanh() = 1  \
\Sigmoid() = 2");
  printf("   Function type?:  ");
  scanf("%d", &yfunc);

  printf("hnnw type : 90 degrees rotation & translation\
\Invariant, order 2");

  printf("\nProduct terms only = 0       Linear & Product\
\terms  = 1");
  printf("\nProduct and square = 2      Lin, Sq & Product\
\terms = 3");
  printf("\n  option (0/1/2/3) : ");
  scanf ("%d",&invtype);
  }/* !relay */
  printf("Create error file?  If so type 1, or type 0 : ");
  scanf("%d",&fplot10);
  xstart=0;
  ystart=0;
  nscene=ninattr;
  learnscr();
}
```

```
    /************* read file for net architecture and
                learning parameters. File name has
                suffice d.dat ********************/

dread (taskname)
char *taskname;
{
   int i,c;
   char var_file_name[20];

   strcpy(var_file_name, taskname);
   strcat(var_file_name, "d.dat");

   if(( fp1 = fopen(var_file_name,"r"))==NULL){
     perror("\n Cannot open data file ");
     exit(0);
   }
   fscanf(fp1,"%d%d%d%f%f%d%d",&ninput,&noutattr,
      &ninattr,&eta,&alpha,&cnt_num,&nprwt);
   fscanf(fp1,"%d%d%d%f%f%f%d",&invtype,&yfunc,
       &weint, &maxe,&maxep,&minerr,&min_iter);
   fscanf(fp1,"%d %f %u",&cnt,&err_curr,&tk);
   rewind(fp1);

   if ((c=fclose(fp1)) != 0)
       printf("\nFile cannot be closed in dread %d",c);
}

       /********************** read file containing weights
             and thresholds. File name has
             suffix w.dat ****************************/

wtread(taskname)
char *taskname;
{
   int i, j, k;
   char wt_file_name[20];

   strcpy(wt_file_name, taskname);
   strcat(wt_file_name,"w.dat");

   if ((fp2 = fopen(wt_file_name,"r")) == NULL){
       perror("\n Cannot open data file ");
       exit(0);
   }
   /* product weight */
   /* Only when the distance of any two points is equal to
      that of a certain strcuture, the product value of
      those two points' input values will be added to the
      "val" field of that structure.                    */

    if (yfunc==2){
```

```
        for (i=0; i<noutattr; i++)
        fscanf(fp2,"%f",&theta[i]);
        /* for sigmoid yfunc use only */
      }

      for (i=0; i<noutattr; i++)
         fscanf(fp2,"%f",&bias[i]);

      for (i=0; i<noutattr ; i++)
        for (j=0; j<ninattr ; j++)
          for (k=0; k<ninattr ; k++)
            fscanf(fp2,"%f",&lin_wt[i][j][k]);

      for (i=0; i<noutattr; i++)
        fscanf(fp2,"%f",&sq_wt[i]);

      for (i=0; i<nprwt ; i++){
        fscanf(fp2,"%d",&pr_wt[i].dq);
        for (j=0; j<noutattr ; j++)
          fscanf(fp2,"%f",&pr_wt[i].wt[j]);
      }
      rewind(fp2);

    if ((fclose(fp2)) != 0)
    printf("\nFile   can't   be   closed   in   wtread   for\
\%s",wt_file_name);
}

        /*************** create file for net architecture
                and learning parameters. File name
                has suffice d.dat  **********************/

dwrite(taskname)
char *taskname;
{
  int i, j;
  char var_file_name[20];

  strcpy(var_file_name,taskname);
  strcat(var_file_name, "d.dat");

  if ((fp1 = fopen(var_file_name,"w+")) == NULL){
      perror("Cannot open data file ");
      exit(0);
  }
  fprintf(fp1,"%u %u %u %f %f %u %u\n",ninput,noutattr,
          ninattr,eta,alpha,cnt_num,nprwt);

  fprintf(fp1,"%u %u %u %f %f %f %u\n",invtype,yfunc,
          weint,maxe,maxep,minerr,min_iter);

  fprintf(fp1,"%d %f %u\n", cnt,err_curr,tl);
```

```
  for (i=0; i<ninput; i++){
       for (j=0; j<noutattr; j++)
        fprintf(fp1,"%f   %f   ",output[i][j],target[i][j]);
        fprintf(fp1,"\n");
  }
  rewind(fp1);
     if ((fclose(fp1)) != 0)
     printf("\nFile cannot be closed %s",var_file_name);
}

     /*************** create file for saving weights and
             thresholds learned from training.
             File name has suffix  w.dat ******************/

wtwrite(taskname)
char *taskname;
{
    int i, j, k,m;
    char wt_file_name[20];

    strcpy(wt_file_name,taskname);
    strcat(wt_file_name, "w.dat");

    if ((fp2 = fopen(wt_file_name,"w+")) == NULL){
     perror("Cannot open data file ");
     exit(0);
    }
    if (yfunc==2){
       for (i=0; i<noutattr; i++)
      fprintf(fp2,"%f ",theta[i]);
      /* for sigmoid yfunc use only */
       fprintf(fp2,"\n");
    }

    for (i=0; i<noutattr; i++)
      fprintf(fp2,"%f ",bias[i]);

    fprintf(fp2,"\n");

    for (i=0; i<noutattr ; i++){
      m=0;
      for (j=0; j<ninattr ; j++)
        for (k=0; k<ninattr ; k++){
          if(m == 8) {
           m = 0;
           fprintf(fp2, "\n");
          }
          fprintf(fp2,"%f ",lin_wt[i][j][k]);
          m++;
        }
      fprintf(fp2,"\n");
    }
```

```
    for (i=0; i<noutattr; i++)
      fprintf(fp2,"%f ",sq_wt[i]);

    fprintf(fp2,"\n");

    m=0;
    for (i=0; i<nprwt ; i++){
      if(m == 2) {
          m = 0;
          fprintf(fp2, "\n");
      }
      fprintf(fp2,"%u ",pr_wt[i].dq);
      for (j=0; j<noutattr ; j++){
        fprintf(fp2,"%f ",pr_wt[i].wt[j]);
      }
      m++;
    }

    fprintf(fp2, "\n");
    rewind(fp2);
    if ((fclose(fp2)) != 0)
     printf("\nFile cannot be closed %s ", wt_file_name);
    else
     printf("\nFiles written");
}


/******* several conditions are checked
         to see whether learning should terminate ******/

int introspective (nfrom,nto)
int nfrom;
int nto;
{
    int i,flag;
                /* reached max. iteration ? */

    if (cnt>=cnt_num) return(FEXIT);

           /* error for each pattern small enough ? */

       nsnew = 0;
       flag = 1;
       for (i=nfrom; (i<nto) && (flag==1); i++) {
         if (ep[i]<=maxep) nsnew++;
         else flag = 0;
       }
       if (flag == 1) return(SEXIT);
       /* every sample error is small enough */

       /* system total error small enough ? */
```

```
        if (err_curr<=maxe) return(SEXIT);
/* some errors are large, but total error is small enough */

        return(CONTNE);
        /* some and total errors are too large */
}

/***************** threshold is treated as weight of link
     from a virtual node whose output value is unity *******/

int rumelhart(from_snum,to_snum)
int from_snum;
int to_snum;
{
     int i,j,k,m;
     float er;
     char err_file[20];

     result = CONTNE;

     strcpy(err_file,task_name);
     strcat(err_file,"e.dat");
     if (fplot10==1){
       if(!relay){
      if ((fp3=fopen(err_file,"w+")) == NULL){
          perror("Cannot open error file");
          exit(0);
      }
       }
       else{
      if ((fp3=fopen(err_file,"a+")) == NULL){
          perror("Cannot open error file");
          exit(0);
      }
       }
     }
     do {
                        err_curr = 0.0;

      for (i=from_snum; i<to_snum; i++) {
      /* for each pattern */

          forward(i);  /* bottom up calculation */
          change(i);

           /* common program error evaluation    */

          ep[i] = 0.0;
          for (m=0; m<noutattr; m++) {
           er = fabs(target[i][m] - output[i][m]);
           ep[i] += er*er/2;
          }
```

```
            ep[i] /= noutattr;
            err_curr += ep[i];
        }

             /* normalized system error */

        err_curr = err_curr/ninput;
        gotoxy(29,y);
        printf("%f",err_curr);
        if (err_curr < minerr){
           minerr = err_curr;
           min_iter=cnt;
           gotoxy(53,y);
           printf("%f",minerr);
           gotoxy(35,y+1);
           printf("%d",min_iter);
        }
/* save errors in file to draw the system error with plot10 */

        if (fplot10==1)
           fprintf(fp3,"%d  %f\n",cnt,err_curr);
        cnt++;
        gotoxy(14,y);
        printf("%d",cnt);
        gotoxy(74,y);
        tl=time()-ts+tk;
        printf("%ds",tl);

         /* check condition for terminating learning */

        result = introspective(from_snum,to_snum);
      } while (result == CONTNE);

            /* update output with changed weights */

      for (i=from_snum; i<to_snum; i++) forward (i);

      gotoxy(1,y+2);
      printf("                press any key to display results\
\");
      getch();
      printf("\n\n");
      for (i=0; i<noutattr; i++)
        printf("Theta[%d]=%8f ",i,theta[i]);
      printf("\n");
      for (i=0; i<noutattr; i++)
        printf("Bias[%d]=%8f ",i,bias[i]);

              p r i n t f ( " \ n l i n e a r \
\weight[%d][%d][%d]\n",noutattr,ninattr,ninattr);
      for (i=0; i<noutattr ; i++){
        m=0;
```

```
      printf("Output node %d \n",i);
      for (j=0; j<ninattr ; j++)
     for (k=0; k<ninattr ; k++){
       if(m == 8) {
           m = 0;
           printf("\n");
       }
       printf("%8f ",lin_wt[i][j][k]);
       m++;
     }
      printf("\n");
    }

    for (i=0; i<noutattr; i++)
      printf("square weight[%d]=%8f ",i,sq_wt[i]);

    printf("\nproduct weight: distance square & weight\n");
    m=0;
    for (i=0; i<nprwt ; i++){
      if(m == 2) {
      m = 0;
      printf("\n");
      }
      printf("    %d ",pr_wt[i].dq);
      for (j=0; j<noutattr ; j++){
     printf("%f ",pr_wt[i].wt[j]);
      }
      m++;
    }

    for (i=0; i<ninput; i++)
    for (j=0; j<noutattr; j++)
      printf("\n sample %d output %d = %f target %d = %f",
           i,j,output[i][j],j,target[i][j]);
    printf("\n\nTotal number of iteration is %d",cnt);
    printf("\nNormalized system error is %f\n",err_curr);
    if (fplot10==1) {
     rewind(fp3);
     if ((fclose(fp3)) != 0)
        printf("\nFile cannot be closed %s ", err_file);
    }
    return(result);

}


/*********** read in the input data file specified
by user during the interactive session ****************/

user_session()
{
   int i, j, k, x1, y1, c, ds, showdata;
```

```
   char fnam[20], dtype[20];
   float temp,pv;

   clrscr();
   printf("\nStart of learning session\n");

   printf("\nEnter  0  for first training, or  1  for relay\
\training : ");
   scanf("%d", &relay);
   if(relay){
     printf("\nEnter the task name (without extension) for\
\NNW structure");
     printf("\nand weight read: ");
     scanf("%s", task_name);
     dread(task_name);
     wtread(task_name);
     }/* relay */

/* for task with name task_name, input data file of the task
is automatically set to be task_name.dat by the program */
   if(!relay){
   printf("\nEnter the task name (without extension) for\
\training : ");
   scanf("%s", task_name);
   printf("No. of pts. in the edge of a scene(4 for TC)(6\
\for Target):");
   scanf("%d",&ninattr);
   /* 384/ninattr must be an integer */
/*   switch (ninattr) {
     case 4 : nprwt=9;  break;
     case 5 : nprwt=14;  break;
     case 6 : nprwt=19;  break;
     case 8 : nprwt=33;  break;
     case 10 : nprwt=50;  break;
     case 12 : nprwt=70;  break;
     case 16 : nprwt=119;  break;
     default : nprwt=NMXPR_WT;
   }                                    */

   nprwt=1;
   pr_wt[0].dq=1;
   x1=0;
   for(j=0;j<ninattr;j++)
     for(k=0;k<ninattr;k++){
       if(x1<(ninattr-1)){
      for(y1=j,x1=k+1;x1<ninattr;x1++){
        ds=(x1-k)*(x1-k);
        c=0;
        do{ if(ds==pr_wt[c].dq){
            c=-1;
            }
            c++;
```

```
        }while(1<=c && c<nprwt);
        if(c>0){ /* c==nprwt; */
        pr_wt[c].dq=ds;
        nprwt++;
        }
      }
       }
       if(y1<(ninattr-1)){
      for(y1=j+1;y1<ninattr;y1++)
        for(x1=0;x1<ninattr;x1++){
          ds=(x1-k)*(x1-k)+(y1-j)*(y1-j);
          c=0;
          do{ if(ds==pr_wt[c].dq){
           c=-1;
           }
           c++;
          }while(1<=c && c<nprwt);
          if(c>0){ /* c==nprwt */
          pr_wt[c].dq=ds;
          nprwt++;
          }
        }
       }
     }

   printf("No. of product weight is %d\n", nprwt);

   printf("No. of output units (max. %d, 2 for TC, 3 for\
\Target): ",NMXOATTR);
   scanf("%d", &noutattr);
   printf("Total number of input samples (max. %d) :\
\",NMXINP);
   scanf("%d",&ninput);
   }/* !relay */
   strcpy(fnam, task_name);
   strcat(fnam,".dat");

   if (( fp1 = fopen(fnam,"r"))==NULL){
       printf("\nFile %s does not exist",fnam);
      }
      printf("\nDo you want to look at data just read?");
   printf("    Answer yes or no : ");
   scanf("%s",dtype);
   showdata = ((dtype[0] == 'y') || (dtype[0] == 'Y'));
   for (i=0; i<ninput; i++) {
       for (j=0; j<ninattr; j++)
        for (k=0; k<ninattr; k++) {
          fscanf(fp1,"%f",&temp);   input[i][j][k]=temp;
          if (showdata) printf("%f  ",input[i][j][k]);
        }
       if (showdata) printf("\n");
       for (j=0; j<noutattr; j++) {
```

```
      fscanf(fp1,"%f",&target[i][j]);
      if (showdata) printf("%f  ",target[i][j]);
     }
     if (showdata) printf("\n");
/* input level output calculation */
  sq[i]=0;
  for (j=0; j<ninattr; j++)
    for (k=0; k<ninattr; k++)
   sq[i] += input[i][j][k]*input[i][j][k];

  for(c=0; c<nprwt; c++)
    pr_val[i][c]=0;
  x1=0;
  for(j=0;j<ninattr;j++)
    for(k=0;k<ninattr;k++){
   if(x1<(ninattr-1)){
     for(y1=j,x1=k+1;x1<ninattr;x1++){
       ds=(x1-k)*(x1-k);
       pv=input[i][j][k]*input[i][y1][x1];
       c=0;
       do{ if(ds==pr_wt[c].dq){
        pr_val[i][c] += pv;   c=-1;
        }
        c++;
       }while(1<=c && c<nprwt);
      if(c != 0) printf("product weight wrong %d ",c);
    /*     if(c>0){ i.e. c==nprwt
       pr_wt[c].dq=ds;
       pr_val[i][c]=pv;
       nprwt++;
       }                                              */
     }
   }
   if(y1<(ninattr-1)){
     for(y1=j+1;y1<ninattr;y1++)
       for(x1=0;x1<ninattr;x1++){
         ds=(x1-k)*(x1-k)+(y1-j)*(y1-j);
         pv=input[i][j][k]*input[i][y1][x1];
         c=0;
         do{ if(ds==pr_wt[c].dq){
          pr_val[i][c] += pv;   c=-1;
          }
          c++;
         }while(1<=c && c<nprwt);
         if(c != 0) printf("product weight wrong %d ",c);
    /*     if(c>0){  i.e. c==nprwt
         pr_wt[c].dq=ds;
         pr_val[i][c]=pv;
         nprwt++;
         }                                             */
       }
   }
```

```
        }
    }
    rewind(fp1);
    if ((c=fclose(fp1)) != 0){
         printf("\nFile  cannot  be  closed  in  user_session\
\%d",c);
      exit(0);
    }
}


/************** main body of output generation ************/

output_generation()
{
  char ans[10];
  char dfile[20];
  int i, j, k,c,x1,y1,ds,ta,ex[NMXINP];
  char out_file_name[20];
  float temp,pv,th[NMXOATTR];

/* If task is already in the memory, data files for task do
not need to be read in. But, if ti is a new task, data files
should be read in to reconstruct the net */

  printf("\nGeneration of outputs for a new pattern");
  printf("\n\t Present task name is %s", task_name);
  printf("\n\t Work on a \" different \" task?   ");
  printf("\n\t   Answer yes or no : ");
  scanf("%s", ans);
  if ((ans[0]=='y') || (ans[0]=='Y')){
     printf("\n\t Type the task name (without extension) :\
\");
     scanf("%s", task_name);
     dread(task_name);
     wtread(task_name);
  }

     /* input data for output generation are created */

  printf("\nFile name for patterns to be processed (without\
\extension) : ");
  scanf("%s",dfile);

  strcpy(out_file_name,dfile);
  strcat(out_file_name, "o.dat");
  strcat(dfile, ".dat");

  if ((fp1=fopen(dfile,"r"))==NULL){
      perror("Cannot open dfile");
      exit(0);
  }
```

```
  printf("No. of pts. in the edge of a scene(4 only TC)( 6\
\8 12 16 ):");
  scanf("%d",&nscene);
  printf("\No. of patterns for processing (max. %d) :\
\",NMXINP);
  scanf("%d", &nsample);
  printf("For noise level 0, recomand output threshold :\ \0.8
0.8  0.8\n");
  printf("For noise level above 0.6, recomand output\
\threshold : ");
  printf("0.995  0.8  0.8\n");
  for(i=0; i<noutattr; i++){
  printf("Enter threshold for output[%d] : ",i);
  scanf("%f",&th[i]);
  }
  printf("Enter  1  for taging non-taget , or  0  for\
\taging target : ");
  scanf("%d",&ta);
  printf("\nProcessing, please wait !");

  if ((fp3 = fopen(out_file_name,"w+")) == NULL){
      perror("Cannot open data file ");
      exit(0);
  }
  for (j=0; j<noutattr; j++){
    fprintf(fp3,"Threshold[%d]=%f ",j,th[j]);
  }
  fprintf(fp3,"\n");

  for (i=0; i<nsample; i++) {
    for (j=0; j<nscene; j++)
      for (k=0; k<nscene; k++){
     fscanf(fp1,"%f",&temp);
     input[i][j][k]=temp;
      }
    for (j=0; j<noutattr; j++)
      fscanf(fp1,"%f",&target[i][j]);
  }
  if ((c=fclose(fp1)) != 0)
      printf("\nFile cannot be closed in out_generation\
\%d",c);

  for (i=0; i<nsample; i++) {
    printf("\nProcess sample %d : ",i);
    exist=0;
    for (j=0; j<noutattr; j++){
      output[i][j]=0.0;
    }
   /* separate the input scene into subscenes */
   for( ystart=0; ystart<=(nscene-ninattr); ystart+=2)
     for( xstart=0; xstart<=(nscene-ninattr); xstart++){
   /* input level output calculation */
```

```
sq[i]=0;
for (j=0; j<ninattr; j++)
  for (k=0; k<ninattr; k++)
 sq[i] += input[i][j+ystart][k+xstart]*
          input[i][j+ystart][k+xstart];

for(c=0; c<nprwt; c++)
  pr_val[i][c]=0;
x1=0;
for (j=0; j<ninattr; j++)
  for (k=0; k<ninattr; k++){
 if(x1<(ninattr-1)){
   for(y1=j,x1=k+1;x1<ninattr;x1++){
     ds=(x1-k)*(x1-k);
     pv=input[i][j+ystart][k+xstart]*
        input[i][y1+ystart][x1+xstart];
     c=0;
     do{ if(ds==pr_wt[c].dq){
      pr_val[i][c] += pv;   c=-1;
      }
      c++;
     }while(1<=c && c<nprwt);
    if(c != 0) printf("product weight wrong %d ",c);
  /*     if(c>0){ i.e. c==nprwt
     pr_wt[c].dq=ds;
     pr_val[i][c]=pv;
     nprwt++;
     }                                              */
   }
 }
 if(y1<(ninattr-1)){
   for(y1=j+1;y1<ninattr;y1++)
     for(x1=0;x1<ninattr;x1++){
       ds=(x1-k)*(x1-k)+(y1-j)*(y1-j);
       pv=input[i][j+ystart][k+xstart]*
          input[i][y1+ystart][x1+xstart];
       c=0;
       do{ if(ds==pr_wt[c].dq){
        pr_val[i][c] += pv;   c=-1;
        }
        c++;
       }while(1<=c && c<nprwt);
       if(c != 0) printf("product weight wrong %d ",c);
  /*     if(c>0){  i.e. c==nprwt
       pr_wt[c].dq=ds;
       pr_val[i][c]=pv;
       nprwt++;
       }                                            */
     }
 }
  }
/* output generation calculation starts */
```

```
      forward(i);
   /*      fprintf(fp3,"Sample   %d   ,block   (%d,%d)   :   "
,i,xstart,ystart); */
      for (c=0; c<noutattr; c++){
   /*          fprintf(fp3,"out[%d]=%f ",c,out[c]);  */
       if (out[c] > output[i][c])  output[i][c] = out[c];
      }
   /*   fprintf(fp3,"\n");  */
     }/* end of subscene */
     fprintf(fp3,"Sample %d : ",i);
     for (j=0; j<noutattr; j++){
       fprintf(fp3,"output[%d]=%f ",j,output[i][j]);
       printf("output[%d]=%f ",j,output[i][j]);
       if(output[i][j]>th[j])  exist++;
     }
     fprintf(fp3,"\nSample %d : ",i);
     for (j=0; j<noutattr; j++){
       fprintf(fp3,"target[%d]=%f ",j,target[i][j]);
     }
     ex[i]=0;
     if(exist>=1
     ){
        ex[i]=1;
        printf("\nTarget exists !");
        fprintf(fp3,"\nTarget exists !\n");
     }else{  printf("\nTarget does not exist !");
          fprintf(fp3,"\nTarget does not exist !\n");
     }
  }/* end of i */
  if ((fclose(fp3)) != 0)
      printf("\nFile cannot be closed %s ", out_file_name);
  if(ta==1) printf("\nWithout Target, samples : ");
  else      printf("\nWith    Target, samples : ");
  for(i=0; i<nsample; i++){
    if ((ta==1) && (ex[i]==0))  printf("%2d ",i);
    if ((ta==0) && (ex[i]==1))  printf("%2d ",i);
  }
}


graph()
{

/* This program is used to show the graph of the samples
   of training set or test set
   The graph is scaled automatically
   Only EGA and VGA mode can be used


*/
   float temp;
   int driver,mode,sno,scan;
```

```
   int j,k,num,xm,ym;
   char exit[2];
   static char up[17][8]={
   {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00}, /* 0 */
   {0x00, 0x22, 0x00, 0x00, 0x00, 0x22, 0x00, 0x00}, /* 1 */
   {0x00, 0x22, 0x00, 0x44, 0x00, 0x22, 0x00, 0x44}, /* 2 */
   {0x88, 0x22, 0x00, 0x44, 0x88, 0x22, 0x00, 0x44}, /* 3 */
   {0x88, 0x22, 0x11, 0x44, 0x88, 0x22, 0x11, 0x44}, /* 4 */
   {0x88, 0x22, 0x99, 0x44, 0x88, 0x22, 0x99, 0x44}, /* 5 */
   {0x99, 0x22, 0x99, 0x44, 0x99, 0x22, 0x99, 0x44}, /* 6 */
   {0x99, 0x66, 0x99, 0x44, 0x99, 0x66, 0x99, 0x44}, /* 7 */
   {0x99, 0x66, 0x99, 0x55, 0x99, 0x66, 0x99, 0x55}, /* 8 */
   {0xdd, 0x66, 0x99, 0x55, 0xdd, 0x66, 0x99, 0x55}, /* 9 */
   {0xdd, 0x66, 0xbb, 0x55, 0xdd, 0x66, 0xbb, 0x55},/* 10 */
   {0xdd, 0x66, 0xbb, 0xdd, 0xdd, 0x66, 0xbb, 0xdd},/* 11 */
   {0xdd, 0x77, 0xbb, 0xdd, 0xdd, 0x77, 0xbb, 0xdd},/* 12 */
   {0xdd, 0x77, 0xbb, 0xff, 0xdd, 0x77, 0xbb, 0xff},/* 13 */
   {0xdd, 0xff, 0xbb, 0xff, 0xdd, 0xff, 0xbb, 0xff},/* 14 */
   {0xff, 0xff, 0xbb, 0xff, 0xff, 0xff, 0xbb, 0xff},/* 15 */
   {0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff} /* 16 */
   };

     /* nsample - no. of samples in              */
     /* the sample file.                         */
     /* ninattr - no. of points in each sample   */
     /* sno - sample no.                         */

   /* scan  - hold scanf() condition, 0 for read int fail
      exit  - hold exit condition
   */

/*   driver = DETECT;
   initgraph(&driver,&mode,"");
   mode = getgraphmode();

     mode = EGA;                  can not use here
     vpix = 349;                  no. of pixels vertically
     hpix = 639;                  no. of pixels horizontally
     orgx = 20;                   x-coordinate of origin
     mb   = 35;                  margin bottom              */

/*     VGA mode      */
     clrscr();
     textmode(19);
     driver = VGA;
     mode = VGAHI;
     initgraph(&driver,&mode,"");
/*     vpix = 479;
     hpix = 639;
     orgx = 20;
     mb   = 50;                    */
```

```
/*        setrgbpalette(63,31,31,31);
        setbkcolor(0);
        setcolor(63);*/
        setlinestyle(0,0,1); /* 3 thick line, 1 thin line */
        for(j=56; j<441; j=j+384/nscene)
      {for(k=123; k<508; k=k+384/nscene)
       {
        line(123,j,507,j);
        line(k,56,k,440);
       }
      }

  do{
   gotoxy(15,3);
   printf("Display graph of sample no. (0-%d) , E to exit :\
\",nsample-1);
   scan = scanf ("%d",&sno);
      if (scan!=1) scanf("%s",exit);  /* scan==0 read int fail
*/
   cleardevice();
       for(j=56; j<441; j=j+384/nscene)
     {for(k=123; k<508; k=k+384/nscene)
      {
       line(123,j,507,j);
       line(k,56,k,440);
      }
     }
   for(j=0; j<nscene; j++)
    for(k=0; k<nscene; k++)
      {temp = input[sno][j][k];
       if(temp<.06667) num = 1;
       else if(temp<.13334) num = 2;
       else if(temp<.20001) num = 3;
       else if(temp<.26667) num = 4;
       else if(temp<.33335) num = 5;
       else if(temp<.40001) num = 6;
       else if(temp<.46668) num = 7;
       else if(temp<.53334) num = 8;
       else if(temp<.60001) num = 9;
       else if(temp<.66667) num = 10;
       else if(temp<.73334) num = 11;
       else if(temp<.80001) num = 12;
       else if(temp<.86667) num = 13;
       else if(temp<.93334) num = 14;
  /*     else if(temp<1.) num = 14;
       else if(temp<1.25) num = 15;  */
       else num = 15;
 /*      setfillstyle(12,63);  */
       setfillpattern(up[num],15);
       xm=123+192/nscene+(k*384/nscene);
       ym=56+192/nscene+(j*384/nscene);
       floodfill(xm,ym,15);
```

```
        }
 /*       settextstyle(TRIPLEX_FONT,0,5);
        outtextxy(45,456," PRESS 'e' or 'E' TO LEAVE ");  */

    gotoxy(10,1);
    if (result==1){
      printf("Graph of sample %d network output =%f %f %f",
sno, output[sno][0], output[sno][1], output[sno][2]);
      gotoxy(10,2);
      printf("Target of sample %d                =%f %f %f",
sno, target[sno][0], target[sno][1], target[sno][2]);
    }
    else {
      printf("Graph of sample %d Target          = %f %f %f",
sno, target[sno][0], target[sno][1], target[sno][2]);
    }
   }while(toupper(exit[0]) != 'E');
   closegraph();

 }

grddata()        /* read data for graph display */
 {

   char filename[15];
   int i,j,k;
   float temp;

   clrscr();
   printf("\n                        Graphical Presentation of\
\Data\n\n");
   printf("\nInput  file  name  for  data  graphing  (without\
\extension) : ");
   scanf ("%s",&filename);
      strcat(filename,".dat");

   printf("\n\nNo. of samples max. %d : ",NMXINP);
   scanf ("%d",&nsample);
   printf("\nNo. of points in the edge of a scene (4 only\
\TC)( 6 8 12 16 ): ");
   scanf ("%d",&nscene);
   printf("\nNo. of output units ( 2 for TC, 3 for Target) :\
\");
   scanf ("%d",&noutattr);

   if(( fp1 = fopen(filename,"r"))==NULL){
       perror("\n Cannot open data file ");
       exit(0);
   }

         /* read data into the array */
    for (i=0; i<nsample; i++){
```

```
      for (j=0; j<nscene; j++)
      for (k=0; k<nscene; k++)
       { fscanf(fp1,"%f",&temp);
         input[i][j][k]=temp;
       }
      for (j=0; j<noutattr; j++)
        fscanf(fp1,"%f",&target[i][j]);
    }
    rewind(fp1);
  }

/******************** bottom up calculation
              of net for input pattern i  ******************/

forward(int i)
{
    int j, k, c, x1, y1, ds;
    float pv,net[NMXOATTR];
    /* thetaj is for sigmod output function and is built in
       bias[NMXOATTR]*/
    /* thetaj=0.5;  */

    /* output layer output calculation */

    for (c=0; c<noutattr; c++) {
      if (invtype == 0){      /* product terms only */
      /* threshold */
      net[c] = bias[c];
      /* product terms */
      for (j=0; j<nprwt ; j++)
        net[c] += pr_val[i][j]*pr_wt[j].wt[c];
      }

      if (invtype == 1){      /* linear & product terms */
      /* linear terms & threshold */
      net[c] = bias[c];
      for (j=0; j<ninattr; j++)
        for (k=0; k<ninattr; k++)
          net[c] +=
          input[i][j+ystart][k+xstart]*lin_wt[c][j][k];
      /* product terms */
      for (j=0; j<nprwt ; j++)
        net[c] += pr_val[i][j]*pr_wt[j].wt[c];
      }

      if (invtype == 2){      /* square and product terms */
      /* threshold and square terms */
      net[c] = bias[c]+sq[i]*sq_wt[c];
      /* product terms  */
      for (j=0; j<nprwt ; j++)
        net[c] += pr_val[i][j]*pr_wt[j].wt[c];
      }
```

```
      if (invtype == 3){
      /* threshold and square terms */
      net[c] = bias[c]+sq[i]*sq_wt[c];
      /* linear terms */
      for (j=0; j<ninattr; j++)
        for (k=0; k<ninattr; k++)
          net[c] +=
          input[i][j+ystart][k+xstart]*lin_wt[c][j][k];
      /* product terms  */
      for (j=0; j<nprwt ; j++)
        net[c] += pr_val[i][j]*pr_wt[j].wt[c];
      }
      if (yfunc==0) {
       if(nscene>ninattr){
           if (net[c]>=0.0)
            out[c] = 1.0;
           else
            out[c] = -1.0;
       }else{     if (net[c]>=0.0)
                  output[i][c] = 1.0;
             else
                  output[i][c] = -1.0;
       }
      }
      if (yfunc==1)
       if(nscene>ninattr){
           out[c] = tanh(net[c]);
       }else{     output[i][c] = tanh(net[c]);
       }
      if (yfunc==2)
       if(nscene>ninattr){
           out[c] = 1.0/(1.0+exp(-net[c]+theta[c]));
       }else{     output[i][c] =
                  1.0/(1.0+exp(-net[c]+theta[c]));
       }
    }
}

/***************** error, delta weights and new weights
                   for higher order net **********************/

change(int i)
{
    int j,k,c;

   /** 90 degrees rotation & translation invariant net  **/

   for (c=0; c<noutattr; c++) {
       if (yfunc==0)
       err[c] = target[i][c]-output[i][c];
       if (yfunc==1)
                  e    r    r    [    c    ]              =
```

```
(target[i][c]-output[i][c])*1/(cosh(output[i][c])
              *cosh(output[i][c]));
      if (yfunc==2)
                e    r    r    [    c    ]                =
(target[i][c]-output[i][c])*(1-output[i][c])*output[i][c];
      if (invtype == 0){
      if(yfunc==2){ /* theta */
        deltheta[c] = -1*eta*err[c]-deltheta[c]*alpha;
        theta[c] += deltheta[c];
      }
    /* bias */
      del_bias[c] = eta*err[c]+del_bias[c]*alpha;
      bias[c] += del_bias[c];
      /* pr_wt */
      for (j=1; j<nprwt; j++) {
         delpr_wt[c][j] = eta*err[c]*pr_val[i][j]
                    +delpr_wt[c][j]*alpha;
           pr_wt[j].wt[c] += delpr_wt[c][j];
      }
      }

      if (invtype == 1){
      if(yfunc==2){ /* theta */
        deltheta[c] = -1*eta*err[c]-deltheta[c]*alpha;
        theta[c] += deltheta[c];
      }
    /* bias */
      del_bias[c] = eta*err[c]+del_bias[c]*alpha;
      bias[c] += del_bias[c];
      /* linear */
      for (j=0; j<ninattr; j++)
        for (k=0; k<ninattr; k++){
          delin_wt[c][j][k] = eta*err[c]
          *input[i][j][k]+delin_wt[c][j][k]*alpha;
          lin_wt[c][j][k] += delin_wt[c][j][k];
        }
      /* pr_wt */
      for (j=1; j<nprwt; j++) {
         delpr_wt[c][j] = eta*err[c]*pr_val[i][j]
                    +delpr_wt[c][j]*alpha;
           pr_wt[j].wt[c] += delpr_wt[c][j];
      }
      }

      if (invtype ==2){
      if(yfunc==2){ /* theta */
        deltheta[c] = -1*eta*err[c]-deltheta[c]*alpha;
        theta[c] += deltheta[c];
      }
    /* bias */
      del_bias[c] = eta*err[c]+del_bias[c]*alpha;
      bias[c] += del_bias[c];
```

```
        /* square */
        delsq_wt[c] = eta*err[c]*sq[i]+delsq_wt[c]*alpha;
        sq_wt[c] += delsq_wt[c];
        /* pr_wt */
        for (j=1; j<nprwt; j++) {
           delpr_wt[c][j] = eta*err[c]*pr_val[i][j]
                      +delpr_wt[c][j]*alpha;
             pr_wt[j].wt[c] += delpr_wt[c][j];
        }
        }

        if (invtype == 3){
        if(yfunc==2){ /* theta */
          deltheta[c] = -1*eta*err[c]-deltheta[c]*alpha;
          theta[c] += deltheta[c];
        }
      /* bias */
        del_bias[c] = eta*err[c]+del_bias[c]*alpha;
        bias[c] += del_bias[c];
        /* linear */
        for (j=0; j<ninattr; j++)
          for (k=0; k<ninattr; k++){
            delin_wt[c][j][k] = eta*err[c]
            *input[i][j][k]+delin_wt[c][j][k]*alpha;
            lin_wt[c][j][k] += delin_wt[c][j][k];
          }
        /* square */
        delsq_wt[c] = eta*err[c]*sq[i]+delsq_wt[c]*alpha;
        sq_wt[c] += delsq_wt[c];
        /* pr_wt */
        for (j=1; j<nprwt; j++) {
           delpr_wt[c][j] = eta*err[c]*pr_val[i][j]
                      +delpr_wt[c][j]*alpha;
             pr_wt[j].wt[c] += delpr_wt[c][j];
        }
        }
    }
}
```

APPENDIX C: PWN.C Simulation Program

```
/* PWN.C */
/* This program generates the number of product weight */
#include"stdio.h"
main()
{
 int c,x,y,x1,y1,xs,ys,h,ds,*wp,a[2000];
 wp=a;
 do{
   printf("Input the edge size for 2D estimate of the\ \number
of product weight.\n");
   printf("xsize ysize : ");
   scanf("%d %d",&xs,&ys);
   h=1;
   *(wp)=1; /* No. of product weight */
   *(wp+1)=1;  /* first distance square 1 */
   x1=0;
   for(y=0;y<ys;y++){
     for(x=0;x<xs;x++){
       if(x1<(xs-1)){
      for(y1=y,x1=x+1;x1<xs;x1++){
        ds=(x1-x)*(x1-x);
        h=1;
        do{ if(ds==*(wp+h)){
            h=-1;
            }
            h++;
        }while(1<h && h<=*wp);
        if(h>0){ /* h=*(wp)+1; */
        *(wp)+=1;
        *(wp+h)=ds;
        }
      }
       }
       if(y1<(ys-1)){
      for(y1=y+1;y1<ys;y1++){
        for(x1=0;x1<xs;x1++){
          ds=(x1-x)*(x1-x)+(y1-y)*(y1-y);
          h=1;
          do{ if(ds==*(wp+h)){
           h=-1;
           }
           h++;
          }while(1<h && h<=*wp);
          if(h>0){ /* h=*(wp)+1; */
          *(wp)+=1;
          *(wp+h)=ds;
          }
       }
```

```
      }
       }
     }
   }
   c=0;
   for(h=1; h<=*wp; h++){
     if(c==16){
       c=0;  printf("\n");
     }
     printf("%4d ",*(wp+h));  c++;
   }
   printf("\nThe number of product weight is %d .\n",*wp);
   printf("Do you want to see another one ? key in 1 (yes)\
\,0 (no)\n");
   scanf("%d",&c);
 }while(c==1);
}
```