NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

Tactile Pattern Recognition Using Neural Networks

by

D. Michael Colven, B.A.Sc

A thesis submitted to the
Faculty of Graduate Studies and Research
in partial fulfillment of the requirements
for the degree
Master of Applied Sciences in Electrical Engineering

Ottawa Carleton Institute for Electrical Engineering
Department of Electrical Engineering
Faculty of Engineering
University of Ottawa
Ottawa, Ontario, Canada

May 5, 1993

ISBN 0-315-89608-6

Canada

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

Abstract

This thesis presents a system for the capture and recognition of tactile images using neural networks. Neural Networks utilizing the backpropagation technique are used to provide a general purpose recognition engine for several classes of pattern recognition problems. Examples of successful networks are presented with discussion of the results and methodology for development of each. The system consists of the following: Image capture, training of network using an iterative approach and testing of the network against independent images not present during training.

Pattern capture is performed by scanning a force sensitive tactile sensor that interfaces to a general purpose computer. Following capture, examples of tactile patterns of desired types are stored in a training file and the training goal of the network set. The goal is determined by the "trainer" who when the patterns are captured indicates the pattern type. Related patterns are given the same class name. The Network is required to consist of as many output neurons as classification types. The goal is that an output neuron becomes "activated" when its pattern types are present and "de-activated" when another type is present.

Patterns in the training file are then recursively applied to the inputs of the Neural Network. Once the Network converges to the desired goal it is tested against a new set of patterns to determine if the network has learned to apply generalization in its recognition of the patterns. The training set and network topology may be modified in heuristic fashion until satisfactory results are achieved.

Acknowledgment

My first thanks have to go to my wife, Katherine for persevering with all those late nights and wondering "why you love that computer more than me," as well as support through all the years of this "part time" endeavor. My second thanks go to Emil (Petriu) for his guidance and thoughts on the future of humanity and civilization. Also some practical and timely advise on matters related to this thesis. In addition, and not to be down rated, are the contributions of my mother and grandmother for asking why I had not already received a Ph.D. whenever I saw either of them and father for always being there. I would also like to thank all the professors at Ottawa and Carleton who helped along the long and winding road.

Table of Contents

Chapter 1
Introduction

There are two primary goals to robotics research: the adaptation of machines to performance of tasks previously undertaken by human beings and the autonomous action of machines to act in environments too harsh for human action. The first is undertaken to relieve humans of mundane tasks or tasks that can be more optimally performed by automata. The second requires greater independence of action from the automata and therefore, a higher level of "intelligent" behavior. It is primarily as a step, towards achieving this second goal that the experimental system presented by this thesis was developed.

Examples of the type of conditions where machines must act in unstructured settings are: space, deep sea operations, military and other harsh environments. These settings have one thing in common, difficulty in controlling and guiding the actions of the automata since, the basic environment is hostile to human operators and therefore, operations must be remote. Due to the constraints of these example environments, it is desirable to develop automata that can operate partially or entirely autonomously within loosely defined problem domains.

Abidi and Gonzalez [Abid90] discuss three reasons for the development of autonomous systems: *Reliability* "When a definite line of dichotomy can be established between 'normal' and 'abnormal' situations, machine-driven decisions can be more reliable than human decisions. The latter can be affected negatively by omissions and illusions that are often induced by stringent conditions...," *Adaptability* "Autonomous systems can and should be designed in a modular fashion to accommodate future growth and to allow subsystems to monitor their own operation as well as the operation of other subsystems. This leads to safer and more reliable overall systems requiring less reliance on human and ground support", *Cost* "Improvements in safety, reliability and adaptability result in higher efficiency, hence a reduction in the overall cost . . . "The method of achieving these goals can be learned from the design of the system whose behavior is to be emulated, the human being. Humans are designed to interact with their environment by integrating the input from many types of sensors: audio, visual, olfactory, taste and tactile. Each of these interfaces uses information available and pertinent to the natural environment of the human. It is the diversity and richness of these senses that aids humans in comprehending and interacting with their environment.

A second, equally important factor in the success of human beings as information processors is the concept of abstraction. Not all information is passed directly to the cognitive part of the brain. As discussed by A. Treisman in the article "Features and Objects in Visual Processing" [Trei86], lower levels of sensing and pattern recognition are

performed by the brain acting as an optical pre-process. The quick recognition of physical motion, the reaction to some familiar shapes such as circles and differentiation based on colours are manifestations of these processes. ... similar types of processing occur in the other sensory systems." The almost instant ability of the brain to recognize and discriminate odors, tastes, visual objects and textures points to the presence of a pattern pre-processing capability.

The third important quality in adaptability of systems is re-use of general purpose resources for solving several classes of problem.

Thus, the three properties of integration of many sensors, abstraction of sensory input and re-usability of resources, are primary factors in the ability of humans to function in varying environments. Therefore, machines that successfully interact with their environment should incorporate these features into their design.

The successful use of image processing and recognition techniques has advanced the technology of robotics in the direction of machine vision. However, as shown in the human analogy, many types of sensors are required to produce a truly autonomous and adaptable system. As discussed by Abidi and Gonzalez [Abid90] the development of truly adaptable robot systems requires the use of multi-sensor data fusion. Therefore, dependent on the environment, audio, electromagnetic, tactile etc. sensors are also required. It is the integration of information from these varying sensors that aids the automation in interacting with its surroundings. One step in the process of achieving this end is in the development of tactile sensing technology. When used in conjunction with the primary sensors tactile sensors can perform an important role in machine perception.

The incorporation of a tactile sensor into the system involves the abstraction of the "raw" cutaneous information into meaningful and generalized concepts that provide the desired results. Thus, a machine confronted with an unknown input from a tactile sensor should recognize familiar abstractions. These could include the number of pressure points on a sensor, a soft or hard surface, the amount of texture in a surface, the presence of straight or curved lines, recognition of objects, directions of edges, etc. The presence of these features can be passed to higher levels of sensory integration and intelligence to provide the required sensory union. In turn actions appropriate to the current machine state and environmental conditions, are produced by the automata.

In order to provide a classification and recognition engine for the tactile pre-sensing problem this thesis explores the use of artificial neural systems. However, many techniques exist for pattern recognition and classification therefore, what basis exists for choosing the neural network paradigm? To understand why it is believed that neural networks provide a good mechanism for this type of processing is addressed by the need for adaptability of methods and resources. One aspect of adaptability of automata is the ability to learn. Artificial neural systems also have this attribute since, the network learns by example. Both positive and contrary examples of the patterns are presented and the network generalizes the information in the connections between the neurons. Therefore, the ability to learn provides a general purpose machine with the ability to be re-

programmed to suit a new environment. Traditional techniques usually employ a fixed algorithm and/or knowledge. Therefore, there is no inherent ability to learn from experience and adaptability is compromised.

The second level of adaptability is the re-use of resources for multiple procedures within the current problem domain. This type of flexibility also exists in the artificial neural system. The algorithms and structure for the various networks are identical, the difference lying in the specific network topology values of the interconnections within a network. These attributes may be stored as parameters describing a particular application. However, there is no essential difference in procedure between the various applications and therefore, the same program may be used for fundamentally different pattern recognition and classification problems. In the case of traditional pattern recognition techniques the algorithm is problem domain specific and therefore, a general non-optimized processing resource must be used if a machine is to be adaptable to many environments. Therefore, little optimization of processing hardware is possible. In addition, since the algorithm is specific to a given problem the developer must have extensive a priori knowledge in order to optimize the recognition technique. In the case of neural networks, the developer need not have this extensive knowledge of the specific classification problem to be solved but can rely on the network to learn from examples of the correctly classified patterns.

This thesis deals with the development of a system for the recognition and categorizations of tactile patterns using a Backpropagation neural network. For reasons previously addressed, it is believed that this type of system represents a flexible approach to the application of a pre-processing function for "intelligent" and adaptive automata. The system consists of the following elements:

i. Capture of 16 by 16 resolution tactile patterns (digitized to 256 levels per sample),

ii. Characterization of captured patterns by type,

iii. Storage of a representative training sets of images,

iv. Training of different configurations of neural networks (changing number of layers, neurons per layer etc.) to find minimum successful topology,

v. testing of networks abilities to correctly classify a new set of patterns (i.e. different from the original training set),

vi. Adding patterns not correctly identified into the original training set and re-training the network with this augmented set,

vii. Repeating steps v and vi until the network displays satisfactory classification capabilities.

The back propagation learning paradigm fits well within this type of training regime. As previously noted, there is no requirement for on-line learning of low level images for the type of recognition system where the network simply acts a pre-cognitive process. In addition, it is advantageous that the networks are of the "supervised" type, so that learning may be directed in a desired direction. The Backpropagation algorithm has these properties, also the trait of requiring no *a priori* knowledge of the problem solution

domain as required by other neural network and pattern recognition paradigm. Therefore, it was felt that this type of neural network would provide a suitable methodology for the tactile pattern classification system. The results presented from networks developed using this paradigm is presented in chapter 4.

Chapter 2 presents information on the Force Sensitive Resistor type of tactile sensor: constraints, applicability and limitations as well as interface concepts to this device.

Chapter 3 presents a discussion of artificial neural systems with emphasis on the Backpropagation algorithm and suitable training methods. The mathematical basis for the application of a Backpropagation neural network to tactile image processing is also discussed.

Chapter 4 presents the results of four experiments of applications of the Backpropagation network to tactile sensing problems: the "One-Two" net recognizes one or two finger pressure points on the sensor surface, the "Big-Block" net classifies embossed letters in several orientations, the "Small-Block" net is similar to "Big-Block" except that smaller letters in single orientations are detected and the "Lines" net detects the orientations of straight lines relative to the sensor surface. The changes of results with each phase of network development are presented along with analysis of the final network are presented for each example.

Chapter 5 provides general conclusions for the experimental results along with recommendations for further work in the field.

Chapter 2
A Discussion of Tactile Sensors and Tactile Sensing

2.1 Introduction

The role of the tactile sensor as an element in a multi sensor fusion system as it relates to machine perception has been expounded upon in chapter 1. Unfortunately, the technology of tactile sensing has not kept pace with those of vision and computational processing for several reasons: the development of inexpensive image processing and sensing equipment has been aided by other markets for such products and the growth of computational power has also been aided by similar growth in personal and main frame systems. However, the development of a high resolution tactile-sensor has lead to the possibility of applying feature determination and other low level pattern classification paradigms to these sensor types. The tactile sensing elements differ from the older force torque sensors in their intended purpose. Primarily, the goal of the force torque type of sensors is in "gross" mechanical feedback to the automata. This type of feedback could come into play in such a system as a machine designed to grasp delicate objects or to avoid forcing mechanical objects beyond their design limits. The tactile-sensor on the other hand, is primarily designed to supply cutaneous information. This information consists of such surface features, texture etc. that are local to the sensor surface. A suitable analogy is that of the finger tips of the human.

A variety of tactile sensor technologies exist, the most common of which are: conductive-elastomer, strain gauge, piezoelectric, capacitive and opto-electronic. These types can be abstracted into two categories based on their operating principals: force-sensitive and displacement-sensitive. Desirable properties of the "ideal" tactile-sensor are: high resolution, compliance, simple interfacing, ruggedness and low cost.

The need for high resolution is apparent from the nature of the problem, that of discrimination between features with a high degree of certainty.

The ability of the sensor to be compliant in this sense is that its structure can conform to the desired shape of the sensing surface. This property is desirable in that rigid sensors cannot conform to the shape of a robotic finger etc. and therefore, they cannot be adapted to the shape of a desired manipulator.

The ability to detect tactile information with a simple interface is necessary to keep the overhead in electronic components low, for a large complex system this can be a driving factor in design.

In addition, a rugged sensor is required for the type of environments where automata are useful. Although delicate sensors may perform well in a laboratory environment their usefulness in practical unstructured environments is limited.

Finally an easily produced inexpensive sensor provides both a research path insensitive and a better cost performance tradeoff to manufacturers of robotic systems. This property aids in the acceptance and increases the use of such sensors further prompting developments in the technology.

A new type of sensor, the Force Sensing Resistor (FSR) shows promise in all these areas. The FSR consists of a polymer sheet with a layer of flexible, tough, low cost sensing film applied to its surface by screen printing. The lithographic nature of the process implies that high resolution, high reproducibility and low manufacturing costs are achievable for this type of product. The polymer nature of the FSR means that it is itself compliant and rugged. In addition the sensor itself acts as a matrix of resistance values (see Figure 2.01). The values of these resistors are inversely proportional to a force applied normal to the surface. Various scanning techniques can be used to "decouple" the resistance values and determine the individual resistance's [Meye91].

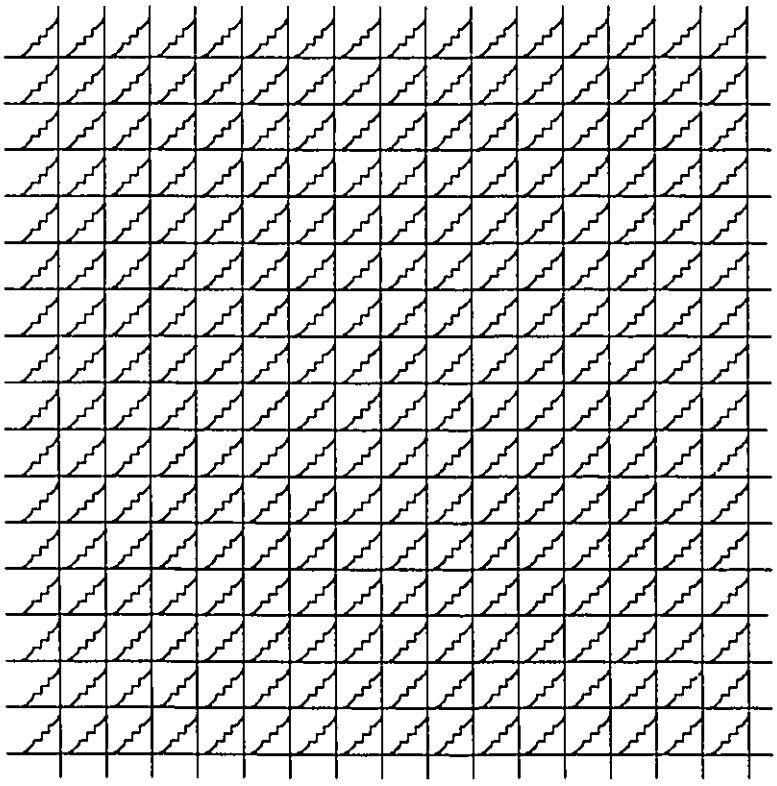


Figure 2.01 - 16 by 16 FSR equivalent resistance matrix

An elastic overlay is added to the FSR to provide a displacement to force transduction that is required for the tactile sensor to achieve desirable interface properties. The operation of the FSR and the elastic overlay is further detailed in the sections to follow.

2.2 FSR Properties

The mechanical structure of the FSR is illustrated in Figure 2.02. As shown by the diagram the FSR consists of two non conductive Mylar sheets with a Polymer material deposited on each. When the two sheets are in contact and an external force is applied to the Mylar, the resistance between the two sheets decreases in relation to the normal force. From each of the two Polymer areas a conductive tab is attached that can be connected and tracked to external electronic components. In practice an array of sensor elements is used, all of which are manufactured in a single printing process. This array consists of 16 by 16 FSRs all manufactured on two sheets of Mylar each approximately one inch square. These sheets form the sensor when brought into contact with each other. The 256 FSRs are arranged in rows and columns in a manner similar to Figure 2.01 and can be scanned using any of several techniques [Meye91]. The simplest method of scanning the array is to apply a voltage to a single row with all other rows excited by a zero potential. No current will flow into the column connection except from the FSR excited by the voltage at the selected row. A selected column is then connected to a virtual ground. Current into the virtual ground is converted to a voltage by an operational amplifier circuit. Each column so connected, may be scanned in sequence to find the current resistance of a single FSR. The selection of the row and column by this method and the conversion of the analog voltage to a digital equivalent has effectively found the current resistance of an FSR. Figure 2.03 depicts this process where the row R_1 has been selected and the current I_{11} flows into the operational amplifier attached to column C_1. The output voltage therefore follows the relationship $V_1 = G_1 \bullet I_1$ where, G1 is the gain of the amplifier and $I_1 = V_{ref}/FSR_{11}$ [Pet92ii].

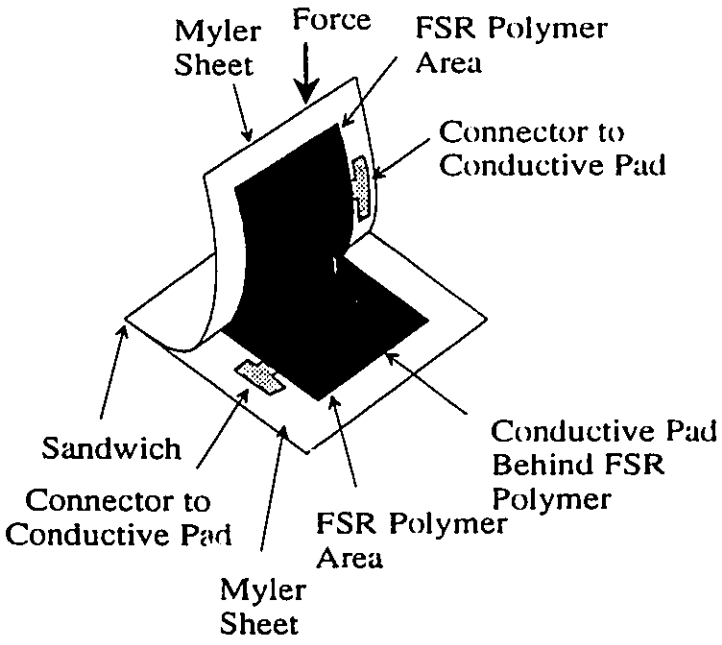


Figure 2.02 - FSR Physical Configuration

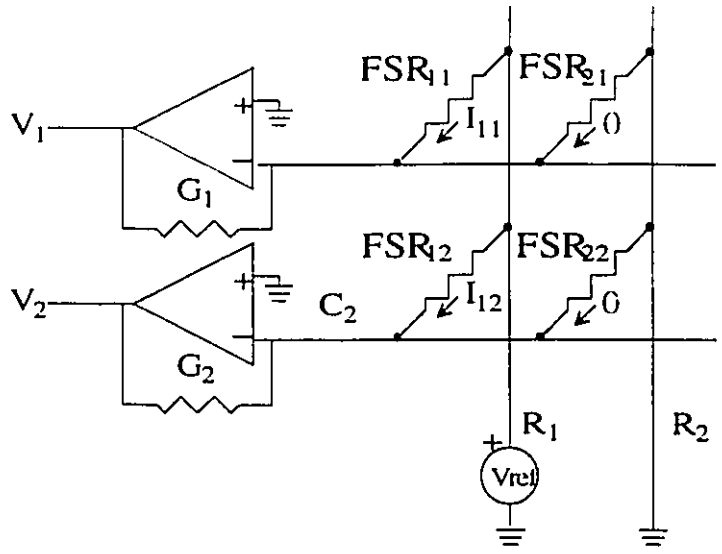


Figure 2.03 - Determining value of resistance FSR_{11}, FSR_{12}

By changing the row to which the reference is attached, it is possible to determine the value of all the resistance's in the matrix. Figure 2.04 presents, a graph of the resistance of the FSR versus the applied pressure. As shown, the relationship between the two is nonlinear. In addition the resistance can vary over two decades of value with pressures between .05 kg/cm^2 and 5 kg/cm^2 where, the practical range of values for pressure falls between these two extremes.

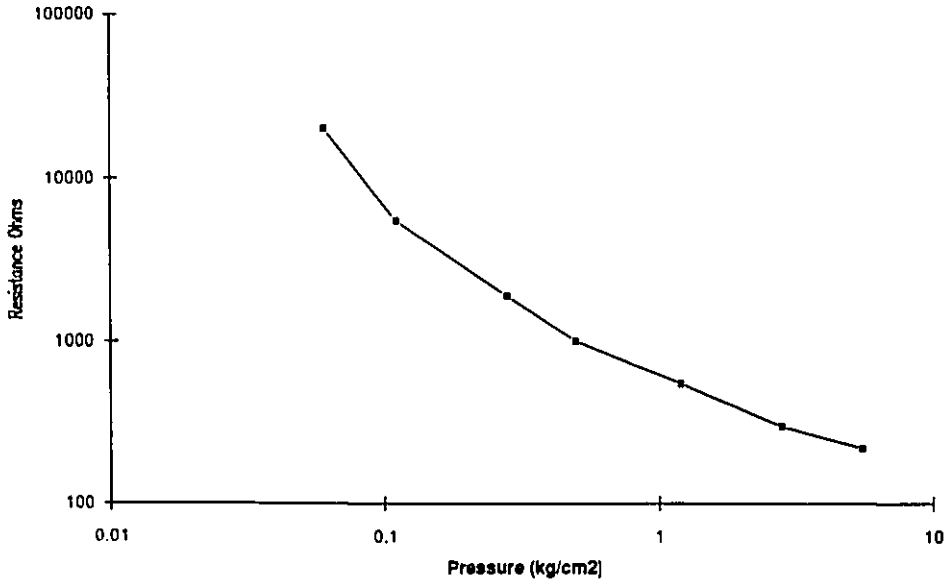


Figure 2.04 - Resistance of FSR versus pressure

2.3 Elastic Overlay

A compliant elastic pad is overlaid over the FSR to form the tactile sensor [Pet92i]. The FSR is a force sensitive device and the tactile sensor requires displacement sensitivity to avoid the problem of point contact. Figure 2.05 and 2.06 illustrate the problem. In Figure 2.05 a small letter M embossed on a wooden block is captured by the sensor. Figure 2.06 depicts the same letter M as captured without the elastic overlay. In the second figure, only the highest points of the letter are input to the sensor and most of the information in the image is lost. This problem occurs because the FSR is essentially a ridged device and therefore, only the highest points of the object make contact with the surface. However, when the elastic overlay is present the displacement of the surface of the overlay is proportional to all the surface features of the object in question and therefore, all the desired information is present in the captured pattern. These surface strains are translated to stresses at the contact between the pad and the FSR and therefore, to varying resistance's by the FSR. Figure 2.07 illustrates the operation of the sensor with the elastic overlay. Using the elastic overlay, the displacement of the surface caused by all the points of the object is translated to forces at the FSR.

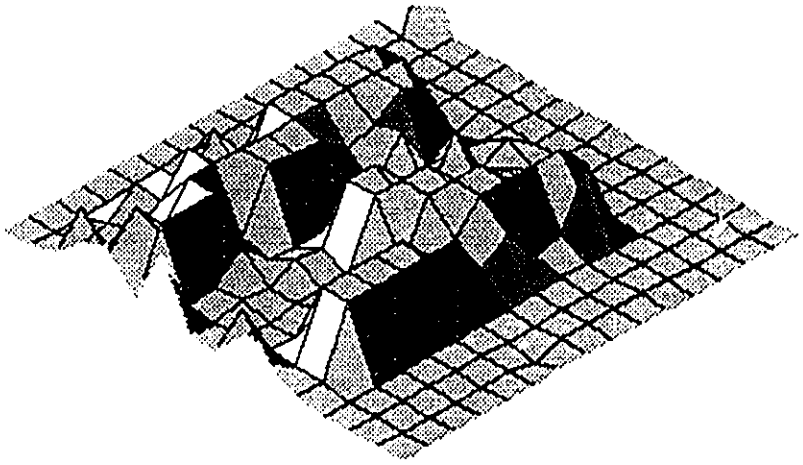


Figure 2.05 - Letter M as captured with elastic overlay

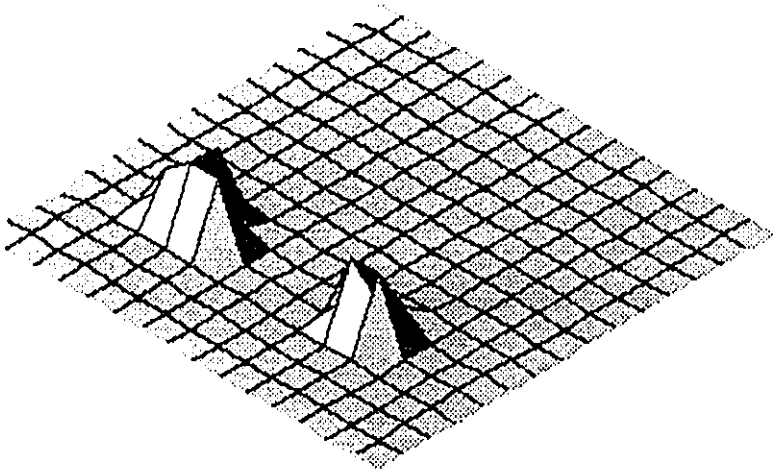


Figure 2.06- Letter M as captured without elastic overlay

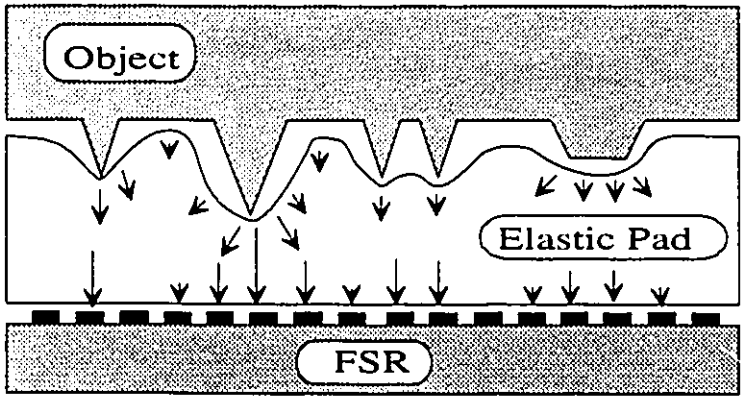


Figure 2.07 - Object impacting overlay and transmitting force to the FSR array

The conversion of displacement to force is a non linear function of the elastic pad and as shown by Figure 2.07 there is a significant amount of cross talk between the elements as the displacement at the surface translates to forces along axis's not aligned to the vector of the primary direction of movement. The relationship between displacement, force and measured resistance is depicted in Figure 2.08.

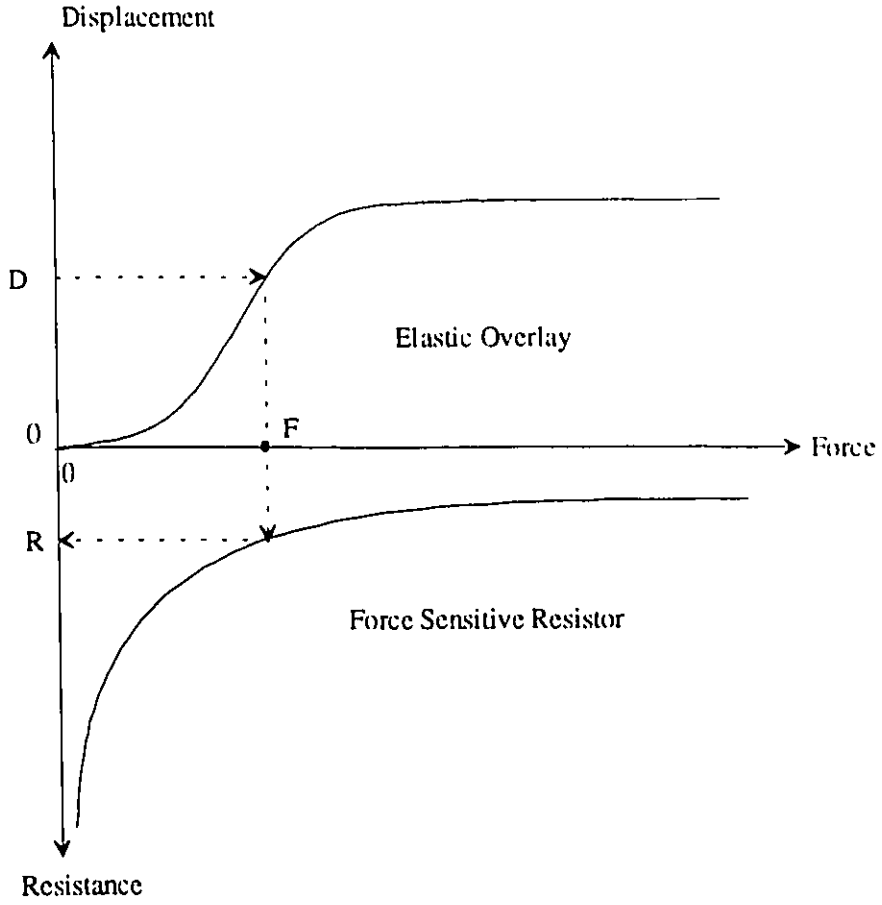


Figure 2.08 - Relationship of force at the sensor to displacement of the elastic overlay

Although, the relationship between the displacement and the measured resistance is nonlinear the ability of a neural network to adapt to such problems is well documented in the literature [Pati92] and therefore, this type of relationship does not complicate the recognition of objects using the nonlinear adaptive filtering techniques employed by the neural network. In addition, the actual practical operation of the sensor is within a small range of input forces and displacement. Therefore, the drastic non-linearities implied by Figure 2.08 are not apparent for most practical scenarios (see chapter 4 for experimental results).

2.3.1 Overlay Design

The design of the overlay is very important in achieving both reduction of cross talk and the desired characteristic displacement to force transfer function. In principle, a geometrically fully constrained one-piece elastic pad cannot be compressed. This situation is depicted in Figure 2.09 where, both the sides in the x, y plane and the bottom of the elastic pad (z direction) are constrained. In practice some displacement will occur as depicted in the diagram for reasons outlined by paragraphs to follow. Hooke's law may be applied to a small unit cube of the elastic material per Equation 2.01.

$$\varepsilon_x = \sigma_x - \frac{(\sigma_y + \sigma_z)}{2} \Big) / E$$

$$\varepsilon_y = \sigma_y - \frac{(\sigma_x + \sigma_z)}{2} \Big) / E \quad \text{Equation 2.01}$$

$$\varepsilon_z = \sigma_z - \frac{(\sigma_y + \sigma_x)}{2} \Big) / E$$

Where, E is the modulus of elasticity, σ is the stress and ε is the strain.

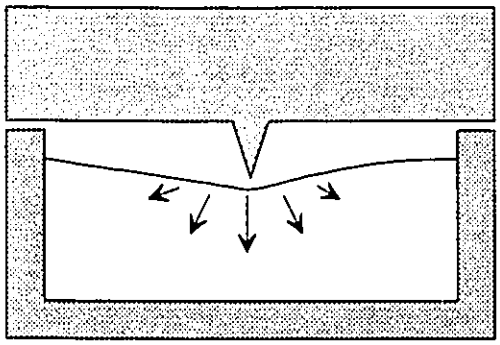


Figure 2.09 - Compression in a one piece elastic pad

The x, y plane is aligned with the pad's surface and the force of the object impacts the elastic pad in the z axis. If the pad is attached to the rigid surfaces of the sensor, then no displacement in the x or y direction can occur when pressure is applied to the surface. Therefore, ε_x and ε_y are both identically zero per Equation 2.02.

$$\varepsilon_x = \left(\sigma_x - \frac{(\sigma_y + \sigma_z)}{2}\right) / E = 0$$

$$\varepsilon_y = \left(\sigma_y - \frac{(\sigma_x + \sigma_z)}{2}\right) / E = 0$$

$$\therefore \quad \sigma_y = \frac{(\sigma_x + \sigma_z)}{2} \qquad \text{Equation 2.02}$$

$$\sigma_x = \frac{(\sigma_y + \sigma_z)}{2}$$

$$\therefore \sigma_x = \sigma_y = \sigma_z$$

From this relationship it can be shown that ε_z is also zero and therefore, no displacement can occur in the z direction. In practice, the pad is compressible since the pad is not a unit cube but must be described by many unit cubes and the surface of the pad is not fully constrained. Because of this property, the surface displacement in the z axis causes displacement in the x and y axis's and distortion of the pad in the z axis in parts of the elastic next to where the original displacement occurs. Consequently the pattern of forces of Figures 2.07 and 2.09 is observed and is therefore, the origin of crosstalk at the sensing elements. The solution to the problem is to develop a pad where the x and y axes are free to move without reduced impact on the adjacent areas of the elastomer. This not only allows the pad to move freely in the z direction and therefore, transfer force to the FSR array but also reduces cross talk by preventing forces from one section of the elastic pad being transmitted to another through translation of forces to the x, y plane and back to the z. Figure 2.10 illustrates such a design. This pad consists of a thin membrane with protruding cylindrical tabs. These tabs, allow the material to expand without any stress in the x and y directions making possible its compression in the z direction. The tabs are arranged such that each is atop one element of the FSR matrix and therefore, this arrangement also provides a *de facto* spatial sampling. The physical shape and size of the pads have considerable effect on the sampling error of the system, phenomena similar to that found in the development of quantizers of all forms. Based on past research and the recommendations of [Yang85] and [Groo88] the circular tabs were designed to occupy 50% of each sampling area.

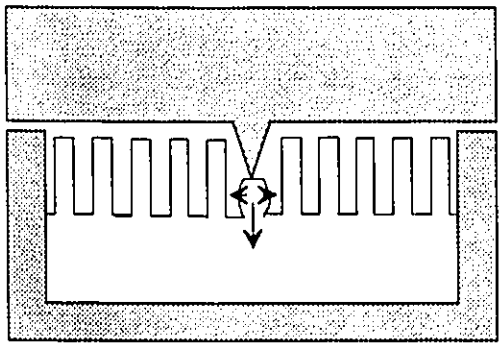


Figure 2.10 - Elastic overlay with cylindrical protruding tabs

With the addition of the elastic overlay of Figure 2.10, the design of the tactile sensor is complete. Chapter 4 describes the performance of the tactile sensor and neural network combination as a tactile pattern recognition system.

Chapter 3
A Discussion of Neural Networks

3.1 Background

The history of artificial neural systems did not start with Marvin Minsky's and Seymour Papert's 1969 book "Perceptrons" [Mins69] but it almost ended there. In this book some "wishful thinking" about the capabilities of adaptive threshold elements are dispelled. The early history of neural nets, before 1969, was based more on simplified biological models of real neurons than on pattern recognition science or signal processing principles. The early pioneers of the field were biologists and physiologists not engineers and computer scientists. For these people it was the modeling of the building block of brain function, the "neuron" that was important. A quotation from Patrick K. Simpson's book [Simp90] serves to illustrate "The model neuron in its simplest form ... can be considered a threshold element that collects inputs and produces an output only if the sum of the inputs exceeds an internal threshold value. As a threshold unit, the neuron collects signals at its synapses and adds them. If the collected signal strength is great enough to exceed the threshold, a signal is sent down the axon which abuts other neurons and dendrites. The soma adds all the signals, gated by the synapses from the impinging dendrites. The total signal is then compared to an internal threshold value of the neuron and propagates a signal to the axon if the threshold is exceeded." Figure 3.01 presents a simplified representation of a biological neuron.

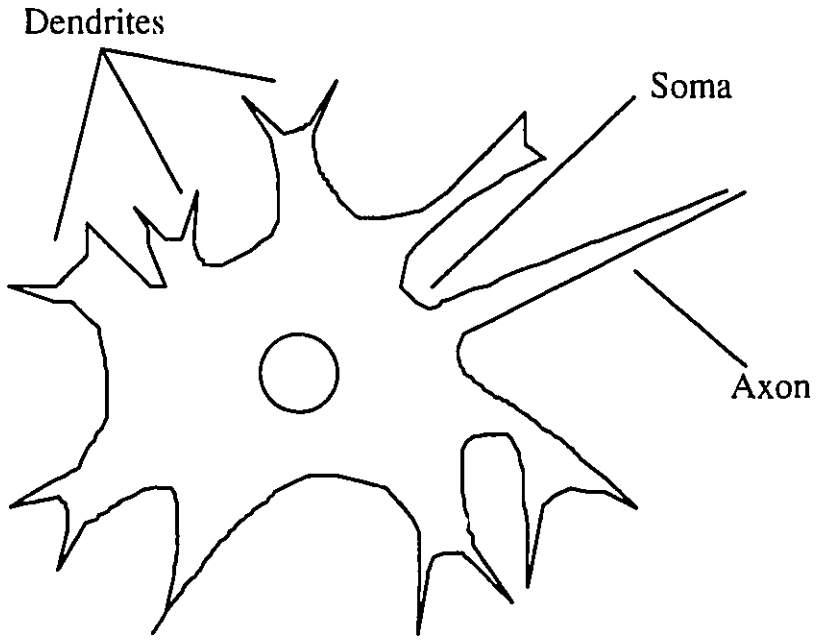


Figure 3.01: Simplified Biological Neuron

It was noted by early researchers that this simple threshold processing element repeated at least 10^{11} times was the basis for all abstract human thought. Thus, it seemed possible to

model the operation of the neuron and thereby, produce an automata capable of learning and reasoning. Apparently a single neurons cannot solve the complex problems that a human being can, however, it was speculated that a single artificial neuron could "learn" and perform simple "reasoning." Donald Hebb originated Hebb's law to explain how a neuron learns: "When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency as one of the cells firing B is increased," from the book *Organization of Behavior* [Hebb49]. In its simplest form this rule can be interpreted as a method for modifying the synaptic connections (weights in the artificial network) based on the excitation of both the feeding and receiving neurons. If both neurons are in an excited state then the connection is strengthened, if both are in a relaxed state then the connection is weakened. Thus, the connection is modified based on the product of the excitation states of the two neurons. Given this model for neural learning and the idea of the biological neuron, research was conducted into creating artificial constructs with similar properties. Figure 3.02 shows how this is commonly achieved.

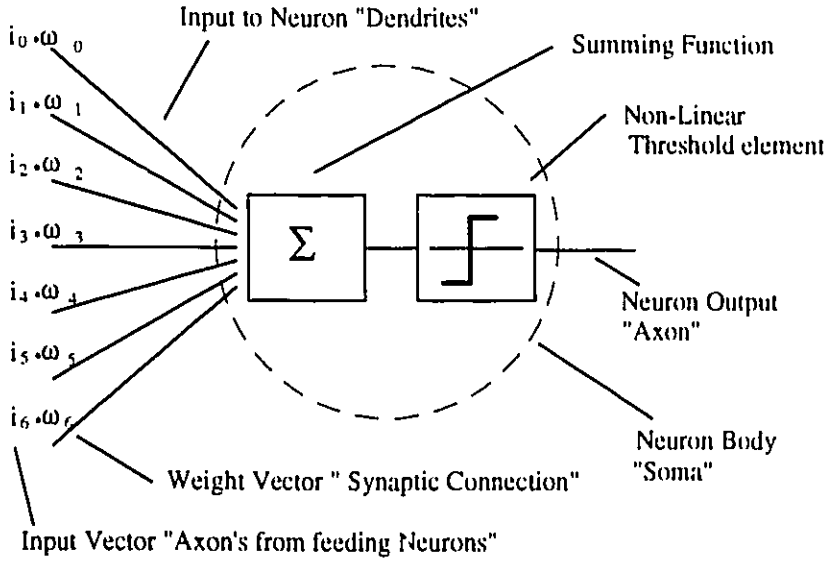


Figure 3.02: Artificial Neuron Model

The artificial neuron models the input from other neurons or the outside world as a vector of continuous values i_0 through i_n. This input vector is multiplied by a weight vector to model the effect of varying conductivity at the synapses. The sum of the products is then performed and the result input to a nonlinear threshold function. In the simplest models, the output is "1" if the sum of the products is greater than the threshold and "0" otherwise. Mathematically this operation can be represented by equation 3.01:

$$\Omega = \sum_{k=0}^{n-1} i_k \cdot \omega_k - \theta_n$$

Where:

Ω = The output of the neuron — Equation 3.01

i_n = The value of the inputs

ω_n = Weight value

θ_i = Variable threshold

The question remains, how does the performance of this "neuron" relate to that of the biological model? It is at this point that the work of Minsky and Papert becomes important. Their composition is essentially a treatise on the limitations of Perceptrons, which are general purpose threshold elements of the type previously described. The name Perceptron, is credited to Rosenblatt [Ross57] however, it came into common usage with the work of Minsky and Papert. A Perceptron is a generalization of equation 3.01 where the i_k do not depict single values in an input space but are generalized to represent predicates acting on a local group of points in space. Therefore, the model of an artificial neuron presented earlier is a specific example of a Perceptron.

The book "Perceptrons" presents proofs on the limitations of single Perceptrons. Before its publication the properties of the artificial neuron had not been explored in a rigorous sense. In fact the Perceptron was considered to preserve the properties of learning and generalization present in human brains, although on a simpler scale. It was this type of mythological thinking that "Perceptrons" attempts to dispel. As Minsky expounds "Our purpose is to explain why there is little chance of much good coming from giving a high-order problem to a quasi-universal Perceptron whose partial functions have not been chosen with any particular task in mind. It may be argued that people are universal learning machines and so a counterexample of this thesis. But our brains are sufficiently structured to be programmable in a more general sense than the Perceptron . . . " [Mins69]. The primary thesis of the book was that the Perceptron can solve only those problems that are linearly separable. However, evidently the recent history of neural networks has produced many promising and useful results in apparent contradiction to this thesis.

The proofs of the capabilities of the single Perceptron are not being refuted. The usefulness of modern networks is not in the use of single neurons but in multiple layers of connected neurons. It can be shown that a network with only three layers of neurons can solve any general classification problem (see the section 3.3.1). The following quotation from Michael W. Roth editor of IEEE Transactions on Neural Networks [Roth91] illustrates these points "Although the Minsky and Papert book proves theorems about the limitations of single-layer feedforward networks, it goes on to identify several technical

challenges about the potential, in general, for neural networks to perform important computational tasks. Many feel that this subsequently led to discouraging agencies from funding neural network research." He goes on to say " One unfortunate aspect of this history is that these specific technical challenges went largely unanswered until relatively recently. Two such examples that have been overcome only in the past few years are the exclusive-OR and connectedness problems that the book posed."

3.2 Network Types

There are two basic types of neural networks, those that require supervised learning and those that learn in an unsupervised sense. In addition, within these global categories several subclasses can be identified each of which has specific properties and limitations.

Networks that learn in an unsupervised fashion have been devised to classify with no *a priori* knowledge of the types of classes that are required. An unsupervised network is presented a set of images to be classified, it then groups these patterns into classes based on some commonality criteria inherent to the network type. The "Self-Organizing Feature Maps" of Kohonen [Koho84] are examples of this type of network. As the following quotation suggests these types focus on the statistics of the training set: "... an almost optimal spatial order, in relation to signal statistics, can completely be determined in simple self-organizing processes under the control of received information." However, for this thesis it is desirable to train the network to recognize predetermined classes of patterns. This type of operation requires supervised training.

In the class of networks using supervision, the types and classes of objects are known before the training phase. The network is required to learn "generalization" from the presented examples and thereby, can classify patterns not in the training set. The most common form of unsupervised network used for classification purposes, is the general feed-forward network presented by Figure 3.03. This network structure is used throughout the development of various networks in this thesis.

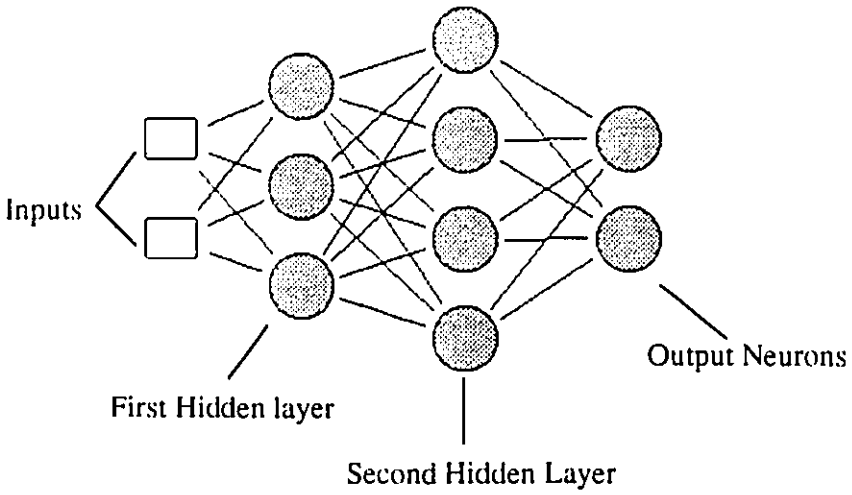


Figure 3.03: Generalized Feed Forward network

The generalized feed-forward network has several identifiable subsections. The input layer is simply a depiction of how the input vector is distributed to the first layer of neurons. In the example, all inputs are connected to all neurons. The output layer is the last layer of neurons that decide the results of network classifications. The intervening or "hidden" layers of neurons perform intermediate stages in the calculation. For practical classification problems, the network would consist of two or more layers. The following sections address the limitations and methods of classification done by this type of structure.

3.3 Characteristics and capabilities of the Feed-Forward structure

3.3.1 Classification in the Feed-Forward structure

As determined by Minsky and Papert [Mins69], a network structure consisting of a single layer of neurons (i.e. the inputs are connected directly to the output neurons) can do a linear separation of classes. Figure 3.04 presents a simple case of this type of classification. Here a network with two inputs and one neuron, divides the domain into two classes. The vertical axis depicts the range of an input (limited between 0 and 1) and the horizontal axis depicts the range of the other input (also limited between 0 and 1). The two classes are determined by the identified areas. As is apparent from the figure, the delineation between the classes is a linear boundary.

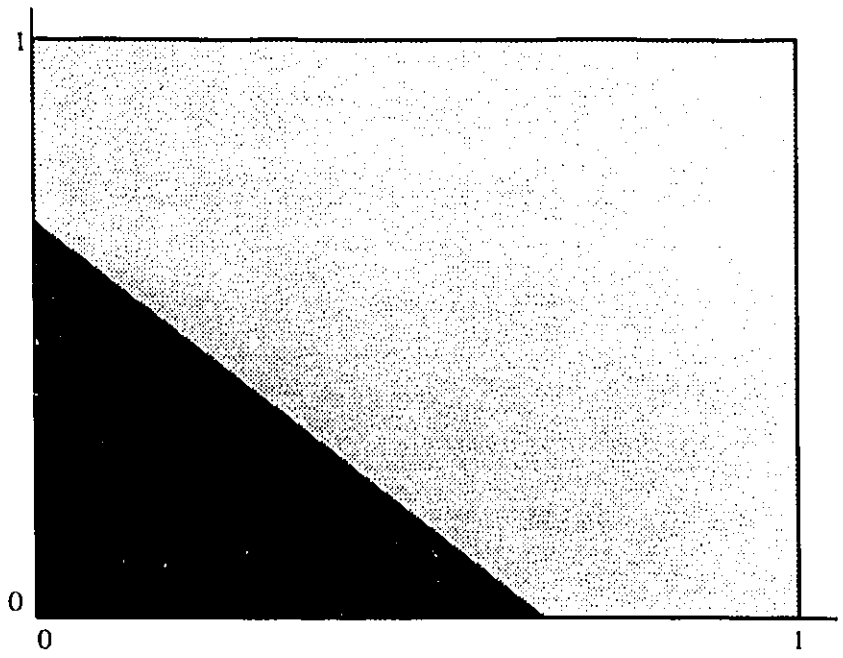


Figure 3.04: Linear Classification performed by single neuron

In the general case of networks with more than two inputs, the delineation of classes is along hyper-planes of dimension one less than the number of inputs. This fact is illustrated by examination of equation 3.01. The division between the two classes occurs when the output is either positive (indicating one class) or negative (indicating the other). Therefore, when the sum exceeds the threshold one class is implied, when it is below the threshold the other class is selected. Therefore, the division of classes is at the equality of the summation and the threshold (Equation 3.02).

$$\theta_i = \sum_{k=0}^{n-1} i_k \cdot \omega_k \quad \text{Equation 3.02}$$

Equation 3.02 is the generalized equation of a hyper-plane in n dimensions. Figure 3.05 depicts the case for a three dimensional space.

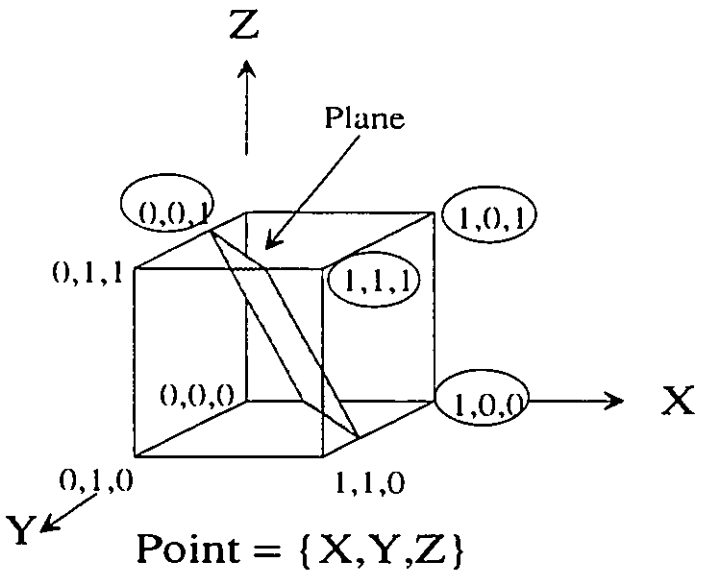


Figure 3.05: classification by a neuron with three inputs

The two dimensional plane identified in the figure separates the space into two volumes. The vertices included in the classification are circled, those not included are left un circled. In the general case, any set of connected vertices[1] can be included by this classification. As more neurons are added to the layer multiple parallel linear separations are possible.

As the order of the network is increased, from one layer to two, more complex and general classifications become achievable, Figure 3.06 illustrates this point by example. As with the single neuron, two inputs of limited range are illustrated with a single output neuron. The number of neurons in the hidden layer is found by analysis.

[1] A connected vertix implies that if two vertices are contained within the classification then they are either nearest neihbours in the cube or that any intervening nodes are also included in the set.

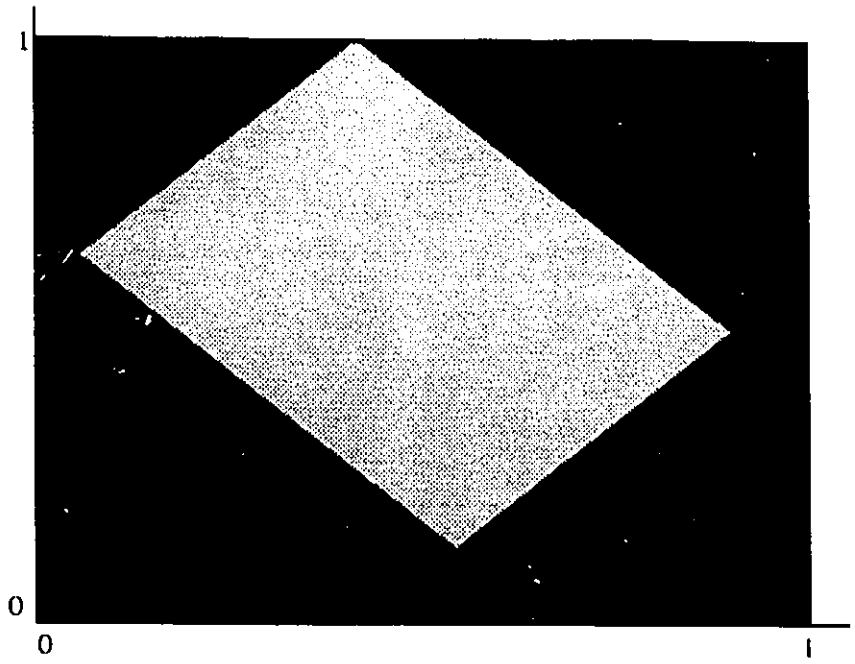


Figure 3.06: classification by a network consisting of two neuron layers

Since, the outputs of the first layer of neurons are essentially binary numbers produced by the threshold elements, the classification by a neuron in the second layer (and all subsequent layers) is a selection of the set of connected vertices of the hyper-cube. Thus, the second layer of neurons manipulates weighted functions of binary numbers and divides the input space into two classes based on a set of adjacent corners of a hyper-cube (the three dimensional case is illustrated in Figure 3.05). One useful special case can be identified from the arguments to follow. Consider a single neuron in the second layer, the neuron weighs the binary inputs compares the sum to a threshold and determines if the sum exceeds the threshold. Therefore, for a given state of the variable inputs as expressed by Equation 3.02, if the output of the chosen neuron in the second layer is in a certain state for a given set of ck (i.e. it has classified the inputs into one of the two possible states of its output).

$\forall k, k < n, i_k = c_k$

Where: Equation 3.03

c_k = constant binay value

Then consider the effect of a change of state of a c_k selected at random c_j. Equation 3.04 expresses this operation mathematically.

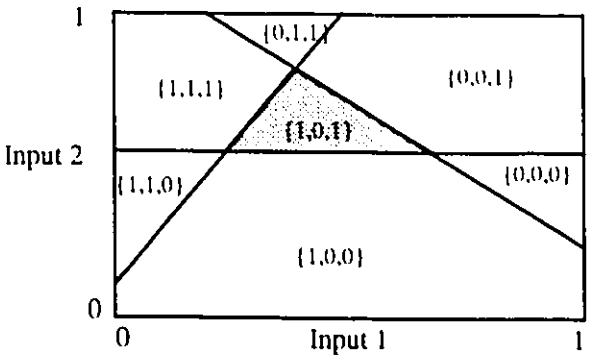
$c_i = F(c_k), k < n$

Where:

c_i = constant binay value Equation 3.04

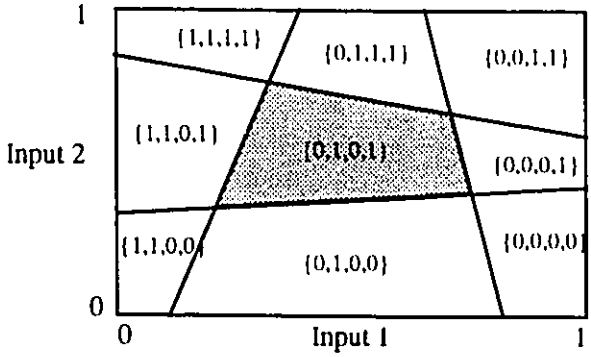
F() = random c_k selection function

The selected c_j is state changed such that $c_j = \bar{c}_j$. Two possibilities are apparent: either the weighted sum of the c_k's is changed sufficiently that the threshold is exceeded and a classification boundary is crossed or else no change of state occurs. If no change of state occurs then the two classes of the first layer separated by the linear boundary, are joined in the second layer. If a change of state does occur, then this boundary is also a boundary between classes in the second layer. If for every member of $\{c_0, c_1 \ldots c_{n-1}\}$ the procedure of inversion causes a change of state, then a single state defines the boundary of classes (i.e. a single vertex of the hyper-cube is selected). This region is convex with the number of sides equal to the number of neurons in the first layer. Figure 3.07 shows how this occurs for a structure with two inputs, three or four neurons in the first hidden layer and a single neuron in the output layer.



{a0, a1, a2 } state of outputs of the hidden neurons.

(a) Hidden layer contains three neurons.



{a0, a1, a2, a3 } state of outputs of the hidden neurons.

(b) Hidden layer contains four neurons.

Figure 3.07: Enclosed polygons defined by networks of two inputs and one output.

Therefore, the second neuron layer (hidden or output) can classify the input space into multidimensional polygons with the number of sides less than or equal to the number of neurons in the previous neuron layer. In the limit more complex classifications are possible however, the convex polygon will be used as the basis for assessing the minimum capabilities of a network consisting of two neuron layers.

As previously mentioned, it requires three layers of neurons to achieve a general classification scheme, a result showed by the following arguments: The outputs of the second layer of neurons classify the inputs into polygons of dimension equal to the dimension of the input layer and with the number of sides equal to the≤ number of neurons in the first neuron layer. The third layer of neurons can classify the outputs of the second neuron layer in the same manner as the second layer classified the outputs of the first layer, into collections of vertices of a hyper-cube. A useful subset of this classification is that a single state may be selected. This state would be the union of polygons defined by the second layer and therefore, any connected or disconnected shape can be classified by addition of enough polygons. This is shown mathematically as follows:

Let $P = \{ P_k, k = 0, n-1 \mid P \in \{\text{hyper-polygons}\} \}$

Let $I = \{ i_j, j = 0, m-1 \mid 0 < i < 1 \}$ be the input vector to the network

Let $F(I, P_k)$ be a binary function with the following formulation:

$F = 1$, if I is a point in P_k

$= 0$ else

$\therefore F(I, P_k)$ represents the output of a neuron in the second hidden layer.

Let $O_k = F(I, P_k)$ be the output of a neuron in the second layer

$\therefore O = \{ O_k, k = 0, l-1 \}$ is the state of the outputs of the second layer

The output of a neuron in the third layer can be activated by a single state in the second layer. A classification function is thereby developed that can stipulate which polygons to accept as follows:

$$\Omega = \sum_{k=0}^{l-1} a_k \times \overline{O}_k - \theta_t \qquad \text{Equation 3.05}$$

Where Ω is the output of the neuron before thresholding. The a_k are selected as follows:

$a_k = 1$ if O_k is to be selected

$= 0$ else

$$\theta t = \sum_{k=0}^{l-1} a_k - 0.5$$

Thus, if Ω is greater than zero then an area external to all the polygons is selected. Since, the function is required to show if the point was in any of the polygons then what is required is the inverse of this and therefore:

$\omega = 1$ if $\Omega < 0$

$= 0$ else

Therefore, the equation can be rewritten in Boolean form:

$$\omega = (a_0 \bullet O_0 + a_1 \bullet O_1 \ldots a_l \bullet O_l)$$

$$\omega = (a_0 \bullet F(I, p_0) + a_1 \bullet F(I, p_1) \ldots a_l \bullet F(I, p_l)) \qquad \text{Equation 3.06}$$

In conclusion, any neuron in the third neuron layer can classify the input space into an arbitrary shape consisting of convex polygons of arbitrary size and dimension. Figure 3.08 continues the example of classifications using a network with two inputs, two hidden neuron layers and a single output.

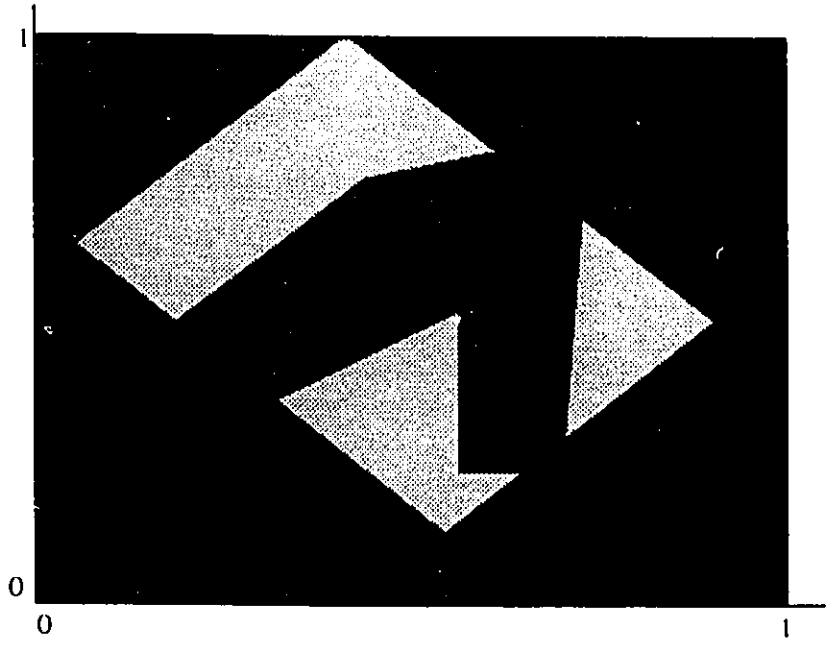


Figure 3.08: classification by a network consisting of three neuron layers

3.4 Learning in the Feed-Forward structure

3.4.1 Memorization vs. Generalization

There are many references in the literature in relation to the concepts of memorization and generalization. These are mutually exclusive concepts as used in regard to neural networks. The desirable property, "generalization" refers to the ability of a network to produce the correct classification when presented with a pattern not in the training set. The contrary property, "memorization" is the idea that the network has simply memorized the classifications for its training set. Therefore, it can correctly identify those patterns in the training set but any other new examples are not classified correctly. Figure 3.09 is an example of the boundaries between classes as identified by a network that has memorized the examples.

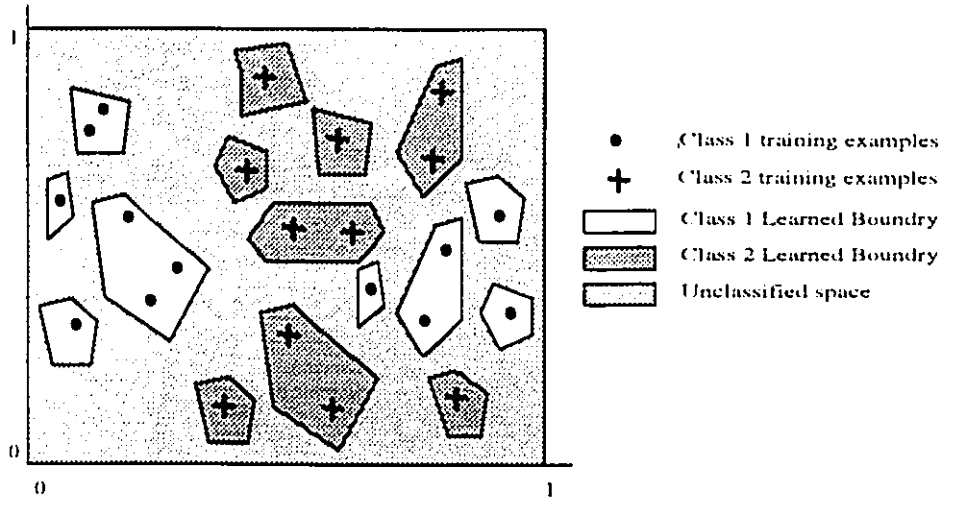


Figure 3.09: classification by a network that "memorizes"

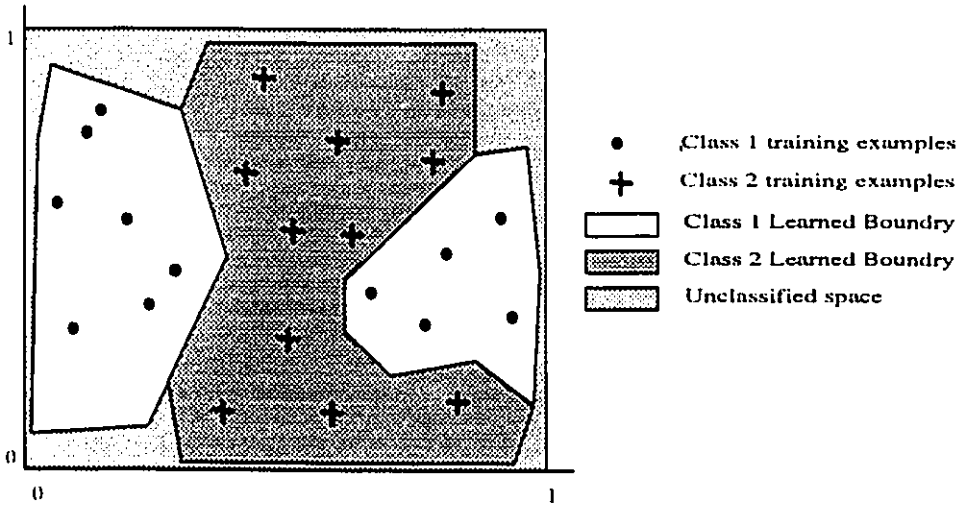


Figure 3.10: classification by a network that "generalizes"

Figure 3.10 depicts a network that has successfully learned to generalize from its training examples. Evidently the memorization process requires a more complex classification scheme (with more disconnected regions) than does the generalization process. From these results, the fewer neurons a network consists of, the better the generalization properties of the network. However, a minimum number of neurons are required to correctly describe the boundaries between classes. As shown in Figure 3.11 a lack of neurons usually results in misclassification of one or more of the training examples. In other words, the network can never converge. This property has been observed experimentally when developing the networks outlined in chapter 4. In addition to limitations in the network topology a second factor may lead to poor generalization, this phenomena is caused by training the network with insufficient examples of the proper type (the desired type of training examples will be the focus of section 3.3.1). In this case the

class boundaries are improperly learned and misclassifications can occur. A solution to this problem is presented in section 3.4.3 that focuses on training the network.

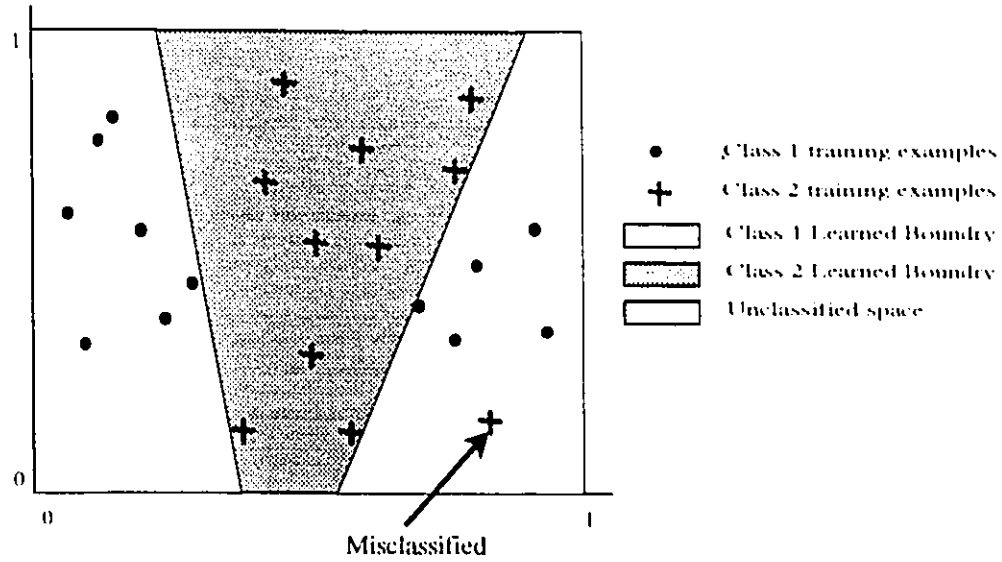


Figure 3.11: network with too few neurons to classify correctly

Therefore, the choice of the number of layers and the number of neurons in each layer greatly affects the ability of the neural network to act as wanted.

3.4.2 Determining network topology

The determination of a topology of a network suitable to the solution of a given problem has historically been a trial and error process. This paradigm may work well for small problems but proves time consuming for problems of reasonable complexity. In the general case, during network development, it is desirable to start with a minimal structure and add neurons should this topology prove insufficient to the task.

The reasons for this are two fold: too many neurons will "learn" to classify the training set but the property of generalization may not be achieved and a smaller network converges faster and if it cannot classify the patterns correctly, will not converge at all. In addition lack of the generalization property may take considerable time in testing to discover.

Since the deficiency of resources becomes apparent during the training phase and the network learns quickly. It is desirable to start with a minimum of network resources and increase as necessary. It is therefore desirable to understand the lower limit on the number of neurons required for training as this will serve as a starting point for network development. The obvious lower bounds are one neuron in each of the two hidden layers and as many output neurons as classes (the activation or "1" state of a given output neuron showing that the input point belongs to a given class). This axiomatic starting topology works well for networks designed to perform trivial classification schemes but proves time consuming for networks of greater complexity. Mehrotra *et al* [Mehr91] show that a lower bound for the number of neurons in the first hidden layer can be found if

one knows the number of clusters required for the classification. A cluster is a disconnected region in hyper-space (this definition is essentially identical to the determination of the hyper-polygons outlined in the previous section). The number of clusters therefore, corresponds to the number of neurons in the second hidden layer. In addition, the number of clusters is always ≥ the number of classes since, a class must contain at least a single cluster (a trivial exception is where one class and another are exact inverts of each other and therefore, share the same clusters). This relationship is expressed by equation 3.07.

$$M \geq O$$

Where:

M = Number of clusters (2nd hidden nodes)

O = Number of classes (outputs)

Equation 3.07

Mehrotra *et al* derive a relation between the number of clusters and the number of neurons in the first hidden layer (Equation 3.08):

$$R(n,d) \geq M$$

$$R(n,d) = \sum_{i=0}^{d} C_i^n$$

Where:

R(n,d) = maximum number of regions

$$C_i^j = \frac{j!}{i!(j-i)!}$$

n = number of hyperplanes (1st layer nodes)

d = dimension of input layer

Equation 3.08

From Equation 3.08 it is possible to find the lower bound on the number of neurons in the first layer by setting R(n,d) = M and determining the value of n (d is given by the required number of inputs and is therefore, known). It is apparent that the better the estimate of the value of M the more accurate the results of the method. Therefore, the heuristic that the number of clusters (M) is greater than or equal to the number of classes may prove too loose a criterion. However, the actual number of clusters required is entirely dependent on the specific classification problem. In most neural network applications it is likely that no *a priori* knowledge of the required clustering is available. In fact it is an often stated benefit of the neural network paradigm that no *a priori* knowledge is required to solve a specific classification problem. However, the number of clusters required is inherent in the training set itself and therefore, it should be possible to estimate the required number of clusters from examining the characteristics of the training set. In practice this may not yield a complete solution as there is no guarantee that any training

set chosen at random, has sufficient information to correctly determine the number of clusters for the following reason: It is considered axiomatic that at least a single sample per cluster (and in practice more than one) should exist in the training set for the network to correctly classify in the general case. Since, there exists no practical way to determine that a training set contains sufficient and necessary information to achieve this goal, the training of the network generally requires several iterations (see section 3.4.3 on Training the Feedforward Network).

3.4.3 Training the Feed-Forward Network

Once the topology of the network is determined by the techniques outlined in the previous sections, (this may require iteration as the process proceeds) the network must be "trained." The training phase consists of the following steps:

i) Set all weights in the network to random values between -1 and +1;

ii) Select a training set of examples from all classes;

iii) Determine the network output from one of the patterns in the training set;

iv) Compare the output of the network to the desired values;

v) Determine the "error" associated with each neuron, typically based on the difference between the desired and actual outputs;

vi) Adjust the weight matrix for the neurons based on the "error";

vii) Select a different pattern from the input set;

viii) If all patterns have been presented to the network then go to to step ix) else repeat steps iii) to viii);

ix) Determine whether all "errors" are within acceptable range;

x) If "errors" are unacceptable then repeat steps iii) to ix);

Except for the choice of the training set all the above steps can be automated. However, at the termination of the procedure it is not guaranteed that the network has achieved the desired characteristics of generalization. Thus, a testing phase is required to decide if these properties have been learned by the network. The testing group of patterns must consist of a set of images different from that used during training (It is inherent that the training patterns would be correctly classified). The testing of the network proceeds in the following manner:

i) Select a pattern from the test set and apply to the previously trained network;

ii) Determine if the desired outputs of the network and the actual outputs match within some tolerance;

iii) If the outputs do not match then, the pattern has been incorrectly classified. If a match occurs then the pattern has been correctly classified;

iv) If the classification is correct, then discard the test pattern, else append the test pattern to the original training set;

v) Repeat steps i) through iv) until the training set has grown to sufficient size or the network has shown sufficient ability to classify (e.g., a specific percentage of tests are correctly classified);

At the end of the testing phase it is decided whether the network has achieved results sufficient for its desired application. The automation of this aspect of network development is involved due to the nature of the concepts to be learned. It is difficult to generalize a criterion for how well a network has learned as this is dependent on the particular task and therefore, the chosen criteria must be task specific. Should performance of the network prove inadequate then the training procedure is repeated using the modified training set. As the number of training patterns is increased it may be necessary to add neurons to the network as more complex class boundaries are required. The following subsections detail some steps outlined in the general development procedure outlined above.

3.4.4 Selecting the initial training set

Given the premise that no a priori knowledge of the distribution and class boundaries of the problem domain is available there exists no systematic procedure for selection of the initial training set. Mehrotra et al [Mehr91] have developed a formula that bounds the required number of training samples for specific conditions (Equation 3.09):

$$B = \Psi(\min(d, n) \bullet M)$$

Where:

B = Minimum number of boundary samples

M = Number of clusters — Equation 3.09

n = number of neurons in first hidden layer

d = dimension of input layer

Ψ = number of boundary samples required to define hyperplane

The application of this formula would be difficult in practice as the value of M is not known before selection of a training set sufficiently large to apply the procedures of the

previous section. However, an absolute minimum bound for B can be found before selection of the training set by assuming that M = O (number of outputs) and that $\Psi = 2$ (i.e. at least two boundary samples are required to separate clusters). Therefore, Equation 3.09 can be rewritten as Equation 3.10:

$$B = 2 \bullet (\min(d, n) \bullet O)$$ Equation 3.10

Where:

B = Minimum number of boundary samples

O = Number of outputs

n = number of neurons in first hidden layer

(where n = Function (O,d); equation &)

d = dimension of input layer

A boundary sample is one that lies close to the desired boundary between clusters (see section on updating training set). In practice the chance of choosing boundary samples at random is small and therefore, more training patterns than specified by Equation 3.10. In addition Equation 3.10 provides a lower bound with inherent assumptions that may also prove invalid in practice and therefore, the required number of training samples is again multiplied. Thus, while the equation may be used as a starting point for the size of the training set (i.e. a lower bound), it is unlikely that a training set of that size would prove adequate for most practical problems. Therefore, the initial training set is used as a starting point for network development and it must be refined through the iterative process previously specified.

In addition to the number of patterns in the training set, some thought must be given to the content of the set. In the literature it is often assumed that the input pattern set is in general position. That is, distributed throughout the space under consideration. Unfortunately, since no *a priori* knowledge of the distribution is known it is impossible to tell whether certain clusters have been left out or that only "good" examples of the data to be classified are chosen. Since no systematic and comprehensive way to collect samples of images exists, it is difficult to determine that all clusters are represented by one or preferably more samples. In addition, the case of automated or manual gathering of data there may be a tendency to collect only "good" examples of the class in question and therefore, the required boundary samples may be absent from the initial training set. Both these tendencies lead to tightly clustered initial training sets with some clusters missing. An example of the problem of missing clusters is given by Figure 3.12. Here the network is presented with three inputs and is required to learn the concept of two of them active (i.e. greater than 0.5). The example for Z active and X active is missing and therefore, the required cluster in this area will be missing. For this example the inadequacy is obvious by inspection of the set, for more complex examples this may not be so.

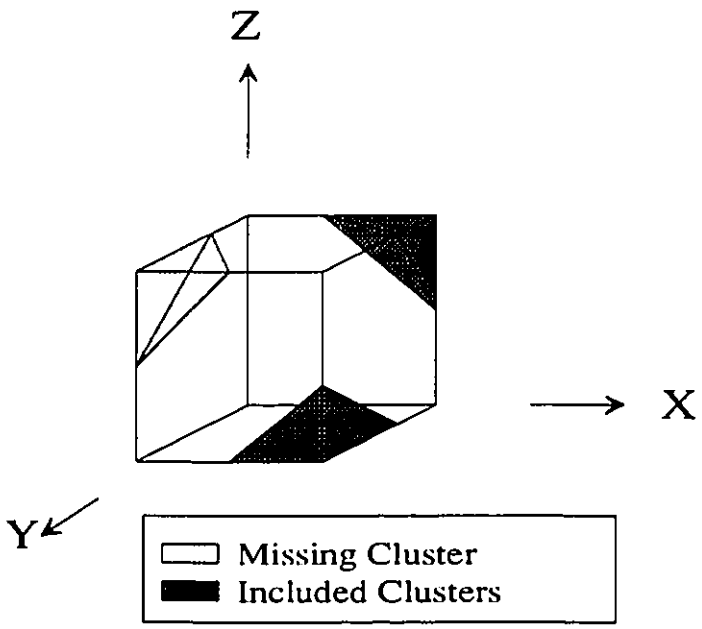


Figure 3.12: Example of missing cluster

The problem with only selecting "good" examples for the training set is displayed by Figure 3.13 and Figure 3.14. In Figure 3.13 the examples are all chosen well away from the boundary between classes and are therefore, labeled as good examples of the desired class. It is apparent that the learned boundaries between classes can vary from the desired boundary by a great degree. This phenomenon occurs because there are no samples in the training set that lie near the boundary and therefore, clearly defines its location. In Figure 3.14 boundary samples are included in the training set and therefore the degree of freedom of the classification border is more restricted. However, since no *a priori* knowledge of the location of the boundaries is assumed it is often difficult to predetermine whether a particular sample represents a boundary or not.

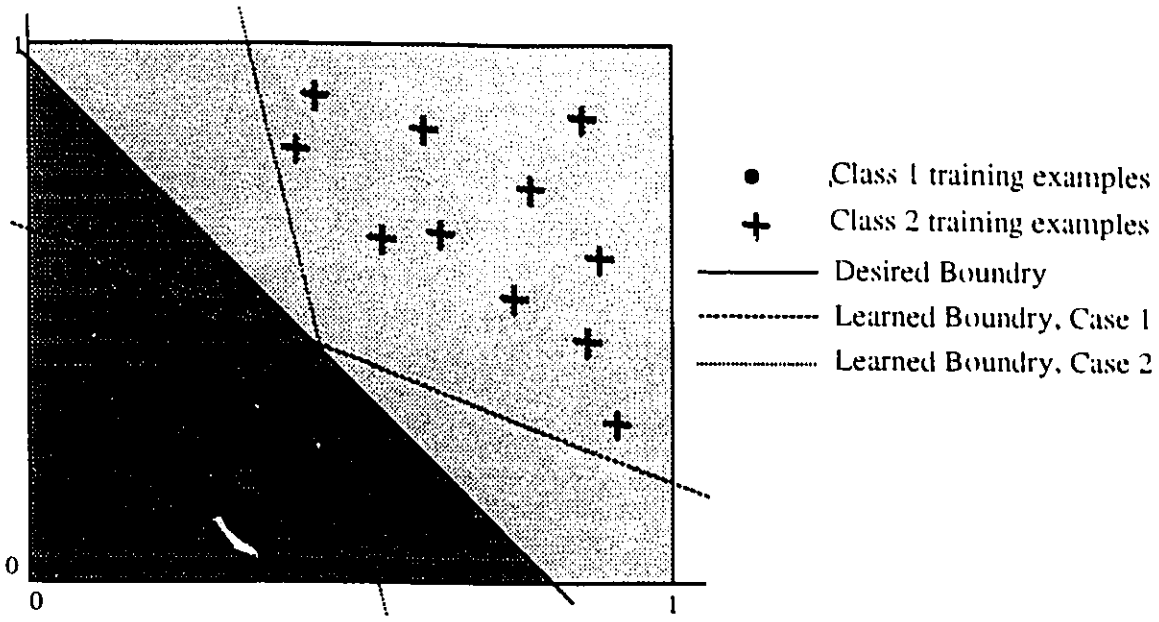


Figure 3.13: Possible boundary variation in the absence of boundary samples

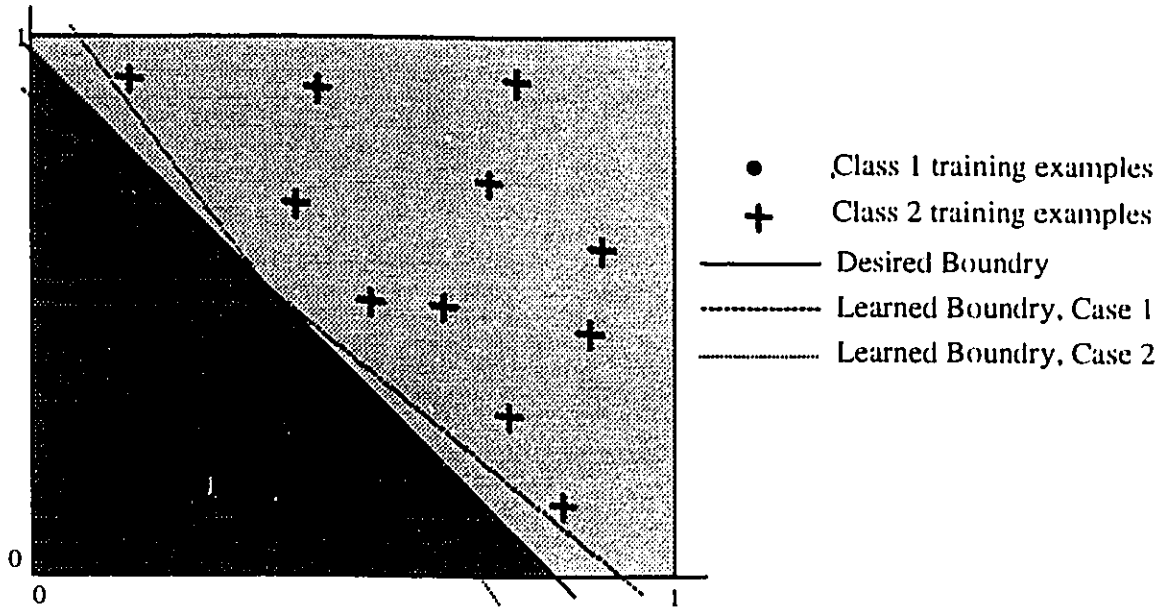


Figure 3.14: Low variation in boundaries due to presence of boundary samples

The need for boundary samples is made clearer using mathematics as in the discussion to follow. As will be shown for the back propagation algorithm the network learns to minimize in the least squares sense and therefore, Equation 3.11 describes the function to be minimized in the case of two inputs and a single neuron:

$$F = \sum_{i=0}^{n-1} [\Phi_i - \Gamma(O_i)]^2$$

$$\therefore F = \sum_{i=0}^{n-1} [T_i]^2$$

Where:

F = function to be minimized

Φ_i = Desired output of neuron for pattern i

O_i = output of neuron for pattern i

$\therefore O_i = \omega_0 \bullet y_i + \omega_1 \bullet x_i + \omega_2$

$\Gamma(\ )$ = non-linear threshold function

T_i = As defined for F

Equation 3.11

The threshold function $\Gamma(\ )$ for this analysis, is as depicted in Figure 3.15. For values of the argument whose absolute value > 1/M the value of $\Gamma(\ )$ is one or negative one dependent on the sign of the argument. For values within the range -1/M to +1/M then $\Gamma(\ )$ takes on the value of its argument (i.e. $\Gamma(x)=x$). The derivative of $\Gamma(\ )$ is M if the argument is in the range -1/M to +1/M and 0 otherwise.

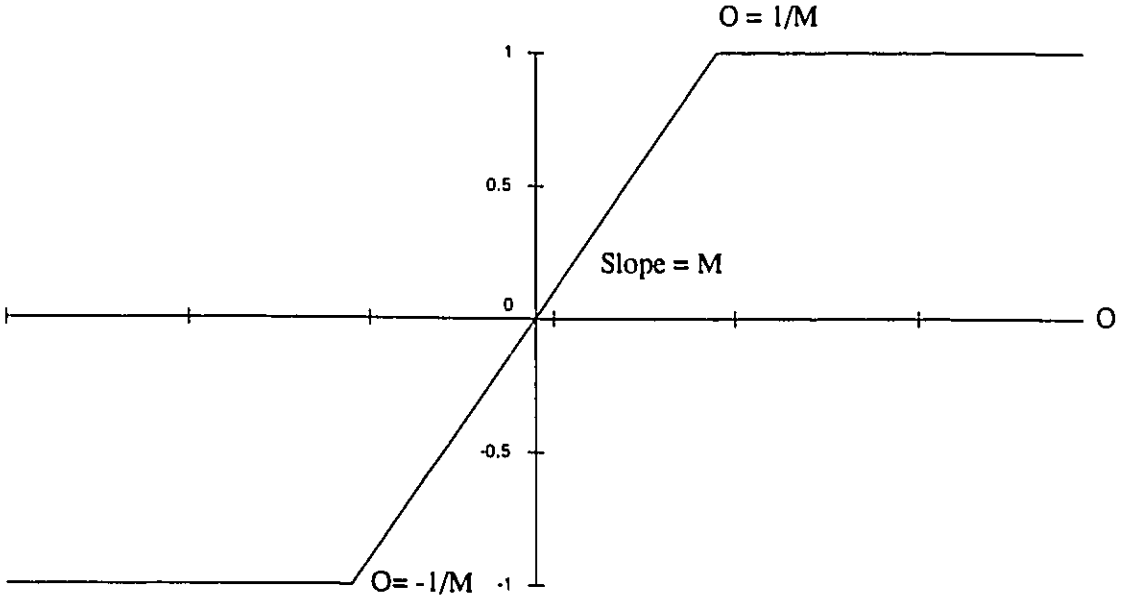


Figure 3.15: Threshold Function for boundary samples discussion

From the well known regression principals, Equations 3.12 are derived (note, the summation is only over those elements of O_i within the range -1/M to +1/M as the multiplication by the derivative forces the remaining terms to zero outside this range):

$$\frac{\partial F}{\partial \omega_0} = -2 \bullet M \sum y_i \bullet T_i = 0$$
$$\frac{\partial F}{\partial \omega_1} = -2 \bullet M \sum x_i \bullet T_i = 0 \quad \text{Equations 3.12}$$
$$\frac{\partial F}{\partial \omega_2} = -2 \bullet M \sum T_i = 0$$

Therefore, given a training set of points in general position and determining $0,\omega\ \omega_1$ and ω_2 from that set (using Equation 3.12) a linear separation is achieved. It is assumed that the line defined by $0,\omega\ \omega_1$ and ω_2 from the minimization of F, does not match the desired class boundary. Let the equation $O_d = \varpi_0 \bullet y + \varpi_1 \bullet x + \varpi_2$ represent the neuron equation for the desired class boundary, then the difference between the required and calculated values is given by Equations 3.13:

$$\omega_0 = \varpi_0 + \varepsilon_0$$
$$\omega_1 = \varpi_1 + \varepsilon_1 \quad \text{Equations 3.13}$$
$$\omega_2 = \varpi_2 + \varepsilon_2$$

The question is, what is the effect of adding a new point to the training set? Let the new values of F, O, $0,\omega\ \omega_1$ and ω_2 be represented by F', O', $\omega_{0'}$, ω_1' and 2',ω then Equation 3.14 is the new equation to be minimized:

$$F = \sum_{i=0}^{n-1} [\Phi_i - \Gamma((O_i)')]^2 + [\Phi_n - \Gamma((O_n)')]^2$$

$$\therefore F = \sum [\Phi_i - (\Gamma 1)]^2 + \sum [\Phi_i - M \bullet (O_i)']^2 + [\Phi_n - M \bullet (O_n)']^2$$

Where: The new point is chosen inside the $-1/M$ to $+1/M$ range or clearly no change in F to F' takes place.

define:

$\omega'_0 \quad \Delta\omega_0 + \omega_0$

$\omega'_1 \quad \Delta\omega_1 + \omega_1$

$\omega'_2 \quad \Delta\omega_2 + \omega_2$

$\Delta_i = \Delta\omega_0 \bullet y_i + \Delta\omega_1 \bullet x_i + \Delta\omega_2$

$$\therefore F = \sum [\Phi_i - (\Gamma 1)]^2 + \sum [\Phi_i - M \bullet (O_i) - M \bullet \Delta_i]^2 + [\Phi_n - M \bullet (O_n) - M \bullet \Delta_n]^2$$

$$\therefore F = \sum [\Phi_i - (\Gamma 1)]^2 + \sum [T_i - M \bullet \Delta_i]^2 + [T_n - M \bullet \Delta_n]^2 \qquad \text{Equation 3.14}$$

It is needed to find the minimum value of F' from the two new variables $\Delta\omega_0$, $\Delta\omega_1$ and $\Delta\omega_2$. Therefore, find the partial derivatives of F' and set them to zero. Equations 3.15 are thereby derived.

$$\frac{\partial F'}{\partial \Delta\omega_0} = -2 \bullet M \sum y_i \bullet [T_i - M\Delta_i] - 2 \bullet M \bullet y_n \bullet [T_n - M\Delta_n]$$

$$\frac{\partial F'}{\partial \Delta\omega_1} = -2 \bullet M \sum x_i \bullet [T_i - M\Delta_i] - 2 \bullet M \bullet x_n \bullet [T_n - M\Delta_n] \qquad \text{Equations 3.15}$$

$$\frac{\partial F'}{\partial \Delta\omega_2} = -2 \bullet M \sum [T_i - M\Delta_i] - 2 \bullet M \bullet [T_n - M\Delta_n]$$

Since, it is assumed that the magnitude of the change in coefficients for the additional n is small, the previous points both inside and outside the range -1/M to +1/M remain in their former relative positions. Therefore, the results of Equations 3.12 can be applied to Equations 3.15 and the partial derivatives set to zero to form Equations 3.16:

$$0 = \sum y_i \bullet -M\Delta_i + y_n \bullet [T_n - M\Delta_n]$$

$$0 = \sum x_i \bullet -M\Delta_i + x_n \bullet [T_n - M\Delta_n]$$

$$0 = \sum -M\Delta_i + [T_n - M\Delta_n]$$

$$\therefore 0 = -M\sum y_i \bullet \Delta_i + y_n \bullet T_n$$

$$\therefore 0 = -M\sum x_i \bullet \Delta_i + x_n \bullet T_n$$

$$\therefore 0 = -M\sum \Delta_i + T_n$$

Where, the Δ_n terms have been absorbed into the summation.

Equations 3.16

The total number of terms can be equated to k and the Δ_is expanded to form Equations 3.17:

$$0 = -M\sum [y_i^2 \bullet \Delta\omega_0 + y_i \bullet x_i \bullet \Delta\omega_1 + y_i \bullet \Delta\omega_2] + y_n \bullet T_n$$

$$0 = -M\sum [y_i \bullet x_i \bullet \Delta\omega_0 + x_i^2 \bullet \Delta\omega_1 + x_i \bullet \Delta\omega_2] + x_n \bullet T_n$$

$$0 = -M\sum [y_i \bullet \Delta\omega_0 + x_i \bullet \Delta\omega_1 + \Delta\omega_2] + T_n$$

The summations of y_i and x_i can be replaced by their expected values.

$$\therefore 0 = E(y_i^2)\Delta\omega_0 + E(y_i \bullet x_i)\Delta\omega_1 + E(y_i)\Delta\omega_2 - \frac{y_n \bullet T_n}{k \bullet M}$$

$$\therefore 0 = E(y_i \bullet x_i)\Delta\omega_0 + E(x_i^2)\Delta\omega_1 + E(x_i)\Delta\omega_2 - \frac{x_n \bullet T_n}{k \bullet M}$$

$$\therefore 0 = E(y_i)\Delta\omega_0 + E(x_i)\Delta\omega_1 + \Delta\omega_2 - \frac{T_n}{k \bullet M}$$

Equations 3.17

After isolation of variables (Equation 3.18):

$$\Delta\omega_0 = \frac{T_n}{k \bullet M} \bullet \frac{N_0\{E(y_i^2), E(y_i \bullet x_i), E(y_i), E(x_i^2), E(x_i), x_n, y_n\}}{D_0\{E(y_i^2), E(y_i \bullet x_i), E(y_i), E(x_i^2), E(x_i), x_n, y_n\}}$$

$$\Delta\omega_1 = \frac{T_n}{k \bullet M} \bullet \frac{N_1\{E(y_i^2), E(y_i \bullet x_i), E(y_i), E(x_i^2), E(x_i), x_n, y_n\}}{D_1\{E(y_i^2), E(y_i \bullet x_i), E(y_i), E(x_i^2), E(x_i), x_n, y_n\}}$$

$$\Delta\omega_2 = \frac{T_n}{k \bullet M} \bullet \frac{N_2\{E(y_i^2), E(y_i \bullet x_i), E(y_i), E(x_i^2), E(x_i), x_n, y_n\}}{D_2\{E(y_i^2), E(y_i \bullet x_i), E(y_i), E(x_i^2), E(x_i), x_n, y_n\}}$$

Where:

N_x = numerator of equation for $\Delta\omega_x$

D_x = denominator of equation for $\Delta\omega_x$

Equations 3.18

Since, the intent of the exercise is to find the effect of the change in the boundary deviation from the "ideal" boundary. The change in this error must be expressed relative to the $\Delta\omega$s, as in Equations 3.19:

$$\omega_0 + \Delta\omega_0 = \varpi_0 + \varepsilon'_0$$

$$\omega_1 + \Delta\omega_1 = \varpi_1 + \varepsilon'_1$$

$$\omega_1 + \Delta\omega_1 = \varpi_1 + \varepsilon'_1$$

Where:

ε'_x = New error in ϖ_x

Applying the relationships $\omega_x = \varpi_x + \varepsilon'_x$

$$\therefore \Delta\omega_0 = \varepsilon'_0 - \varepsilon_0$$

$$\therefore \Delta\omega_1 = \varepsilon'_1 - \varepsilon_1$$

$$\therefore \Delta\omega_2 = \varepsilon'_2 - \varepsilon_2$$

Equations 3.19

It is also required that T_n be expanded in terms of the errors, Equation 3.20:

$T_n = \Phi_n - MO_n$

$\therefore T_n = \Phi_n - M(\omega_0 y_n + \omega_1 x_n + \omega_2)$

$\therefore T_n = \Phi_n - M(\varpi_0 y_n + \varpi_1 x_n + \varpi_2 + \varepsilon_0 y_n + \varepsilon_1 x_n + \varepsilon_2)$

Let $\tau_n = \Phi_n - M(\varpi_0 y_n + \varpi_1 x_n + \varpi_2)$ Equation 3.20

$\therefore \tau_n$ represents the deviation of (y_n, x_n) from the "true" class boundary.

$\therefore T_n = \tau_n - M(\varepsilon_0 y_n + \varepsilon_1 x_n + \varepsilon_2)$

Combining the results of Equations 3.18, Equations 3.19 and Equations 3.20 leads to the formulation of Equations 3.21;

$$\varepsilon'_0 = \varepsilon_0 - \frac{\varepsilon_0 y_n N_0}{k \bullet D_0} - \frac{\varepsilon_1 x_n N_0}{k \bullet D_0} - \frac{\varepsilon_2 N_0}{k \bullet D_0} + \frac{\tau_n N_0}{k \bullet D_0 \bullet M}$$

$$\varepsilon'_1 = \varepsilon_1 - \frac{\varepsilon_0 y_n N_1}{k \bullet D_1} - \frac{\varepsilon_1 x_n N_1}{k \bullet D_1} - \frac{\varepsilon_2 N_1}{k \bullet D_1} + \frac{\tau_n N_1}{k \bullet D_1 \bullet M} \qquad \text{Equations 3.21}$$

$$\varepsilon'_2 = \varepsilon_2 - \frac{\varepsilon_0 y_n N_2}{k \bullet D_2} - \frac{\varepsilon_1 x_n N_2}{k \bullet D_2} - \frac{\varepsilon_2 N_2}{k \bullet D_2} + \frac{\tau_n N_2}{k \bullet D_2 \bullet M}$$

The further reduction of these equations can be achieved by expansion of the Nx and Dx terms. It is instructive to do this for the ε'_1 term, the results for the other two terms are presented but not derived. Two problems can be answered by use of these equations:

i) Is the selection of points close to the desired border a good long term strategy?

ii) Given a previous border between classes close to that required. What is the effect of choosing samples points at varying non-zero distances from that boundary.

The first problem is given by setting τ_n arbitrarily close to zero and finding the long term tendency of the error terms, as in Equation 3.22;

$$\varepsilon'_1 = \varepsilon_1 - \frac{\varepsilon_0 y_a N_1}{k \bullet D_1} - \frac{\varepsilon_1 x_a N_1}{k \bullet D_1} - \frac{\varepsilon_2 N_1}{k \bullet D_1}$$

$\therefore$ Evaluate the parts of the equation in stages

$$\therefore \frac{\varepsilon_0 y_a N_1}{k \bullet D_1} = \frac{\varepsilon_0 y_a \{[E(x_1) - x_a][E(y_1^2)E(x_1) - E(y_1x_1)E(y_1)] - [E(x_1)E(y_1) - E(y_1x_1)][E(x_1)y_a - E(y_1)x_a]\}}{k \bullet \{[E(y_1^2)E(x_1) - E(y_1x_1)E(y_1)][E(x_1)E(x_1) - E(x_1^2)] - [E(x_1)E(y_1) - E(y_1x_1)][E(x_1)E(y_1x_1) - E(y_1)E(x_1^2)]\}}$$

$$\therefore \frac{\varepsilon_0 y_a N_1}{k \bullet D_1} = \frac{\varepsilon_0 \{[E(x_1)y_a - x_a y_a][E(y_1^2)E(x_1) - E(y_1x_1)E(y_1)] - [E(x_1)E(y_1) - E(y_1x_1)][E(x_1)y_a^2 - E(y_1)x_a y_a]\}}{k \bullet \{[E(y_1^2)E(x_1) - E(y_1x_1)E(y_1)][E(x_1)E(x_1) - E(x_1^2)] - [E(x_1)E(y_1) - E(y_1x_1)][E(x_1)E(y_1x_1) - E(y_1)E(x_1^2)]\}}$$

Since, it is the long term tendencies of the equations that are at issue, the values of x_a and y_a (as well as their respective combinations) can be replaced by their Expected values.

$$\therefore \frac{\varepsilon_0 y_a N_1}{k \bullet D_1} = \frac{\varepsilon_0 \{[E(x_1)E(y_1) - E(y_1x_1)][E(y_1^2)E(x_1) - E(y_1x_1)E(y_1)] - [E(x_1)E(y_1) - E(y_1x_1)][E(y_1^2)E(x_1) - E(y_1x_1)E(y_1)]\}}{k \bullet \{[E(y_1^2)E(x_1) - E(y_1x_1)E(y_1)][E(x_1)E(x_1) - E(x_1^2)] - [E(x_1)E(y_1) - E(y_1x_1)][E(x_1)E(y_1x_1) - E(y_1)E(x_1^2)]\}}$$

The two parts of the numerator are equal and therefore, subtract to zero:

$$\therefore \frac{\varepsilon_0 y_a N_1}{k \bullet D_1} = 0$$

$$\therefore \frac{\varepsilon_1 x_a N_1}{k \bullet D_1} = \frac{\varepsilon_1 x_a \{[E(x_1) - x_a][E(y_1^2)E(x_1) - E(y_1x_1)E(y_1)] - [E(x_1)E(y_1) - E(y_1x_1)][E(x_1)y_a - E(y_1)x_a]\}}{k \bullet \{[E(y_1^2)E(x_1) - E(y_1x_1)E(y_1)][E(x_1)E(x_1) - E(x_1^2)] - [E(x_1)E(y_1) - E(y_1x_1)][E(x_1)E(y_1x_1) - E(y_1)E(x_1^2)]\}}$$

$$\therefore \frac{\varepsilon_1 x_a N_1}{k \bullet D_1} = \frac{\varepsilon_1 \{[E(x_1)x_a - x_a^2][E(y_1^2)E(x_1) - E(y_1x_1)E(y_1)] - [E(x_1)E(y_1) - E(y_1x_1)][E(x_1)y_a x_a - E(y_1)x_a^2]\}}{k \bullet \{[E(y_1^2)E(x_1) - E(y_1x_1)E(y_1)][E(x_1)E(x_1) - E(x_1^2)] - [E(x_1)E(y_1) - E(y_1x_1)][E(x_1)E(y_1x_1) - E(y_1)E(x_1^2)]\}}$$

$$\therefore \frac{\varepsilon_1 x_a N_1}{k \bullet D_1} = \frac{\varepsilon_1 \{[E(x_1)E(x_1) - E(x_1^2)][E(y_1^2)E(x_1) - E(y_1x_1)E(y_1)] - [E(x_1)E(y_1) - E(y_1x_1)][E(x_1)E(y_1x_1) - E(y_1)E(x_1^2)]\}}{k \bullet \{[E(y_1^2)E(x_1) - E(y_1x_1)E(y_1)][E(x_1)E(x_1) - E(x_1^2)] - [E(x_1)E(y_1) - E(y_1x_1)][E(x_1)E(y_1x_1) - E(y_1)E(x_1^2)]\}}$$

The numerator and denominator are equal and therefore, divide to give one:

$$\therefore \frac{\varepsilon_1 x_a N_1}{k \bullet D_1} = \frac{\varepsilon_1 x_a}{k}$$

$$\therefore \frac{\varepsilon_2 N_1}{k \bullet D_1} = \frac{\varepsilon_2 \{[E(x_1) - x_a][E(y_1^2)E(x_1) - E(y_1x_1)E(y_1)] - [E(x_1)E(y_1) - E(y_1x_1)][E(x_1)y_a - E(y_1)x_a]\}}{k \bullet \{[E(y_1^2)E(x_1) - E(y_1x_1)E(y_1)][E(x_1)E(x_1) - E(x_1^2)] - [E(x_1)E(y_1) - E(y_1x_1)][E(x_1)E(y_1x_1) - E(y_1)E(x_1^2)]\}}$$

$$\therefore \frac{\varepsilon_2 N_1}{k \bullet D_1} = \frac{\varepsilon_1 \{[E(x_1) - E(x_1)][E(y_1^2)E(x_1) - E(y_1x_1)E(y_1)] - [E(x_1)E(y_1) - E(y_1x_1)][E(x_1)E(y_1) - E(y_1)E(x_1)]\}}{k \bullet \{[E(y_1^2)E(x_1) - E(y_1x_1)E(y_1)][E(x_1)E(x_1) - E(x_1^2)] - [E(x_1)E(y_1) - E(y_1x_1)][E(x_1)E(y_1x_1) - E(y_1)E(x_1^2)]\}}$$

Both halves of the numerator have a zero term and therefore, the total is zero:

$$\therefore \frac{\varepsilon_2 N_1}{k \bullet D_1} = 0$$

$$\therefore \varepsilon'_1 = \varepsilon_1 - \frac{\varepsilon_0(0)}{k} - \frac{\varepsilon_1(1)}{k} - \frac{\varepsilon_2(0)}{k}$$

$$\therefore \varepsilon'_1 = \varepsilon_1\left(1 - \frac{1}{k}\right)$$

Equation 3.22

Similar, techniques can be applied to the other terms and therefore, Equations 3.23 are derived:

$$\varepsilon'_0 = \varepsilon_0(1-\frac{1}{k})$$

$$\varepsilon'_1 = \varepsilon_1(1-\frac{1}{k}) \quad \text{Equation 3.23}$$

$$\varepsilon'_2 = \varepsilon_2(1-\frac{1}{k})$$

The limit of the errors as the number of added boundary points approaches infinity is of interest as in Equations 3.24:

$$\lim_{m\to\infty} \varepsilon'_0 = \varepsilon_0 \prod_{k=m}^{m}(1-\frac{1}{k}) = \varepsilon_0 \frac{\prod_{k=2}^{m}(1-\frac{1}{k})}{\prod_{k=2}^{n-1}(1-\frac{1}{k})} = \varepsilon_0 \frac{\frac{1}{m}}{\frac{1}{n}} = 0$$

$$\lim_{m\to\infty} \varepsilon'_1 = \varepsilon_1(0) = 0 \qquad \text{Equation 3.24}$$

$$\lim_{m\to\infty} \varepsilon'_2 = \varepsilon_2(0) = 0$$

In conclusion despite the initial deviation from the correct class boundary (with the stipulation that the correct boundary lies within the range -1/M to +1/M for the given values of ω_0, ω_1 and ω_2) choosing boundary points ($\tau_n = 0$) will ultimately result in the establishment of the correct boundary. In addition adding "good" examples (Those already classified correctly) to the training set does not change the boundary and therefore, serves no purpose. The second related, problem was to show that if the boundary error is already small then choosing τ_n not equal to zero is counterproductive. Therefore, Equations 3.25 are rewritten with the assumption that the boundary errors are zero:

$$\varepsilon'_0 = \frac{\tau_n N_0}{k \bullet D_0 \bullet M}$$

$$\varepsilon'_1 = \frac{\tau_n N_1}{k \bullet D_1 \bullet M} \qquad \text{Equations 3.25}$$

$$\varepsilon'_2 = \frac{\tau_n N_2}{k \bullet D_2 \bullet M}$$

From these equations evidently the errors are proportional to the value of τ_n. Therefore, any value of τ_n other than zero can only cause the error to deviate from zero.

Besides the previously outlined difficulties there exists another common problem with training set selection, that of unidentified correlation. The most famous example of this is the "tank" problem. It is purported that an early neural network was trained to recognize the presence or absence of tanks (military vehicles) in various stages of obstruction by vegetation. The network training set consisted of photographs of tanks and vegetation and vegetation without tanks. The network converged successfully and was tested with different photographs. It was extremely adept at detecting the presence of the vehicles even when they were almost totally obscured. Unfortunately, when tested in an unstructured environment the network was unable to detect the presence of tanks within its field of view. Upon reexamination of the training and test sets it was found that all the pictures with tanks had been taken on a sunny day, while those without were taken in overcast conditions. The network had discovered this inadvertent correlation. This type of phenomena is possible in practice if care is not taken in choosing the training set (e.g., a network trained to detect circles and squares may rely on the fact that in the test set the circles always covered larger areas etc.). Figure 3.15 depicts the case of a two input network that has classified its inputs as linearly separable from the fact that the chosen training set did not contain examples by which this simple scheme was untenable.

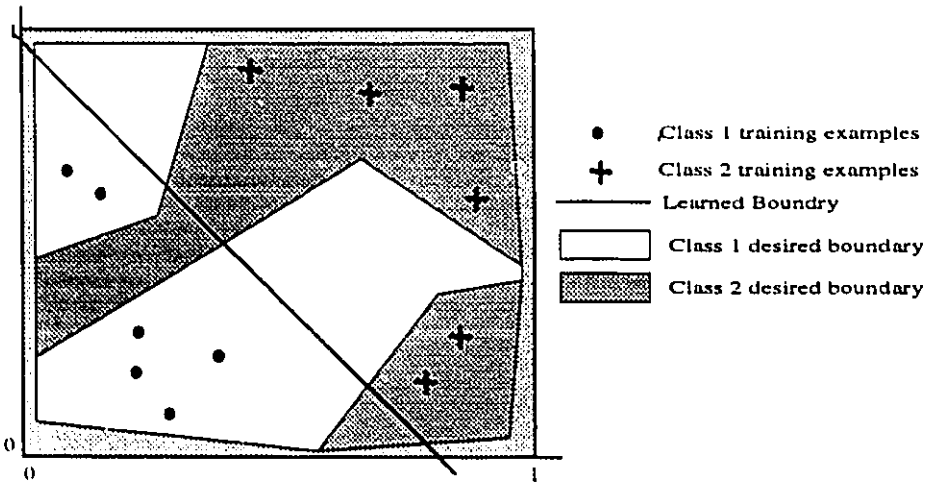


Figure 3.15: Example of a network that has discovered inadvertent correlation's

The following section describes how inadequacies in the training set may be minimized through an iterative development procedure.

3.4.5 Updating the training set

As demonstrated by the previous section it is desirable to have boundary samples for training the network. However, the location of the boundary is unknown and therefore, the first set of training samples can in effect be considered chosen in general position within the clusters. In addition it is unlikely that all clusters will be present in the first iteration of the training cycle (given that points are chosen at random). The first design of the network is therefore, likely to involve too few neurons and define boundaries in the wrong places. Therefore, employing a large training set to the initial iteration of the network may prove counterproductive. The same clusters are likely to be apparent and the tendency may be to choose "good" examples and not boundary samples. To alleviate these problems an iterative regime is proposed. A training set of minimal size is chosen along with an appropriate network topology (using the rules as described in previous sections). The next step is to test the network to decide if it has adequate generalization properties. When the network evaluates each pattern in the test set, two results are apparent. Either the network reports the correct classification or it does not. If the correct classification is reported then the network has defined a boundary for that pattern correctly and the example is discarded. If the classification is incorrect then either the boundary is in the wrong position or the sample may represent a boundary sample. In either case the sample is appended to the original training set. This procedure is not guaranteed to keep samples that may prove useful in future iterations (a discarded sample may have been classified correctly essentially at random and not because, the training set clearly defined the hyper-volume in which it existed). However, all the samples that are kept will prove useful as they represent space that should be designated to a particular class that currently is not. From this argument it is shown that each iteration of the training set will cause an improvement in the ability of the network to correctly generalize. As is evident from section 3.4.3, the size of the network will vary as more training examples are added. However, it is not required to add neurons at every iteration. A better method is to check the convergence of the network at each iteration (this will be apparent during the learning phase). Should the network not converge then the topology can be reevaluated and adjusted. Following this regime will lead to a network capable of classifying correctly to any desired degree of accuracy. However, as the training sets become more complex the time until convergence becomes exponentially longer and therefore, there are diminishing returns to the tradeoff between accuracy and development time. The only practical confirmation of a neural network's abilities comes from extensive statistical testing and its application to practical examples as shown in chapter 4.

3.4.6 Backpropagation as a learning technique

The Backpropagation algorithm is by far the most popular method for training feedforward networks. Its introduction has played a large part in the re emergence of the neural network paradigm in recent years. Most of the problems proposed by Minsky and

Papert have been solved by the application of multi-layer neural nets and in particular those that learn using Backpropagation. Most of Rosenblatts original difficulties were in the teaching of multi-layer networks of Perceptrons. In essence the ideas inherent in Backpropagation are simple and straightforward, although the mathematical formulations are nonlinear and therefore, involved. The Backpropagation algorithm is essentially a gradient descent technique that minimizes a cost function based on a least squares approach. Its primary benefit has been in its applicability to networks of any number of layers and of any complexity. Equation 3.26 represents the function to be minimized by the Backpropagation technique:

$$\Lambda_k = \sum_{i=0}^{n-1} (Y_{ik} - O_{ik})^2$$

Equation 3.26

Where:

Λ_k = Value to minimize

Y_{ik} = Desired output of neuron in output layer for pattern i

O_{ik} = Actual output of neuron for pattern i

k = Integer representing particular ouput neuron

Equation 3.25 must be minimized for every neuron k. The values of Y_{ik} are fixed by the problem and therefore, O_{ik} must be modified to achieve the minimum value of Λ_k. It is well documented that one method of minimizing this function is to modify O_{ik} in a direction opposite to the gradient of Λ_k. Figure 3.16 presents an example for the two dimensional case. The "error" function F(x,y) must be minimized therefore, given an initial point (x_0,y_0) the gradient of F(x,y) is found (this is the maximum rate of change of F(x,y) at a point for a vector of unit length). The optimum path for minimizing F(x,y) is in a direction opposite to the gradient. Therefore, the values of x_0 and y_0 are modified in this direction by small increments, the gradient found for the new value of F(x,y) and the procedure repeated, until all directions lead to increasing values of F(x,y). This process will find a local minima of F(x,y) at least.

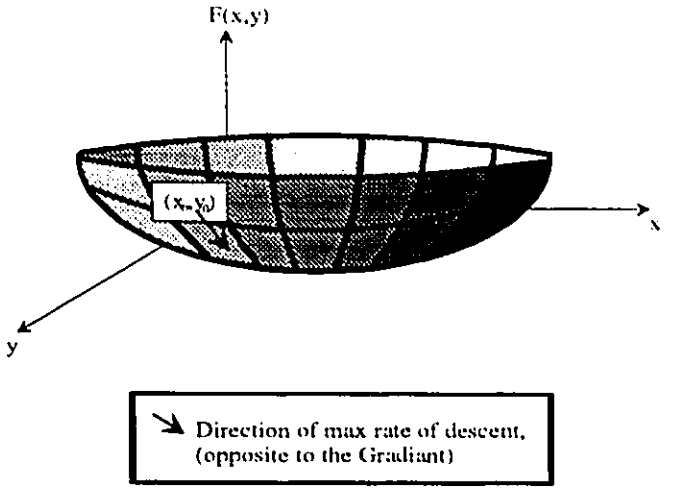


Figure 3.16: Gradient descent for the two dimensional case

This operation is applied to Λ_k and the result is presented in Equation 3.27:

$$\nabla\Lambda_k = \sum_{i=0}^{n-1} \frac{\partial(Y_{ik} - O_{ik})^2}{\partial O_{ik}} \quad \text{Equation 3.27}$$

The outputs O_{ik} are nonlinear functions of the sum of the products of the weights and the neuron outputs from the previous layer therefore, the equation can be written as Equation 3.28:

$$\text{Let } C_{ik} = \sum_{j=0}^{m} \omega_{kj} \bullet M_{ij}$$

In this equation the threshold θ_i has been replaced
by $\omega_{km} \bullet M_{mi}$ (where $M_{im} = 1$) and ω_{km} is adjusted in
the same way as all other weights
$\therefore O_k = \Gamma(C_k)$ Where Γ is a non - linear function
Since, O_k is not directly modifiable the gradiant must be
taken WRT C_k

$$\therefore \nabla\Lambda_k = \sum_{i=0}^{n-1} -2(Y_{ik} - O_{ik}) \bullet \frac{d\Gamma(C_{ik})}{dC_{ik}} \quad \text{Equation 3.28}$$

Since, it is desirable to modify the weight vector the gradient should be taken with respect to the specific weight to be modified. This is equivalent to multiplying Equation 3.28 by

the partial derivative of C_{ik} with respect to a given weight ω_{kj}. Equation 3.29 therefore, represents the gradient of Λ_k with respect to the weight vectors.

$$\nabla\Lambda_k = \sum_{j=0}^{m}\sum_{i=0}^{n-1} -2(Y_{ik} - O_{ik}) \bullet \frac{d\Gamma(C_{ik})}{dC_{ik}} \bullet \frac{\partial\ C_{ik}}{\partial\ \omega_{jk}}$$

$$\therefore \nabla\Lambda_k = \sum_{j=0}^{m}\sum_{i=0}^{n-1} -2(Y_{ik} - O_{ik}) \bullet \frac{d\Gamma(C_{ik})}{dC_{ik}} \bullet M_{ik} \qquad \text{Equation 3.29}$$

Gradient descent requires that each variable ω_{kj} be modified in proportion to its partial derivative in a direction opposite to that derivative (i.e. the maximum rate of descent). Therefore, Equation 3.30 describes how each weight is modified:

$$\Delta\omega_{jk} \propto \sum_{i=0}^{n-1} (Y_{ik} - O_{ik}) \bullet \frac{d\Gamma(C_{ik})}{dC_{ik}} \bullet M_{ik}$$

$$\text{Let } \Delta\omega_{jk} = \sum_{i=0}^{n-1} \Delta\omega_{ijk} \qquad \text{Equation 3.30}$$

$$\therefore \Delta\omega_{ijk} = \eta \bullet (Y_{ik} - O_{ik}) \bullet \frac{d\Gamma(C_{ik})}{dC_{ik}} \bullet M_{ik}$$

The proportionality constant η is often called the learning rate as it determines how far along the gradient path the weights are modified. The term $(Y_{ik}-O_{ik})$ is called the "error" term of the network. Therefore, for each pattern i, the weights are modified in proportion to the products of the learning rate, the error in the output, the derivative of the non-linearity and the output of the previous layer (for a given pattern i). The form presented by Equation 3.30 is suitable for an iterative process by which, a pattern is presented to the network M_{ik}, C_{ik} and O_{ik} are calculated and then the weight vector is modified. After the network has been presented with all the patterns i, a test is made to check whether the outputs O_{ik} are sufficiently close to the desired outputs Y_{ik} for all patterns. Under the condition that the small weight changes made during presentation of the pattern set do not affect the responses of C_{ik} and O_{ik} to M_{ik} to any large extent (a low learning rate), the weight changes will sum to reflect Equation 3.30. This analysis has shown that the weights between the second hidden layer and the output neurons can be modified so that the sum of squares of the error terms $(Y_{ik}-O_{ik})$ is minimized. However, the question remains of how to modify the weights of the hidden neurons? In essence it is to this problem that Backpropagation provides an elegant solution. It is desirable to apply Equation 3.30 to the hidden layers of the network in addition to the output layer. To do this requires that an error term be specified for the hidden neurons. To this end, Backpropagation defines a principal of "Backpropagation" of the errors to the hidden nodes (hence the name). It is

assumed that hidden neurons contribute to the errors at the next layer in proportion to the weights that join the hidden neuron to the next layer. Therefore, the sum of the errors in the output layer times the weight joining the hidden neuron to the output neuron represents the error. Equation 3.31 defines the error for the second hidden layer:

$$\varepsilon_j = \sum_k (Y_{ik} - O_{ik}) \bullet \omega_{kj}$$

Where: Equation 3.31

ε_j = The error defined for neuron j

The error is frequently modified to include the derivative of $d\Gamma(C_k) / d(C_k)$ for reasons derived in the next section. Therefore, the result is Equation 3.32.

$$\varepsilon_j = \sum_k (Y_{ik} - O_{ik}) \bullet \omega_{kj} \bullet \frac{d\Gamma(C_{ik})}{dC_{ik}} \qquad \text{Equation 3.32}$$

To show that this result is consistent with the gradient descent technique, the following arguments suffice: Consider the effect of a given weight connecting the first hidden layer to the second hidden layer. The minimization function,Λ must include all the output neurons (all the neurons are affected by a neuron in this layer). Therefore, the function to be minimized is given by Equation 3.33.

$$\Lambda = \sum_k \Lambda_k$$
$$\therefore \Lambda = \sum_k \sum_i (Y_{ik} - O_{ik})^2 \qquad \text{Equation 3.33}$$

Figure 3.17 defines presents a graphical representation of how the variables used in this analysis apply to the topology of the neural network.

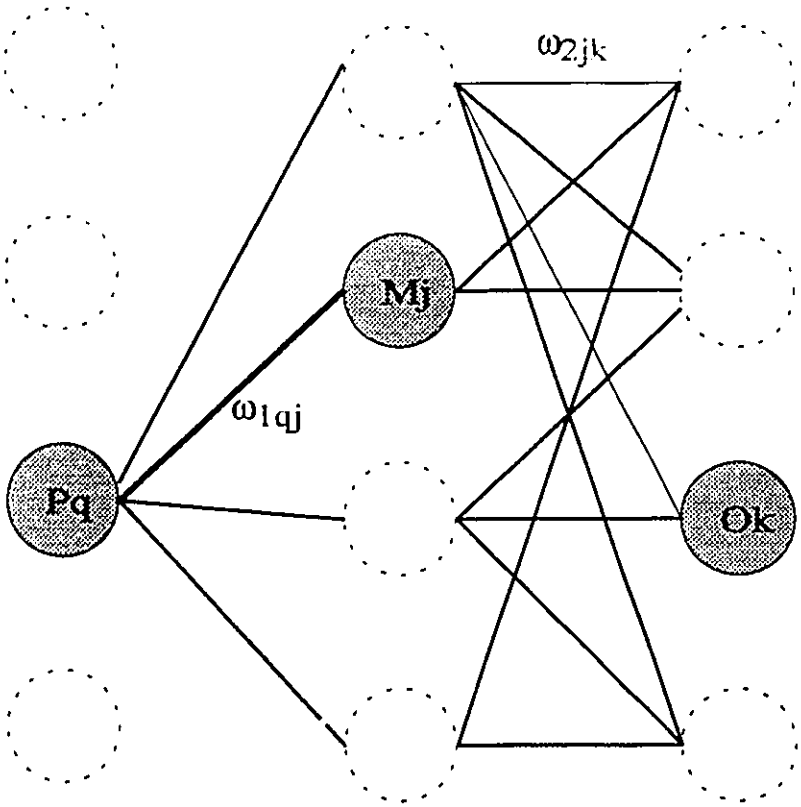


Figure 3.17: Topology for Backpropagation notation

From Equation 3.29 the gradient vector is taken WRT the weights 1qj,ωthe result is Equation 3.34:

$$\nabla\Lambda = \sum_{q}\sum_{j}\frac{\partial\ \Lambda}{\partial\ \omega_{1qj}} \qquad \text{Equation 3.34}$$

Since, the rates of change of the weights are proportional to the partial derivatives of Λ it is only required that a single instance of ω_{1qj} be considered, let this instance be represented by the weight $\omega_{1\alpha\beta}$ therefore isolating the partial derivative yields Equation 3.35:

$$\frac{\partial\ \Lambda}{\partial\ \omega_{1\alpha\beta}} = \frac{\partial\ (\sum_{k}\sum_{i}(Y_{ki} - O_{ki})^2)}{\partial\ \omega_{1\alpha\beta}} \qquad \text{Equation 3.35}$$

Following the principles defined in the previous discussion of weight changes for the outputs, the weight $\omega_{1\alpha\beta}$, can be modified after the presentation of each pattern by an

amount proportional to Λ's derivative with respect to it. Equation 3.36 presents the result of this manipulation.

$$\Delta\omega_{1i\alpha\beta} \propto \frac{\partial \sum_{k} (Y_{ki} - O_{ki})^2}{\partial\ \omega_{1\alpha\beta}} \qquad \text{Equation 3.36}$$

Let $C_{ki}=\Gamma^{-1}(O_{ki})$ as in the previous discussion and therefore, Equation 3.36 is rewritten as Equation 3.37:

$$\Delta\omega_{1i\alpha\beta} \propto \sum_{k} (Y_{ki} - O_{ki}) \bullet \frac{d\Gamma(C_{ki})}{d(C_{ki})} \bullet \frac{\partial\ (C_{ki})}{\partial\ \omega_{1i\alpha\beta}} \qquad \text{Equation 3.37}$$

It is necessary to find the relationship between C_{ki} and $\omega_{1i\alpha\beta}$ to do this the derivation of Equation 3.38 is presented:

$C_{ki} = \sum_{j} \omega_{2kj} \bullet M_{ji}$ (As in Equation 3.28)

Define: $F_{ji} = \Gamma^{-1}(M_{ji})$

$\therefore F_{ji} = \sum_{q} \omega_{1jq} \bullet P_q$

$\therefore C_{ki} = \sum_{j} \omega_{2kj} \bullet \Gamma(F_{ji})$

$= \sum_{j} \omega_{2kj} \bullet \Gamma(\sum_{q} \omega_{1jq} \bullet P_q)$ Equation 3.38

Taking the partial derivative of C_{ki} WRT to $\omega_{1\alpha\beta}$ now yields Equation 3.39:

$$\frac{\partial C_{ki}}{\partial \omega_{1\alpha\beta}} = \sum_{j} \omega_{2kj} \bullet \frac{d\Gamma(F_{ji})}{d(F_{ji})} \bullet \frac{\partial \left(\sum_{q} \omega_{1jq} \bullet P_{qi}\right)}{\partial \omega_{1\alpha\beta}}$$

The value of $\frac{\partial \left(\sum_{q} \omega_{1jq} \bullet P_{qi}\right)}{\partial \omega_{1\alpha\beta}}$ can be determined as follows:

$$\frac{\partial \left(\sum_{q} \omega_{1jq} \bullet P_{qi}\right)}{\partial \omega_{1\alpha\beta}} = 0 \text{ if } j \neq \alpha$$

$$= P_{\beta i} \text{ else}$$

$$\therefore \frac{\partial C_{ki}}{\partial \omega_{1\alpha\beta}} = \omega_{2k\alpha} \bullet \frac{d\Gamma(F_{\alpha i})}{d(F_{\alpha i})} \bullet P_{\beta i} \qquad \text{Equation 3.39}$$

From this result Equation 3.37 can be rewritten as Equation 3.40:

$$\Delta\omega_{1\alpha\beta} \propto \left[\sum_{k} (Y_{ki} - O_{ki}) \bullet \frac{d\Gamma(C_{ki})}{d(C_{ki})} \bullet \omega_{2k\alpha}\right] \bullet \frac{d\Gamma(F_{\alpha i})}{d(F_{\alpha i})} \bullet P_{\beta i} \qquad \text{Equation 3.40}$$

The quantity $\sum_{k} (Y_{ki} - O_{ki}) \bullet \frac{d\Gamma(C_{ki})}{d(C_{ki})} \bullet \omega_{2k\alpha}$ has previously been identified by Equation 3.32 as ε_{α} therefore, with the addition of the learning rate the desired result is given by Equation 3.41:

$$\Delta\omega_{1\alpha\beta} = \eta \bullet \varepsilon_{\alpha} \bullet \frac{d\Gamma(F_{\alpha i})}{d(F_{\alpha i})} \bullet P_{\beta i} \qquad \text{Equation 3.41}$$

In conclusion, it has been shown that the Backpropagation of errors according to their relative weights is consistent with a general learning algorithm based on a gradient descent technique. The errors of the second hidden layer can in sequence be propagated back to the first hidden layer per Equation 3.42:

Define $\omega_{0i\beta\chi}$ as a weight in the first hidden layer, G as the output of a neuron in this layer prior to thresholding.

$\therefore G_{\beta i} = \Gamma^{-1}(P_{\beta i})$

Define $I_{\chi i}$ as the input to the network at neuron χ for pattern i.

$$\therefore \Delta\omega_{0i\beta\chi} = \eta \bullet \varepsilon_{\beta} \bullet \frac{d\Gamma(G_{\beta i})}{d(G_{\beta i})} \bullet I_{\chi i}$$

Equation 3.42

Where:

$$\varepsilon_{\beta} = \sum_{j} \varepsilon_{j} \bullet \omega_{\beta j} \bullet \frac{d\Gamma(F_{ji})}{d(F_{ji})}$$

3.4.7 Importance of the non-linearity

The role of the non-linearity in the neural network is two fold:

i) allows the network to emulate nonlinear functions

ii) allows the network to learn by gradient descent

To properly understand the role of the non-linearity it is necessary to remove it from the governing equations of the network and determine the result. An example is obtained by considering the generalized formulation of a two-layer network. Equation 3.43 describes the situation for the first layer:

$$H_{1ji} = \Gamma\{\sum_{m} \omega_{1jm} \bullet I_{im}\}$$

Where:

H_{1ji} = Output of first layer for neuron j and pattern i — Equation 3.43

Γ = The threshold function

ω_{1jm} = The m^{th} weight for the j^{th} neuron

I_{im} = The m^{th} input for the i^{th} pattern

The output of the first layer is input to the second as in Equation 3.44:

$$H_{2ki} = \Gamma\{\sum_{j} \omega_{2kj} \bullet H_{1ji}\}$$

Where:

H_{1ji} = Output of first layer for neuron j and pattern i Equation 3.44

H_{2ki} = Output of second layer for neuron k and pattern i

Γ = The threshold function

ω_{1kj} = The j^{th} weight for the k^{th} neuron

If $\Gamma(x)$ is a simple linear mapping $\Gamma(x)=ax+b$ then the equation for the second layer can be rewritten as Equation 3.45:

$$H_{2ki} = a \bullet \{\sum_{j} \omega_{2kj} \bullet H_{1ji}\} + b$$

$$\therefore H_{2ki} = a \bullet \{\sum_{j} \omega_{2kj} \bullet [a \bullet (\sum_{m} \omega_{1jm} \bullet I_{mi}) + b]\} + b$$

$$\therefore H_{2ki} = \{\sum_{j} [\sum_{m} a^2 \bullet \omega_{2kj} \bullet \omega_{1jm} \bullet I_{mi}] + a \bullet \omega_{2kj} \bullet b\} + b$$

Let $c = \sum_{j} a \bullet \omega_{2kj} \bullet b + b$ a constant, and interchange

the order of summation

$$\therefore H_{2ki} = \{\sum_{m} I_{mi} \bullet \sum_{j} a^2 \bullet \omega_{2kj} \bullet \omega_{1jm}\} + c$$

$$\text{let } \omega_{km} = \sum_{j} a^2 \bullet \omega_{2kj} \bullet \omega_{1jm}$$

$$\therefore H_{2ki} = \{\sum_{m} I_{mi} \bullet \omega_{km}\} + c$$

Equation 3.45

Equation 3.45 is identical to that of a single layer neural network. In conclusion, the absence of the non-linearity reduces the capabilities of the network from a general mapping of arbitrary disconnected clusters to that of a simple linear discriminant. It is therefore, the non-linearity in the mapping that provides the full capability of the network model.

The Backpropagation paradigm relies on a gradient descent technique and therefore, on both the form of the non-linearity and its first derivative. To clearly understand the behavior of this learning algorithm it is necessary to understand how the shape of the various possible non-linearities influences that behavior. There, are many potential nonlinear functions however, they must all incorporate the following property: The magnitude of the function should be zero for a zero input, for a large negative input the magnitude should equal minus one and for a large positive input the magnitude should

equal plus one. These rules apply to the case of a neuron with bipolar output, the case of a unipolar neuron is simply a scaling function from the bipolar case. Three functions used extensively are displayed in Figure 3.18:

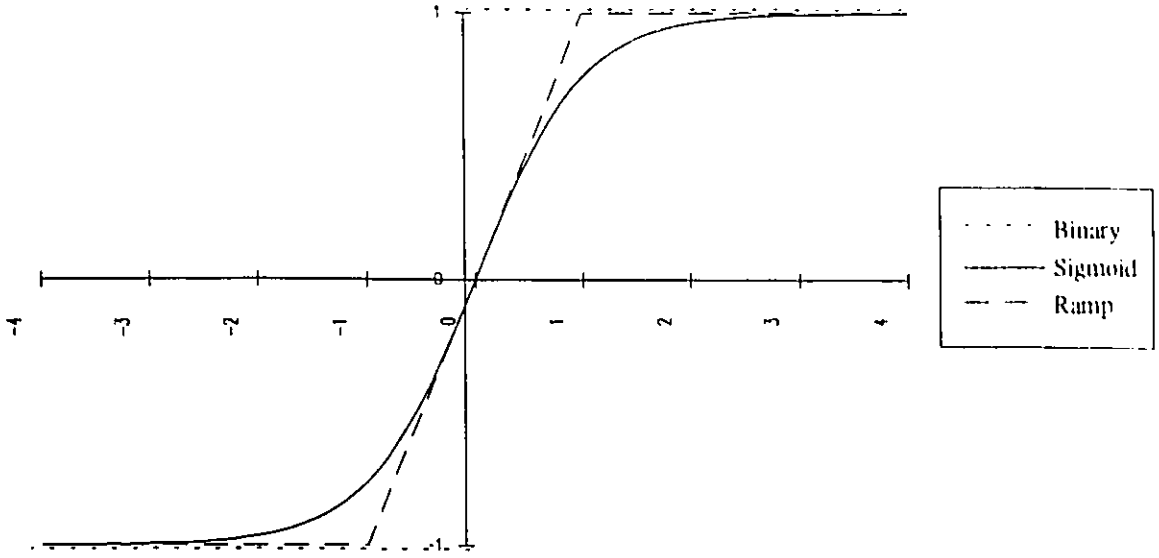


Figure 3.18 - Selection of possible nonlinear functions

Each of the three functions have the desired properties that allow class determination and are described by the Equations 3.46:

$$\Gamma(x) = \frac{\|x\|}{x} \qquad \text{Binary}$$

$$\Gamma(x) = \frac{\|x\|}{x} \quad \text{if } \|x\| \geq \frac{1}{M}$$

$$M \cdot x \quad \text{if } \|x\| < \frac{1}{M} \qquad \text{Ramp} \qquad \text{Equations 3.46}$$

$$\Gamma(x) = \frac{1 - e^{-\alpha x}}{1 + e^{-\alpha x}} \qquad \text{Sigmoid}$$

Each of these functions have the basic properties for neural classification however, there are other requirements for the Backpropagation method of learning. The use of the binary function is precluded by the fact that Backpropagation relies on the derivative of $\Gamma(x)$. Since the derivative is infinite at zero and zero else, the weight changes, as calculated by Equation 3.42 is either zero or infinite. Therefore, its use is prohibited for gradient descent style learning. The ramp function, though it does not have continuous derivatives, can be used for Backpropagation learning. In practice a decision is made for the derivative based

on the current value of x. In the region where the magnitude of x is greater than 1/M the derivative is zero else, the derivative is M. However, this formulation has difficulties that are shown by Figure 3.19. In the figure the error terms for the Ramp and Sigmoid functions are graphed against the value of x. These errors are computed on the assumption that the desired output is one and therefore, the equations are the difference between 1 and $\Gamma(x)$ times the derivative of $\Gamma(x)$ at x. Equation 3.47 expresses this relationship for both the Ramp and Sigmoid functions.

$$\varepsilon = \{1 - \Gamma(x)\} \cdot \frac{d\Gamma(x)}{dx} \quad \text{Equation 3.47}$$

The values for both functions approach zero as x becomes a positive value. Conceptually this means that x is "correctly" classified. When $\Gamma(x)$ is less than one then x is incorrectly classified. It is desirable that as x deviates further from the correct classification that the error increases and therefore, the weight changes are larger for those neurons in greater error. However, the problem in a ramp non-linearity occurs when the value of $x < -1/M$ is reached. At this point the weight changes are set to zero. In other words if x is misclassified at a distance of more than 2/M from the current boundary then it can have no influence on the boundary. Therefore, the sigmoid function has both the advantage of continuous derivatives and the fact that the error does not become zero for highly misclassified values of x. It should be noted that the requirement that large negative values of x when evaluated by $\Gamma(x)$, approach -1 asymptotically conflict with the desire that the error gets larger. The derivative of $\Gamma(x)$ with respect to x must eventually approach zero and therefore, so must the error. In addition if the slope M is too small, the classification boundary is "fuzzy." Here regions exist where the network is neither classifying the current pattern as one type nor another. Here a large value of M provides the cleanest boundary, a property highly desirable in a classification system. The same type of effect can be observed for the sigmoid function as the value of α becomes large in Equation 3.46. The boundary becomes less "fuzzy" however, values of x in error cause less weight correction and learning time is increased. It is possible to mitigate some of this effect by setting the initial weights to small random values. Therefore, the argument of $\Gamma(x)$ is small and the errors cannot be zero. Equation 3.48 expresses this relationship mathematically.

$$x = \sum_{j=0}^{n-1} \omega_j \cdot I_j \quad \text{The neuron equation}$$

$$\therefore \lim_{\omega_j \to 0} = \{1 - \Gamma(x)\} \cdot \frac{d\Gamma(x)}{dx} \quad \text{Equation 3.48}$$

$$= \alpha$$

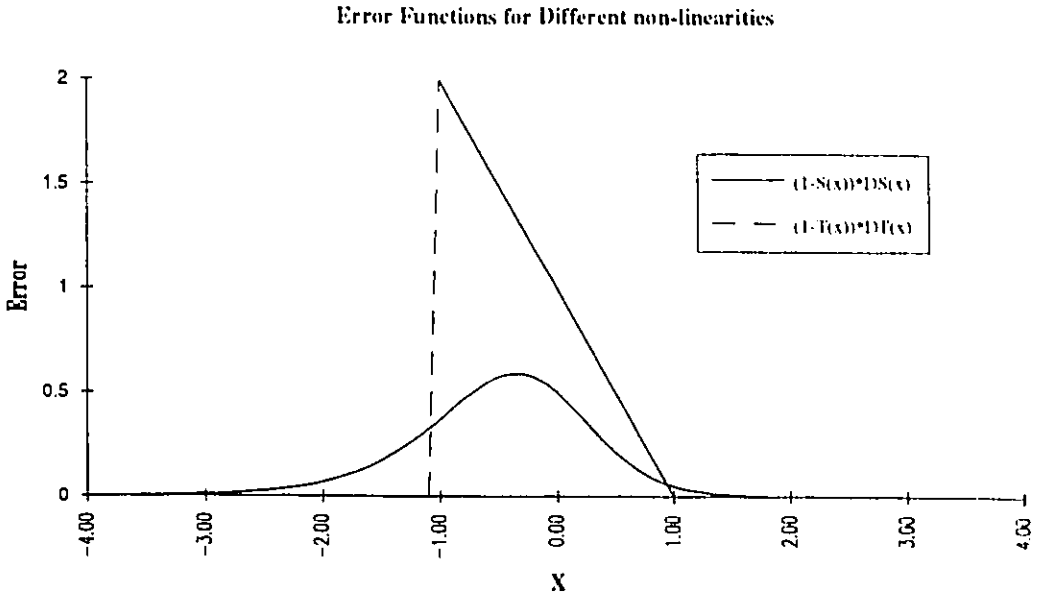


Figure 3.19: Errors for Sigmoid (S) and Ramp (T) Functions

3.4.8 Convergence of the method

As previously indicated, there are no guarantees that the network will converge to a state that produces a pattern recognition engine applicable to a desired problem. In fact several problems may arise during network developments that are associated with the convergence of the Backpropagation algorithm:

i) Is the network guaranteed to converge?

ii) Does the network converge to a global minima?

iii) Is the convergence timely?

The Perceptron convergence theorem states that if a weight matrix does exist that causes the network to reach a minima then a procedure such as Backpropagation will find a solution to the minimization problem. However, this procedure does not guarantee that the solution found is identical to the one that is known to exist. From the results of section 3.3.1, it is shown that a solution to the classification problem exists for some unknown network topology consisting of three layers of neurons. However, since no *a priori* knowledge of the minimum network topology is known the application of the theorem is difficult in practice. In addition, the theorem only states that a solution will be found it does not state that this solution will be the desired one.

The related problem of discovery of local minima is also of concern. In Figure 3.20 the error surface as a function of two weight variables is graphed. As can be seen from the figure, there exists the possibility that the gradient descent technique will find the local rather than the global minima. In practice this phenomenon is rare and was not observed in the development of the networks in chapter 4.

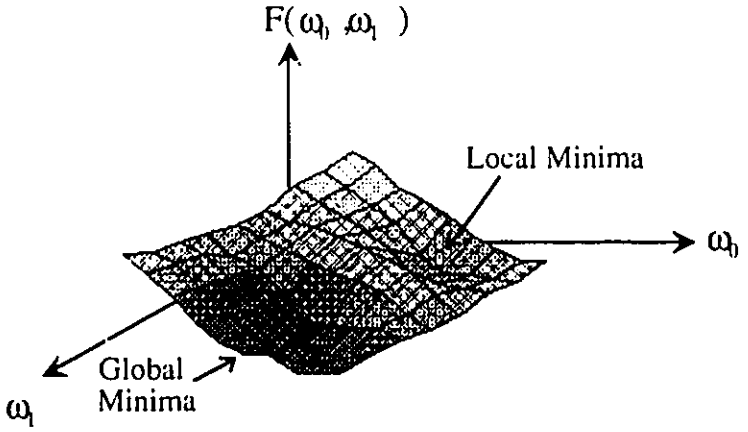


Figure 3.20: Global and Local Minima

Backpropagation requires an extremely long learning cycle when compared to its recall time. This is due to three reasons:

i) Many passes through the training set must be attempted to achieve convergence.

ii) Large numbers of patterns may be required to train a network of practical size, and

iii) The learning rate must be reduced when more patterns are added to the training set.

The first two items are self explanatory. The third is a side effect of Equation 3.30. In this equation it was assumed that the small weight changes throughout a iteration of the learning (an iteration of learning is the presenting of all input patterns to the network once, with coincident modification to the weight values) would be equivalent to finding the state of the network at a given time for all patterns, and modifying the weights globally. However, as the number of patterns in the training set increases the greater the deviations of the weight values from the start of an iteration until its end. Therefore, the assumption that the small weight changes do not affect the state of the network during an iteration no longer applies. To counter this effect, it is necessary to reduce the learning rate in proportion to the number of patterns in the training set, this reduction in learning rate causes an increase in the number of iterations required for convergence to be reached.

Chapter 4
Presentation and Discussion of Experimental Results

4.1 Introduction

This chapter will focus on the experimental results obtained from applying a neural network paradigm to the object recognition of images captured by tactile sensing. Four examples of networks trained to classify tactile images are presented. Each of the four was developed using the Backpropagation learning technique and the iterative training set enhancement process described by chapter 3. The four were chosen to exemplify types of different pattern recognition problems. They therefore, provide an overview of the capabilities of the neural network as a low level pattern classification engine as applied to an FSR tactile sensor. Details of the learning parameters, final weight values, network configuration and performance of each network are presented by this chapter. In addition some of these details are provided for intermediate stages in the development of the networks so that the evolution of the topology and performance can be evaluated against the final configuration. The four networks presented are: *Big Blocks*, a network that uses large wooden blocks with embossed lettering to provide the tactile images; *Small Blocks*, similar to *Big Blocks*, except that smaller blocks are used; *One Two*, a network trained to recognize whether one or two fingers are pressed on the sensor surface and *Lines*, a network designed to find the angle of a straight line (formed by force from any object with a linear edge) relative to an arbitrary axis of the tactile sensor.

4.1.1 System Overview

The tactile sensor converts the force profile from the presented object into a 16 x 16 grid of resistance measurements that are each inversely proportional to the point applied force. The resistance measurements are converted into voltage levels using circuits based on operational amplifiers and analog multiplexors as described in chapter 2. A general purpose host computer is used to input a "grid address" into the sensor electronics. This 8 bit address consists of two 4 bit numbers representing the row and column of the sensor element to be scanned. Outputs of the sensor electronics are voltage levels that are proportional to the applied force (see figure 2.08 for the exact relationship). An Analog to Digital converter interface, resident in the host computer, is used to quantize this voltage also provides an interface the host backplane bus. Each point of the grid is scanned by the host computer in sequence and the results input to the "Sensor" software application. After all 256 points are scanned into the application, a tactile image is created and may be manipulated by the host for: display, storage and further processing. An example of such an image is presented in figure 4.01, figure 4.02 shows the experimental setup:

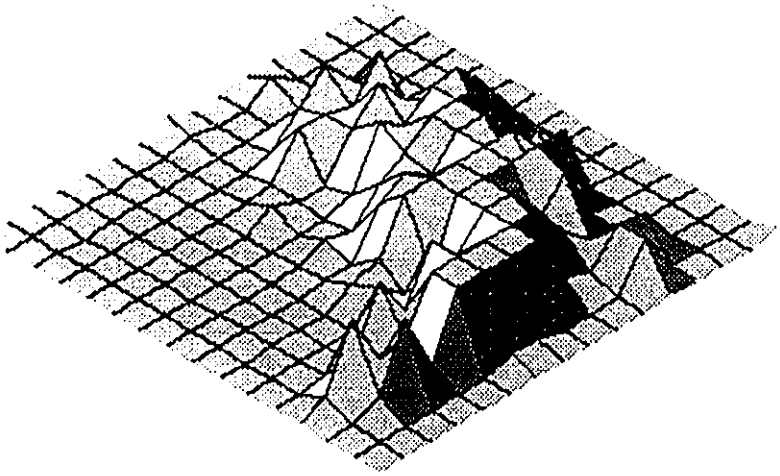


Figure 4.01 - Typical Tactile image

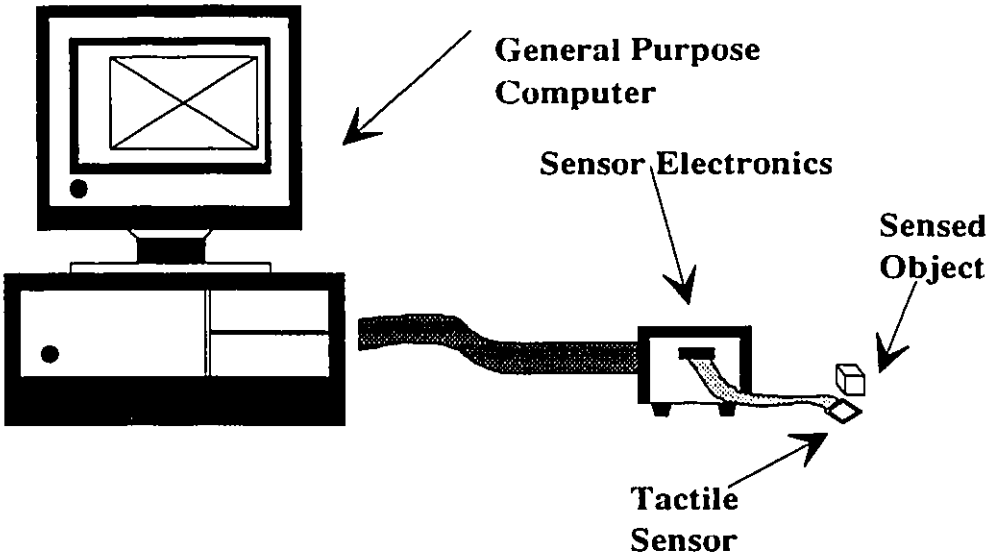


Figure 4.02 - Experimental Setup

The "Sensor" application (see Appendix A for a partial listing of this application) provides a user interface that allows an image capture to be initiated. Once captured, a set of images is maintained in a file and can be used as either a "training" or a "testing" set. As an image is captured its desired classification is input by the trainer to the application for storage and use during the training phase. A training set is employed to "teach" the neural network to recognize images of these predefined classifications. Figure 4.03 depicts the user interface for the "Sensor Input" application.

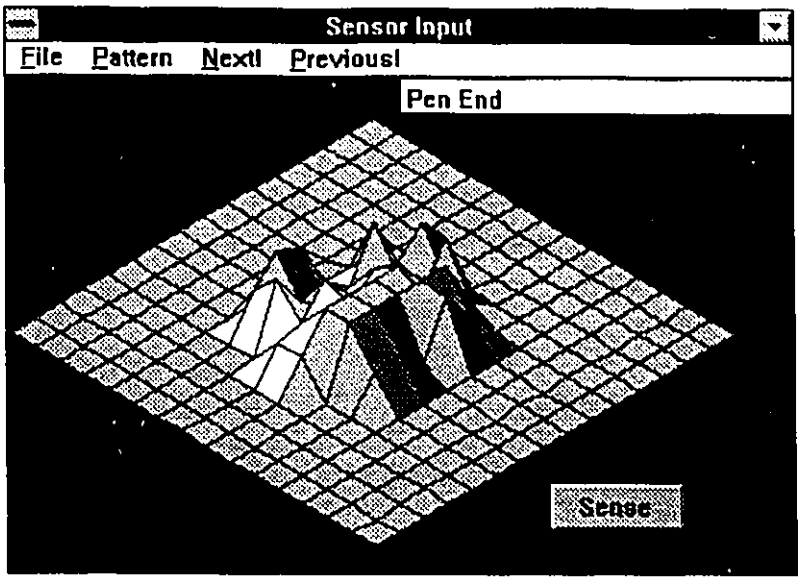


Figure 4.03 - Sensor Input application

A second application "Network Developer" (see Appendix B for a partial listing of this program) is used to design and test neural networks. The user determines the required number of inputs to the network, outputs from the network and the number of, and form of, each hidden layer. Once the network topology is defined the weight values are initialized to random values between -1 and +1. Figure 4.04 depicts such a configuration graphically (note: The weight's connections are shown as lines whose width is proportional to the magnitude of the connection and the shade dependent on the sign, black positive, gray negative). The weight values connecting the inputs and the first layer are not depicted in this graphic to avoid clutter.

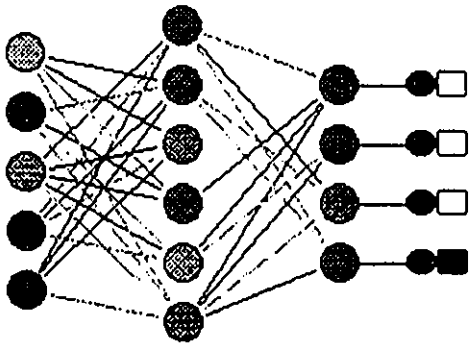


Figure 4.04 - Initialized network

A set of training patterns is input to the application and the network trained using the iterative training regime outlined in chapter 3. Four variable parameters are input by the user to influence learning. The first is the "convergence criteria." This parameter sets the

maximum amount of "error" allowed in any of the output neurons when presented with the entire pattern suite. The second parameter "learning rate," is a linear multiplier of the current error term as described in section 3.4.6. The third parameter "weight momentum," is used to set the extent to which the weight changes are proportional to their rate of change at the last iteration. In practice the inclusion of a momentum term can speed learning in the Backpropagation algorithm. The fourth term is the "Recognition Criteria," this parameter is used to set an amount by which the outputs of neurons must differ such that the network is suggesting an unambiguous classification (this parameter will be described in greater detail in later parts of this chapter).

After the building of the network topology and the input of the training set, the network is set into learn mode. In this mode, the network weights are modified according to the procedures outlined in chapter 3 and learning is completed when the network converges or learning is halted by the user. A status window is provided in the application to display the percentage of images correctly classified and the network's classification of the image currently being displayed. In addition the intended "Desired" classification originally entered by the trainer is displayed. Once learning is complete, the trained networks may be stored in files for future reference and testing. Figure 4.05 depicts the user interface to the Network Developer application.

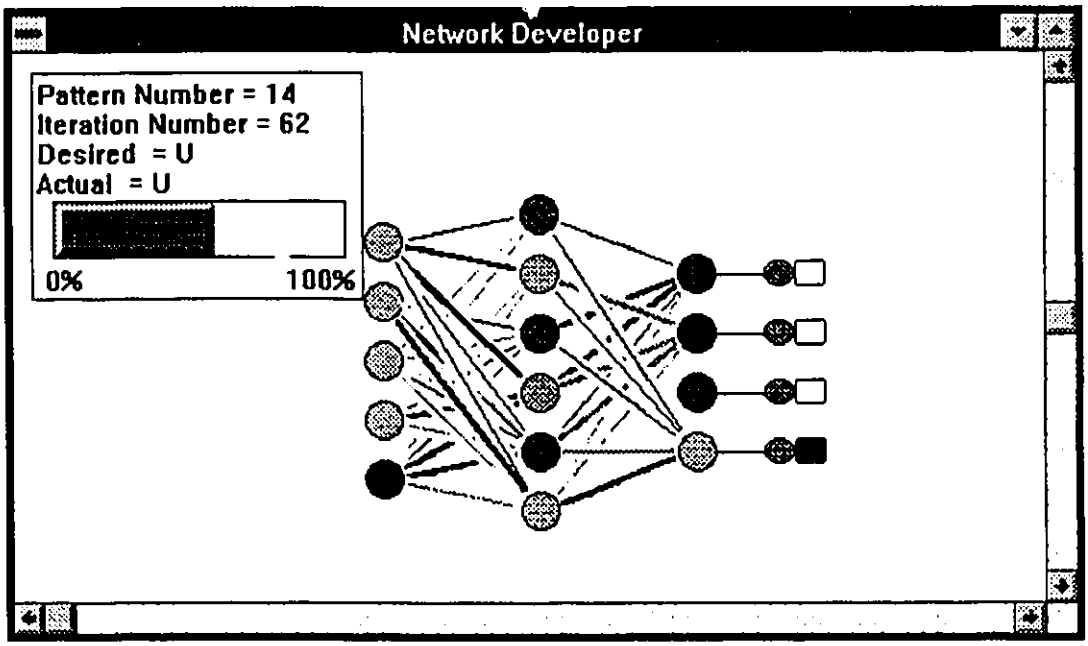


Figure 4.05 - Network Developer user interface

Stored networks can be used in one of two ways: Interactively, by which an image is captured in real time and the network is requested to recognize the image based on its stored classifications or background testing, where a file of images are presented to the network and the network finds the percent of correctly classified images and whether the images fall within one of its classification regions. In the interactive mode, the "Sensor Input" application captures the image and relays it to the "Network Developer"

application. Network Developer responds with its classification of the pattern based on the current network in the system. In the non-interactive mode, a file of test patterns is read into Network Developer and a mode selected where the results of the classification are displayed in the status box. Three results are possible from the classification of any image: An image is correctly classified (i.e. the trainer and the network agree on the classification), an image is incorrectly classified (i.e. the classification proposed by the network is different from that identified by the trainer) and unknown (represented by the ? mark). In the "unknown" categorization, the image does not fall explicitly within the boundaries of any one class as decided by the "Recognition Criteria."

To provide visual feedback, network developer shows the percent of correctly identified patterns identified during the learning process. The current percentage of correctly classified patterns is evident from the display. Figures 4.06 and 4.07 depict the displayed image at various stages of learning.

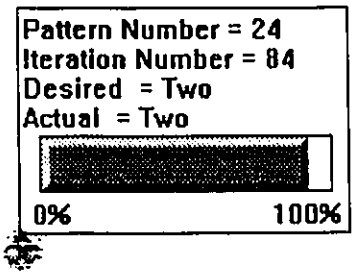


Figure 4.06 - network close to convergence

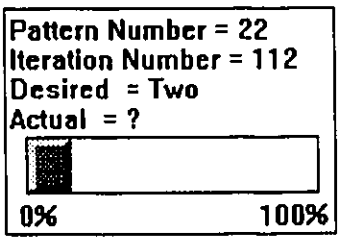


Figure 4.07 - network moved from convergence

To further improve the development process, the network developer application was designed to show a neuron's activity state (output value) and weights to connected neurons, by colour changes. This type of display aids in determining if a neuron is actively participating in the solution to the current problem. If the neuron is unused there exists the possibility of culling the neuron from the network topology. A second use for the visual display is the determination of the internal states of the network that may be present for a given input pattern class. In essence one colour represents the -1 output and the other the +1 output. States between the two are represented by intermediate shades. In monochrome mode for example, the colour white represents a -1 output and the colour black +1. All states between +1 and -1 are represented by shades of gray.

The weight display employs a similar system under which, negative weights are gray and positive black and the width of the connection between neurons suggests the magnitude of the weight. Figure 4.08 shows an example of a simple network that has learned the Exclusive Or problem with the neuron and weight states indicated for the four possible input pattern forms.

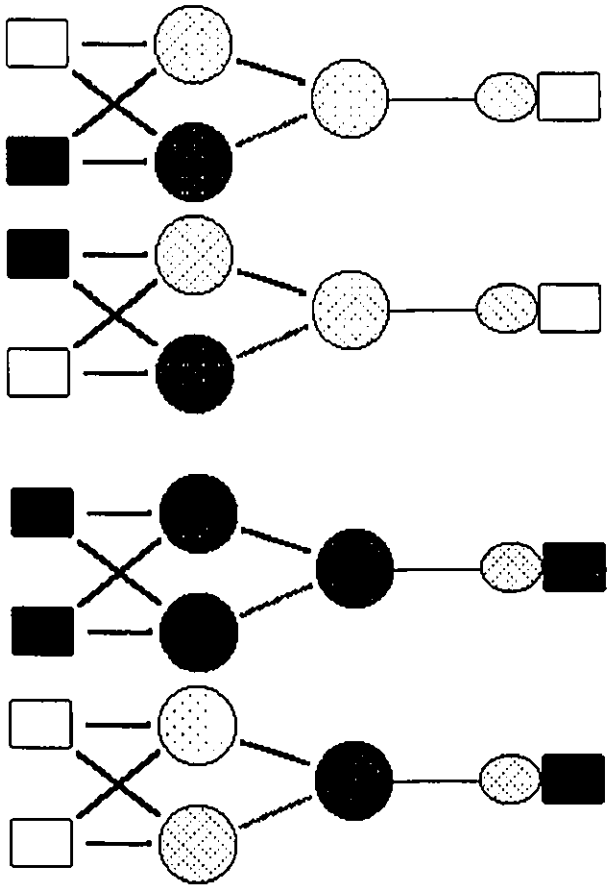


Figure 4.08 - ExOr problem neuron states and weight values

Besides the weights and neuron states, a third visual indicator is provided to aid in network development, the output oval. As can be seen in Figure 4.08, this oval becomes white (in monochrome mode) to show that the output of the neuron and the *a priori* classified output are in the same state. A black oval would indicate that they are in different states.

An initial pattern set is interactively selected using the "Sensor Input" application and used to train an initial network topology in "Network Developer." The "Sensor Input" application is then used with the trained network in the interactive mode, to decide if any pattern's inputs from the tactile sensor are misclassified. Any misclassified images are appended to the training set with the correct classification identified. The network is retrained with the appended patterns and the entire procedure repeated until satisfactory

results are achieved. This implementation follows the theoretical procedure outlined in chapter 3.

4.1.2 Performance Evaluation

In the general case, the performance of the various configurations is evaluated compared with a set of test patterns each of which is collected in the same manner as the training set but do not contain images explicitly in that set. Since, in no case is the applied force or orientation of the applied object controlled these images represent a wide range of variants. These variations produce the irregular images required to simulate real world "noisy" conditions. If the parameters are in tighter control then the pattern recognition problem is simplified. However, it was felt that simplification by this method, although showing better results on paper would not represent a realistic scenario for use in an uncontrolled, unstructured environment. Beyond the performance evaluation against a similar but independent pattern set, the networks are evaluated for performance outside the limits of their original problem domains. For instance, a network trained to recognize letters aligned to one axis of the sensor are rotated and the performance tested outside the limits of the original training set.

4.2 One Two Net

4.2.1 Background

The "One-Two" network was designed to test the neural network's ability to solve the connectedness problem as explored by Minsky and Papert. They can show that a single perceptron cannot solve the problem of whether an image input to the network contains a single "connected" region or more than one "disconnected" region. This result is equivalent to showing that the problem is not linearly separable per the criteria established in chapter 3. The Exclusive Or enigma is an example of such a problem. In this puzzle a network is provided with two inputs and it can be shown that a minimum of two layers is required to reproduce the Exclusive Or function. The One-Two problem is a more complex extension of the Exclusive Or problem. In this problem either one or two finger presses are imprinted onto the tactile sensor, the network is then required to recognize which of these events has occurred. Similarity of the One-Two problem to the exclusive or problem is outlined by Figure 4.09.

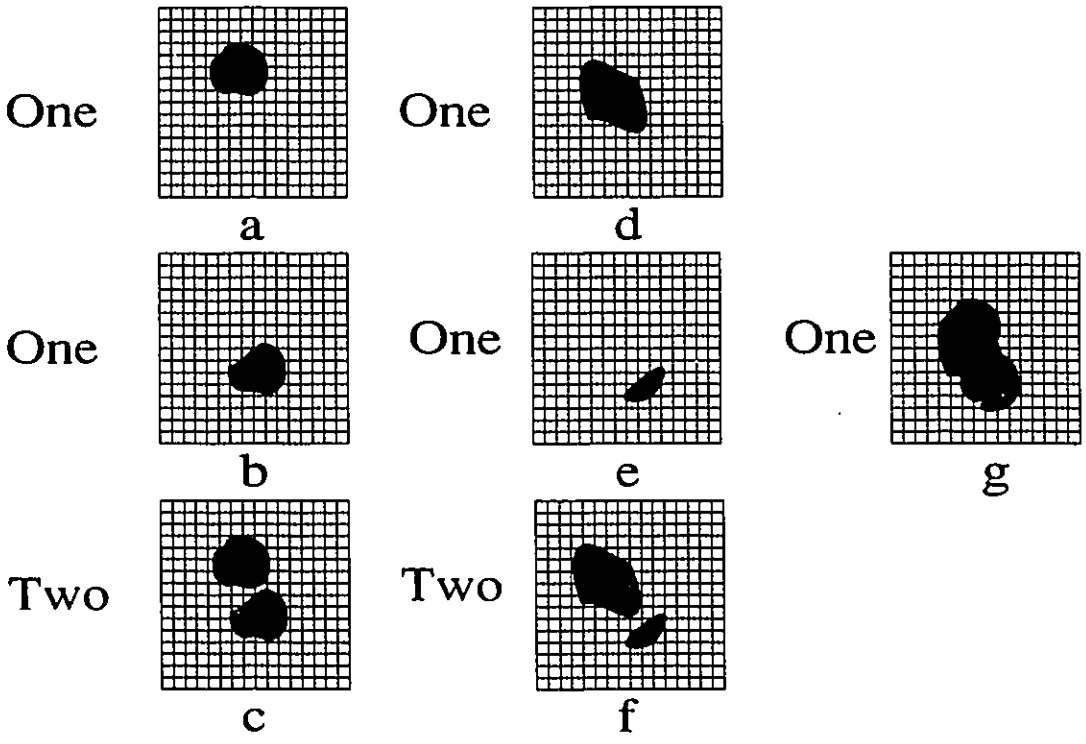


Figure 4.09 - One-Two Net, Exclusive Or similarity

The patterns a,b,d,e suggest the types of tactile images that result from single finger presses, each pattern is labeled "One" to indicate that fact. Patterns c and f are labeled "Two," this indicates that two points of finger pressure are present. Obviously c is a combination of a and b and that f is a combination of d and e. In addition image g is a combination of the four images a,b,d and e. This image is again labeled "One," as it also could be formed from the force of a single pressure point. This image set poses the

connectedness problem posed by Minsky and cannot be solved by a single neuron as shown by the following argument. Consider Equation 3.01, the formula for the output of a neuron, the sum of products of weighted inputs compared to a threshold.

Assume that a weighted sum that falls below the threshold causes a neuron output of -1 that represents a classification of "One." The patterns a,b,d and e do not cause the weighted sum to increase beyond this threshold and therefore, are classified as "One." However, the additions of a and b to equal c produce the weighted sum to increase beyond the threshold and therefore, the classification is +1 or "Two." A similar result holds for the addition of d and e to produce f. Apparently the addition of c and f cannot decrease the weighted sum (as each in isolation is greater than the threshold) and therefore, the result must be classified by the network as "Two." Since, this classification is incorrect the single neuron cannot produce the correct categorization for this problem.

4.2.2 Problem Statement

The problem is to detect whether one or two fingers are applying pressure to the tactile sensor. This determination should be independent of the total force applied by either one or both fingers and be independent of the position of the finger presses on the sensor surface.

Figures 4,10 through 4.12 depict examples of valid single finger presses whereas, Figures 4.13 through 4.15 depict examples of two finger presses.

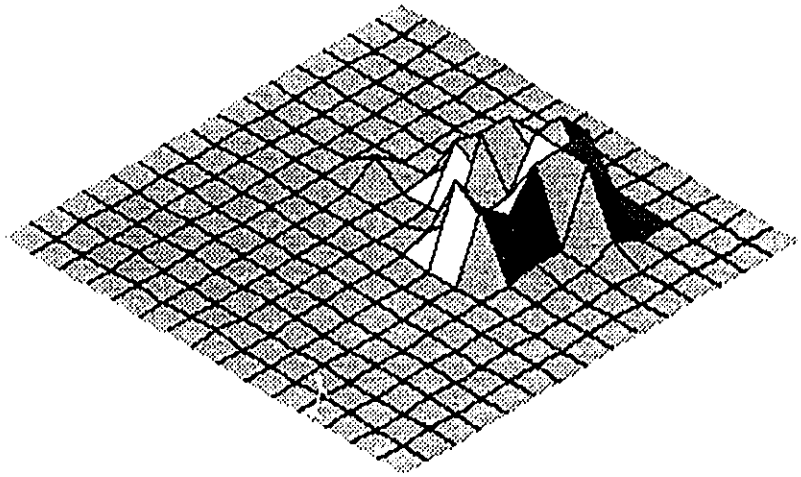


Figure 4.10 - "One" example 1

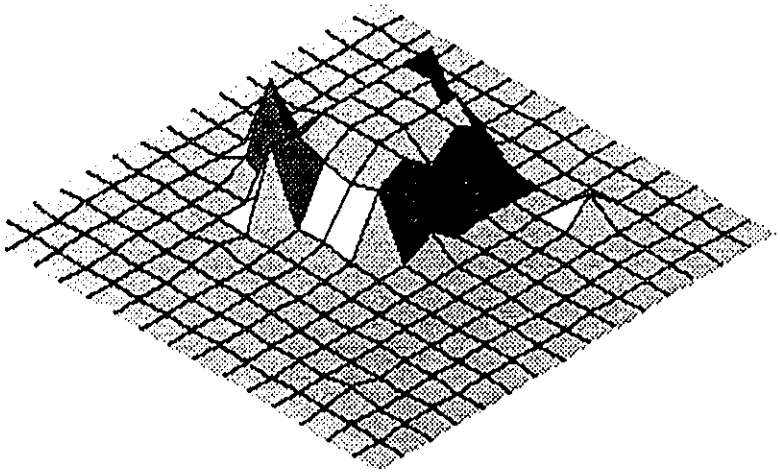


Figure 4.11 - "One" example 2

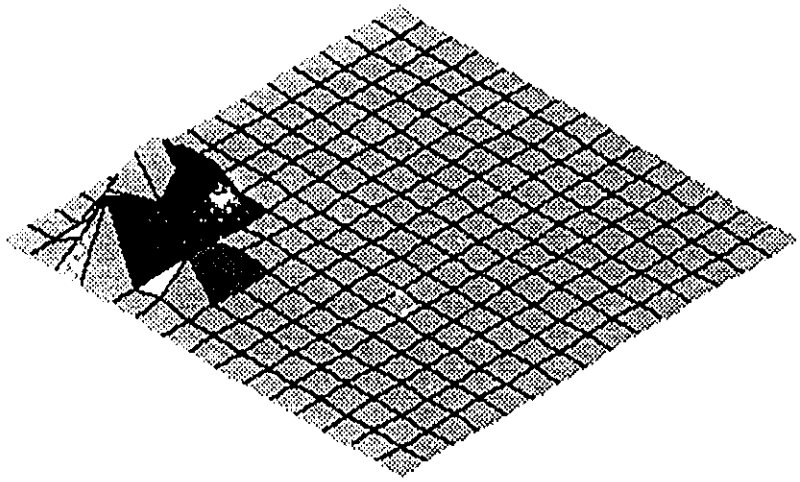


Figure 4.12 - "One" example 3

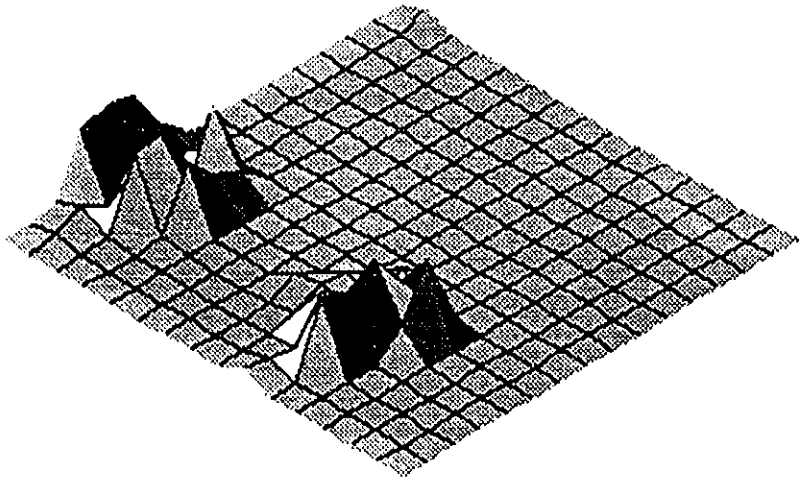


Figure 4.13 - "Two" example 1

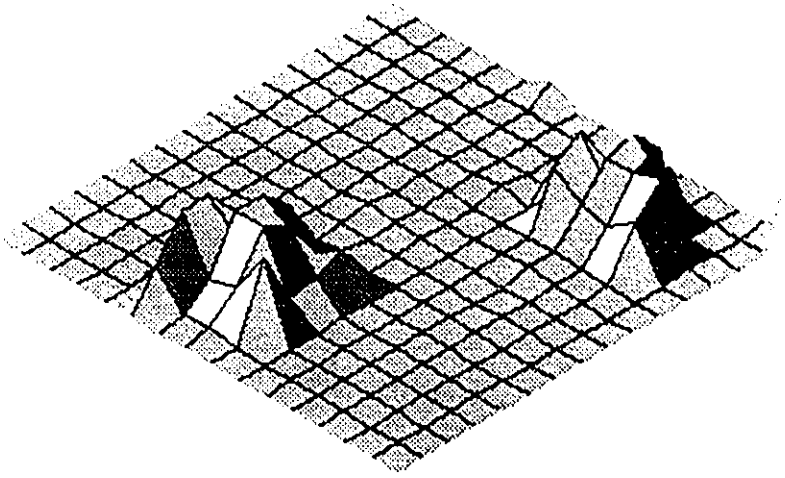


Figure 4.14 - "Two" example 2

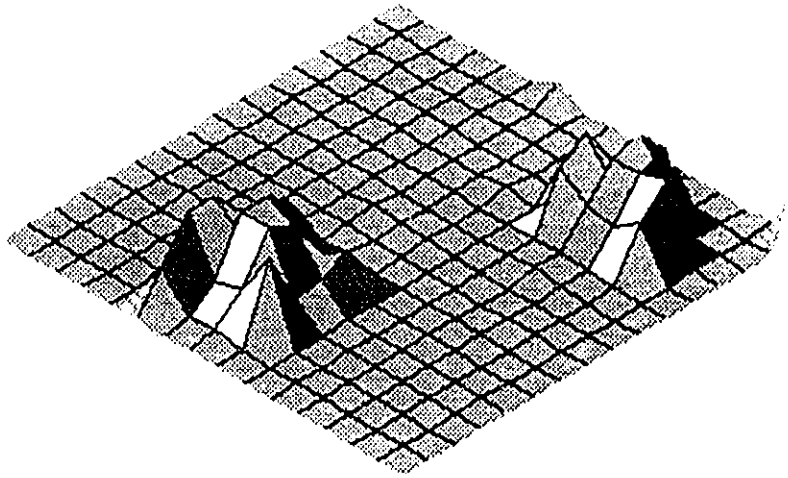


Figure 4.15 - "Two" example 3

4.2.3 Network Development

The network is developed through application of the ideas outlined in chapter 3. The number of inputs to the network is fixed to equal the number of force sensitive points on the tactile sensor (i.e. 16 x 16 = 256). The number of outputs is set to equal to two: A neuron to indicate one finger press is present and a neuron to show that two finger presses are present. In two output classifications it would be possible to use a single neuron to show "One" and Not "One" (i.e. "Two"). However, this implies that the network has no means by which to produce an "unsure" verdict for its classification. This is undesirable in a practical implementation since, an automata controlling the sensor if presented with an unknown input, could theoretically adjust the force and position of the sensor until a known classification is detected. The certainty of the classification is then extremely high. If a single neuron is used then every pattern despite ambiguity, is forced into one of the two classifications possibly in error. Therefore, there would exist no procedure by which the accuracy of the classification could be enhanced.

The number of neurons in the second layer is found from the number of clusters, which as a minimum is equal to the number of outputs. The minimum number of clusters is two and therefore, the minimum number of layer 2 neurons is two and layer 1 neurons is 1 (from Equation 3.08). Initial the network is built around these values and an original training set is then selected.

4.2.4 Training Set Selection

The selection of the training set is a heuristic process and is dependent on the problem domain. As described in chapter 3, it is desirable to choose boundary samples to correctly define the classification regions. However, as no *a priori* knowledge of the boundary locations exists the initial set is chosen in general position. This set should represent the domain of the problem and therefore, patterns should be selected to characterize the types of abstractions that it is desirable for the network to make. In addition, any obvious correlation's to unwanted abstractions, should be avoided (for instance the network should not learn to assume that two fingers will always produce a larger sensor impression than one). For the example of the "One Two" net Figure 4.16 depicts these abstractions.

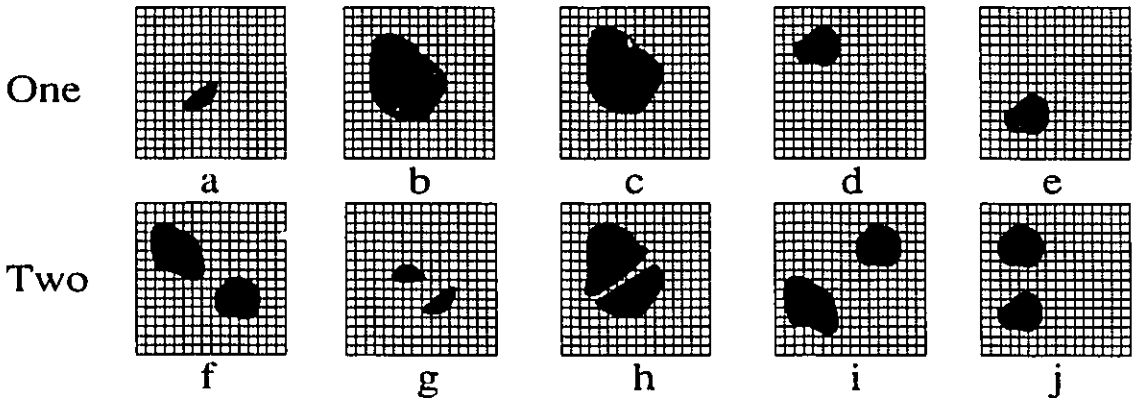


Figure 4.16 - Examples of One-Two net abstractions

Patterns a,b,f and g show examples of the force independence abstraction. In a and f the total force produced by one finger is less than the force produced by two. However, in b and g this is not so and therefore, patterns of both types should be in the initial training set to avoid the network inadvertently learning a correlation between total force and number of fingers. Another correlation the network may learn is to relate certain impression shapes to the number of finger presses. Patterns of types c and h are included to alleviate this problem. In addition the finger presses should be distributed throughout the surface of the tactile sensor to avoid the problem of classification by position. Patterns d,e,i and j represent examples of this type of distribution.

Although the initial training set is important, it should not be too large for the following reasons: the iterative training process will add boundary samples and is therefore, a better means of selecting training images and the number of neurons in each layer is likely to be

too low and therefore, a faster training process is required to determine this lack and allow the trainer to correct the problem in a timely fashion.

Keeping in mind the constraints outlined in the previous paragraph, the initial training set was chosen to be equal to 30 images. This allowed for a small training set and the inclusion of all the abstractions already identified.

4.2.5 Building the Network

If as with the One-Two net, the number of neurons originally selected is too few then more must be added. However, since there are two variables in this problem (first and second layer neurons), it is impossible to have a purely systematic approach to the selection of which layer should be augmented and by how many neurons. Therefore, the following procedure was adopted:

i) Add neurons to both layers

ii) Check for network convergence

iii) If network will not converge [2] then go to i) else go to iv)

iv) Remove a neuron from the second layer

v) Train network and check for convergence

vi) If the network converges, then go to step iv)

vii) Add the neuron back into the second layer

viii) Remove a neuron from the first layer and train and check for convergence

ix) If the network converges then go to step viii)

x) Add neuron back into the first layer

The procedure outlined above, is guaranteed to derive the minimum network topology for a given training set. However, it can be a slow procedure to implement if the number of neurons in the first and second layers is significantly different. The use of colour variations to indicate different neuron states can be used as an aid in determining the size of a network. Usually a layer with too many neurons will have some that are not used as part of the solution to the problem. These would appear as neurons with intermediate states for all input patterns (i.e. some shade of gray) or one to which all weight connections have small magnitudes. If either of these phenomena is apparent for a neuron added to a layer then normally, that particular layer is of sufficient size to perform the current task and no new neurons are needed for that layer. The initial stages of development of this network are depicted in Figures 4.17 through 4.20. As shown by 4.17, the second layer has more than enough neurons.

---

[2] Lack of convergence is generally characterized by one or more of the output neurons responding to an input pattern with the wrong polarity, that is a state of -1 when it should have been +1 or vice versa .

This is evident from the observation that the neurons are in an indeterminate "gray" state and the weight connections to the output neurons are of low magnitude. As neurons are added to the first layer Figures 4.18 and 4.19, the activity in the second layer increases however, the network will not converge until there are 16 neurons in the first layer as in Figure 4.20. These figures do not depict the weights connected to the input layer as there are 256 inputs and therefore, their display is much too entangled to be instructive.

To speed the development of this network it was decided to add neurons to the first hidden layer in groups of four after the fifth iteration.

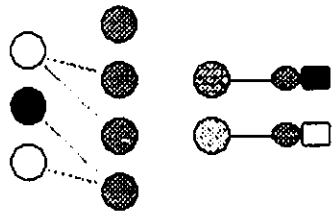


Figure 4.17 - third stage of development

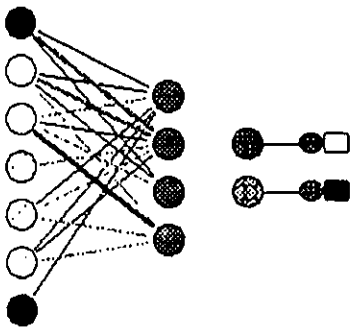


Figure 4.18 - fifth stage of development

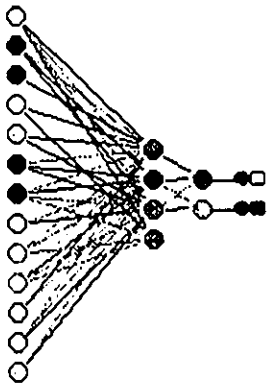


Figure 4.19 - seventh stage of development

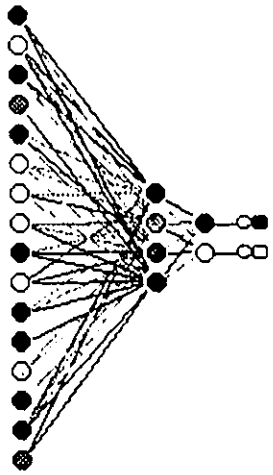


Figure 4.20 - tenth stage of development

After the number of neurons in the first layer reached 17 their numbers were decreased until convergence no longer occurred. A similar procedure was applied to the second layer and the result was 14 neurons in the first layer and 2 in the second. Besides modifying the number of neurons in a layer it is necessary to adjust the learning rates and weight momentum downwards as the number of neurons in the network increases. If this is not done then the network may exhibit an oscillatory learning behavior. In this circumstance the network appears to progress towards convergence and then move past the global minima to some new equilibrium point and then again towards convergence. To detect this behavior network developer shows the percent of correctly identified patterns during learning. The oscillatory behavior is evident from this display when the percentage approaches 100% and then moves back towards 0%, frequently within one learning iteration. Following this initial development more patterns were added to the training set through the procedure described in chapter 3 and the number of images appended to 60 by iteration 23. The minimum network size for the 60 pattern training set was found to be 15 in the first layer and 4 in the second. This is very close to the final network configuration that involved 16 neurons in the first layer and 4 in the second. Figure 4.21 depicts variations in the number of neurons through the initial stages of development, including those that do not converge.

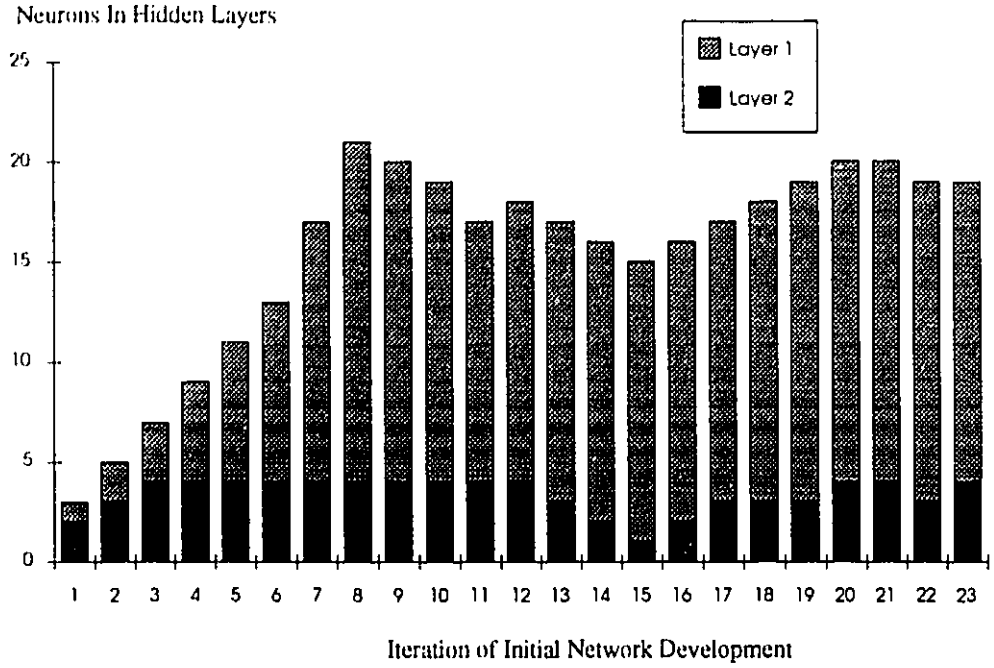


Figure 4.21 - Initial stages of network development

Figure 4.22 details the change in learning rate and weight momentum parameters during this development. Whereas, figure 4.23 depicts the number of iterations required for those networks that reached convergence and those that did not.

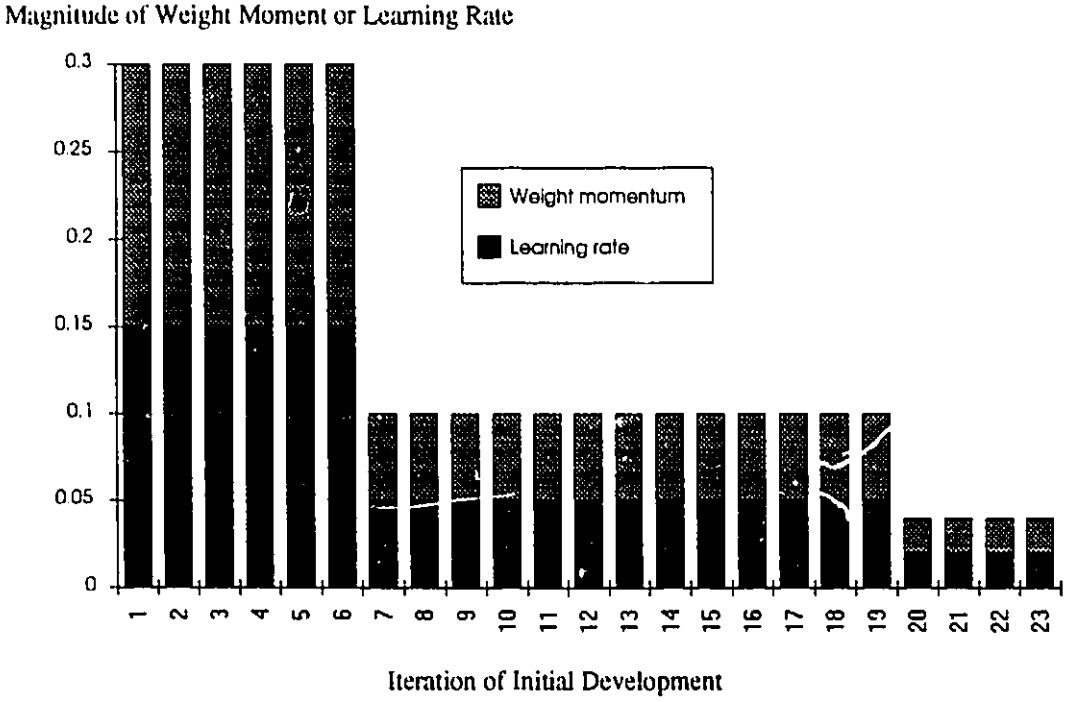


Figure 4.22 - Change in weight momentum and learning rate parameters

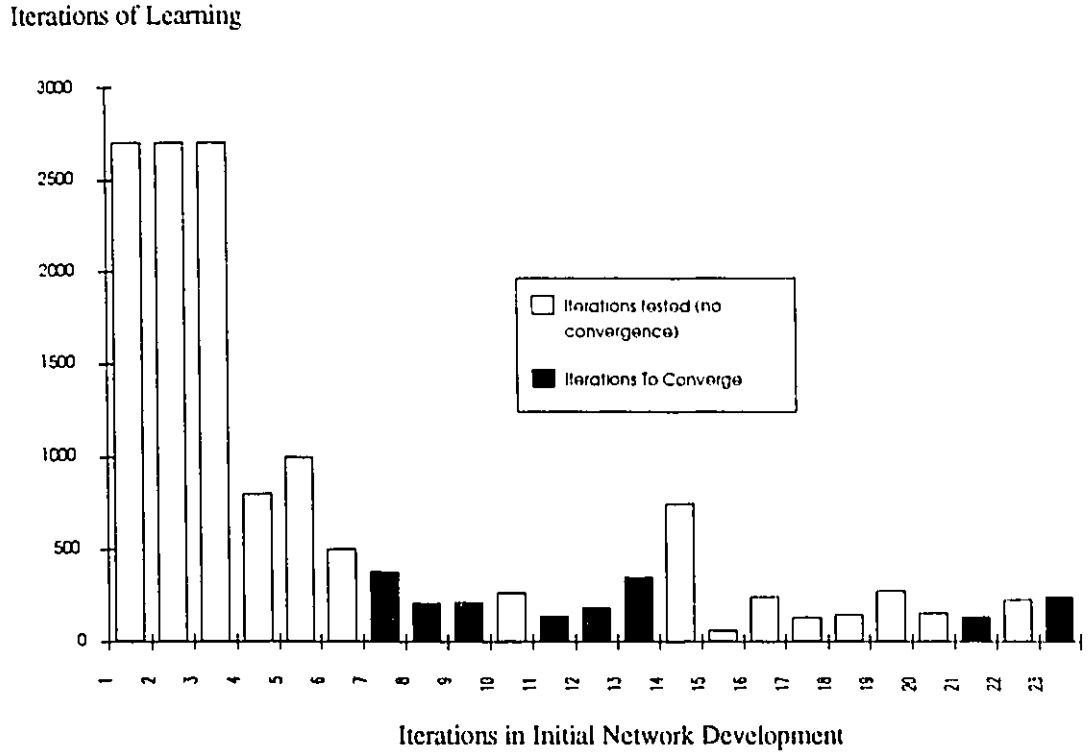


Figure 4.23 - Iterations required for nets that converge and those that do not

Following the first two stages of development the network is advanced through the iterative procedure of test and pattern addition as outlined in chapter 3. In total thirteen iterations (beyond those to find the initial convergence point) were developed. The criterion for determining the termination of the development cycle was derived from application of a 200-image test set. If the test set showed that the network was no longer improving in recognition ability then no more iterations were performed. In Figure 4.24 the number of patterns used to train each stage of development is depicted. During the development process it was sometimes necessary to delete patterns from the training set as the hyper polygon positions become redefined. In essence, some patterns are sufficiently corrupted by noise that they cross the threshold between classification areas. In the previous training sets these patterns may be used to define the boundaries to include themselves, as later examples become available the boundaries may shift towards their correct positions.

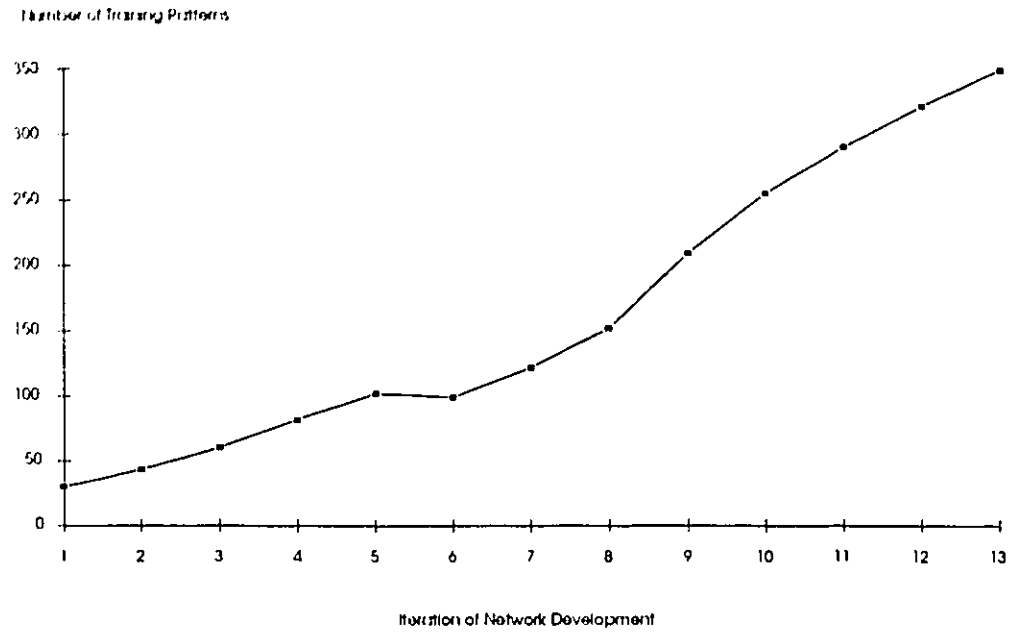


Figure 4.24 - Number of training patterns in iterations of the One-Two net

To show the correlation's between the number of iterations required for learning and the pattern set size the network was retrained with each of the training sets using a constant learning rate and weight momentum. The results of this test are demonstrated in figure 4.25.

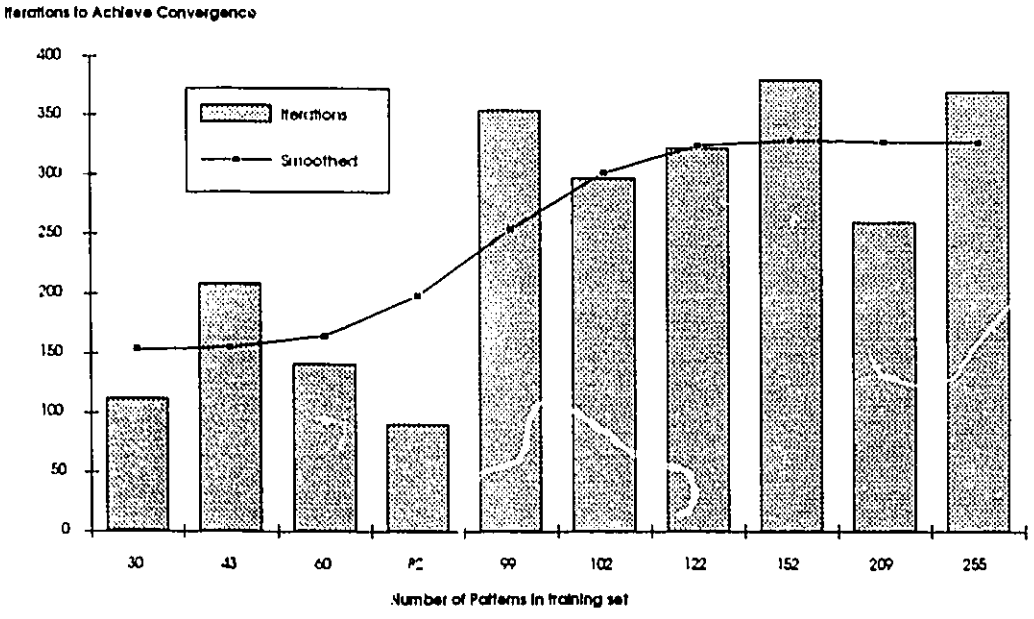


Figure 4.25 - Number of iterations until convergence with a learning rate of .02 and weight momentum of .02

The smoothed curve depicts an apparent correlation between the number of patterns and the required number of iterations to achieve convergence. However, the data shows that this number can vary drastically for any given instance. This is attributable to the actuality that the initial weights in the network are set to random values and therefore, the process of moving between the current error in weight space and the true minimum is a stochastic process. Figure 4.26 displays the variability of convergence from different randomly chosen starting points in weight space. These datums were collected from the training set that contains the 30 initial patterns with a learning rate of .05 and a weight momentum of .05.

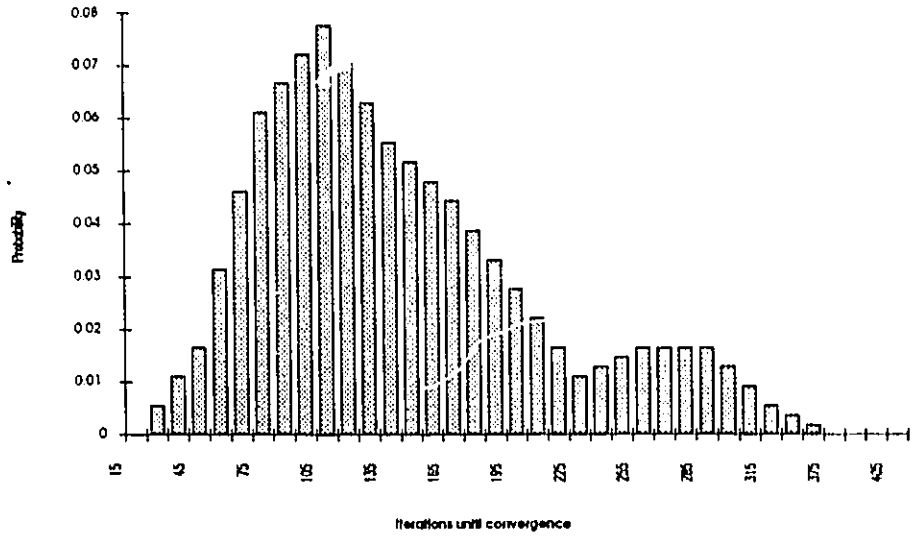


Figure 4.26 - Variance in iterations to convergence from random initial conditions

From the graph apparently the number of iterations until convergence is greatly dependent on the initial weight matrix.

Following development of the network a set of test patterns independent of any of the training sets, but gathered in the same manner, was used to gauge the effectiveness of the network as applied to the One-Two classification problem.

4.2.6 Results

Figures 4.27 through 4.30 depict the results of the training regime when the iterations of the network are assessed against the two hundred test images.

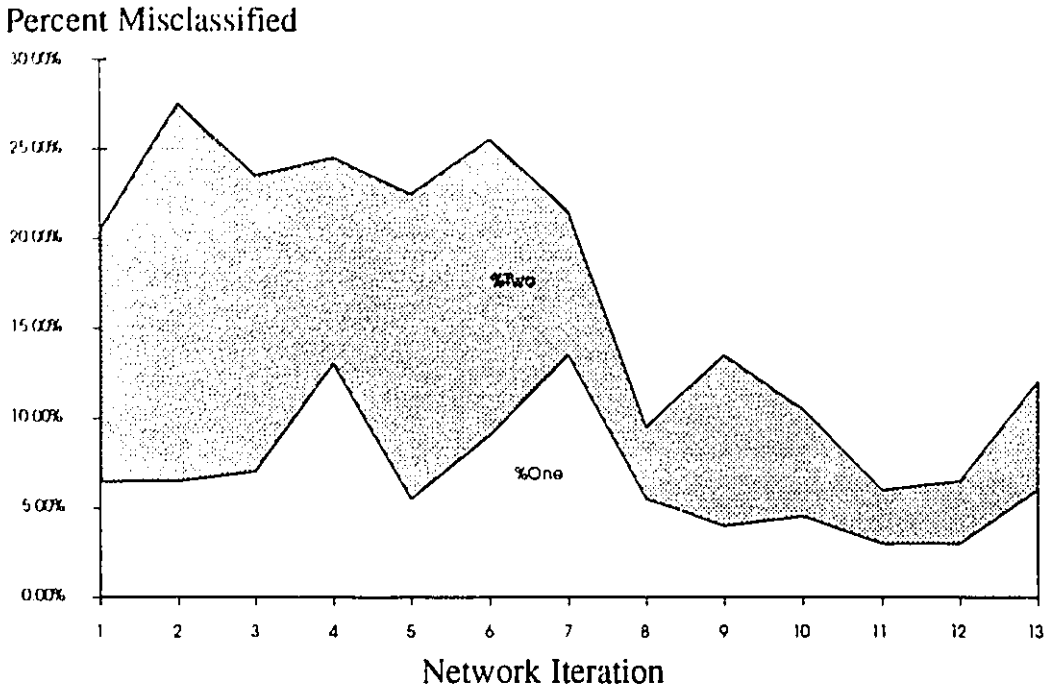


Figure 4.27 - Percentage of misclassified patterns against iterations of the training set

In Figure 4.27 the percentage of misclassified patterns (those classed as the other type) is graphed against the development iteration. The indication for "%One" implies that these patterns were intended to be classed as ones but were misclassified as twos in error. Similar logic applies to the "%Two" class, those that should have been classed as twos were classed as ones in error. The iterative training scheme successfully transformed the 20.5% misclasifications from the original training set to 6.0% misclasifications with the final set. Also, of interest is the change in the relative percent of misclassified patterns of ones and twos. While appending new patterns to the training set the same number of each type of pattern, either "One" or "Two" was input to the previously trained network for classification. If the network had particular problems with either of the two patterns then more of the images of that type would be added for training during the subsequent iteration. Because of this, the next iteration of the network may have a greater or lesser number of patterns of any particular type (i.e. the numbers of "Ones" may not balance the number of "Twos"). Therefore, this paradigm leads to an automatic correcting of balance in the subsequent network iterations. This phenomenon is particularly apparent between iterations 8 and 9. In iteration 8 the percentage of misclassified "Twos" is 4.0% and 5.5% for "Ones." Whereas, in iteration 9 the shift away from "Twos" is apparent, in this iteration the percent of misclasssified "Twos" is 9.5% and 4.0% for "Ones." By iteration 11 the balance of 3.0% each is reached.

Figure 4.28 depicts how the relative number of "Twos" and "Ones" changes in the training set for the various iterations. During iteration 13 the number of misclassified patterns increased. As shown by Figure 4.29 the number of patterns classified as "Unknown" decreased during this iteration. The explanation for this is that the learning capabilities of

the network have saturated with the training set of 349 patterns and therefore, the network has divided its classification space into "Ones" and "Twos" with little space left unclassified. Therefore, most of the patterns will be strongly classified either correctly or incorrectly.

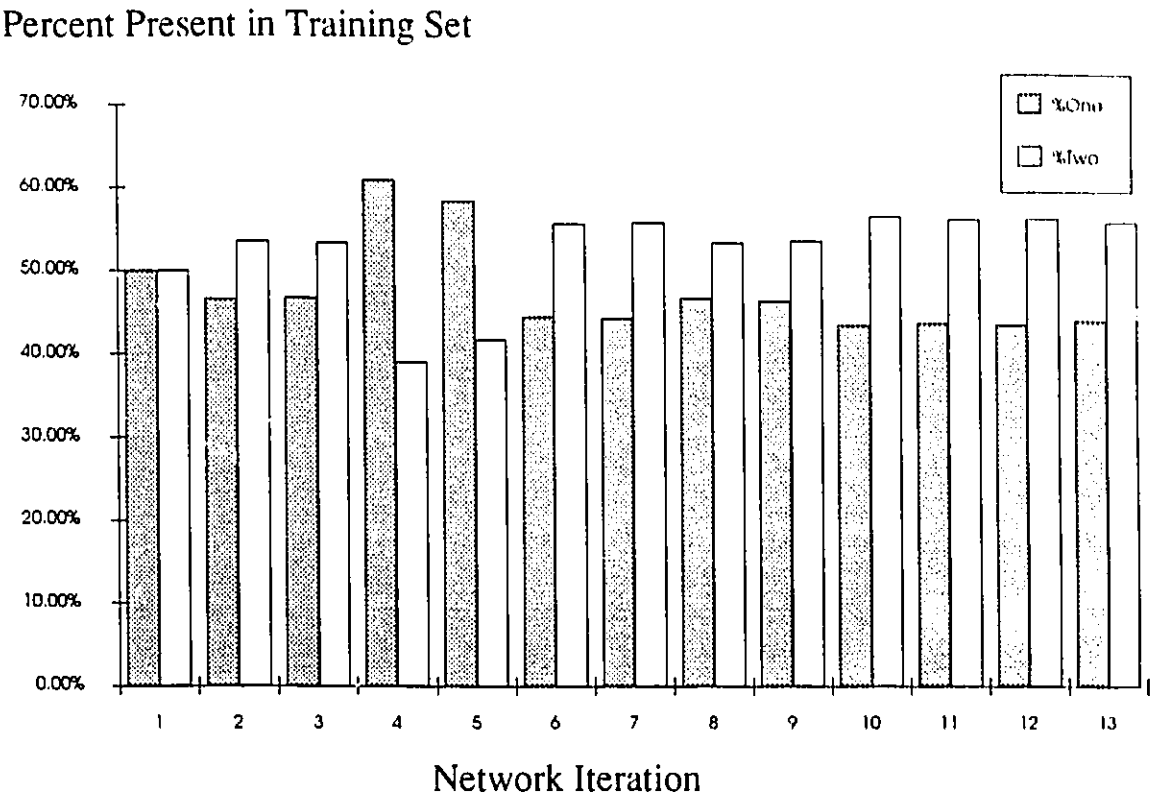


Figure 4.28 - Relative percentage of "Ones" and "Twos" in training set

Figure 4.29 depicts the second source of error in the network, those patterns that were not clearly classified as either "One" or "Two." Network developer uses a thresholding algorithm to decide if the current outputs clearly identify a class or not. This algorithm is based on a variable "Recognition Criteria." If the currently most active neuron's output is greater than the "Recognition Criteria" and no other neuron's output is within twice the "Recognition Criteria" then the classification is considered clearly defined. In any other circumstance, the classification is questionable and the network is "unsure." For instance, the "Recognition Criteria" for all networks developed in this thesis was set at 0.6 so that comparisons between networks could be drawn. Therefore, if the most active neuron's output was 0.9 (above the 0.6 value) then the second most active neuron must have an output less than -0.3 (i.e. 0.9 - 2*0.6).

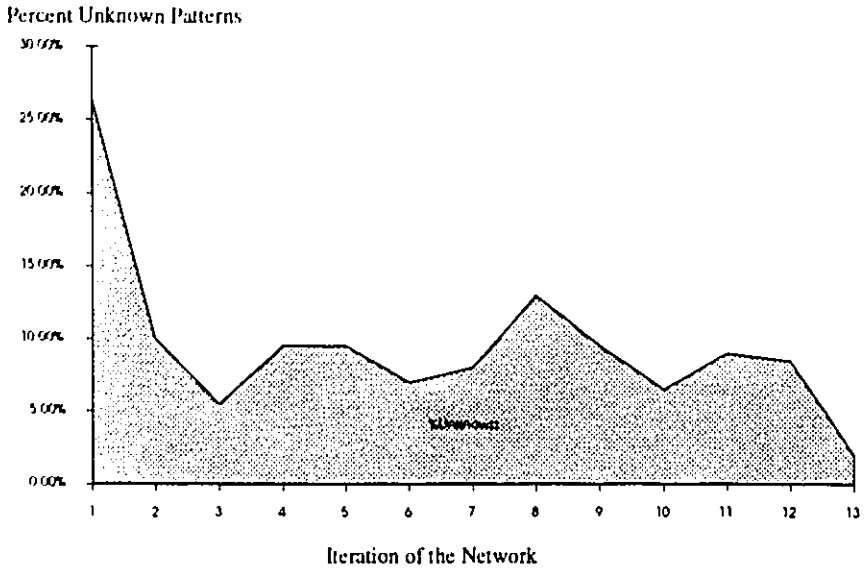


Figure 4.29 - Percentage of patterns of unknown classification against iterations of the training set

The number 0.6 is harsh in this context in that images on the border are sometimes classed as "Unknowns." However, it is considered less of an indemnity to have an "Unknown" classification rather than an incorrect classification. If the class of an image is "Unknown" there exists the possibility of further activity, minor positional changes and force changes etc., which could result in the correct classification. Whereas, an incorrect classification is likely to prompt some improper system action that is based on this input. Therefore, iteration 11 or 12 of the network is considered to achieve the optimum performance for this problem domain.

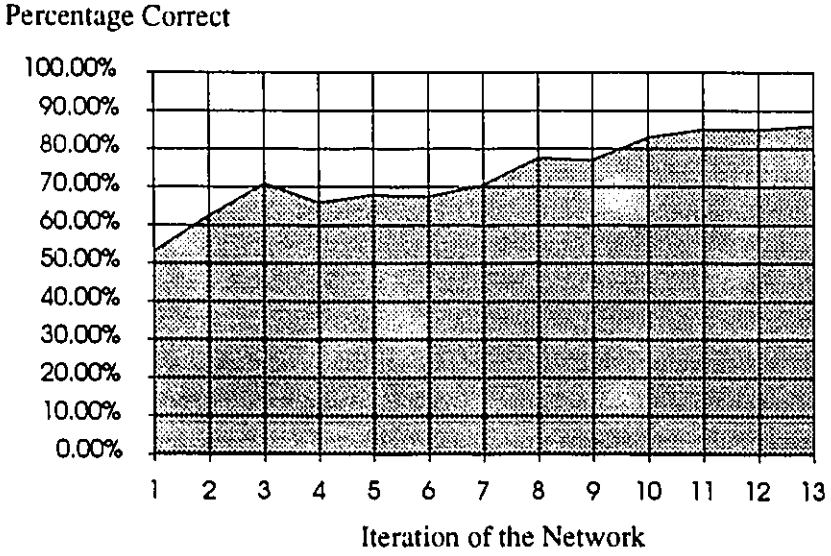


Figure 4.30 - Percentage correct as a function of network iteration

Figure 4.30 depicts the overall results of the training regime. The percentage of correctly classified patterns (vertical axis) is graphed against the iteration of the network (horizontal axis).

4.2.7 Network States

Figures 4.31 and 4.32 depict the neuron states in the first and second network layers for iteration 12. Each figure shows five different patterns of states each of which is typical of those present for a recognized "One" or "Two." All of the 85.5% correctly classified patterns in the test set fall into one of these ten state categories or are minor deviations from them. For instance, the first neuron in the hidden layer may be more or less active etc.

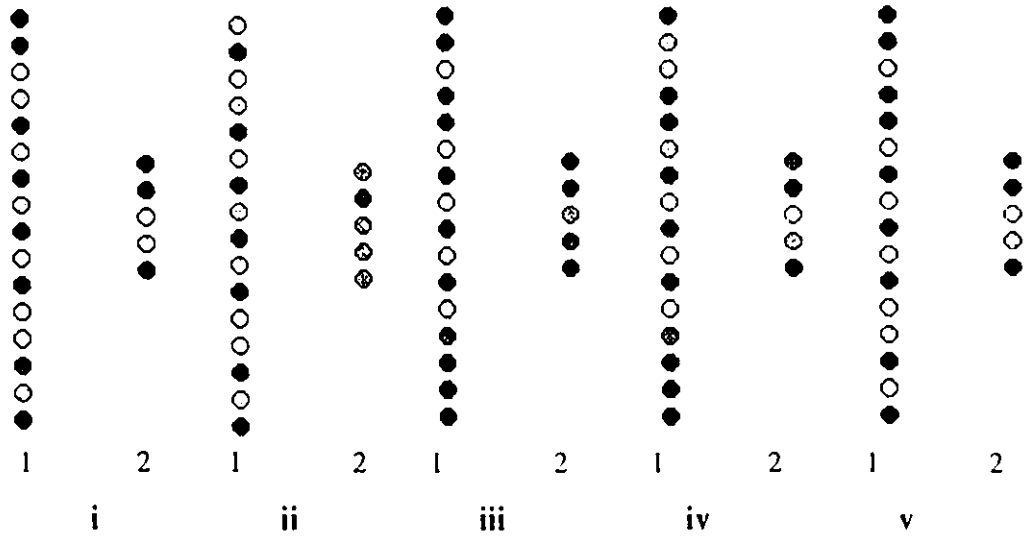


Figure 4.31 - Neuron states resulting in a classification of "One"

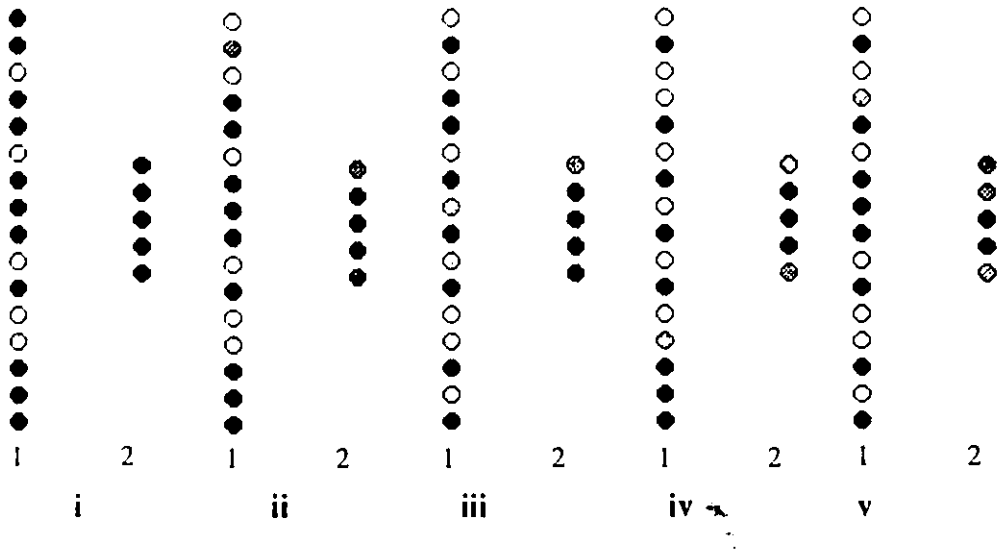


Figure 4.32 - Neuron states resulting in a classification of "Two"

In isolation, these patterns of states are difficult to interpret however, it is educational to compare them to the patterns of states produced by misclassified patterns. Figure 4.33 presents states from patterns intended for classification as "One" that were classified as "Two" in error. Figure 4.34 depicts the relationships for patterns intended for classification as "Two." To aid in correlation of the relationships between these states and those of Figures 4.31 and 4.32, Table 4.01 was developed. In this table the number of state differences between the miscast patterns and the correctly classified patterns is shown. A state difference occurs when a neuron is in the opposite state for the state pattern in question to that of the state pattern to which it is compared and only for the first hidden layer. On occasion, the neuron is "Gray" that is, somewhere between the two states and therefore, this is indicated as a possible (e.g., 2/3 indicates that there are two or three state differences between the state patterns in comparison and therefore, one state is Gray). Each of the four misclassified state patterns is compared to the ten patterns of Figures 4.31 and 4.32.

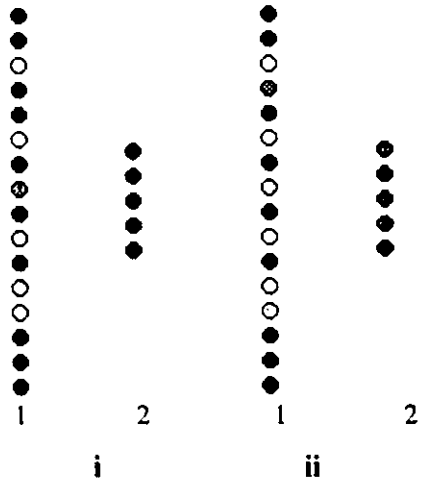


Figure 4.33 - Neuron states for patterns that should have been classified as one but were not.

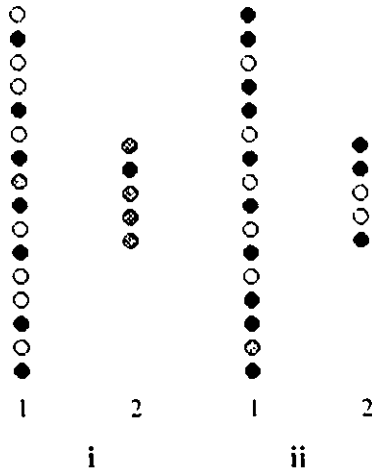


Figure 4.34 - Neuron states for patterns that should have been classified as two but were not.

Correctly Classified Pattern Misclassified Pattern	One.i	One.ii	One.iii	One.iv	One.v	Two.i	Two.ii	Two.iii	Two.iv	Two.v
One.i	3	4	2	3	2	0/1	1	3	3	3
One.ii	2	3	1	2	1	1	2	2	2	4
Two.i	1	0/1	4	5	3	3/4	3/4	1	1	1
Two.ii	2 /3	2	0/1	1	2	2	3/4	2/3	3	5

Table 4.01 - Comparison of states of correctly classified patterns and incorrectly classified patterns

From the table apparently the incorrectly classified patterns have a single state change from the states of their incorrect classifications. However, in some cases there is also a single state change from the correct classification, suggesting that this particular set of states is on the borderline for that class. To appreciate the lack of apparent correlation between the states of the hidden layer neurons and the tactile images to which they relate, Figures 4.35 and 4.36 are presented. Figure 4.35 depicts the image misclassified as "Two" from figure 4.33.i. Figure 4.36 depicts an image correctly classified as "Two." There is no apparent correlation between the two images however, both give rise to virtually identical neuron states in the hidden layers (reference figure 4.32.i). The difference is only apparent in the eighth neuron where for the misclassified pattern this neuron is "uncertain" and therefore gray.

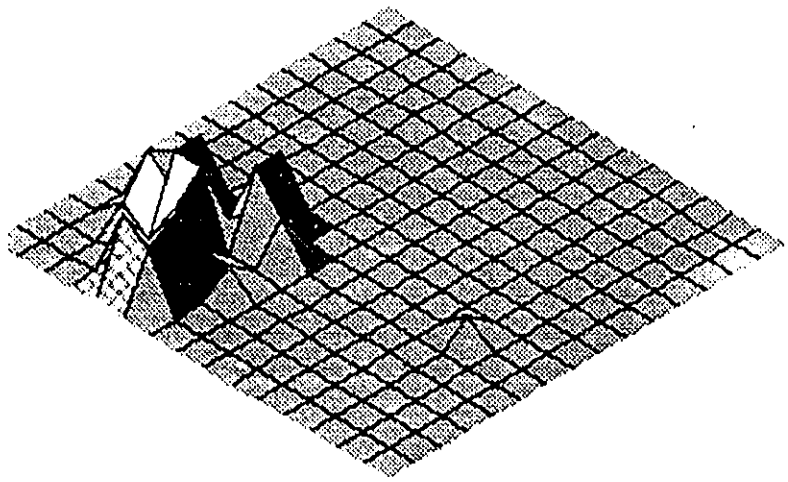


Figure 4.35 - Pattern misclassified as "Two"

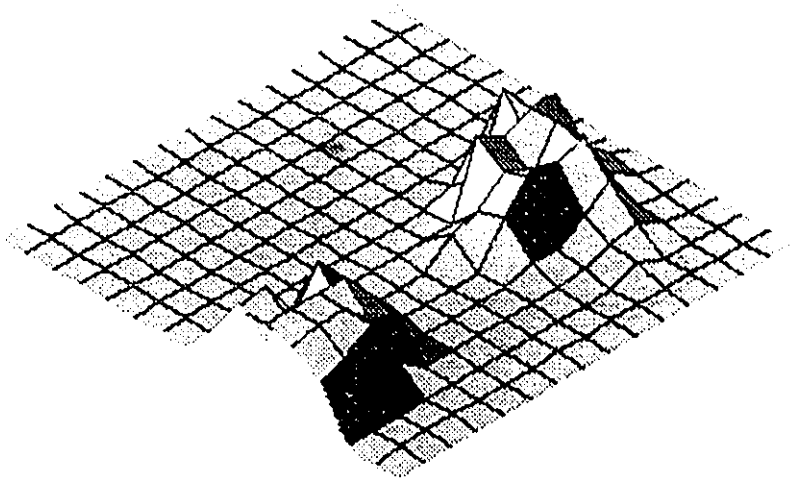


Figure 4.36 - Pattern correctly classified as "Two"

4.2.8 Results Outside Problem Domain

For this network the evaluation of results outside the problem domain are relatively straight forward. The original intent of the network was to recognize the difference between one and two finger presses. Therefore, if more than two finger presses are applied or less than one finger press the response of the network should be educational. Figure 4.37 depicts the results of this evaluation versus the network iteration for the case where no fingers are pressed on the pad. For most cases the result is "One" as expected.

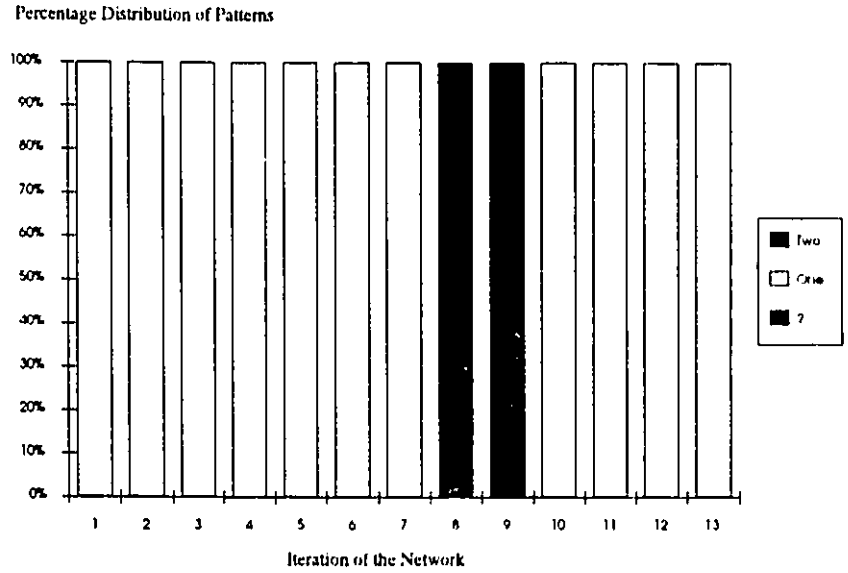


Figure 4.37 - Results of no finger presses versus network iteration

Figure 4.38 depicts the results of applying a test set of 10 images all with three finger presses. As expected for most of the cases, three finger presses were interpreted as two finger presses.

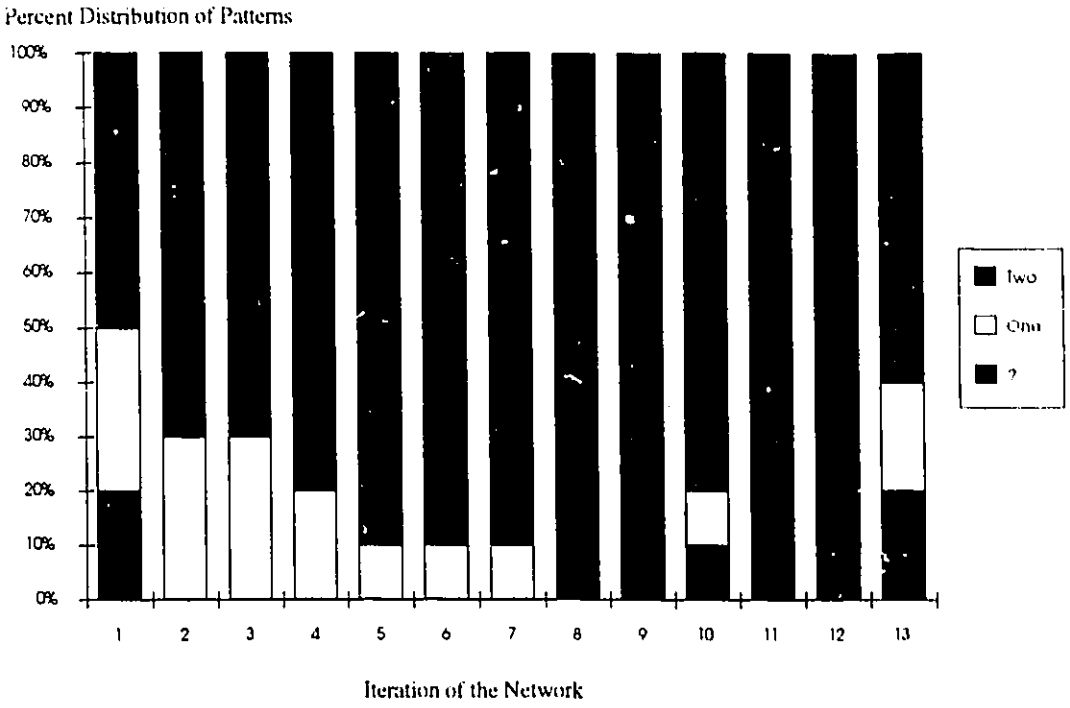


Figure 4.38 - Results of Three finger presses on the pad

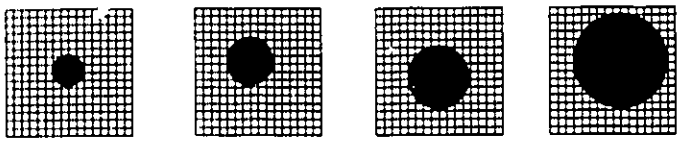
4.2.9 Summary

This section has addressed the results of the development of the "One-Two" net. A network designed to determine if one or two fingers are impressing the tactile sensor. The most effective network was successfully capable of classifying 85.5% of all patterns within the problem domain when evaluated against an independent test set of 200 patterns with a "Recognition Criteria" set at 0.6.

4.3 Big Block Net

4.3.1 Background

The Big-Block net was designed to test the neural network's ability to recognize complex shapes in several orientations. The ability to group several distinct patterns into the same class is important in several types of problems where meta classes are required. Abstract concepts such as "circle," "square" etc. are examples of types of classification problems where many distinct patterns may be grouped into a single class. In terms of neural networks, the members of the class should form distinct clusters in hyperspace and therefore, the number of neurons required in the second hidden layer should be relatively large. Figure 4.39 depicts the idea of "circle" as represented by several subclasses of images.



i ii iii iv

Figure 4.39 - A number of patterns each depicting the concept "circle"

4.3.2 Problem Statement

The "Big-Blocks" problem is a problem of recognition of four embossed letters on wooden blocks. Two large wooden blocks with two letters on each, are used for training and testing of this network. Each letter is approximately the size of the sensor (2.5cm x 2.5cm) and is raised from the surface of the block by 1 to 2 millimeters. To show the network's ability to "group" several distinct patterns into meta classes, the letters were aligned to all four axises of the tactile sensor and therefore, a single letter should be identified in any of the four positions. The letters used both in training and recalls were: **T, H, U** and **G**. No position or force control was utilized in the training or subsequent testing of the network and therefore, the problem is unconstrained and deemed to represent a practical image recognition scenario. Figure 4.40 depicts the 16 possible pattern forms (i.e. 4 letters times 4 orientations).

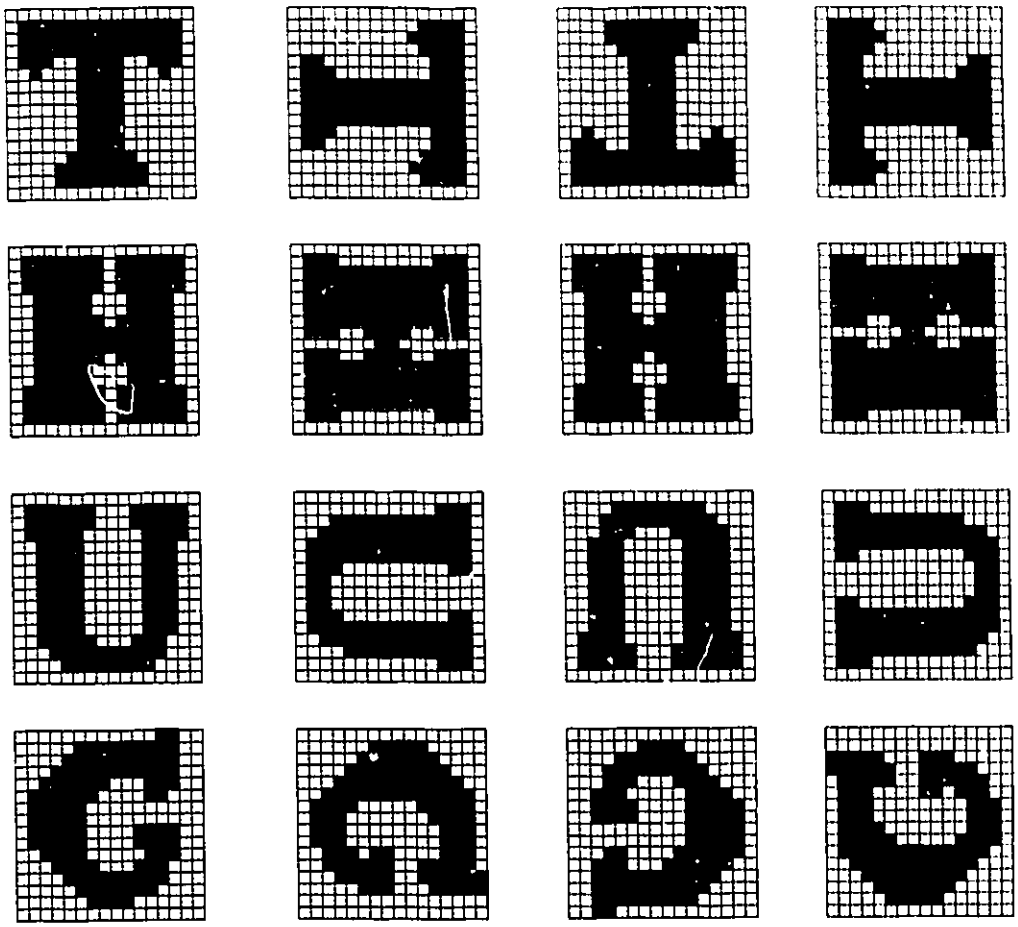


Figure 4.40 - Input pattern configurations

The problem is to detect which letter is applied to the sensor in any of the four rotations. This determination should be independent of the total force applied and also independent of positional translations along vectors parallel to the sides of the sensor surface. Rotations of the patterns by multiples of 90 degrees as depicted in Figure 4.40 are allowed. Rotations by any other amount are outside the problem domain. Figures 4.41 through 4.44 depict examples of the tactile images collected from the blocks. It should be noted that these particular images are relatively noise free in comparison to most of the samples and hence, do not represent the norm for this problem.

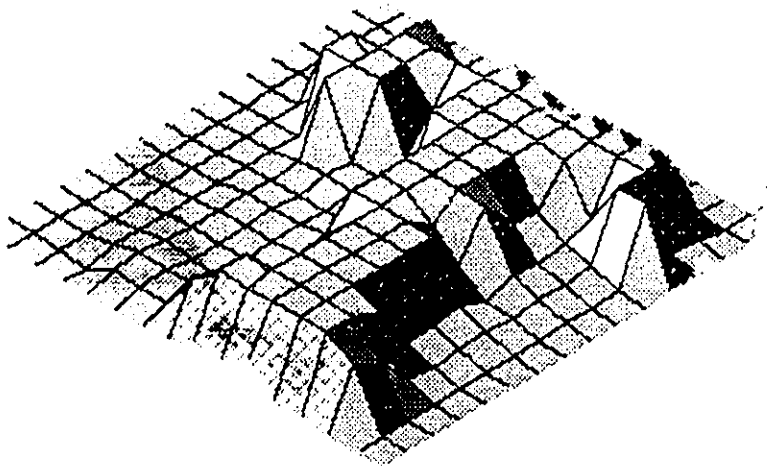


Figure 4.41 - example of "**T**"

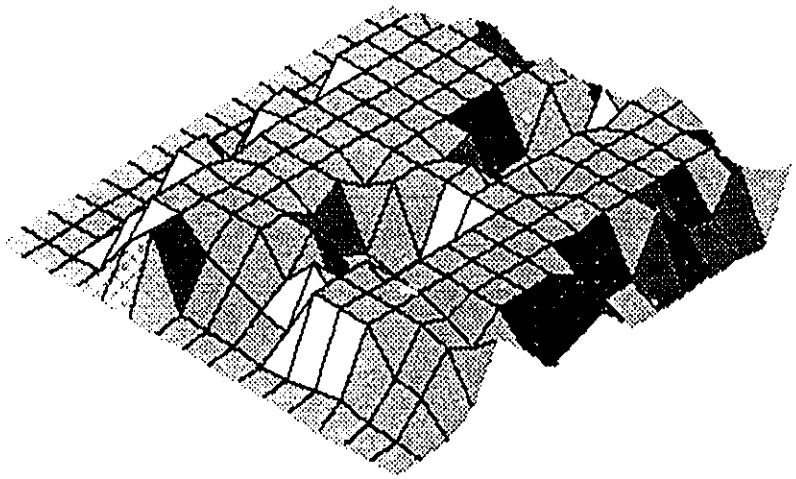


Figure 4.42 - example of "**H**"

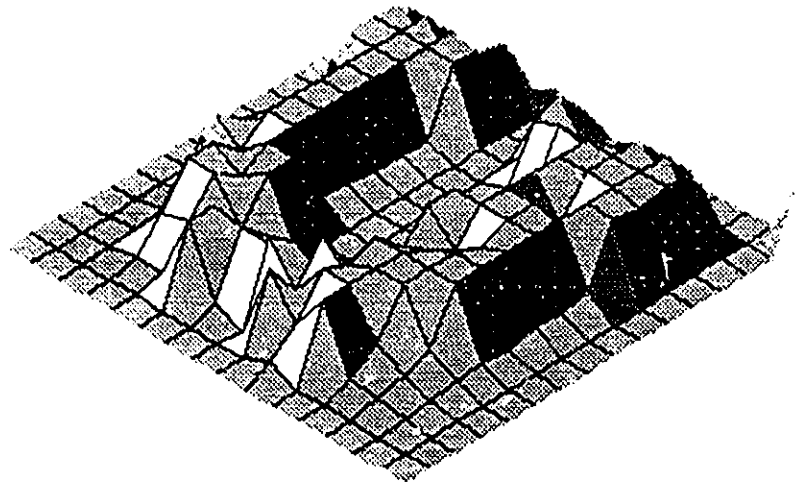


Figure 4.43 - example of "**U**"

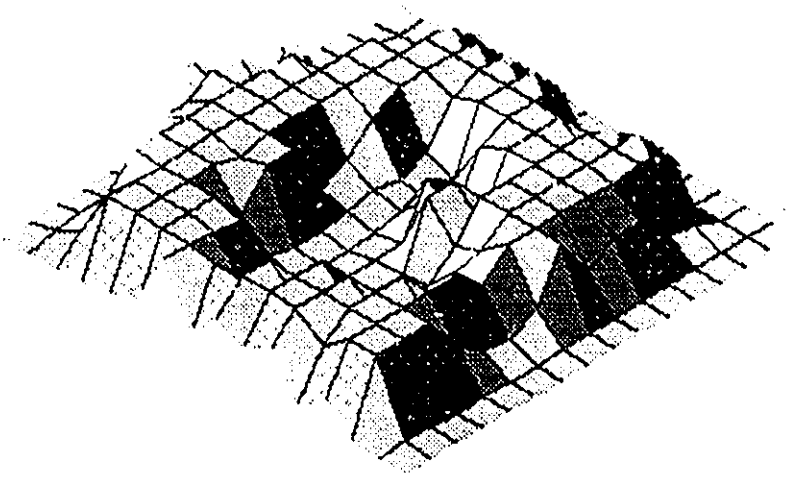


Figure 4.44 - example "G"

4.3.3 Network Development

The network is developed by application of the concepts outlined in chapter 3. The number of inputs to the network is fixed to equal the number of force sensitive points on the tactile sensor (i.e. 16 x 16 = 256). The number of outputs is set to be equal to four: a neuron to indicate each of the four letter types. The number of neurons in the second layer is found from the number of clusters that as a minimum, are equal to the number of outputs, or four. Although the probable upper bound on this number was sixteen, four letters times four rotations, there is no guarantee that this many are required. Therefore, in keeping with the philosophy of minimization of neuron numbers the geometric mean of the minimum of four and the maximum of sixteen was chosen, eight. The number of clusters is eight and therefore, the number of layer 2 neurons is eight and layer 1 neurons is 3 (from Equation 3.07). The initial network topology is built around these values and an initial training set is then selected.

4.3.4 Training Set Selection

The selection of the training set is a heuristic process and is dependent on the problem domain. As described in chapter 3, it is desirable to choose boundary samples to correctly define the classification regions. However, no *a priori* knowledge of the boundary locations exists and therefore, the initial set is chosen in general position. This set should represent the domain of the problem and therefore, patterns should be selected to represent the types of abstractions that are desirable by the network. In addition, any obvious correlation's to unwanted abstractions should be avoided. For the example of the "Big-Block" net Figure 4.45 depicts these abstractions.

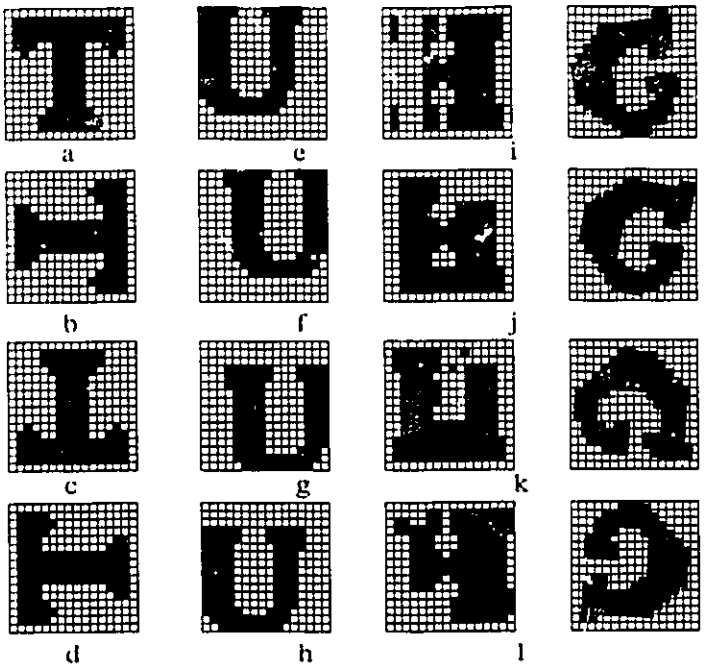


Figure 4.45 - Examples of Big-Block net abstractions

Patterns a,b,c and d depict examples of the rotation independence abstraction. All four alignments relative to the tactile sensor are elements of the problem domain and therefore, all four types should be present in the initial training set.

Patterns of types e,f,g and h should also be included to train the network to learn the concept of positional independence. Since the control of the centering of the image with respect to the sensor is not under strict control, the training images should be distributed throughout the surface of the tactile sensor to avoid the problem of classification by position.

Beyond positional parameters, images i, j, k and l demonstrate another type of property required for this type of image recognition, that of, missing information. Two factors can cause the letters to be partially complete or have excess pressure points: uneven pressure on the sensor and noise. If the pressure on the sensor is not even then, parts of the image may be distorted or absent. If noise is present, extra information unrelated to the pattern may confuse the recognition engine, especially together with incomplete pattern information. To define these boundaries, it is required that incomplete and noisy patterns be present in the training set as they represent the true boundaries between classes. Gathering these types of patterns is usually not a problem as noisy incomplete patterns represent the natural tendency in any case. The more difficult problem is to eliminate images so corrupted by noise that they cross the intended boundaries between classes. Inclusion of these types can cause the network to produce badly defined boundaries or not to converge at all. Pattern k is an example of an H that is close to the borderline in the sense that it is more like a U. Elimination of corrupted patterns is a trainer action based on the visible evidence of the pattern in question.

Patterns m, n, o and p display the final requirement for this network, the ability to recognize minor rotations from the primary pattern locations. The inclusion of these patterns in the training set is again axiomatic with the procedure utilized in acquisition.

The initial training set was chosen to be equal to 16 images. This is the minimum training set that includes all four letters in all four rotations.

4.3.5 Building the Network

The procedure for achieving the initial convergence of the network is similar to that used for the One-Two net as outlined in section 4.2. However, unlike the One-Two net the number of neurons in both the first and second hidden layers varied throughout the subsequent development of the network. Following this initial development phase, images were added to the training set through the iterative procedure described in chapter 3. The number of images in the training set grew to 255 images by iteration 18. Figure 4.46 depicts the variations in the number of neurons in each of the two hidden layers against the iterations of the training set. The fractional iterations are variations in the number of neurons for the initial development and therefore, the same set of training data is employed within each fractional iteration.

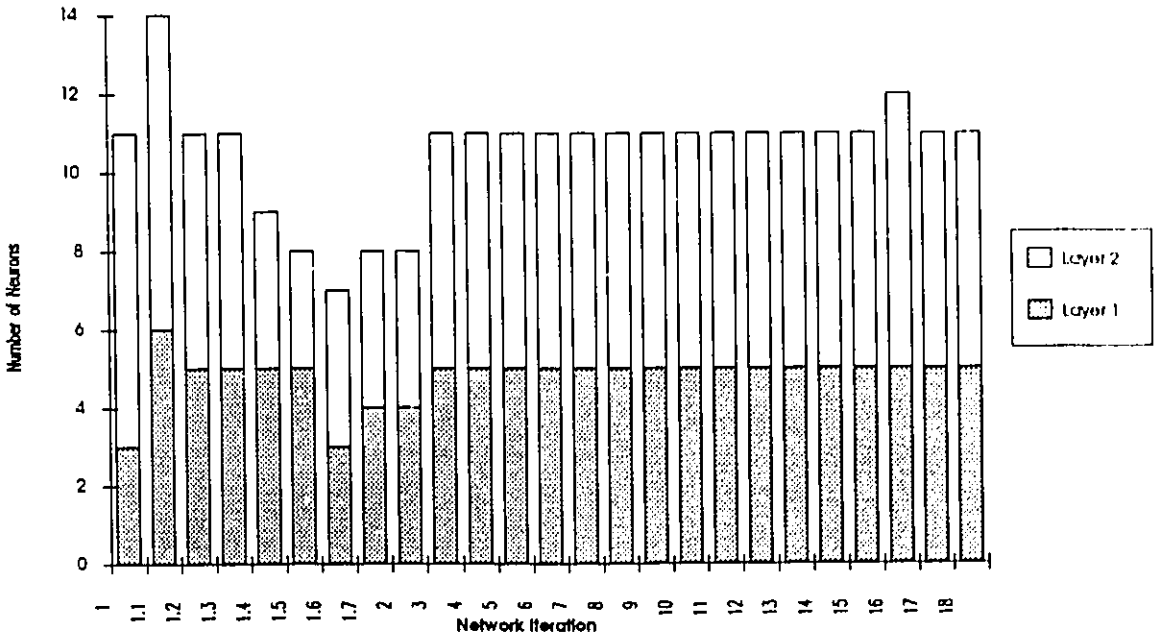


Figure 4.46 - Variations in number of neurons graphed against iterations of network development

The criterion for termination of the development cycle was derived from application of a 120-image test set. If the test set showed that the network was no longer improving in recognition ability then no more iterations were performed. Figure 4.47 shows how many patterns were used to train the network at each stage. During the development it was sometimes necessary to delete patterns from the pattern set as the hyper polygon positions become redefined. In essence some patterns are sufficiently corrupted by noise that they

cross the threshold between classification regions. In the previous training sets, these patterns may be used to define the boundaries to include themselves, as later examples become available the boundaries may shift towards their correct positions.

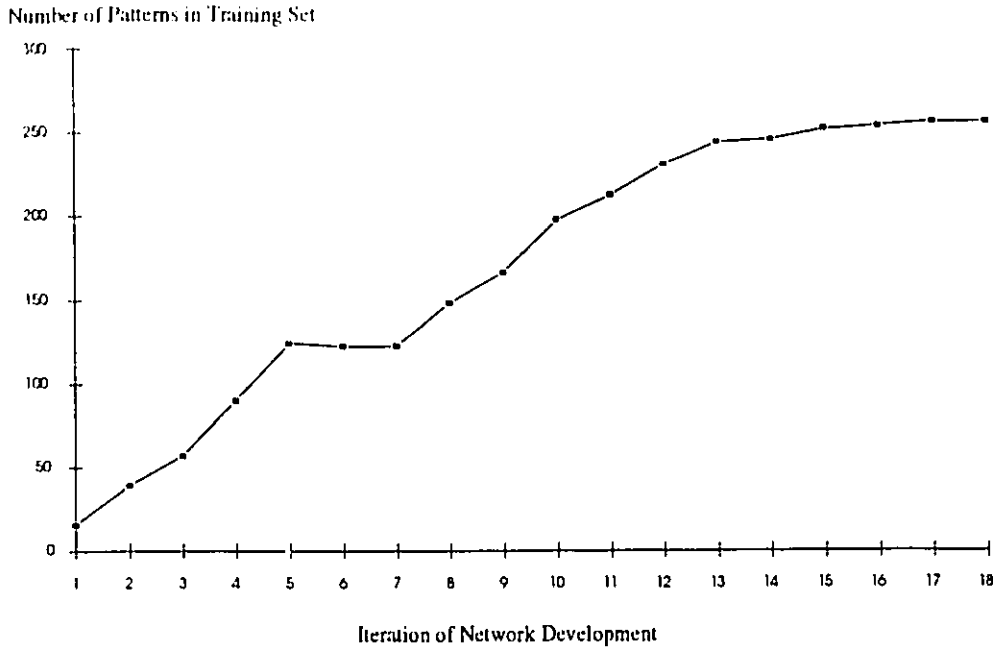


Figure 4.47 - Number of training patterns in iterations of the Big-Block net

4.3.6 Results

Figures 4.48 through 4.50 depict the results of the training regime when the iterations of the network are assessed using the one hundred and twenty test images.

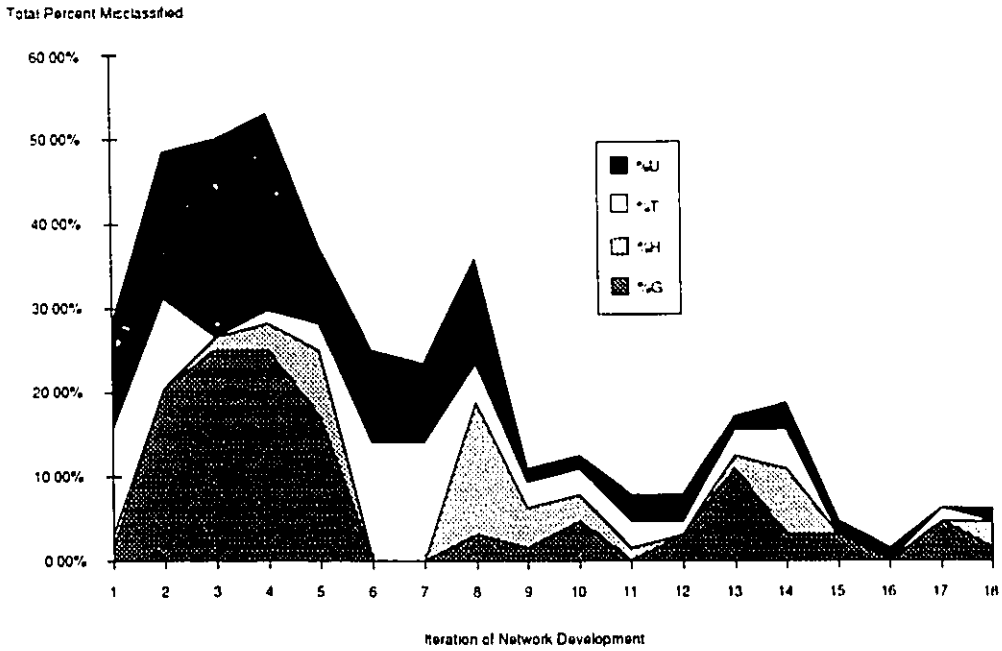


Figure 4.48 - Percentage of misclassified patterns against iterations of the training set

In Figure 4.48 the percentage of misclassified patterns (those classed as one of the other types) is graphed against the development iteration. The indication for "%H" implies that these patterns were intended to be classed as H but were misclassified as another pattern in error, similar logic applies to the other classes of patterns. The iterative training scheme successfully transformed the 28.1% misclassifications from the original training set to 1.56% misclasifications with the sixteenth set. However, at this point the number of "Unknown" patterns were excessive (see Figure 4.49) and therefore, iteration 17 is considered to give the best overall results.

Figure 4.49 depicts the second source of error in the network, those patterns that were not clearly classified as either **T**, **H**, **U** or **G**. In any other circumstance the classification is questionable and the network is unsure.

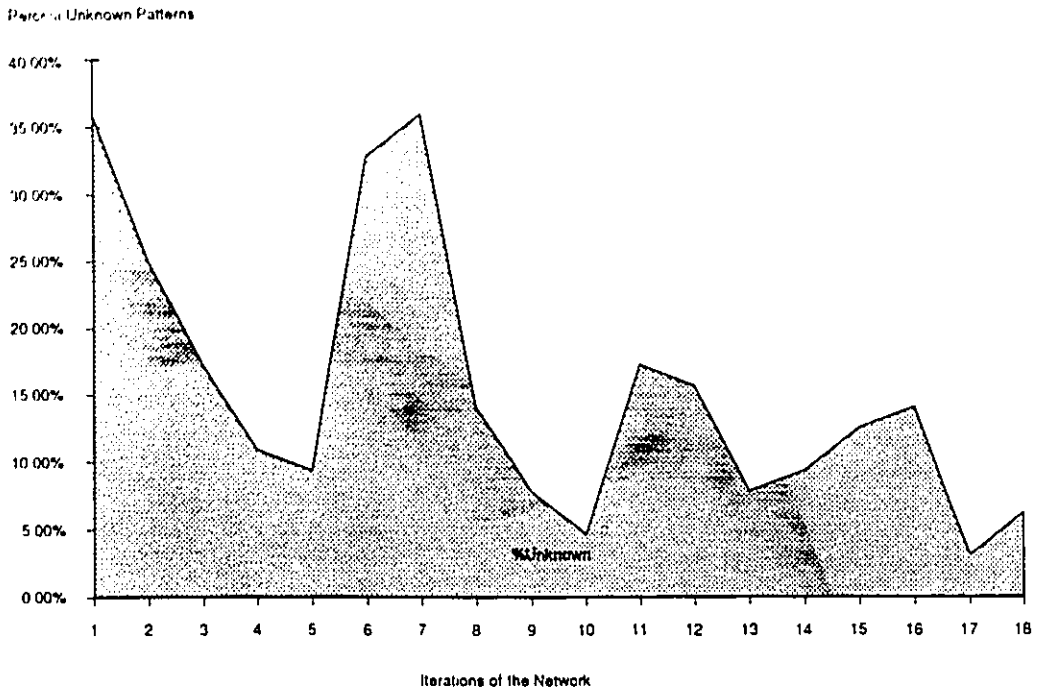


Figure 4.49 - Percentage of patterns of unknown classification against iterations of the training set

Figure 4.50 depicts the percent of correctly classified patterns against the iterations of the network for each stage of development.

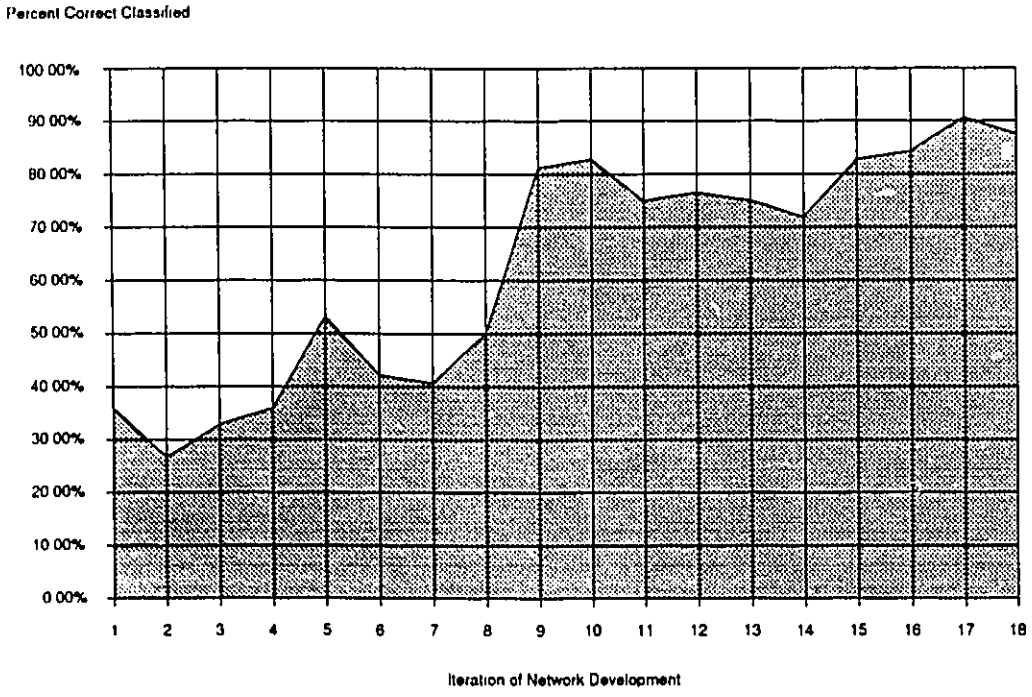


Figure 4.50 - Percent of patterns classified correctly graphed against the iterations of the training set

4.3.7 Network States

Figures 4.51 through 4.54 depict the neuron states in the first and second network layers for iteration 17. All of the 90% correctly classified patterns in the test set fall into one of these state categories or are minor deviations from them.

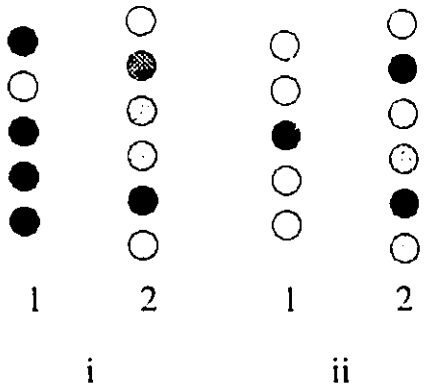


Figure 4.51 - Neuron states resulting in a classification of "**T**"

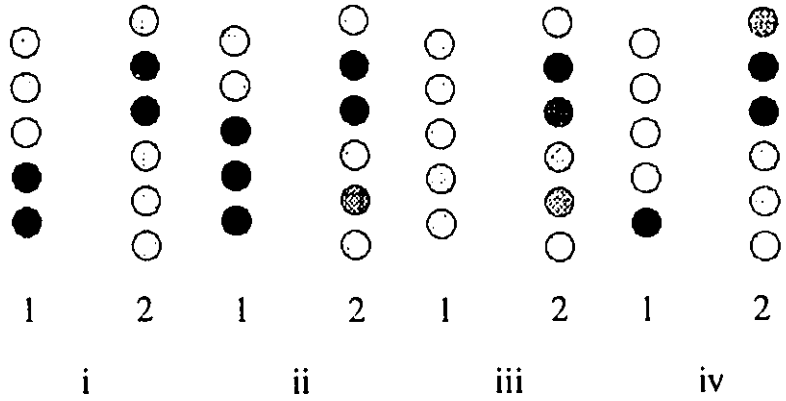


Figure 4.52 - Neuron states resulting in a classification of "**H**"

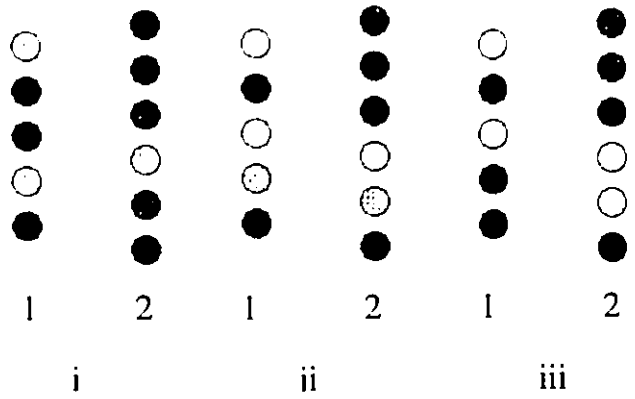


Figure 4.53 - Neuron states resulting in a classification of "U"

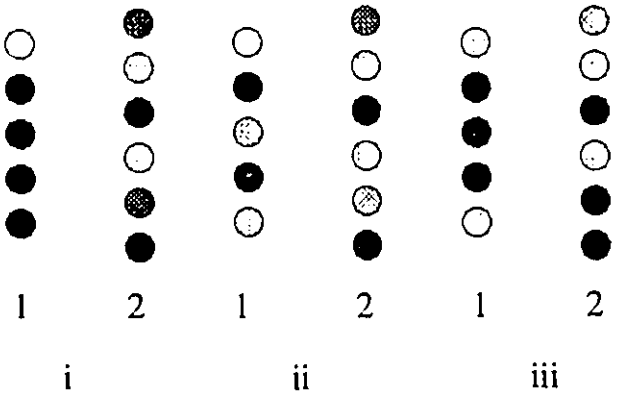


Figure 4.54 - Neuron states resulting in a classification of "G"

As shown by the Figures, the letter T excites the network into one of two states, H into one of four states, U into one of three states and "G" into one of three states. For this network, it is useful to compare the states of the neural network compared to the various rotations of the patterns. Figure 4.55 graphs the various states of the network (as indicated from figure 4.51 to figures 4.54) each state is shown as the pattern type that prompts it followed by a period and a number to indicate which of the states in the set are excited. For example G.1 means the first state prompted by G (labeled i in figure 4.54). Several relationships are apparent from this graph: The letter T has some degree of vertical symmetry and uses the same classes for rotations of 0 and 180 degrees and rotations of 90 and 270 degrees, state T.2 appears in rotations 0 and 180 whereas T.1 appears in rotations 90 and 270. This last relationship is surprisingly not apparent for the letter H though it has a much higher degree of vertical symmetry than T. In addition, no distinct relationships are apparent between rotation and utilized states for H. In fact state H.1 appears in all four rotations. For U, rotation 0 uses only state U.1 and rotation 270 uses only state U.2. Both 180 and 90 degree rotations use two states. G has the interesting relationship that all three rotations 0, 180 and 270 share the same state G.1 whereas, the remaining two states G.2 and G.3 are used for the single 90 degree rotation. For this

network the classification performance for G was poor for the 270 degree rotation where it was frequently misclassified as U.

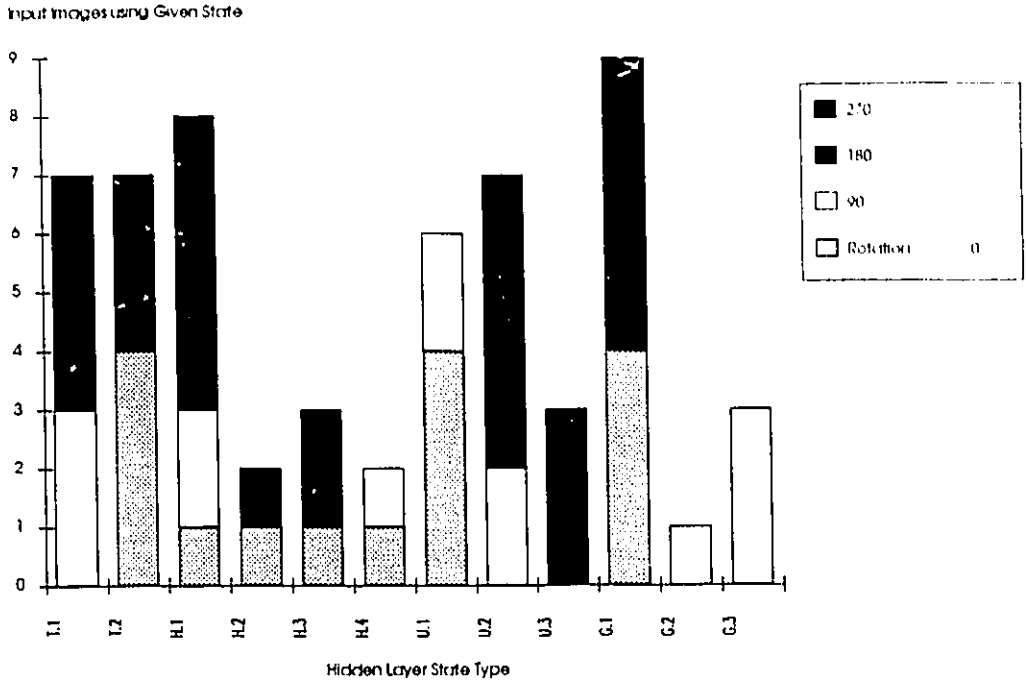


Figure 4.55 - Graph of rotations versus states

Figures 4.56 and 4.57 compare two instances of the letter G in this rotation. What is intriguing about this example is that there is very little apparent difference between the two patterns and yet one was miscast as U whereas the other was correctly identified as G.

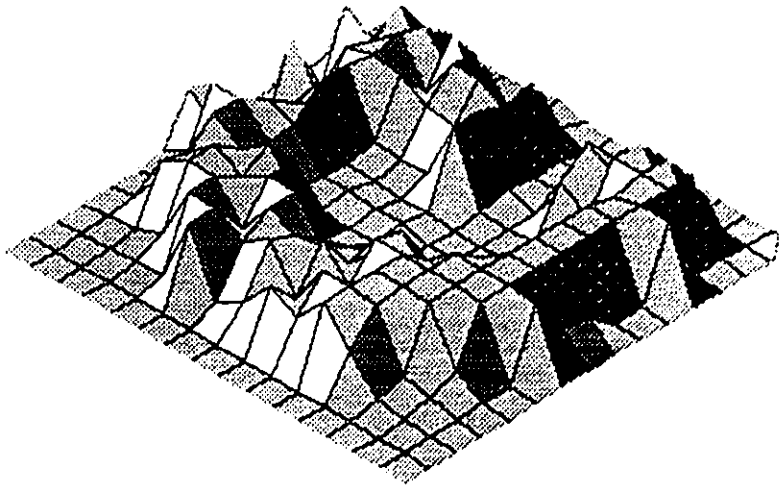


Figure 4.56 - "G" Pattern misclassified as "U"

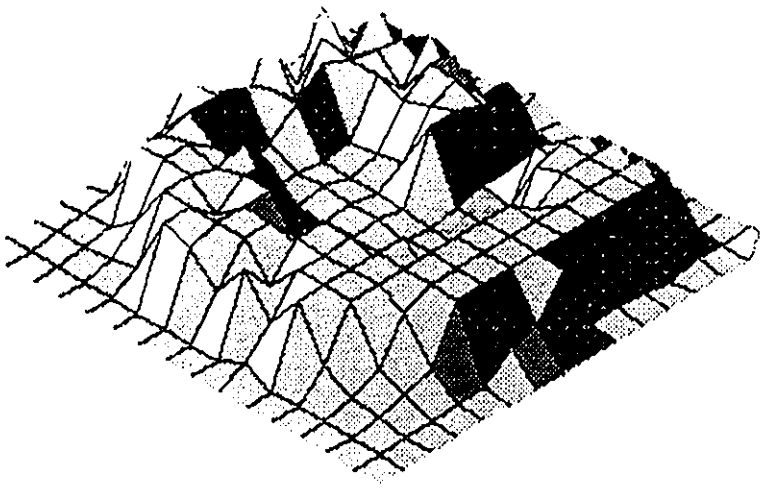


Figure 4.57 - "G" Pattern correctly classified as "G"

4.3.8 Results Outside Problem Domain

The original intent of the network was to recognize the difference between four letters in varying rotations, each 90 degrees apart and aligned with the sides of the tactile sensor. Therefore, rotations of angles of 60, 45, 15 etc. degrees are outside the problem domain and the response of the network to them should be of interest. A test set of images at rotations 45, 135, 225 and 315 degrees were prepared and the response of the network recorded. Four images for each letter at each rotation were used.

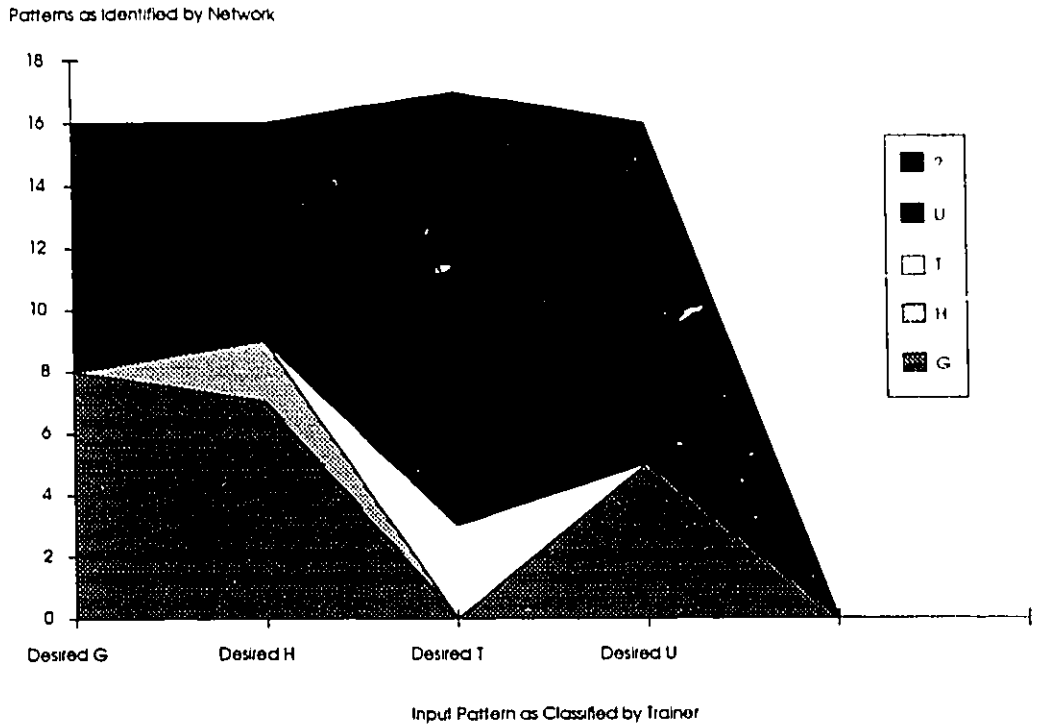


Figure 4.58 - Results of applying rotations outside problem domain

From the figure, there is some apparent correlation between the desired patterns and the classes determined by the network. However, only in the case of U is this correlation significant. Figure 4.59 depicts the total correct classifications versus desired types.

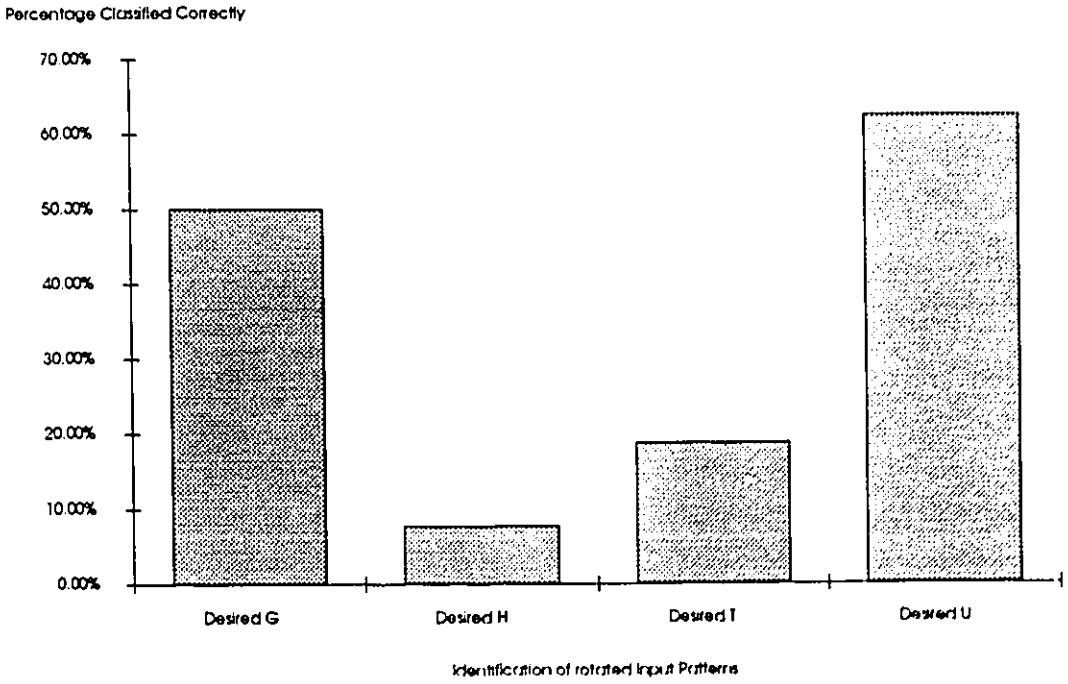


Figure 4.59 - Percent correct classifications by input pattern type

4.3.9 Summary

This section has addressed the results of the "Big Block" net. A network developed that decides if any of the four embossed letters **T**, **H**, **U** or **G** are impressing the tactile sensor. The most effective network was successfully capable of classifying 90.0% of all patterns within the problem domain, when evaluated against an independent test set of 120 patterns.

4.4 Small Block Net

4.4.1 Background

The Small-Block net was designed to test the neural network's ability to distinguish many complex shapes with small features. Since, the tactile sensor produces images with a low signal to noise ratio the small features of the small wooden blocks can be highly corrupted by noise and therefore, there is increased difficulty in distinguishing features. In addition, with many classes of objects the probability of overlap of feature sets becomes more pronounced and therefore, more of a problem for the recognition engine. Figure 4.60 depicts the relative difficulty in classification of small features versus large features especially in the presence of noise.

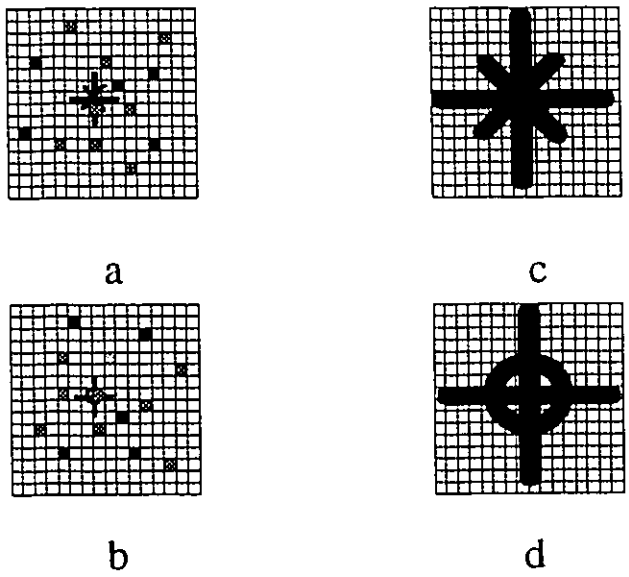


Figure 4.60 - Small and noisy patterns versus large noise free pattern discrimination

Patterns a and c are scaled versions of each other as are b and d. From this example apparently distinguishing pattern a from b is a more difficult problem than distinguishing c from d.

4.4.2 Problem Statement

The "Small-Blocks" problem is a problem of recognition of eight embossed letters on wooden blocks. Four small wooden blocks with two letters on each are used for training and testing of this network. Each letter is approximately half the size of the sensor (1.25 cm x 1.25 cm) and is raised from the surface of the block by approximately 1 millimeter. The letters used both in training and recalls were: **C, D, E, I, L, M, S** and **T**. No position or force control was utilized in the training or subsequent testing of the network and therefore, the problem is unconstrained and deemed to represent a practical image recognition scenario. Figure 4.61 depicts the 8 possible pattern configurations.

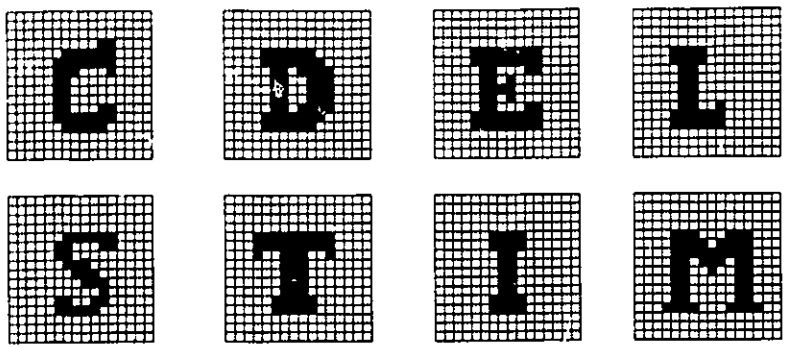


Figure 4.61 - Input pattern configurations

The determination of image type should be independent of the total force applied and independent of positional translations along vectors parallel to the sides of the sensor surface. Rotations by any other than small amounts are outside the problem domain. Figure 4.62 through 4.69 depict examples of the tactile images collected from the blocks.

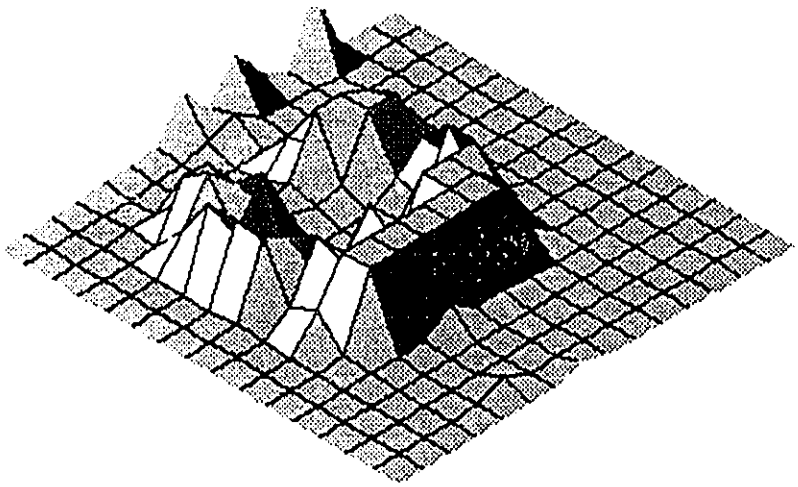


Figure 4.62 - example of "C"

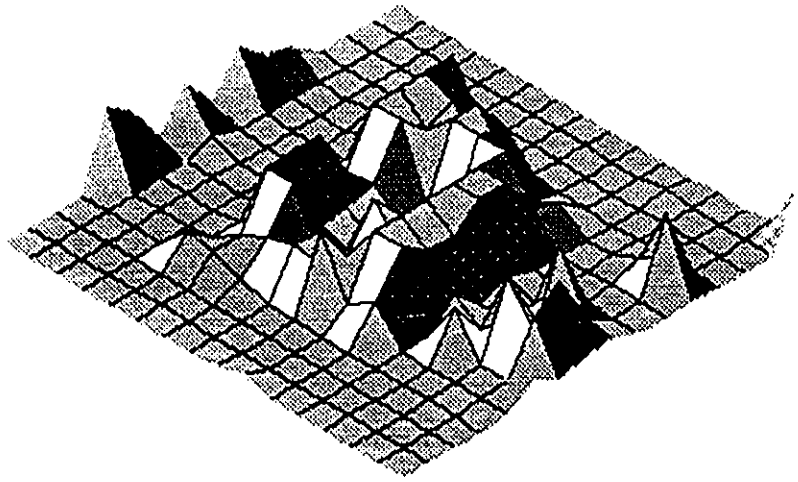


Figure 4.63 - example of "D"

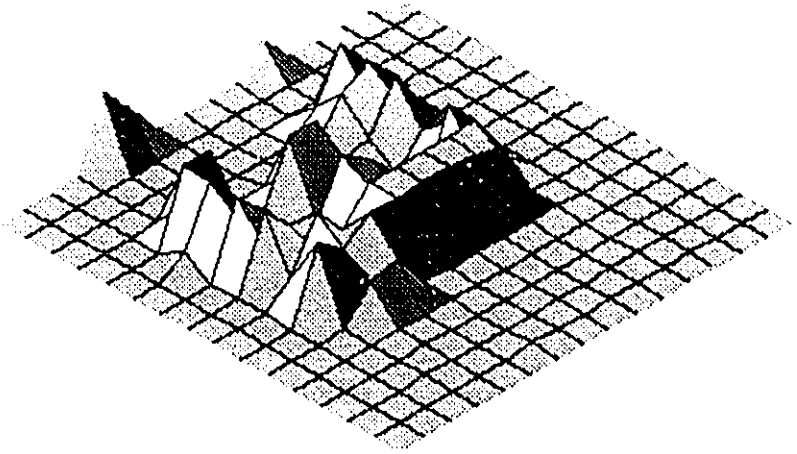


Figure 4.64 - example of "**E**"

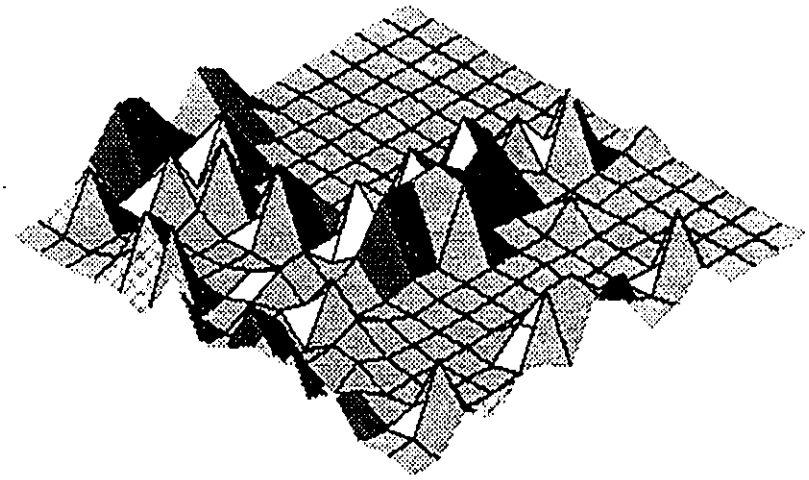


Figure 4.65 - example "**I**"

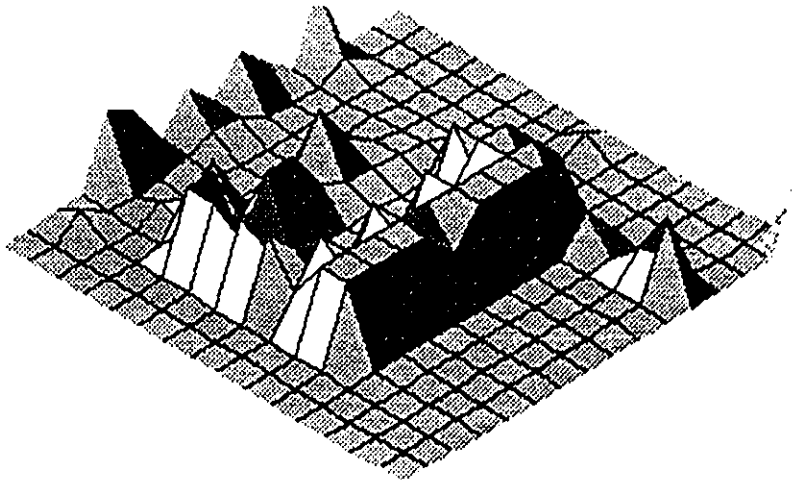


Figure 4.66 - example "**L**"

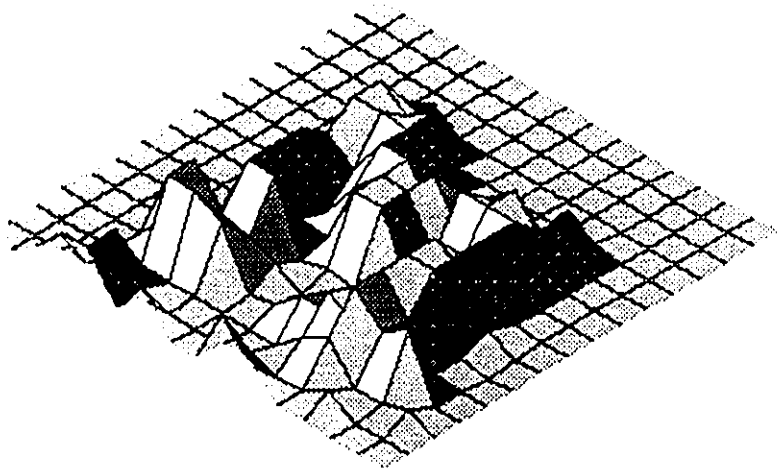


Figure 4.67 - example "**M**"

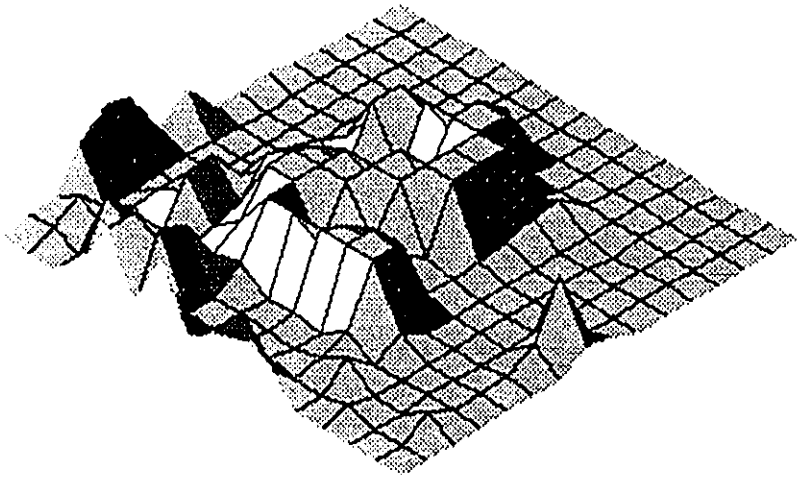


Figure 4.68 - example "**S**"

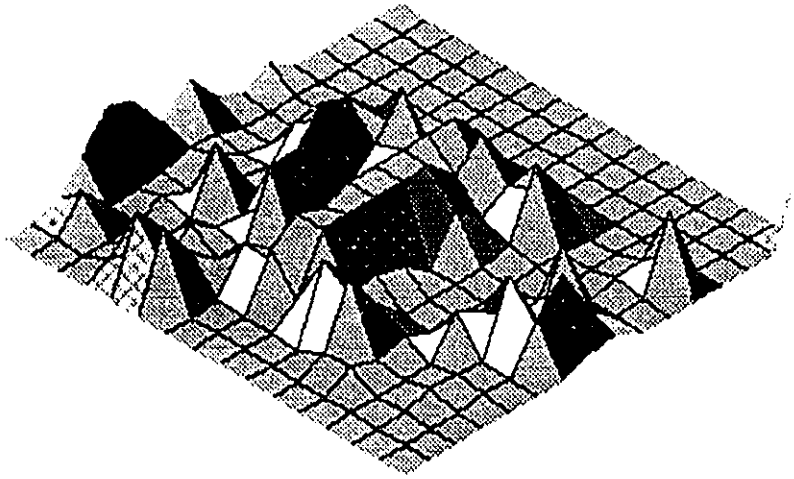


Figure 4.69 - example "**T**"

4.4.3 Network Development

A network capable of solving this problem, is developed by application of the concepts outlined in chapter 3. The number of inputs to the network is fixed to be equal to the number of force sensitive points on the tactile sensor (i.e. 16 x 16 = 256). The number of outputs is set to be equal to eight, a neuron to show each of the eight letter types. The number of neurons in the second layer is determined from the number of clusters that as a minimum is equal to the number of outputs or eight. The number of clusters is eight and therefore, the number of layer 2 neurons is eight and layer 1 neurons is 3 (from Equation 3.07). The initial network is built around these values and an initial training set is then selected.

4.4.4 Training Set Selection

A training set for the Small-Block network is selected by the same criteria as that for the Big-Block network. The abstractions for the Small-Block network are the same as those for the Big-Block network with the exception that rotations by multiples of 90 degrees are disallowed. Figure 4.45 depicts these abstractions where, patterns a, d, e and d do not apply to the Small-Blocks problem.

Patterns of types e,f,g and h should be included to train the network to learn the concept of positional independence. Beyond the idea of positional parameters, images i, j, k and l demonstrate the concept of missing information. To correctly define class boundaries it is required that incomplete and noisy patterns be present in the training set as they represent the true boundaries between classes. Patterns m, n, o and p display the final requirement for this network, the ability to recognize minor rotations from the primary pattern locations.

The initial training set was chosen to be equal to 32 images, four of each type of letter.

4.4.5 Building the Network

The procedure for achieving the initial convergence of the network is similar to that used for the One-Two and Big-Blocks networks as outlined in section 4.2 and 4.3. Following the initial development, images were added to the training set through the iterative procedure described in chapter 3 and the number of images grew to 256 by iteration 8. Figure 4.70 depicts the variations of neurons in each of the two hidden layers against the iterations of the training set.

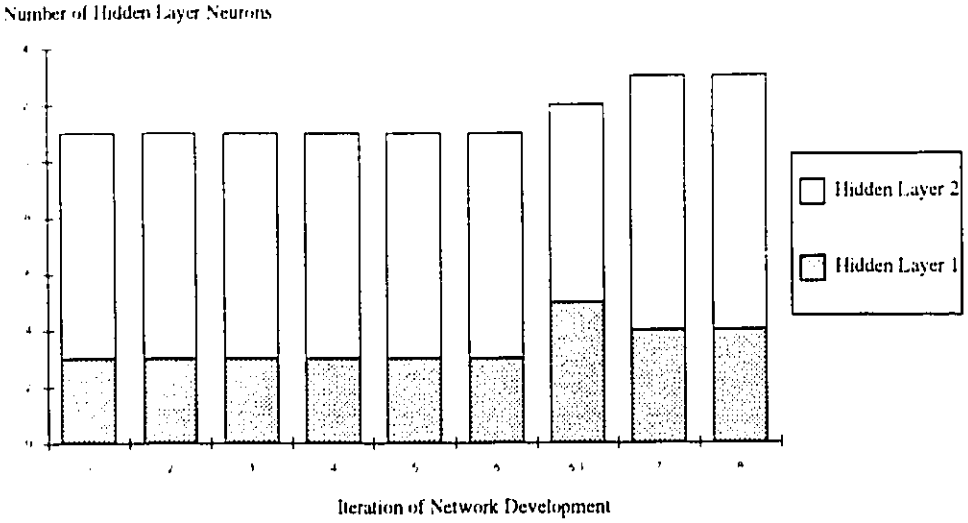


Figure 4.70 - Variations in number of neurons graphed against iterations of network

The criterion for determining the termination of the development phase was derived from application of a 128-image test set. If the test set showed that the network was no longer improving in recognition ability then no more iterations were performed. Figure 4.71 shows how many patterns were used to train each stage.

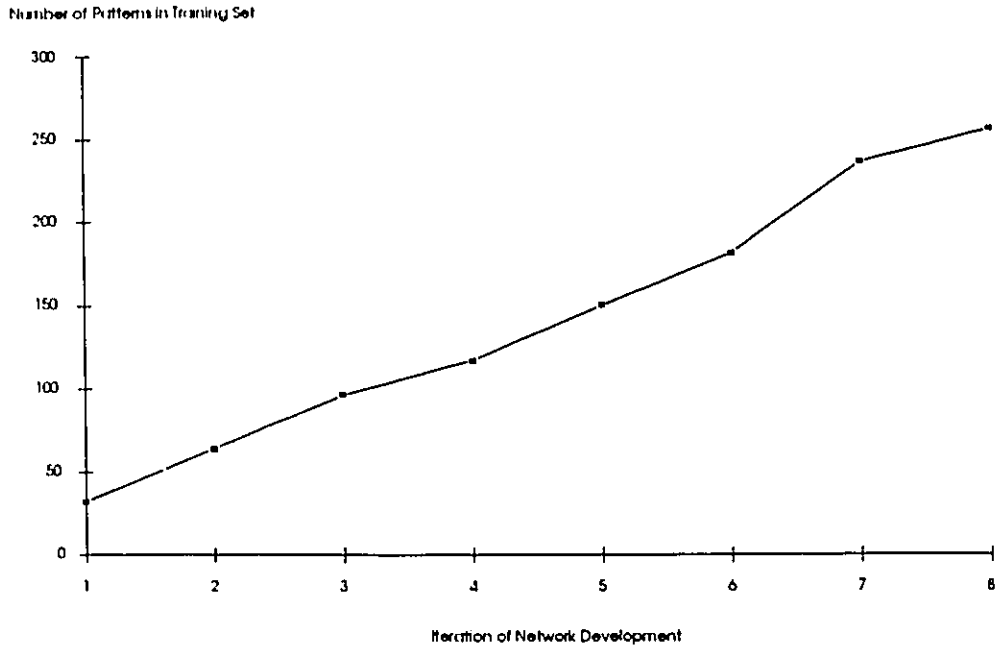


Figure 4.71 - Number of training patterns in iterations of the Small-Block net

4.4.6 Results

Figures 4.72 through 4.74 depict the results of the training regime when the iterations of the network are assessed using the one hundred and twenty-eight test images.

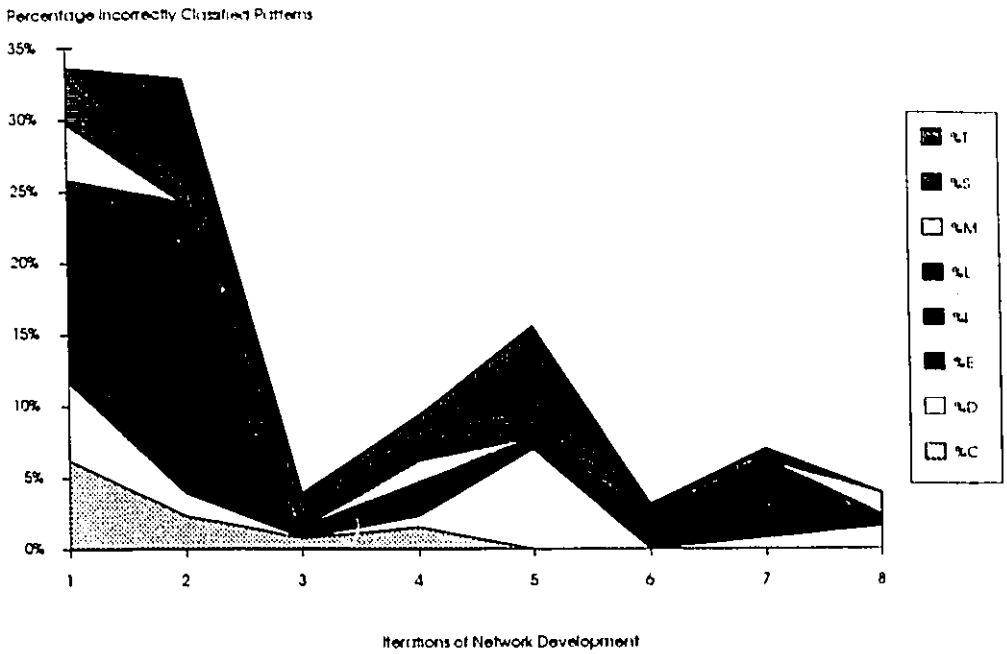


Figure 4.72 - Percentage of misclassified patterns against iterations of the training set

In Figure 4.72 the percentage of misclassified patterns (those classed as an other type) is graphed against the development iteration. The iterative training scheme successfully transformed the 34% misclassifications from the original training set to 4.0% misclasifications with the sixth iteration. This iteration achieved the best overall results and is therefore, used in all further general discussions of this network.

Figure 4.73 depicts the percentage of classifications where the network is unsure of the category for a particular image.

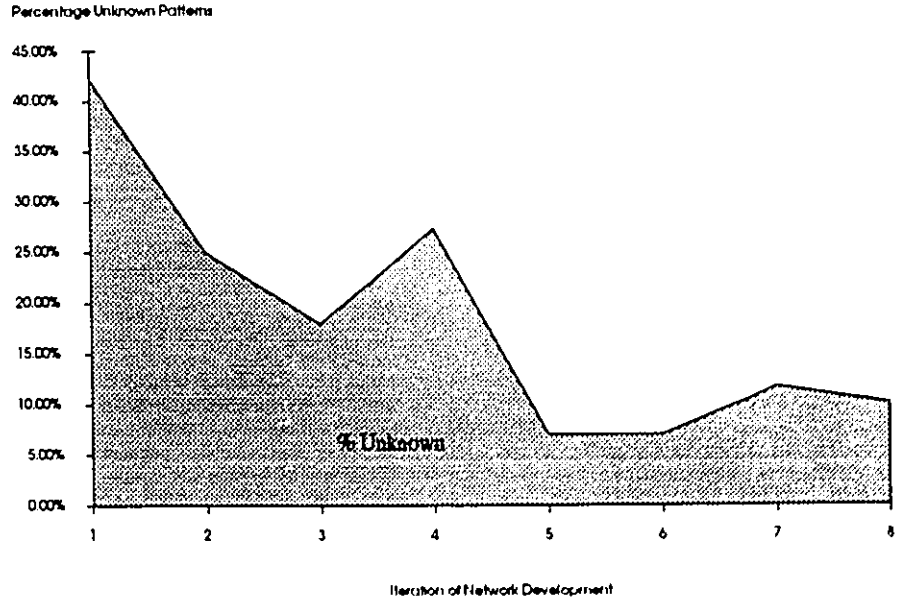


Figure 4.73 - Percentage of patterns of unknown classification against iterations of the training set

Figure 4.74 depicts the percent of correctly classified patterns against the iterations of the network at each stage of development.

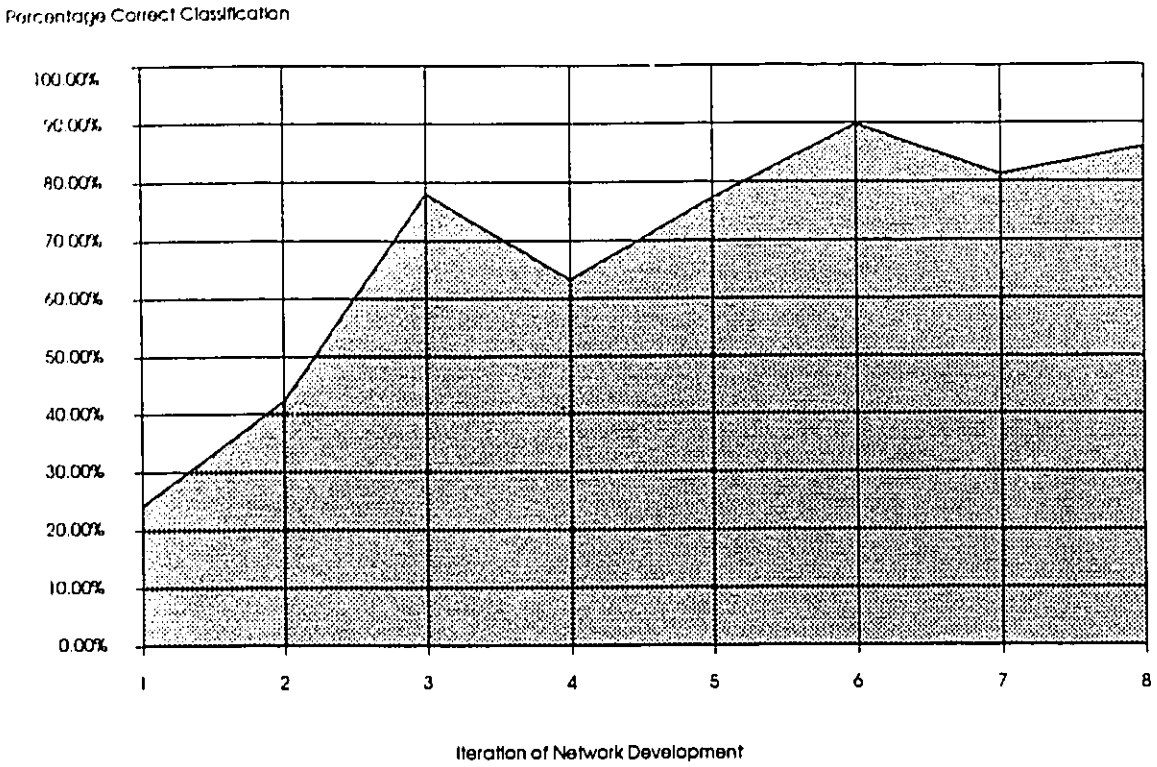


Figure 4.74 - Percent of patterns classified correctly graphed against the iterations of the training set

Although, the results for this network appear excellent, further random tests proved somewhat inconsistent for this particular network. This lack of consistency is perhaps attributable to the high distortion of the smaller images with noise. The affects of noise are more apparent for these images than with the large patterns used in the Big-Blocks problem, which produce clean tactile images. Sources of this noise are: Electromagnetic in the analog part of the interface; thermal, in the Force Sensitive Resistance's and cross talk, in the sensor.

4.4.7 Network States

Figures 4.75 through 4.82 depict the neuron states in the first and second network layers for iteration 6. All of the 90% correctly classified patterns in the test set fall into one of these state categories or are minor deviations from them.

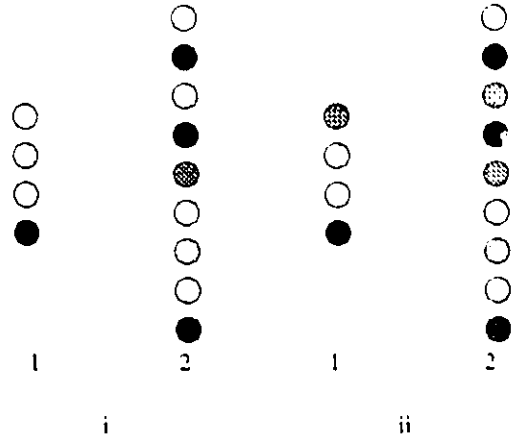


Figure 4.75 - Neuron states resulting in a classification of "C"

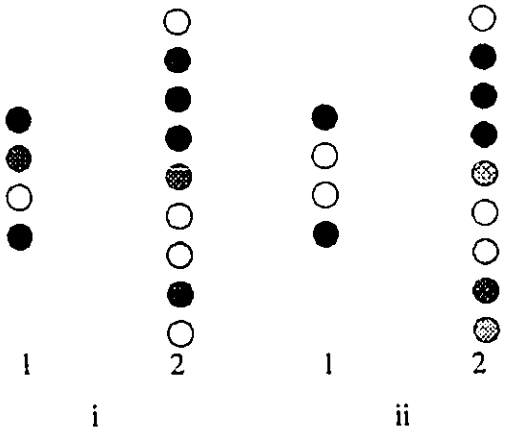


Figure 4.76 - Neuron states resulting in a classification of "D"

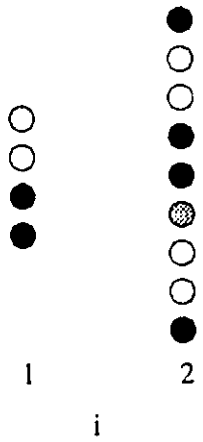


Figure 4.77 - Neuron states resulting in a classification of "E"

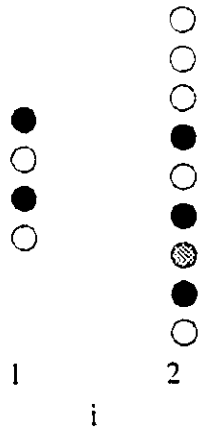


Figure 4.78 - Neuron states resulting in a classification of "**I**"

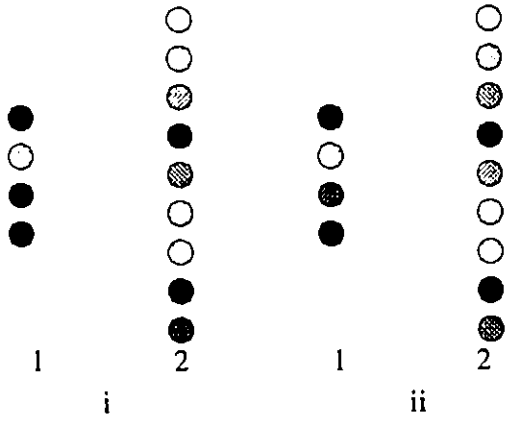


Figure 4.79 - Neuron states resulting in a classification of "**L**"

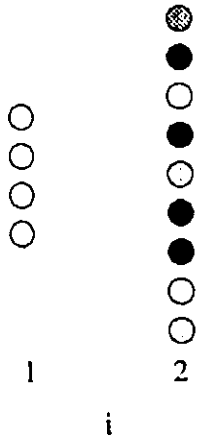


Figure 4.80 - Neuron states resulting in a classification of "**M**"

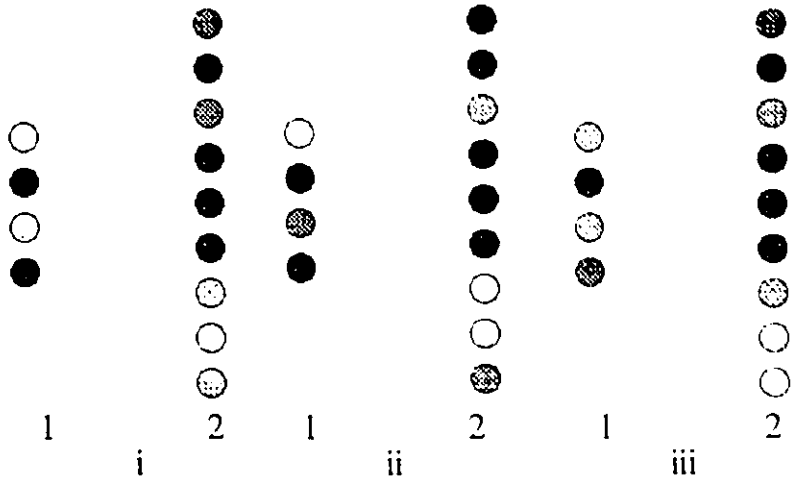


Figure 4.81 - Neuron states resulting in a classification of "S"

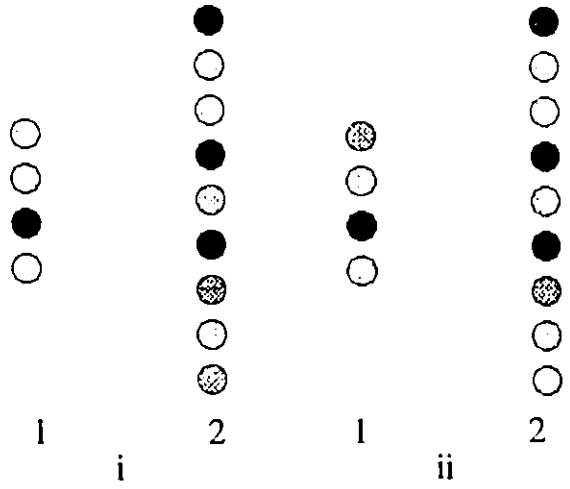


Figure 4.82 - Neuron states resulting in a classification of "T"

As shown by the Figures, most of the letters only excite a single state. Even in the case of "S" where three states are indicated, the similarity among the three is apparent. The probable cause of this simplicity is in the single orientation and scale of the input images. The network is required only to ignore random noise and position within the sensor surface.

4.4.8 Results Outside Problem Domain

The original intent of the network was to recognize the difference between eight letters with small features. Therefore, the only variables were random pressure, position of image and noise. One problem outside the domain of this network not previously explored is in the scaling of the letters. To this end the letter "T" common to both the large and small letter sets is used in the following tests. Twenty impressions of the large block were taken in the same orientation as that of the small block "T." All twenty impressions were input to iterations of the network 6, 7 and 8 with the accumulated results graphed in Figure 4.83.

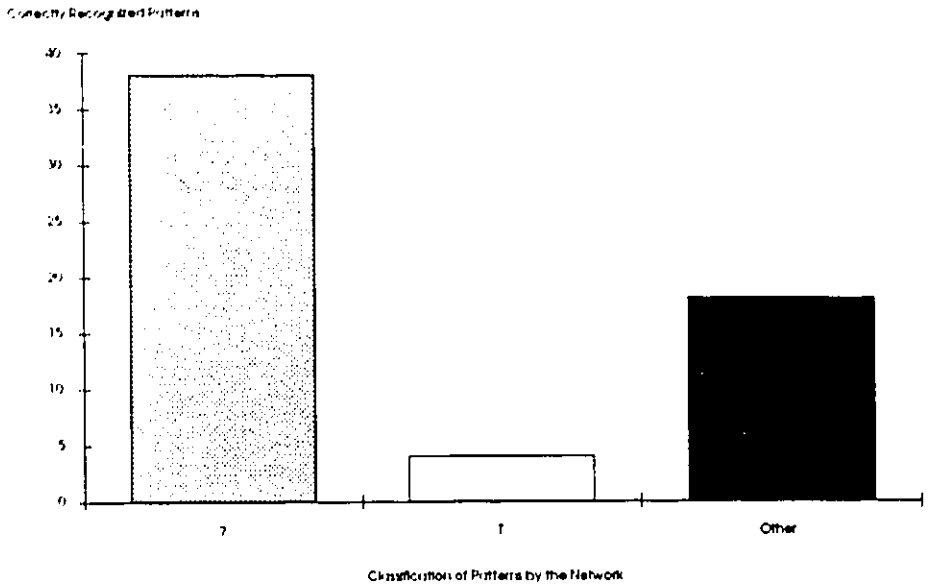


Figure 4.83 - Recognition of sixty patterns consisting of large "T"s

Apparently the large "T" primarily produced responses of "Unsure" from the network. In addition, of the twenty-four patterns classified as known only four were classified as T. Since, there are eight possible letters, statistically 24 / 8 or three should be classified as T assuming random selection with equal probability. Therefore, the recognition of four Ts is not considered statistically significant. From this example, apparently scaling is not a function inherent in the neural network paradigm but that this property must be learned from the appropriate examples.

4.4.9 Summary

This section has addressed the results obtained from the "Small-Block" net. A network developed that learns if any of the eight embossed letters **C, D, E, I, L, M, S** and **T** are impressing the tactile sensor. The most effective network was successfully capable of classifying 90.0% of all patterns within the problem domain, when evaluated against an independent test set of 128 patterns. However, the results for this network can be inconsistent. This is probably attributable to the noisy images produced with blocks of such small features.

4.5 Lines Net

4.5.1 Background

The Lines net was designed to test the neural networks ability to extract primitive features and orientations of those features. The following of object edges by automata is a problem encountered in many robotic systems. Frequently, this problem is managed using visual sensors and a tracking algorithm based on image processing techniques. In the case of the tactile sensor, image processing techniques can also be applied to the problem [Math93i]. However, these techniques can be either augmented by or replaced by, the neural network should this paradigm prove to have superior performance [Math93ii]. The primary ideas involved are in deciding the angle a straight edge impressed on the sensor surface, makes with respect to the sensor edge. An automata could then choose the next position of its manipulator based on this angle and the point of intercept of the straight edge with the sensor sides. This problem requires the use of two neural networks: one for the determination of the intercept point and one for the angle of the edge relative to the sides of the sensor. It is the former network that was generated and tested for this example, the one that provides determination of the angle, as it is deemed to represent the more stringent and difficult problem. Figure 4.84 depicts the idea of "angle" as represented by several subclasses of images where, the angle is relative to a single arbitrary axis of the sensor.

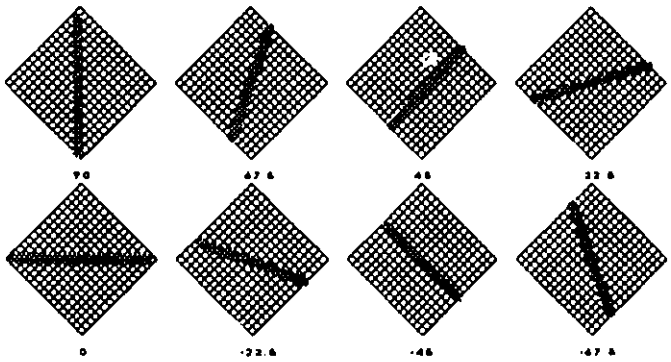


Figure 4.84 - Straight Edges in various orientations

4.5.2 Problem Statement

The "lines" problem is a problem of determining the angle of the captured image of a straight edge relative to an axis of the tactile sensor. The network is trained to recognize the orientation at one of eight possible angles. 90, 67.5, 45, 22.5, 0, -22.5, -45 and -67.5 degrees as depicted in Figure 4.84. This set is complete in the sense that all possible orientations of interval 22.5 degrees are included (assuming 0 is an angle included as the starting point). The noisy nature of the tactile patterns makes finding of the angles to finer resolutions than 22.5 degrees extremely difficult. However, this angular resolution is deemed sufficient for most practical problems where feedback to correct errors may be employed. A straight edge of width of 1 to 2 mm is used to gather datum for this problem. Figure 4.85 through 4.92 depict examples of the tactile images collected from the straight edge in question. These images are typical of those obtained for this problem domain.

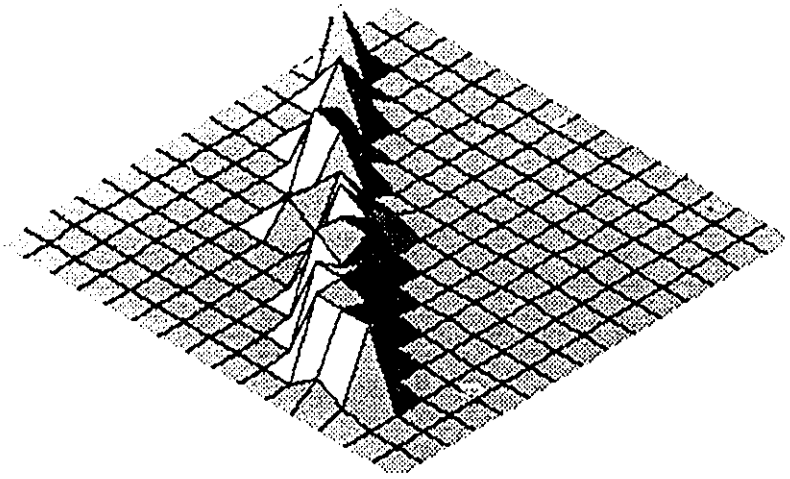


Figure 4.85 - example of "**90**"

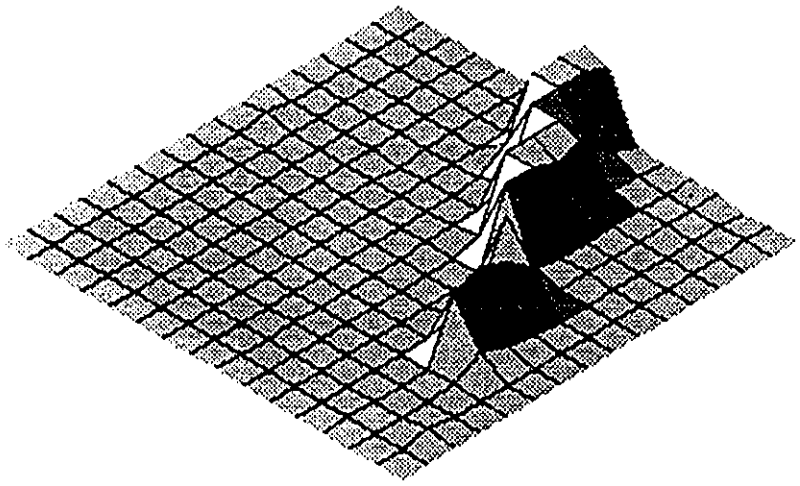


Figure 4.86 - example of "**67.5**"

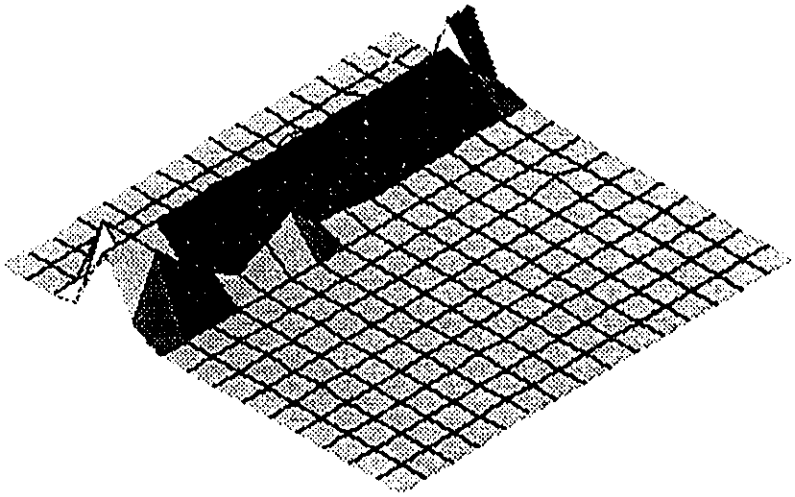


Figure 4.87 - example of "**45**"

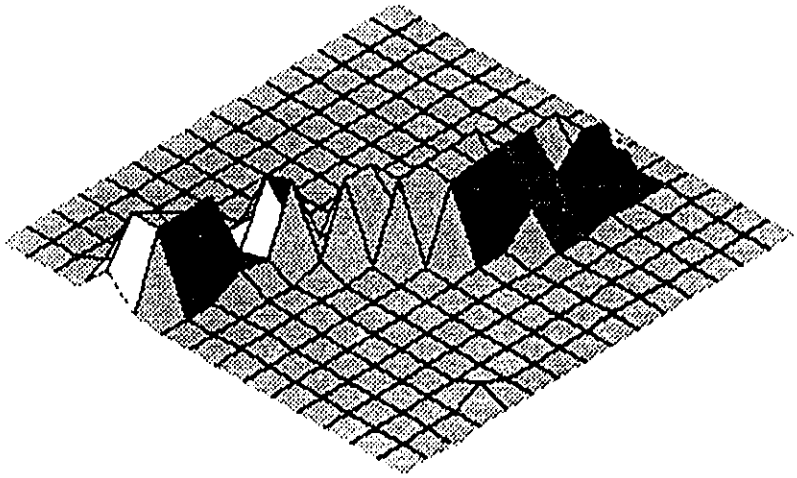


Figure 4.88 - example "**22.5**"

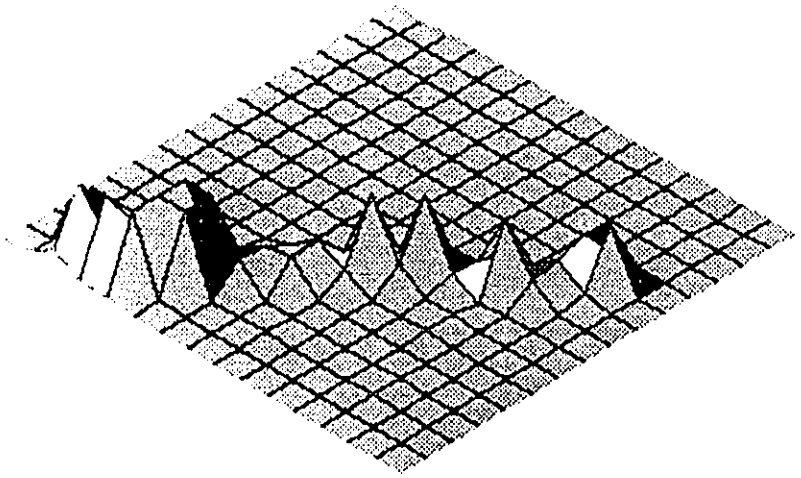


Figure 4.89 - example "**0**"

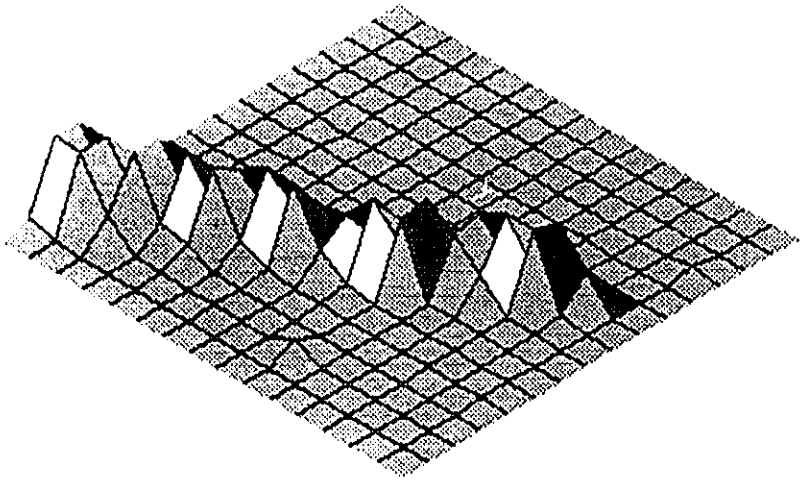


Figure 4.90 - example of "**-22.5**"

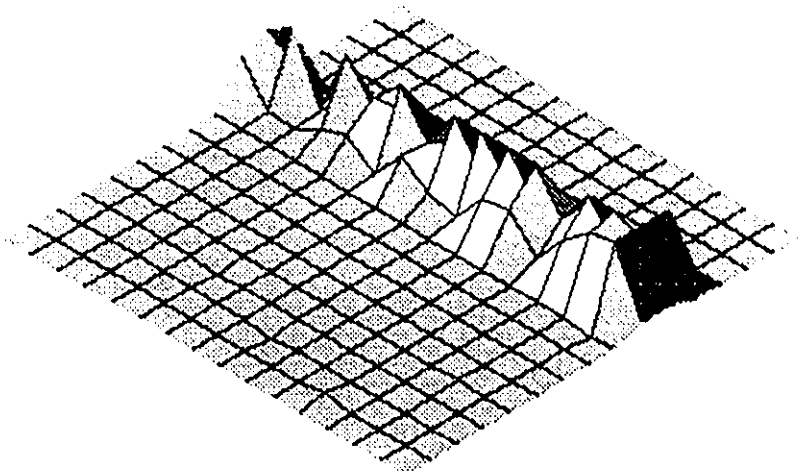


Figure 4.91 - example of "**-45**"

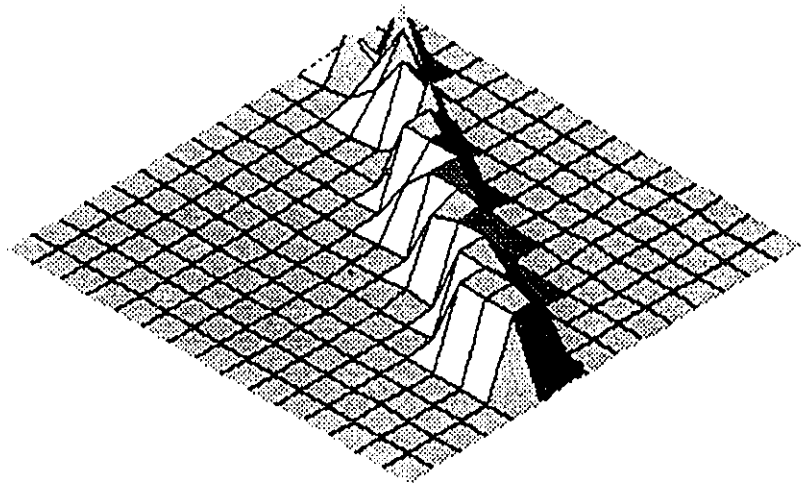


Figure 4.92 - example "**-67.5**"

4.5.3 Network Development

The network is developed by application of the ideas outlined in chapter 3. The number of inputs to the network is fixed to be equal to the number of points on the tactile sensor (i.e. 16 x 16 = 256). The number of outputs is set to be equal to nine: a neuron to indicate each of the orientations and an additional neuron trained to recognize examples of non-linearities (i.e. any image that does not represent a straight edge). This neuron was added in an effort to develop a network that could decide both if a straight edge was present in the tactile data or not. Its output of "None" is stimulated if no straight edge is present. The addition of this neuron proved to be an unsuccessful approach to the problem. Therefore, the addition of additional training examples for this neuron was abandoned in later iterations of the network.

The number of neurons in the second layer is found from the number of clusters that as a minimum is equal to the number of outputs or nine. The number of clusters is nine and

therefore, the number of layer 2 neurons is nine and layer 1 neurons is 3 (from equation 3.07). The initial network is built around these values and an initial training set is then selected.

4.5.4 Training Set Selection

As described in chapter 3, it is desirable to choose boundary samples to correctly define the classification regions. In addition several abstractions are required. Figure 4.93 depicts the concept of intercept independence, here a translation perpendicular to the axis of orientation is deemed to be classified into identical categories. In the figure several examples of orientation 90 and 0 degrees are illustrated.

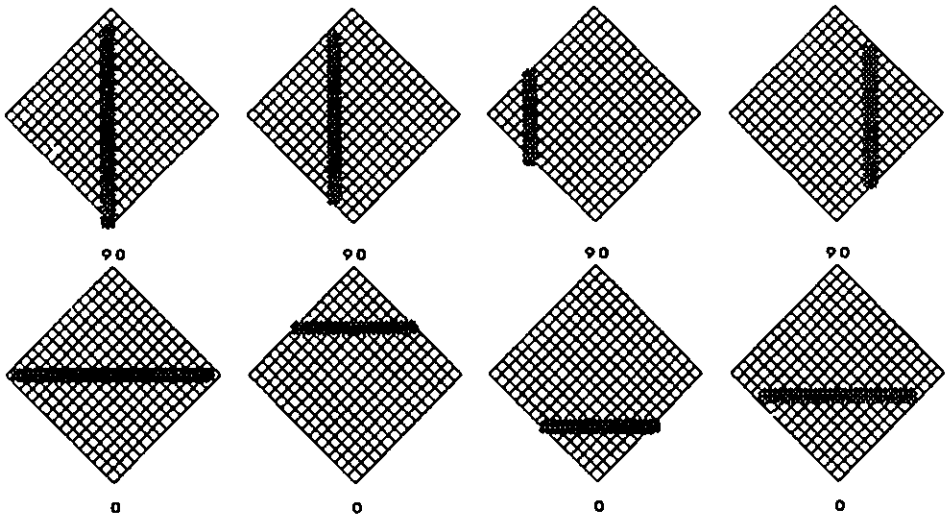


Figure 4.93 - Illustration of translation independence

Figure 4.94 depicts the second abstraction required of the network, that of missing and noisy information.

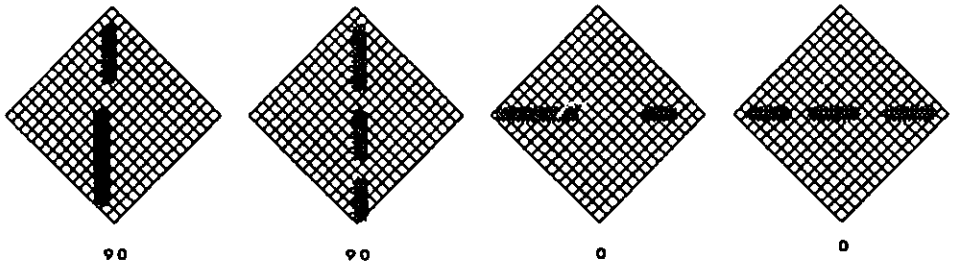


Figure 4.94 - Incomplete information in images

Examples of these abstract ideas are included within the images chosen for the initial training set. This initial training set was chosen to be equal to 30 images.

4.5.5 Building the Network

The procedure for achieving the initial convergence of the network is similar to that used for the One-Two net as outlined in section 4.2. Figure 4.95 depicts the variations of neurons in each of the two hidden layers against the iterations of the training set.

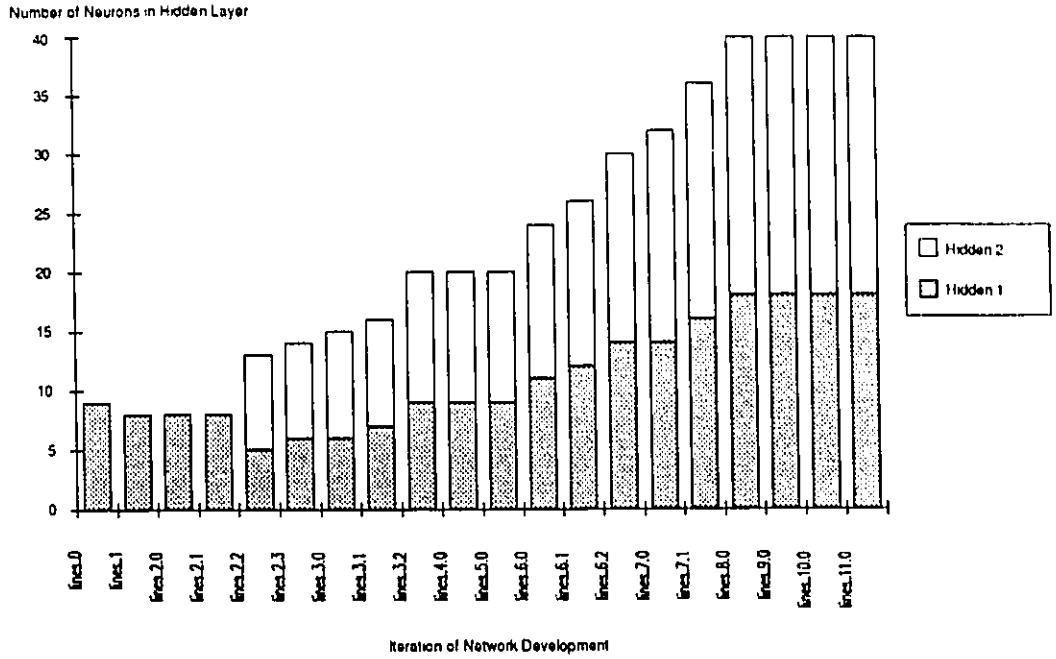


Figure 4.95 - Variations in number of neurons graphed against iterations of network development

The test set size for this network consisted of 100 images. Figure 4.96 shows how many patterns were used in training at each stage of network evolution. Development of this network proceeded similarly as that for previously described networks.

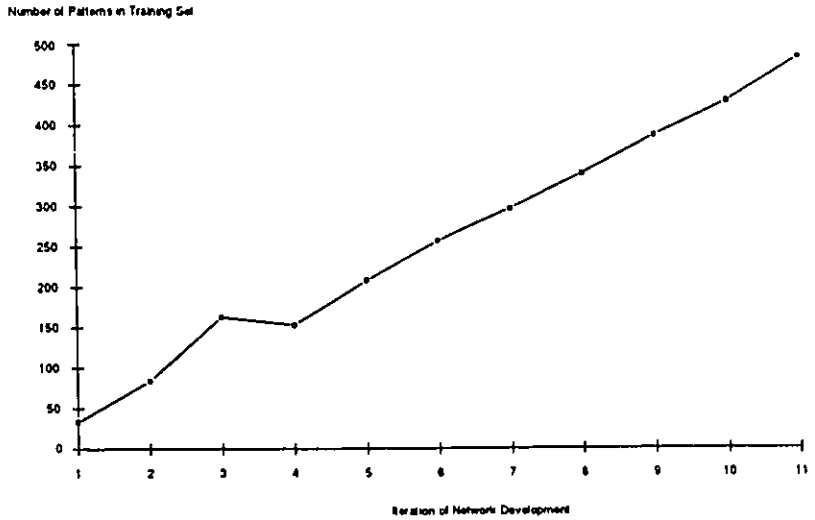


Figure 4.96 - Number of training patterns in iterations of the Lines net

4.5.6 Results

Figures 4.97 through 4.99 depict the results of the training regime when the iterations of the network are assessed using the one hundred test images. For this network a "Score" criterion was adopted for assessment purposes. The reasoning is as follows: the lines network is designed to detect the varying angles of intersection of straight edges relative to the tactile sensor. It is apparent that a deviation of one interval of classification (i.e. 22.5 degrees) is less of an error than greater intervals. Therefore, a rating system was developed by which no error in classification gained a score of 100%, an error of one interval (22.5 degrees) was considered to score 50% an error of two intervals 25%. More than two intervals gained scores reduced by half for every additional interval. Using this criterion, the results depicted in Figure 4.97 are derived.

Figure 4.97 - Network "Score" against iterations of the training set

The iterative training scheme successfully transformed the 15% "Score" from the original training set to 75% misclasifications with iteration 11.3 of the network, trained with the 11^{th} iteration of input data.

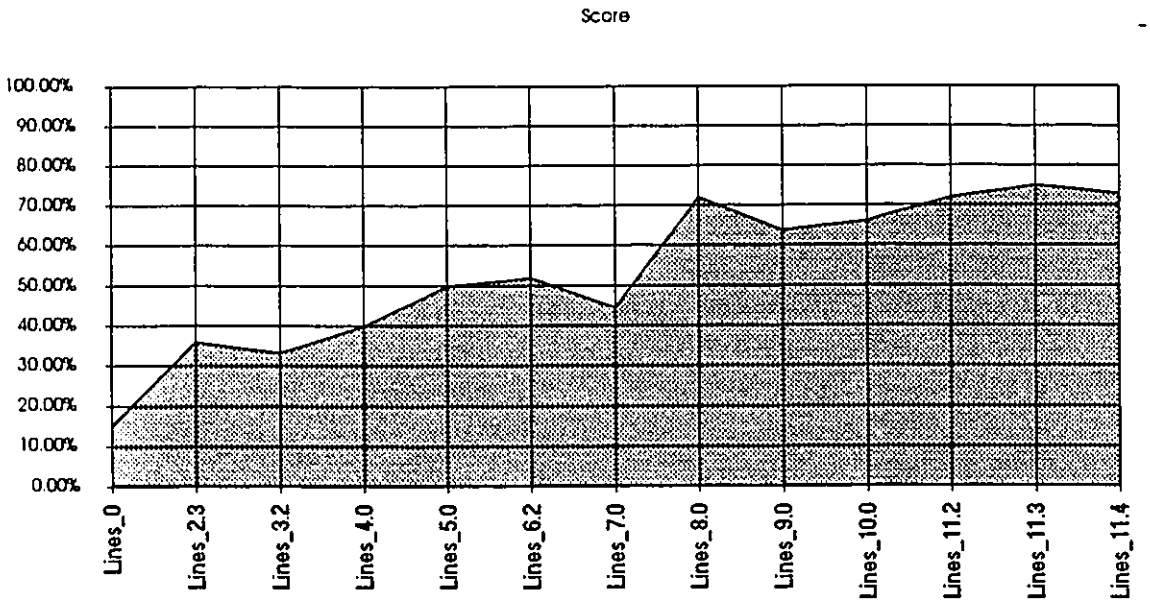


Figure 4.98 illustrates the variation is the percentage of unknown patterns versus iterations of the network.

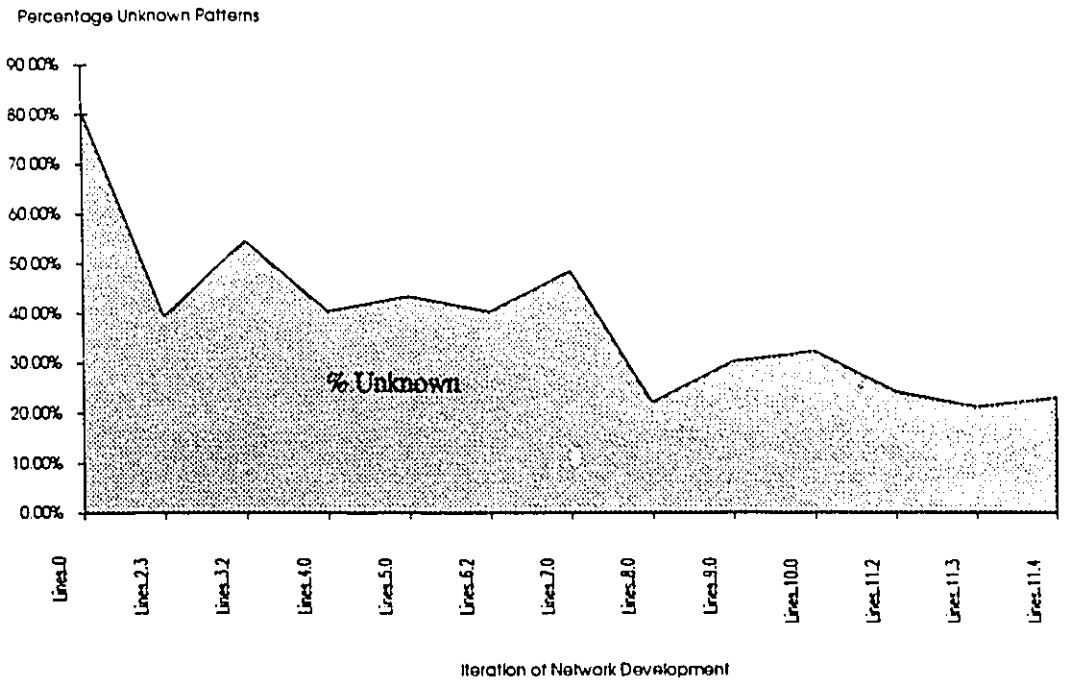


Figure 4.98 - Percentage of patterns of unknown classification against iterations of the training set

For this network it is enlightening to examine the variation in the number of patterns classified correctly, with an error of one angular interval and with an error of two or more angular intervals, as a function of the iteration of the network. Figure 4.99 depicts this relationship. As shown by the figure, the early iterations of the network tended to misclassify more often than later iterations. In general, only a small percentage of patterns are misclassified by an adjacent rotation and an even smaller percentage are misclassified by more than one.

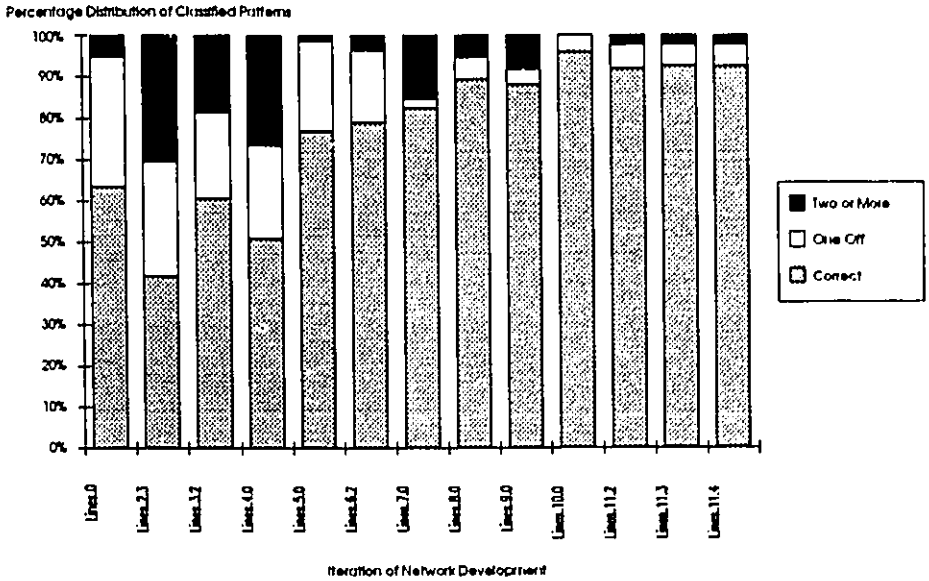


Figure 4.99 - Percentage of classifications that are correct, off by one or off by more than one.

It was decided that the reason for the high percentage of unknown patterns was intrinsic to the nature of this problem. In essence, the orientation problem is a continuous spectrum of alternatives rather than a discrete pattern classification problem. Therefore, it is possible that a line orientation between two of the standard orientations may be classified as either or both (i.e. an orientation of 15 degrees may be classed as both 0 and 22.5). Due to this fact, often two neurons become relatively active during the classification testing. The solution to this ambiguity is to "weaken" the aforementioned recognition criterion from 0.6 to 0.1. Results for the 11.3 network iteration were reevaluated against this new criteria and a new score for the network calculated using the standard test patterns. The overall score for the network increased from 75% to 85%, in addition no significant increase of misclassifications of more than one angular interval was observed. The comparative results of the experiment are graphed in Figure 4.100.

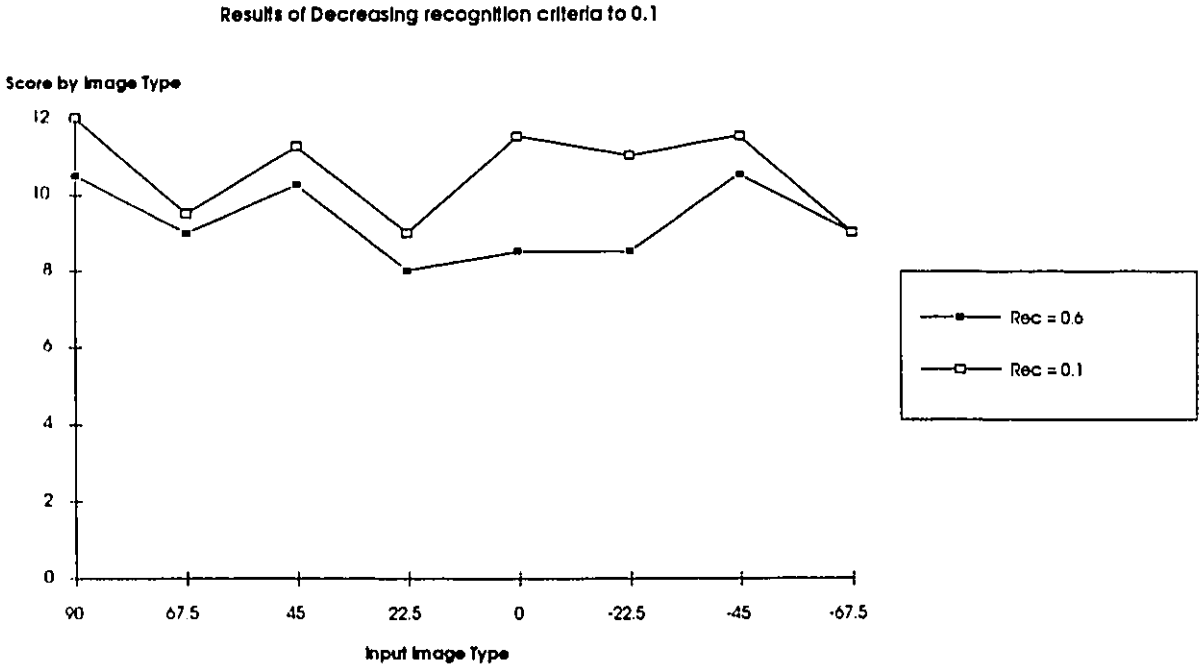


Figure 4.100 - Result of decreasing the convergence criteria to 0.1

4.5.7 Network States

Due to the large number of neurons in the final topology of the network the number of possible states for any given class was excessive. Therefore, no conclusions could be drawn from visual inspection of these states and they are not included in this section.

4.5.8 Results Outside Problem Domain

The original intent of the network was to recognize the difference between straight edges in varying rotations each 22.5 degrees apart as aligned with the sides of the tactile sensor. Therefore, the detection of the approximate angles of curved surfaces is outside the problem domain. The use of curved lines is considered *a propo* as the traditional pattern recognition and image classification approaches to the problem do not lend themselves

well to curved lines. Consequently two test sets were accumulated, the first consisted of lines taken from a curved edge with diameter 6 cm and the second from an edge of diameter 8 cm. In both cases, the angle with respect to the tactile sensor was assessed to be between the two points of intersection with the sensor periphery.
To test the capability of curved edge recognition, 40 patterns were accumulated for each of the 6 cm and 8 cm examples. This represented five patterns from each of the rotational possibilities. Figure 4.101 depicts an example from the set of 6 cm images whereas, Figure 4.102 depicts an example from the 8 cm set.

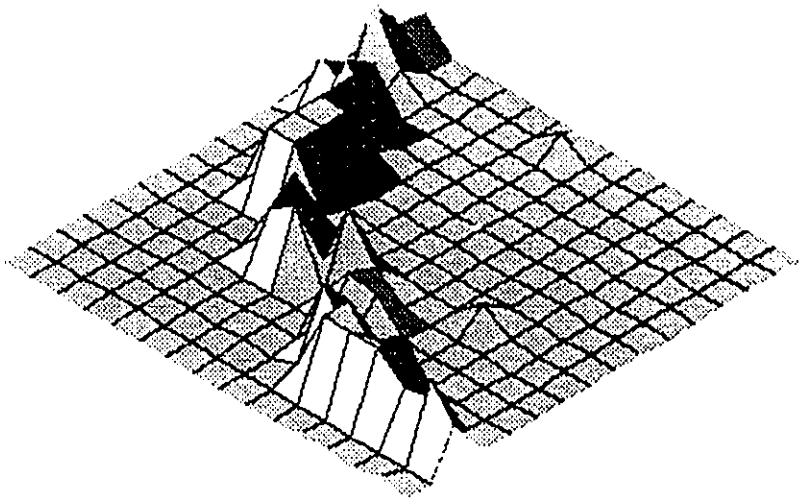


Figure 4.101 - Example of 6cm diameter line

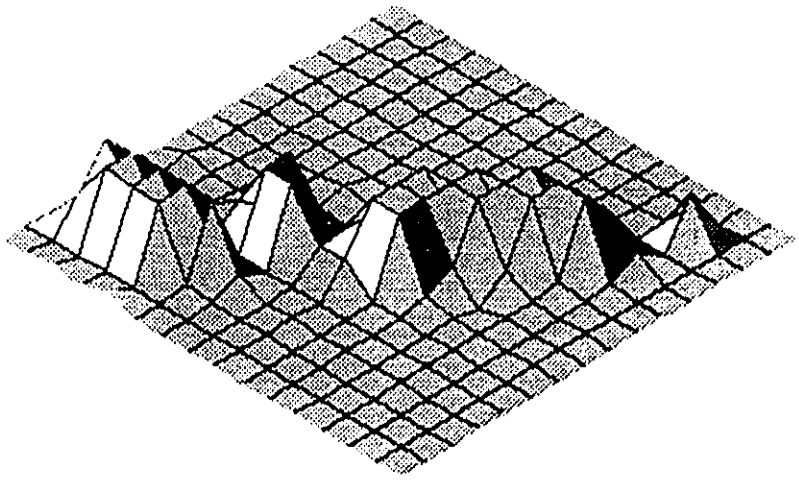


Figure 4.102 - Example of 8 cm diameter line

The test sets were assessed using network 11.3 and the standard recognition criteria of 0.6, the results are graphed in Figure 4.103 . From this graph it is apparent that the network has learned to classify angles in the general sense although no specific examples of curved lines were included within the training set. From these results apparently

development of a network capable of detecting the angular orientation of general curved lines should be possible.

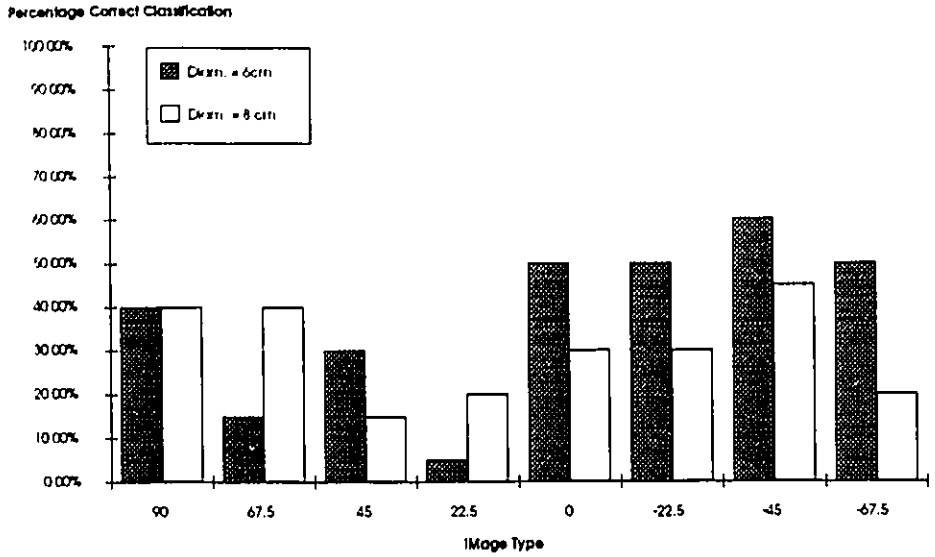


Figure 4.103 - Results of application of the "lines" network to curved lines

Beyond the score for the curved lines it is interesting that the distribution of classifications that are correct, off by one or off by two are different from that of the straight lines. For the curved lines, greater deviations from the nominal classifications are observed, as depicted by Figure 4.104. A possible explanation for this phenomenon is that the curved lines represent combinations of the trained images and therefore, the classifications made by the network are correlation's to multiple types.

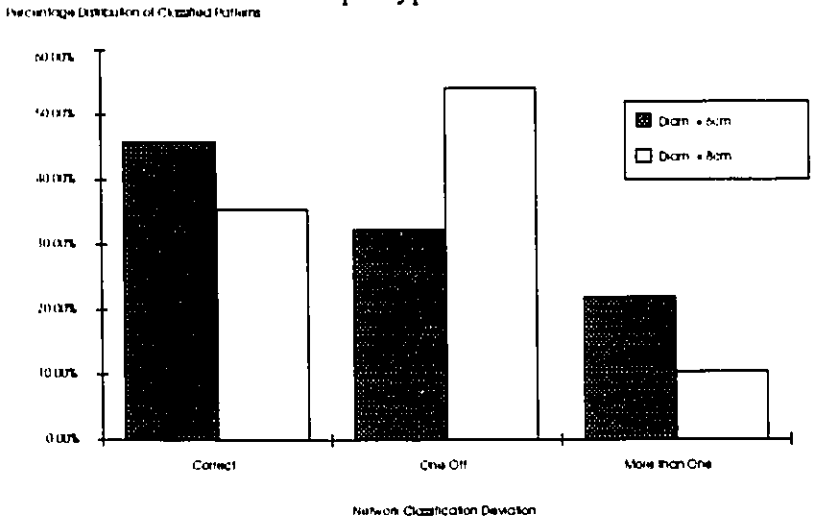


Figure 4.104 - Distribution of classified patterns for 6 cm and 8 cm curved lines.

4.5.9 Summary

This section has addressed the results gathered during the development of the "Lines" net. A network designed to find the relative orientation of straight edges captured by the tactile sensor. The most effective network was capable of scoring 75% using the standard recognition criteria of 0.6. However, when the recognition criterion was reduced to a value more appropriate for the problem, 0.1, a score of 85% was achieved.

Chapter 5
Conclusions and Recommendations for Further Work

5.1 Conclusions

The Backpropagation Neural Network has proven itself as a flexible and efficient paradigm for the classification of high resolution tactile images produced by a displacement sensitive tactile sensor. The flexibility of the solution is considered demonstrated by the exhibition that several fundamentally different pattern recognition problems can be solved utilizing the same software and hardware structures. In particular, the solutions differ only in parametric changes to the connections between neurons and the number of neurons in any given layer. The universality of the technique is further shown by using a universal training regime for all the generated networks. In addition, the use of an iterative approach to learning proved extremely successful. Networks were presented with an initial training set that was randomly selected from the set of all possible input patterns. After categorizing this initial set, additional images not correctly identified by the network were appended and the network retrained. After several iterations of this procedure the number of correctly identified patterns (against a previously unseen test set) increased dramatically. The results of this training regime as applied to the experimental networks of chapter 4, was that between 85% and 90% of the test patterns were correctly identified. From the excellence of these results, it is demonstrated that the neural network may be used as a pre-processor for a machine perception system, identifying features of both high and low complexity for use by a higher level of multi-sensor data fusion.

The results of chapter 4 clearly demonstrated the following advancements in the field of tactile sensor applications: pattern recognition of actual not simulated tactile data is possible through the use of neural networks and the applications of neural networks to nonlinear tactile data is not limited to the inverse elastic overlay problem but may be applied to the total pattern recognition problem. In addition, a learning regime for Back propagation neural networks involving an iterative pattern acquisition process proved successful in developing neural networks capable of classifying tactile images with a high degree of certainty.

5.2 Recommendations for Further Work

It has been suggested that the neural network be used as a low level feature / pattern recognition and classification engine. Several areas for future work are evident.

i) Increase the recognition rate of the network through use of multiple image acquisitions.

ii) The use of several neural nets in parallel to recognize more varieties of features.

iii) The use of neural networks as part of a larger pattern recognition engine.

iv) The use of ii) and iii) in conjunction with other machine sensors to produce an integrated approach to machine perception.

Although, the neural network has displayed good pattern classification abilities, occasionally patterns that intuitively should have been recognized were not. However, subtle variations of those patterns were correctly identified. To increase the probability of correct classification, tactile sensors connected to robot manipulators could gather many images in minor orientation and force variations. The various outputs of the network could then be assessed statistically to determine the overall classification of the current image.

The use of several neural networks operating in parallel extends the range of recognition ability, is an obvious corollary to the current work. Rather than attempt to design a network capable of classification of the entire gamut of features and patterns that may be required for a practical system, it is expedient to design several networks operating in parallel. This type of operational concept would reduce the total training time and produce more predictable results than the single large homogeneous network paradigm. The use of several networks in parallel requires the property of mutual exclusivity. By which, each network contains a neuron indicating that none of the type of features that it is designed to recognize are present. To achieve this goal, the following regime is suggested:

i) Determine the optimum networks for the specific sets of features as in chapter 4.

ii) Add a neuron to the output of each network indicating non-recognition.

iii) Add to the training images for each network the training images for all other networks and set the desired output for each of these images to "None"

iv) Retrain all the networks with the appended patterns.

Following this regime will produce networks with mutually exclusive outputs. However, it should be noted, that the final phase of retraining all the nets and testing the results may prove time consuming.

The second corollary to this work is the use of the network in a larger pattern recognition engine. The use of several networks as described in the previous paragraph may also be utilized in this context. Here a neural network or set of neural networks may be used to recognize features and patterns in the physical space constraints of the single tactile sensor. Several images displaced in space and/or time may be input to a machine for classification. An instance of this would be to use a syntactic approach for the recognition

of "words" embossed on surfaces. An extended version of the "Big Block" network described by chapter 4, may be used to recognize a single letter. The tactile sensor is then positioned on the next letter and that letter is in sequence identified. The letters could then be recognized using a syntactic approach to decide which letters exist beside which others and the number of spaces between words to allow recognition of individual words and phrases. This type of technique could be extended to more general cases. In addition the possibility for recognition of time varying sequence of images presents itself. This may be the case where a static manipulator is attempting to recognize images in an assembly line environment.

Bibliography

[Abid90] M.A. Abidi, R.C.Gonzalez, "The Use of Multisensor Data for Robotic Applications", IEEE Transactions on Robotics and Automation, vol. 6, No 2, April 1990

[Gonz87] R.C.Gonzalez, P.Wintz, "Digital Image Processing", Second Edition, Addison-Wesley, 1987

[Groo88] A.W. De Groot, "Effect of Sensor Size in Robotic Tactile Sensor Arrays", Robotica, Vol. 6, pp. 285-285, 1988

[Hebb49] D. Hebb, "Organization of Behavior", New York, John Wiley and Sons, 1949

[Hopp86] J.J. Hopfield, D.W. Tank, "Simple 'Neural' Optimization Networks:...",IEEE Transactions on Circuits and Systems, vol. CAS-33, No 5, May 1986

[Huan91] S.-C.Huang, Y.-F.Huang, "Bounds on the Number of Hidden Neurons in Multilayer Perceptrons, IEEE Transactions on Neural Networks, Vol. 2, No. 1, January 1991

[Inte89] Anonymous, "Interlink Electronics Tactile Array Robotics", Interlink Electronics, Santa Barbara, CA., 1989

[Khan90] T. Khanna, "Foundations of Neural Networks", Addison-Wesley, Title QP363.3.K43, 1990

[Kino75] G-I,Kinoshita, S. Aida, M. Mori, "A Pattern Classification by Dynamic Tactile Sense Information Processing", *Pattern Recognition*, Pergamon Press, Vol. 7, pp. 243-251, 1975

[Koho84] T. Kohonen, "Self-Organization and Associative Memory", Springer-Verlag, 1984

[Larc83] M.H.E. Larcombe, "Tactile Sensing", Robotic Technology, Short Run Press, London 1983

[Math91] W.S. McMath, S.K.Yeung, E.Petriu, N.Trif, "Tactile Sensor for Geometric Profile Perception", IECON'91, CH2976-9/91/0000-0893, 1991

[Math93i] W.S.McMath, S.K.Yeung, E.M.Petriu, D.C.Petriu, M.D. Colven, "High-Sampling Resolution Tactile Sensor", Pending Publication, 1993

[Math93ii] W.S. McMath, M.D. Colven, S.K. Yeung, E.M. Petriu, "Tactile Pattern Recognition using Neural Networks", Pending Publication, 1993

[Mehr91] K.G.Mehrotra, C.K.Mohan, S.Ranka, "Bounds on the Number of Samples Needed for Neural Learning", IEEE Transactions on Neural Networks, Vol. 2, No. 6, Nov. 1991

[Meye91] D.G. Meyer, "Some Tradeoffs for Interfaces to Piezoresistive Sensor Arrays", IEEE Transactions on Robotics and Automation, Vol. 7, No. 1, February 1991

[Mill90] W.T. Miller, R.P.Hewes, F.H.Glanz, L.G.Kraft, "Real-Time Dynamic Control ... Using a Neural-Network-Based Learning Controller, IEEE Transactions of Robots and Automation, Vol. 6, No. 1, February 1990

[Mins69] M. Minsky, S. Papert, "Perceptrons", Cambridge MA, MIT Press, Library of Congress 69-14379, 1969

[Pati89] Y.C.Pati, D.Friedman, P.S.Krishnaprasad, C.T.Yao, M.C.Peckerar, R.Yang, C.R.K.Marrian, "Neural Networks for Tactile Perception", *Naval Research Laboratory*, Code 6804, Washington D.C.,20375

[Pati92] Y.C. Pati, P.S. Krishnaprasad, M.C. Peckerar, "An Anolog Neural Network Solution to the Inverse Problem of 'Early Taction'", IEEE Transactions on Robotics and Automation, Vol. 8, No. 2, April 1992

[Pet92i] E.Petriu, W.S.McMath, S.K.Yeung, N.Triff, "Active Tactile Perception of Object Surface Geometric Profiles," IEEE Transactions on Instrumentation and Measurement, Vol. 41, No. 1, 1992

[Pet92ii] E. Petriu, D.M.Colven, N. Trif, "Development of Force Control Tactile Sensor applicable to Special Purpose Dextrous Manipulator (SPDM)", Contract 9F011-1-0925/00, April 1991-March 1992

[Rieg86] R. Riegart, Interlink Electronics, "The Force Sensing Resistor - A New Tool in Sensor Technology", Proceedings Sensor Expo, Sept 17-19, 1986, Vol. I

[Ross57] F. Rossenblatt, "The perceptron: A perceiving and recognizing automation (project PARA)", Cornell Aeronautical Laboratory Report, VG-1196-G-1, 1957

[Roth91] M.W.Roth, "Editorial:Anological Versus Logical or Connectionist Versus Symbolic or Scruffy Versus Neat", IEEE Transactions on Neural Networks, Vol. 2, No. 6, Nov. 1991

[Shin77] Shin-Yee Lu, King-Sun Fu, "Stochastic Error-Correcting Syntax Analysis for Recognition of Noisy Patterns", IEEE Transactions on Computers, Vol. C-26, No. 12, December 1977

[Simp90] P.K. Simpson, "Artificial Neural Systems, Foundations, Paradigms, Applications and Implementations", Pergamon Press, 1990

[Tour89] D.S. Touretzky, D.A. Pomerleau, "What's Hidden in the Hidden Layers?", BYTE magazine, August 1989

[Trei86] A. Treisman, "Features and Objects in Visual Processing", *Sceintific American*, Vol. 255, No. 5, Nov. 1986

[Trou91] T. Troudet, W.Merrill, "Neuromorthic Learning of Continuous-Valued Mappings from Noise-Corrupted Data", IEEE Transactions on Neural Networks, Vol. 2, No. 2, March 1991

[Vemu88] V. Vemuri, "Artificial Neural Networks: Theoretical Concepts", The Computer Society of the IEEE, Library of Congress no. 88-71053, 1988

[Widr92] B. Widrow, M.A. Lehr, "Feedforward Networks", Stanford U. Dept. Electrical Engineering, Stanford, CA 94305-4055, December 1992

[Yang85] R.Q. Yang, M.W. Ziegal, "A Finger-Tip Optical Sensor Array Sorting System", Proc. Sensors 1985 Conf. CASA/SME Technical Paper MS85-991, pp. MS85-991-1 / MS85-991-11, 1985

Appendix A

This Appendix supplies a partial listing for the program "Network Developer". A full listing would occupy numerous pages and therefore, for the sake of brevity is not included. The eliminated code fragments are primarily "Windows" specific and do not add to the overall program comprehension. The code specific to the neuron learning and recall is included in its entirety. This program allows a user to: design, display, store, input patterns for learning, change learning parameters, initiate learning and provides interactive tools for the development of Backpropagation Neural Networks.

NEURAL.H (Partial Listing)

```
// This file containes the neural.H header infomation for network developer. This version
// has a large protion of the code eliminated for brevity. The missing pieces are indicated by ++++++++++++++++

// Function Prototypes that manage windows
++++++++++++++++++++++++++++++
// Function Prototypes that initialize windows
++++++++++++++++++++++++++++++
// Functions that manage the network
++++++++++++++++++++++++++++++
// Functions that manage patterns.
++++++++++++++++++++++++++++++
// Utility Functions for windows maintenance
++++++++++++++++++++++++++++++
// Defines for windows return constants
++++++++++++++++++++++++++++++
// Defines for Colours
++++++++++++++++++++++++++++++
// User Message Types
++++++++++++++++++++++++++++++
// Child Window Defines
++++++++++++++++++++++++++++++
// Functions that are used while learning
float sigmoid ( float  );
float del_sigmoid ( float  );
void calculate_output ( int, int );
BOOL calculate_output_errors ( int, int );
void backpropagate_errors ( int );
void modify_weights ( int, int );
unsigned int learn ( );
void ModifyColours ( int );
int CalculatePercentCorrect ();
++++++++++++++++++++++++++++++
//          This section contains classes for the network.
++++++++++++++++++++++++++++++
// Typedefs for neuron and layer management
typedef struct {
int NumberIterations;
int PercentageCorrect;
char far *DesiredPattern;
char far *ActualPattern;
} Child_Param_Type;

typedef struct {
float weight [MAX_WEIGHTS+1];
float delta [MAX_WEIGHTS+1];
float x1 [MAX_WEIGHTS];
float x2 [MAX_WEIGHTS];
float y1 [MAX_WEIGHTS];
float y2 [MAX_WEIGHTS];
int k[MAX_WEIGHTS];
BOOL visible;
BOOL selected;
```

```
float sum;
float output;
float error;
} neuron_type ;

typedef struct {
int neurons;
int inputs;
BOOL connect;
HANDLE hNeuron [MAX_NEURONS];
BOOL DisplayedAsStatus [MAX_NEURONS];
int xoff;
int yoff;
int x[MAX_NEURONS];              // x and y are the neurons screen position
int y[MAX_NEURONS];
long colour[MAX_NEURONS];

} hidden_type;

typedef struct {
int number;
int status;
} pattern_status_type;

typedef struct {
float value[MAX_NEURONS];
char PatternType[LENGTH_OF_STRING];
} output_pattern_type;

typedef struct {
char Type[LENGTH_OF_STRING];
} pattern_string;

typedef struct {
int number;
int Dimensions;
HANDLE hPattern[MAX_PATTERNS];
BOOL visible;
int xoff;
int yoff;
int x[MAX_WEIGHTS];
int y[MAX_WEIGHTS];
long colour[MAX_WEIGHTS];
} input_type;

typedef struct {
int number;
int SecondDim;
BOOL connect;
HANDLE hPattern[MAX_PATTERNS];
BOOL visible;
BOOL visdiff;
int xoff;
int yoff;
int x[MAX_NEURONS];
int y[MAX_NEURONS];
pattern_string Pattern[MAX_NEURONS];
long colour[MAX_NEURONS];
long coldiff[MAX_NEURONS];
} output_type;

// Functions that draw in windows
++++++++++++++++++++++++++++++
// Windows variables

++++++++++++++++++++++++++++++
```

```
// Global Network Data

BOOL net_converged, edit_mode, net_initialized, Monochrome;
BOOL learn_mode, show_mode, Looping, Paint_Loop;

int num_layers, num_patterns, NextPaintPattern;
hidden_type network[MAX_LAYERS];
input_type input;
output_type output;
++++++++++++++++++++++++++++
```

NEURAL.CPP (Partial Listing)

```
// This File contains the source code for the windows application
// Network Developer. This application works with the Microsoft
// Windows operating system to allow a user to build and train a
// Neural Network using the back propagation algorithm.
// This version of the File represents only the core program routines,
// the majority of the User Interface has been stripped out to facilitate
// a compact listing. Missing Code Fragments are indicated by ++++++++++++++++++++++++

int PASCAL WinMain( HANDLE hInstance,        // current program instance
                    HANDLE hPrevInstance, // previous program instance
                    LPSTR lpCmdLine,     // command line string
                    int nCmdShow )       // icon or window appearance
{
++++++++++++++++++++++++++++

 // Call window registration routine
++++++++++++++++++++++++++++
 // Create new instance of a window
++++++++++++++++++++++++++++
 // Message loop where Windows sends messages for input to Main Window

 //          Set up a timer for use during Learn mode so that the learning
 //          will restart if the user does not access the keyboard or the mouse.
++++++++++++++++++++++++++++
}

// The next procedure is used to perform the primary message processing for
//          all windows functions. Commands from the menu and other messages are
//          all passed from the main function to this procedure where they are acted
//          upon.

long FAR PASCAL _export WindowProcedure(HWND hwnd,unsigned msg,WORD wParam,LONG lParam)

{
++++++++++++++++++++++++++++
//          This routine maintains the main window looking after messages and
//          processing them as appropriate
            switch(msg)
            {
// If the window is first created then some initialization must occur
            case WM_CREATE:
++++++++++++++++++++++++++++
              }
              break;

//          The WM_CLOSE message is used to indicate that the window is about
//            to close and therefore, it is time to ask if the user wishes to
//            save the final changes to the network.

            case WM_CLOSE:
              {
++++++++++++++++++++++++++++
              }
              break;
++++++++++++++++++++++++++++
```

```
//          The WM_COMMAND message is from the main menu. It passes the users
//          selections onto the appropriate routines for action.
            case WM_COMMAND:
              {
                switch (wParam)
                {
//          If the user chooses OPEN then a dialog box appears to choose
//          the file to open.
                        case OPEN:
                        {
++++++++++++++++++++++++++++++
                        }
                        break;
//          If the user chooses NEW then a dialog box appears to prompt
//          the user to save the network if information exists.
                        case NEW:
                        {
++++++++++++++++++++++++++++++
                        }
                        break;
//          If the user chooses Save As then a dialog box appears to choose
//          the file to save.
                        case SAVEAS:
                        {
++++++++++++++++++++++++++++++
                        }
                        break;
//          If the user chooses Save then a dialog box appears to choose
//          the file to save if there is not an already valid file name.
                        case SAVE:
                        {
++++++++++++++++++++++++++++++
                        }
                        break;
//          This function removes all the patterns currenlty active and thereby
//          allows the user to change the sizes of the input and output layers.
                        case REMOVE:
                        {
++++++++++++++++++++++++++++++
                        }
                        break;
//          If the user chooses Input Pattern then a dialog box appears to choose
//          the file to open for learning.
                        case PATTERN_INPUT:
                        {
++++++++++++++++++++++++++++++
                        }
                        break;
//          If the user chooses RESIZE then a dialog box appears to choose
//          the absolute scale for the network
                        case RESIZE:
                        {
++++++++++++++++++++++++++++++
                        }
                        break;

// The user can choose to zoom in or out, the scale is changed

                        case ZOOM_IN:
                        {
++++++++++++++++++++++++++++++
                        }
                        break;
                        case ZOOM_OUT:
                        {
++++++++++++++++++++++++++++++
                        }
                        break;
//          The user can choose to exit the application. This has the same
//          effect as closing the system menu
```

```
                    case EXIT:
                    {
++++++++++++++++++++++++++++++
                    }
                    break;
//          The user can choose to copy the network to the clipboard.
                    case COPY_NETWORK:
                    {
++++++++++++++++++++++++++++++
                    }
                    break;
//          The user can choose which items are to be visible on the screen
//            the following sets of routines aid in the development of this
//            interface
//          If the use chooses to view the weights then a dialog box appears asking
//            for parameters on how they should be viewed.
                    case VIEW_WEIGHTS:
                    {
++++++++++++++++++++++++++++++
                    }
                    break;
//          If the use chooses to view the Input Layer then a check mark is shown
//          to show that the input layer is visible. If it is already visible then
//          the check mark is removed and the input layer is made invisible.
                    case VIEW_INPUT:
                    {
++++++++++++++++++++++++++++++
                    }
                    break;
//          If the user chooses to view the Output Layer then a check mark is shown
//          to show that the output layer is visible. If it is already visible then
//          the check mark is removed and the output layer is made invisible.
                    case VIEW_OUTPUT:
                    {
++++++++++++++++++++++++++++++
                    }
                    break;
//          If the user chooses to view the results on a mochrome display
//          then this method allows that option to be chosen.
                    case VIEW_MONOCHROME:
                    {
++++++++++++++++++++++++++++++
                    }
                    break;
//          If the user chooses to view a selected Neuron or group of neurons
//          then the following methods provide that ability.
                    case VIEW_NEURON_VISIBLE:
                    {
++++++++++++++++++++++++++++++
                    }
                    break;
                    case VIEW_NEURON_INVISIBLE:
                    {
++++++++++++++++++++++++++++++
                    }
                    break;
// If the user chooses to inialize the weights then all the weights
// are initialized to random values between -1 and +1
                    case INITIALIZE:
                    {
++++++++++++++++++++++++++++++
                        initialize_network ();
                        NumberIterations = 0;
                        PercentageCorrect = 0;
                        net_initialized = TRUE;
                        SetThresholds ();
++++++++++++++++++++++++++++++
                    }
                    break;
```

```
//          If the user chooses to alter the learning parameters then a dialog box
//          appears to allow the change.
                    case PARAMETERS:
                    {
++++++++++++++++++++++++++++
                        Parameters.LearningRate = LearningRate;
                        Parameters.WeightMoment = WeightMoment;
                        Parameters.ErrorCriteria = ErrorCriteria;
++++++++++++++++++++++++++++
                    }
                    break;
//          If the user chooses to send the patterns to a server then this method is envoked.
                    case SEND:
                    {
++++++++++++++++++++++++++++
                    }
                    break;
//          If the user chooses to make the network learn then a
//          sequence of back propogation operations is initiated
//          that sends user messages to perform a set of learning operations.
//          This allows the user to interupt the learning process at any time to
//          observe the results.
                    case LEARN:
                    {
++++++++++++++++++++++++++++
                        if ( !net_initialized )
                        {
                            initialize_network ();
                            NumberIterations = 0;
                            PercentageCorrect = 0;
                        }
++++++++++++++++++++++++++++
                    }
                    break;
//          If the user chooses to make the network show what it has learned then a
//          sequence of patterns is shown to the net and the results displayed. The user can pause the show
//          at any time by hitting any keyboard key.
                    case SHOW:
                    {
++++++++++++++++++++++++++++
                    }
                    break;
//          The next messages are received from the popup menu in show mode
//          to allow the user flexability in operation.
// Loop through the patterns.
                    case LOOP:
                    {
++++++++++++++++++++++++++++
                    }
                    break;
// Set the pattern to view if a single pattern is desired.
                    case SET_PATTERN:
                    {
++++++++++++++++++++++++++++
                    }
                    break;
// Continue to learn if directed to.
                    case CONTINUE_LEARNING:
                    {
++++++++++++++++++++++++++++
                    }
                    break;
// Step forward to the next the pattern.
                    case STEP_FORWARD:
                    {
                        NextPaintPattern ++;
                        if ( NextPaintPattern >= num_patterns )
                            NextPaintPattern = 0;
++++++++++++++++++++++++++++
                    }
                    break;
```

```
// Step backward to the last pattern.
                    case STEP_BACK:
                    {
                        NextPaintPattern --;
                        if ( NextPaintPattern < 0 )
                            NextPaintPattern += num_patterns;
++++++++++++++++++++++++++++
                    }
                    break;
//          Return to Edit mode posts an end message for either the show or learn
//          modes of operation.
                    case EDIT_RETURN:
                    {
++++++++++++++++++++++++++++
                    }
                    break;
//          If the user chooses to delete selected neurons then the following
//          procedure suffices.
                    case DELETE_SELECTED:
                    {
++++++++++++++++++++++++++++
                            DeleteSelected ();
                            net_initialized = FALSE;
++++++++++++++++++++++++++++
                            SetupWeights ();
++++++++++++++++++++++++++++
                        }
                    }
                    break;
//          The user can choose to create a layer of either inputs, outputs, or
//          hidden types. If an individual neuron is added then it must be part
//          of an existing layer.
                    case CREATE_INPUT:
                    {
++++++++++++++++++++++++++++
                    }
                    break;
//          These next two methods are used for either creating a hidden layer
//          or editing an existing layer.
                    case CREATE_HIDDEN:
                    {
++++++++++++++++++++++++++++
                    }
                    break;
                    case EDIT_HIDDEN:
                    {
++++++++++++++++++++++++++++
                    }
                    break;
//          This method is used tocreate an output layer
//          or editing an existing output layer.
                    case CREATE_OUTPUT:
                    {
++++++++++++++++++++++++++++
                    }
                    break;
//          This routine displays an about box to tell the user about the
//          program.
                    case ABOUT:
                    {
            ++++++++++++++++++++++++++++
                }
                    break;
                }
            }
            break;
// This last break is the end of the command loop
//          This message is generated if the user stops the learning process
//          with either an input from the keyboard or the mouse.
```

```
            case PM_END_PASS:
                {
//          Display the arrow cursor and send a reassuring message to the user.
++++++++++++++++++++++++++++
                }
                break;
//          This message is generated to perform learning pass through all the vectors
//          at the end of the message execution the PM_PAINT_PASS message is generated
//          if learning has not been completed.
            case PM_LEARNING_PASS:
                {
++++++++++++++++++++++++++++
//          The learning function returns TRUE only if there where any unconverged
//          patterns during the previous pass. If this is the case then post a message
//          to continue learning and paint the next pattern.

                    UnConverged = FALSE;
                    for ( i = 0; i < LEARNING_PASSES; i++ )
                    {
                        if ( (PercentageCorrect = learn ()) < 100 ) UnConverged = TRUE;
                            NumberIterations++ ;
                    }

                    if ( UnConverged )
                    {
++++++++++++++++++++++++++++
                    }
                    else
                    {
                        Looping = FALSE;
                        MessageBox ( hwnd, (LPSTR)"All Patterns Converged",(LPSTR)"Learning
                              Complete",MB_ICONINFORMATION );
                        PostMessage ( hwnd, PM_END_PASS, NULL, NULL );
                    }
                }
                break;
//          This message is generated to perform a show pass through all the vectors
//          at the end of the message execution the PM_PAINT_PASS message is generated to display the results of the show
            case PM_SHOW_PASS:
                {
++++++++++++++++++++++++++++
                }
                break;
//          This messsage is used during learning to allow a paint of the
//          network at the end of the learning pass.
            case PM_PAINT_PASS:
                {
++++++++++++++++++++++++++++
                }
                break;
//          This section is used to respond to the various DDE messages and may
//          be such that the network is acting as both a server and client.

            case WM_DDE_INITIATE:
                //  This message is sent to the window from the client when the
                //  window is acting as the server.
                {
                    // If the window is the client the message is sent to itself
                    // and should be ignored.
++++++++++++++++++++++++++++
                }
                break;
            case WM_DDE_POKE:
                {
                //  This message is sent to the window acting as the server
                //  to transfer a pattern. A response to the client is required.
++++++++++++++++++++++++++++
                }
                break;
//          The following messages are used if DDE is being used to access a
//          server.
```

```
            case WM_DDE_ACK:
                {
                    // This message is sent to the application acting as a client
                    // when the server is responding.
++++++++++++++++++++++++++++
                }
                break;
            case WM_DDE_ADVISE:
                {
                   //  This message is not being used in the way that was intended
                   //  by Microsoft. It is being used to advise the client window by
                   // the server that the server is being closed.
++++++++++++++++++++++++++++
                }
                break;
//          This messsage is used during learn mode and show mode to initiate another
//          cycle it is ignored otherwise.
              case WM_TIMER:
                {
++++++++++++++++++++++++++++
                }
                break;
//          When the user presses vertical scroll the y position is updated and
//          the damaged portion of the window repainted.
            case WM_VSCROLL:
              {
++++++++++++++++++++++++++++
              }
              break;
//          When the user presses horizontal scroll the x position is updated and
//          the damaged portion of the window repainted.
            case WM_HSCROLL:
              {
++++++++++++++++++++++++++++
              }
              break;
//          The paint message is generated whenever the window must be fixed
//          or updated. The network is repainted and control handed back to
//          windows.
            case WM_PAINT:
              {
++++++++++++++++++++++++++++
              }
                 break;
//          The following routines are used to select the various layers and
//          neurons from the mouse input. The left button selects a single item;
//          a click and drag selects a certain number of neurons and a right mouse
//          button selects the entire layer.
            case WM_LBUTTONDOWN:
              {
++++++++++++++++++++++++++++
              }
                 break;
//          The left  button double click does the same as the left button but uses the entire
//          layer.
            case WM_LBUTTONDBLCLK:
              {
++++++++++++++++++++++++++++
              }
                 break;
//          This routine takes care of the right button. The right button instigates a popup menu and if the
// is in learn or show mode, pauses the current state.
            case WM_RBUTTONDOWN:
              {
++++++++++++++++++++++++++++
              }
                 break;
          }
          return(NULL);
}
```

```
// ************************************************************
//*              Layer Initialization Routines                 *
// ************************************************************
BOOL AddToHidden ()
{
//              This routine Appends New Neurons to a hidden layer
++++++++++++++++++++++++++++
}
void SetupInput ()
{
//              This routine initializes the input layer to its default values.
++++++++++++++++++++++++++++
}
BOOL SetupHidden ()
{
//              This routine initializes a hidden layer to its default values.
++++++++++++++++++++++++++++
}

void SetupWeights ()
{
//              This routine initializes the weight matrix to the correct
//              relative positions on the screen.
++++++++++++++++++++++++++++
}
void SetupOutput ()
{
//              This routine initializes the output layer to its default values.
++++++++++++++++++++++++++++
}
int SetupNetwork ( RECT sr )
{
//            This routine places the layers relative to each other
//            on the display screen.
++++++++++++++++++++++++++++
}
// ************************************************************
//*              Picture Painting routines                     *
// ************************************************************
void Paint_Network (int scr_x, int scr_y, int y_zero, int x_zero, unsigned scale, HDC hdc )
{
// This function paints the network on the screen
//            each of the layers is painted if they are visible.
++++++++++++++++++++++++++++
              for ( i = 0; i < num_layers; i++ )
              {
                 DrawNetWeights ( i, scr_x, scr_y, y_zero, x_zero, scale, hdc );
              }
              DrawInputLayer ( scr_x, scr_y, y_zero, x_zero, scale, hdc );
              for ( i = 0; i < num_layers; i++ )
              {
                 DrawHiddenLayer ( i, scr_x, scr_y, y_zero, x_zero, scale, hdc );
              }
              DrawOutputLayer ( scr_x, scr_y, y_zero, x_zero, scale, hdc );
++++++++++++++++++++++++++++
}
void DrawInputLayer (int scr_x, int scr_y, int y_zero, int x_zero, unsigned scale, HDC hdc )
{
// This function determines whether the input layer is visible.
//            If it is those elements currently within the client area are
//            painted on the screen.
++++++++++++++++++++++++++++
}
void DrawHiddenLayer (int hidden, int scr_x, int scr_y, int y_zero, int x_zero, unsigned scale, HDC hdc )
{
// This function determines whether the hidden layer is visible.
//            If it is those elements currently within the client area are
//            painted on the screen.

++++++++++++++++++++++++++++
}
```

```
void DrawStatus ( int hidden, int scr_x, int scr_y, int y_zero, int x_zero, unsigned scale, HDC hdc )
{
// This function determines whether to paint the status boxes on the
//          screen or not and then the boxes are,
//          painted on the screen.
++++++++++++++++++++++++++++
}
void DrawNetWeights ( int position, int scr_x, int scr_y, int y_zero, int x_zero, unsigned scale,HDC hdc )
{
// This function determines whether the Weights are visible.
//          If they are those elements currently within the client area are
//          painted on the screen.
++++++++++++++++++++++++++++
}
void DrawOutputLayer (int scr_x, int scr_y, int y_zero, int x_zero, unsigned scale, HDC hdc )
{
// This function determines whether the input layer is visible.
// If it is those elements currently within the client area are
//          painted on the screen.
++++++++++++++++++++++++++++
}
//**************************************************************
//*              Utility Routines for Drawing                  *
//**************************************************************
BOOL IntClient (int sx, int sy, int yz, int xz, int x, int y )
{

//          This routine checks to see if an object has any parts within
//          the client area and should therefore, be re-drawn. This routine
//          does not work with weight vectors, they use a different method.
++++++++++++++++++++++++++++
}
RECT FindNetExtent ()
{
// This routine determines the size of the network and returns the extents
//          as part of the windows defined RECT structure
++++++++++++++++++++++++++++
}
void CreatePens ()
{
// This routine creates the pens for drawing the weights. Various sizes
// are created for the thresholds. A dashed pen represents a negative value.
++++++++++++++++++++++++++++
}
//**************************************************************
//*              Utility Routines for Main                     *
//**************************************************************
void SaveChangedNet ( HWND hwnd )
{
// This routine saves the network to a file.
++++++++++++++++++++++++++++
}
          void InitGlobal ( HWND hwnd, int x_zero, int y_zero, int scale )
{
//          This function initializes global variables to their default
//          values.
++++++++++++++++++++++++++++
}
int Rescale ( int sx, int sy, RECT r, int scale )
{
//          This routine finds the correct scale factor for zoom to set
//          the network to scale into the displayed window
++++++++++++++++++++++++++++
}
```

```
//*****************************************************************
//*              Routines for Neurons  to learn                   *
//*****************************************************************
float sigmoid ( float x )
// This function evaluates the sigmoid function on the variable
//          x and gives the output value y.
                                -x
                        1 - e
                y =     ------
                                -x
                        1 + e
{
          double y;
          if ( fabs (x) < 100.0 )
            y = ( 1 - exp( - x ) ) / ( 1 + exp( - x ) );
//        If the value of x is greater than 100 then e to the x will be too
//        large or small and then an error will occur.
          else
          {
            if ( x > 0.0 )
               y = 0.99999999;
            else
               y = -0.99999999;
          }
          return ( y );
}
float del_sigmoid ( float x )
{
// This function evaluates the derivative of the sigmoid function
//          on the variable x and gives the output value dy.
                                -x
                          2 e
          y =           ------
                                -x   2
                        ( 1 + e )
          double dy;
          if ( fabs(x) < 100.0 )
            dy = 2.0 * exp( - x ) / pow (( 1 + exp( - x )), 2.0 );
          else
            dy = 7.4e-44;
//        If mag x greater than 100 then set dy to eliminate floating point
// error.
            return ( dy );
}
void calculate_output ( int Pattnum, int layer_num )
//          This function calculates the outputs from a given layer
//          given the ouputs from the previous layer or the input
//          vector in the case of the first layer.
{
          int i, j;
          float sum;
          neuron_type far *neuron;
          float vector[ max ( MAX_WEIGHTS, MAX_NEURONS ) ];
          InputClass far *input_vector;
//        The input to this layer of the network depends on if it
//        is the first hidden layer or subsequent layers.
          if ( layer_num == 0 )
          {
            input_vector = (InputClass far *) GlobalLock ( input.hPattern[Pattnum] );
            for ( i = 0; i < input.number; i ++ )
               vector[i] = input_vector -> GetValue(i);
            GlobalUnlock ( input.hPattern[Pattnum] );
          }
```

```
            else
            {
              for ( i = 0; i < network[layer_num].inputs; i ++ )
              {
                 neuron = (neuron_type far *) GlobalLock ( network[layer_num-1].hNeuron[i] );
                 vector[i] =  neuron -> output;
                 GlobalUnlock (  network[layer_num-1].hNeuron[i] );
              }
            }
// Calculate the sum of products for the weights and the vector input to the neurons
// for ( i = 0; i < network[layer_num].neurons; i++ )
            {
              sum = 0.0;
              neuron =  (neuron_type far *)GlobalLock ( network[layer_num].hNeuron[i] );
              for ( j = 0; j <= network[layer_num].inputs; j++ )
              {
                 if ( j != 0 )
                    sum += neuron -> weight[j] * vector[j-1];
                 else
                    sum += neuron -> weight[j];
              }
              neuron -> sum = sum;
              neuron -> output = sigmoid ( sum );
              GlobalUnlock ( network[layer_num].hNeuron[i] );
            }
}
BOOL calculate_output_errors ( int Pattnum, int last_layer )
//            This function calculates the error vector for the output layer.
{
            int number_neurons, i;
            float error, quant_error;
            BOOL any_pass_error;
            output_pattern_type far *output_vector;
            neuron_type far *neuron;
            number_neurons = network[last_layer].neurons;
            any_pass_error = FALSE;
            output_vector = (output_pattern_type far *) GlobalLock ( output.hPattern[ Pattnum ] );
// Calculate the errors associated with each of the outputs
for ( i = 0; i < number_neurons; i++ )
            {
              neuron =  (neuron_type far *)GlobalLock ( network[last_layer].hNeuron[i] );
              error = ( output_vector -> value[i] - neuron -> output )
                          * del_sigmoid ( neuron -> sum );
              neuron -> error = error;
              quant_error = fabs ( output_vector -> value[i] -  neuron -> output );
              if ( quant_error > ErrorCriteria )
                 any_pass_error = TRUE;
              GlobalUnlock ( network[last_layer].hNeuron[i] );
            }
            GlobalUnlock ( output.hPattern[ Pattnum ] );
            return ( any_pass_error );
}
void backpropagate_errors ( int layer_num )
//            This function backprogates the errors from the next
//            network layer to determine the errors made by the current
//            layer.
{
            int number_neurons, i, num_errors, k;
            float error_sum, error, weight;
            neuron_type far *neuron;
            number_neurons = network[layer_num].neurons;
            num_errors = network[layer_num + 1].neurons;
```

```
            for ( i = 0; i < number_neurons; i++ )
            {
              error_sum = 0;
              for ( k = 0; k < num_errors; k++ )
              {
                 neuron = (neuron_type far *)GlobalLock ( network[layer_num+1].hNeuron[k] );
                 weight = neuron -> weight[i+1];
                 error = neuron -> error;
                 error_sum += error * weight;
                 GlobalUnlock ( network[layer_num+1].hNeuron[k] );
              }
              neuron = (neuron_type far *)GlobalLock ( network[layer_num].hNeuron[i] );
              error = ( error_sum )     * del_sigmoid ( neuron -> sum );
              neuron -> error = error;
              GlobalUnlock ( network[layer_num].hNeuron[i] );
            }
}
void modify_weights ( int layer_num, int Pattnum )
{
// This function modifies the weights of the neurons in the network
// Using the output and back propagated errors.
            int num_neurons, i, j;
            float delta_weight, error, last_delta;
            neuron_type far *neuron;
            InputClass far *input_vector;
            float vector[ max( MAX_WEIGHTS, MAX_NEURONS ) ];
            num_neurons = network[layer_num].neurons;
            for ( i = 0; i < network[layer_num].inputs; i++ )
            if ( layer_num == 0 )
            {
              input_vector = (InputClass far *) GlobalLock ( input.hPattern[Pattnum] );
              for ( i = 0; i < input.number; i ++ )
                 vector[i] =  input_vector -> GetValue(i);
              GlobalUnlock ( input.hPattern[Pattnum] );
            }
            else
            {
              for ( i = 0; i < network[layer_num].inputs; i ++ )
              {
                 neuron = (neuron_type far *)GlobalLock ( network[layer_num-1].hNeuron[i] );
                 vector[i] =  neuron -> output;
                 GlobalUnlock (  network[layer_num-1].hNeuron[i] );
              }
            }
            for ( i = 0; i < num_neurons; i ++ )
            {
              neuron = (neuron_type far *)GlobalLock ( network[layer_num].hNeuron[i] );
              error = neuron -> error;
              for ( j = 0; j <= network[layer_num].inputs; j++ )
              {
                 last_delta = neuron -> delta[j];
                 if ( j != 0 )
                    delta_weight = LearningRate * error * vector[j-1] + WeightMoment * last_delta;
                 else
                    delta_weight = LearningRate * error + WeightMoment * last_delta;
                 neuron -> delta[j] = delta_weight;
                 neuron -> weight[j] += delta_weight;
              }
              GlobalUnlock ( network[layer_num].hNeuron[i] );
            }
}
```

```
unsigned int learn ( )
{
// This routine allows the network to learn by applying back propagation untill
// all the outputs are within an error tolerance or the user pauses the operation.
            BOOL any_errors, error_test;
            int Pattnum, layer_num, PercentErrors;
            any_errors = FALSE;
            PercentErrors = 0;
//          There are currently no errors for this pass through the learning
//          process.

            for ( Pattnum = 0; Pattnum < num_patterns; Pattnum ++ )
            {
                for ( layer_num = 0; layer_num < num_layers; layer_num++ )
                    calculate_output ( Pattnum, layer_num );
//          Calculate the outputs from the layers excited by the input pattern.
                error_test = calculate_output_errors ( Pattnum, num_layers - 1 );
//          Calculate the errors at the output layer. The difference between
//          the desired output and the actual output times the derivative of
//          the sigmoid function.
                for ( layer_num = num_layers - 2; layer_num >= 0; layer_num -- )
                    backpropagate_errors ( layer_num );
//          Backpropagate errors from the previous layers to the hidden network
//          layers. The error is weighted by the sigmoid function and the weight
//          of the connection between the neurons.
                for ( layer_num = 0; layer_num < num_layers; layer_num ++ )
                    modify_weights ( layer_num, Pattnum );
//          Modify the weights dependent on the back propogated errors.
                if ( error_test )
                {
                    any_errors = TRUE;
                    PercentErrors++;
                }
            }
            return ( (num_patterns <= 0)?(0):((100*(num_patterns-PercentErrors)) / num_patterns ) );
}
int CalculatePercentCorrect ( )
{
// Calculate the networks current percent correct outputs
++++++++++++++++++++++++++
}
void ModifyColours ( int PaintPattern )
{
// Modify the displayed colours of the neurons based on their current outputs
++++++++++++++++++++++++++++
}
// ***************************************************************
// *              Utility Routines for Neurons                   *
// ***************************************************************

int SingleLayerSelected ()
{
//          This function determines which objects in the network are currently
//          selected and determines if they represent a single layer. If they
//          do then the layer number 1 to num_layers, is returned or NULL.
++++++++++++++++++++++++++++
}
void DeleteSelected ( )
{
//          This function determines which objects in the network are currently
//          selected and deletes those objects.
++++++++++++++++++++++++++++
}
select_type Find_Object ( int sx, int sy, int yz, int xz, int scale )
{
//          This function determines which object in the network is currently
//          pointed at by the mouse.
++++++++++++++++++++++++++++
}
```

```
void UnselectAll ()
{
//              This function deselects all the objects in the network.
++++++++++++++++++++++++++++
}
void SelectNeuron ( HANDLE hNeuron )
{
//              This routine selects the neuron pointed at by the handle.
++++++++++++++++++++++++++++
}
BOOL Allocate ( int i, int j )
//              This routine allocates memory for the weight vectors
//              if it is unable to do so a value of FALSE is returned.
{
++++++++++++++++++++++++++++
}
BOOL initialize_neurons ( int layer_num )
{
//              This routine deselects and makes visible all the neurons in the
//              layer_num. It is used as part of the startup and viewing routines.
++++++++++++++++++++++++++++
}
void MakeSelectedVisible ( BOOL visible )
{
//              This function makes all selected neurons either invisible
//              or visible dependent on the parameter visible.
++++++++++++++++++++++++++++
}
float neuron_output ( HANDLE hNeuron )
{
//              This routine finds the output associated with the neuron pointed at by
//              the handle.
++++++++++++++++++++++++++++
}
float neuron_error ( HANDLE hNeuron )
{
//              This routine finds the error associated with the neuron pointed at by
//              the handle.
++++++++++++++++++++++++++++
}
BOOL neuron_selected ( HANDLE hNeuron )
{
//              This routine checks to see if the neuron pointed at by
//              the handle is selected.
++++++++++++++++++++++++++++
}
BOOL neuron_visible ( HANDLE hNeuron )
{
//              This routine checks to see if the neuron pointed at by
//              the handle is selected.
++++++++++++++++++++++++++++
}
void SetThresholds ()
{
//              This function sets the weight thresholds values according
//              to the value of the weights.
++++++++++++++++++++++++++++
}
BOOL initialize_network ()
{
//              This function sets the weight matrix to initial values randomly
//              distributed between -1 and +1.
++++++++++++++++++++++++++++
}
++++++++++++++++++++++++++++
BOOL AllocateWholeNet ( )
{
//              Allocate memory for all the neurons to see if there is a problem,
//              if there is then return a false indication!
++++++++++++++++++++++++++++
}
```

```
void    DrawChildWindow ( HANDLE hdc,
                          int CurrentPaintPattern,
                          long NumberIterations,
                          long PercentageCorrect,
                          char *DesiredPattern ,
                          char *ActualPattern  )
{
//      This code fragment is used in the learn or show modes to place the
//      current pattern number, iteration, status in the top left corner of the screen.
*****************************
}
```

Appendix B

This Appendix supplies a partial listing for the program "Sensor Input". A full listing would occupy numerous pages and therefore, for the sake of brevity is not included. The eliminated code fragments are primarily "Windows" specific and do not add to the overall program comprehension. The code specific to the input and display of the tactile images is included in its entirety. This program allows a user to: capture, display, store and manipulate tactile images as well as send the images to a neural network for recognition.

SENSOR.H

```
//          Prototypes for dialog boxes

BOOL FAR PASCAL _export PatternFileProc (HWND, WORD, WORD, LONG );
BOOL FAR PASCAL _export SavePatDlgProc (HWND, WORD, WORD, LONG );

//          Defines for drawing

#define MAX_WIDTH 16
#define MAX_HEIGHT 16
#define MAX_ARRAY MAX_WIDTH * MAX_HEIGHT
#define NUMBER_WIDTH 5
#define NULL_STRING ""
#define X_OFFSET 2
#define Y_OFFSET 17
#define Y_SPACING 16
#define X_SPACING (16 * 1.7 )
#define X ((MAX_WIDTH * X_SPACING)/2.0)
#define Y ((MAX_HEIGHT * Y_SPACING)/2.0)
#define TYPE_SIZE 16
#define BUTTON_HEIGHT 30
#define BUTTON_WIDTH 80

//          Defines for sensor menu parameters

#define SENSE_PATTERN -1000
#define NEXT_PATTERN 101
#define ADD_PATTERN 102
#define PB_SENSE 103
#define PREVIOUS_PATTERN 104
#define DELETE_PATTERN 105
```

SENSOR.CPP (Partial Listing)

```
// This windows program is used to input information from the tactile
// sensor and display it on the screen in a pseudo three dimensional representation
// It also allows for the naming and storage of the captured patterns.
// This file is a partial listing, missing information is depicted by ++++++++++++++++++++++++++
++++++++++++++++++++++
long FAR PASCAL _export WndProc( HWND hWnd, WORD iMessage,
                                                WORD wParam, LONG lParam );
// This is the procedure that handles the main window

++++++++++++++++++++++
// The client class is an object used to communicate with a server in
// another application.
class ClientClass {

private:    ATOM atomApplication, atomTopic;
++++++++++++++++++++++

public:     ClientClass ( );
++++++++++++++++++++++
};

void ClientClass::StartConversation ( HANDLE HandleSent )
{
//          This routine is used to start a conversation
//          with a server if one is not already started.
++++++++++++++++++++++
}
ClientClass::ClientClass ( )
{
// This constructer is used to start initialize the client structure.
          InConversation = FALSE;
}
void ClientClass::Initiate ( HWND hwndClientDDE )
{
// This function is used to start a conversation with a server.
++++++++++++++++++++++
}
BOOL ClientClass::SendServerData ( LPSTR szItemValue )
{
// This function sends an integer to the server process.
++++++++++++++++++++++
}
class InputClass
{
//          This class is used to manipulate and store input patterns.
++++++++++++++++++++++
};

void InputClass::Initialize ()
{
          int i;
//          This function initializes the values of the strings
//          to zero if a constructor can't be called.
++++++++++++++++++++++
}
LPSTR InputClass::GetString ( int MaxNumber )
{
// This function is used to retrieve a string with the input pattern
// values imbedded for transfer to another application.
++++++++++++++++++++++
}
```

```
void InputClass::PlaceFloat ( char NumberString[], float value )
{
// This routine puts the floating point number "value"
// into the string NumberString. It guarantees that exactly NUMBER_WIDTH+1
// characters are placed in the string.
++++++++++++++++++++++++
}void InputClass::PutString ( int MaxNumber )
{
// This function is used to place a string with the input pattern
// values imbedded into the float values.
++++++++++++++++++++++++
}
float InputClass::ConvertToFloat ( char NumberString[] )
{
//          This function converts to a float the number in the string NumberString
//          it only applies to this class and is not generic.
++++++++++++++++++++++++
}
class OutputClass
{
//          This class is used to output stored patterns to a file.
private:
            int CurrentOutputType, NumberOutputTypes;
            char ObjectType[LENGTH_OF_STRING];

            BOOL StringSave ( HANDLE hFile, LPSTR String )
            {
            // This function saves strings into the file pointed to
            // by the file handle for MaxChar.
++++++++++++++++++++++++
            }
public:

++++++++++++++++++++++++
            void SetType ( LPSTR String )
            {
            // This function sets the current pattern type equal to the String
            // and determines what the current output type should be.
++++++++++++++++++++++++
            }

            BOOL WritePattern ( HANDLE hFile, LPSTR String )
            {
            // This function writes the pattern into the currently open file
            // pointed to by hFile.
++++++++++++++++++++++++
          }
};

class PatternClass
{
//          This class is used to manipulate and store patterns.
public:

            typedef struct {
            float value[MAX_NEURONS];
            char PatternType[LENGTH_OF_STRING];
            } output_pattern_type;

            typedef struct {
            char Type[LENGTH_OF_STRING];
            } pattern_string;

            struct {
            int number;
            HANDLE hPattern[MAX_PATTERNS];
            } input;
```

```
        struct {
        int number;
        HANDLE hPattern[MAX_PATTERNS];
        pattern_string Pattern[MAX_NEURONS];
        } output;
        ScanfStruct ExtractFloat ( char string[], int Length );
        int PatternClass::ReadInputPattern ( HANDLE hFile, int PattNum );
        int PatternClass::ReadOutputPattern ( HANDLE hFile, int PattNum );
        BOOL PatternClass::getstring ( HANDLE hFile, char string[] );
public:
        typedef struct {
        int number;
        int status;
        } pattern_status_type;

++++++++++++++++++++++++
        {
        // This function retrieves the name of the current pattern.
          output_pattern_type far *pattern;
          if ( ( PattNum >= 0 ) && ( PattNum < num_patterns ) )
          {
              pattern = ( output_pattern_type far *) GlobalLock ( output.hPattern[PattNum] );
              if ( pattern != NULL )
                  lstrcpy ( String, (LPSTR) pattern -> PatternType );
              else
                  lstrcpy ( String, (LPSTR) NULL_STRING );
          }
          else
              lstrcpy ( String, (LPSTR) NULL_STRING );
        }
        void PutType ( int PattNum, LPSTR String )
        {
        // This function sets the name of the current pattern.
          output_pattern_type far *pattern;
          if ( ( PattNum >= 0 ) && ( PattNum < num_patterns ) )
          {
              pattern = ( output_pattern_type far *) GlobalLock ( output.hPattern[PattNum] );
              if ( pattern != NULL )
                  lstrcpy ( (LPSTR) pattern -> PatternType, String );
          }
        }
        PatternClass ()
// This is the pattern class constructor
{
++++++++++++++++++++++++
        }
};

BOOL PatternClass::SaveOutputs ( HANDLE hFile )
{
//      This function is used to save the outputs in a file pointed to
//      by hFile.
++++++++++++++++++++++++
}
void PatternClass::SetOutputs ()
{
//      This function is used to determine how many neurons will be required
//      by searching through the patterns and determining the number of different
//      types.
          int i, j;
          pattern_string String;
          output_pattern_type far *PattPoint;
          BOOL FoundPattern;
```

```
            for ( i = 0; i < MAX_NEURONS; i++ )
            {
                lstrcpy ( (LPSTR) String.Type, NULL_STRING );
                output.Pattern[i] = String;
            }
            output.number = 0;
            for ( i = 0; i < num_patterns; i++ )
            {
                PattPoint = ( output_pattern_type far * ) GlobalLock ( output.hPattern[i] );
                if ( PattPoint == NULL ) continue;
                j = 0;
                String = output.Pattern[0];
                FoundPattern = FALSE;
                while ( lstrcmp ( (LPSTR) String.Type, (LPSTR) NULL_STRING ) != 0 )
                {
                    if ( lstrcmp ( (LPSTR) String.Type, (LPSTR) PattPoint -> PatternType ) == 0 )
                    {
                        FoundPattern = TRUE;
                        break;
                    }
                    j++;
                    if ( j >= MAX_NEURONS ) break;
                    String = output.Pattern[j];
                }
                if ( !FoundPattern )
                {
                    lstrcpy ( (LPSTR) String.Type, (LPSTR) PattPoint -> PatternType );
                    output.Pattern[output.number] = String;
                    output.number ++;
                }
            }
}

void PatternClass::SortOutputs ()
{
//          This function is used to sort the different pattern types alphabetically.
++++++++++++++++++++++++
}

void PatternClass::FreeAllPatterns ()
{
//          This function frees all the memory allocated for patterns.
++++++++++++++++++++++++
}

void PatternClass::DeletePattern ( int PattNum )
{
//          This function frees all the memory allocated for PattNum
//          and re-organizes the other patterns. The last pattern is copied
//          into the freed space.
++++++++++++++++++++++++
}
++++++++++++++++++++++++
pattern_status_type PatternClass::ReadPatternFile ( HANDLE hFile )
{
//            This function reads the patterns in the file specified by the file
//            handle.
++++++++++++++++++++++++
}
int PatternClass::AddPattern ( char *String )
{
//          This routine is used to add patterns to the current set.
++++++++++++++++++++++++
}
float PatternClass::GetValue ( int i, int j, int PattNum )
{
//          This routine gets the value at a location specified by the co-ordinates
//            (i,j).
++++++++++++++++++++++++
}
```

```
void PatternClass::PutValue ( float value, int i, int j, int PattNum )
{
//          This routine puts the value into a location specified by the co-ordinates
//            (i,j).
++++++++++++++++++++++++
}
int PatternClass::ReadInputPattern ( HANDLE hFile, int PattNum )
{
//          This routine reads an input pattern from the file defined
//            by the file handle. The format for a valid input file is the string
//            "INPUT" followed by a series of floats for the data and then the string
//            "END".
++++++++++++++++++++++++
}
int PatternClass::ReadOutputPattern ( HANDLE hFile, int PattNum )
{
//          This routine reads an output pattern from the file defined
//            by the file handle. The format for a valid output file is the string
//            "OUTPUT" followed by a series of floats for the data and then the string
//            "END".
++++++++++++++++++++++++
}

BOOL PatternClass::getstring ( HANDLE hFile, char istring[] )
{
// This function gets a string from the file addressed by hFile.
++++++++++++++++++++++++
}
class SensorClass
{
//          This class is used to manipulate and store input patterns.
private:
          float fvalue[MAX_ARRAY];
          char far *StrPnt;
          HANDLE StrHandle;
          void PlaceFloat ( char NumberString[], float value );
          float ConvertToFloat ( char NumberString[] );
          float Execute_AI_VRead ();
          void SetAddress (int i, int j);
          int SizeOfString;

public:
          float GetValue ( int i, int j ) { return fvalue[i*MAX_WIDTH + j]; }
          float GetValue ( int i ) { return fvalue[i]; }
          void SensePattern ();
          void PutValue ( float v, int i, int j )
          {
            if ( v > 1.0 ) v = 1.0;
            if ( v < -1.0 ) v = -1.0;
            fvalue[i*MAX_WIDTH + j] = v;
          }
          void PutValue ( float v, int i )
          {
            if ( v > 1.0 ) v = 1.0;
            if ( v < -1.0 ) v = -1.0;
            fvalue[i] = v;
          }
          SensorClass ()
          {
            int i;
//          This constructor initializes the values of the strings
//          to zero.
            SizeOfString = MAX_WIDTH * MAX_HEIGHT * ( NUMBER_WIDTH + 2 );
            for ( i = 0; i < MAX_ARRAY; i++ )
                fvalue [i] = -1.0;
          }
```

```
        LPSTR GetString ( int MaxNumber );
        void PutString ( int MaxNumber );
        BOOL LockString ()
        {
            StrHandle = GlobalAlloc ( GMEM_MOVEABLE, SizeOfString );
            if ( !StrHandle ) return ( FALSE );
            StrPnt = (LPSTR) GlobalLock ( StrHandle );
            if ( StrPnt == NULL )
            {
               GlobalFree ( StrHandle );
               return ( FALSE );
            }
            return ( TRUE );
        }
        void UnlockString ()
        {
            GlobalUnlock ( StrHandle );
            GlobalFree ( StrHandle );
        }
};

//          This function inputs a pattern from the sensor.
void SensorClass::SensePattern ()
{
        HCURSOR hNewCur,          // new (hourglass) cursor
                hOldCur;          // old (arrow) cursor
        int i, j;
        float volts;

// change the cursor to the hourglass
        hNewCur = LoadCursor(NULL,IDC_WAIT);
        hOldCur = SetCursor(hNewCur);

        for ( i = 0; i < MAX_WIDTH; i++ )
        {
          for ( j = 0; j < MAX_HEIGHT; j++ )
          {
            SetAddress ( i, j );
            volts = Execute_AI_VRead ();
            volts = ( 4.0 * volts ) - 1.0;
            if ( volts > 13.0 ) volts = 13.0;
            if ( volts < -1.0 ) volts = -1.0;
            PutValue ( volts, i, j );
          }
        }
// change the cursor back to the previous value
        SetCursor(hOldCur);
}

//  -- This function calls the LabDriver function call AI_VRead, which
//     reads the voltage on the board as given by the input parameters
float SensorClass::Execute_AI_VRead ()
{int    brd,    // board number
       ch,              // channel numbe
       gain,            // voltage gain
       err;             // return error code
  double  volt;         // voltage
        brd = 1;
        ch = 0;
        gain = 1;
  // call DLL function
        err = AI_VRead(brd,ch,gain,&volt);
        if ( err != 0 )
          volt = 0.0;
        return ( volt );
}
//  -- This function calls the LabDriver function call DIG_Out_Port, which
//     sets the data port on the board as given by the input parameters
void SensorClass::SetAddress ( int i, int j )
{
        int    brd,             // board number
```

```
            port,           // port number
            err;            // return error code
        int   address;          // voltage
        brd = 1;
        port = 0;
        address = ((i << 4 ) + ( j & 0xff )) & 0xff;
 // call DLL function
        err = DIG_Out_Port (brd,port,address);
}
LPSTR SensorClass::GetString ( int MaxNumber )
{
// This function is used to retrieve a string with the input pattern
// values imbedded for transfer to another application.
        int i;
        char NumberString [NUMBER_WIDTH+2];
        lstrcpy ( StrPnt, NULL_STRING );
        if ( MaxNumber > MAX_ARRAY )
          MaxNumber = MAX_ARRAY;
        for ( i = 0; i < MaxNumber; i++ )
        {
          PlaceFloat ( NumberString, fvalue[i] );
          lstrcat ( StrPnt, NumberString );
        }
        return ( StrPnt );
}
void SensorClass::PlaceFloat ( char NumberString[], float value )
{
// This routine puts the floating point number "value"
// into the string NumberString. It guarantees that exactly NUMBER_WIDTH+1
// characters are placed in the string.
++++++++++++++++++++++++
}
void SensorClass::PutString ( int MaxNumber )
{
// This function is used to place a string with the input pattern
// values imbedded into the float values.
++++++++++++++++++++++++
}

float SensorClass::ConvertToFloat ( char NumberString[] )
{
//          This function converts to a float the number in the string NumberString
//          it only applies to this class and is not generic.
++++++++++++++++++++++++
}
PatternClass PatternGroup;

class Main
{
// Standard Windows Main Message Loop and initialization
++++++++++++++++++++++++
}
class Window
{
++++++++++++++++++++++++
                // Pure virtual function makes Window an abstract class.
                virtual long WndProc( WORD iMessage, WORD wParam, LONG lParam ) = 0;
};

class MainWindow : public Window
{
// This class sets the attributes of the Main window
++++++++++++++++++++++++
```

```
void MainWindow::DrawPattern ( HDC hdc, int PattNum )
{
        typedef struct {
        float x;
        float y;
        } vector;
// This routine Draws the sensed pattern on the video monitor
        #define SCALE 20.0
        HANDLE hpen, hbr, hOldbr, hOldpen;
        int x, y;
        int x1, x2, x3, x4, y1, y2, y3, y4;
        float v1, v2, v3, v4;
        float Dither, slope;
        POINT vertex[4];
        vector xvec, yvec;
        RECT r;
        char dString[LENGTH_OF_STRING];
        char tString[20];
        hbr = GetStockObject ( BLACK_BRUSH );
        hOldbr = SelectObject ( hdc, hbr );
        xvec.x = (float) X / (float) MAX_WIDTH;
        xvec.y = (float) Y / (float) MAX_WIDTH;
        yvec.x = (float) X / (float) MAX_HEIGHT;
        yvec.y = - (float) Y / (float) MAX_HEIGHT;
        GetClientRect(hWnd, &r);
        Rectangle ( hdc  , 0, 0, r.right, r.bottom );
        if ( PattNum != SENSE_PATTERN )
//      This next piece of code is used to draw an info. box in the top of
//      the screen that displays the number of the current pattern.
        {
            ShowWindow ( hEPattNum, SW_SHOWNORMAL );
            if ( PatternGroup.num_patterns > 0 )
                itoa( PattNum+1, tString, 10 );
            else
                itoa( PattNum, tString, 10 );
            lstrcpy ( (LPSTR) dString, (LPSTR) " Pattern = " );
            lstrcat ( (LPSTR) dString, (LPSTR) tString );
            lstrcat ( (LPSTR) dString, (LPSTR) "  of  " );
            itoa( PatternGroup.num_patterns, tString, 10 );
            lstrcat ( (LPSTR) dString, (LPSTR) tString );
            if ( hEPattNum )
                SetWindowText( hEPattNum, (LPSTR) dString );
//      This next piece of code displays the pattern type in an edit box.
            PatternGroup.GetType ( PattNum, (LPSTR) dString );
            if ( hEPattType )
                SetWindowText( hEPattType, (LPSTR) dString );
        }
        else
            ShowWindow ( hEPattNum, SW_HIDE );
        hpen = CreatePen ( PS_SOLID, 1, PALETTERGB(0,0,0) );
        hOldpen = SelectObject ( hdc, hpen );
        for ( x = 0; x < (MAX_WIDTH-1); x++ )
        {
            for ( y = 0; y < (MAX_HEIGHT-1); y++ )
            {
                if ( PattNum != SENSE_PATTERN )
                {
                    v1 = ( 1.0 + PatternGroup.GetValue ( x,  y  , PattNum )) * SCALE;
                    v2 = ( 1.0 + PatternGroup.GetValue ( x,  y+1, PattNum )) * SCALE;
                    v3 = ( 1.0 + PatternGroup.GetValue ( x+1, y+1, PattNum )) * SCALE;
                    v4 = ( 1.0 + PatternGroup.GetValue ( x+1, y  , PattNum )) * SCALE;
                }
                else
                {
                    v1 = ( 1.0 + Pattern.GetValue ( x,  y  )) * SCALE;
                    v2 = ( 1.0 + Pattern.GetValue ( x,  y+1 )) * SCALE;
                    v3 = ( 1.0 + Pattern.GetValue ( x+1, y+1 )) * SCALE;
                    v4 = ( 1.0 + Pattern.GetValue ( x+1, y  )) * SCALE;
                }
```

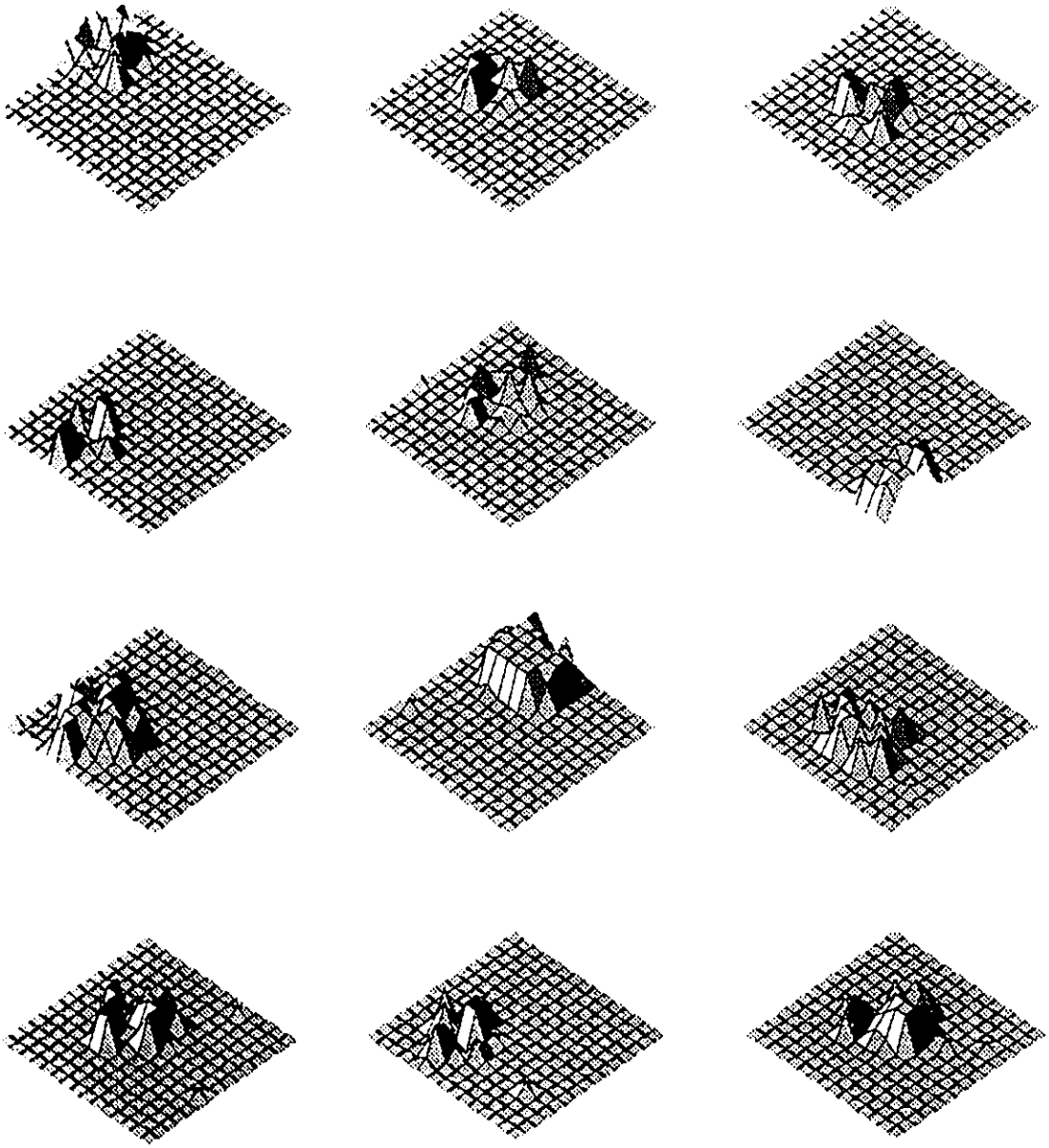
```
                    x1 = X_OFFSET + x * xvec.x + y * yvec.x;
                    y1 = Y_OFFSET + x * xvec.y + y * yvec.y + Y - v1;
                    x2 = X_OFFSET + x * xvec.x + (y+1) * yvec.x;
                    y2 = Y_OFFSET + x * xvec.y + (y+1) * yvec.y + Y - v2;
                    x3 = X_OFFSET + (x+1) * xvec.x + (y+1) * yvec.x;
                    y3 = Y_OFFSET + (x+1) * xvec.y + (y+1) * yvec.y + Y - v3;
                    x4 = X_OFFSET + (x+1) * xvec.x + y * yvec.x;
                    y4 = Y_OFFSET + (x+1) * xvec.y + y * yvec.y + Y - v4;
                    slope = (4.0 * (v3 - v1) - fabs ( v2 - v4 )) / (16.0 * SCALE);
                    hbr = GetStockObject ( DKGRAY_BRUSH );
                    if ( slope > -0.3 ) hbr = GetStockObject ( GRAY_BRUSH );
                    if ( slope > -0.2 ) hbr = GetStockObject ( LTGRAY_BRUSH );
                    if ( slope >  0.15 ) hbr = GetStockObject ( WHITE_BRUSH );
                    hOldbr = SelectObject ( hdc, hbr );
                    vertix[0].x = x1;
                    vertix[0].y = y1;
                    vertix[1].x = x2;
                    vertix[1].y = y2;
                    vertix[2].x = x3;
                    vertix[2].y = y3;
                    vertix[3].x = x4;
                    vertix[3].y = y4;
                    Polygon ( hdc, (LPPOINT) vertix, 4 );
                }
            }
            SelectObject ( hdc, hOldpen );
            DeleteObject ( hpen );
}

long MainWindow::WndProc( WORD iMessage, WORD wParam, LONG lParam )
{

            switch (iMessage)
            {
                case WM_CREATE:
                    {
// Used to create the window
++++++++++++++++++++++++
                    }
                    break;

                case WM_PAINT:
                    {
                        Paint( CurrentPattern );
                    }
                        break;

                case WM_DDE_ACK:
                    // This message is sent to the application acting as a client
                    // when the server is responding.
                        {
                            ClientStartConversation ( wParam );
                        }
                        break;

                case WM_COMMAND:
                    {
                        switch (wParam)
                        {
//          The Save As function allows the user to store the patterns in a file
//          in a format that the Neural Net and other applications can read.
```

```
                        case SAVEAS:
                        {
++++++++++++++++++++++
                |
                        break;
//          If the user chooses to sense a pattern then the sensor is poled for its
//          current values and drawn on the screen.
                        case PB_SENSE:
                        {
                            HDC hdc;
                            LPSTR StrPnt;
                            char String[LENGTH_OF_STRING];
                            Pattern.SensePattern ();
                            hdc = GetDC ( hWnd );
                            DrawPattern ( hdc, SENSE_PATTERN );
                            if ( Client.Conversing () )
                            {
                                if ( Pattern.LockString () )
                                {
                                    StrPnt = Pattern.GetString ( MAX_WIDTH*MAX_HEIGHT );
                                    Client.SendServerData ( StrPnt );
                                    Pattern.UnlockString ();
                                }
                            }
                            if ( CurrentPattern != SENSE_PATTERN )
                            {
                            //  If this is the first of the sense group then fill
                            //  the pattern type box with the name of the last type.

                                PatternGroup.GetType ( CurrentPattern, (LPSTR) String );
                                if ( hEPattType )
                                    SetWindowText( hEPattType, (LPSTR) String );
                            }
                            CurrentPattern = SENSE_PATTERN;
                            ReleaseDC ( hWnd, hdc );
                        }
                        break;

//          If the user chooses to add a pattern then the last sensed pattern
//          is added to the current pattern set. A dialog box appears to allow
//          the user to name the new pattern.

                        case ADD_PATTERN:
                        {
++++++++++++++++++++++++
                            if ( PattNum > 0 )
                            {
                                for ( i = 0; i < MAX_WIDTH; i++ )
                                    for ( j = 0; j < MAX_HEIGHT; j++ )
                                    {
                                        value= Pattern.GetValue ( i, j );
                                        PatternGroup.PutValue( value, i, j, PattNum );
                                    }
                                hdc = GetDC ( hWnd );
                                DrawPattern ( hdc, PattNum );
                                CurrentPattern = PattNum;
                                ReleaseDC ( hWnd, hdc );
                            }
                        }
                        break;
//          If the user chooses Input Pattern then a dialog box appears to choose
//          the file to open for learning.
                        case PATTERN_INPUT:
                        {
++++++++++++++++++++++++
                        }
                        break;
          This function allows the user to flip through the patterns by incrementing
          the current pattern.
```

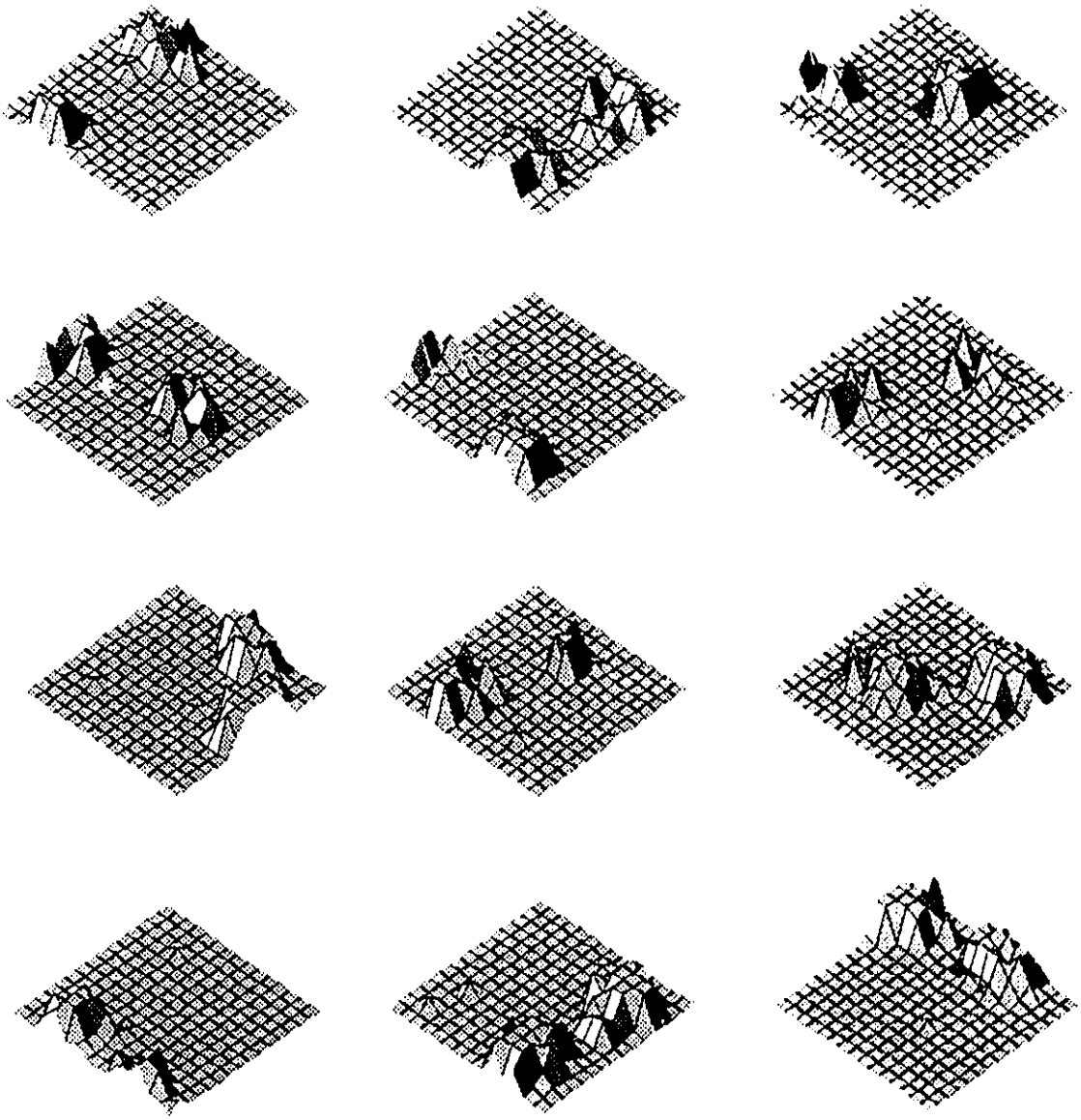
```
                        case NEXT_PATTERN:
                        {
++++++++++++++++++++++++
                        }
                        break;
//          This function allows the user to flip through the patterns by decrementing
//          the current pattern.
                        case PREVIOUS_PATTERN:
                        {
++++++++++++++++++++++++
                        }
                        break;
//          This next function is used to delete a pattern from the those saved
//          in memory.
                        case DELETE_PATTERN:
                        {
++++++++++++++++++++++++
                        }
                        break;
//          This menu option is provided so that the network can receive patterns
//          directly.
                        case SEND:
                        {
                            Client.Initiate ( hWnd );
                        }
                        break;
                        case EXIT:
                        {
                            PatternGroup.FreeAllPatterns ();
                            DestroyWindow ( hWnd );
                        }
                        break;
                    }
                }
                    break;
                case WM_CLOSE:
++++++++++++++++++++++++
                    break;
                case WM_DESTROY:
                    PostQuitMessage( 0 );
                    break;
                default:
                    return DefWindowProc( hWnd, iMessage, wParam, lParam );
            }
}

++++++++++++++++++++++++
long FAR PASCAL _export WndProc( HWND hWnd, WORD iMessage, WORD wParam,
                                                    LONG lParam )
{
++++++++++++++++++++++++
}
int PASCAL WinMain( HANDLE hInstance, HANDLE hPrevInstance, LPSTR lpszCmdLine,
        int nCmdShow )
{
++++++++++++++++++++++++
  // A Windows class should be registered with Windows before any windows
  // of that type are created.
  // Register here all Windows classes that will be used in the program.
        if ( ! Main::hPrevInstance ) {
            MainWindow::Register();
  }
  MainWindow MainWnd;
        return Main::MessageLoop();
}
```

Appendix C

One-Two Network - Selection of correctly categorized patterns from the test set

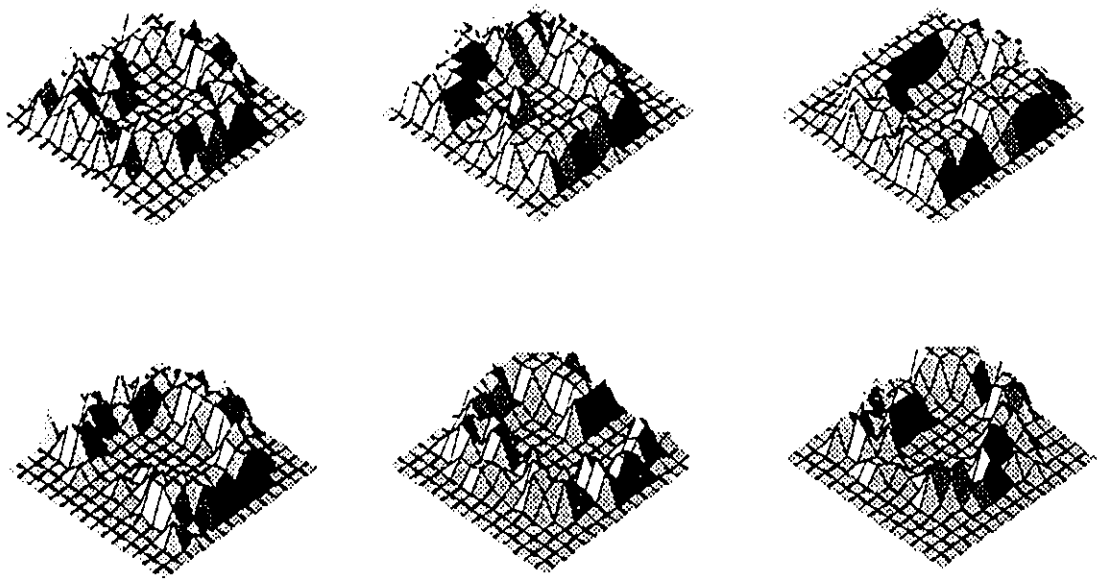


Examples of patterns correctly classified as "One"

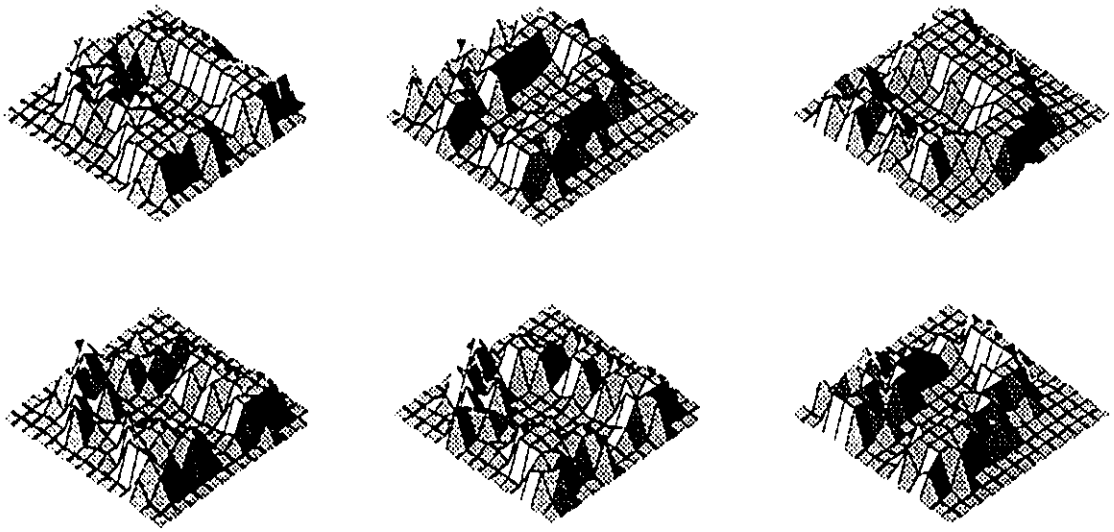


Examples of patterns correctly classified as "Two"

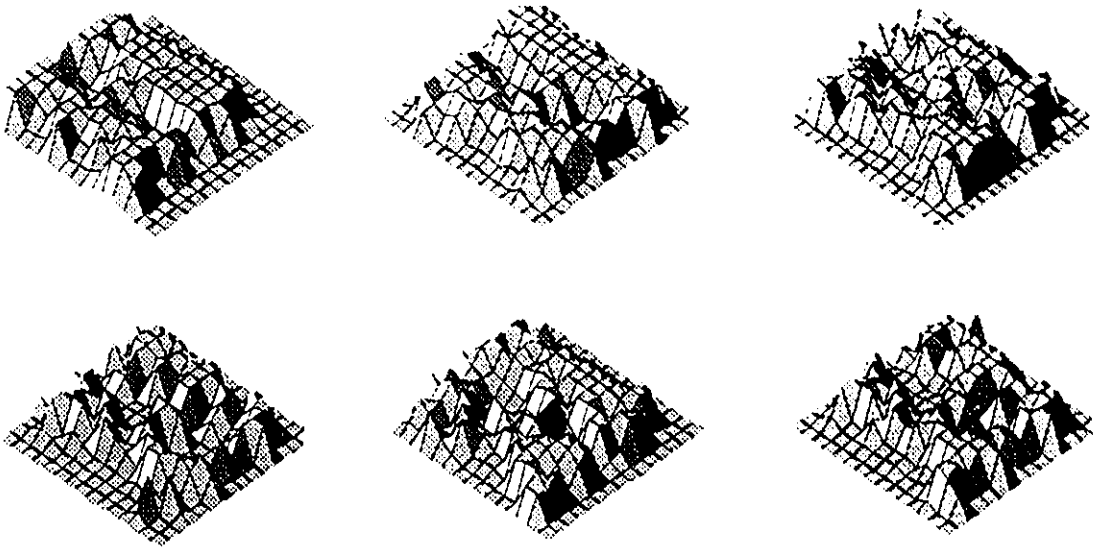
Big Block Network - Selection of correctly categorized patterns from the test set



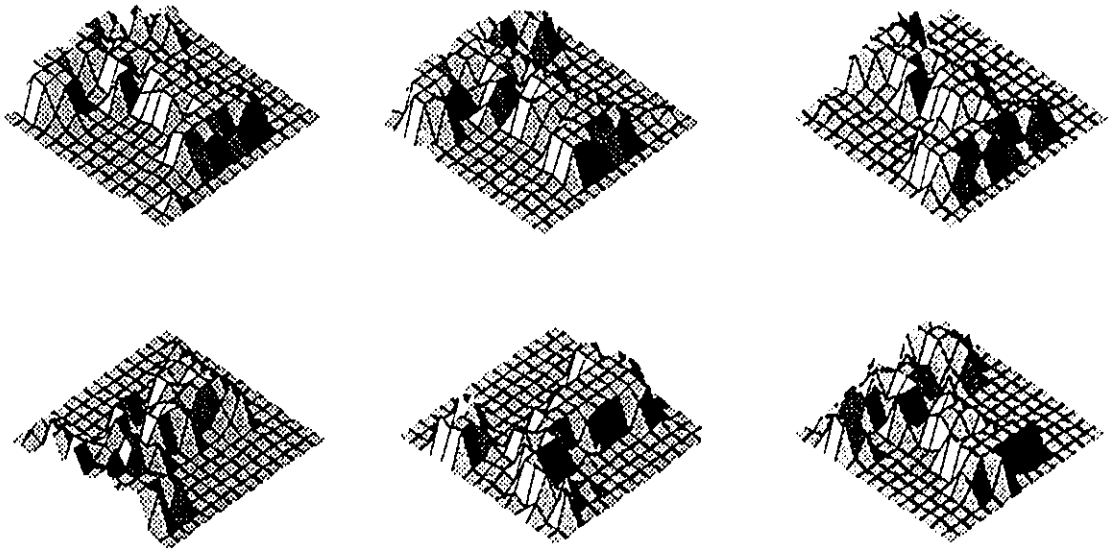
Examples of patterns correctly classified as "**G**"



Examples of patterns correctly classified as "**U**"

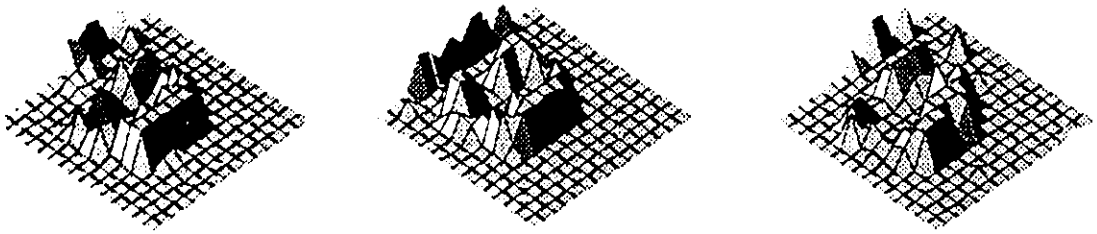


Examples of patterns correctly classified as "**H**"

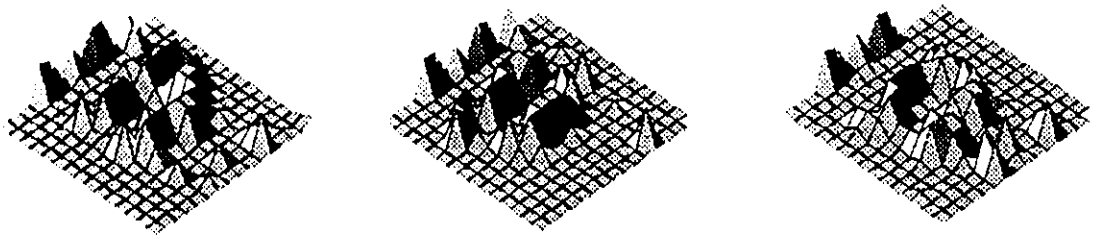


Examples of patterns correctly classified as "**T**"

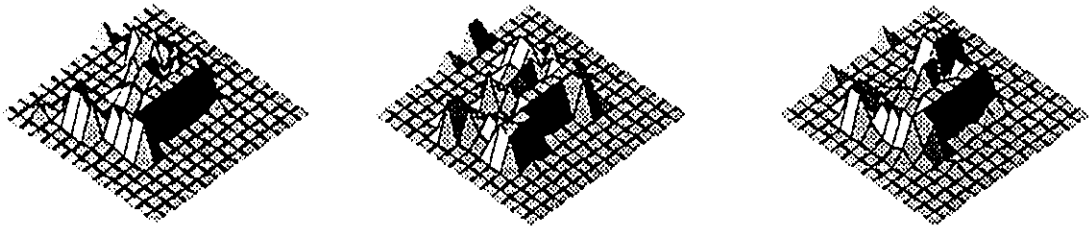
Small Block Network - Selection of correctly categorized patterns from the test set



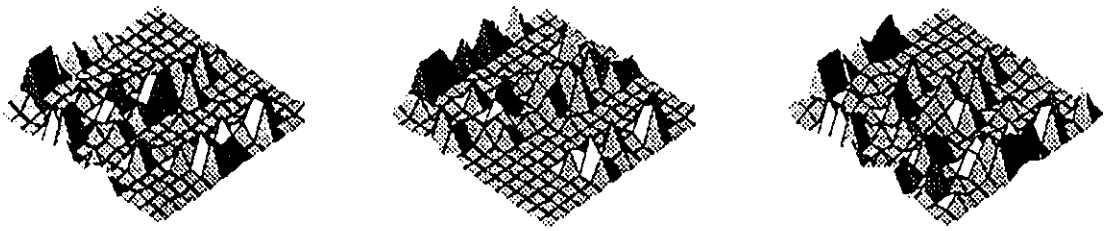
Examples of patterns correctly classified as "**C**"



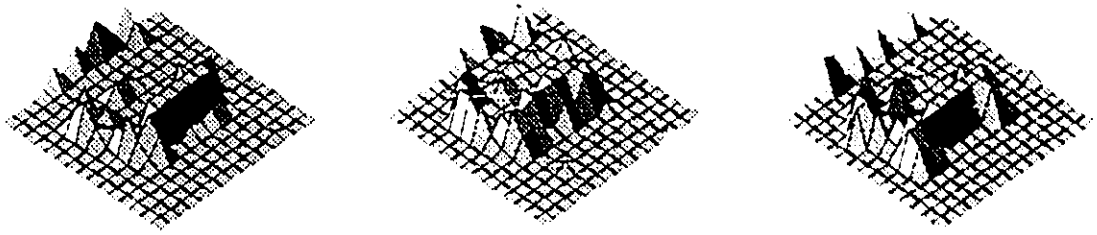
Examples of patterns correctly classified as "**D**"



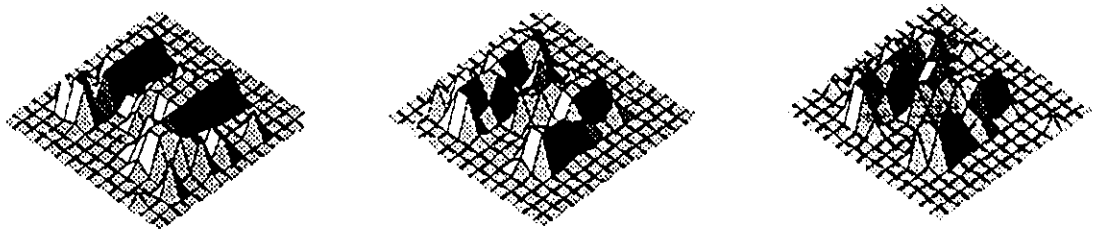
Examples of patterns correctly classified as "**E**"



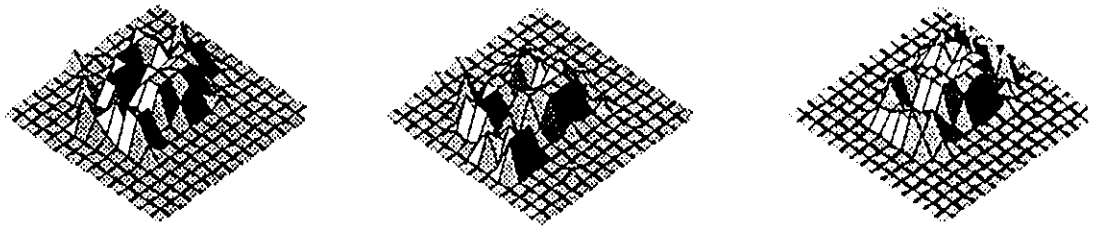
Examples of patterns correctly classified as "**I**"



Examples of patterns correctly classified as "**L**"



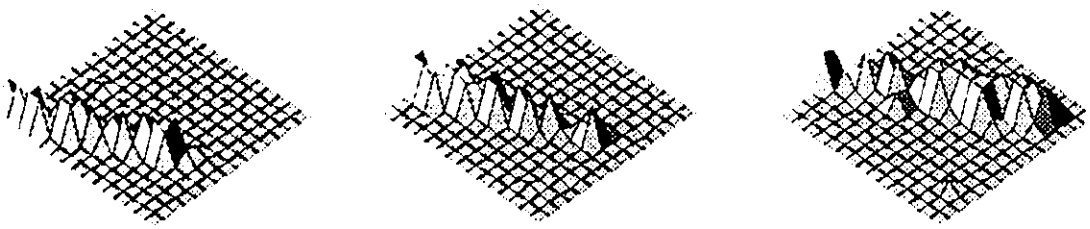
Examples of patterns correctly classified as "**M**"



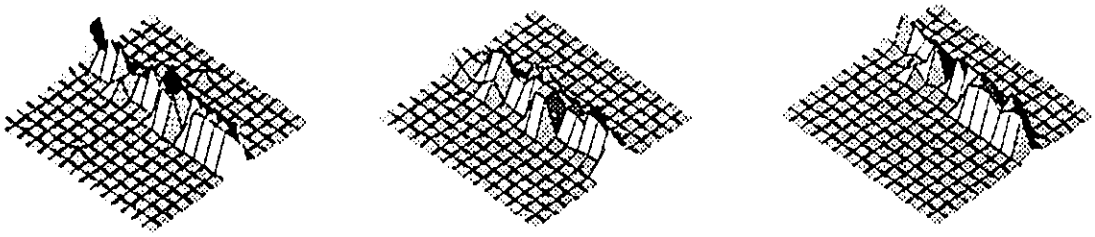
Examples of patterns correctly classified as "**S**"

Examples of patterns correctly classified as "**T**"

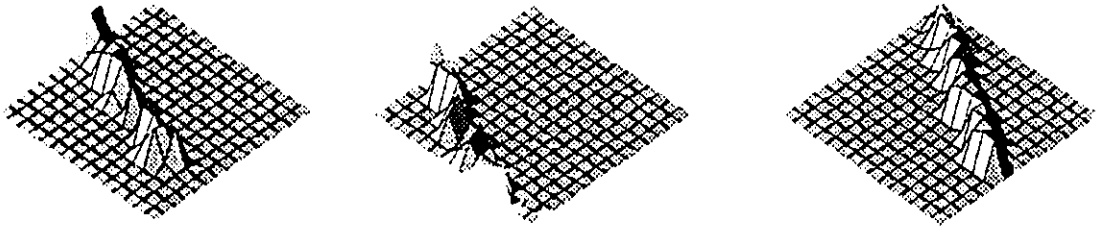
Lines Network - Selection of correctly categorized patterns from the test set



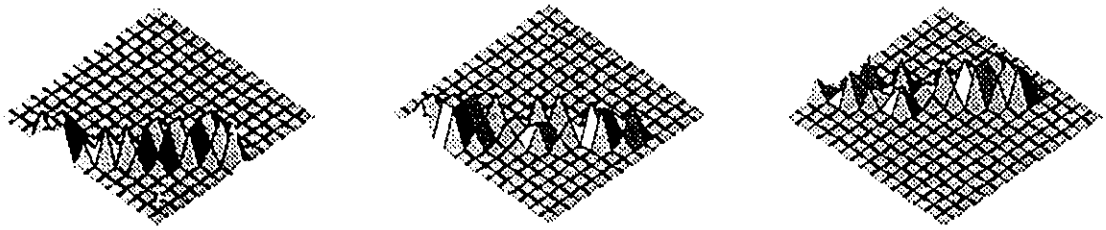
Examples of patterns correctly classified as "-22.5 Degrees"



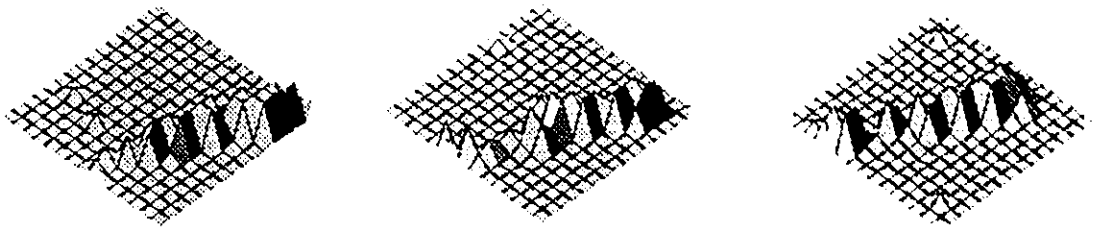
Examples of patterns correctly classified as "-45 Degrees"



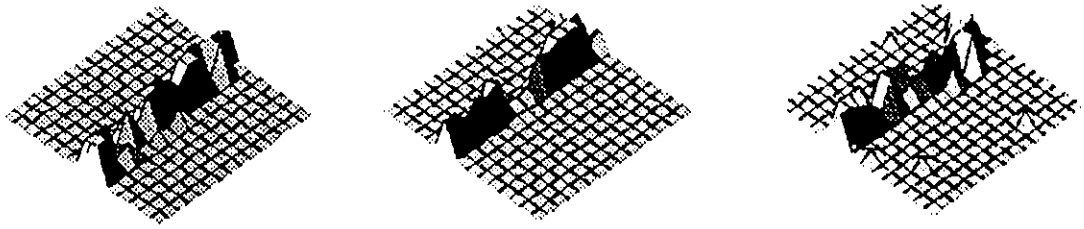
Examples of patterns correctly classified as "-67.5 Degrees"



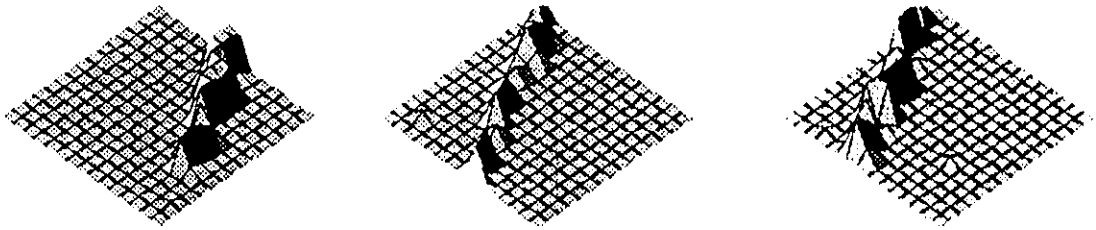
Examples of patterns correctly classified as "0 Degrees"



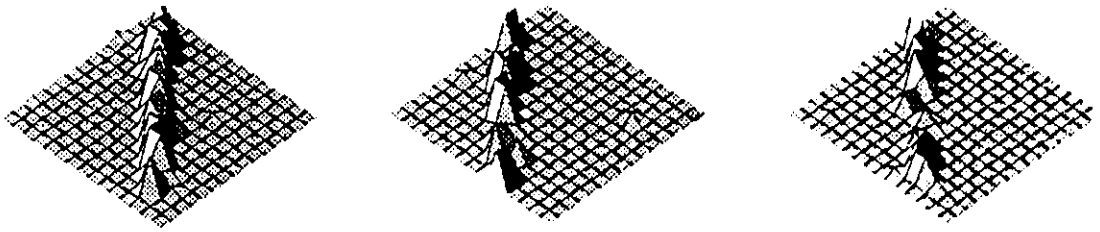
Examples of patterns correctly classified as "+22.5 Degrees"



Examples of patterns correctly classified as "+45 Degrees"



Examples of patterns correctly classified as "+67.5 Degrees"



Examples of patterns correctly classified as "+90 Degrees"