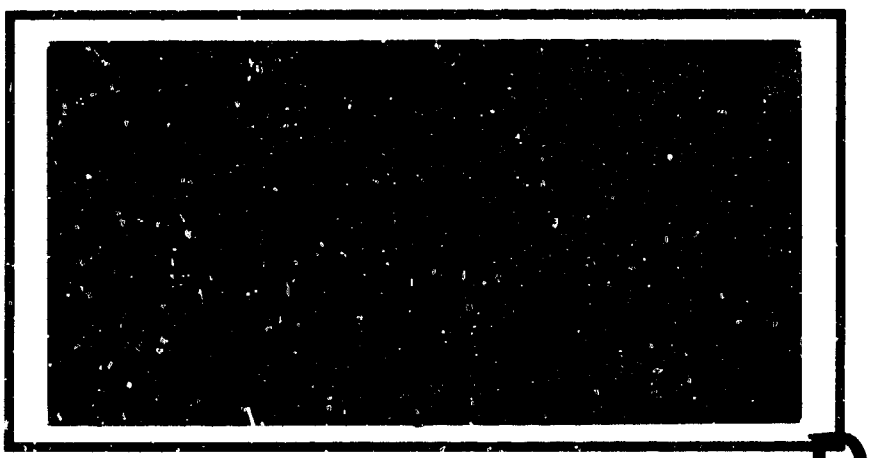
1990-12

A Biologically-Inspired Neural Network Architecture for Image Processing

Lazofson, Laurence E.



DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GEO/ENG/90D-04

A BIOLOGICALLY-INSPIRED NEURAL NETWORK ARCHITECTURE FOR IMAGE PROCESSING

THESIS

Laurence Edward Lazofson
Captain, USAF

AFIT/GEO/ENG/90D-04

Approved for public release; distribution unlimited.

AFIT/GEO/ENG/90D-04

A BIOLOGICALLY-INSPIRED NEURAL NETWORK ARCHITECTURE FOR IMAGE PROCESSING

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Electrical Engineering

Laurence Edward Lazofson, B.S., B.S.E.E.
Captain, USAF

December, 1990

Preface

Tackling this thesis project required a search for knowledge among several areas previously beyond my grasp. These included neurophysiology, mathematics, pattern recognition and artificial neural network culture, and software development. Thanks to Dr. Steven K. Rogers, Dr. Matthew Kabrisky, Captain Dennis Ruck and Captain Greg Tarr, I had access to expertise in all of these disciplines. I would never have conceived of this research project, let alone attempted a crack at it, without their guidance. I am further indebted to my wife, Tonda, for her support and understanding throughout my pursuit of the degree requirements. This experience has immeasurably improved my knowledge of science and engineering, also leaving me with an appreciation for many complex concepts beyond my comprehension. As expected, earning this MSEE also entailed plenty of grief and aggravation. In fact, I almost turned down this rare educational opportunity before it began, until I pondered its monetary cost. It was free.

Laurence Edward Lazofson

Table of Contents

List of Figures

Abstract

This thesis project included a literature survey of biological and artificial neural network research followed by development and testing of high-order and image recognition hierarchical neural network algorithms. Following training, performance testing of second-order and third-order networks yielded maximum accuracies comparable to those achieved by multilayer perceptron classifiers operating on test data sets. Several versions of an image classification algorithm were tested for learning performance using pixel data from forward-looking infrared (FLIR) images of tanks, trucks, target boards, and clutter. Employing the biologically-motivated Lambertization and contrast normalization of pixel windows, correlations with multiple Gabor function wavelets, and a "phase synchronizing" local averaging routine, the image classification network extracted data features. Different network versions fed the extracted features to varying output classification schemes. To improve separation of problem classes, recommendations were made for varying the parameters of the Gabor function wavelets and modifying the phase synchronization scheme to extract more suitable features from image pixel data.

A BIOLOGICALLY-INSPIRED NEURAL NETWORK ARCHITECTURE FOR IMAGE PROCESSING

I. Introduction

1.1 Summary of Current Knowledge

It is widely known that in the striate (visual) cortex, certain key features of visual image data are segmented following mapping from the retinas. These separately distinguished aspects of visual information include intensity (of three colors), motion, binocular disparity, and texture (14). Texture is essentially characterized by spatial frequency, a term often used to describe both pitch and orientation of edges. In other words, texture consists of localized sinusoidal grating patterns, or intensity functions, with varying periodicity and angular orientation in the 2-D visual field (13). Figure 1 illustrates this concept.

Seeking a better understanding of cortical processes, neurophysiological researchers have investigated the functioning of cortical tissue. As a "crumpled sheet" of neural mass, the mammalian cortex is divided into *cortical columns.* This 2-D array of adjacent cortical columns functionally resembles a honeycomb (13). In the visual cortex (Area 17), each cortical column is selectively tuned to respond to a different aspect of segmented visual data. Blasdel and Salama employed "voltage-sensitive dyes" to observe "detailed maps of orientation selectivity," or functional organization of cortical columns, in the visual cortex of macaque monkeys (2).

Experimental observations of neuronal functioning in the visual cortex are used in this thesis to develop a biologically-motivated artificial neural network system for

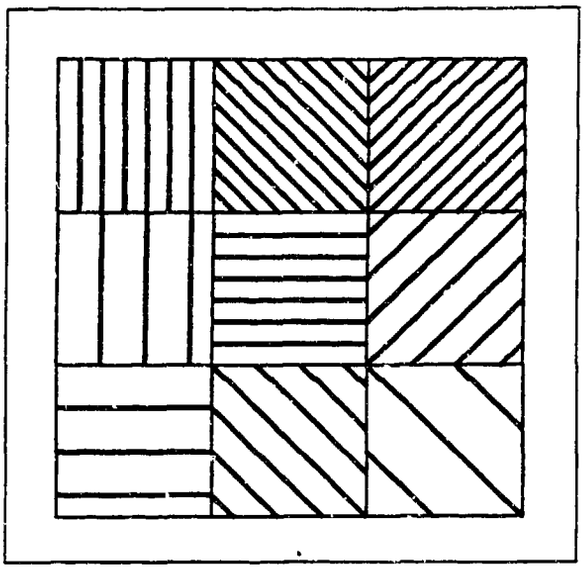


Figure 1. 2-D Windowed Patterns of Varying Periodicities and Orientations

image processing. Observed neuronal processes highlighted in this research included stimulus-specific responsiveness (4, 9, 10, 11, 12, 26), phase synchronous firing of groups of neurons (4, 9, 26), the semblance of Gabor functions in neuronal receptive field plots (4, 11, 12), and "axo-axonic interconnections" (19) in biological neural networks. The artificial neural network system developed in this thesis emulated the characteristics of these observed biological phenomena. Details are discussed in Chapter II.

1.1.1 Neuronal Stimulus Specificity Ervin, Gray and Singer, Hubel and Wiesel, and Jones, Stepnoski, and Palmer independently observed stimulus-specific responsiveness of neurons in cat visual cortex (4, 9, 10, 12). Each research team varied stimuli exposed to the visual field of test subjects and measured neuron firing outputs. One unified result from these researchers seems to verify that the visual cortex is an organized collection of columns, with some of these columns sensitive to specific orientations of a stimulus (14).

1.1.2 Phase Synchrony Gray and Singer further observed *synchronous oscillations* of stimulus-specific neurons in the visual cortex of cats (9). These findings indicate a possible cortical mechanism that enables neurons responding to unique stimuli to simultaneously broadcast their message to higher cortical levels (26). Such a mechanism facilitates the "binding" of local stimulus details into larger, global feature aspects. Attempting to mathematically model these findings, Kammen, Holmes, and Koch proposed and tested two algorithms for phase locking artificial cortical columns (15). These concepts are discussed in greater detail in Chapter II of this thesis.

1.1.3 Gabor Functions Modeling Neuronal Responses Several biological researchers also recorded neuronal receptive field responses that mapped to the general form of Gabor functions (Figure 2) (4, 11, 12). This may be a highly significant finding, as a Gabor elementary function might model a transfer function of a neuron being stimulated within a linear region of operation (14).

An elementary Gabor function consists of a sinusoidal function multiplied (or windowed) by a Gaussian function. A Gabor function is consequently characterized by the frequency of the sinusoidal cofactor and the variance of the Gaussian cofactor. The Gaussian function localizes, or truncates, the sinusoid. The sinusoid discriminates frequency. (1)

Independent research by Ayer and Fretheim implemented 2-D spatial Gabor functions in segmenting image features. Both researchers correlated image scenes with multiple Gabor functions of differing spatial frequencies and variances (1, 6). It is possible that the visual cortex employs a similar correlation process in discerning texture information on an image (14).

1.1.4 "Axo-Axonic" Neuronal Connections In addition to the well-known "axo-dendritic" neural connection model depicted in Figure 3, neurophysiological research also indicates the existence of "axo-axonic" synapses in biological neural

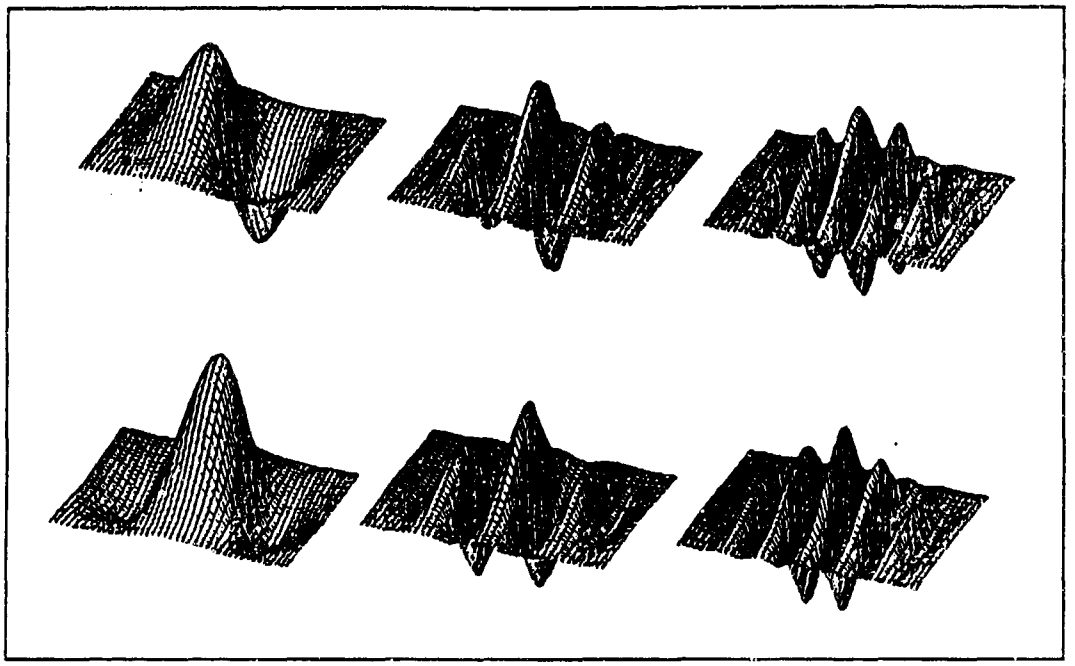


Figure 2. Examples of 2-D Gabor Functions (1:3)

networks (19:2). These connections suggest processing interactions, or modulations, among neuron outputs. The "axo-axonic" connection serves as a model for high-order artificial neural networks discussed later in this document (19).

1.1.5 Theory The pursuit of this thesis work required tandem consideration of previously developed artificial neural network algorithms and the cited biological observations of neural functioning in cat visual cortex. In the visual cortex, phase synchronous firing of stimulus-specific groups of neurons may indeed temporally unite their conveyed messages. This would enable simultaneous summation and processing of these dynamic neuronal responses at higher cortical levels (26). However, in implementing a *software model* of a network, computer memory storage essentially freezes time by holding calculated numerical outputs of artificial neurons. A subsequent step then sums these stored values in other artificial neurons (typically in a higher layer). Software modeling thus removes the temporal aspect of neuronal oscillatory firing by storing numerical output values that represent firing

rate. This skirts the need for synchronizing outputs of software neurons, whereas phase locking phenomena seem crucial in biological neural networks with transient oscillatory outputs. Similarly, it is questionable whether biologically measured temporal latency, an aspect of phase delay, would prove valuable in software models of neural networks (23). This concept is discussed further in Chapter II.

Results from cited researchers also allude to neuronal stimulus specificity that functionally responds in the semblance of unique Gabor spatial mappings. It is widely believed that neurons in the visual cortex operate as Gabor detectors, responding to unique combinations of pitch and orientation to segment localized image textures (14). This implication influenced the direction of this thesis work.

1.2 Background

1.2.1 Artificial Neural Networks For several decades, researchers have been developing and testing a variety of artificial neural network algorithms for use on certain classes of problems. These algorithms, whether supervised, unsupervised, or hybrid models, are capable of "learning" functional relationships among data sets. Network learning, also referred to as "training," occurs as the network is presented with examples of vector feature data. For some classes of networks, the training process drives iterative mathematical adjustments of node connection "weights," or functional multipliers, until network convergence. Following training, a viable network should be able to "generalize," or solve a problem, with new, unknown data inputs. (23)

Unlike conventional electronic digital computer serial processing, neural network algorithms employ densely interconnected nodes modeling massive parallelism. As seen in biological neural networks, massively parallel architectures yield a more robust system, characterized by graceful degradation rather than being prone to total failure due to one malfunctioning computational element (18, 23). Ultimately, software-developed neural networks may be holographically based or implemented

electronically on silicon chips.

Emulating some of the observed aspects of biological neurons (Figure 3) and neural networks, artificial neural networks have been designed to solve classification, identification and optimization problems, perform functional estimation and prediction, enhance signal-to-noise ratios, and serve in a variety of other applications (3, 23). Image and speech recognition and prediction of chaotic time series are specific examples of potential applications of artificial neural networks.

1.2.2 High-Order Artificial Neural Networks High-order neural network algorithms have been shown to surpass the performance of "classical" multilayer feedforward networks for some applications (8, 19, 20). By feeding polynomial functional combinations of vector data components into a *single layer* of artificial neurons, these networks may efficiently "carve out" complex, functional decision regions in multidimensional feature spaces. High-order networks can thus tackle problems that are not linearly separable. Compared to multilayer first-order networks, tests of these high-order models indicated faster convergence and greater probability of attaining solutions for some problems (8, 19, 20).

Multilayer first-order networks (Figure 4) also functionally adapt to nonlinearities by implicitly implementing higher orders in their hidden node layers. However, the added network layers and additional weight modifications may yield slower, and less probable, convergence to solutions. For these reasons, high-order network algorithms have been pursued in solving more complex, nonlinearly-separable problems. (8, 21)

1.3 Problem Statement

There existed a need to perform a consolidated literature review of biological and artificial neural network research. Incorporating the concepts and phenomena culled from this integrated literature survey, there was a subsequent need to expand

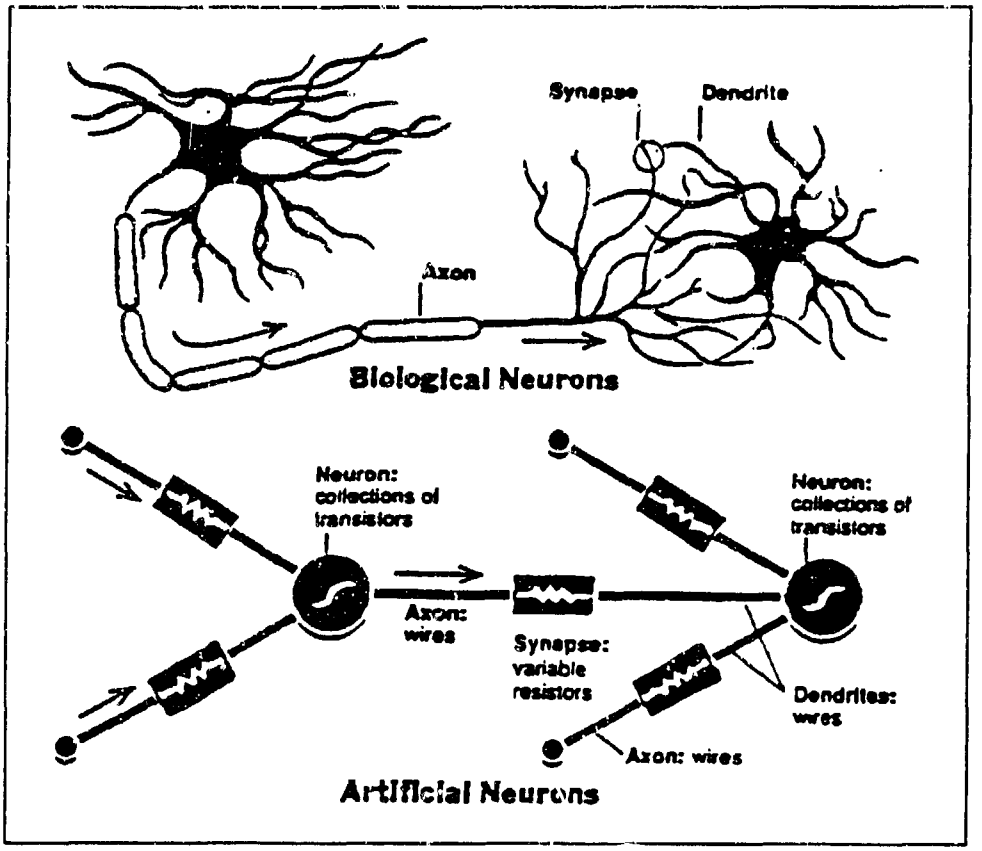


Figure 3. Emulating the Biological Neuron (5:8)

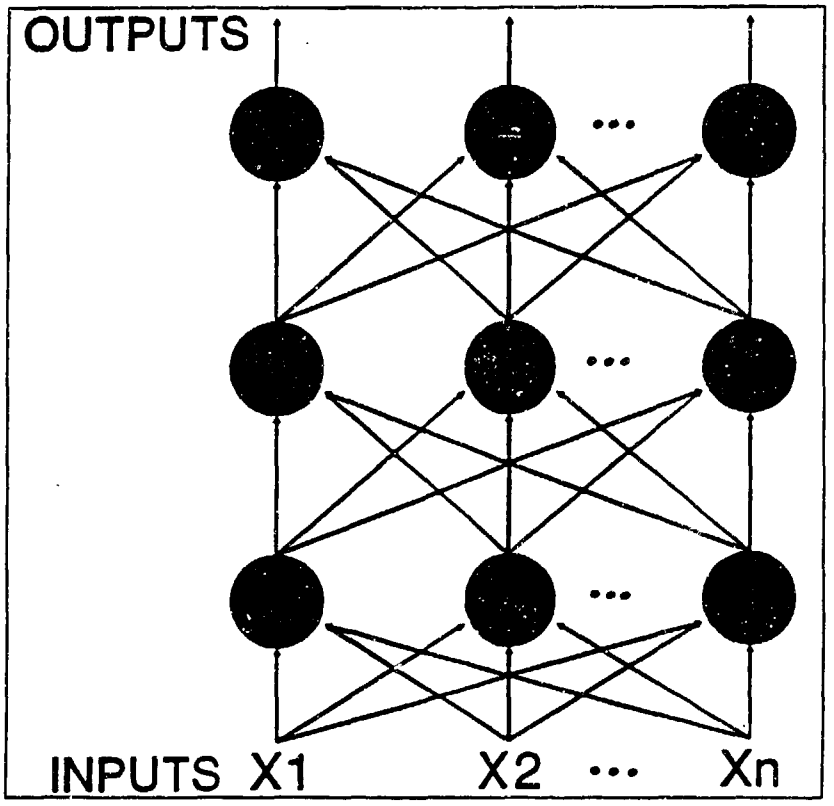


Figure 4. Example of Multilayer Network of Artificial Neurons (23)

the state of the art of existing neural network algorithms.

1.4 Objective

This thesis work sought to survey, integrate, and document a subset of biological and artificial neural network research. Having initially reconciled this body of knowledge, this thesis subsequently aimed to develop and test a software model consistent with the cited research findings. This entailed emulating some of the biological observations of cortical functioning. The intent was to employ high-order networks and implement Gabor functions and the concept of phase synchrony to create a better model for solving certain classes of engineering problems, including image processing. The model ultimately employed Lambertization and contrast normalization of windowed image regions, correlations with multiple Gabor functions, a phase synchronizing local averaging routine, and a feedforward network classification layer. Details are covered in Chapter III.

1.5 Scope and Limitations

This work did not attempt to provide a comprehensive, unified explanation of cortical processing. Rather, this thesis intended to describe and emulate some of the observed characteristics of neuronal functioning in the visual cortex, and extend previously developed artificial neural network models. This work did not address the validity of biological observations and proposed cortical phenomena by cited researchers. This thesis did not assess whether some of the described cortical measurements may have been artifacts.

This research focused on a *subset* of observed neural characteristics, specifically citing work by Ervin (4), Gray and Singer (9), and Jones, Stepnoski, and Palmer (11, 12). Attempting to unify the biological observations of these researchers, this thesis subsequently sought to develop an artificial neural network model consistent with the cited aspects of neurophysiology. This model was related to current network

algorithms and applied to the target recognition problem for forward-looking infrared (FLIR) images.

1.6 Definitions

1-D One-dimensional.

2-D Two-dimensional.

Artifact An observed measurement *not* related to a process or phenomena of interest, typically resulting from a random or systematic error in experimental or measurement procedures.

Artificial Neural Network (Net) Software or hardware emulation of simple, observable characteristics of biological neural networks.

Artificial Neuron A computational element that mathematically models a biological neuron.

Chaotic Time Series A deterministic function that is seemingly random (23).

Gabor Function A sinusoidal function multiplied (or windowed by) a Gaussian function, characterized by the frequency of the sinusoidal cofactor and the variance of the Gaussian cofactor (1).

Neuron Nerve cell.

Node A single computational element of a neural network.

Receptive (Visual) Field The visual region over which a specific neuron responds (in the visual cortex) (10:109).

Segment To distinguish features or aspects of interest from background.

Spatial Frequency Pitch (periodicity) and angular orientation of a function or structure of interest, such as an image.

Striate Cortex Section of cortex that receives mapped visual information, also known as visual cortex, V1, or Area 17 (14).

Weights Multiplicative factors between nodes or between inputs and nodes, analogous to the strength of synaptic connections between neurons (17). Also called "connection weights."

Wetware Biological cell computational elements (23).

1.7 Sequence of Presentation

This introductory chapter discussed the problem and background concepts for this thesis. Chapter II provides the literature review for this research. Chapter III presents the research approach, discussing methods and algorithms. Results and recommendations are covered in Chapter IV and Chapter V, respectively.

II. Literature Review

Several of the published research findings cited in this thesis depict various experimental observations of neuronal processes in the striate (visual) cortex. These observations of apparent cortical functioning include stimulus-specific responsiveness and phase synchrony of neuronal firing, the semblance of Gabor functions in characterizing neuronal receptive fields, and "axo-axonic interconnections" (19) in biological neural networks. A key aspect of this thesis work, addressed in this chapter, aimed to reconcile the different cited experimental observations. The subsequent research goal sought the development of biologically-motivated neural network algorithms for solving certain classes of engineering problems. For this thesis work, the problem of interest was image processing.

2.1 Neurophysiological Research

2.1.1 Ervin Research published by Ervin in 1965 (4) provides a solid basis for stimulus specificity of neuronal response, phase synchronicity in groups of firing neurons, and use of Gabor functions in characterizing neuronal receptive field profiles. Ervin's paper helps unify these aspects of cortical processing into a coherent view supported by more current research articles. For this reason, his relatively early discoveries are discussed along with recently published research. Ervin's experimental observations followed Hubel and Wiesel's widely acknowledged research characterizing neuronal receptive fields and firing responses in cat visual cortex (10).

Using a computer display to generate a variety of visual stimuli in front of the eyeball of an immobilized cat, Ervin experimentally tracked response outputs in the cat's visual cortex. This was done with a microelectrode and a gross electrode, both inserted into the visual cortex and positioned relatively close together. Limited in size, the microelectrode measured firing output of a single neuron responding to each visual stimulus. The larger gross electrode, piercing through many neurons in

the surrounding cortical region, measured the average evoked response to the same visual stimulus. Data were stored by computer. (4)

Unfortunately, Ervin's article did not explicitly state whether or not the cat was anesthetized. In assessing the validity of measured neuronal responses, this may be a significant factor, as neurons in an anesthetized subject might not display typical behavior (13).

While monitoring the firing rate of a single neuron with the microelectrode, Ervin initially measured the neuron's output response as a function of position in its visual receptive field. A small spot of light, analogous to an impulse input, was shifted point by point across the visual field. The neuronal firing rate was measured for each field position of the constant-intensity spot of light (4). Ervin employed this initial procedure to locate the "visual field center" (4:37), pinpointing the location in the neuron's receptive field where the light spot induced the greatest firing response. Over twenty years later, Jones and Palmer performed a similar experimental procedure to characterize the spatial receptive field response of neurons in cat striate cortex (11).

Having located the visual field center, Ervin employed it as a center point for visual stimuli. Applied stimuli included lines of varying angular orientation, circles, squares, and triangles of different sizes, and multiple point patterns. Each unique stimulus flashed before the cat's eye for a duration of 400 *ms*. The microelectrode and gross electrode recorded the firing responses for each test neuron and its surrounding region, respectively. (4:36–38)

For each specific stimulus, Ervin plotted the individual neuron's firing rate, the average evoked response of the area surrounding the neuron, and the stimulus duration, all versus time (Figure 5). Actually, he divided the time axis into discrete 20 *ms* interval bins. For each interval, the number of firings of the test neuron were counted and plotted (4:39–40). The resulting histogram effectively conveys firing frequency versus time, though in an indirect way.

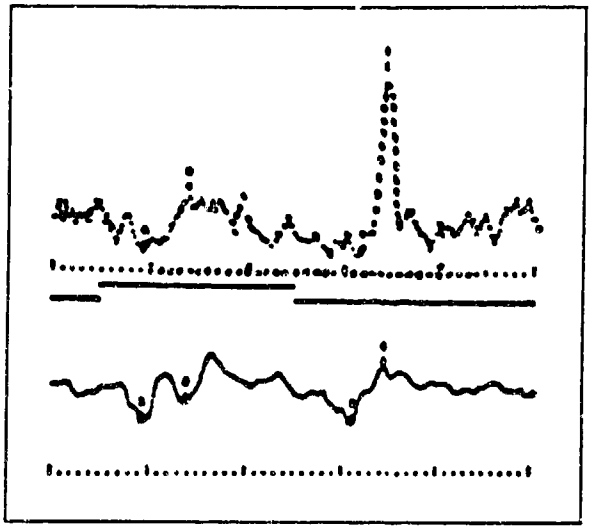


Figure 5. Neuronal Firing Histogram (Top), Stimulus Duration (Middle), and Average Evoked Response (Bottom) Plots with 20ms Bins (4:40)

Comparisons of the resulting one-dimensional plots revealed a startling discovery. A myriad of different plot patterns for varying stimuli indicated stimulus-specific responses in neurons (4, 9, 10, 11, 12, 26). Each stimulus generated a unique pattern of "phasic," "tonic," and inhibitory components along the time scale (4:40). As used by Ervin, phasic components referred to relatively transient peaking points on the firing rate plots, while tonic components described plateau regions (4). Ervin's observations of stimulus specific responsiveness followed previous work by Hubel and Wiesel, who claimed "shape, position, and orientation" (10:151) of stimuli affect neuronal firing responses.

Ervin conveyed another crucial concept by juxtaposing the plotted response of each neuron with the average evoked response of its surrounding region. In general, the plot patterns for both the individual neuron and the average evoked response were similar (4:40). Figure 5 conveys this idea. This may indicate synchronous oscillatory activity of groups of neurons (4, 9, 26). Such an inference, if true, may be of tremendous significance in understanding higher cortical processes. Recent research by Gray and Singer further suggested this phenomena (9).

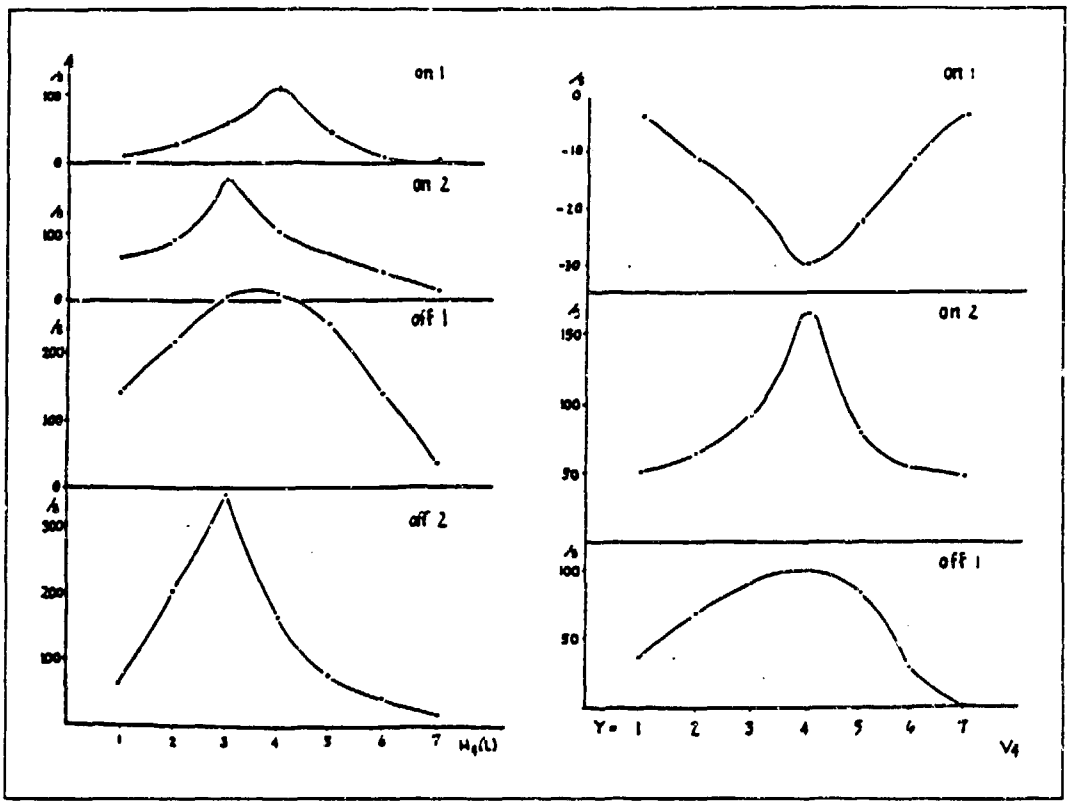


Figure 6. 1-D Plots of Neuronal Receptive Field Responses (4:43)

Similar to his procedure for initially determining a neuron's visual field center, Ervin collected additional data to spatially characterize a neuron's entire receptive field for different phasic components of its firing response. This process essentially froze time at the occurrence of a phasic component on a neuron's plot of firing rate versus time. Firing rate was then plotted as a function of spatial position across the neuron's 2-D visual receptive field (4:42–44). Ervin illustrated this relationship on both 1-D and 2-D spatial coordinate plots (Figure 6). It is interesting to note that *these plots resemble the envelopes of 1-D Gabor functions.* This finding is in accord with later work by Jones and Palmer (11). Ervin observed that different phasic components (early and late) resulting from a single stimulus produced unique receptive field response mappings (4:42).

Ervin's plots of firing rate versus time also indicated that responses to simultaneous stimuli at multiple points in a neuron's receptive field did not necessarily sum (4:45). This suggests that, lacking the property of superposition, a neuron is not obliged to respond linearly to stimuli. This should preclude unrestricted use of a previously characterized impulse response to predict cell, or system, output (13, 21).

Depending on a given neuron's "choice" of specific stimulus, however, it *might linearly add* the correct combination of inputs (13, 21). For example, a neuron with a "preference" for edges at a given orientation may respond linearly to two points of light extrapolating to a line following this "preferred" orientation. Otherwise, if two light points form a segment of different orientation in the visual field, this same neuron may be inhibited and behave nonlinearly. Ervin suggests this idea of orientational preference (4:50). Furthermore, the linearity assumption for neural response may be valid for limited ranges of stimuli (14).

2.1.2 Jones and Palmer In 1987, Jones and Palmer employed a procedure they termed "reverse correlation" (11) to characterize the 2-D spatial receptive field responses of neurons in cat striate cortex. Similar to Ervin's results, their clear, 2-D mappings of the visual receptive field response closely approximated the form of Gabor functions.

Fourteen anesthetized, adult cats were exposed to visual stimuli on an oscilloscope screen. The stimuli consisted of small, rectangular, bright and dark spots presented for varying time durations, typically 50 or 100 *ms* (11). The reverse correlation process electronically stored, and later matched, measured neuron spike outputs with the appropriate stored stimuli generated on the oscilloscope screen (11:1191). Receptive field functions for bright and dark stimuli were then plotted separately (firing response versus spatial field position). To fuse the receptive field plots for both bright and dark stimuli, Jones and Palmer subtracted the dark stimulus output functions from corresponding light stimulus output functions. Thus the dark

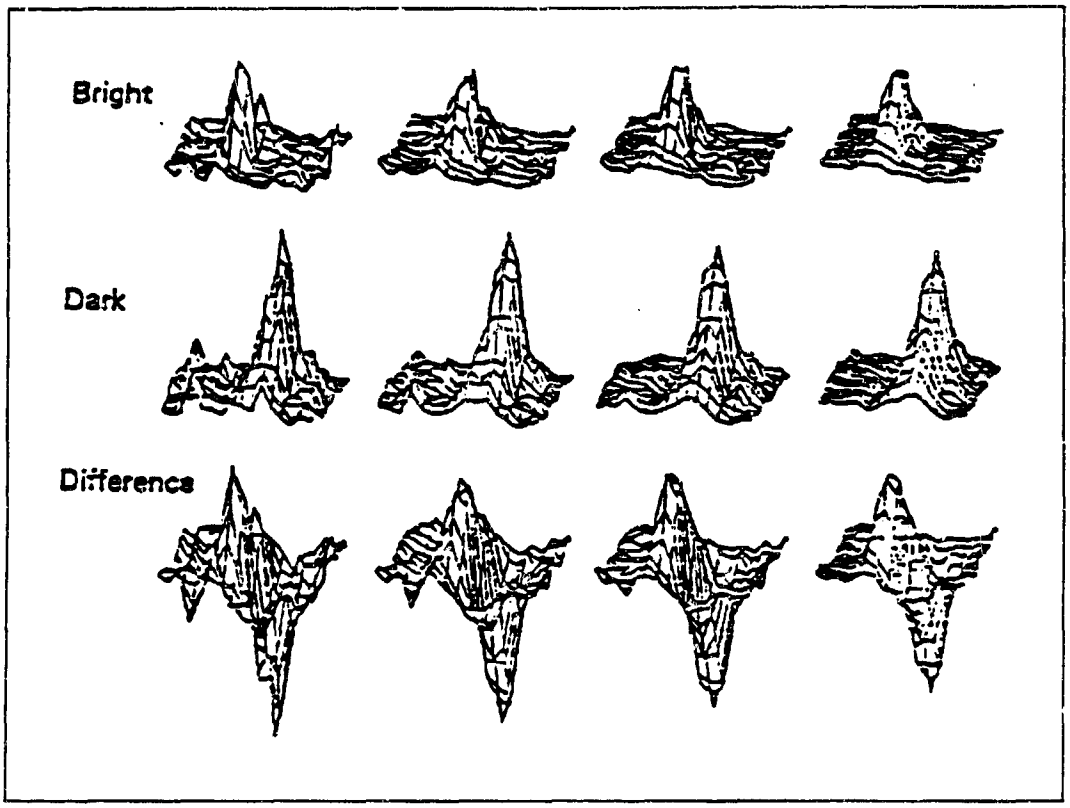


Figure 7. 2-D Spatial Plots of Neuronal Receptive Field Responses (11:1196)

stimulus output patterns emulated inhibition of brightness in the 2-D spatial domain (11:1192). As noted by Jones and Palmer (11), the resulting receptive field response functions appeared to be Gabor functions (Figure 7).

As Jones and Palmer indicated, previous work by other researchers attempting to map neuronal receptive fields with one-dimensional measurements might prove invalid if the overall response functions were not Cartesian separable (11:1188). Rather than independently sweep across length and width axes, Jones and Palmer characterized the receptive fields in two spatial dimensions. This method provided accurate 2-D representations of the field mappings, regardless of their functional Cartesian separability (11:1188). In fact, Jones and Palmer ascertained that approximately half of the neurons investigated displayed 2-D receptive field functions that were *not* Cartesian separable (11:1208).

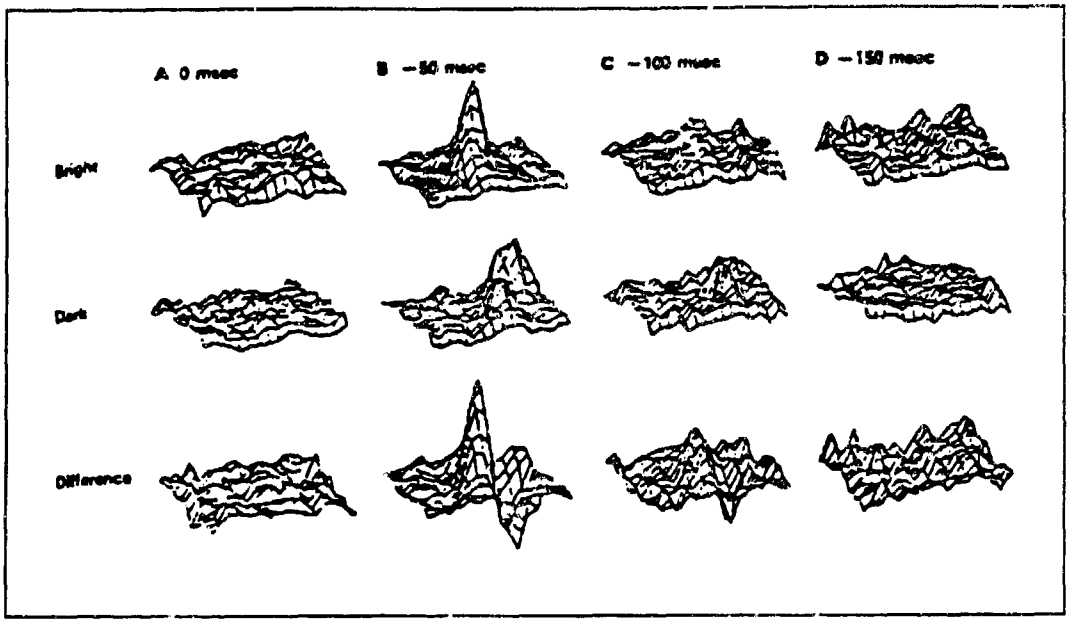


Figure 8. Varying Delay Times for 2-D Spatial Plots of Neuronal Receptive Field Response (11:1197)

Jones and Palmer further employed the reverse correlation process to assess the time delay between stimuli and corresponding output spikes in the test neuron. They varied the delay time at values of 0, 50, 100, and 150 ms in the reverse correlation algorithm. Their graphical results showed that at a time delay of 50 ms after stimulus presentation, a distinguishable functional form of the neuronal receptive field response emerged (Figure 8). At time delays of 0, 100, and 150 ms after stimulus presentation, the receptive field plots depicted mostly noise with no apparent response function. (11:1195–1197)

Jones' and Palmer's observations can be reconciled with Ervin's findings. For a given test neuron and a specific applied stimulus, emergence of a receptive field response, in the form of a Gabor function, occurred 50 ms after stimulus presentation (11:1195–1197). This point in time marked what Ervin called a phasic component, or transient maxima of neuronal firing, indicating the time until appearance of the receptive field function. In light of Ervin's findings, the unique receptive field response profile and its 50 ms delay are both probably unique to the stimulus

presented and the test neuron (4, 11).

It would be interesting to see an extension of Jones' and Palmer's results through continued use of their reverse correlation process beyond a 150 *ms* delay time. Jones and Palmer observed the emergence of a receptive field response, in the form of a Gabor function, 50 *ms* following the visual stimulus. They discontinued the reverse correlation process after a 150 *ms* delay (11:1197). For the applied stimulus, there may yet have existed additional, later phasic components for the neuronal response, as Ervin observed on firing rate versus time plots (4:40). These later components, perhaps emerging at delays of 200 *ms* and beyond, may give rise to different receptive field response mappings, although still perhaps in the general form of Gabor functions. It is possible that different combinations of test neurons and stimuli might yield unique delay times for one or more phasic components, each with unique 2-D Gabor field mappings. Extending the reverse correlation procedure in this manner could confirm Ervin's observations and expand understanding of stimulus specificity in neuronal responses.

2.1.3 Jones, Stepnoski, and Palmer Beyond the previously described research characterizing 2-D spatial receptive field responses of neurons, Jones, Stepnoski, and Palmer examined the 2-D *spectral* response of 36 neurons in cat visual cortex (12:1212). Their experimental procedure was similar to that of the Jones and Palmer experiment cited earlier. In this experiment, however, the visual stimuli were sinusoidal gratings drifting across the neuronal receptive field (12:1212).

Varying the spatial frequency and orientation of these 2-D sinusoidal intensity functions, Jones *et al* measured and graphically depicted neuronal response (spiking) as a function of spatial frequency and orientation appearing in the visual receptive field (12). Jones *et al* noted that a cortical cell's response to the drifting gratings yielded "a rectified sinusoidal modulation of the spike frequency" where "the degree of rectification varied from cell to cell, but for each cell, the form of

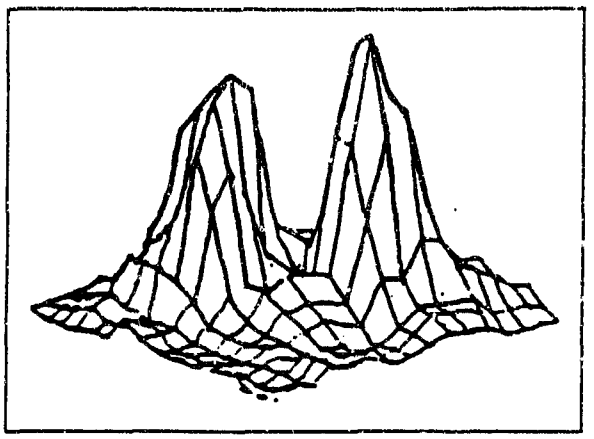


Figure 9. 2-D Spectral Plot of Neuronal Receptive Field Response (12:1222)

the response was constant irrespective of stimulus spatial frequency, orientation, or contrast" (12:1212).

Examining the plots in this article, these neuronal spectral responses appeared in the form of Fourier transforms of Gabor functions, manifested as two, separate 2-D Gaussian functions (Figure 9). This is consistent, since Gabor functional forms in the *spatial* receptive field should correspond to Fourier transforms of these functions in the *spectral* domain. Mathematically described, a sinusoid multiplied by a Gaussian function Fourier transforms to two impulses in the frequency domain convolved with a Gaussian (22). This convolution yields two Gaussians. Figure 10 illustrates this mathematical process. It should also be noted that the spectral response plots were polar and were obtained assuming polar separability of the orientation (ϑ) and spatial frequency (ρ) one-dimensional functions (12:1223).

These results further support the idea of stimulus specificity of neurons in the visual cortex (4, 9, 10, 11, 12, 26). Responses to varying sinusoidal grating stimuli indicate that neurons segment on the image features of spatial frequency and orientation (12). Jones *et al* apparently define spatial frequency as pitch distance between edges. In this definition, they do *not* include the angular orientation of the parallel grating edges, considering orientation a separate feature (12).

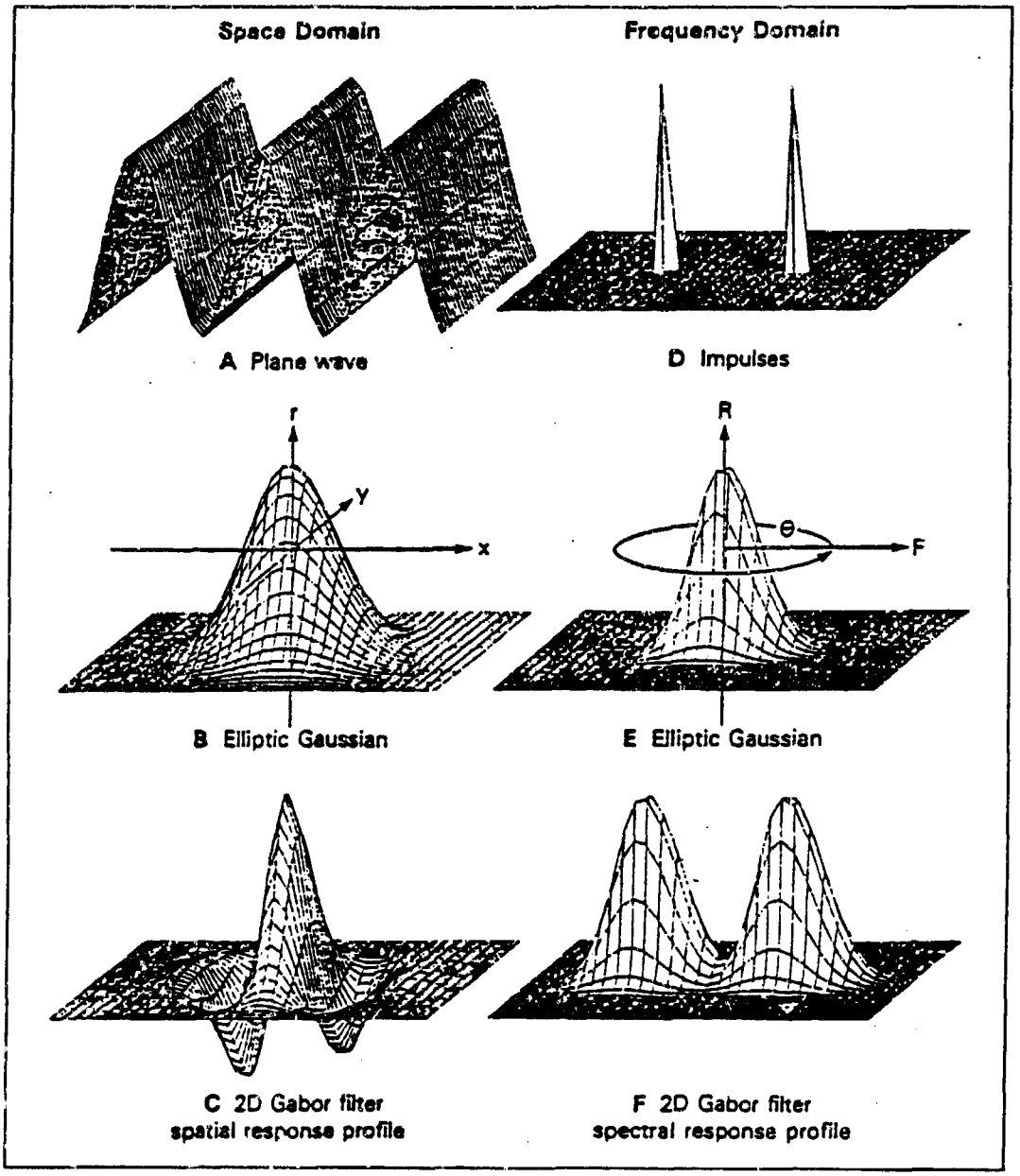


Figure 10. 2-D Plots Depicting Fourier Transforming of a Gabor Function; F Models Neuronal Spectral Response (6)

2.1.4 Gray and Singer In 1988, Gray and Singer documented experimental data suggesting that "local neuronal populations in the visual cortex engage in stimulus-specific synchronous oscillations" (9:1698). Their findings, in accord with other research cited here, were obtained from cortical measurements on 15 adult cats and 12 kittens, all anesthetized (9:1698). Stimuli exposed to the visual receptive fields consisted of illuminated line segments at varying orientations, velocities, and direction of movement (9:1699).

Gray and Singer observed stimulus-specific responses in the test neurons (9). Perhaps even more impressive was their observation that, for a given stimulus, "adjacent neurons" fired "simultaneously and in synchrony" (9:1701). Equally significant was their observation that different groups of firing neurons, "spatially separated" (9:1702) on the cortex by up to 7 mm, oscillated in phase, provided they selected for the "same orientation specificity" (26:298).

Assessing Gray and Singer's results, Stryker summarized, "receptive fields of the neurons at the two sites had a common orientation specificity and were aligned so that they could be stimulated by a single long bar of light" (26:298). Stryker further suggested that the "global property of the stimulus" (26:298) affected firing correlations between spatially separated groups of neurons. He concluded this since one long light bar, rather than two separate light bars of the same orientation, yielded synchronous firings of spatially separated cortical areas (26:298). Evidently, the two separate bars "did not bridge the gap between the two receptive fields" (26:298).

Gray and Singer suggested that "adjacent neurons" firing "simultaneously and in synchrony" for a specific stimulus are "confined approximately to a single orientation column" (9:1701). They also stated that, based on measurements with multiple electrodes, "synchronization across spatially separate columns does occur" (9:1702). Stryker deduced that many "neurons in the visual cortex," responding simultaneously to the same stimulus, can better convey their message to higher cortical areas by broadcasting "in unison" (26:297). Such phenomena may provide ideas for an

enhanced neural network model as "the phase of the oscillatory response may be used as a further dimension of coding" (9:1702). The concept of phase synchrony essentially "binds" local stimulus details into larger, global feature aspects.

2.2 Biologically-Motivated Neural Network Models

2.2.1 Kammen, Holmes, Koch Based on Gray and Singer's observations of cortical processes, Kammen, Holmes, and Koch subsequently proposed two algorithmic models to mathematically simulate phase locking among groups of neurons (15). Their first model incorporated a "one-dimensional array" of symbolic cortical columns, each "coupled" to its two "nearest neighbors" (15:I-182). Their second model connected each cortical column to "a common comparator which feeds back a function of the average phase" (15:I-182). See Figure 11 for these representations.

Analyzing their first model, the one emulating "nearest neighbor couplings," Kammen *et al* determined that it did not successfully phase lock mathematical inputs (cortical columns) "separated by large numbers of inactive (unstimulated) units" (15:I-182). Increasing the number of cortical units, or lengthening the one-dimensional array, can be expected to exacerbate the problem. It is apparent that this algorithm did not accurately model the biological phenomena observed by Gray and Singer (15).

The second model, however, successfully coupled the phase and firing frequency of the excited (stimulated) cortical columns, and moved them out of phase (and off frequency) with the unstimulated columns (15:I-182). More succinctly, Kammen *et al* determined that "not only do the excited units fire at the same rate, but they remain *exactly* in phase *regardless of their geometrical arrangement or separation*" (15:I-182). Recall that this model implemented a common comparator for average phase feedback (15:I-182). These results offer further direction for a phase-synchronous, enhanced hardware neural network model.

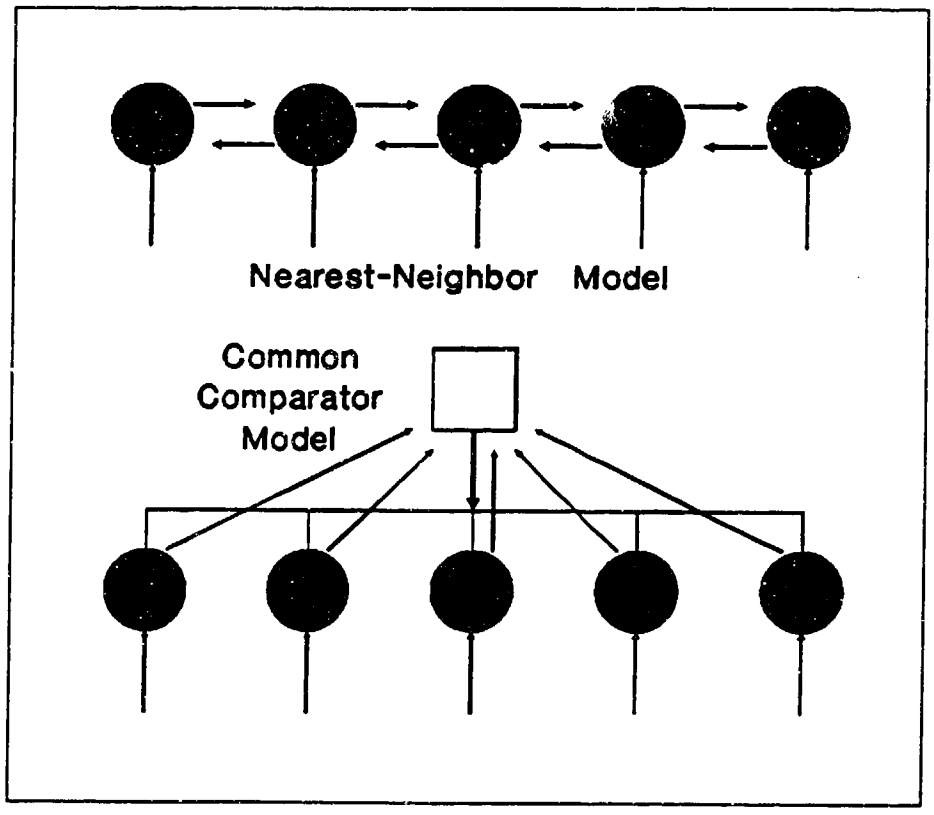


Figure 11. Phase Synchronous Models: Nearest Neighbor and Common Comparator (15:I-183)

2.2.2 Daugman Recognizing the importance of Gabor functions in modeling neuronal receptive field responses, Daugman developed a three-layer neural network to output Gabor transform coefficients for input images. The image inputs were 2-D intensity functions mapped across an array of pixels. (3)

According to Daugman, typical images are characterized by localized regularities in texture, brightness, and edge continuation. These consistencies make such images good candidates for minimizing their coding information. Daugman's neural network accomplishes this by transforming a 2-D pixel array intensity function into Gabor coefficients. This algorithm reduces the amount of electronic data storage needed to describe and reproduce an image. (3)

2.2.3 High-Order Artificial Neural Networks

2.2.3.1 Basic Theory High-order networks feed vector inputs via high-order polynomial functions into a *single layer* of artificial neurons. Compared to standard multilayer networks, high-order networks more efficiently "carve out" complex, functional decision regions within feature spaces for some problems. Standard first-order backpropagation networks implicitly implement higher orders in their hidden layers of artificial neurons. However, the standard feedforward networks' added layers and additional weight modifications may result in slower convergence toward problem solutions. (8, 21)

2.2.3.2 Meador Assuming the existence of axon-to-axon connections in biological neural networks, Meador developed single-layer, high-order, artificial neural networks to model modulations, or multiplicative interactions, among axons. Meador indicated that axo-axonic interconnections are physiologically documented, yet have not been widely discussed in artificial neural network literature. For relative simplicity, Meador limited the scope of his models to implement the widely recognized "axo-dendritic" and the lesser acknowledged "axo-axo-dendritic

synapses." (19)

Limiting his mathematical algorithms, Meador modeled second-order connections in his networks, although higher order connections are possible to implement (28). Meador's first algorithm, an axo-axonic model with adjustable weights for linear and second-order polynomial input functions, appeared as the following equation (19:3):

$$net_i = \sum_{j=1}^{I} d_{ij}(1 + \sum_{k=1}^{I} a_{ijk}x_k)x_j \tag{1}$$

In Meador's equation shown above, $1 \leq i \leq N$, where I is the number of inputs, N is the number of neurons, each x is an input vector component value, d_{ij} the linear connection weights, and a_{ijk} the higher order (axo-axonic) weight factors. Meador revised this equation to an equivalent form he termed "a second-order quadratic interconnect" (19:4):

$$net_i = \sum_{j=1}^{I} D_{ij}x_j + \sum_{j=1}^{I}\sum_{k=1}^{I} A_{ijk}x_k x_j \tag{2}$$

In comparison to the first equational model, this equation is characterized by a different weight space where $D_{ij} = d_{ij}$ and $A_{ijk} = d_{ij}a_{ijk}$ (19:4). To modify the network weights, Meador used a gradient-descent method. This mathematical rule minimized the squared error (difference) between the desired output and the actual output for input vectors.

Testing the high-order networks on three quadratically separable problems (including the XOR problem), Meador found that the axo-axonic weight set converged to solutions faster than the model implementing quadratic interconnect weights. However, both of these single-layer, high-order networks converged faster, and with greater probability, than a standard, multilayer backpropagation network with no high-order inputs (19:15).

2.2.3.3 Tenorio and Lee Similar to Meador's model, Tenorio and Lee developed a "Self Organizing Neural Network" (28). This algorithm modified internal weights, and selected appropriate linear and higher order functions to solve identification and classification problems (28:57). They tested the algorithm with a MacKay-Glass differential equation (28).

Tenorio and Lee did *not* use a mean-square error method for weight adjustment. Instead, supervised network learning was achieved with a "modified Minimum Description Length (MDL) criterion," also referred to as "Structure Estimation Criterion (SEC)" (28:58). This weight modification scheme mathematically selected a network function based on simplicity and best estimate. The transfer function of the network was composed of a combination of linear and higher order functional inputs. (28:58)

As the mathematical basis for their algorithm, Tenorio and Lee cited the Kolmogorov-Garbor polynomial, as shown in the following equation (28:57):

$$y = a_0 + \sum_i a_i x_i + \sum_i \sum_j a_{ij} x_i x_j + \ldots \tag{3}$$

Variables x and y represent system input and output, respectively (28:58). Note the mathematical similarity to Meador's previously cited approach (19). Like Meador, Tenorio and Lee opted to simplify their model by limiting high-order functions to quadratic polynomials (28:58).

To update weights, the model was trained with the following equation for MDL (28:60):

$$MDL = -\log f(x \mid \theta) + 0.5k \log N \tag{4}$$

"where $f(x \mid \theta)$ is the estimated probability density function of the model, k is the number of parameters, and N is the number of observations" (28:60). Tenorio and Lee contended this was a viable model based on successful prediction of the MacKay-Glass equation (a chaotic time series). They also noted the trade-off between level

of complexity and modeling accuracy for the algorithm (28:63).

2.2.3.4 Zhang and Miller Zhang and Miller proposed a high-order network scheme aimed at improving a model developed in 1987 by Heilenberg. Heilenberg's model was designed "to explain how sensory maps could enhance resolution through orderly arrangement of broadly tuned receptors" (29:444). In essence, the model was intended to mathematically emulate the phenomenon of hyperacuity, a condition attained by a system resolving inputs to detail finer than "inter-receptor spacing" (29:444–445).

While Heilenberg's model employed only linear weights, Zhang and Miller included Hermitian polynomial weighting functions, converting the Heilenberg algorithm to a higher order version (29:445–446). They showed that the improved algorithm successfully modeled hyperacuity for an array of receptor inputs with a defined inter-receptor spacing (29).

The general form of the Hermitian polynomial, $H_p(t)$, where p is the order of the polynomial, is represented as:

$$H_p(t) = (-1)^p e^{t^2} \frac{d^p}{dt^p} e^{-t^2} \tag{5}$$

2.2.3.5 Namatame Namatame presented a unique high-order network employing Chebychev polynomials as inputs to a *multilayer* system (Figure 12). The network was designed to learn nonlinear continuous functions. Namatame contended that "first-order multi-layer networks and the high-order flat networks without hidden units ... are inadequate to generalize and to learn the nonlinear structures underlying the continuous mappings" (20:I-682). Namatame demonstrated the use of nonmonotonic Chebychev polynomials feeding three layers of artificial neurons (20:I-682).

Namatame is in accord with previously cited research regarding the improved

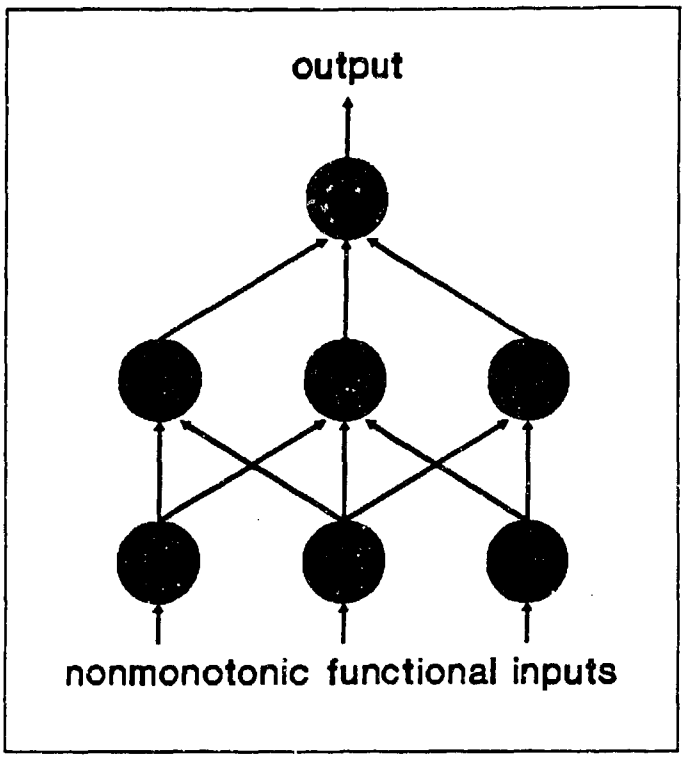


Figure 12. A Multilayer High-Order Neural Network (20:I-683)

capabilities of high-order networks over standard first-order, multilayer models. However, Namatame's use of *multiple layers* in conjunction with high-order functional inputs deviates from the "classical" single slab versions of high-order implementations. As the results suggest, for learning and predicting nonlinear continuous functions, the added aspect of multiple layers may prove a superior approach for high-order networks. (20)

The Chebychev polynomials (Figure 13), of order i, are defined for $0 \leq x \leq 1$ in the following general equation (20:I-681):

$$g_i(x) = \frac{\cos(i \arccos(2x-1)) + 1}{2} \tag{6}$$

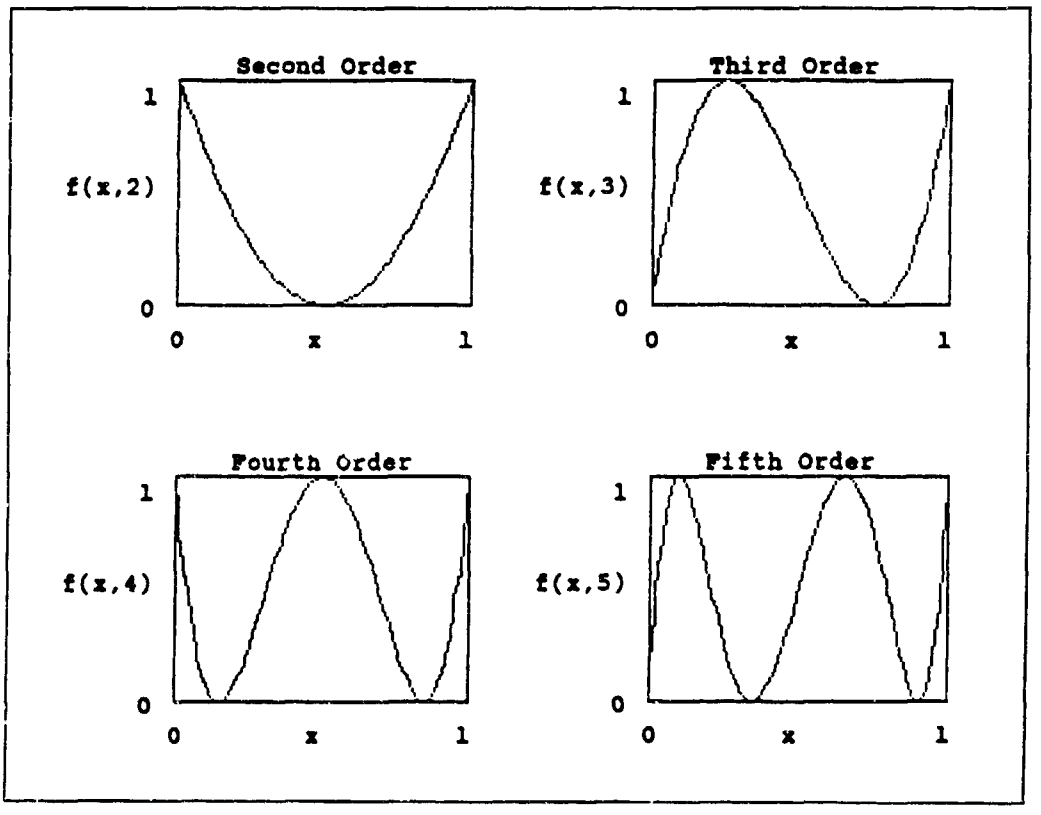


Figure 13. Examples of Chebychev Polynomials

2.2.3.6 Le Cun An image recognition algorithm, tested on handwritten numerical digits, was developed by Le Cun *et al.* The algorithm employed multiple layers of nodes in a network scheme designed for "extracting local features and combining them to form higher-order features" (16:44). Handwritten digits for the data set were obtained from zip code numerals off addressed envelopes processed by the US Postal Service. The training set was comprised of 7291 digit exemplars while 2007 digits were segregated for the test set (16:41).

Each input image was mapped on a 16-by-16 pixel array. The first layer of nodes was organized into twelve groups of 8-by-8 node arrays. Weights were constrained for each group of nodes, with each node "viewing" a 5-by-5 pixel block from the input image. With 64 constrained weights per node group, and *unconstrained* thresholds, weight updating was performed via backpropagation using a computed average error (16). This architecture is similar to Fukushima's hierarchical neocognitron except that it is supervised and not binary (7).

The next layer of nodes consisted of twelve groups of 4-by-4 node arrays, each node being fed from a 5-by-5 node block from the previous layer. Outputs from this layer were fully connected to a layer of thirty hidden nodes, each subsequently feeding all ten output nodes (16). Figure 14 diagrams the network architecture developed by Le Cun *et al.* Following training, this network achieved 95% accuracy on the test data (16:45). This algorithm also inspired, in part, the proposed image recognition network developed in this thesis and detailed in Chapter III.

2.3 Literature Review Summary

Neurophysiological researchers cited in this literature review have observed a variety of functional measurements in the neomammalian visual cortex. Their observations lend credence to the proposed phenomena of stimulus specificity and phase synchronous firing of groups of neurons, and the potential use of Gabor functions in modeling neuronal (spatial) receptive field profiles. Meador, whose high-order

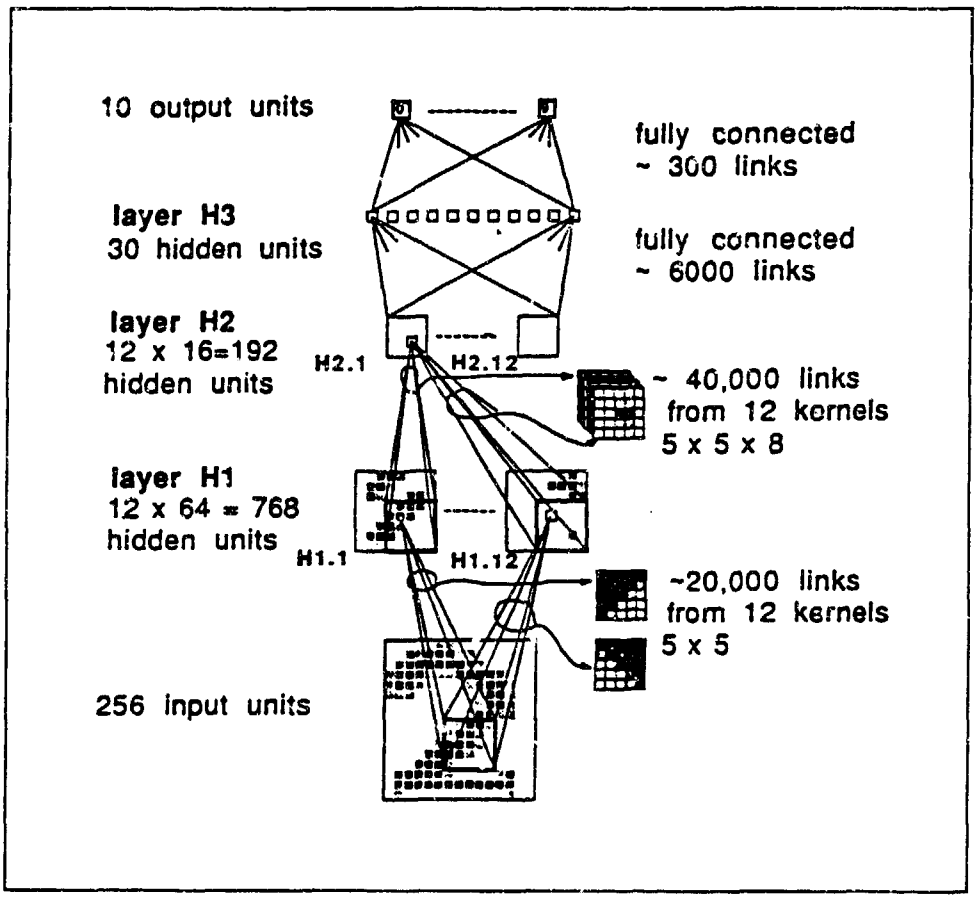


Figure 14. Le Cun's Digit Recognition Network (16:44)

networks outperformed standard first-order networks, also cited biological literature supporting axon-to-axon connections among neurons. It is possible, however, that some of these biological measurements are artifacts representing some random or systematic noise in the experimental processes, rather than observations of actual cortical phenomena.

To segment image features, Ayer and Fretheim independently correlated image scenes with Gabor functions of differing spatial frequencies and variances (1, 6). The visual cortex may possibly employ a similar process for discerning texture (14).

Based on the assumption that the cited biological observations represented real processes in the "wetware," this thesis research sought to explore new aspects of neural network algorithms with ideas gained from measured cortical processes. This did *not* assume a comprehensive understanding of cortical functioning. Rather, this work attempted to unify a limited body of biological, mathematical, and network algorithmic knowledge to develop an improved model for solving certain classes of engineering problems.

This chapter discussed and integrated experimental work published by several biological and artificial neural network researchers. The following chapter covers the methodology and approach used in this thesis.

III. Methodology

Citing the work of several neurophysiological researchers, Chapter II covered a subset of observed biological processes in the mammalian visual cortex along with recent research into artificial neural network algorithms. Specifically noted were the phenomena of stimulus-specific responsiveness and phase-synchronous firing of neurons in the visual cortex, the semblance of Gabor functions in characterizing neuronal receptive fields, and "axo-axonic interconnections" (19) in biological neural networks. This thesis work sought to emulate some of the simple aspects of these biological phenomena in software algorithms designed for feature extraction and image classification (target identification).

This chapter covers the development and testing of proposed software-based algorithms. One of the discussed solution methodologies employs multiple Gabor function correlations across image pixel arrays. This process, intended for feature extraction within an image, may emulate stimulus-specific responsiveness of some cortical neurons to unique local spatial frequencies. Also discussed is a local averaging routine intended to model phase-synchronous neuronal firing in the visual cortex.

The first section of this chapter outlines the design and testing of high-order classification engines. Operating on extracted features of segmented images, these classifiers are theoretically analogous in function to higher cortical associative processes employing axo-axonic connections among neurons.

3.1 Investigation of High-order Neural Network Classifiers

The first phase of this work entailed the development and testing of single-layer, high-order artificial neural networks (see Appendix A for software code). This involved the software implementation of second and third-order network algorithms. Written in standard C programming language and compiled and run on two ELXSI

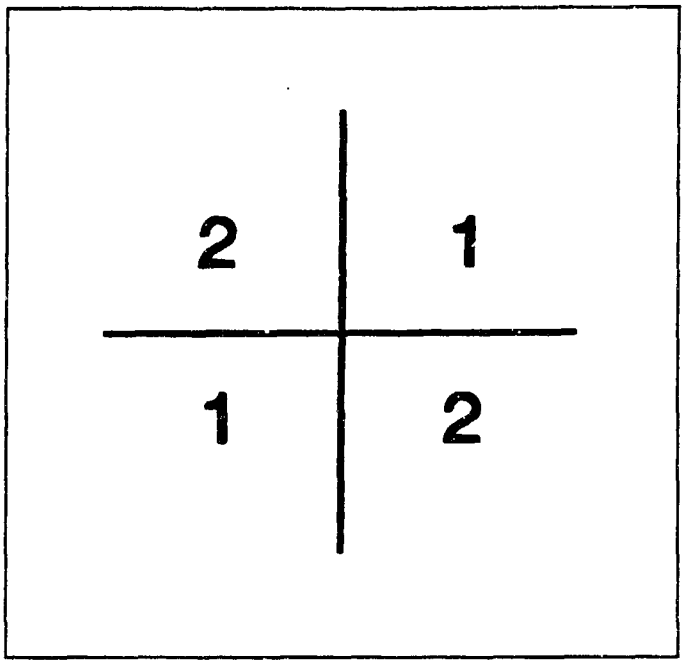


Figure 15. Two Segregated Classes of the 2-D XOR Problem

6400 computers, these networks were tested on three separate classification problems to assess their learning performance (speed of solution convergence and maximum achieved accuracies). The purpose of this segment of the research was to determine the potential usefulness of high-order neural networks as classifiers for a variety of problems, including image recognition.

The test problems consisted of data sets for the 2-D exclusive-or (XOR) problem (Figure 15), the 2-D "mesh" problem (Figure 16), and Ruck's calculated moments of pixel data from military vehicle images. The mesh data, consisting of 1000 vectors, was obtained from Tarr, whose previous work at AFIT used the data to test multilayer perceptron classifiers. The image moment data, consisting of 81 vectors, was computed by Ruck from pixel data and used in testing the capabilities of multilayer perceptron classifiers by Ruck and Tarr at AFIT. The computed moments were position, scale, and rotation invariant.

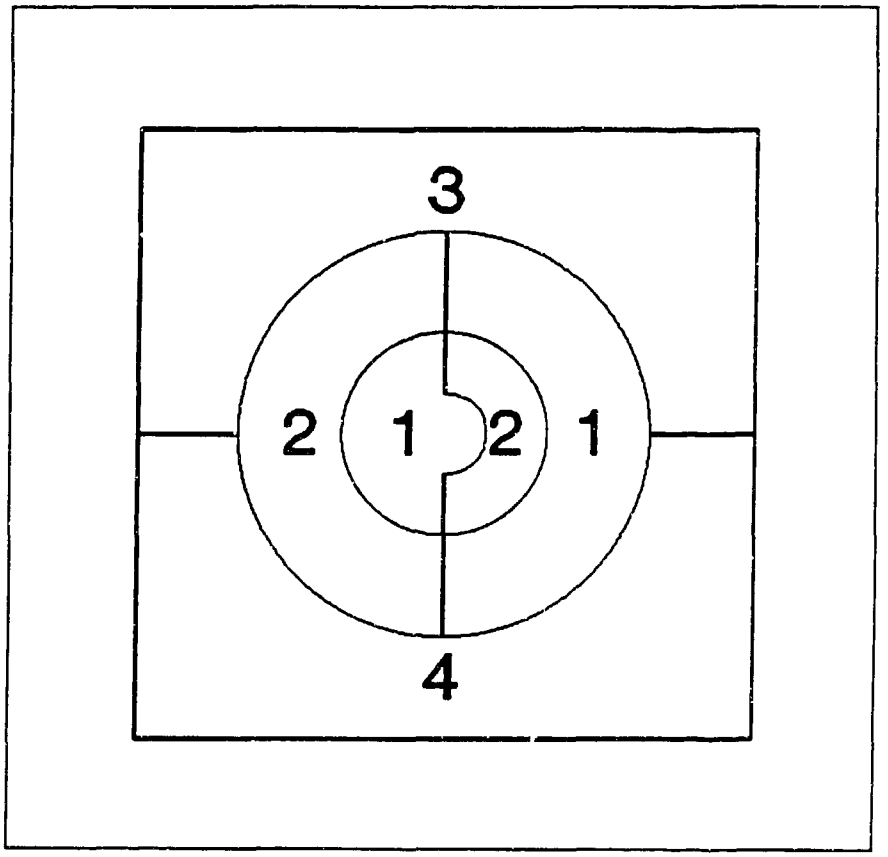


Figure 16. Four Segregated Classes of the 2-D Mesh Problem (24:60)

3.1.1 Algorithm for the High-order Classifiers The networks developed for the second and third-order classifiers were similar with a few exceptions. The second-order network fed linear and second-order combinations of feature inputs, each multiplied by separate weighting factors, to the sigmoid function comprising each artificial neuron. The third-order network fed linear, second-order, *and third-order* combinations of feature inputs to each artificial neuron. The Kolmogorov-Garbor polynomial, shown in Equation 3, depicts a generalized mathematical version of the summation of these high-order input combinations. The polynomial summation, added to a threshold value (θ) for each artificial neuron, was fed to the sigmoid function internal to each artificial neuron.

Second and third-order combinations of feature inputs are simply the multiplied exponential and cross-product combinations of the vector components. The sigmoid functions to which the summation of these first, second, and third-order combinations were fed may be represented by:

$$y_n = \frac{1}{1 + e^{-(\alpha+\theta_n)}} \tag{7}$$

where y_n is the sigmoid output and θ_n the threshold for the *nth* artificial neuron. The variable α represents a truncated form of the Kolmogorov-Garbor polynomial. For the second-order network algorithm:

$$\alpha = \sum_i w_i x_i + \sum_i \sum_j w_{ij} x_i x_j \tag{8}$$

For the third-order network algorithm:

$$\alpha = \sum_i w_i x_i + \sum_i \sum_j w_{ij} x_i x_j + \sum_i \sum_j \sum_k w_{ijk} x_i x_j x_k \tag{9}$$

The array variable x represents the vector components, or feature inputs, and w is a corresponding array of weight factors for feature inputs and second and third-order

multiplied combinations of the feature inputs.

The software programs were written allowing the user to specify key parameters for each compiled run. These variable parameters included the number of output classes for the problem, the number of exemplar vectors, training vectors, and components (features) per vector as dictated by the data set, the number of training iterations run between each accuracy test of the network, and the total number of training iterations to be run and plotted. Since these networks consisted of a single layer of artificial neurons, the number of output classes, number of output neurons, and total number of neurons were all equal.

The user also specified the learning rate, η, and the name of the data file to be accessed by the program. There was also a "switch" in the program which could be set to divide the learning rate by the *fan in*, or number of inputs fed to each artificial neuron's sigmoid function. Dividing η by the fan in may aid network learning when relatively large numbers of inputs are fed to the artificial neurons (21). When *not* selected, the fan in variable was hard set to one so that η was *not* divided by the number of inputs. This was the case for test running the 2-D XOR and mesh problems. For the twenty-two-dimensional Ruck data, however, η was divided by the fan in due to the vastly increased number of sigmoid inputs created by the higher dimensionality of the data vectors.

After opening the data file and loading into storage arrays the flag (index) number, feature component values, and desired output class for each vector, a program module calculated the number of second-order and third-order multiplicative combinations for each vector's components. These calculated numbers, used for dynamically allocating storage arrays for second and third-order input combinations, are functions of the number of features per vector. Having correctly allocated the array sizes, the program multiplied out the second and third-order combinations of the features and stored these high-order input values in the arrays.

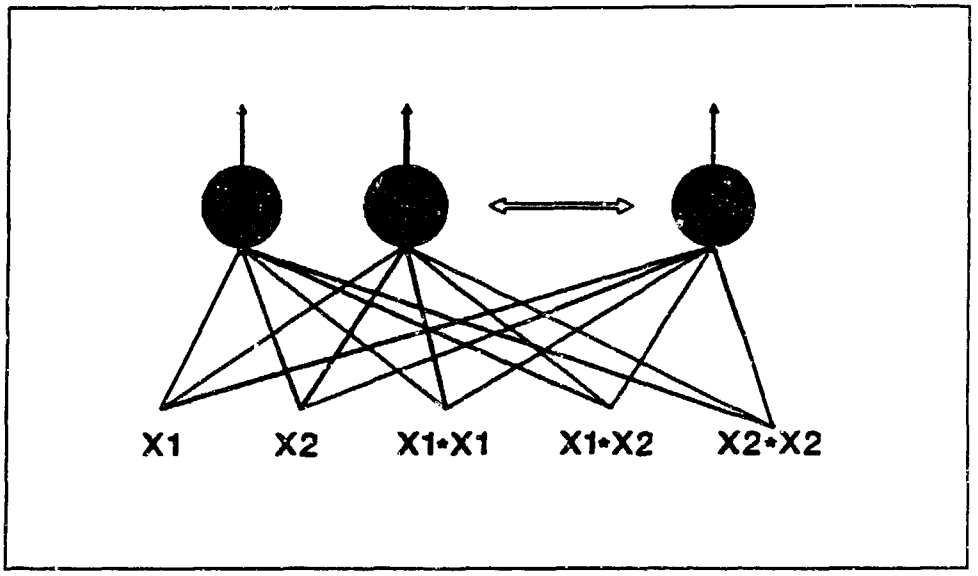


Figure 17. Second-order Network Algorithm with Two Feature Inputs

To save memory, the algorithm avoids the commutative redundancy of multiplied input combinations. For example, for two feature components, x_1 and x_2, the second-order network stores and uses the inputs x_1, x_2, x_1^2, x_2^2, and $x_1 * x_2$ (Figure 17). Each squared exponential is calculated and used only once and the multiplied combination $x_2 * x_1$ is not calculated. By eliminating the redundancy of commutativity, the algorithm saves a huge number of floating-point operations and substantial memory, particularly when operating on data vectors of high dimensionality.

Initial testing of the network algorithms on the mesh problem data resulted in erratic fluctuations of accuracy during the learning process. This occurred using learning rates of 0.35 and 1.0. The seemingly random vacillations in learned accuracy indicated that perhaps the networks were not learning to any degree. It was postulated that the disappointing results were due to data inputs that were not normalized, including the higher-order multiplied combinations derived from the first-order inputs. Second and third-order multiplications of fractional input values yield relatively small high-order inputs. Miniscule high-order inputs may contribute little to weight updating and learning, thus the need for normalization.

Normalization program modules were subsequently developed and inserted into the networks to vertically normalize the first, second, and third-order vector data arrays. The mean, μ, and standard deviation, σ, were calculated for each feature component column of the exemplar vectors. All first, second, and third-order vector components, x_i, for exemplar and test vectors, were then vertically normalized using the equation (21):

$$x_i(normalized) = \frac{x_i - \mu}{\sigma} \tag{10}$$

After completing the initialization routine of the high-order algorithms, the array of θ values, one for each neuron, and the arrays of weight values, one for each first, second, and third-order input for each neuron, were filled with random floating point numbers between -0.5 and 0.5. These values lie within the region of greatest change in the sigmoid function. For each network run, the random function generator was itself randomly seeded using the computer's time clock. This ensured a new set of initial weights and thresholds for the start of each learning run.

During the training loops of the second and third-order algorithms, exemplar vectors from the data set were selected in random sequence for exposure to the network. For each exemplar vector shown to the network, inputs consisting of the vector's components and their higher-order combinations were multiplied by corresponding weight factors and fed to the sigmoid functions comprising each neuron. Based on the input vector, this yielded a unique output value, y, for each neuron. A desired output, d, was assigned to each neuron such that $d = 1$ *only* for the neuron corresponding to the output class of the presently run exemplar vector. For the other neurons in the layer, each of which selected for other classes, the algorithm set $d = 0$. For these single-layer networks, the number of neurons equaled the number of output classes for a problem. A neuron would output high to indicate network selection of its corresponding output class.

The error, Δw, was calculated for each neuron using the actual neuron output,

y, and its desired output, d, for each exemplar vector. The well-known gradient error equation for the sigmoid function is:

$$\Delta w = y * (1 - y) * (d - y) \tag{11}$$

For each neuron, each weight value, w_i, was updated via the equation:

$$w_i(updated) = w_i + \eta * \Delta w * x_i \tag{12}$$

where x_i represents the corresponding input for the weight (including higher-order inputs for updating higher-order weights). As previously mentioned, η may be divided by the fan in to aid network learning, particularly when many inputs are fed to the artificial neurons (21). Threshold values (θ) for each neuron were updated using the same equation, where the value of x_i was set to one.

Through successive training iterations, or sequential random exposures to exemplar vectors, the process of weight updating serves to train a network. If a paradigm is capable of using the input vectors' features to converge to a network transfer function that accurately distinguishes each class, the network has successfully learned the problem.

To assess the degree of learning achieved by the networks on each problem, a test loop sequentially accessed all designated test vectors from the data set. The test vectors were originally segregated from the training vectors and were thus "unseen" by the network during training. This was required to truly determine the generalizing capability of the trained network.

During testing, each test vector's components and higher-order combinations were fed once through the network algorithm. This entailed multiplying each input by its corresponding weight factor stored in memory, summing the results, adding corresponding neuron threshold values, and feeding the summation through the sig-

moid function of each artificial neuron. This process was mathematically identical to determining the output, y, for each neuron from the exemplar vectors during training. However, in this case, there was no subsequent updating of weights to further train the net. During testing, the resulting outputs of the network for all test vectors were tallied to determine the number of correct class selections, yielding a percent accuracy for the present state of the network (determined by its stored weights). Following a testing session, which culminated with a stored value for network accuracy, the program returned to the training loop to continue updating the weights based on additional exposures to exemplar vectors.

For testing purposes, the criteria for determining a correct network response to a test vector required that the in-class node output a value greater than 0.8 *and* all other nodes (corresponding to different classes) output less than 0.2. This criteria is based on the networks' estimation of the conditional probability densities (25). Note that for the employed sigmoid function the output range was between 0 and 1. Network accuracy was calculated by dividing the total number of test vectors into the number of correct network responses.

Each full network run yielded a series of accuracy values for training iteration intervals, reflecting network learning as a function of exposures to training data. The software programs averaged the results of ten separate learning runs of each network on each problem to create the plotted results shown in Chapter IV. For the third-order network operating on a statistically normalized version of Ruck's moment data set, results were averaged from only five runs to limit the extensive computation. Averaging the results from several runs served to smooth any unique convergence results caused by a specific combination of initial random weights for any one training session.

3.2 *Development of an Image Classification Network*

Several versions of a biologically-motivated image classification network were developed and tested in the second phase of this thesis. The network algorithm was inspired by the biological research observations of cortical functioning detailed in Chapter II and, in part, by research conducted by Le Cun *et al* (16). The network operated on a data set consisting of segmented pixel arrays of forward-looking infrared (FLIR) images of tanks, trucks, target boards, and clutter. The first layer of the network essentially correlated each image with four Gabor functions of differing angular orientations. This process was intended to emulate, in a highly simplified model, the stimulus-specific responsiveness of biological neurons in the visual cortex to varying spatial frequencies, or textures. Chapter II discussed neurophysiological evidence of cortical neurons functioning as Gabor function detectors.

The second layer of the image classification network performed a local averaging routine on the outputs of each Gabor "orientation column." In a simplistic way, this process mathematically modeled phase synchronous firing of neurons in the visual cortex, an observed biological phenomena discussed in Chapter II. This higher layer was employed to glean larger global properties from an input image, rather than smaller stimulus details and noise extracted in the first layer of the network.

Processed outputs from the second layer, serving as extracted features of the input pixel data, were fed to the final output layer of the network for classification. The output layer consisted of four sigmoidal nodes, each representing one of the four desired output classes. Subsequent versions of the network modified the final output layer to find a preferred classification scheme. Two network variations employed high-order networks at the output layer to determine if various second-order combinations of extracted features could successfully separate the data. Another version implemented a multilayer perceptron with a variable number of hidden layer nodes. Final network versions statistically normalized the features extracted from the Gabor wavelet correlations and local averaging routine, feeding the results to

the output classification layer. A diagram of the basic feature extraction algorithm (*without* details of each output layer variation for each network version) is depicted in Figure 18.

3.2.1 Production of the Image Data Set A data set was developed for testing the image classification network. The data set consisted of integer gray-scale values from pixel arrays representing 88 separate FLIR images. Using software developed and modified by Ayer (1) to window and excise pixel arrays of images contained within larger scenes, 21 tanks, 23 trucks, 23 target boards, and 21 clutter images were segmented from FLIR scenes used by Roggemann in earlier work at AFIT. Ayer's software was run on VMS machines in the AFIT Graphics Laboratory to extract the 88 images and convert each to a list sequence of integer gray-scale pixel values ranging from 0 to 255. Each image was windowed within a rectangular pixel array of 63 rows by 128 columns, yielding 8064 integer component values per image. Images were selected to minimize variations in size and were roughly centered and positioned upright within the extraction window.

Each of the 88 segmented images was assigned a unique index number, from 0 through 87, inserted before the first of the 8064 pixel values comprising each image's list. A class identification number was then appended to the tail of each image's numerical list to identify the image class: 0 for target board, 1 for clutter, 2 for tank, or 3 for truck. Each image's completed sequence of numbers consisted of 8066 integers including an index number, 8064 gray-scale pixel values, and a class identification number, in that order. The 88 separate numerical sequences representing the 88 segmented images were finally ported to one of AFIT's ELXSI 6400 computers and concatenated into one large file comprising the data set. The numerical data set for the FLIR images was thus structured in a "vector-style" format for insertion into the network software.

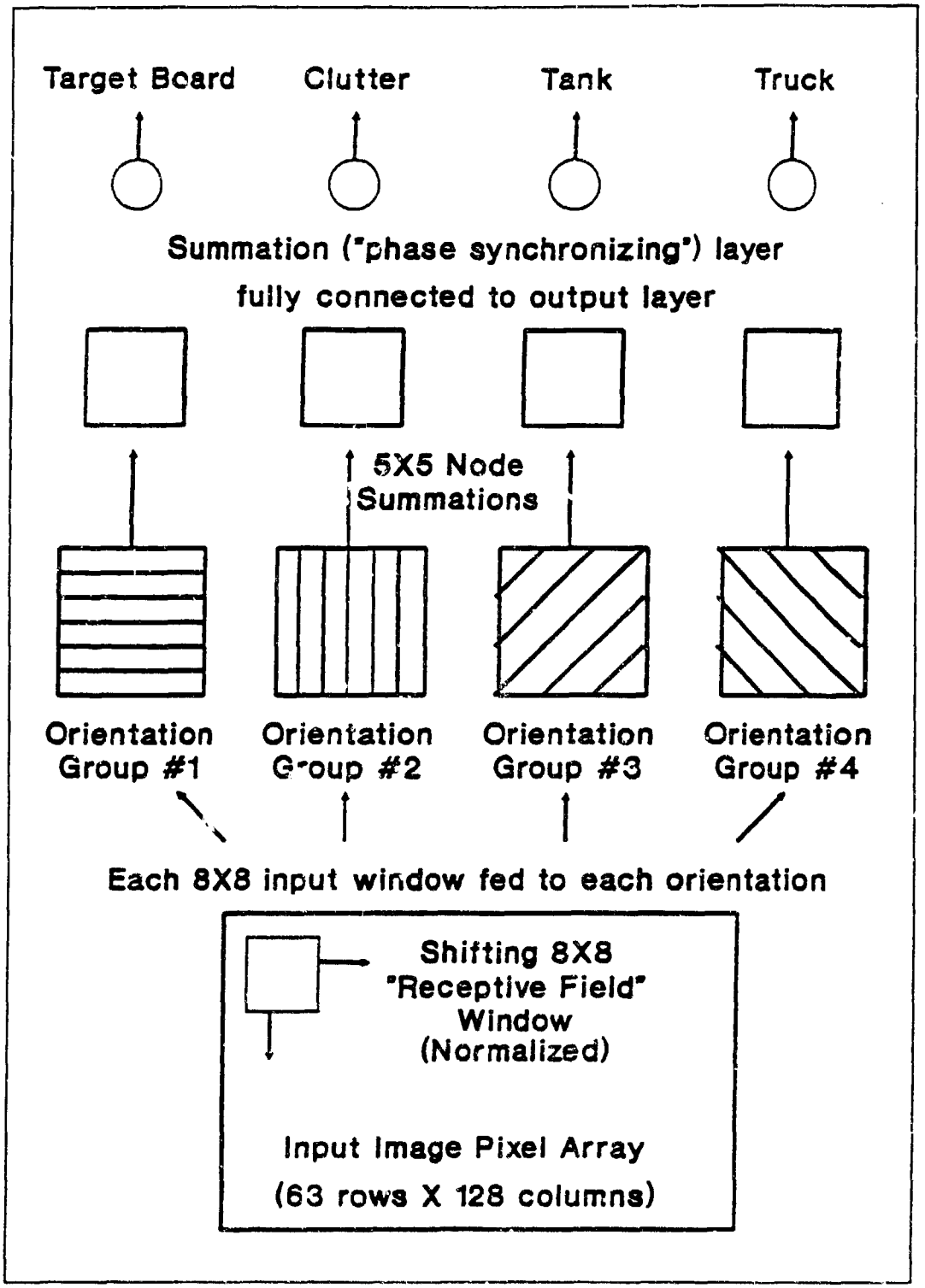


Figure 18. Biologically-Motivated Image Classification Network Algorithm

3.2.2 The Biologically-Motivated Image Classification Algorithm The software code for the image classification algorithm was written in C programming language and compiled and run on two ELXSI 6400 computers at AFIT. Programs were written allowing the user to readily vary the number of exemplar versus training images designated from the input data set, the number of training iterations run between network accuracy tests, the total number of training iterations, and the learning rate, η. Based on the FLIR image data set produced for this thesis, the number of numerical components per input image was hard set at 8064. The number of exemplar vectors and test vectors were both set at 44. Learning rates used for various program runs were either 0.35 or 1. The software did not store and average results from multiple runs due to the computationally intense nature of the algorithm and time constraints on the project.

Program arrays for the index, class, and image gray-scale pixel values were first loaded from the data set. For each program run, the random function was seeded from the computer's time clock. Arrays for weights and thresholds were filled with random floating point values between -0.5 and 0.5, as values in this range fall within the region of greatest change for the sigmoid function. To preclude saturation of the sigmoid function, these weights and thresholds, along with the learning rate, η, were subsequently divided by the node fan in, counteracting the "explosive" effect of huge numbers of node inputs. Also, as detailed in a separate section of this chapter, four arrays of constrained hard-wired weights for the first processing layer were filled with 2-D discrete Gabor functions.

Training and testing loops contained identical network propagation routines. The training loop, randomly selecting image exemplars for network propagation, additionally contained standard backpropagation routines for updating weights. The testing loop sequentially accessed designated test images, all "unseen" by the network during training, and determined if the network's propagated outputs matched the desired classes. Correct test outputs were tallied and divided by the number of

test images, yielding a percentage for network accuracy. A correct network output was defined by an in-class node output greater than 0.8 *and* all other node outputs less than 0.2 (for the output layer). Printouts for each network run displayed a series indicating the cumulative number of training iterations and the corresponding network accuracies. Specific details regarding the software are contained in Appendix B.

The proposed propagation algorithm was intended to emulate, in a simple way, observed cortical phenomena that *might* relate to the processing of visual information in the brain. Each input image was "blocked off" into 8-by-8 pixel arrays to mimic the limited receptive fields processed by neurons in the visual cortex. Using a 50% overlap in both image dimensions among these "receptive fields," an input image of 63 rows by 128 columns of pixels was segregated into 434 overlapping "receptive fields" (14 rows by 31 columns). Each 8-by-8 pixel "receptive field" was processed by a separate node in the first processing layer of the network. This was meant to imitate a receptive field "viewed" by a biological neuron within a cortical orientation column of the visual cortex.

An engineering judgement choosing the 8-by-8 size of the segregated pixel blocks was made based on the observed size of physical features in images from the data set. Ideally, the "receptive field" sizes were intended to capture local texture changes within the different images that might aid the network in discerning the classes. Also, due to the computationally intense nature of the proposed network, the 50% overlap among "receptive fields" was chosen to limit the number of processing nodes employed at higher levels of the algorithm. This 50% overlap could be used as the "mother wavelet" for a multiresolution hierarchy.

Processing an input image through the first processing layer of the network entailed the correlation of each separate "receptive field" with four discrete Gabor function wavelets. A correlation multiplied each of 64 constrained hard-wired weights representing a Gabor wavelet by the corresponding pixel values in a "receptive field"

and summed the resulting 64 products. Details regarding these Gabor function representations of the constrained weight sets are discussed in a later section of this chapter. It is also noteworthy that each "receptive field" was first Lambertized and contrast normalized prior to correlation with the Gabor functions. Details of the contrast normalization routine, along with justification for employing it in the algorithm, are also covered in a later section of this chapter.

The correlated result of each "receptive field" with each Gabor function was fed to a unique sigmoidal node. The network structure of this processing layer thus consisted of 434 nodes for *each* of the four Gabor functions. Grouping 434 nodes, one per "receptive field", for each Gabor function mimics the organization by "orientation columns" of neurons in the visual cortex. From this layer, the sigmoid outputs from each Gabor orientation group fed a subsequent layer of nodes, also segregated into four groups to separately process each Gabor orientation. This subsequent layer of nodes, linear rather than sigmoidal functions, served to emulate the phenomena of phase synchronization of neuronal firing discussed in Chapter II.

The phase synchronization layer, with four groups of nodes uniquely processing the corresponding four Gabor orientation groups, contained 270 nodes per group. Each node in this layer served to average the outputs of 5-by-5 blocks of nodes from the Gabor function layer below. Just as the Gabor correlation layer preserved the 2-D locational nature of the image input data in its node arrays, the phase synchronization layer also preserved the 2-D locational information in processing 5-by-5 node blocks from the Gabor correlation layer. The 5-by-5 node blocks from the Gabor correlation layer, with maximum overlap (shifting one node in each dimension), yielded 10 rows by 27 columns of blocks per Gabor group. Each linear node in the phase synchronization layer averaged a separate block of 25 nodes by computing the arithmetic mean of their outputs. The resulting layer structure consisted of four groups of 270 nodes per group, each node computing a unique average output from the layer below. *This created 1080 output values representing extracted fea-*

tures transformed from the original pixel data for each image. These 1080 numerical outputs (per image) served as inputs to the final classification (output) layer of the network. Through its local averaging routine, the phase synchronization layer was intended to "bind" local details into larger, global features of an image. Figure 18 illustrates the propagation hierarchy through the network layers.

The final output layer of the network consisted of four sigmoidal classification nodes representing target board, clutter, tank, and truck. For each of these "top" nodes, outputs greater than 0.8 were considered high (in-class) and outputs less than 0.2 were considered low (out-of-class). Several different versions of the output classification layer were tested. These network variations were needed to fully assess the capability of the biologically-motivated feature extraction algorithm.

3.2.2.1 Variations of the Output Classification Layer The first network version employed a single layer of nodes for output classification. The output layer consisted of four sigmoidal nodes fully connected to nodes from the phase synchronization layer. This configuration assumed linear separability of the features extracted from the images' pixel data.

In following the first part of this thesis work investigating high-order networks, two subsequent versions of the image classification network implemented high-order classification schemes. One of these versions multiplied second-order combinations of the 270 node outputs within each of the four separate orientation groups of the phase synchronization layer. The resulting 146,340 total multiplied combinations were fed, along with the unmultiplied features, into all four output classification nodes in the single-layer perceptron. The other high-order scheme multiplied second-order combinations of corresponding node outputs, or locations, *among* the four groups of the phase synchronization layer. With 2700 multiplied second-order combinations, this approach proved far less computationally intense than the previous high-order

network version. The high-order network versions assumed separability of second-order combinations of extracted feature values.

For more thorough testing, the network was modified to employ a multilayer perceptron, with two layers of trained weights, as the output classification layer. The hidden layer of the multilayer perceptron allowed for a variable number of nodes while the ultimate output layer consisted of four nodes, one representing each class. Varying the number of hidden layer nodes provided more testing versatility by altering the order of mathematical computation of the classifier. The intent was to variably scale the computational complexity of the network to eventually achieve solution convergence. Software code implementing the various network versions is included in Appendix B.

Noting the initial failure of these classification schemes in achieving solution convergence on the extracted features, *it was postulated that these features might require statistical normalization to enable successful classification.* Following the Gabor wavelet correlations and the local averaging algorithm, a vertical normalization routine was used to normalize, by column, the extracted feature components, x_i, of all 88 image vectors. The mean, μ, and standard deviation, σ, for each feature component column were calculated using only the exemplar vectors. All image vector components were then vertically normalized using the equation (21):

$$x_i(normalized) = \frac{x_i - \mu}{\sigma} \tag{13}$$

Feeding the statistically normalized feature values into the multilayer perceptron, the most versatile of the classification schemes, the network exhibited learning with increased training iterations. The statistically normalized features were also fed to the high-order network version multiplying second-order combinations of corresponding node positions among the four orientation groups. This high-order network version also exhibited learning on the normalized features. Detailed result of network

testing are covered in Chapter IV.

3.2.2.2 Lambertization and Contrast Normalization of Pixel Values To ensure network learning of relative differences among pixel values, rather than of absolute pixel values in an image, a normalization routine was implemented to operate on 8-by-8 grouped blocks of pixel values prior to network processing. The 8-by-8 pixel array blocks were those sectioned off as "receptive fields" for the Gabor function detectors in the first network layer. To simplistically emulate biological retinal preprocessing, normalization of the gray-scale pixel data was accomplished using Lambertization and contrast normalization algorithms (13, 14).

Employing each 8-by-8 "receptive field" as a 2-D windowed region of pixels within the image, Lambertization was accomplished by computing the average brightness, A, of the 64 pixel values within each window such that:

$$A = \frac{1}{64} * \sum_{i=1}^{64} x_i \tag{14}$$

where x_i represents each pixel value in the 8-by-8 window. A local contrast, x_i^{LC}, was then computed for each of the 64 pixel values by finding the difference between the average window brightness and each pixel value:

$$x_i^{LC} = x_i - A \tag{15}$$

The Lambertization algorithm was used to preserve local changes in image intensity while eliminating systematic brightness variations. (13, 14)

Following Lambertization, contrast normalization was used to mathematically convert vector length ("energy"), E, to unity. First, vector length was calculated by:

$$E = \sqrt{\sum_{i=1}^{64} (x_i^{LC})^2} \tag{16}$$

Each normalized pixel value, $x_i(normalized)$, was then calculated by:

$$x_i(normalized) = \frac{x_i^{LC}}{E} \tag{17}$$

This contrast normalization procedure was used to enhance contrast where it was too low within the image and reduce contrast where it was too high. (13, 14)

3.2.2.3 Discrete 2-D Gabor Functions Comprising First Layer Weights

The network's first layer of nodes, consisting of four groups of 434 sigmoids per group, processed correlations of four unique Gabor functions with each of the 434 "receptive fields" sectioned off within each image. The 2-D Gabor functions chosen for implementing the first processing layer of the network were 8-by-8, discretized wavelets possessing the same periodicity but characterized by different angular orientations (0, 45, 90, and 135 degrees). The 64 values corresponding to each discrete Gabor function served as constrained, hard-wired weight factors for inputs feeding the sigmoids within each Gabor orientation group. The four discrete Gabor functions were mathematically generated in software to fill four arrays of constrained weights with values between -0.5 and 0.5. This limited the Gabor function weight values to the region of greatest change for the sigmoid function. The following well-known Gabor equation (14), with selected σ values, was used to generate each function, $g(x, y, f_x, f_y)$:

$$g(x, y, f_x, f_y) = 0.5 * \exp[-\pi * (\frac{x^2}{3} + \frac{y^2}{3})] * \cos[2 * \pi * (f_x * x + f_y * y)] \tag{18}$$

where parameters f_x and f_y represent spatial frequencies covering two dimensions and input variables x and y were each discretely varied from -0.7 to 0.7 in increments of 0.2 to fill the 8-by-8 Gabor function weight arrays. Figure 19 depicts a plotted example of a Gabor function generated in this manner.

The four different Gabor orientations were uniquely produced by varying the

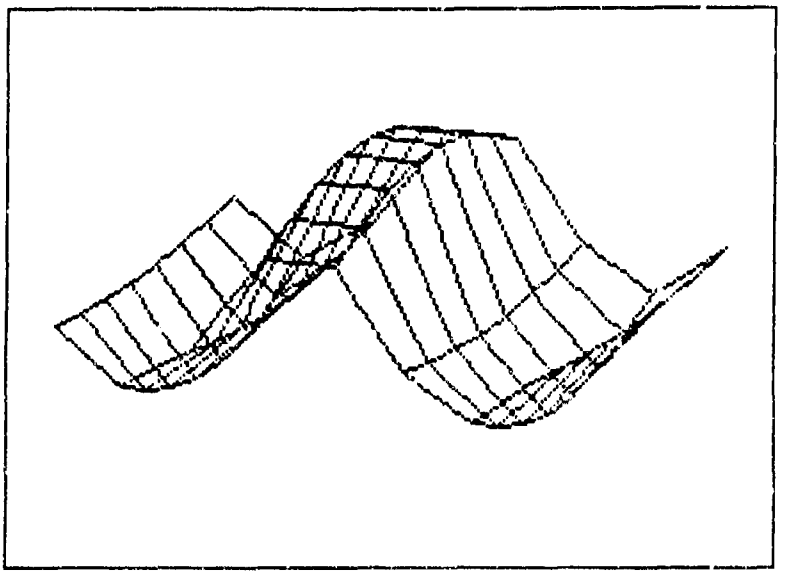


Figure 19. Plotted Example of a Mathematically Generated 8-by-8 Discrete Gabor Function

combination of the f_x and f_y parameters inserted in the function. For a wavelet of 0 degree orientation, $f_x = 1$ and $f_y = 0$, whereas a wavelet of 90 degree orientation was produced by $f_x = 0$ and $f_y = 1$. Wavelets at 45 degrees and 135 degrees, possessing the same spatial periodicity as the 0 degree and 90 degree wavelets, were generated by combinations of $f_x = 1.4142$ with $f_y = 1.4142$, and $f_x = 1.4142$ with $f_y = -1.4142$, respectively.

This chapter covered the details of the proposed high-order and image classification network algorithms. The following chapter discusses the testing results and performance of these networks.

IV. Results

4.1 High-order Network Performance Results

Chapter III covered details of the algorithms for the second and third-order network classifiers. The following plots depict the learning performance results for both of these networks applied to the three test classification problems. Results for the second and third-order networks, each tested using two different η values, were plotted separately. Based on previous neural network research at AFIT, $\eta = 0.35$ and $\eta = 1.0$ were selected as learning rates for testing each network (21). The following plots show the *averaged results from multiple runs* for each network on each test problem.

4.1.1 2-D XOR Problem Network learning performance results for the XOR data are depicted in the following plots. While both networks eventually converged to 100% accuracy, the second-order network outperformed the third-order network, reaching maximum accuracy in far fewer training iterations for a given η. Also, for either network, a greater η yielded more rapid solution convergence.

It is noteworthy that, for the XOR problem, the higher-order networks did *not* seem to outperform standard feedforward, multilayer perceptrons trained through backpropagation. Research by Tarr showed that, for the XOR problem, a multilayer perceptron network achieved 100% accuracy in 1300 training iterations. Tarr's algorithms employed sigmoidal nodes and a learning rate of $\eta = 0.3$ for backpropagation training. Tarr's criteria for correct network class selections required an output greater than 0.8 for the in-class output node and outputs less than 0.2 for out-of-class output nodes. (27)

4.1.2 2-D Mesh Problem The following plots illustrate network learning for the mesh problem. Neither network proved capable of learning the mesh test data to

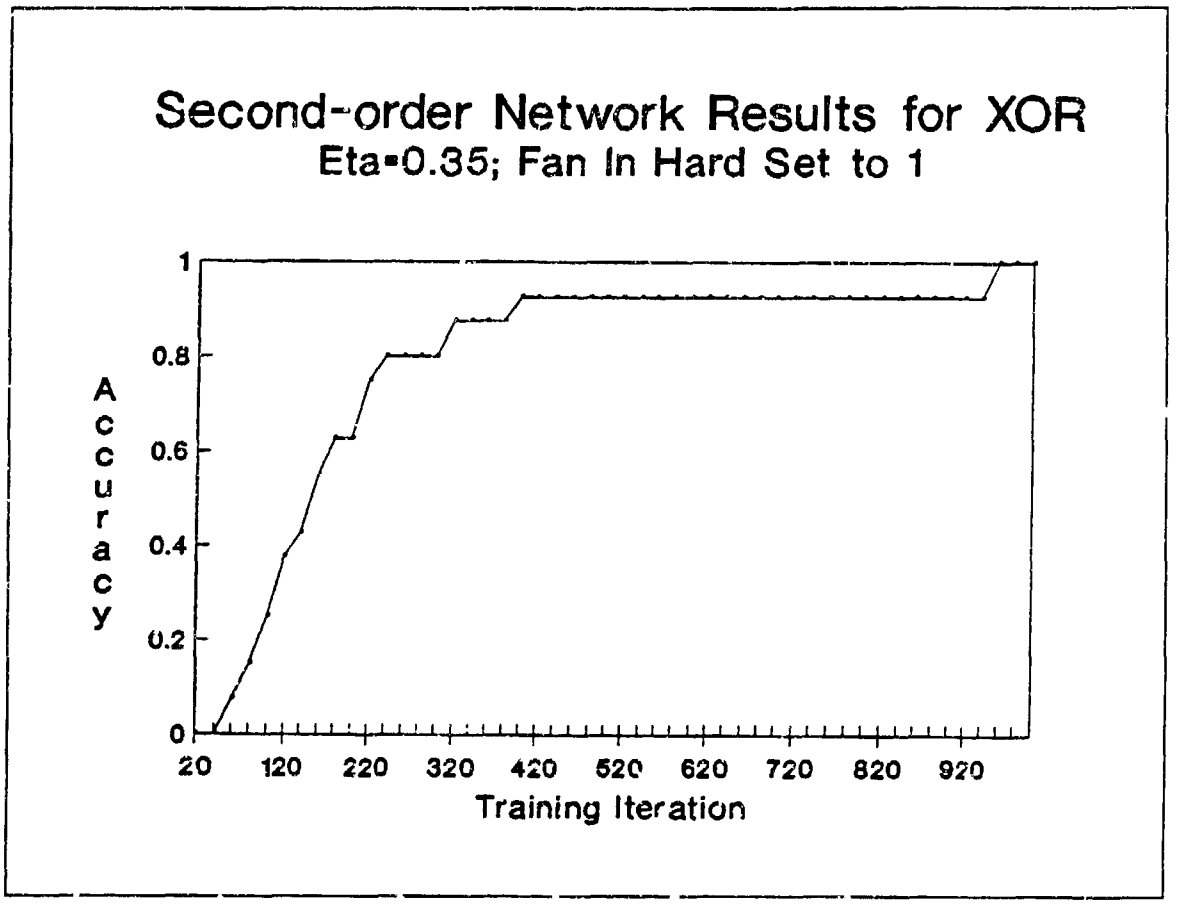


Figure 20. Second-order Network Learning XOR Data (Eta=0.35)

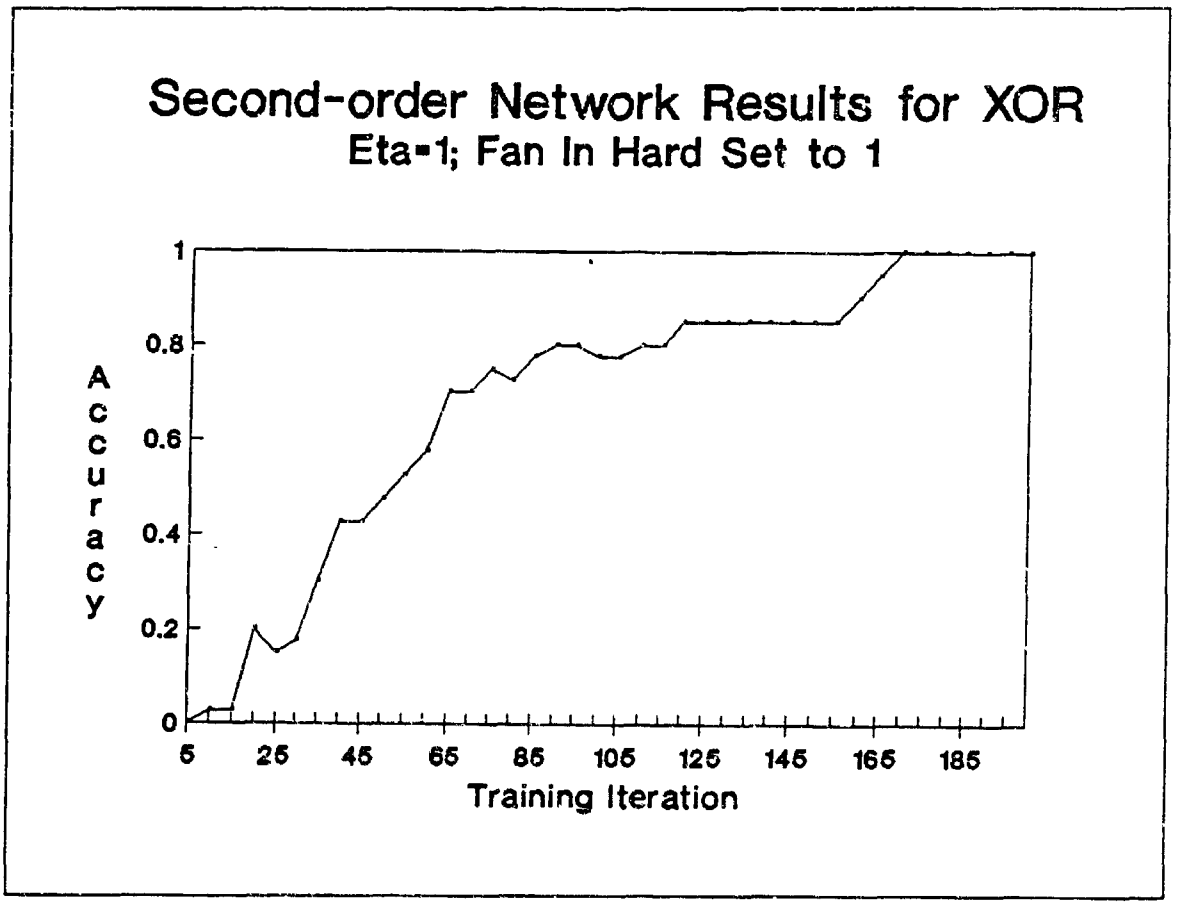


Figure 21. Second-order Network Learning XOR Data (Eta=1.0)

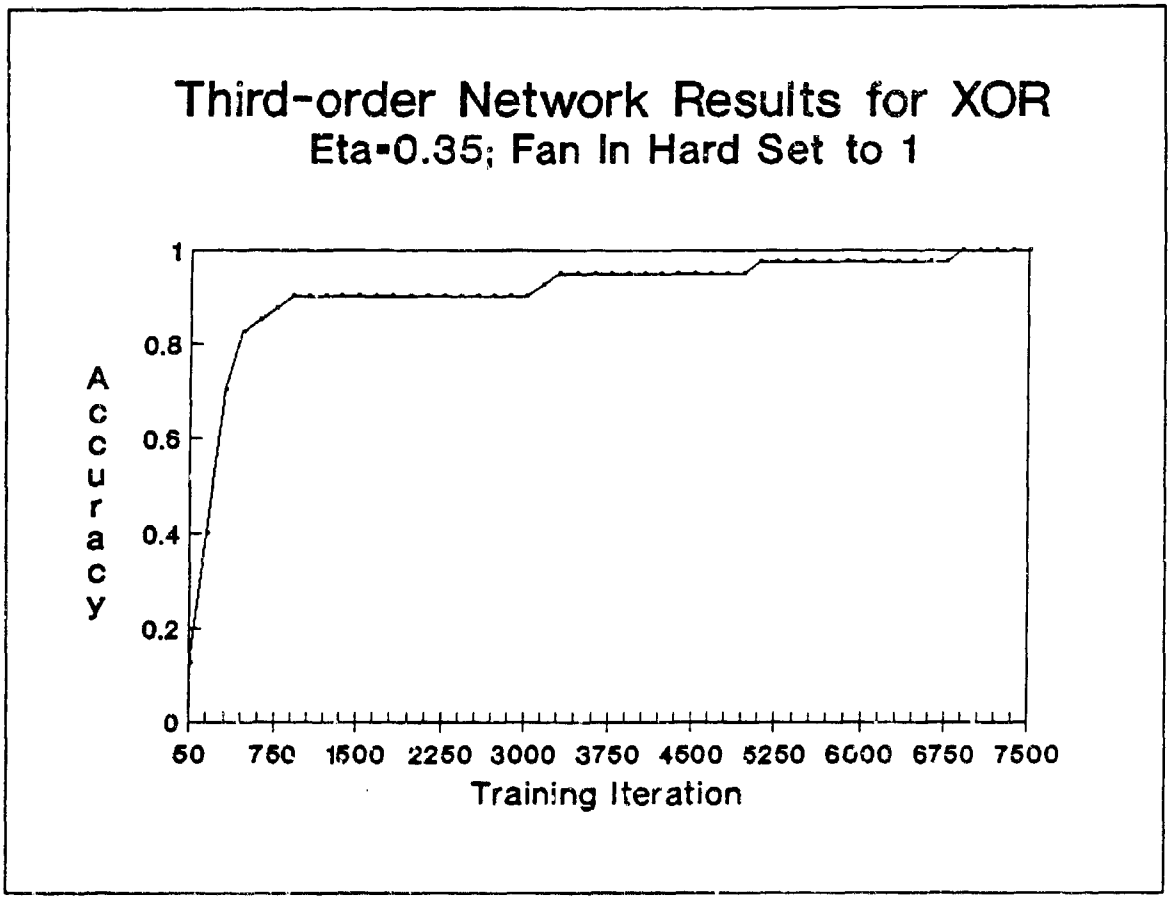


Figure 22. Third-order Network Learning XOR Data (Eta=0.35)

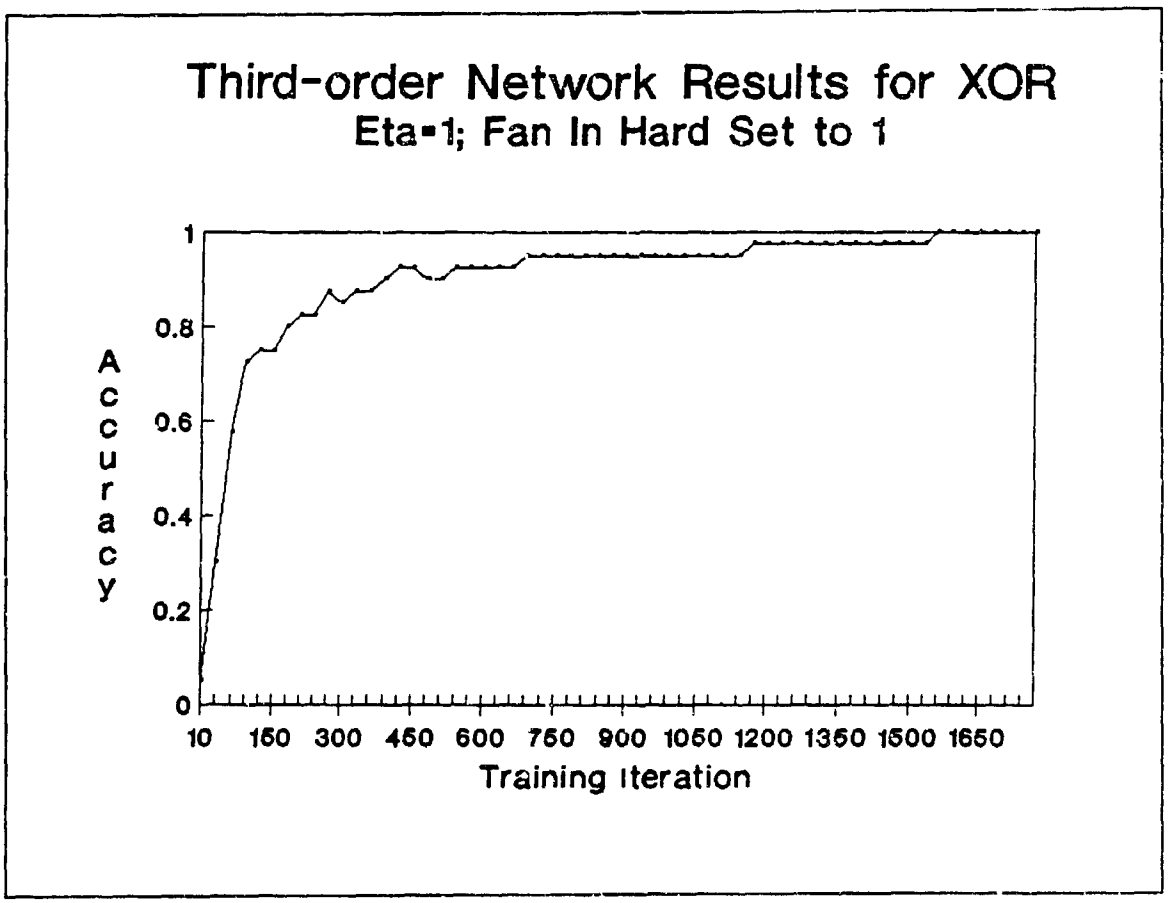


Figure 23. Third-order Network Learning XOR Data (Eta=1.0)

100% accuracy. Even after one million training iterations, the second-order network, for $\eta = 0.35$ and $\eta = 1.0$, achieved maximum accuracies of 64% and 66%, respectively. For both learning rates, the network actually approached approximate final accuracies within 500,000 training iterations.

The third-order network yielded nearly identical results. Through one million training iterations, the network maintained 65% accuracy for $\eta = 0.35$ and 67% accuracy for $\eta = 1.0$. As with the second-order network, the third-order network also converged to approximate final accuracies long before the millionth iteration was reached. Research by Ruck showed that a multilayer perceptron network trained via backpropagation could achieve nearly 70% accuracy on the mesh data in 8000 training iterations, outperforming the higher-order networks investigated (24:61). The following plots convey the convergence properties of the higher-order networks and their limited maximum learning accuracies for the mesh problem.

4.1.3 22-D Moments of Pixel Data (Ruck Data) The following plots depict network learning performance on Ruck's twenty-two-dimensional computed moments from pixel data of military vehicle images. As with the mesh data, this problem proved beyond the learning capabilities of both high-order networks, neither achieving 100% accuracy. The second-order network, for learning rates of 0.35 and 1.0, maintained approximate maximum accuracies of 31% and 33%, respectively, through one million training iterations. The third-order network maintained these same final accuracies through 500,000 training iterations. As depicted in the result plots, both networks actually converged to their maximum learned accuracies within a few hundred-thousand training iterations. Tarr's research showed that a multilayer perceptron network (with $\eta = 0.3$) achieved approximately 75% accuracy in 50,000 training iterations on Ruck's data (27). It was apparent that the high-order multiplied combinations of the moment features were not suitable for separating the classes.

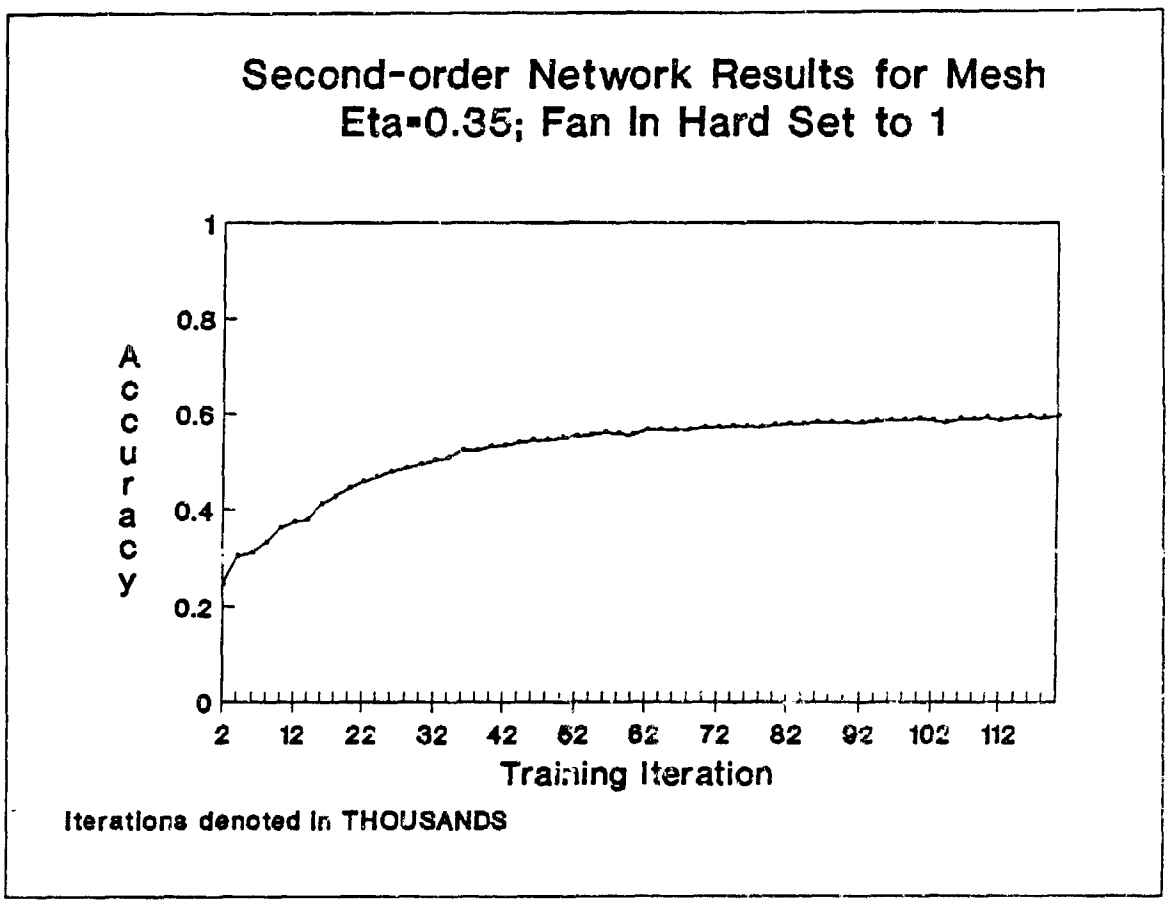


Figure 24. Second-order Network Learning Mesh Data (Eta=0.35)

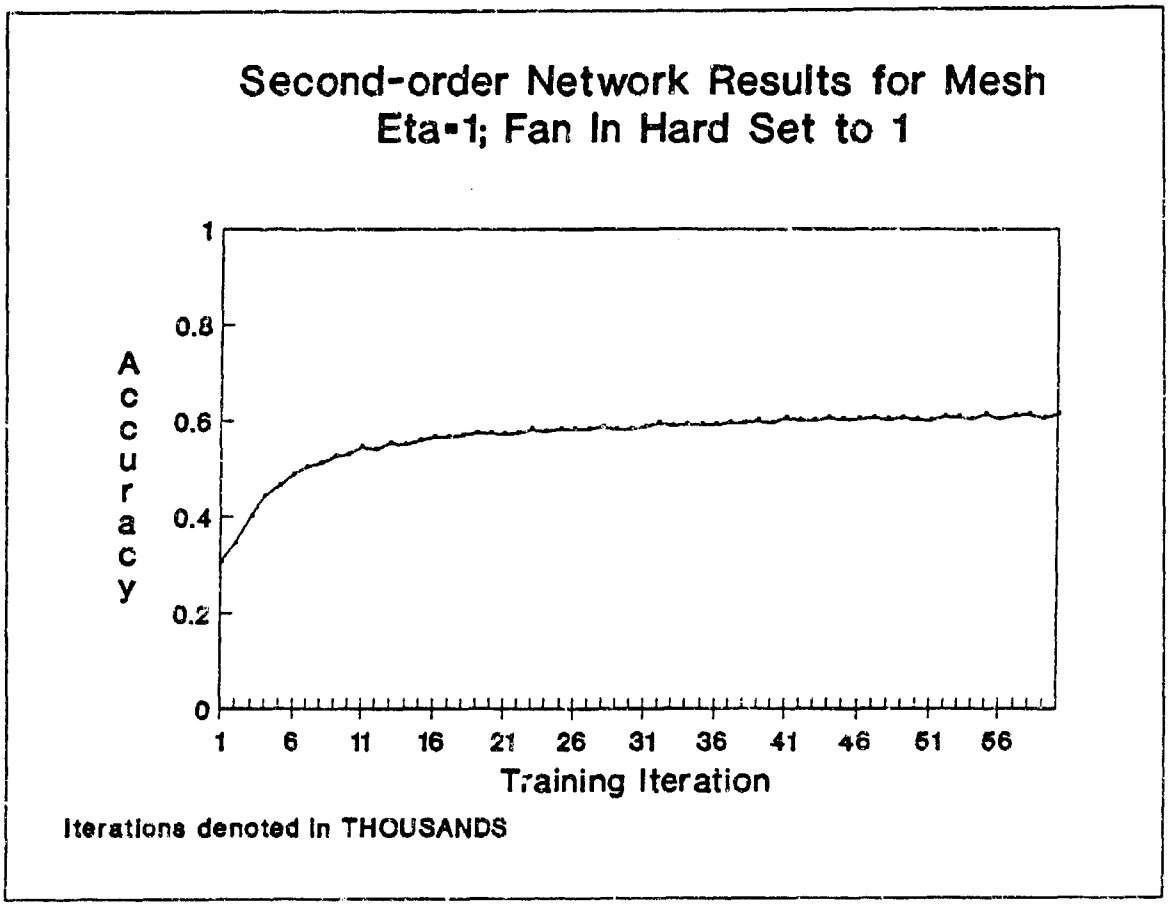


Figure 25. Second-order Network Learning Mesh Data (Eta=1.0)

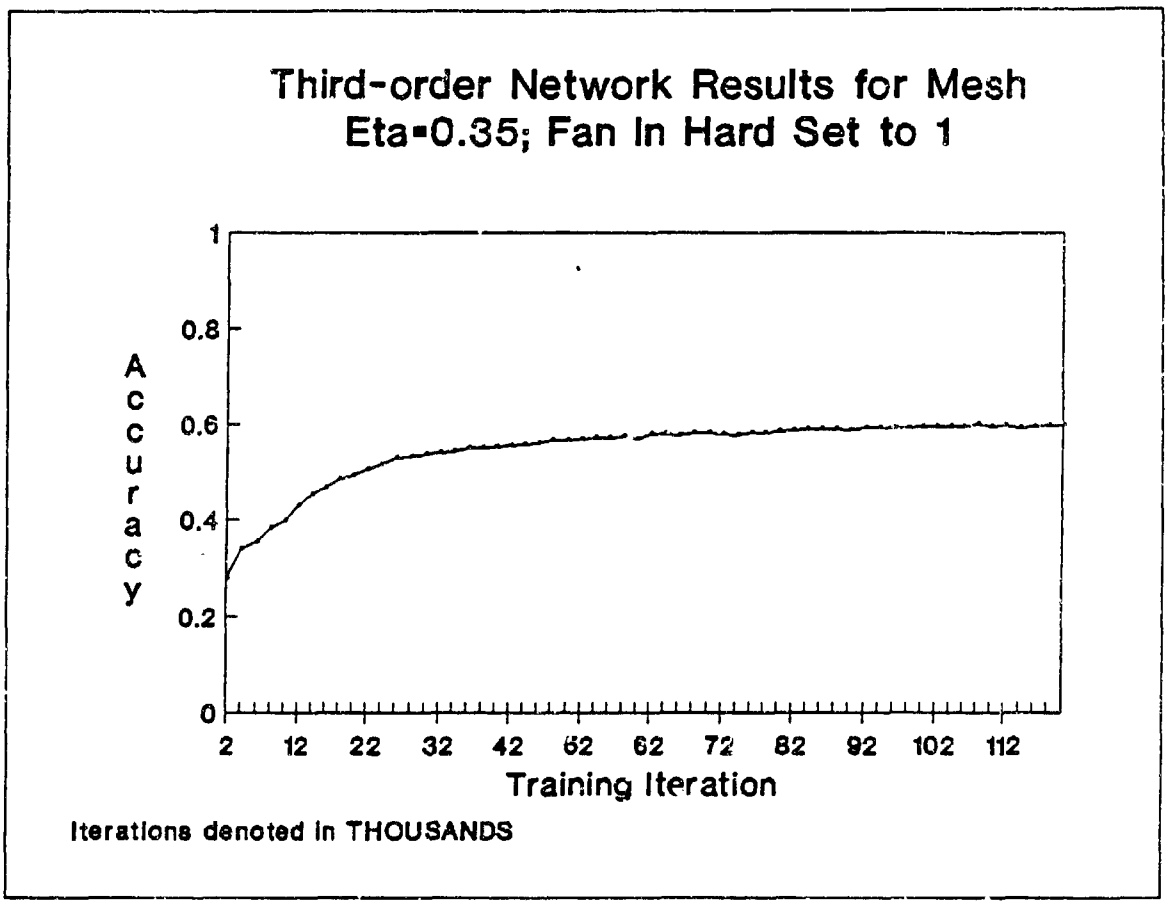


Figure 26. Third-order Network Learning Mesh Data (Eta=0.35)

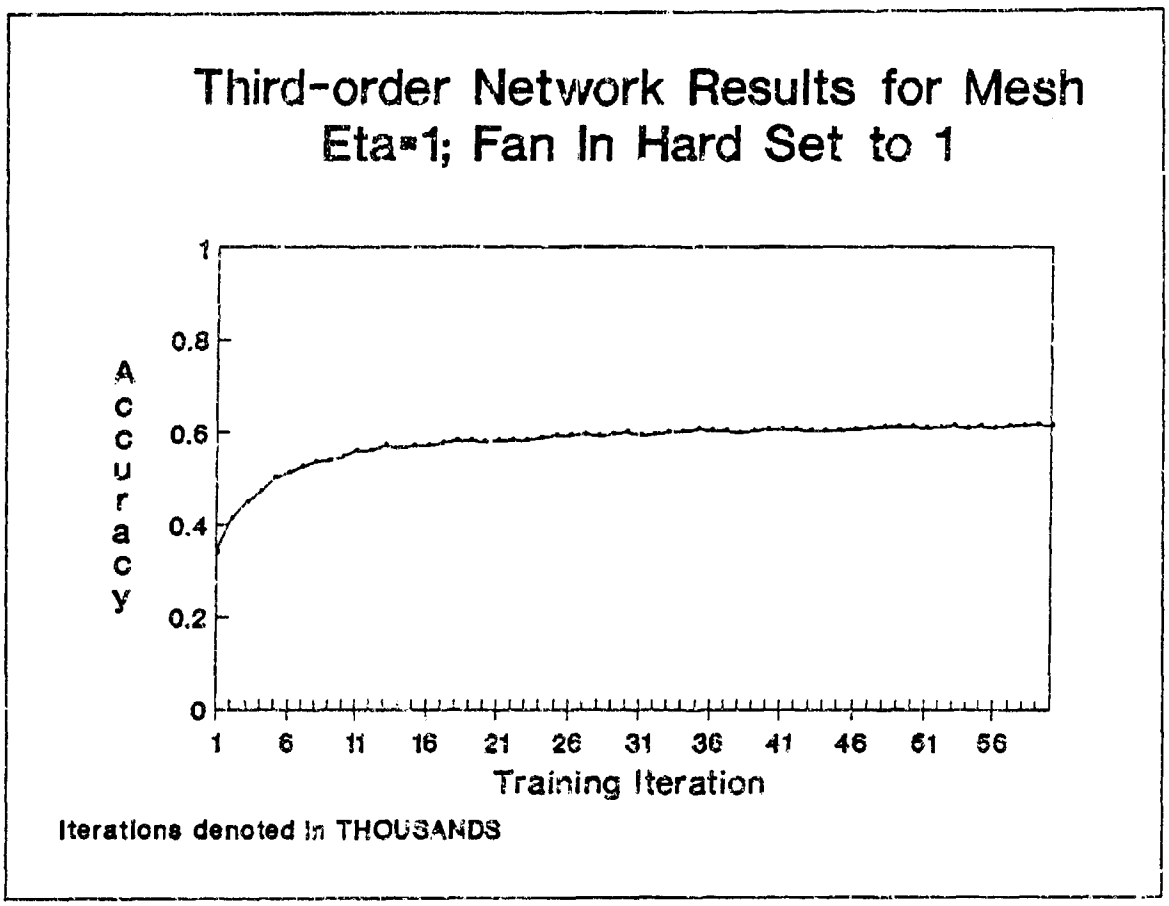


Figure 27. Third-order Network Learning Mesh Data (Eta=1.0)

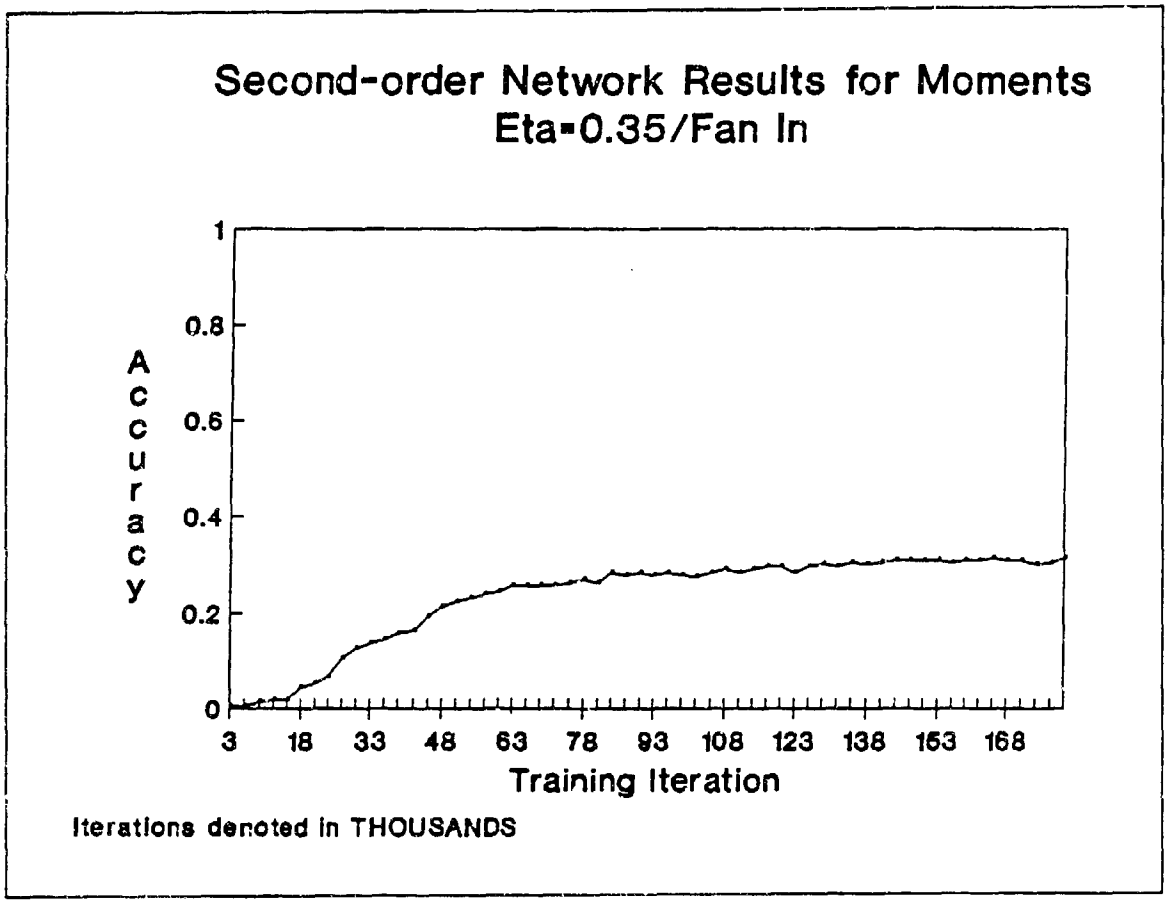


Figure 28. Second-order Network Learning Ruck Data (Eta=0.35)

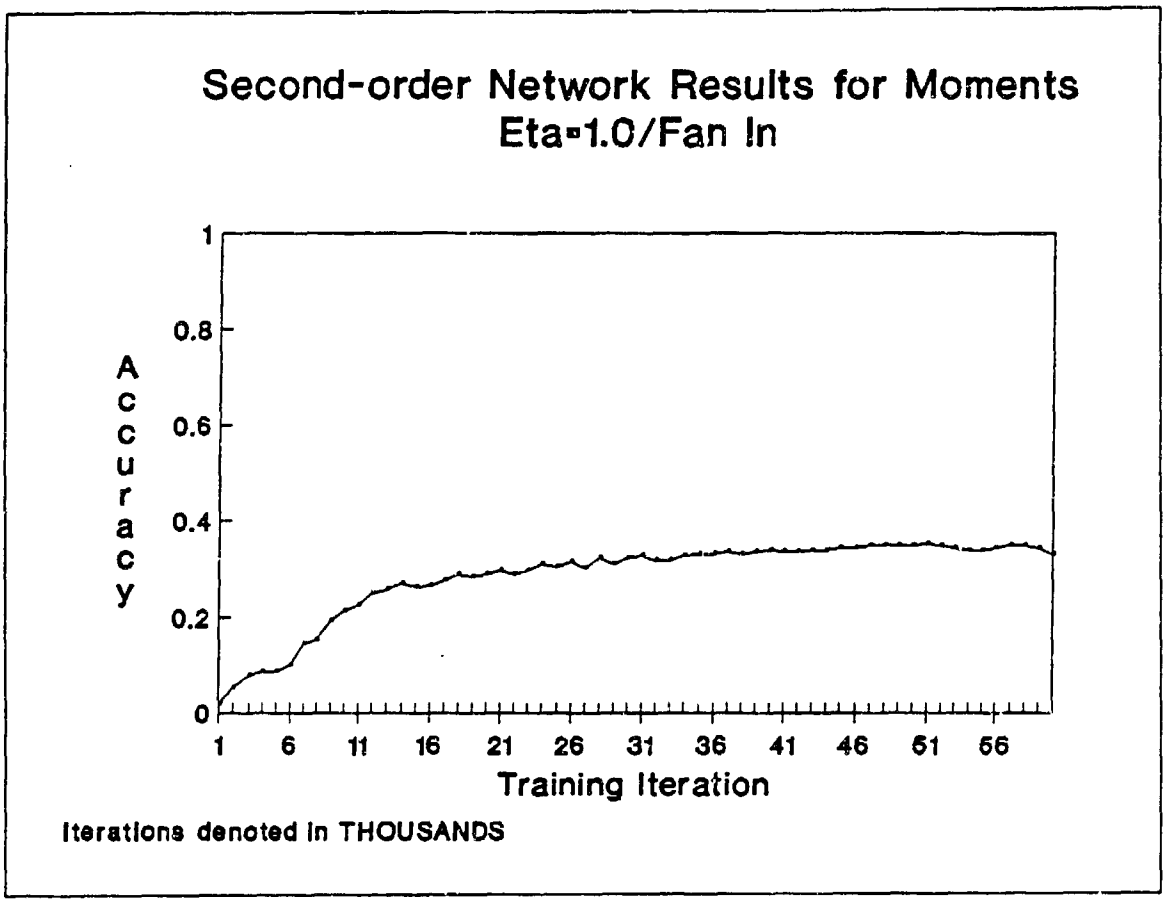


Figure 29. Second-order Network Learning Ruck Data (Eta=1.0)

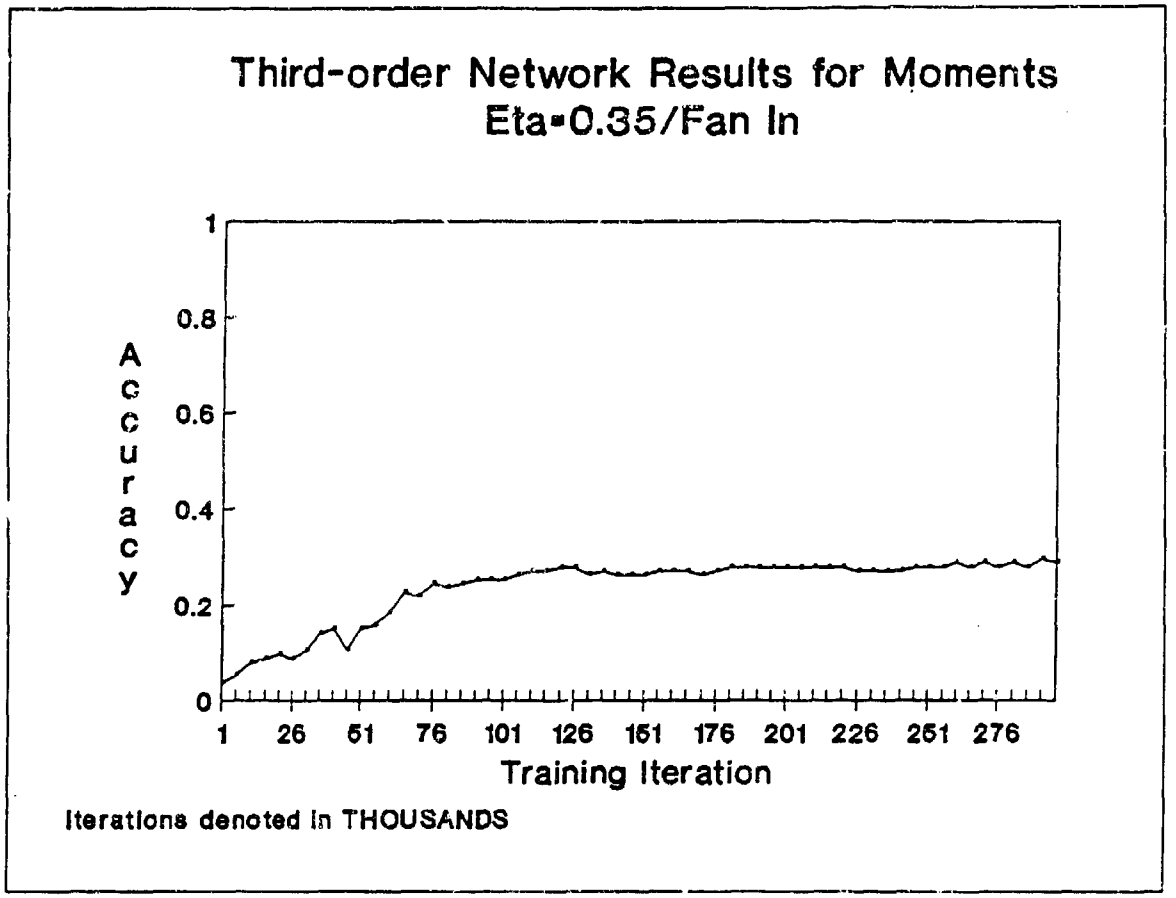


Figure 30. Third-order Network Learning Ruck Data (Eta=0.35)

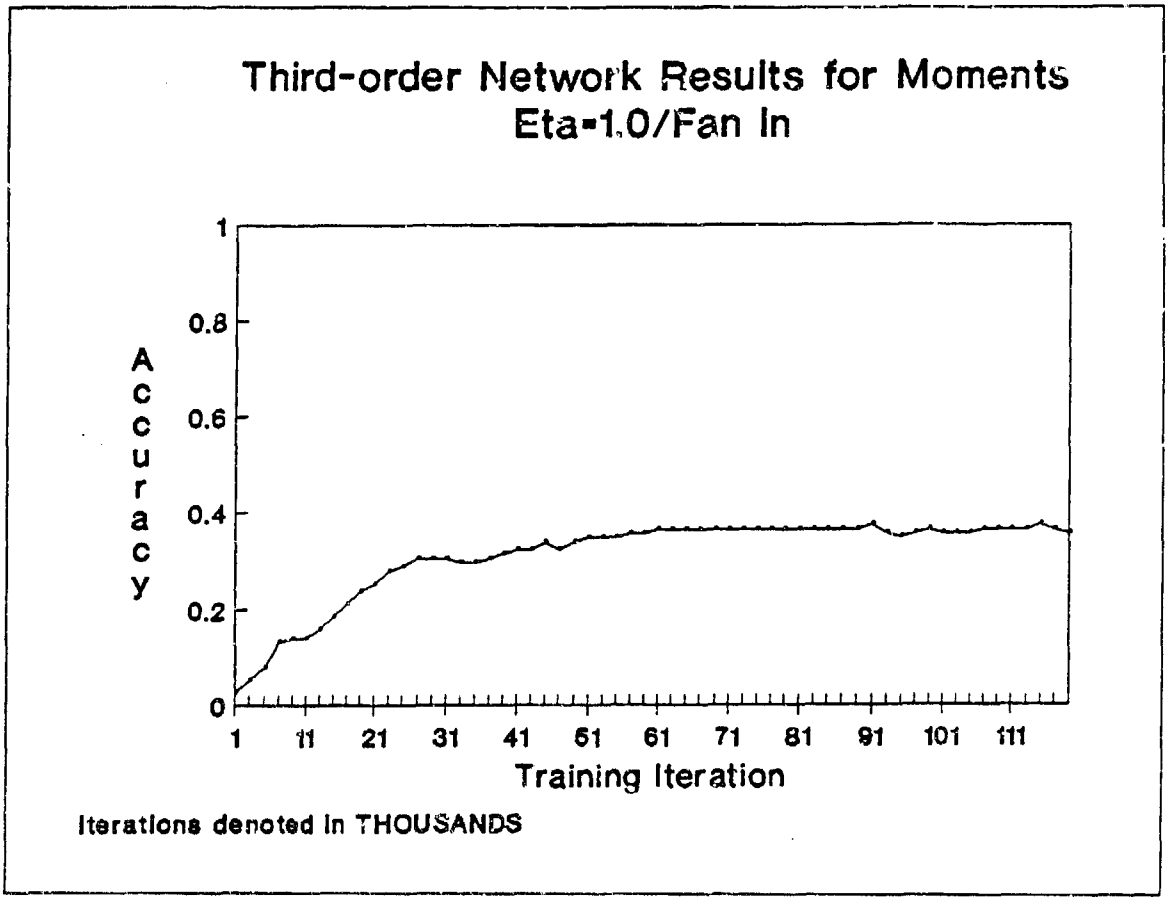


Figure 31. Third-order Network Learning Ruck Data (Eta=1.0)

4.2 Image Classification Network Performance Results

Chapter III discussed details of the proposed image classification algorithm and the implemented variations of its output classification layer. Testing results for the initial version of the network, employing a single layer of perceptrons as the output classification layer, indicated that this scheme was unable to separate the features extracted (without statistical normalization) from the Gabor correlations and phase synchronous summations. Successful learning by this network version would have required linearly separable extracted features. All network versions were tested with learning rates of 0.35 and 1.

Both high-order versions of the image classifier also proved unable to achieve solution convergence on unnormalized features through one million training iterations. As with the previous network variation, these versions employed a single layer of perceptrons for the output classification layer. However, the high-order network implementations fed second-order multiplicative combinations of the extracted features to the output sigmoids. Learning success for these network versions consequently required separability of the second-order combinations of the unnormalized features.

Another network version employed a multilayer perceptron with a variable number of hidden layer nodes at the output classification level. Network testing with this versatile classifier was intended to exhaust research troubleshooting possibilities at the highest processing level of the algorithm. Solution convergence was not achieved through 200,000 training iterations, employing ten, thirty, and fifty nodes in the hidden layer of the perceptron, and using $\eta = 1.0$. Following the initial failure of this version of the network, it was postulated that extracted features might require statistical normalization prior to processing by the classification scheme. A subsequent version of this network, performing statistical normalization on extracted features before feeding them to the multilayer perceptron, exhibited definite learning with increased training iterations. The number of nodes in the hidden layer of

the perceptron classifier were varied using two, three, five, ten, thirty, and fifty hidden nodes. Learning performance results from some of the single program runs are depicted in Figure 32, Figure 33, Figure 34, and Figure 35.

The statistically normalized features were also fed to a high-order network version multiplying second-order combinations of corresponding node locations among the four orientation groups. High-order network runs learned and converged, within a few thousand training iterations, to maximum accuracies less than 10%. This limited performance indicated that second-order combinations of the features were poorly suited for separating the problem classes.

The multilayer perceptron version of the network achieved a 43.2% maximum accuracy, better than the high-order classifier, yet still limited in learning capability. It is likely that the Gabor functions correlated with the input images did not possess the optimum parameters for extracting features to distinguish the problem classes. An ideal set of Gabor functions, with optimum orientations, periodicities, dilations, and window sizes, may yield a significantly higher maximum learned accuracy for the image data set. This is discussed further along with other recommendations in the final chapter.

This chapter covered performance results of the neural network algorithms developed for this thesis project. The following chapter discusses conclusions and recommendations.

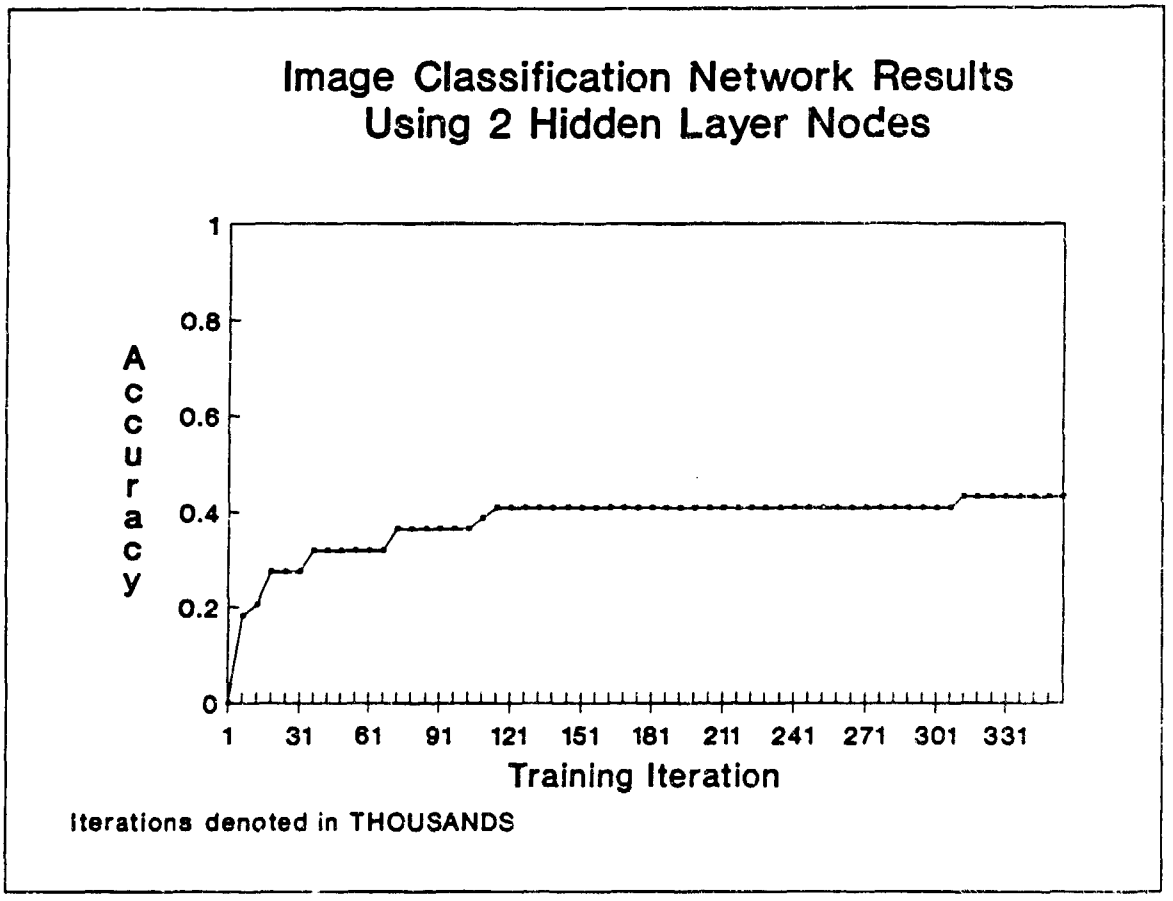


Figure 32. Image Classification Network Results: 2 Hidden Layer Nodes

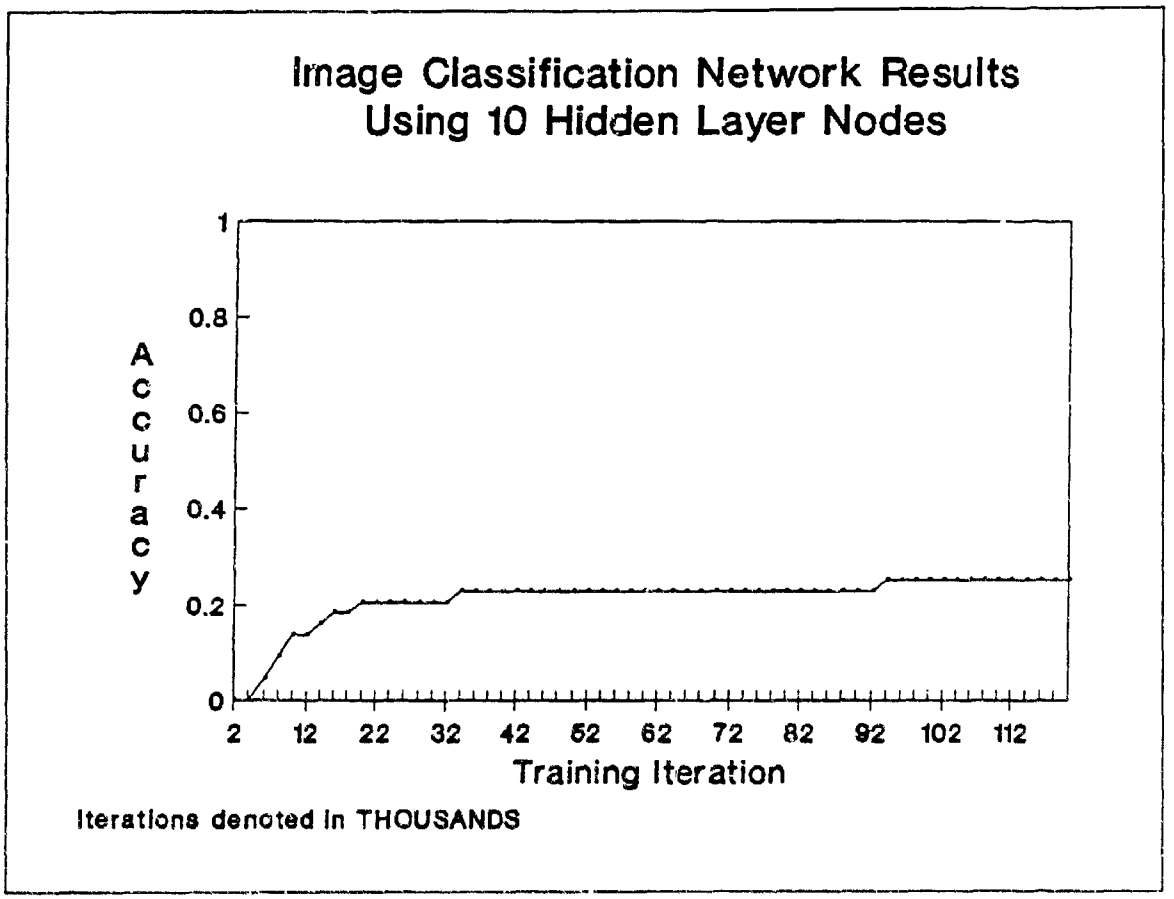


Figure 33. Image Classification Network Results: 10 Hidden Layer Nodes

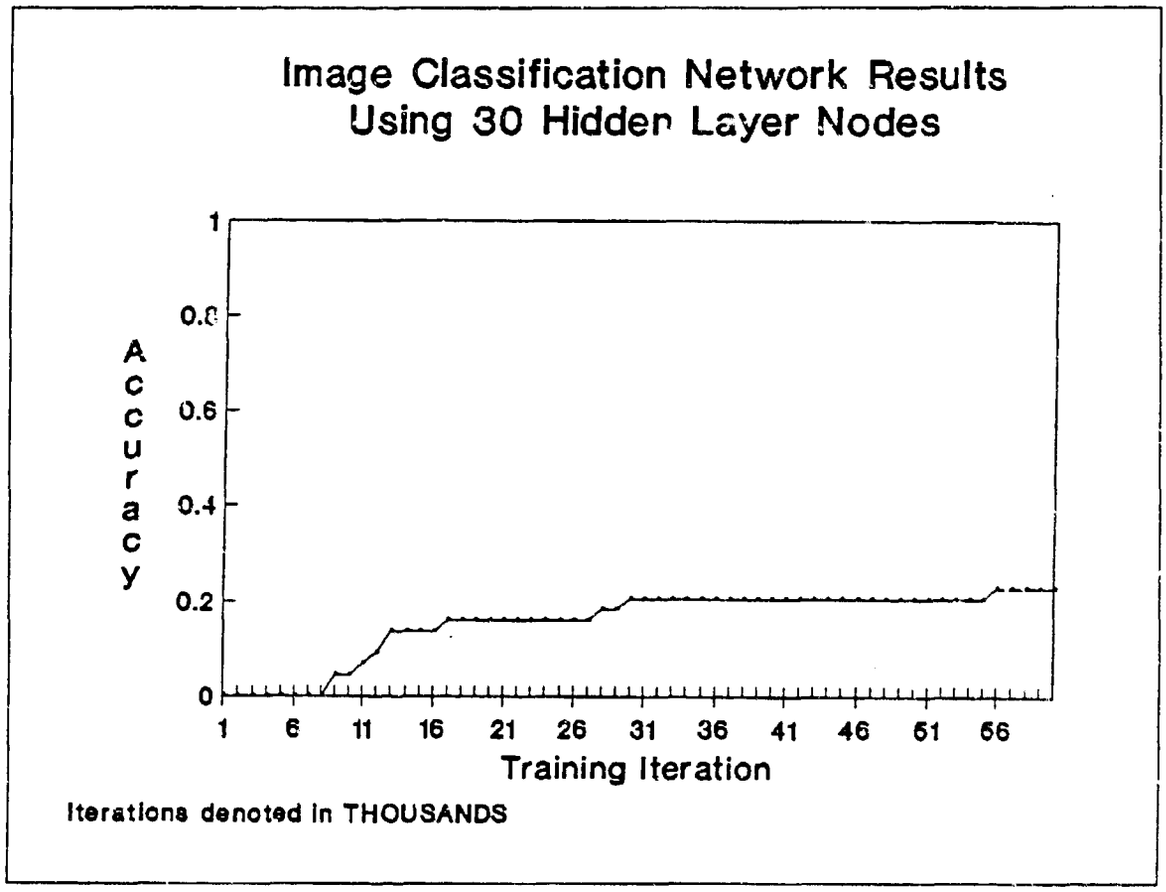


Figure 34. Image Classification Network Results: 30 Hidden Layer Nodes

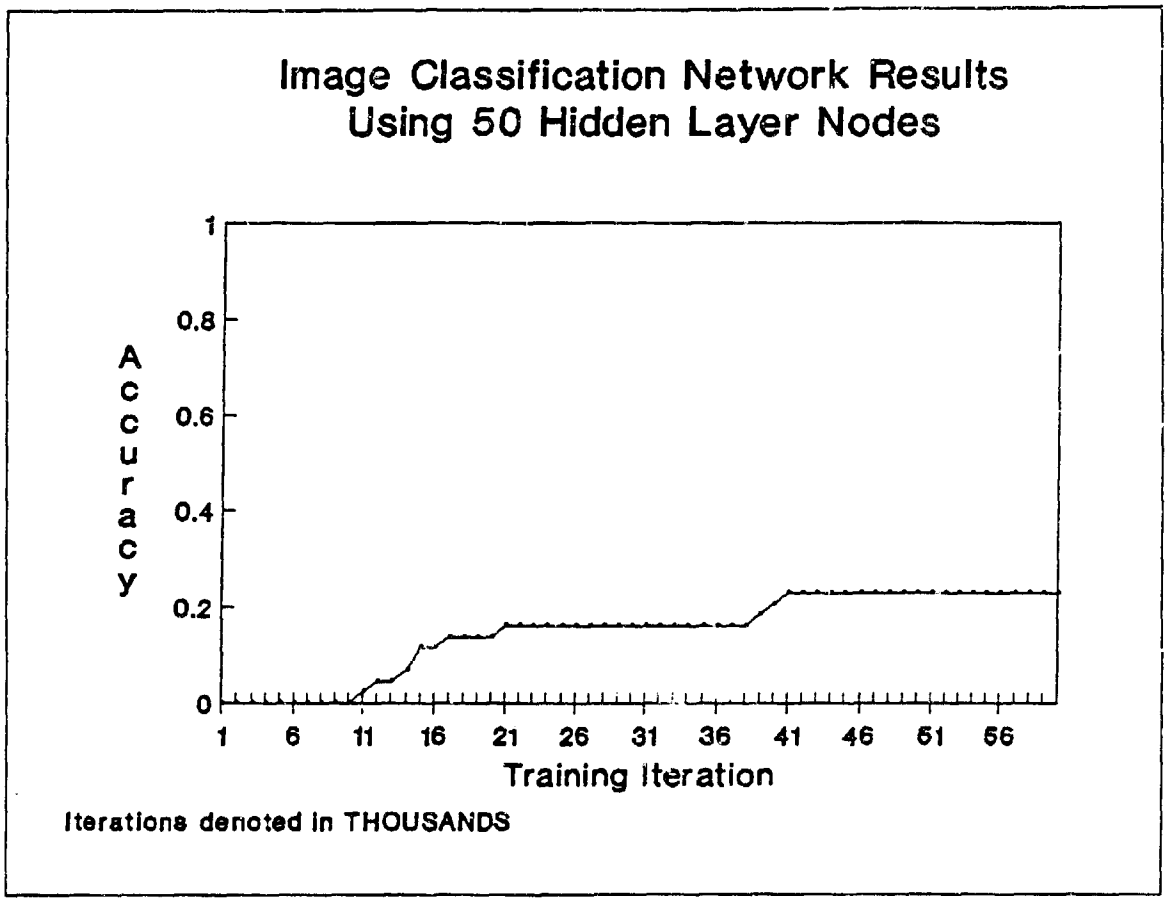


Figure 35. Image Classification Network Results: 50 Hidden Layer Nodes

V. *Conclusions and Recommendations*

The previous chapter detailed testing and performance results for the neural network algorithms developed in this thesis project. This chapter offers conclusions and recommendations for continued research in this area.

5.1 Investigation of the High-Order Neural Network Classifiers

The software-based algorithms for the second and third-order neural networks successfully implemented and tested both high-order classification paradigms. For both networks, learning accuracy increased with sequential training iterations until reaching a maximum accuracy for each problem. For the three classification problems tested in this research, the second-order network generally converged to a maximum learned accuracy in fewer training iterations, while both high-order networks achieved the same approximate maximum learned accuracies for the test problems. Consequently, for the classification problems investigated, the third-order network seemed to offer no advantage over the second-order network. Furthermore, the third-order network was more computationally intense and required significantly more computer run time than the second-order network. However, further research may prove the third-order network more capable for solving some classification problems.

Both networks converged to a learned accuracy of 100% for the XOR problem, the second-order network learning the problem much faster (in fewer iterations) than the third-order network. The high-order networks achieved maximum learned accuracies of approximately 66% on the mesh problem and approximately 32% on the Ruck data. Increasing the learning rate, η, merely reduced the number of training iterations required for solution convergence.

Developing high-order algorithms beyond the third-order may produce networks more capable of solving these classification problems, although each added

order vastly increases computational requirements. This is due to explosive growth in the number of cross-product and exponential combinations of data features that must be calculated and subsequently fed to the sigmoid functions in each node. Also, it is apparent that a high-order network, of a specific order, is limited with respect to the classification problems it can solve.

A lesson learned during the early developmental stages of the high-order algorithms was the need to *normalize all inputs* fed to the artificial neurons. These included the higher-order multiplied combinations of the data feature values (cross products and exponentials) as well as the data set features. If data set feature values are relatively small (between 0 and 1), their higher-order multiplied combinations are even smaller. Miniscule high-order inputs may not contribute to weight updating, thus precluding network learning. Prior to normalizing the high-order inputs, network learning accuracies widely vacillated even after many training iterations, never settling to a maximum learned accuracy. These erratic fluctuations in network accuracy were eliminated following insertion of normalization code modules for the high-order network inputs.

The high-order networks did not outperform feedforward multilayer perceptrons developed by Ruck and Tarr and tested on the same classification problems. In selecting a general classification network for learning a data set, the multilayer perceptron may prove superior in many cases. The multilayer perceptron allows for varying the number of nodes in its hidden layers to "match" the order of a problem and achieve maximum learning. A good, easily modified classifier should be able to separate output classes for many different problems, even with no *a priori* knowledge of the organization of the input feature data. For example, examining a plot of the segregated class regions of the 2-D mesh data (Figure 16), it seems likely that using radial basis functions or $\log \rho, \theta$ inputs in a network might readily solve this specific problem (13). Yet while *a priori* knowledge of a specific problem may be highly useful in developing a network solution, such information is not always available,

particularly for data sets consisting of higher dimensional vectors.

5.2 Investigation of the Image Classification Algorithm

The inability of the initial version of the image classification network to achieve solution convergence prompted the development of subsequent variations in the output classification layer. These network variations, which included two high-order implementations and a multilayer perceptron with a variable number of hidden nodes, were employed to thoroughly test and verify inadequacy of the feature extraction scheme by exhausting possibilities of deficiencies in the output classification layer(s). After employing statistical normalization on extracted features prior to classification, the multilayer perceptron and a high-order network were capable of converging to solutions as learning followed increased training iterations.

To extract data features more suitable for distinguishing designated output classes, future optimization of the algorithm may vary aspects of the feature extraction scheme, including the discrete wavelet correlations and phase synchronizing local averaging routine. Worthy of further exploration would be the use of discrete Gabor functions characterized by different periodicities, orientations, dilations, and window sizes relative to the four Gabor wavelets employed in this research. Varying these aspects in the correlation process would extract different features from the image pixel data. Tarr proposed a methodology in his draft dissertation for selecting optimum Gabor function parameters for image feature extraction (27). The phase synchronization scheme may also be modified to implement inhibition among the four phase synchronized groups derived from the four different Gabor function correlations. Successful future algorithms could implement an energy thresholding scheme in the Lambertization and contrast normalization routine, possibly optimizing the program (13). Varying the degree of overlap for pixel windows in the input plane and node blocks processed by the local averaging routine may also serve to optimize the feature extraction algorithm. The classifier may ultimately be modified

to recognize problem classes with position, scale, and rotation invariance.

This research project employed ELXSI 6400 computers, with single network programs often running for days, and sometimes weeks. Due to the computational intensity of these image classification algorithms operating on pixel data, it may be useful in the future to run software models on a Cray computer. Furthermore, care should be taken to avoid exceeding machine memory capacity when running intensive processes. This problem is exacerbated when using double precision variables to preserve critical small differences in interim program values.

A recommendation that may improve the tallying of network convergence entails the use of less stringent criteria for defining correct network class responses during testing. This thesis research defined correct network responses from the output layer as the in-class node firing greater than 0.8 *and* all other nodes (out-of-class) firing less than 0.2. A less exacting requirement for determining a correct class response might be simply declaring the highest output node as the in-class node.

The FLIR image data set, consisting of tanks, trucks, target boards, and clutter, proved to be a challenging classification problem. Reducing the output classes, or running the image classification network on a more easily separable data set, may also facilitate optimization of the algorithm.

Appendix A. *High-order Network Code*

```
/* Single-layer, second-order neural network algorithm with
   automatic vertical normalization of all input combos.
   Results averaged over 10 runs. */
/* Name: NORMNET2.C */

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

#define numneur 4     /* Specify total number of neurons across slab */
                      /* Number of neurons = number of classes */
#define numexvec 58    /* Specify number of exemplar vectors */
#define numttvec 23    /* Specify number of test vectors */
#define numfeat 22     /* Specify number of features per vector */
                      /* numfeat also represents # FEATURE COLUMNS */
#define numruns 1000    /* Specify number of result plot points */
#define numtrn 1000    /* Specify # training iters between test/plot */

#define eta 1.0      /* Specify training factor */

char *malloc();       /* Must define for use of malloc thru program */
                      /*Need for this statement is compiler-dependent*/
main(){

/* flag[] holds ID number of each data vector, class[] holds training
   class associated with each data vector, x[][] holds feature
   components of data vectors, quadin[][] holds second-order multiplied
  combos of x values, wlin[][] and wquad[][] hold weights for all inputs
   (including higher order combos), theta[] holds threshold values for
   each neuron */

int *flag;
int i, j, n, randnum, numquad, qcount, fanin;
int row, col, avgrun;
int neurtcnt, numright, bigloop, trnloop, testvec, itnum;
float d, y, error, neweta, accuracy, addup;
float *x, *class, *wquad, *quadin;
float theta[numneur], wlin[numneur*numfeat], avrun[10][numruns];
float finalavg[numruns];
float sum, linsum, quadsum, mean, s, sumsqdif;

/* This module opens an input file and reads data into the
   arrays flag[], class[], and x[][] */

FILE *fp;
fp=fopen("ruckdata","r");        /* Insert correct data file name here */
if(fp==NULL) exit(0);

/* Dynamic alloc of arrays prior to loading in data file values */

flag = (int *) malloc((numexvec+numttvec) * sizeof(int));
class = (float *) malloc((numexvec+numttvec) * sizeof(float));
x = (float *) malloc((numexvec+numttvec)*numfeat * sizeof(float));
```

```
for(i=0; i<(numexvec+numttvec); ++i){
fscanf(fp,"%d",&flag[i]);

  for(j=0; j<numfeat; ++j){
  fscanf(fp,"%f",&x[i*numfeat+j]);
  }                          /* end inner loop */

fscanf(fp, "%f",&class[i]);
}                            /* end outer loop */

fclose(fp);

/* This test module prints out input data containing first two and
   final feature components of each vector; checks the above file
   opening module (comment out for final program) */

/* for(i=0; i<(numexvec+numttvec); ++i){
printf("For flag=%d, class=%f, check features are %f %f %f\n",
flag[i], class[i], x[i*numfeat+0], x[i*numfeat+1],
x[i*numfeat+(numfeat-1)]);
} */

/* This module establishes numbers of second-order
   combos of feature inputs FOR EACH VECTOR (without commutative
   redundancy).  Will be used for array allocation. */

numquad=0;
for(i=1; i<=numfeat; i++){
numquad=numquad+i;
}

/* Fill 2-D arrays for second order input combos:
   quadin[numexvec+numttvec][numquad] */

quadin=(float *) malloc((numexvec+numttvec)*numquad*sizeof(float));
for(row=0; row<(numexvec+numttvec); row++){
qcount=0;
  for(i=0; i<numfeat; i++){
    for(j=i; j<numfeat; j++){
    quadin[row*numquad+qcount]=x[row*numfeat+i]*x[row*numfeat+j];
    qcount=qcount+1;
    }
  }
}
/* Test printout module */
/* for(row=0; row<(numexvec+numttvec); row++){
  for(col=0; col<numquad; col++){
  printf("For row=%d  col=%d  quadin=%f\n",row,col,
  quadin[row*numquad+col]);
  }
} */
```

```
/* Begin VERTICAL NORMALIZATION of x, quadin arrays */

/* Vertically normalize x[numexvec+numttvec][numfeat] */
for(col=0; col<numfeat; col++){
sum=0.0;
  for(row=0; row<numexvec; row++){
  sum=sum+x[row*numfeat+col];
  }
mean=sum/(float)(numexvec);

sumsqdif=0.0;
  for(row=0; row<numexvec; row++){
  sumsqdif=sumsqdif+(x[row*numfeat+col]-mean)*(x[row*numfeat+col]-mean);
  }
s=(float)(sqrt(sumsqdif/(float)(numexvec)));
  for(row=0; row<(numexvec+numttvec); row++){
  x[row*numfeat+col]=(x[row*numfeat+col]-mean)/s;
  }
}  /* End column incrementing loop */
/* Test printout loop */
/* for(row=0; row<(numexvec+numttvec); row++){
  for(col=0; col<numfeat; col++){
  printf("For row=%d  col=%d  normxvalue=%f\n",row,col,
  x[row*numfeat+col]);
  }
} */

/* Vertically normalize quadin[numexvec+numttvec][numquad] */
for(col=0; col<numquad; col++){
sum=0.0;
  for(row=0; row<numexvec; row++){
  sum=sum+quadin[row*numquad+col];
  }
mean=sum/(float)(numexvec);

sumsqdif=0.0;
  for(row=0; row<numexvec; row++){
  sumsqdif=sumsqdif+
  (quadin[row*numquad+col]-mean)*(quadin[row*numquad+col]-mean);
  }
s=(float)(sqrt(sumsqdif/(float)(numexvec)));
  for(row=0; row<(numexvec+numttvec); row++){
  quadin[row*numquad+col]=(quadin[row*numquad+col]-mean)/s;
  }
}  /* End column incrementing loop */
/* Test printout loop */
/* for(row=0; row<(numexvec+numttvec); row++){
  for(col=0; col<numquad; col++){
  printf("For row=%d  col=%d  normquadvalue=%f\n",row,col,
  quadin[row*numquad+col]);
  }
} */
/* End of vertical normalization */
```

```
for(avgrun=0; avgrun<10; avgrun++){  /* 10 runs thru program for avg */
srandom((unsigned)time(NULL)); /* Init random() seed off clock */

/* Loop to initialize theta array (one for each neuron) with random
   floating point value between -0.5 and 0.5 */

for(i=0; i<numneur; i++){
theta[i]=(float)(random() % 101)/100.00 - 0.5;
/* printf("For i=%d  theta=%f\n",i,theta[i]); */
}

/* Loop to initialize linear input weights array wlin[numneur][numfeat]
   with random floating point values between -0.5 and 0.5 */

for(n=0; n<numneur; n++){
/* printf("For neuron # %d\n",n); */
  for(i=0; i<numfeat; i++){
  wlin[n*numfeat+i]=(float)(random() % 101)/100.00 - 0.5;
  /* printf("For i=%d  wlin=%f\n",i,wlin[n*numfeat+i]); */
  }
} /* end of nested loop */

fanin = numfeat + numquad;
/* fanin = 1; */
neweta = (float) eta / (float) fanin;
/* printf("neweta= %f\n",neweta); */

/* Loop to initialize quadratic input weights array
   wquad[numneur][numquad] with random floating point values between
   -0.5 and 0.5 */

wquad = (float *) malloc(numneur*numquad*sizeof(float));
for(n=0; n<numneur; n++){
/* printf("For neuron # %d\n",n); */
  for(i=0; i<numquad; i++){
  wquad[n*numquad+i]=(float)(random() % 101)/100.00 - 0.5;
  /* printf("For i=%d quadweight=%f\n",i,wquad[n*numquad+i]); */
  }
}  /* end of nested loop */

/* End of initialization; Begin loops for training and testing */
for(bigloop=1; bigloop<=numruns; bigloop++){

/* Begin training loop */
for(trnloop=0; trnloop<numtrn; trnloop++){

/* Randomly select an exemplar vector row (flag #) */
randnum=(random() % numexvec);
/* printf("Randomly selected vector flag number is %d\n",randnum); */

for(n=0; n<numneur; n++){  /* Loop to update weights for each neuron */
linsum=0.0;
  for(i=0; i<numfeat; i++){
  linsum=linsum+wlin[n*numfeat+i]*x[randnum*numfeat+i];
  }
```

```
quadsum=0.0;
  for(i=0; i<numquad; i++){
  quadsum=quadsum+wquad[n*numquad+i]*quadin[randnum*numquad+i];
  }

sum=linsum+quadsum;
if ((sum+theta[n])<-20.0) {y=0.0;}
  else if ((sum+theta[n])>20.0) {y=1.0;}
    else {y=(float)(1.0/(1.0+exp(-(sum+theta[n]))));}

/* Criteria for training each class to fire a specific neuron */
if (class[randnum]==(float)(n)) {d=1.0;}
else {d=0.0;}

error=y*(1.0-y)*(d-y);
/* printf("error=%f\n",error); */
  for(i=0; i<numfeat; i++){
 wlin[n*numfeat+i]=wlin[n*numfeat+i]+neweta*error*x[randnum*numfeat+i];
  /* printf("For i=%d  wlin=%f\n",i,wlin[n*numfeat+i]); */
  }
  for(i=0; i<numquad; i++){
  wquad[n*numquad+i]=wquad[n*numquad+i]+
  neweta*error*quadin[randnum*numquad+i];
  /* printf("For i=%d  quadweight=%f\n",i,wquad[n*numquad+i]); */
  }

theta[n]=theta[n]+neweta*error;
/* printf("For neuron # %d  theta+%f\n",n,theta[n]); */
}  /* end of weight training loop for each neuron's weights */

}  /* end of complete training loop */

/* Begin test loop to run thru each test vector */
numright=0;
for(testvec=numexvec; testvec<(numexvec+numttvec); testvec++){
/* printf("testvec=%d\n",testvec); */

neurtcnt=0;
  for(n=0; n<numneur; n++){
  linsum=0.0;
    for(i=0; i<numfeat; i++){
    linsum=linsum+wlin[n*numfeat+i]*x[testvec*numfeat+i];
    }
  quadsum=0.0;
    for(i=0; i<numquad; i++){
    quadsum=quadsum+wquad[n*numquad+i]*quadin[testvec*numquad+i];
    }
  sum=linsum+quadsum;

  if ((sum+theta[n])<-20.0) {y=0.0;}
    else if ((sum+theta[n])>20.0) {y=1.0;}
      else {y=(float)(1.0/(1.0+exp(-(sum+theta[n]))));}
  /* printf("y=%f  class[testvec]=%f\n",y,class[testvec]); */
```

```
  /* Test decision criteria for identifying classes */
  if ((y>0.8) && (class[testvec]==(float)(n))) {neurtcnt=neurtcnt+1.}
  if ((y<0.2) && (class[testvec]!=(float)(n))) {neurtcnt=neurtcnt+1.}

  }  /* end of neurons (n) correct test loop */

if (neurtcnt==numneur) {numright=numright+1;}
}  /* end of testing loop running thru each test vector (testvec) */

itnum=bigloop*numtrn;
accuracy=(float)numright/(float)numttvec;
/*printf("Training iteration #: %d   Accuracy=%f\n",itnum,accuracy).*/
avrun[avgrun][bigloop-1]=accuracy;

}  /* end of bigloop */
}  /* end of 10-run loop for averaging */

/* Final averaging routine */
for(col=0; col<numruns; col++){
addup=0.0;
  for(row=0; row<10; row++){
  addup=addup+avrun[row][col];
  }  /* end row add up loop */
finalavg[col]=addup/10.0;
printf("Training Epoch #: %d   Average Accuracy=%f\n",
(col+1)*numtrn,finalavg[col]);
} /* end of column loop */

}  /* end of main */
```

```
/* Single-layer, third-order neural network algorithm with
   automatic vertical normalization of all input combos.
   Results averaged over 10 runs. */
/* Name: NORMNET3.C */

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

#define numneur 2     /* Specify total number of neurons across slab */
                      /* Number of neurons = number of classes */
#define numexvec 4    /* Specify number of exemplar vectors */
#define numttvec 4    /* Specify number of test vectors */
#define numfeat 2     /* Specify number of features per vector */
                      /* numfeat also represents # FEATURE COLUMNS */
#define numruns 300    /* Specify number of result plot points */
#define numtrn 10    /* Specify # training iters between test/plot */

#define eta 0.35      /* Specify training factor */

char *malloc();       /* Must define for use of malloc thru program */
                      /*Need for this statement is compiler-dependent*/
main(){

/* flag[] holds ID number of each data vector, class[] holds training
   class associated with each data vector, x[][] holds feature
   components of data vectors, quadin[][] holds second-order multiplied
   combos of x values, cubein[][] holds third-order multiplied combos
   of x values, wlin[][], wquad[][], and wcube[][] hold weights for all
   inputs (including higher order combos), theta[] holds threshold
   values for each neuron */

int *flag;
int i, j, n, randnum, numquad, numcube, qcount, cubcount, fanin;
int startcnt, upcount, row, col, avgrun;
int neurtcnt, numright, bigloop, trnloop, testvec, itnum;
float d, y, error, neweta, accuracy, addup;
float *x, *class, *wquad, *wcube, *quadin, *cubein;
float theta[numneur], wlin[numneur*numfeat], avrun[10][numruns];
float finalavg[numruns];
float sum, linsum, quadsum, cubesum, mean, s, sumsqdif;

/* This module opens an input file and reads data into the
   arrays flag[], class[], and x[][] */

FILE *fp;
fp=fopen("xordata","r");       /* Insert correct data file name here */
if(fp==NULL) exit(0);

/* Dynamic alloc of arrays prior to loading in data file values */

flag = (int *) malloc((numexvec+numttvec) * sizeof(int));
class = (float *) malloc((numexvec+numttvec) * sizeof(float));
x = (float *) malloc((numexvec+numttvec)*numfeat * sizeof(float));
```

```
for(i=0; i<(numexvec+numttvec); ++i){
fscanf(fp,"%d",&flag[i]);

  for(j=0; j<numfeat; ++j){
  fscanf(fp,"%f",&x[i*numfeat+j]);
  }                          /* end inner loop */

fscanf(fp, "%f",&class[i]);
}                            /* end outer loop */

fclose(fp);

/* This test module prints out input data containing first two and
   final feature components of each vector; checks the above file
   opening module (comment out for final program) */

/* for(i=0; i<(numexvec+numttvec); ++i){
printf("For flag=%d, class=%f, check features are %f %f %f\n",
flag[i], class[i], x[i*numfeat+0], x[i*numfeat+1],
x[i*numfeat+(numfeat-1)]);
} */

/* This module establishes numbers of second-order and third-order
   combos of feature inputs FOR EACH VECTOR (without commutative
   redundancy).  Will be used for array allocation. */

numquad=0;
numcube=0;
for(i=1; i<=numfeat; i++){
numquad=numquad+i;
numcube=numcube+numquad;
}

/* Fill 2-D arrays for second and third order input combos:
   quadin[numexvec+numttvec][numquad] and
   cubein[numexvec+numttvec][numcube] */

quadin=(float *) malloc((numexvec+numttvec)*numquad*sizeof(float));
for(row=0; row<(numexvec+numttvec); row++){
qcount=0;
  for(i=0; i<numfeat; i++){
    for(j=i; j<numfeat; j++){
    quadin[row*numquad+qcount]=x[row*numfeat+i]*x[row*numfeat+j];
    qcount=qcount+1;
    }
  }
}
/* Test printout module */
/* for(row=0; row<(numexvec+numttvec); row++){
  for(col=0; col<numquad; col++){
  printf("For row=%d  col=%d  quadin=%f\n",row,col,
  quadin[row*numquad+col]);
  }
} */
```

```
cubein=(float *) malloc((numexvec+numttvec)*numcube*sizeof(float));
for(row=0; row<(numexvec+numttvec); row++){
cubcount=0;
startcnt=0;
upcount=numfeat;
  for(i=0; i<numfeat; i++){
    for(j=startcnt; j<numquad; j++){
    cubein[row*numcube+cubccunt]=x[row*numfeat+i]*quadin[row*numquad+j];
    cubcount=cubcount+1;
    }
  startcnt=startcnt+upcount;
  upcount=upcount-1;
  }
}
/* Test printout loop */
/* for(row=0; row<(numexvec+numttvec); row++){
  for(col=0; col<numcube; col++){
  printf("For row=%d  col=%d  cubein=%f\n",row,col,
  cubein[row*numcube+col]);
  }
} */

/* Begin VERTICAL NORMALIZATION of x, quadin, cubein arrays */

/* Vertically normalize x[numexvec+numttvec][numfeat] */
for(col=0; col<numfeat; col++){
sum=0.0;
  for(row=0; row<numexvec; row++){
  sum=sum+x[row*numfeat+col];
  }
mean=sum/(float)(numexvec);

sumsqdif=0.0;
  for(row=0; row<numexvec; row++){
  sumsqdif=sumsqdif+(x[row*numfeat+col]-mean)*(x[row*numfeat+col]-mean);
  }
s=(float)(sqrt(sumsqdif/(float)(numexvec)));
  for(row=0; row<(numexvec+numttvec); row++){
  x[row*numfeat+col]=(x[row*numfeat+col]-mean)/s;
  }
}  /* End column incrementing loop */
/* Test printout loop */
/* for(row=0; row<(numexvec+numttvec); row++){
  for(col=0; col<numfeat; col++){
  printf("For row=%d  col=%d  normxvalue=%f\n",row,col,
  x[row*numfeat+col]);
  }
} */
```

```
/* Vertically normalize quadin[numexvec+numttvec][numquad] */

for(col=0; col<numquad; col++){
sum=0.0;
  for(row=0; row<numexvec; row++){
  sum=sum+quadin[row*numquad+col];
  }
mean=sum/(float)(numexvec);

sumsqdif=0.0;
  for(row=0; row<numexvec; row++){
  sumsqdif=sumsqdif+
  (quadin[row*numquad+col]-mean)*(quadin[row*numquad+col]-mean);
  }
s=(float)(sqrt(sumsqdif/(float)(numexvec)));
  for(row=0; row<(numexvec+numttvec); row++){
  quadin[row*numquad+col]=(quadin[row*numquad+col]-mean)/s;
  }
}  /* End column incrementing loop */
/* Test printout loop */
/* for(row=0; row<(numexvec+numttvec); row++){
  for(col=0; col<numquad; col++){
  printf("For row=%d  col=%d  normquadvalue=%f\n",row,col,
  quadin[row*numquad+col]);
  }
} */


/* Vertically normalize cubein[numexvec+numttvec][numcube] */

for(col=0; col<numcube; col++){
sum=0.0;
  for(row=0; row<numexvec; row++){
  sum=sum+cubein[row*numcube+col];
  }
mean=sum/(float)(numexvec);

sumsqdif=0.0;
  for(row=0; row<numexvec; row++){
  sumsqdif=sumsqdif+
  (cubein[row*numcube+col]-mean)*(cubein[row*numcube+col]-mean);
  }
s=(float)(sqrt(sumsqdif/(float)(numexvec)));
  for(row=0; row<(numexvec+numttvec); row++){
  cubein[row*numcube+col]=(cubein[row*numcube+col]-mean)/s;
  }
}  /* End column incrementing loop */
/* Test printout loop */
/* for(row=0; row<(numexvec+numttvec); row++){
  for(col=0; col<numcube; col++){
  printf("For row=%d  col=%d  normcubevalue=%f\n",row,col,
  cubein[row*numcube+col]);
  }
} */
/* End of vertical normalization */
```

```
for(avgrun=0; avgrun<10; avgrun++){  /* 10 runs thru program for avg */
srandom((unsigned)time(NULL)); /* Init random() seed off clock */

/* Loop to initialize theta array (one for each neuron) with random
   floating point values between -0.5 and 0.5 */

for(i=0; i<numneur; i++){
theta[i]=(float)(random() % 101)/100.00 - 0.5;
/* printf("For i=%d  theta=%f\n",i,theta[i]); */
}

/* Loop to initialize linear input weights array wlin[numneur][numfeat]
   with random floating point values between -0.5 and 0.5 */

for(n=0; n<numneur; n++){
/* printf("For neuron # %d\n",n); */
  for(i=0; i<numfeat; i++){
  wlin[n*numfeat+i]=(float)(random() % 101)/100.00 - 0.5;
  /* printf("For i=%d  wlin=%f\n",i,wlin[n*numfeat+i]); */
  }
}  /* end of nested loop */

/* fanin = numfeat + numquad + numcube; */
fanin = 1;
neweta = (float) eta / (float) fanin;
/* printf("neweta= %f\n",neweta); */

/* Loop to initialize quadratic input weights array
   wquad[numneur][numquad] with random floating point values between
   -0.5 and 0.5 */

wquad = (float *) malloc(numneur*numquad*sizeof(float));
for(n=0; n<numneur; n++){
/* printf("For neuron # %d\n",n); */
  for(i=0; i<numquad; i++){
  wquad[n*numquad+i]=(float)(random() % 101)/100.00 - 0.5;
  /* printf("For i=%d  quadweight=%f\n",i,wquad[n*numquad+i]); */
  }
}  /* end of nested loop */

/* Loop to initialize cubic input weights array wcube[numneur][numcube]
   with random floating point values between -0.5 and 0.5 */

wcube = (float *) malloc(numneur*numcube*sizeof(float));
for(n=0; n<numneur; n++){
/* printf("For neuron # %d\n",n); */
  for(i=0; i<numcube; i++){
  wcube[n*numcube+i]=(float)(random() % 101)/100.00 - 0.5;
  /* printf("For i=%d cubeweight=%f\n",i,wcube[n*numcube+i]); */
  }
}  /* end of nested loop */
```

```
/* End of initialization; Begin loops for training and testing */
for(bigloop=1; bigloop<=numruns; bigloop++){

/* Begin training loop */
for(trnloop=0; trnloop<numtrn; trnloop++){

/* Randomly select an exemplar vector row (flag #) */
randnum=(random() % numexvec);
/* printf("Randomly selected vector flag number is %d\n",randnum); */


for(n=0; n<numneur; n++){  /* Loop to update weights for each neuron */
linsum=0.0;
  for(i=0; i<numfeat; i++){
  linsum=linsum+wlin[n*numfeat+i]*x[randnum*numfeat+i];
  }
quadsum=0.0;
  for(i=0; i<numquad; i++){
  quadsum=quadsum+wquad[n*numquad+i]*quadin[randnum*numquad+i];
  }
cubesum=0.0;
  for(i=0; i<numcube; i++){
  cubesum=cubesum+wcube[n*numcube+i]*cubein[randnum*numcube+i];
  }

sum=linsum+quadsum+cubesum;
if ((sum+theta[n])<-20.0) (y=0.0;)
  else if ((sum+theta[n])>20.0) (y=1.0;)
    else (y=(float)(1.0/(1.0+exp(-(sum+theta[n])))));)

/* Criteria for training each class to fire a specific neuron */
if (class[randnum]==(float)(n)) (d=1.0;)
else (d=0.0;)

error=y*(1.0-y)*(d-y);
/* printf("error=%f\n",error); */
  for(i=0; i<numfeat; i++){
 wlin[n*numfeat+i]=wlin[n*numfeat+i]+neweta*error*x[randnum*numfeat+i];
  /* printf("For i=%d  wlin=%f\n",i,wlin[n*numfeat+i]); */
  }
  for(i=0; i<numquad; i++){
  wquad[n*numquad+i]=wquad[n*numquad+i]+
  neweta*error*quadin[randnum*numquad+i];
  /* printf("For i=%d  quadweight=%f\n",i,wquad[n*numquad+i]); */
  }
  for(i=0; i<numcube; i++){
  wcube[n*numcube+i]=wcube[n*numcube+i]+
  neweta*error*cubein[randnum*numcube+i];
  /* printf("For i=%d  cubeweight=%f\n",i,wcube[n*numcube+i]); */
  }

theta[n]=theta[n]+neweta*error;
/* printf("For neuron # %d  theta=%f\n",n,theta[n]); */
}  /* end of weight training loop for each neuron's weights */
}  /* end of complete training loop */
```

```
/* Begin test loop to run thru each test vector */
numright=0;
for(testvec=numexvec; testvec<(numexvec+numttvec); testvec++){
/* printf("testvec=%d\n",testvec); */
neurtcnt=0;
  for(n=0; n<numneur; n++){
  linsum=0.0;
    for(i=0; i<numfeat; i++){
    linsum=linsum+wlin[n*numfeat+i]*x[testvec*numfeat+i];
    }
  quadsum=0.0;
    for(i=0; i<numquad; i++){
    quadsum=quadsum+wquad[n*numquad+i]*quadin[testvec*numquad+i];
    }
  cubesum=0.0;
    for(i=0; i<numcube; i++){
    cubesum=cubesum+wcube[n*numcube+i]*cubein[testvec*numcube+i];
    }
  sum=linsum+quadsum+cubesum;

  if ((sum+theta[n])<-20.0) {y=0.0;}
    else if ((sum+theta[n])>20.0) {y=1.0;}
      else {y=(float)(1.0/(1.0+exp(-(sum+theta[n]))));}
  /* printf("y=%f  class[testvec]=%f\n",y,class[testvec]); */

  /* Test decision criteria for identifying classes */
  if ((y>0.8) && (class[testvec]==(float)(n))) {neurtcnt=neurtcnt+1;}
  if ((y<0.2) && (class[testvec]!=(float)(n))) {neurtcnt=neurtcnt+1;}

  }  /* end of neurons (n) correct test loop */

if (neurtcnt==numneur) {numright=numright+1;}
}  /* end of testing loop running thru each test vector (testvec) */

itnum=bigloop*numtrn;
accuracy=(float)numright/(float)numttvec;
/*printf("Training iteration #: %d   Accuracy=%f\n",itnum,accuracy);*/
avrun[avgrun][bigloop-1]=accuracy;

}  /* end of bigloop */
}  /* end of 10-run loop for averaging */

/* Final averaging routine */
for(col=0; col<numruns; col++){
addup=0.0;
  for(row=0; row<10; row++){
  addup=addup+avrun[row][col];
  }  /* end row add up loop */
finalavg[col]=addup/10.0;
printf("Training Epoch #: %d   Average Accuracy=%f\n",
(col+1)*numtrn,finalavg[col]);
}  /* end of column loop */

}  /* end of main */
```

Appendix B. *Image Classification Network Code*

```
/* Image classification network.  This algorithm processes input
   data sets consisting of an index number, a class number, and
   8064 components per vector (for 63 X 128 image pixel arrays).
   Input "receptive field" windows are set at 8 X 8 pixels.  There
   are two hidden layers feeding four output nodes representing
   tanks, trucks, target boards, and clutter.  Name: IMAGENET.C */

#include <stdio.h>
#include <math.h>
#include <struct.h>

#define numexvec 44  /* Specify number of exemplar vectors */
#define numttvec 44  /* Specify number of test vectors */
#define numfeat 8064 /* Specify number of features per vector */

#define numruns 60   /* Specify number of result plot points */
#define numtrn 1000  /* Specify # training epochs between test/plot */

#define eta 0.35      /* Specify learning rate */
#define pi 3.14159265358979

/* Structure creating 4-D weight array feeding output neurons:
   outnode[i].block2[j].layotwgt[k] */

struct tag1 {
     double layotwgt[270];
};
struct {
     struct tag1 block2[4];
} outnode[4];

char *malloc();

main(){

int *flag, *x, *class, neurtcnt, itnum, gabcount;
int i, j, k, n, bigloop, trnloop, testvec, numright, randnum, bshift;
int rowstpix, colstpix, rowsthl, colsthl, index, lay1posn, lay2posn;
double rfarray[64], absum, avbright, esum, energy, lay2wgt;
double outtheta[4], sum, fx, fy, a, b;
double lay1out[4][434], lay2out[4][270], pharray[25], y[4];
double gabor[4][64];
double d[4], errorout[4];
float accuracy;
/* Open and read in data file to arrays flag[], class[], x[][] */

FILE *fp;
fp=fopen("bigimage","r");
if(fp==NULL) exit(0);

flag = (int *) malloc((numexvec+numttvec) * sizeof(int));
class = (int *) malloc((numexvec+numttvec) * sizeof(int));
x = (int *) malloc((numexvec+numttvec)*numfeat * sizeof(int));
```

```
for(i=0; i<(numexvec+numttvec); i++){
fscanf(fp,"%d",&flag[i]);
  for(j=0; j<numfeat; j++){
  fscanf(fp,"%d",&x[i*numfeat+j]);
  }
fscanf(fp,"%d",&class[i]);
}
fclose(fp);

/* Test printout module: */
/* for(i=0; i<(numexvec+numttvec); i++){
printf("Flag=%d\n",flag[i]);
} */

srandom((unsigned)time(NULL)); /* Init random() seed off clock */

/* Initialize output theta and weight arrays */

for(i=0; i<4; i++){
outtheta[i]=(double)(random() % 101)/100.00 - 0.5;
}

for(i=0; i<4; i++){          /* Normalize initial values by fan in */
outtheta[i]=outtheta[i]/1080.00;
}

for(i=0; i<4; i++){
  for(j=0; j<4; j++){
    for(k=0; k<270; k++){
      outnode[i].block2[j].layotwgt[k]=
      (double)(random() % 101)/100.00 - 0.5;
    }
  }
}

for(i=0; i<4; i++){          /* Normalize initial values by fan in */
  for(j=0; j<4; j++){
    for(k=0; k<270; k++){
    outnode[i].block2[j].layotwgt[k]=
                          (outnode[i].block2[j].layotwgt[k])/1080.00;
    }
  }
}

lay2wgt=1.0/25.0;  /* Constant weight for all 2nd hidden layer nodes */

/* Initialize four Gabor function weight arrays for first layer
 (maximum value of each function is 0.5 to aid sigmoid computation) */

fx=0.0;
fy=1.0;
gabcount=0;
for(i=0; i<=7; i++){
  for(j=0; j<=7; j++){
  a= 0.2*(float)i - 0.7;
```

```
  b= 0.2*(float)j - 0.7;
  gabor[0][gabcount]=
              0.5*exp(-pi*(a*a/3.0+b*b/3.0))*cos(2.0*pi*(fx*a+fy*b));
  gabcount=gabcount+1;
  }
}

fx=1.0;
fy=0.0;
gabcount=0;
for(i=0; i<=7; i++){
  for(j=0; j<=7; j++){
  a= 0.2*(float)i - 0.7;
  b= 0.2*(float)j - 0.7;
  gabor[1][gabcount]=
              0.5*exp(-pi*(a*a/3.0+b*b/3.0))*cos(2.0*pi*(fx*a+fy*b));
  gabcount=gabcount+1;
  }
}

fx=1.4142136;
fy=1.4142136;
gabcount=0;
for(i=0; i<=7; i++){
  for(j=0; j<=7; j++){
  a= 0.2*(float)i - 0.7;
  b= 0.2*(float)j - 0.7;
  gabor[2][gabcount]=
              0.5*exp(-pi*(a*a/3.0+b*b/3.0))*cos(2.0*pi*(fx*a+fy*b));
  gabcount=gabcount+1;
  }
}

fx=1.4142136;
fy=0.0-1.4142136;
gabcount=0;
for(i=0; i<=7; i++){
  for(j=0; j<=7; j++){
  a= 0.2*(float)i - 0.7;
  b= 0.2*(float)j - 0.7;
  gabor[3][gabcount]=
              0.5*exp(-pi*(a*a/3.0+b*b/3.0))*cos(2.0*pi*(fx*a+fy*b));
  gabcount=gabcount+1;
  }
}

/* End of initialization; begin loops for training and testing. */
for(bigloop=1; bigloop<=numruns; bigloop++){

/* Begin training loop */
for(trnloop=0; trnloop<numtrn; trnloop++){
randnum=(random() % numexvec);
```

```
/* Begin propagation thru network */
/* Begin receptive field window shifting loop */
laylposn=0;
for(rowstpix=0; rowstpix<=6656; rowstpix=rowstpix+512){
  for(colstpix=rowstpix; colstpix<=rowstpix+120; colstpix=colstpix+4){
  index=0;
    for(i=colstpix; i<=colstpix+896; i=i+128){
      for(j=i; j<=(i+7); j++){
      rfarray[index]=(double)x[(randnum*8064)+j];
      index=index+1;
      }
    }

/* LAMBERTIZATION AND CONTRAST NORMALIZATION MODULE (processes each
   receptive field window) */
absum=0.0;
for(i=0; i<64; i++){
absum=absum+rfarray[i];
}

avbright=absum/64.0;
for(i=0; i<64; i++){
rfarray[i]=rfarray[i]-avbright;
}

esum=0.0;
for(i=0; i<64; i++){
esum=esum+(rfarray[i]*rfarray[i]);
}

energy=sqrt(esum);
if (energy != 0.0)  {        /* Do not normalize if energy=0.0 */
  for(i=0; i<64; i++){
  rfarray[i]=rfarray[i]/energy;
  }
}  /* end normalization module */

/* Propagate thru to first hidden layer outputs */

for(i=0; i<4; i++){  /* Load each 1st layer block from single window */
sum=0.0;
  for(j=0; j<64; j++){
  sum = sum + gabor[i][j]*rfarray[j];
  } /* Finished calculating one block, one position (first layer) */

if(sum<-20.0) {laylout[i][laylposn]=0.0;}
else if(sum>20.0) {laylout[i][laylposn]=1.0;}
else {laylout[i][laylposn]=(double)(1.0/(1.0+exp(-sum)));}
} /* Finished calculating four blocks, one position (first layer) */

laylposn=laylposn+1;

  }
}  /* end of receptive field window shifting loops (all positions) */
```

```
/* Propagate thru second layer */
for(bshift=0; bshift<4; bshift++){
lay2posn=0;
for(rowsth1=0; rowsth1<=279; rowsth1=rowsth1+31){
  for(colsth1=rowsth1; colsth1<=rowsth1+26; colsth1++){
  index=0;
    for(i=colsth1; i<=colsth1+124; i=i+31){
      for(j=i; j<=(i+4); j++){
      pharray[index]=layout[bshift][j];
      index=index+1;
      }
    }
    sum=0.0;
    for(i=0; i<25; i++){
    sum=sum+pharray[i];
    }
    lay2out[bshift][lay2posn] = sum*lay2wgt;

    lay2posn = lay2posn + 1;
  }
}
}

/* Propagate thru output layer */
for(i=0; i<4; i++){
sum=0.0;
  for(j=0; j<4; j++){
    for(k=0; k<270; k++){
    sum = sum + outnode[i].block2[j].layotwgt[k]*lay2out[j][k];
    }
  }

if((sum+outtheta[i])<-20.0) {y[i]=0.0;}
else if((sum+outtheta[i])>20.0) {y[i]=1.0;}
else {y[i]=(double)(1.0/(1.0+exp(-(sum+outtheta[i]))));}
} /* End calculation for all FOUR output nodes y[4] */
/* End propagation thru network */

/* Backpropagation to update output weights */

for(n=0; n<4; n++){
if (class[randnum] == n) {d[n]=1.0;}
else {d[n]=0.0;}
}  /* Filled desired output values, d[4], based on given class */

for(n=0; n<4; n++){
errorout[n] = y[n]*(1.0-y[n])*(d[n]-y[n]);
}

for(i=0; i<4; i++){
  for(j=0; j<4; j++){
    for(k=0; k<270; k++){
    outnode[i].block2[j].layotwgt[k] =
                  outnode[i].block2[j].layotwgt[k] +
                            (eta/1080.0)*errorout[i]*lay2out[j][k];
```

```
    }
  }
}

for(n=0; n<4; n++){
outtheta[n] = outtheta[n] + (eta/1080.0)*errorout[n];
}
/* end of backprop */

}  /* end of training loop (trnloop) */

/* Begin test loop to run thru each test vector */
numright=0;
for(testvec=numexvec; testvec<(numexvec+numttvec); testvec++){

/* Begin propagation thru network */
/* Begin receptive field window shifting loop */
laylposn=0;
for(rowstpix=0; rowstpix<=6656; rowstpix=rowstpix+512){
  for(colstpix=rowstpix; colstpix<=rowstpix+120; colstpix=colstpix+4){
  index=0;
    for(i=colstpix; i<=colstpix+896; i=i+128){
      for(j=i; j<=(i+7); j++){
      rfarray[index]=(double)x[(testvec*8064)+j];
      index=index+1;
      }
    }

/* LAMBERTIZATION AND CONTRAST NORMALIZATION MODULE (processes each
   receptive field window) */
absum=0.0;
for(i=0; i<64; i++){
absum=absum+rfarray[i];
}

avbright=absum/64.0;
for(i=0; i<64; i++){
rfarray[i]=rfarray[i]-avbright;
}

esum=0.0;
for(i=0; i<64; i++){
esum=esum+(rfarray[i]*rfarray[i]);
}

energy=sqrt(esum);
if (energy != 0.0) {              /* Do not normalize if energy=0.0 */
  for(i=0; i<64; i++){
  rfarray[i]=rfarray[i]/energy;
  }
}  /* end normalization module */
```

```
/* Propagate thru to first hidden layer outputs */

for(i=0; i<4; i++){  /* Load each 1st layer block from single window */
sum=0.0;
  for(j=0; j<64; j++){
  sum = sum + gabor[i][j]*rfarray[j];
  } /* Finished calculating one block, one position (first layer) */

if(sum<-20.0) {laylout[i][laylposn]=0.0;}
else if(sum>20.0) {laylout[i][laylposn]=1.0;}
else {laylout[i][laylposn]=(double)(1.0/(1.0+exp(-sum)));}
} /* Finished calculating four blocks, one position (first layer) */

laylposn=laylposn+1;

  }
}  /* end of receptive field window shifting loops (all positions) */

/* Propagate thru second layer */
for(bshift=0; bshift<4; bshift++){
lay2posn=0;
for(rowsthl=0; rowsthl<=279; rowsthl=rowsthl+31){
  for(colsthl=rowsthl; colsthl<=rowsthl+26; colsthl++){
  index=0;
    for(i=colsthl; i<=colsthl+124; i=i+31){
      for(j=i; j<=(i+4); j++){
      pharray[index]=laylout[bshift][j];
      index=index+1;
      }
    }
    sum=0.0;
    for(i=0; i<25; i++){
    sum=sum+pharray[i];
    }
    lay2out[bshift][lay2posn] = sum*lay2wgt;

    lay2posn = lay2posn + 1;
  }
}
}

/* Propagate thru output layer */
for(i=0; i<4; i++){
sum=0.0;
  for(j=0; j<4; j++){
    for(k=0; k<270; k++){
    sum = sum + outnode[i].block2[j].layotwgt[k]*lay2out[j][k];
    }
  }

if((sum+outtheta[i])<-20.0) {y[i]=0.0;}
else if((sum+outtheta[i])>20.0) {y[i]=1.0;}
else {y[i]=(double)(1.0/(1.0+exp(-(sum+outtheta[i]))));}
} /* End calculation for all FOUR output nodes y[i] */
/* End propagation thru network */
```

```
/* Decision criteria for correct classification */
neurtcnt=0;
for(n=0; n<4; n++){
if((y[n]>0.8) && (class[testvec]==n)) {neurtcnt=neurtcnt+1;}
if((y[n]<0.2) && (class[testvec]!=n)) {neurtcnt=neurtcnt+1;}
} /* End loop for checking all four output neurons */
if(neurtcnt==4) {numright=numright+1;}

}  /* end of testing loop (testvec) */

itnum = bigloop*numtrn;
accuracy = (float)numright/(float)numttvec;
printf("Training Epoch #: %d   Accuracy=%f\n", itnum, accuracy);

}  /* end of bigloop */
}  /* end of main */
```

```
/* Image classification network with high-order (second-order)
   network as output layer.  Passes DOUBLE precision values to
   NORMALIZING routine for second-order processing.  This
   version exceeded machine memory capacity and was scrapped
   before completion.  See later program versions.
   Name: IMAGNET2.C */

#include <stdio.h>
#include <math.h>
#include <struct.h>

#define numexvec 44
#define numttvec 44
#define numvec 88  /* Specify total number of vectors */
#define numfeat 8064 /* Specify number of features per vector */

#define pi 3.14159265358979

struct {
       double x2[4][270];
       double xquad[4][36585];
} vector[88];

char *malloc();

main(){

int *flag, *x, *class, gabcount;
int i, j, n, bigloop, tshift, row, qcount, block;
int rowstpix, colstpix, rowsthl, colsthl, index, laylposn, lay2posn;
double rfarray[64], absum, avbright, esum, energy, lay2wgt;
double sum, fx, fy, a, b, sumsqdif, s, mean;
double laylout[4][434], lay2out[4][270], pharray[25];
double gabor[4][64];



/* Open and read in data file to arrays flag[], class[], x[][] */

FILE *fp;
fp=fopen("bigimage","r");
if(fp==NULL) exit(0);

flag = (int *) malloc(numvec * sizeof(int));
class = (int *) malloc(numvec * sizeof(int));
x = (int *) malloc(numvec * numfeat * sizeof(int));

for(i=0; i<numvec; i++){
fscanf(fp,"%d",&flag[i]);
  for(j=0; j<numfeat; j++){
  fscanf(fp,"%d",&x[i*numfeat+j]);
  }
fscanf(fp,"%d",&class[i]);
}
fclose(fp);
```

```
/* Test printout module: */
/* for(i=0; i<numvec; i++){
printf("Flag=%d  Class=%d\n",flag[i],class[i]);
} */

/* Establish constant "weight" values for Gabor function orientations
   and "phase synchronizing" summation layer */

lay2wgt=1.0/25.0;  /* Constant weight for all 2nd hidden layer nodes */

/* Initialize four Gabor function weight arrays for first layer
 (maximum value of each function is 0.5 to aid sigmoid computation) */

fx=0.0;
fy=1.0;
gabcount=0;
for(i=0; i<=7; i++){
  for(j=0; j<=7; j++){
  a= 0.2*(float)i - 0.7;
  b= 0.2*(float)j - 0.7;
  gabor[0][gabcount]=
              0.5*exp(-pi*(a*a/3.0+b*b/3.0))*cos(2.0*pi*(fx*a+fy*b));
  gabcount=gabcount+1;
  }
}

fx=1.0;
fy=0.0;
gabcount=0;
for(i=0; i<=7; i++){
  for(j=0; j<=7; j++){
  a= 0.2*(float)i - 0.7;
  b= 0.2*(float)j - 0.7;
  gabor[1][gabcount]=
              0.5*exp(-pi*(a*a/3.0+b*b/3.0))*cos(2.0*pi*(fx*a+fy*b));
  gabcount=gabcount+1;
  }
}

fx=1.4142136;
fy=1.4142136;
gabcount=0;
for(i=0; i<=7; i++){
  for(j=0; j<=7; j++){
  a= 0.2*(float)i - 0.7;
  b= 0.2*(float)j - 0.7;
  gabor[2][gabcount]=
              0.5*exp(-pi*(a*a/3.0+b*b/3.0))*cos(2.0*pi*(fx*a+fy*b));
  gabcount=gabcount+1;
  }
}
```

```
fx=1.4142136;
fy=0.0-1.4142136;
gabcount=0;
for(i=0; i<=7; i++){
  for(j=0; j<=7; j++){
  a= 0.2*(float)i - 0.7;
  b= 0.2*(float)j - 0.7;
  gabor[3][gabcount]=
              0.5*exp(-pi*(a*a/3.0+b*b/3.0))*cos(2.0*pi*(fx*a+fy*b));
  gabcount=gabcount+1;
  }
}

/* End of module establishing weights.  Begin loop thru all vectors. */

for(bigloop=0; bigloop<numvec; bigloop++){

/* Begin propagation thru network */
/* Begin receptive field window shifting loop */
laylposn=0;
for(rowstpix=0; rowstpix<=6656; rowstpix=rowstpix+512){
  for(colstpix=rowstpix; colstpix<=rowstpix+120; colstpix=colstpix+4){
  index=0;
    for(i=colstpix; i<=colstpix+896; i=i+128){
      for(j=i; j<=(i+7); j++){
      rfarray[index]=(double)x[(bigloop*8064)+j];
      index=index+1;
      }
    }

/* LAMBERTIZATION AND CONTRAST NORMALIZATION MODULE (processes each
   receptive field window) */
absum=0.0;
for(i=0; i<64; i++){
absum=absum+rfarray[i];
}

avbright=absum/64.0;
for(i=0; i<64; i++){
rfarray[i]=rfarray[i]-avbright;
}

esum=0.0;
for(i=0; i<64; i++){
esum=esum+(rfarray[i]*rfarray[i]);
}

energy=sqrt(esum);
if (energy != 0.0)  {        /* Do not normalize if energy=0.0 */
  for(i=0; i<64; i++){
  rfarray[i]=rfarray[i]/energy;
  }
}  /* end normalization module */
```

```
/* Propagate thru to first hidden layer outputs */

for(i=0; i<4; i++){  /* Load each 1st layer block from single window */
sum=0.0;
  for(j=0; j<64; j++){
  sum = sum + gabor[i][j]*rfarray[j];
  } /* Finished calculating one block, one position (first layer) */

if(sum<-20.0) {laylout[i][laylposn]=0.0;}
else if(sum>20.0) {laylout[i][laylposn]=1.0;}
else {laylout[i][laylposn]=(double)(1.0/(1.0+exp(-sum)));}
} /* Finished calculating four blocks, one position (first layer) */

laylposn=laylposn+1;

  }
}  /* end of receptive field window shifting loops (all positions) */

/* Propagate thru second layer */
for(bshift=0; bshift<4; bshift++){
lay2posn=0;
for(rowsth1=0; rowsth1<=279; rowsth1=rowsth1+31){
  for(colsth1=rowsth1; colsth1<=rowsth1+26; colsth1++){
  index=0;
    for(i=colsth1; i<=colsth1+124; i=i+31){
      for(j=i; j<=(i+4); j++){
      pharray[index]=laylout[bshift][j];
      index=index+1;
      }
    }
    sum=0.0;
    for(i=0; i<25; i++){
    sum=sum+pharray[i];
    }
    lay2out[bshift][lay2posn] = sum*lay2wgt;

    lay2posn = lay2posn + 1;
  }
}
}
/* End propagation of one vector thru output of summation layer */

/* For each run thru bigloop (over each new vector), fill structure
   array vector[88].x2[4][270] with propagated results from outputs
   of summation layer.  The filled x2 array will be the input for
   the final, high-order layer. */

for(i=0; i<4; i++){
  for(j=0; j<270; j++){
  vector[bigloop].x2[i][j] = lay2out[i][j];
  }
}

} /* end of bigloop */
```

```
/* Begin high-order modules */
/* Fill arrays for second-order input combos
   vector[88].xquad[4][36585] */

for(row=0; row<numvec; row++){
  for(block=0; block<4; block++){
  qcount=0;
    for(i=0; i<270; i++){
      for(j=i; j<270; j++){
      vector[row].xquad[block][qcount] =
                  vector[row].x2[block][i] * vector[row].x2[block][j];
      qcount=qcount+1;
      }
    }
  }
}


}  /* end of main */
```

```
/* Image classification network with high-order (second-order)
   network as output layer.  Passes DOUBLE precision values
   for second-order processing.  Second-order combinations
   are multiplied among the 270 nodes within each of the
   four separate orientation groups of the phase
   synchronization layer outputs.
   Name: IMAGNET3.C */

#include <stdio.h>
#include <math.h>
#include <struct.h>

#define numexvec 44
#define numttvec 44
#define numvec 88     /* Specify total number of vectors */
#define numfeat 8064 /* Specify number of features per vector */

#define numruns 60    /* Specify number of result plot points */
#define numtrn 1000   /* Specify # training iters between test/plot */

#define eta 0.35

#define pi 3.14159265358979

struct {
       double x2[4][270];
} vector[88];

struct {
       double linwgt[4][270];
       double quadwgt[4][36585];
} neuron[4];

char *malloc();

main(){

int *flag, *x, *class, gabcount, fulloop, trnloop, randnum, neurtcnt;
int i, j, n, bigloop, bshift, row, qcount, block, testvec, numright;
int rowstpix, colstpix, rowsthl, colsthl, index, laylposn, lay2posn;
int itnum;
double rfarray[64], absum, avbright, esum, energy, lay2wgt;
double sum, fx, fy, a, b, linsum, quadsum, y, d, error;
double laylout[4][434], lay2out[4][270], pharray[25];
double gabor[4][64], theta[4], xquad[4][36585];
float accuracy;


/* Open and read in data file to arrays flag[], class[], x[][] */

FILE *fp;
fp=fopen("bigimage","r");
if(fp==NULL) exit(0);
```

```
flag = (int *) malloc(numvec * sizeof(int));
class = (int *) malloc(numvec * sizeof(int));
x = (int *) malloc(numvec * numfeat * sizeof(int));

for(i=0; i<numvec; i++){
fscanf(fp,"%d",&flag[i]);
  for(j=0; j<numfeat; j++){
  fscanf(fp,"%d",&x[i*numfeat+j]);
  }
fscanf(fp,"%d",&class[i]);
}
fclose(fp);

/* Test printout module: */
/* for(i=0; i<numvec; i++){
printf("Flag=%d  Class=%d\n",flag[i],class[i]);
} */

/* Establish constant "weight" values for Gabor function orientations
   and "phase synchronizing" summation layer */

lay2wgt=1.0/25.0;  /* Constant weight for all 2nd hidden layer nodes */

/* Initialize four Gabor function weight arrays for first layer
 (maximum value of each function is 0.5 to aid sigmoid computation) */

fx=0.0;
fy=1.0;
gabcount=0;
for(i=0; i<=7; i++){
  for(j=0; j<=7; j++){
  a= 0.2*(float)i - 0.7;
  b= 0.2*(float)j - 0.7;
  gabor[0][gabcount]=
              0.5*exp(-pi*(a*a/3.0+b*b/3.0))*cos(2.0*pi*(fx*a+fy*b));
  gabcount=gabcount+1;
  }
}

fx=1.0;
fy=0.0;
gabcount=0;
for(i=0; i<=7; i++){
  for(j=0; j<=7; j++){
  a= 0.2*(float)i - 0.7;
  b= 0.2*(float)j - 0.7;
  gabor[1][gabcount]=
              0.5*exp(-pi*(a*a/3.0+b*b/3.0))*cos(2.0*pi*(fx*a+fy*b));
  gabcount=gabcount+1;
  }
}
```

```
fx=1.4142136;
fy=1.4142136;
gabcount=0;
for(i=0; i<=7; i++){
  for(j=0; j<=7; j++){
  a= 0.2*(float)i - 0.7;
  b= 0.2*(float)j - 0.7;
  gabor[2][gabcount]=
              0.5*exp(-pi*(a*a/3.0+b*b/3.0))*cos(2.0*pi*(fx*a+fy*b));
  gabcount=gabcount+1;
  }
}

fx=1.4142136;
fy=0.0-1.4142136;
gabcount=0;
for(i=0; i<=7; i++){
  for(j=0; j<=7; j++){
  a= 0.2*(float)i - 0.7;
  b= 0.2*(float)j - 0.7;
  gabor[3][gabcount]=
              0.5*exp(-pi*(a*a/3.0+b*b/3.0))*cos(2.0*pi*(fx*a+fy*b));
  gabcount=gabcount+1;
  }
}

/* End of module establishing weights.  Begin loop thru all vectors. */

for(bigloop=0; bigloop<numvec; bigloop++){

/* Begin propagation thru network */
/* Begin receptive field window shifting loop */
laylposn=0;
for(rowstpix=0; rowstpix<=6656; rowstpix=rowstpix+512){
  for(colstpix=rowstpix; colstpix<=rowstpix+120; colstpix=colstpix+4){
  index=0;
    for(i=colstpix; i<=colstpix+896; i=i+128){
      for(j=i; j<=(i+7); j++){
      rfarray[index]=(double)x[(bigloop*8064)+j];
      index=index+1;
      }
    }

/* LAMBERTIZATION AND CONTRAST NORMALIZATION MODULE (processes each
   receptive field window) */
absum=0.0;
for(i=0; i<64; i++){
absum=absum+rfarray[i];
}

avbright=absum/64.0;
for(i=0; i<64; i++){
rfarray[i]=rfarray[i]-avbright;
}
```

```
esum=0.0;
for(i=0; i<64; i++){
esum=esum+(rfarray[i]*rfarray[i]);
}

energy=sqrt(esum);
if (energy != 0.0)  {          /* Do not normalize if energy=0.0 */
  for(i=0; i<64; i++){
  rfarray[i]=rfarray[i]/energy;
  }
}  /* end normalization module */

/* Propagate thru to first hidden layer outputs */

for(i=0; i<4; i++){  /* Load each 1st layer block from single window */
sum=0.0;
  for(j=0; j<64; j++){
  sum = sum + gabor[i][j]*rfarray[j];
  } /* Finished calculating one block, one position (first layer) */

if(sum<-20.0) {laylout[i][laylposn]=0.0;}
else if(sum>20.0) {laylout[i][laylposn]=1.0;}
else {laylout[i][laylposn]= (double)(1.0/(1.0+exp(-sum)));}
} /* Finished calculating four blocks, one position (first layer) */

laylposn=laylposn+1;

  }
}  /* end of receptive field window shifting loops (all positions) */

/* Propagate thru second layer */
for(bshift=0; bshift<4; bshift++){
lay2posn=0;
for(rowsth1=0; rowsth1<=279; rowsth1=rowsth1+31){
  for(colsth1=rowsth1; colsth1<=rowsth1+26; colsth1++){
  index=0;
    for(i=colsth1; i<=colsth1+124; i=i+31){
      for(j=i; j<=(i+4); j++){
      pharray[index]=laylout[bshift][j];
      index=index+1;
      }
    }
    sum=0.0;
    for(i=0; i<25; i++){
    sum=sum+pharray[i];
    }
    lay2out[bshift][lay2posn] = sum*lay2wgt;

    lay2posn = lay2posn + 1;
  }
}
}
/* End propagation of one vector thru output of summation layer */
```

```
/* For each run thru bigloop (over each new vector), fill structure
   array vector[88].x2[4][270] with propagated results from outputs
   of summation layer.  The filled x2 array will be the input for
   the final, high-order layer. */

for(i=0; i<4; i++){
  for(j=0; j<270; j++){
  vector[bigloop].x2[i][j] = lay2out[i][j];
  }
}
} /* end of bigloop */

/* Begin high-order modules */
/* Initialization */
srandom((unsigned)time(NULL));   /* Init random() seed off clock */

for(n=0; n<4; n++){
  for(block=0; block<4; block++){
    for(j=0; j<270; j++){
    neuron[n].linwgt[block][j] = (double)(random() % 101)/100.00 - 0.5;
    neuron[n].linwgt[block][j] = neuron[n].linwgt[block][j]/147420.00;
    }
  }
}

for(n=0; n<4; n++){
  for(block=0; block<4; block++){
    for(j=0; j<36585; j++){
    neuron[n].quadwgt[block][j]=(double)(random() % 101)/100.00 - 0.5;
    neuron[n].quadwgt[block][j]=neuron[n].quadwgt[block][j]/147420.00;
    }
  }
}

for(n=0; n<4; n++){
theta[n] = (double)(random() % 101)/100.00 - 0.5;
theta[n] = theta[n]/147420.00;
}
/* End of initialization; Begin training and testing loops */

for(fulloop=1; fulloop<=numruns; fulloop++){
for(trnloop=0; trnloop<numtrn; trnloop++){
randnum=(random() % numexvec);  /* Select random exemplar vector */

/* Multiply out second-order combinations for present exemplar */
for(block=0; block<4; block++){
qcount=0;
  for(i=0; i<270; i++){
    for(j=i; j<270; j++){
    xquad[block][qcount] =
           vector[randnum].x2[block][i] * vector[randnum].x2[block][j];
    qcount=qcount+1;
    }
  }
}
```

```
for(n=0; n<4; n++){     /* Loop to change neuron */
linsum=0.0;
  for(block=0; block<4; block++){
    for(j=0; j<270; j++){
    linsum = linsum +
             neuron[n].linwgt[block][j]*vector[randnum].x2[block][j];
    }
  } /* end block */

quadsum=0.0;
  for(block=0; block<4; block++){
    for(j=0; j<36585; j++){
    quadsum = quadsum + neuron[n].quadwgt[block][j]*xquad[block][j];
    }
  } /* end of block */

sum = linsum + quadsum;

if ((sum+theta[n])<-20.0) {y=0.0;}
else if ((sum+theta[n])>20.0) {y=1.0;}
else {y=(double)(1.0/(1.0+exp(-(sum+theta[n]))));}

/* Criteria for training each class to fire a specific neuron */
if (class[randnum]==n) {d=1.0;}
else {d=0.0;}

error=y*(1.0-y)*(d-y);

/* Update weights */
for(block=0; block<4; block++){
  for(j=0; j<270; j++){
  neuron[n].linwgt[block][j] = neuron[n].linwgt[block][j] +
                  (eta/147420.00)*error*vector[randnum].x2[block][j];
  }
}
for(block=0; block<4; block++){
  for(j=0; j<36585; j++){
  neuron[n].quadwgt[block][j] = neuron[n].quadwgt[block][j] +
                        (eta/147420.00)*error*xquad[block][j];
  }
}
theta[n] = theta[n] + (eta/147420.00)*error;

} /* End of neuron (n) changing loop */
} /* End of trnloop */

/* Begin test loop to run thru each test vector */
numright=0;
for(testvec=numexvec; testvec<numvec; testvec++){
neurtcnt=0;
/* Multiply out second-order combinations for present test vector */
for(block=0; block<4; block++){
qcount=0;
  for(i=0; i<270; i++){
    for(j=i; j<270; j++){
```

```
    xquad[block][qcount] =
           vector[testvec].x2[block][i] * vector[testvec].x2[block][j];
    qcount=qcount+1;
    }
  }
}

for(n=0; n<4; n++){    /* Loop to change neuron */
linsum=0.0;
  for(block=0; block<4; block++){
    for(j=0; j<270; j++){
    linsum = linsum +
             neuron[n].linwgt[block][j]*vector[testvec].x2[block][j];
    }
  } /* end block */

quadsum=0.0;
  for(block=0; block<4; block++){
    for(j=0; j<36585; j++){
    quadsum = quadsum + neuron[n].quadwgt[block][j]*xquad[block][j];
    }
  } /* end of block */

sum = linsum + quadsum;

if ((sum+theta[n])<-20.0) {y=0.0;}
else if ((sum+theta[n])>20.0) {y=1.0;}
else {y=(double)(1.0/(1.0+exp(-(sum+theta[n]))));}

/* Test decision criteria for identifying classes */
if ((y>0.8) && (class[testvec]==n)) {neurtcnt=neurtcnt+1;}
if ((y<0.2) && (class[testvec]!=n)) {neurtcnt=neurtcnt+1;}

} /* End of neuron (n) changing loop */
if (neurtcnt==4) {numright=numright+1;}
} /* End of test loop running thru each test vector (testvec) */

itnum=fulloop*numtrn;
accuracy=(float)numright/(float)numttvec;
printf("Training Epoch #: %d   Accuracy=%f\n", itnum, accuracy);

} /* End of fulloop */
}  /* end of main */
```

```
/* Image classification network with high-order (second-order)
   network as output layer.  Passes DOUBLE precision values
   for second-order processing.  Multiplies out second-order
   combos among the four orientations for each of the 270
   nodes (per orientation) feeding final layer.
   Name: IMAGNET4.C */

#include <stdio.h>
#include <math.h>
#include <struct.h>

#define numexvec 44
#define numttvec 44
#define numvec 88     /* Specify total number of vectors */
#define numfeat 8064 /* Specify number of features per vector */

#define numruns 1000   /* Specify number of result plot points */
#define numtrn 1000  /* Specify # training iters between test/plot */

#define eta 1.0

#define pi 3.14159265358979

struct {
       double x2[4][270];
} vector[88];

struct {
       double linwgt[4][270];
       double quadwgt[270][10];
} neuron[4];

char *malloc();

main(){

int *flag, *x, *class, gabcount, fulloop, trnloop, randnum, neurtcnt;
int i, j, n, bigloop, bshift, row, qcount, block, testvec, numright;
int rowstpix, colstpix, rowsthl, colsthl, index, lay1posn, lay2posn;
int itnum, blocki, blockj, spot;
double rfarray[64], absum, avbright, esum, energy, lay2wgt;
double sum, fx, fy, a, b, linsum, quadsum, y, d, error;
double lay1out[4][434], lay2out[4][270], pharray[25];
double gabor[4][64], theta[4], xquad[270][10];
float accuracy;


/* Open and read in data file to arrays flag[], class[], x[][] */

FILE *fp;
fp=fopen("bigimage","r");
if(fp==NULL) exit(0);
```

```
flag = (int *) malloc(numvec * sizeof(int));
class = (int *) malloc(numvec * sizeof(int));
x = (int *) malloc(numvec * numfeat * sizeof(int));

for(i=0; i<numvec; i++){
fscanf(fp,"%d",&flag[i]);
  for(j=0; j<numfeat; j++){
  fscanf(fp,"%d",&x[i*numfeat+j]);
  }
fscanf(fp,"%d",&class[i]);
}
fclose(fp);

/* Test printout module: */
/* for(i=0; i<numvec; i++){
printf("Flag=%d  Class=%d\n",flag[i],class[i]);
} */

/* Establish constant "weight" values for Gabor function orientations
   and "phase synchronizing" summation layer */

lay2wgt=1.0/25.0;  /* Constant weight for all 2nd hidden layer nodes */

/* Initialize four Gabor function weight arrays for first layer
 (maximum value of each function is 0.5 to aid sigmoid computation) */

fx=0.0;
fy=1.0;
gabcount=0;
for(i=0; i<=7; i++){
  for(j=0; j<=7; j++){
  a= 0.2*(float)i - 0.7;
  b= 0.2*(float)j - 0.7;
  gabor[0][gabcount]=
              0.5*exp(-pi*(a*a/3.0+b*b/3.0))*cos(2.0*pi*(fx*a+fy*b));
  gabcount=gabcount+1;
  }
}

fx=1.0;
fy=0.0;
gabcount=0;
for(i=0; i<=7; i++){
  for(j=0; j<=7; j++){
  a= 0.2*(float)i - 0.7;
  b= 0.2*(float)j - 0.7;
  gabor[1][gabcount]=
              0.5*exp(-pi*(a*a/3.0+b*b/3.0))*cos(2.0*pi*(fx*a+fy*b));
  gabcount=gabcount+1;
  }
}
```

```
fx=1.4142136;
fy=1.4142136;
gabcount=0;
for(i=0; i<=7; i++){
  for(j=0; j<=7; j++){
  a= 0.2*(float)i - 0.7;
  b= 0.2*(float)j - 0.7;
  gabor[2][gabcount]=
              0.5*exp(-pi*(a*a/3.0+b*b/3.0))*cos(2.0*pi*(fx*a+fy*b));
  gabcount=gabcount+1;
  }
}

fx=1.4142136;
fy=0.0-1.4142136;
gabcount=0;
for(i=0; i<=7; i++){
  for(j=0; j<=7; j++){
  a= 0.2*(float)i - 0.7;
  b= 0.2*(float)j - 0.7;
  gabor[3][gabcount]=
              0.5*exp(-pi*(a*a/3.0+b*b/3.0))*cos(2.0*pi*(fx*a+fy*b));
  gabcount=gabcount+1;
  }
}

/* End of module establishing weights.  Begin loop thru all vectors. */

for(bigloop=0; bigloop<numvec; bigloop++){

/* Begin propagation thru network */
/* Begin receptive field window shifting loop */
laylposn=0;
for(rowstpix=0; rowstpix<=6656; rowstpix=rowstpix+512){
  for(colstpix=rowstpix; colstpix<=rowstpix+120; colstpix=colstpix+4){
  index=0;
    for(i=colstpix; i<=colstpix+896; i=i+128){
      for(j=i; j<=(i+7); j++){
      rfarray[index]=(double)x[(bigloop*8064)+j];
      index=index+1;
      }
    }

/* LAMBERTIZATION AND CONTRAST NORMALIZATION MODULE (processes each
   receptive field window) */
absum=0.0;
for(i=0; i<64; i++){
absum=absum+rfarray[i];
}

avbright=absum/64.0;
for(i=0; i<64; i++){
rfarray[i]=rfarray[i]-avbright;
}
```

```
esum=0.0;
for(i=0; i<64; i++){
esum=esum+(rfarray[i]*rfarray[i]);
}

energy=sqrt(esum);
if (energy != 0.0)  {        /* Do not normalize if energy=0.0 */
  for(i=0; i<64; i++){
  rfarray[i]=rfarray[i]/energy;
  }
}  /* end normalization module */

/* Propagate thru to first hidden layer outputs */

for(i=0; i<4; i++){  /* Load each 1st layer block from single window */
sum=0.0;
  for(j=0; j<64; j++){
  sum = sum + gabor[i][j]*rfarray[j];
  } /* Finished calculating one block, one position (first layer) */

if(sum<-20.0) {laylout[i][laylposn]=0.0;}
else if(sum>20.0) {laylout[i][laylposn]=1.0;}
else {laylout[i][laylposn]=(double)(1.0/(1.0+exp(-sum)));}
} /* Finished calculating four blocks, one position (first layer) */

laylposn=laylposn+1;

  }
}  /* end of receptive field window shifting loops (all positions) */

/* Propagate thru second layer */
for(bshift=0; bshift<4; bshift++){
lay2posn=0;
for(rowsth1=0; rowsth1<=279; rowsth1=rowsth1+31){
  for(colsth1=rowsth1; colsth1<=rowsth1+26; colsth1++){
  index=0;
    for(i=colsth1; i<=colsth1+124; i=i+31){
      for(j=i; j<=(i+4); j++){
      pharray[index]=laylout[bshift][j];
      index=index+1;
      }
    }
    sum=0.0;
    for(i=0; i<25; i++){
    sum=sum+pharray[i];
    }
    lay2out[bshift][lay2posn] = sum*lay2wgt;

    lay2posn = lay2posn + 1;
  }
}
}
/* End propagation of one vector thru output of summation layer */
```

```
/* For each run thru bigloop (over each new vector), fill structure
   array vector[88].x2[4][270] with propagated results from outputs
   of summation layer.  The filled x2 array will be the input for
   the final, high-order layer. */

for(i=0; i<4; i++){
  for(j=0; j<270; j++){
  vector[bigloop].x2[i][j] = lay2out[i][j];
  }
}
} /* end of bigloop */

/* Begin high-order modules */
/* Initialization */
srandom((unsigned)time(NULL));   /* Init random() seed off clock */

for(n=0; n<4; n++){
  for(block=0; block<4; block++){
    for(j=0; j<270; j++){
    neuron[n].linwgt[block][j] = (double)(random() % 101)/100.00 - 0.5;
    neuron[n].linwgt[block][j] = neuron[n].linwgt[block][j]/147420.00;
    }
  }
}

for(n=0; n<4; n++){
  for(spot=0; spot<270; spot++){
    for(j=0; j<10; j++){
    neuron[n].quadwgt[spot][j]=(double)(random() % 101)/100.00 - 0.5;
    neuron[n].quadwgt[spot][j]=neuron[n].quadwgt[spot][j]/147420.00;
    }
  }
}

for(n=0; n<4; n++){
theta[n] = (double)(random() % 101)/100.00 - 0.5;
theta[n] = theta[n]/147420.00;
}
/* End of initialization; Begin training and testing loops */

for(fulloop=1; fulloop<=numruns; fulloop++){
for(trnloop=0; trnloop<numtrn; trnloop++){
randnum=(random() % numexvec);  /* Select random exemplar vector */

/* Multiply out second-order combinations for present exemplar */
for(spot=0; spot<270; spot++){
qcount=0;
  for(blocki=0; blocki<4; blocki++){
    for(blockj=blocki; blockj<4; blockj++){
    xquad[spot][qcount] =
   vector[randnum].x2[blocki][spot] * vector[randnum].x2[blockj][spot];
    qcount=qcount+1;
    }
  }
}
```

```
for(n=0; n<4; n++){     /* Loop to change neuron */
linsum=0.0;
  for(block=0; block<4; block++){
    for(j=0; j<270; j++){
    linsum = linsum +
             neuron[n].linwgt[block][j]*vector[randnum].x2[block][j];
    }
  } /* end block */

quadsum=0.0;
  for(spot=0; spot<270; spot++){
    for(j=0; j<10; j++){
    quadsum = quadsum + neuron[n].quadwgt[spot][j]*xquad[spot][j];
    }
  }

sum = linsum + quadsum;

if ((sum+theta[n])<-20.0) {y=0.0;}
else if ((sum+theta[n])>20.0) {y=1.0;}
else {y=(double)(1.0/(1.0+exp(-(sum+theta[n]))));}

/* Criteria for training each class to fire a specific neuron */
if (class[randnum]==n) {d=1.0;}
else {d=0.0;}

error=y*(1.0-y)*(d-y);

/* Update weights */
for(block=0; block<4; block++){
  for(j=0; j<270; j++){
  neuron[n].linwgt[block][j] = neuron[n].linwgt[block][j] +
                  (eta/147420.00)*error*vector[randnum].x2[block][j];
  }
}
for(spot=0; spot<270; spot++){
  for(j=0; j<10; j++){
  neuron[n].quadwgt[spot][j] = neuron[n].quadwgt[spot][j] +
                         (eta/147420.00)*error*xquad[spot][j];
  }
}
theta[n] = theta[n] + (eta/147420.00)*error;

} /* End of neuron (n) changing loop */
} /* End of trnloop */

/* Begin test loop to run thru each test vector */
numright=0;
for(testvec=numexvec; testvec<numvec; testvec++){
neurtcnt=0;
/* Multiply out second-order combinations for present test vector */
for(spot=0; spot<270; spot++){
qcount=0;
  for(blocki=0; blocki<4; blocki++){
    for(blockj=blocki; blockj<4; blockj++){
```

```
    xquad[spot][qcount] =
   vector[testvec].x2[blocki][spot] * vector[testvec].x2[blockj][spot];
    qcount=qcount+1;
    }
  }
}

for(n=0; n<4; n++){     /* Loop to change neuron */
linsum=0.0;
  for(block=0; block<4; block++){
    for(j=0; j<270; j++){
    linsum = linsum +
             neuron[n].linwgt[block][j]*vector[testvec].x2[block][j];
    }
  } /* end block */

quadsum=0.0;
  for(spot=0; spot<270; spot++){
    for(j=0; j<10; j++){
    quadsum = quadsum + neuron[n].quadwgt[spot][j]*xquad[spot][j];
    }
  }

sum = linsum + quadsum;

if ((sum+theta[n])<-20.0) {y=0.0;}
else if ((sum+theta[n])>20.0) {y=1.0;}
else {y=(double)(1.0/(1.0+exp(-(sum+theta[n]))));}

/* Test decision criteria for identifying classes */
if ((y>0.8) && (class[testvec]==n)) {neurtcnt=neurtcnt+1;}
if ((y<0.2) && (class[testvec]!=n)) {neurtcnt=neurtcnt+1;}

} /* End of neuron (n) changing loop */
if (neurtcnt==4) {numright=numright+1;}
} /* End of test loop running thru each test vector (testvec) */

itnum=fulloop*numtrn;
accuracy=(float)numright/(float)numttvec;
printf("Training Epoch #: %d   Accuracy=%f\n", itnum, accuracy);

} /* End of fulloop */
}  /* end of main */
```

```
/* Image classification network using multilayer perceptron for final
   output processing; variable number of hidden layer nodes.
   Name: IMAGNET5.C */

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

#define numexvec 44  /* Specify number of exemplar vectors */
#define numttvec 44  /* Specify number of test vectors */
#define numfeat 8064 /* Specify number of features per vector */
#define midnodes 30  /* Specify number of hidden layer nodes */

#define numruns 50   /* Specify number of result plot points */
#define numtrn 1000  /* Specify # training epochs between test/plot */

#define eta 1.0      /* Specify learning rate */
#define pi 3.14159265358979

char *malloc();

main(){

int *flag, *x, *class, neurtcnt, itnum, gabcount, loadcnt;
int i, j, k, n, bigloop, trnloop, testvec, numright, randnum, bshift;
int rowstpix, colstpix, rowsthl, colsthl, index, laylposn, lay2posn;
double rfarray[64], absum, avbright, esum, energy, lay2wgt;
double outtheta[4], midtheta[midnodes], sum, fx, fy, a, b, errsum;
double laylout[4][434], lay2out[4][270], pharray[25];
double gabor[4][64], input[1080], outwgts[4][midnodes];
double midwgts[midnodes][1080];
double d[4], errorout[4], errormid[midnodes], yout[4], ymid[midnodes];
float accuracy;

/* Open and read in data file to arrays flag[], class[], x[][] */

FILE *fp;
fp=fopen("bigimage","r");
if(fp==NULL) exit(0);

flag = (int *) malloc((numexvec+numttvec) * sizeof(int));
class = (int *) malloc((numexvec+numttvec) * sizeof(int));
x = (int *) malloc((numexvec+numttvec)*numfeat * sizeof(int));

for(i=0; i<(numexvec+numttvec); i++){
fscanf(fp,"%d",&flag[i]);
  for(j=0; j<numfeat; j++){
  fscanf(fp,"%d",&x[i*numfeat+j]);
  }
fscanf(fp,"%d",&class[i]);
}
fclose(fp);
/* Test printout module: */
/* for(i=0; i<(numexvec+numttvec); i++){
printf("Flag=%d\n",flag[i]);
} */
```

```
srandom((unsigned)time(NULL)); /* Init random() seed off clock */

/* Initialize output theta and weight arrays */

for(i=0; i<4; i++){
outtheta[i] = (double)(random() % 101)/100.00 - 0.5;
outtheta[i] = outtheta[i]/(double)midnodes;
}

for(i=0; i<midnodes; i++){
midtheta[i] = (double)(random() % 101)/100.00 - 0.5;
midtheta[i] = midtheta[i]/1080.00;
}

for(i=0; i<4; i++){
  for(j=0; j<midnodes; j++){
  outwgts[i][j] = (double)(random() % 101)/100.00 - 0.5;
  outwgts[i][j] = outwgts[i][j]/(double)midnodes;
  }
}

for(i=0; i<midnodes; i++){
  for(j=0; j<1080; j++){
  midwgts[i][j] = (double)(random() % 101)/100.00 - 0.5;
  midwgts[i][j] = midwgts[i][j]/1080.00;
  }
}


lay2wgt=1.0/25.0;  /* Constant weight for all 2nd hidden layer nodes */

/* Initialize four Gabor function weight arrays for first layer
 (maximum value of each function is 0.5 to aid sigmoid computation) */

fx=0.0;
fy=1.0;
gabcount=0;
for(i=0; i<=7; i++){
  for(j=0; j<=7; j++){
  a= 0.2*(float)i - 0.7;
  b= 0.2*(float)j - 0.7;
  gabor[0][gabcount]=
              0.5*exp(-pi*(a*a/3.0+b*b/3.0))*cos(2.0*pi*(fx*a+fy*b));
  gabcount=gabcount+1;
  }
}

fx=1.0;
fy=0.0;
gabcount=0;
for(i=0; i<=7; i++){
  for(j=0; j<=7; j++){
  a= 0.2*(float)i - 0.7;
  b= 0.2*(float)j - 0.7;
  gabor[1][gabcount]=
```

```
                0.5*exp(-pi*(a*a/3.0+b*b/3.0))*cos(2.0*pi*(fx*a+fy*b));
  gabcount=gabcount+1;
  }
}

fx=1.4142136;
fy=1.4142136;
gabcount=0;
for(i=0; i<=7; i++){
  for(j=0; j<=7; j++){
  a= 0.2*(float)i - 0.7;
  b= 0.2*(float)j - 0.7;
  gabor[2][gabcount]=
                0.5*exp(-pi*(a*a/3.0+b*b/3.0))*cos(2.0*pi*(fx*a+fy*b));
  gabcount=gabcount+1;
  }
}

fx=1.4142136;
fy=0.0-1.4142136;
gabcount=0;
for(i=0; i<=7; i++){
  for(j=0; j<=7; j++){
  a= 0.2*(float)i - 0.7;
  b= 0.2*(float)j - 0.7;
  gabor[3][gabcount]=
                0.5*exp(-pi*(a*a/3.0+b*b/3.0))*cos(2.0*pi*(fx*a+fy*b));
  gabcount=gabcount+1;
  }
}

/* End of initialization; begin loops for training and testing. */
for(bigloop=1; bigloop<=numruns; bigloop++){

/* Begin training loop */
for(trnloop=0; trnloop<numtrn; trnloop++){
randnum=(random() % numexvec);

/* Begin propagation thru network */
/* Begin receptive field window shifting loop */
laylposn=0;
for(rowstpix=0; rowstpix<=6656; rowstpix=rowstpix+512){
  for(colstpix=rowstpix; colstpix<=rowstpix+120; colstpix=colstpix+4){
  index=0;
    for(i=colstpix; i<=colstpix+896; i=i+128){
      for(j=i; j<=(i+7); j++){
      rfarray[index]=(double)x[(randnum*8064)+j];
      index=index+1;
      }
    }
```

```
/* LAMBERTIZATION AND CONTRAST NORMALIZATION MODULE (processes each
   receptive field window) */
absum=0.0;
for(i=0; i<64; i++){
absum=absum+rfarray[i];
}

avbright=absum/64.0;
for(i=0; i<64; i++){
rfarray[i]=rfarray[i]-avbright;
}

esum=0.0;
for(i=0; i<64; i++){
esum=esum+(rfarray[i]*rfarray[i]);
}

energy=sqrt(esum);
if (energy != 0.0)  {        /* Do not normalize if energy=0.0 */
  for(i=0; i<64; i++){
  rfarray[i]=rfarray[i]/energy;
  }
}  /* end normalization module */

/* Propagate thru to first hidden layer outputs */

for(i=0; i<4; i++){  /* Load each 1st layer block from single window */
sum=0.0;
  for(j=0; j<64; j++){
  sum = sum + gabor[i][j]*rfarray[j];
  } /* Finished calculating one block, one position (first layer) */

if(sum<-20.0) {laylout[i][laylposn]=0.0;}
else if(sum>20.0) {laylout[i][laylposn]=1.0;}
else {laylout[i][laylposn]=(double)(1.0/(1.0+exp(-sum)));}
} /* Finished calculating four blocks, one position (first layer) */

laylposn=laylposn+1;

  }
}  /* end of receptive field window shifting loops (all positions) */

/* Propagate thru second layer */
for(bshift=0; bshift<4; bshift++){
lay2posn=0;
for(rowsth1=0; rowsth1<=279; rowsth1=rowsth1+31){
  for(colsth1=rowsth1; colsth1<=rowsth1+26; colsth1++){
  index=0;
    for(i=colsth1; i<=colsth1+124; i=i+31){
      for(j=i; j<=(i+4); j++){
      pharray[index]=laylout[bshift][j];
      index=index+1;
      }
    }
    sum=0.0;
```

```
    for(i=0; i<25; i++){
    sum=sum+pharray[i];
    }
    lay2out[bshift][lay2posn] = sum*lay2wgt;

    lay2posn = lay2posn + 1;
  }
}
}

/* Load input array for multilayer perceptron */
loadcnt=0;
for(i=0; i<4; i++){
  for(j=0; j<270; j++){
  input[loadcnt] = lay2out[i][j];
  loadcnt=loadcnt+1;
  }
}
/* Test printout module for input array */
/*printf("flag=%d\n",flag[randnum]);
for(i=0; i<1080; i++){
printf("input=%f\n",input[i]);
}
printf("class=%d\n",class[randnum]);*/

/* Propagate thru multilayer perceptron */

for(n=0; n<midnodes; n++){
sum=0.0;
  for(i=0; i<1080; i++){
  sum = sum + midwgts[n][i]*input[i];
  }
if ((sum+midtheta[n])<-20.0) {ymid[n]=0.0;}
else if ((sum+midtheta[n])>20.0) {ymid[n]=1.0;}
else {ymid[n]=(double)(1.0/(1.0+exp(-(sum+midtheta[n]))));}
}  /* End (n) loop changing hidden layer nodes */

for(n=0; n<4; n++){
sum=0.0;
  for(i=0; i<midnodes; i++){
  sum = sum + outwgts[n][i]*ymid[i];
  }
if ((sum+outtheta[n])<-20.0) {yout[n]=0.0;}
else if ((sum+outtheta[n])>20.0) {yout[n]=1.0;}
else {yout[n]=(double)(1.0/(1.0+exp(-(sum+outtheta[n]))));}
}  /* End (n) loop changing output layer nodes */

/* End propagation thru network */

/* Backpropagation to update perceptron weights */

for(n=0; n<4; n++){
if (class[randnum] == n) {d[n]=1.0;}
else {d[n]=0.0;}
}
```

```
for(n=0; n<4; n++){
errorout[n] = yout[n]*(1.0-yout[n])*(d[n]-yout[n]);
}

for(n=0; n<midnodes; n++){
errsum=0.0;
  for(i=0; i<4; i++){
  errsum = errsum + errorout[i]*outwgts[i][n];
  }
errormid[n] = ymid[n]*(1.0-ymid[n])*errsum;
}

for(i=0; i<4; i++){
  for(j=0; j<midnodes; j++){
  outwgts[i][j] = outwgts[i][j] +
                  (eta/(double)midnodes)*errorout[i]*ymid[j];
  }
}

for(i=0; i<4; i++){
outtheta[i] = outtheta[i] + (eta/(double)midnodes)*errorout[i];
}

for(i=0; i<midnodes; i++){
  for(j=0; j<1080; j++){
  midwgts[i][j] = midwgts[i][j] + (eta/1080.00)*errormid[i]*input[j];
  }
}

for(i=0; i<midnodes; i++){
midtheta[i] = midtheta[i] + (eta/1080.00)*errormid[i];
}

/* end of backprop */

}  /* end of training loop (trnloop) */

/* Begin test loop to run thru each test vector */
numright=0;
for(testvec=numexvec; testvec<(numexvec+numttvec); testvec++){

/* Begin propagation thru network */
/* Begin receptive field window shifting loop */
laylposn=0;
for(rowstpix=0; rowstpix<=6656; rowstpix=rowstpix+512){
  for(colstpix=rowstpix; colstpix<=rowstpix+120; colstpix=colstpix+4){
  index=0;
    for(i=colstpix; i<=colstpix+896; i=i+128){
      for(j=i; j<=(i+7); j++){
      rfarray[index]=(double)x[(testvec*8064)+j];
      index=index+1;
      }
    }
```

```
/* LAMBERTIZATION AND CONTRAST NORMALIZATION MODULE (processes each
   receptive field window) */
absum=0.0;
for(i=0; i<64; i++){
absum=absum+rfarray[i];
}

avbright=absum/64.0;
for(i=0; i<64; i++){
rfarray[i]=rfarray[i]-avbright;
}

esum=0.0;
for(i=0; i<64; i++){
esum=esum+(rfarray[i]*rfarray[i]);
}

energy=sqrt(esum);
if (energy != 0.0) {              /* Do not normalize if energy=0.0 */
  for(i=0; i<64; i++){
  rfarray[i]=rfarray[i]/energy;
  }
}  /* end normalization module */

/* Propagate thru to first hidden layer outputs */

for(i=0; i<4; i++){  /* Load each 1st layer block from single window */
sum=0.0;
  for(j=0; j<64; j++){
  sum = sum + gabor[i][j]*rfarray[j];
  } /* Finished calculating one block, one position (first layer) */

if(sum<-20.0) {laylout[i][laylposn]=0.0;}
else if(sum>20.0) {laylout[i][laylposn]=1.0;}
else {laylout[i][laylposn]=(double)(1.0/(1.0+exp(-sum)));}
} /* Finished calculating four blocks, one position (first layer) */

laylposn=laylposn+1;

  }
}  /* end of receptive field window shifting loops (all positions) */

/* Propagate thru second layer */
for(bshift=0; bshift<4; bshift++){
lay2posn=0;
for(rowsth1=0; rowsth1<=279; rowsth1=rowsth1+31){
  for(colsth1=rowsth1; colsth1<=rowsth1+26; colsth1++){
  index=0;
    for(i=colsth1; i<=colsth1+124; i=i+31){
      for(j=i; j<=(i+4); j++){
      pharray[index]=laylout[bshift][j];
      index=index+1;
      }
    }
    sum=0.0;
```

```
    for(i=0; i<25; i++){
    sum=sum+pharray[i];
    }
    lay2out[bshift][lay2posn] = sum*lay2wgt;

    lay2posn = lay2posn + 1;
  }
}
}

/* Load input array for multilayer perceptron */
loadcnt=0;
for(i=0; i<4; i++){
  for(j=0; j<270; j++){
  input[loadcnt] = lay2out[i][j];
  loadcnt=loadcnt+1;
  }
}


/* Propagate thru multilayer perceptron */

for(n=0; n<midnodes; n++){
sum=0.0;
  for(i=0; i<1080; i++){
  sum = sum + midwgts[n][i]*input[i];
  }
if ((sum+midtheta[n])<-20.0) {ymid[n]=0.0;}
else if ((sum+midtheta[n])>20.0) {ymid[n]=1.0;}
else {ymid[n]=(double)(1.0/(1.0+exp(-(sum+midtheta[n]))));}
}  /* End (n) loop changing hidden layer nodes */

for(n=0; n<4; n++){
sum=0.0;
  for(i=0; i<midnodes; i++){
  sum = sum + outwgts[n][i]*ymid[i];
  }
if ((sum+outtheta[n])<-20.0) {yout[n]=0.0;}
else if ((sum+outtheta[n])>20.0) {yout[n]=1.0;}
else {yout[n]=(double)(1.0/(1.0+exp(-(sum+outtheta[n]))));}
}  /* End (n) loop changing output layer nodes */

/* End propagation thru network */

/* Decision criteria for correct classification */
neurtcnt=0;
for(n=0; n<4; n++){
if((yout[n]>0.8) && (class[testvec]==n)) {neurtcnt=neurtcnt+1;}
if((yout[n]<0.2) && (class[testvec]!=n)) {neurtcnt=neurtcnt+1;}
} /* End loop for checking all four output neurons */
if(neurtcnt==4) {numright=numright+1;}

}  /* end of testing loop (testvec) */
```

```
itnum = bigloop*numtrn;
accuracy = (float)numright/(float)numttvec;
printf("Training Epoch #: %d   Accuracy=%f\n", itnum, accuracy);

}  /* end of bigloop */
}  /* end of main */
```

```
/* This preprocessing program propagates all image inputs from
   the input data set through two hard-wired layers of the
   biologically-motivated image classification network.  The
   program transforms input pixel data vectors to extracted
   feature vectors.  The feature vectors, along with flag
   and class designating integers, are written to an output
   file which can be accessed later by a separate
   classification routine.  Name: PREPROC.C */

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

#define numvec 88       /* Specify total number of vectors */
#define numfeat 8064    /* Specify number of features per vector */

#define pi 3.14159265358979

char *malloc();

main(){

int *flag, *x, *class, gabcount;
int i, j, bigloop, bshift;
int rowstpix, colstpix, rowsthl, colsthl, index, laylposn, lay2posn;
double rfarray[64], absum, avbright, asum, energy, lay2wgt;
double sum, fx, fy, a, b;
double laylout[4][434], lay2out[4][270], pharray[25];
double gabor[4][64];

/* Open and read in data file to arrays flag[], class[], x[][] */

FILE *fp;
fp=fopen("bigimage","r");
if(fp==NULL) exit(0);

flag = (int *) malloc(numvec * sizeof(int));
class = (int *) malloc(numvec * sizeof(int));
x = (int *) malloc(numvec * numfeat * sizeof(int));

for(i=0; i<numvec; i++){
fscanf(fp,"%d",&flag[i]);
  for(j=0; j<numfeat; j++){
  fscanf(fp,"%d",&x[i*numfeat+j]);
  }
fscanf(fp,"%d",&class[i]);
}
fclose(fp);

/* Test printout module: */
/*for(i=0; i<numvec; i++){
printf("Flag=%d  Class=%d\n",flag[i],class[i]);
} */
```

```
/* Establish constant "weight" values for Gabor function orientations
   and "phase synchronizing" summation layer */

lay2wgt=1.0/25.0;  /* Constant weight for all 2nd hidden layer nodes */

/* Initialize four Gabor function weight arrays for first layer
 (maximum value of each function is 0.5 to aid sigmoid computation) */

fx=0.0;
fy=1.0;
gabcount=0;
for(i=0; i<=7; i++){
  for(j=0; j<=7; j++){
  a= 0.2*(float)i - 0.7;
  b= 0.2*(float)j - 0.7;
  gabor[0][gabcount]=
              0.5*exp(-pi*(a*a/3.0+b*b/3.0))*cos(2.0*pi*(fx*a+fy*b));
  gabcount=gabcount+1;
  }
}

fx=1.0;
fy=0.0;
gabcount=0;
for(i=0; i<=7; i++){
  for(j=0; j<=7; j++){
  a= 0.2*(float)i - 0.7;
  b= 0.2*(float)j - 0.7;
  gabor[1][gabcount]=
              0.5*exp(-pi*(a*a/3.0+b*b/3.0))*cos(2.0*pi*(fx*a+fy*b));
  gabcount=gabcount+1;
  }
}

fx=1.4142136;
fy=1.4142136;
gabcount=0;
for(i=0; i<=7; i++){
  for(j=0; j<=7; j++){
  a= 0.2*(float)i - 0.7;
  b= 0.2*(float)j - 0.7;
  gabor[2][gabcount]=
              0.5*exp(-pi*(a*a/3.0+b*b/3.0))*cos(2.0*pi*(fx*a+fy*b));
  gabcount=gabcount+1;
  }
}

fx=1.4142136;
fy=0.0-1.4142136;
gabcount=0;
for(i=0; i<=7; i++){
  for(j=0; j<=7; j++){
  a= 0.2*(float)i - 0.7;
  b= 0.2*(float)j - 0.7;
  gabor[3][gabcount]=
```

```
                0.5*exp(-pi*(a*a/3.0+b*b/3.0))*cos(2.0*pi*(fx*a+fy*b));
   gabcount=gabcount+1;
   }
}

/* End of module establishing weights.  Begin loop thru all vectors. */

for(bigloop=0; bigloop<numvec; bigloop++){

/* Begin propagation thru network */
/* Begin receptive field window shifting loop */
laylposn=0;
for(rowstpix=0; rowstpix<=6656; rowstpix=rowstpix+512){
  for(colstpix=rowstpix; colstpix<=rowstpix+120; colstpix=colstpix+4){
  index=0;
    for(i=colstpix; i<=colstpix+896; i=i+128){
      for(j=i; j<=(i+7); j++){
      rfarray[index]=(double)x[(bigloop*8064)+j];
      index=index+1;
      }
    }

/* LAMBERTIZATION AND CONTRAST NORMALIZATION MODULE (processes each
   receptive field window) */
absum=0.0;
for(i=0; i<64; i++){
absum=absum+rfarray[i];
}

avbright=absum/64.0;
for(i=0; i<64; i++){
rfarray[i]=rfarray[i]-avbright;
}

esum=0.0;
for(i=0; i<64; i++){
esum=esum+(rfarray[i]*rfarray[i]);
}

energy=sqrt(esum);
if (energy != 0.0)  {        /* Do not normalize if energy=0.0 */
  for(i=0; i<64; i++){
  rfarray[i]=rfarray[i]/energy;
  }
}  /* end normalization module */

/* Propagate thru to first hidden layer outputs */

for(i=0; i<4; i++){  /* Load each 1st layer block from single window */
sum=0.0;
  for(j=0; j<64; j++){
  sum = sum + gabor[i][j]*rfarray[j];
  } /* Finished calculating one block, one position (first layer) */
```

```
if(sum<-20.0) (laylout[i][laylposn]=0.0;)
else if(sum>20.0) (laylout[i][laylposn]=1.0;)
else (laylout[i][laylposn]=(double)(1.0/(1.0+exp(-sum)));)
) /* Finished calculating four blocks, one position (first layer) */

laylposn=laylposn+1;

  }
}  /* end of receptive field window shifting loops (all positions) */

/* Propagate thru second layer */
for(bshift=0; bshift<4; bshift++){
lay2posn=0;
for(rowsthl=0; rowsthl<=279; rowsthl=rowsthl+31){
  for(colsthl=rowsthl; colsthl<=rowsthl+26; colsthl++){
  index=0;
    for(i=colsthl; i<=colsthl+124; i=i+31){
      for(j=i; j<=(i+4); j++){
      pharray[index]=laylout[bshift][j];
      index=index+1;
      }
    }
    sum=0.0;
    for(i=0; i<25; i++){
    sum=sum+pharray[i];
    }
    lay2out[bshift][lay2posn] = sum*lay2wgt;

    lay2posn = lay2posn + 1;
  }
}
}
/* End propagation thru output of summation layer */

/* Print to output file: flag, new component values,
   and class (for each vector) */

printf("%d\n",flag[bigloop]);
for(i=0; i<4; i++){
  for(j=0; j<270; j++){
  printf("%1.16f\n",lay2out[i][j]);
  }
}
printf("%d\n",class[bigloop]);

} /* end of bigloop */
}  /* end of main */
```

```
/*  This program inputs a data set consisting of previously
    extracted feature vectors such as those transformed by
    the biologically-motivated image processing layers in
    PREPROC.C.  This routine vertically normalizes the
    input feature vectors and feeds them into a multilayer
    perceptron classifier with a variable number of hidden-
    layer nodes.  Input and interim values are
    double precision.
    Name:  MLPNORM.C */

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

#define numexvec 44    /* Specify # of exemplar vectors */
#define numttvec 44    /* Specify # of test vectors */
#define numfeat 1080   /* Specify # of features per vector */
#define midnodes 30    /* Specify # of hidden layer nodes */

#define numruns 50     /* Specify # of result plot points */
#define numtrn 1000    /* Specify # training epochs between test/plot */

#define eta 1.0        /* Specify learning rate */


main(){
int *flag, *class, i, j, n, neurtcnt, numright, itnum, randnum;
int bigloop, trnloop, testvec, row, col;
double *x, outtheta[4], midtheta[midnodes], sum, errsum;
double outwgts[4][midnodes], midwgts[midnodes][numfeat];
double d[4], errorout[4], errormid[midnodes], yout[4], ymid[midnodes];
double mean, sumsqdif, s;
float accuracy;



/* Open and load in transformed data file of extracted features */

FILE *fp;
fp=fopen("preproc.out","r");
if (fp==NULL) exit(0);

flag = (int *) malloc((numexvec+numttvec)*sizeof(int));
class = (int *) malloc((numexvec+numttvec)*sizeof(int));
x = (double *) malloc((numexvec+numttvec)*numfeat*sizeof(double));

for(i=0; i<(numexvec+numttvec); i++){
fscanf(fp,"%d",&flag[i]);
  for(j=0; j<numfeat; j++){
  fscanf(fp,"%lf",&x[i*numfeat+j]);
  }
fscanf(fp,"%d",&class[i]);
}
fclose(fp);
```

```
/* Test printout module */
/* for(i=0; i<(numexvec+numttvec); i++){
printf("%d\n",flag[i]);
  for(j=0; j<numfeat; j++){
  printf("%1.16f\n",x[i*numfeat+j]);
  }
printf("%d\n",class[i]);
} */

/* Vertically normalize x[numexvec+numttvec][numfeat] */
for(col=0; col<numfeat; col++){
sum=0.0;
  for(row=0; row<numexvec; row++){
  sum=sum+x[row*numfeat+col];
  }
mean=sum/(double)(numexvec);

sumsqdif=0.0;
  for(row=0; row<numexvec; row++){
  sumsqdif=sumsqdif+(x[row*numfeat+col]-mean)*(x[row*numfeat+col]-mean);
  }
s=(double)(sqrt(sumsqdif/(double)(numexvec)));
  for(row=0; row<(numexvec+numttvec); row++){
  x[row*numfeat+col]=(x[row*numfeat+col]-mean)/s;
  }
}  /* End column incrementing loop */
/* Test printout loop */
/* for(row=0; row<(numexvec+numttvec); row++){
  for(col=0; col<numfeat; col++){
  printf("For row=%d  col=%d  normxvalue=%f\n",row,col,
  x[row*numfeat+col]);
  }
} */


srandom((unsigned)time(NULL)); /* Init random() seed off clock */

/* Initialize output theta and weight arrays */

for(i=0; i<4; i++){
outtheta[i] = (double)(random() % 101)/100.00 - 0.5;
outtheta[i] = outtheta[i]/(double)midnodes;
}

for(i=0; i<midnodes; i++){
midtheta[i] = (double)(random() % 101)/100.00 - 0.5;
midtheta[i] = midtheta[i]/(double)numfeat;
}

for(i=0; i<4; i++){
  for(j=0; j<midnodes; j++){
  outwgts[i][j] = (double)(random() % 101)/100.00 - 0.5;
  outwgts[i][j] = outwgts[i][j]/(double)midnodes;
  }
}
```

```
for(i=0; i<midnodes; i++){
  for(j=0; j<numfeat; j++){
  midwgts[i][j] = (double)(random() % 101)/100.00 - 0.5;
  midwgts[i][j] = midwgts[i][j]/(double)numfeat;
  }
}

/* End of initialization; begin loops for training and testing. */
for(bigloop=1; bigloop<=numruns; bigloop++){

/* Begin training loop */
for(trnloop=0; trnloop<numtrn; trnloop++){
randnum=(random() % numexvec);

/* Propagate thru multilayer perceptron */
for(n=0; n<midnodes; n++){
sum=0.0;
  for(i=0; i<numfeat; i++){
  sum = sum + midwgts[n][i]*x[randnum*numfeat+i];
  }
if ((sum+midtheta[n])<-20.0) {ymid[n]=0.0;}
else if ((sum+midtheta[n])>20.0) {ymid[n]=1.0;}
else {ymid[n]=(double)(1.0/(1.0+exp(-(sum+midtheta[n]))));}
}  /* End (n) loop changing hidden layer nodes */

for(n=0; n<4; n++){
sum=0.0;
  for(i=0; i<midnodes; i++){
  sum = sum + outwgts[n][i]*ymid[i];
  }
if ((sum+outtheta[n])<-20.0) {yout[n]=0.0;}
else if ((sum+outtheta[n])>20.0) {yout[n]=1.0;}
else {yout[n]=(double)(1.0/(1.0+exp(-(sum+outtheta[n]))));}
}  /* End (n) loop changing output layer nodes */

/* End propagation thru network */

/* Backpropagation to update perceptron weights */

for(n=0; n<4; n++){
if (class[randnum] == n) {d[n]=1.0;}
else {d[n]=0.0;}
}

for(n=0; n<4; n++){
errorout[n] = yout[n]*(1.0-yout[n])*(d[n]-yout[n]);
}

for(n=0; n<midnodes; n++){
errsum=0.0;
  for(i=0; i<4; i++){
  errsum = errsum + errorout[i]*outwgts[i][n];
  }
errormid[n] = ymid[n]*(1.0-ymid[n])*errsum;
}
```

```
for(i=0; i<4; i++){
  for(j=0; j<midnodes; j++){
  outwgts[i][j] = outwgts[i][j] +
                  (eta/(double)midnodes)*errorout[i]*ymid[j];
  }
}

for(i=0; i<4; i++){
outtheta[i] = outtheta[i] + (eta/(double)midnodes)*errorout[i];
}

for(i=0; i<midnodes; i++){
  for(j=0; j<numfeat; j++){
  midwgts[i][j] = midwgts[i][j] +
                (eta/(double)numfeat)*errormid[i]*x[randnum*numfeat+j];
  }
}

for(i=0; i<midnodes; i++){
midtheta[i] = midtheta[i] + (eta/(double)numfeat)*errormid[i];
}

/* end of backprop */

}  /* end of training loop (trnloop) */

/* Begin test loop to run thru each test vector */
numright=0;
for(testvec=numexvec; testvec<(numexvec+numttvec); testvec++){

/* Propagate thru multilayer perceptron */

for(n=0; n<midnodes; n++){
sum=0.0;
  for(i=0; i<numfeat; i++){
  sum = sum + midwgts[n][i]*x[testvec*numfeat+i];
  }
if ((sum+midtheta[n])<-20.0) {ymid[n]=0.0;}
else if ((sum+midtheta[n])>20.0) {ymid[n]=1.0;}
else {ymid[n]=(double)(1.0/(1.0+exp(-(sum+midtheta[n]))));}
}  /* End (n) loop changing hidden layer nodes */

for(n=0; n<4; n++){
sum=0.0;
  for(i=0; i<midnodes; i++){
  sum = sum + outwgts[n][i]*ymid[i];
  }
if ((sum+outtheta[n])<-20.0) {yout[n]=0.0;}
else if ((sum+outtheta[n])>20.0) {yout[n]=1.0;}
else {yout[n]=(double)(1.0/(1.0+exp(-(sum+outtheta[n]))));}
}  /* End (n) loop changing output layer nodes */

/* End propagation thru network */
```

```
/* Decision criteria for correct classification */
neurtcnt=0;
for(n=0; n<4; n++){
if((yout[n]>0.8) && (class[testvec]==n)) (neurtcnt=neurtcnt+1;)
if((yout[n]<0.2) && (class[testvec]!=n)) (neurtcnt=neurtcnt+1;)
} /* End loop for checking all four output neurons */
if(neurtcnt==4) (numright=numright+1;)

}  /* end of testing loop (testvec) */

itnum = bigloop*numtrn;
accuracy = (float)numright/(float)numttvec;
printf("Training Epoch #: %d   Accuracy=%f\n", itnum, accuracy);

}  /* end of bigloop */

} /* end of main */
```

```
/* Image classification network version for processing input
   from PREPROC.C.  Classifier incorporates a single-layer,
   second-order neural network algorithm with automatic
   vertical normalization of inputs and multiplied combos.
   Multiplied combos are of same location node outputs
   among the four different orientation groups. */
/* Name: IMGNORM2.C */

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

#define numneur 4     /* Specify total number of neurons across slab */
                      /* Number of neurons = number of classes */
#define numexvec 44    /* Specify number of exemplar vectors */
#define numttvec 44    /* Specify number of test vectors */
#define numfeat 1080    /* Specify number of features per vector */
                      /* numfeat also represents # FEATURE COLUMNS */
#define numruns 500    /* Specify number of result plot points */
#define numtrn 100    /* Specify # training iters between test/plot */

#define eta 1.0       /* Specify training factor */

char *malloc();       /* Must define for use of malloc thru program */
                      /*Need for this statement is compiler-dependent*/
main(){

/* flag[] holds ID number of each data vector, class[] holds training
   class associated with each data vector, x[][] holds feature
   components of data vectors, quadin[][] holds second-order multiplied
  combos of x values, wlin[][] and wquad[][] hold weights for all inputs
   (including higher order combos), theta[] holds threshold values for
   each neuron */

int *flag, *class;
int i, j, n, randnum, numquad, qcount;
int row, col, spot;
int neurtcnt, numright, bigloop, trnloop, testvec, itnum;
double d, y, error, neweta;
double *x, *wquad, *quadin;
double theta[numneur], wlin[numneur*numfeat];
double sum, linsum, quadsum, mean, s, sumsqdif, fanin;
float accuracy;
/* This module opens an input file and reads data into the
   arrays flag[], class[], and x[][] */

FILE *fp;
fp=fopen("preproc.out","r");   /* Insert correct data file name here */
if(fp==NULL) exit(0);

/* Dynamic alloc of arrays prior to loading in data file values */

flag = (int *) malloc((numexvec+numttvec) * sizeof(int));
class = (int *) malloc((numexvec+numttvec) * sizeof(int));
x = (double *) malloc((numexvec+numttvec)*numfeat * sizeof(double));
```

```
for(i=0; i<(numexvec+numttvec); ++i){
fscanf(fp,"%d",&flag[i]);

  for(j=0; j<numfeat; ++j){
  fscanf(fp,"%lf",&x[i*numfeat+j]);
  }                              /* end inner loop */

fscanf(fp, "%d",&class[i]);
}                                /* end outer loop */

fclose(fp);

/* This test module prints out input data containing first two and
   final feature components of each vector; checks the above file
   opening module (comment out for final program) */

/* for(i=0; i<(numexvec+numttvec); ++i){
printf("For flag=%d, class=%f, check features are %f %f %f\n",
flag[i], class[i], x[i*numfeat+0], x[i*numfeat+1],
x[i*numfeat+(numfeat-1)]);
} */

/* This statement establishes the number of second-order
   combos of feature inputs FOR EACH VECTOR (without commutative
   redundancy), unique to this problem.  Will be used for array
   allocation. */

numquad=2700;

/* Fill 2-D arrays for second order input combos:
   quadin[numexvec+numttvec][numquad] */

quadin=(double *) malloc((numexvec+numttvec)*numquad*sizeof(double));
for(row=0; row<(numexvec+numttvec); row++){
qcount=0;

for(spot=0; spot<270; spot++){
  for(i=spot; i<=(spot+810); i=i+270){
    for(j=i; j<=(spot+810); j=j+270){
    quadin[row*numquad+qcount] = x[row*numfeat+i] * x[row*numfeat+j];
    qcount = qcount +1;
    }
  }
}

} /* end vector row incrementing loop */
/* Test printout module */
/* for(row=0; row<(numexvec+numttvec); row++){
  for(col=0; col<numquad; col++){
  printf("For row=%d  col=%d  quadin=%f\n",row,col,
  quadin[row*numquad+col]);
  }
} */
```

```
/* Begin VERTICAL NORMALIZATION of x, quadin arrays */

/* Vertically normalize x[numexvec+numttvec][numfeat] */
for(col=0; col<numfeat; col++){
sum=0.0;
  for(row=0; row<numexvec; row++){
  sum=sum+x[row*numfeat+col];
  }
mean=sum/(double)(numexvec);

sumsqdif=0.0;
  for(row=0; row<numexvec; row++){
  sumsqdif=sumsqdif+(x[row*numfeat+col]-mean)*(x[row*numfeat+col]-mean);
  }
s=(double)(sqrt(sumsqdif/(double)(numexvec)));
  for(row=0; row<(numexvec+numttvec); row++){
  x[row*numfeat+col]=(x[row*numfeat+col]-mean)/s;
  }
}  /* End column incrementing loop */
/* Test printout loop */
/* for(row=0; row<(numexvec+numttvec); row++){
  for(col=0; col<numfeat; col++){
  printf("For row=%d  col=%d  normxvalue=%f\n",row,col,
  x[row*numfeat+col]);
  }
} */

/* Vertically normalize quadin[numexvec+numttvec][numquad] */
for(col=0; col<numquad; col++){
sum=0.0;
  for(row=0; row<numexvec; row++){
  sum=sum+quadin[row*numquad+col];
  }
mean=sum/(double)(numexvec);

sumsqdif=0.0;
  for(row=0; row<numexvec; row++){
  sumsqdif=sumsqdif+
  (quadin[row*numquad+col]-mean)*(quadin[row*numquad+col]-mean);
  }
s=(double)(sqrt(sumsqdif/(double)(numexvec)));
  for(row=0; row<(numexvec+numttvec); row++){
  quadin[row*numquad+col]=(quadin[row*numquad+col]-mean)/s;
  }
}  /* End column incrementing loop */
/* Test printout loop */
/* for(row=0; row<(numexvec+numttvec); row++){
  for(col=0; col<numquad; col++){
  printf("For row=%d  col=%d  normquadvalue=%f\n",row,col,
  quadin[row*numquad+col]);
  }
} */
/* End of vertical normalization */
```

```
srandom((unsigned)time(NULL)); /* Init random() seed off clock */

/* Loop to initialize theta array (one for each neuron) with random
   floating point value between -0.5 and 0.5 */

for(i=0; i<numneur; i++){
theta[i]=(double)(random() % 101)/100.00 - 0.5;
/* printf("For i=%d  theta=%f\n",i,theta[i]); */
}

/* Loop to initialize linear input weights array wlin[numneur][numfeat]
   with random floating point values between -0.5 and 0.5 */

for(n=0; n<numneur; n++){
/* printf("For neuron # %d\n",n); */
  for(i=0; i<numfeat; i++){
  wlin[n*numfeat+i]=(double)(random() % 101)/100.00 - 0.5;
  /* printf("For i=%d  wlin=%f\n",i,wlin[n*numfeat+i]); */
  }
} /* end of nested loop */

fanin = (double)numfeat + (double)numquad;
/* fanin = 1; */
neweta = (double) eta / fanin;
/* printf("neweta= %f\n",neweta); */

/* Loop to initialize quadratic input weights array
   wquad[numneur][numquad] with random floating point values between
   -0.5 and 0.5 */

wquad = (double *) malloc(numneur*numquad*sizeof(double));
for(n=0; n<numneur; n++){
/* printf("For neuron # %d\n",n); */
  for(i=0; i<numquad; i++){
  wquad[n*numquad+i]=(double)(random() % 101)/100.00 - 0.5;
  /* printf("For i=%d quadweight=%f\n",i,wquad[n*numquad+i]); */
  }
}  /* end of nested loop */

/* End of initialization; Begin loops for training and testing */
for(bigloop=1; bigloop<=numruns; bigloop++){

/* Begin training loop */
for(trnloop=0; trnloop<numtrn; trnloop++){

/* Randomly select an exemplar vector row (flag #) */
randnum=(random() % numexvec);
/* printf("Randomly selected vector flag number is %d\n",randnum); */

for(n=0; n<numneur; n++){  /* Loop to update weights for each neuron */
linsum=0.0;
  for(i=0; i<numfeat; i++){
  linsum=linsum+wlin[n*numfeat+i]*x[randnum*numfeat+i];
  }
quadsum=0.0;
```

```
  for(i=0; i<numquad; i++){
  quadsum=quadsum+wquad[n*numquad+i]*quadin[randnum*numquad+i];
  }

sum=linsum+quadsum;
if ((sum+theta[n])<-20.0) {y=0.0;}
  else if ((sum+theta[n])>20.0) {y=1.0;}
    else {y=(double)(1.0/(1.0+exp(-(sum+theta[n]))));}

/* Criteria for training each class to fire a specific neuron */
if (class[randnum]==n) {d=1.0;}
else {d=0.0;}

error=y*(1.0-y)*(d-y);
/* printf("error=%f\n",error); */
  for(i=0; i<numfeat; i++){
 wlin[n*numfeat+i]=wlin[n*numfeat+i]+neweta*error*x[randnum*numfeat+i];
  /* printf("For i=%d  wlin=%f\n",i,wlin[n*numfeat+i]); */
  }
  for(i=0; i<numquad; i++){
  wquad[n*numquad+i]=wquad[n*numquad+i]+
  neweta*error*quadin[randnum*numquad+i];
  /* printf("For i=%d  quadweight=%f\n",i,wquad[n*numquad+i]); */
  }

theta[n]=theta[n]+neweta*error;
/* printf("For neuron # %d  theta+%f\n",n,theta[n]); */
}  /* end of weight training loop for each neuron's weights */

}  /* end of complete training loop */

/* Begin test loop to run thru each test vector */
numright=0;
for(testvec=numexvec; testvec<(numexvec+numttvec); testvec++){
/* printf("testvec=%d\n",testvec); */

neurtcnt=0;
  for(n=0; n<numneur; n++){
  linsum=0.0;
    for(i=0; i<numfeat; i++){
    linsum=linsum+wlin[n*numfeat+i]*x[testvec*numfeat+i];
    }
  quadsum=0.0;
    for(i=0; i<numquad; i++){
    quadsum=quadsum+wquad[n*numquad+i]*quadin[testvec*numquad+i];
    }
  sum=linsum+quadsum;

  if ((sum+theta[n])<-20.0) {y=0.0;}
    else if ((sum+theta[n])>20.0) {y=1.0;}
      else {y=(double)(1.0/(1.0+exp(-(sum+theta[n]))));}
  /* printf("y=%f  class[testvec]=%f\n",y,class[testvec]); */
```

```
  /* Test decision criteria for identifying classes */
  if ((y>0.8) && (class[testvec]==n)) {neurtcnt=neurtcnt+1;}
  if ((y<0.2) && (class[testvec]!=n)) {neurtcnt=neurtcnt+1;}

  }  /* end of neurons (n) correct test loop */

if (neurtcnt==numneur) {numright=numright+1;}
}  /* end of testing loop running thru each test vector (testvec) */

itnum=bigloop*numtrn;
accuracy=(float)numright/(float)numttvec;
printf("Training iteration #: %d   Accuracy=%f\n",itnum,accuracy);

}  /* end of bigloop */

}  /* end of main */
```

Bibliography

1. Ayer, Capt Kevin W. *Gabor Transforms for Forward Looking Infrared Image Segmentation*. MS thesis, AFIT/GEO/ENG/89D-1. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1989 (AD-A215046).
2. Blasdel, Gary G. and Guy Salama. "Voltage-sensitive Dyes Reveal a Modular Organization in Monkey Striate Cortex," *Nature, 321*: 579-585 (5 June 1986)
3. Daugman, John G. "Complete Discrete 2-D Gabor Transforms by Neural Networks for Image Analysis and Compression," *IEEE Transactions on Acoustics, Speech, and Signal Processing, 36*: 1169-1179 (July 1988).
4. Ervin, Frank R. "On-line Computer Techniques for Analysis of the Visual System," *Proceedings of the Symposium on the Analysis of Central Nervous System and Cardiovascular Data Using Computer Methods*. 35-51. Washington DC: NASA, 1965.
5. Fisher, Lawrence M. "A Pioneer is Out on a Limb Again," *The New York Times*, 8 (21 January 1990).
6. Fretheim, Capt Erik, AFIT Doctoral Candidate. "An Introduction to Neural Networks." Special presentation in EENG 621, Pattern Recognition II. Air Force Institute of Technology (AU), Wright-Patterson AFB OH, 6 April 90.
7. Fukushima, K. "Analysis of the Process of Visual Pattern Recognition by the Neocognitron," *Neural Networks, 2*: 413-420 (1989).
8. Giles, C. Lee and Tom Maxwell. "Learning, Invariance, and Generalization in High-order Neural Networks," *Applied Optics, 26*: 4972-4978 (1 December 1987).
9. Gray, Charles M. and Wolf Singer. "Stimulus-specific Neuronal Oscillations in Orientation Columns of Cat Visual Cortex," *Proceedings of the National Academy of Sciences of the United States of America, 86*: 1698-1702 (March 1989).
10. Hubel, D.H. and T.N. Wiesel. "Receptive Fields, Binocular Interaction and Functional Architecture in the Cat's Visual Cortex," *Journal of Physiology (London), 160*: 106-154 (1962)
11. Jones, Judson P. and Larry A. Palmer. "The Two-dimensional Spatial Structure of Simple Receptive Fields in Cat Striate Cortex," *Journal of Neurophysiology, 58*: 1187-1211 (December 1987).
12. -----, Aaron Stepnoski and Larry A. Palmer. "The Two-dimensional Spectral Structure of Simple Receptive Fields in Cat Striate Cortex," *Journal of Neurophysiology, 58*: 1212-1232 (December 1987).

13. Kabrisky, Matthew, Professor, Electrical Engineering. Personal interviews. Air Force Institute of Technology (AU), Wright-Patterson AFB OH, 2 January through 12 October 1990.

14. -----. Class lectures in EENG 621, Pattern Recognition II. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, April 1990.

15. Kammen, Daniel M. *et al.* "Collective Oscillations in Neuronal Networks: Functional Architecture Drives the Dynamics," *Proceedings of the 1990 International Joint Conference on Neural Networks.* I-181–I-184. Hillsdale NJ: Lawrence Erlbaum Associates, 1990.

16. Le Cun, Y. *et al.* "Handwritten Digit Recognition: Applications of Neural Network Chips and Automatic Learning," *IEEE Communications Magazine*, 41-46 (November 1989).

17. Leopold, George. "Neural Network Technology Moves Forward," *Defense News*, (8 January 1990).

18. Lippmann, Richard F. "An Introduction to Computing with Neural Nets," *IEEE ASSP Magazine*, 4-22 (April 1987).

19. Meador, Jack L. "Fast Quadratic Separation Using a Single-layer Interconnect Model," Poster presentation at the 1990 International Joint Conference on Neural Networks. Washington DC, 15–19 Jan 1990.

20. Namatame, Akira. "Backpropagation Learning with High-Order Functional Networks and Analyses of its Internal Representation," *Proceedings of the 1990 International Joint Conference on Neural Networks.* I-680–I-683. Hillsdale NJ: Lawrence Erlbaum Associates, 1990.

21. Rogers, Steven K., Professor, Electrical Engineering. Personal interviews. Air Force Institute of Technology (AU), Wright-Patterson AFB OH, 2 January through 12 October 1990.

22. -----. Class lectures in EENG 515, Linear Systems, Fourier Transforms, and Optics. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, September 1989.

23. ----- *et al. An Introduction to Biological and Artificial Neural Networks.* Bellingham Washington: SPIE, 1990.

24. Ruck, Capt Dennis W. *Characterization of Multilayer Perceptrons and their Application to Multisensor Automatic Target Detection.* PhD dissertation, AFIT/DS/ENG/90-2. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1990.

25. ----- *et al.* "The Multilayer Perceptron as an Approximation to a Bayes Optimal Discriminant Function," Accepted for Publication in *IEEE Transactions on Neural Networks.*

26. Stryker, Michael P. "Is Grandmother an Oscillation?," *Nature, 338*: 297-298 (23 March 1989).

27. Tarr, Capt Gregory L., AFIT Doctoral Candidate. Personal interviews. Air Force Institute of Technology (AU), Wright-Patterson AFB OH, 2 January through 12 October 1990.

28. Tenorio, Manoel F. and Wei-Tsih Lee. "Self Organizing Neural Networks for the Identification Problem," *Proceedings of Advances in Neural Information Processing Systems I.* 57-64. San Mateo CA: Morgan Kaufmann Publishers, 1989.

29. Zhang, Jun and John P. Miller. "A Model for Resolution Enhancement (Hyperacuity) in Sensory Representation," *Proceedings of Advances in Neural Information Processing Systems I.* 444-450. San Mateo CA: Morgan Kaufmann Publishers, 1989.

Vita

Born July 21, 1961 in Philadelphia, Pennsylvania, Captain Laurence E. Lazofson was raised in the northern suburbs of the city. A 1979 graduate of George School, a private preparatory school in Newtown, Pennsylvania, he ranked at the top of his class and competed on the Varsity Cross Country Team. Captain Lazofson earned a Bachelor of Science degree in Natural Science from Muhlenberg College, Allentown, Pennsylvania, graduating cum laude in June 1983.

He entered USAF Officer Training School, San Antonio, Texas, in September 1983. Receiving his commission as an Air Force Officer in December of that year, he then began a two-year tour in the Civilian Institution Program of the Air Force Institute of Technology. During this tour, Captain Lazofson completed a Bachelor of Science in Electrical Engineering at the University of New Mexico, Albuquerque, New Mexico, graduating in December 1985.

In January 1986, he began a tour of duty at Newark AFB, Newark, Ohio, serving over three years as the Reliability and Maintainability Program Manager in the Directorate of Metrology. His responsibilities included the development and implementation of projects aimed at improving dependability and reducing support requirements for test, measurement, and diagnostic equipment acquisitions for Air Force Precision Measurement Equipment Laboratories located throughout the world. He additionally served as a technology application consultant, technical project evaluator, and International Training Management Officer. Captain Lazofson was appointed a Regular Air Force commission in May 1987 and was awarded the Meritorious Service Medal in July 1989 for his contributions during this assignment.

Following completion of Squadron Officer School at Maxwell AFB in Montgomery, Alabama, he entered the School of Engineering at the Air Force Institute of Technology, Wright-Patterson AFB, Ohio in May 1989. He is married to the former Tonda Leigh Johnson of Heath, Ohio.

Permanent address: 3100 Boardwalk
Apartment 1615-1
Atlantic City, New Jersey
08401

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1990	Master's Thesis

4. TITLE AND SUBTITLE

A BIOLOGICALLY-INSPIRED NEURAL NETWORK ARCHITECTURE FOR IMAGE PROCESSING

5. FUNDING NUMBERS

6. AUTHOR(S)

Laurence E. Lazofson, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GEO/ENG/90D-04

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This thesis project included a literature survey of biological and artificial neural network research followed by development and testing of high-order and image recognition hierarchical neural network algorithms. Following training, performance testing of second-order and third-order networks yielded maximum accuracies comparable to those achieved by multilayer perceptron classifiers operating on test data sets. Several versions of an image classification algorithm were tested for learning performance using pixel data from forward-looking infrared (FLIR) images of tanks, trucks, target boards, and clutter. Employing the biologically-motivated Lambertization and contrast normalization of pixel windows, correlations with multiple Gabor function wavelets, and a "phase synchronizing" local averaging routine, the image classification network extracted data features. Different network versions fed the extracted features to varying output classification schemes. To improve separation of problem classes, recommendations were made for varying the parameters of the Gabor function wavelets and modifying the phase synchronization scheme to extract more suitable features from image pixel data.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Artificial Intelligence, Image Processing, Neural Nets, Pattern Recognition, Target Recognition, Theses. (JS)	157
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL