The parallel implementation of a backpropagation neural network and its applicability to SPECT image reconstruction

by

John Patrick Kerr

A Thesis Submitted to the

Graduate Faculty in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Major: Biomedical Engineering

Approved: Members of the Committee:

Iowa State University
Ames, Iowa

1992

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Although the backpropagation neural network was initially developed in the late 1960's (Bryson and Ho, 1969), the capabilities of this or any other artificial neural network (ANN) have only become widely appreciated in recent years. The interest in ANNs is due to their ability to provide solutions to problems that conventional computer architectures cannot solve readily, or at all. Areas in which ANNs have shown promise are speech and pattern recognition (Tom and Tenorio, 1991), control systems (Garg and Floyd, 1991), and image enhancement (Kuperstein, 1991).

The architecture of a feedforward neural network is modeled after the neural structure in the cerebral cortex of the brain. Feedforward neural networks are constructed of two types of simple building blocks: processing elements and interconnections (Figure 1.1). Each processing element (PE) is functionally equivalent to the soma (body) of a neuron. A PE sums the input values it receives and computes a single output via an activation function. An interconnection weighs the input value it receives from a PE and delivers this value to a PE on the next higher layer, much as an action potential generated at the hillock of a soma travels across its axon and synapses to a dendrite of another neuron in the cerebral cortex (Figure 1.2). The knowledge the network ultimately possesses is represented by the weight values associated with the interconnections. In order to generate the appropriate weights to solve the problem

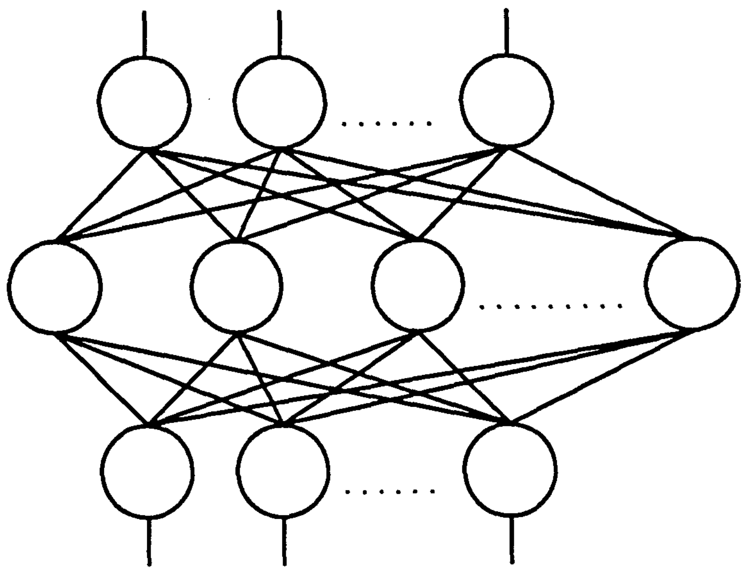

Figure 1.1 Simple neural network block diagram

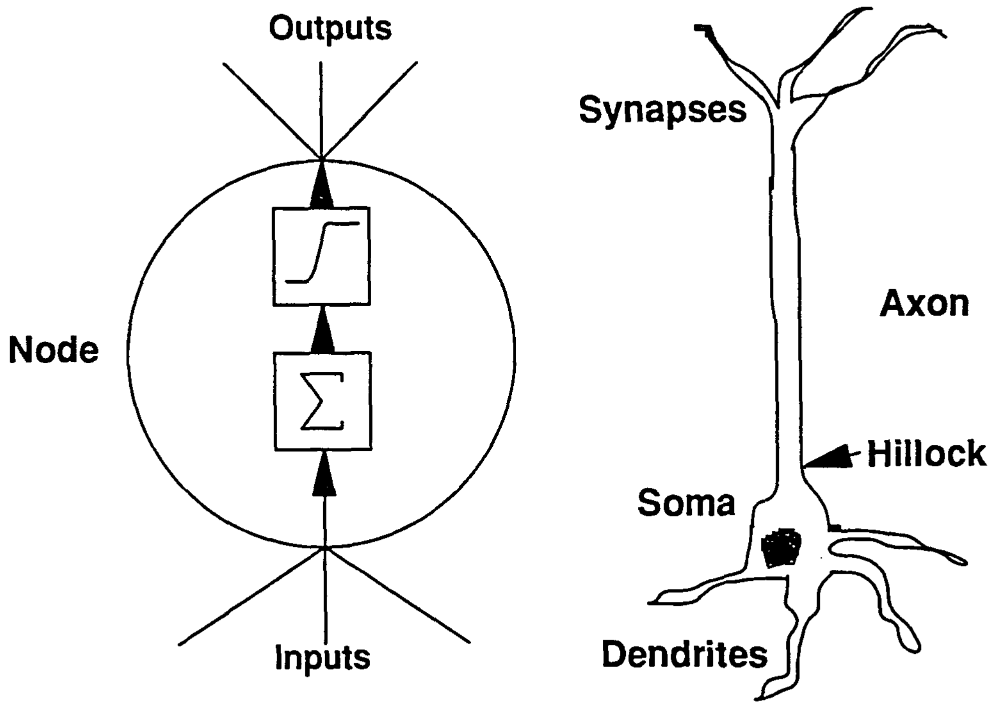

Figure 1.2 Comparison of ANN processing element and neuron

at hand, the network must be trained.

Backpropagation is a popular, often implemented training algorithm. In this algorithm a known set of training data is fed through the network and the error, which is the calculated difference between the expected output and the actual output, is then fed back through the network. The interconnection weights are adjusted with respect to the backpropagated error to minimize the error in the outputs (Hecht-Nielsen, 1989). Thoroughly training a network often requires presenting the entire training set thousands or even hundreds of thousands of times. As the size of the problem being addressed by the neural network grows, the ability of the ANN to be trained in a reasonable amount of time becomes the limiting factor in using it, particular when the network is implemented on a serial (single processor) computer.

The architecture of a multi-layer neural network has a natural parallel structure. The PEs in each layer receive inputs simultaneously from the previous layer and process individual outputs independent of each other. Consequently, implementing an ANN on a traditional single processor, or von Neumann, computer requires a considerable amount of processing time to execute the same instructions for each PE one at a time. The vast majority of ANNs developed to date have been implemented on serial machines. However, an interconnected array of processors does have a comparable architecture to that of a multi-layer ANN and could be adapted through programming to accommodate a neural network.

The DAP (Distributed Array of Processors) architecture, initially developed in 1972 at ICL's Research and Advanced Development Centre (Flanders et al., 1990), possesses exactly this type of massively parallel, fine-grain processing power. The DAP is a single instruction multiple data (SIMD) system. This means that each processor, or processing unit (PU), executes the same instructions as all other PUs, but with a different set of data. The time saved training an ANN implemented on a parallel computer might provide the improved training time necessary to apply neural networks to more complex problems that require large architectures.

Medical image reconstruction is a good example of a larger problem. Reconstruction would require a network to produce a three-dimensional construct of the data collected from a series of planar images taken of a target organ, or target area of a patient. The input to such a network would be a large image array (e.g. 64 x 64). Training a network of this size (e.g. 4196 input nodes and 4196 output nodes) on a serial computer could not typically be accomplished in a reasonable amount of time, but training this size of network on a DAP may be very practical. One area of image reconstruction in which a neural network may improve on the reconstruction algorithms used presently, is in the nuclear medical field of emission computed tomography (ECT).

ECT is a radiopharmaceutical imaging technique that provides three-dimensional distribution mapping of a radionuclide after its administration to a patient. The radionuclide is scanned with a gamma ray camera after it localizes in the target area of the body. When reconstructed, the three-dimensional image provides information on pathologic processes and physiological functioning in the target organ. ECT imaging is primarily used for identifying organ lesions and evaluating metabolic processes. Reconstruction is typically achieved via a filtered back*projection* algorithm (Larsson, 1980). Although this method works well, it does have several limitations that create a high percentage of statistical uncertainty in the reconstructed images (Budinger, 1983).

2. OBJECTIVE

The objective of this study was to determine the feasibility of using an ANN, in particular a backpropagation ANN, to improve the speed and quality of the reconstruction of three-dimensional SPECT (single photon emission computed tomography) images. In addition, since the PEs in each layer of an ANN are independent of each other, the speed and efficiency of the neural network architecture could be better optimized by implementing the ANN on a massively parallel computer.

The specific goals of this research were:

1. To implement a fully interconnected backpropagation neural network on a serial computer and a SIMD parallel computer,
2. To identify any reduction in the time required to train these networks on the parallel machine versus the serial machine,
3. To determine if these neural networks can learn to recognize SPECT data by training them on a section of an actual SPECT image, and
4. To determine from the knowledge obtained in this research if full SPECT image reconstruction by an ANN implemented on a parallel computer is feasible both in time required to train the network, and in quality of the images reconstructed.

3. BACKGROUND

3.1 Backpropagation Neural Network

The backpropagation neural network is by far the most widely utilized ANN today . Although originally discovered in 1969 (Bryson and Ho, 1969) and independently rediscovered several times since then (Parker, 1986; Werbos, 1987), credit for developing backpropagation into a widely known, viable ANN technique belongs with Rumelhart and collaborators (Rumelhart et al., 1986; Rumelhart and McClelland, 1986)./Perhaps the most notable step in the development of backpropagation involved the use of a sigmoidal activation function (Figure 3.1) as the output, or decision function, of each PE rather than the more traditional hard-limiting quantizer. A hard-limiting quantizer outputs either +1 or -1, which is determined by the sum of the inputs and the preset threshold of each PE. Unlike the hard-limiting quantizers, the nonlinearity of the sigmoidal function provides a flexibility in the network that allows it to generalize, and thus learn more efficiently (Lippmann, 1987).

A backpropagation neural network is often referred to as a *mapping neural network* (Hecht-Nielsen, 1989). Mapping difficult functions, or functional relationships which would otherwise be hard to evaluate, can be readily trained into the neural network. The network is trained on a set of randomly selected inputs from the mapping

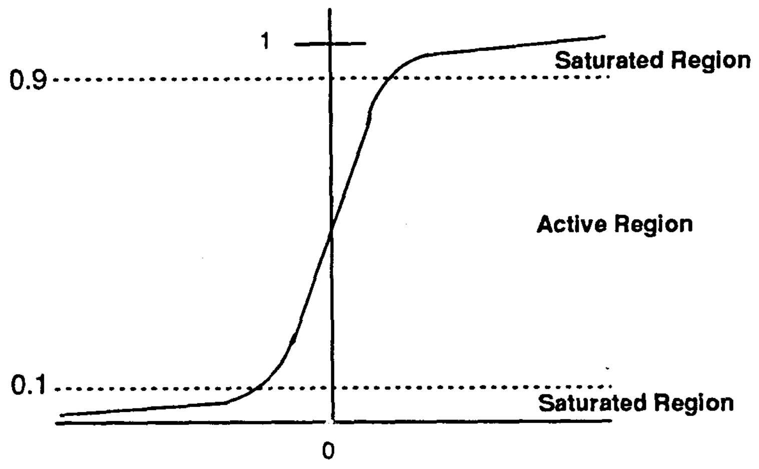

Figure 3.1 Sigmoidal activation function

area range, for which the associated outputs are known. The network does not simply memorize the training set, but generalizes the training set data to closely approximate the mapping between and around the known data.

3.1.1 Backpropagation Architecture

While the basic building blocks of the backpropagation neural network, as seen in Figure 3.2, are the same for each network, the actual architecture of the ANN depends primarily on the problem at hand. A detailed block diagram of a fully-interconnected backpropagation neural network is shown in Figure 3.3. The architecture has an input layer, an output layer, and a user selectable number of hidden layers, typically one. The number of PEs in the input layer and in the output layer is defined by the size of the data vector being evaluated and by the size of the output vector desired from the ANN.

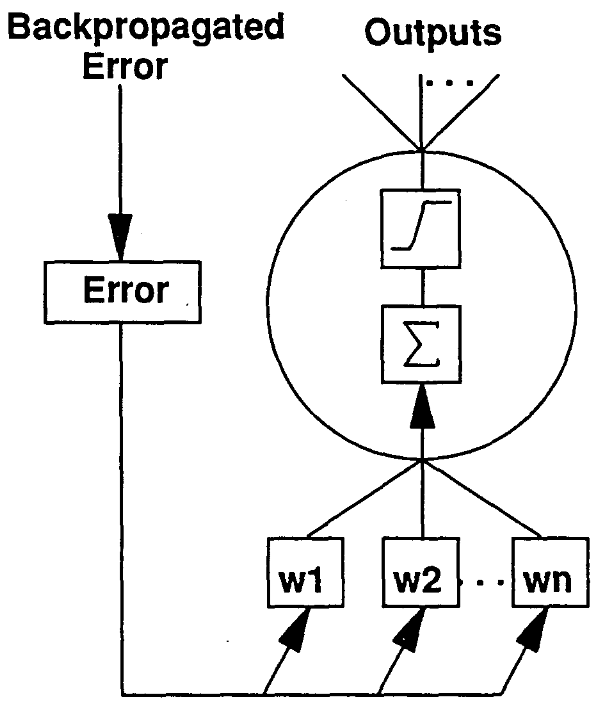

Figure 3.2 Backpropagation processing element diagram

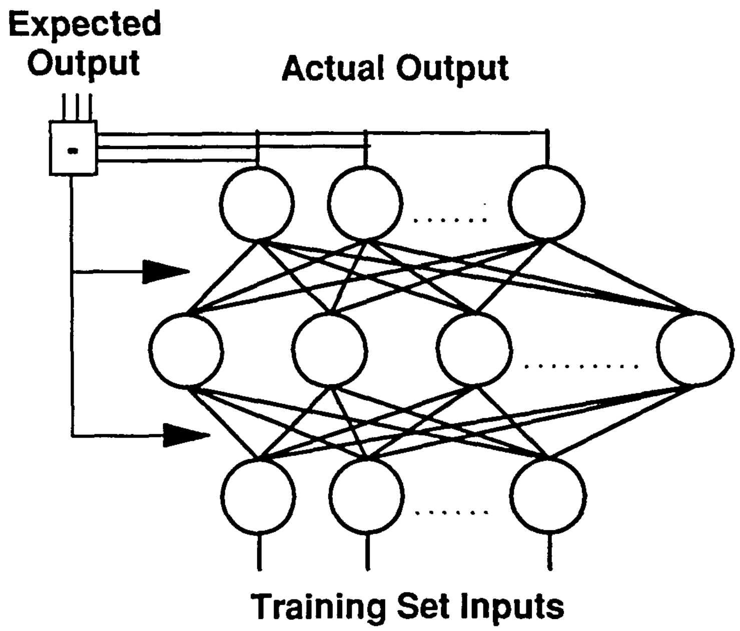

Figure 3.3 Functional diagram of backpropagation ANN

The size of the hidden layer, or layers, is not as well defined. If the hidden layer is too small, the network will learn slowly and will require an excessive number of iterations of the training set. If the hidden layer is too large, the network will *grandmother* (Caudill, 1988a), or memorize the training set. Grandmothering tends to inhibit generalization by the network and thus, after training, the network does not produce accurate outputs for new, unfamiliar input patterns. There are some "rules of thumb" for choosing the hidden layer size. One rule suggests determining the number of nodes by multiplying the square root of the sum of the input and output nodes by 1.2 and rounding that value up to the next integer. While there is research into the use of a dynamic node architecture (Bartlett and Basu, 1991), or a genetic algorithm (Heistermann, 1990) to determine the number of hidden nodes, the most prevalent method is by trial and error; training a variety of architectures to determine which one performs the best.

Once a particular architecture has been chosen to be trained, a random weight value is assigned to each interconnection. Like various neurotransmitters in the synaptic connections between neurons, positive weight values are excitatory, and negative weights are inhibitory, to the next node. These weights are adjusted during training, and represent the knowledge possessed by the network when training is complete.

3.1.2. Feedforward Data Flow

In the typical backpropagation algorithm, a datum vector applied to the input layer propagates through the network, generating an output vector that is determined by the architecture and weight values of the network. Each node value from the input layer is multiplied by the weight value associated with each interconnection from that node and delivered to a node in the hidden layer. Each node in the hidden layer then sums the weighted values it receives from the input layer (Equation 3.1).

$$I_{li} = g\{ \sum_{j=0}^{N} w_{lij} * z_{(l-1)ij} \} \qquad (3.1)$$

where

I_{li}	=	The sum of the inputs to the ith PE in layer l.
g	=	The gain term, which affects the slope of the activation function.
w_{lij}	=	The weight value of the interconnection from the jth PE of the previous layer, to the ith PE in layer l.
$z_{(l-1)ij}$	=	The output of the jth PE in layer (l - 1) to the ith PE in layer l.

The outputs of the PEs in the hidden layer(s) and output layer are characterized by a nonlinear sigmoidal activation function, such as Equation 3.2.

$$f(\alpha) = 1 / (1 + e^{-(\alpha - \theta)}) \qquad (3.2)$$

where

α	=	I_{li} (Equation 3.1)

θ = The internal offset, or threshold.

This output is again multiplied by each interconnection weight as it is passed to the PEs in the next layer. The values from the sigmoidal functions in the output layer nodes represent the network result for the given input vector.

3.1.3. Training the Network

Since the interconnections are initialized with random weights, these weights must be adjusted in order for the ANN to produce the proper outputs. This is achieved by feeding the output error back through the network to modify the interconnection weights; "training" the network.

When a network is being used to map a function, the training set is chosen randomly from the input domain as determined by a fixed probability density function, p(x) (Hecht-Nielsen, 1989). In practical applications the problem domain may not be well defined, but it is important that the training set be a good representation of the problem domain. The more diverse the training set, the better the generalization of the network and the better the performance of the network.

The error or "cost" of the network during training is determined by Equation 3.3.

$$C = \{1/(N^*J) * [\sum_{n=0}^{l} \sum_{j=0}^{m} (D_{nj} - Y_{nj})^2]^{1/2}\} \qquad (3.3)$$

where

N	=	The number of output nodes.
J	=	The number of patterns in the training set.
D_{nj}	=	The desired output of the jth node for the nth training vector.
Y_{nj}	=	The actual output of the jth node of the nth training vector.

Equation 3.3 represents the root mean square (RMS) error, also referred to as the least mean square (LMS) error, of the desired output minus the actual output. Part of the error associated with each interconnection's weight is fed back through the network, decreasing the magnitude of the RMS error. The learning rule used to adjust network weights is called the Delta Rule (Equation 3.4).

$$w_{new\,ji} = w_{old\,ji} + \eta * \sigma_j * y_i \qquad (3.4)$$

where

$w_{old\,ji}$	=	The present weight value of the interconnection from the ith node of the previous layer to the jth node of the present layer.
η	=	The learning rate, or gain of the network (typically, $0.0 < \eta < 1.0$).

σ_j	=	The difference between the desired output and the actual output of the jth node, multiplied by the derivative of the sigmoidal activation function.
y_i	=	The actual output of the ith node of the previous layer.

The learning rate determines the magnitude of change in each weight update. A large learning rate may converge more rapidly toward the absolute minima, but it may also tend to oscillate around local minima in the cost function.

The sigmoidal activation function in Equation 3.4 is important because it is a nonlinear function, which provides a network the ability to learn the relationship between training data. The network derives the actual correlation between data points and not just a linear interpolation. The sigmoid is also important because its derivative is a straightforward, easily programmable computation (Equation 3.5).

$$f'(\alpha) = f(\alpha) * (1 - f(\alpha)) \qquad (3.5)$$

This derivative is incorporated into σ_j of Equation 3.4. This derivative is a gaussian-shaped curve which results in interconnection weights of nodes with outputs near zero or one to be changed very little, and outputs near 0.5 to be changed more by the backpropagation. This is particularly useful in networks with binary outputs, since the interconnection weights to indecisive nodes are adjusted more during training. For the interconnections that feed from the last hidden layer into the output layer:

$$\sigma j = x_j(1 - x_j) * (d_j - x_j) \quad \text{Delta Rule} \qquad (3.6)$$

where

x_j	=	The actual output of the jth node.
d_j	=	The desired output of the jth output node.

and, for the interconnections that input into a hidden layer:

$$\sigma j = x_j(1 - x_j) * \sum_{k=0}^{N} (d_k - x_k) * w_{jk} \quad \text{Generalized Delta Rule} \qquad (3.7)$$

where

d_k	=	The desired output of the kth node in the next feedforward layer.
w_{jk}	=	The weight of the interconnection from the jth node in the hidden layer to the kth node in the next feedforward layer.

With each iteration of the training set through the network, and backpropagation of the error, the cost ideally approaches zero (Equation 3.3). While the goal of training is to reach the absolute minimum cost, there may be local minima and plateaus on which the network can become indecisive as to which direction along the error space to proceed (Figure 3.4). This is a result of the backpropagation algorithm minimizing the present error without regard to previous adjustments. The training time required for problems in which the network tends to be indecisive may be improved with the addition of a momentum term (Equation 3.8).

$$w_{new\,ji} = w_{old\,ji} + \eta * \sigma_j * x_j + \alpha(w_{old\,ji} - w_{previous\,ji}) \qquad (3.8)$$

where

α	=	The momentum term ($0 < \alpha < 1$).
$w_{previous\,ji}$	=	The weight value of the interconnection prior to $w_{old\,ji}$.

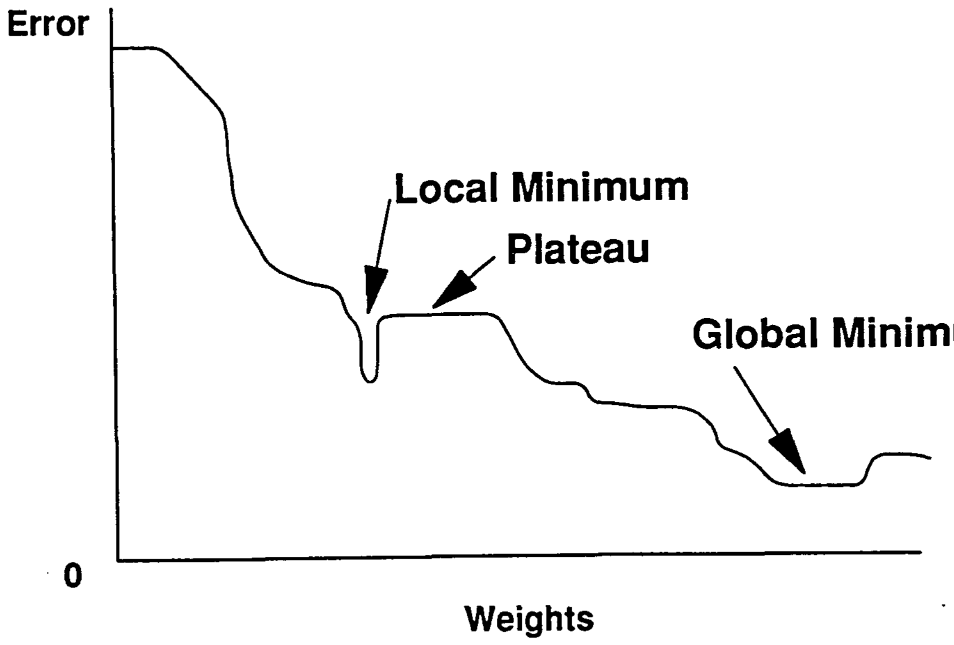

Figure 3.4 Example of ANN descending-search error function

The larger α is the more momentum there is in the direction the weights are being adjusted. The momentum term gives an impetuous to the training network to get out of local minima and to move in one direction on plateaus, instead of oscillating back and forth (Caudill, 1988b).

The main drawback in using a backpropagation ANN is frequently the great mount of time required to train the network. Each data vector of the training set must feedforward through the network, and the error must be fed back. This, coupled with the fact that the entire training set may have to be presented to the network several thousand times, makes training very time consuming. This is particularly so for an ANN implemented on a serial machine. These limitations to backpropagation architectures can be significantly reduced, however, by implementing the neural network on a parallel machine.

3.2 Massively Parallel Computing

Massively parallel processing involves optimizing the processing power of a large number of individual processors, or processing units (PUs), which are interconnected with high speed communications links. A significant improvement in performance over single processor, or von Neumann type, architectures can be achieved by distributing a problem across many, often thousands, of processors; all of which are executing code simultaneously. Large problems can be solved quickly with this type of system.

The primary distinction between types of parallel architectures is in whether the PUs execute *identical* instructions on different data (single instruction multiple data (SIMD)), or whether the PUs execute *unique* sections of code on separate data streams (multiple instruction multiple data (MIMD)) (Desrochers, 1987). While MIMD can be used in a wider range of applications than can SIMD (Almasi and Gottlieb, 1989), the MIMD PUs must be larger and more complex, since they have to decode instructions and handle program counters. Conversely, SIMD PUs execute decoded instructions broadcast to them by a controller processor. The SIMD processors are less complex, and, thus, larger arrays of PUs can be more readily built.

3.2.1 SIMD System Architecture

The SIMD parallel system is composed of two individual systems: the front-end computer, and the massively parallel computer. The front-end computer is a serial machine which manages access to the parallel computer and also provides an environment in which the parallel code can be developed. The front-end can host multiple users and typically provides an array of programming tools. Access to the parallel machine, or data processing unit (DPU), can be provided to only one user at a time. A compiled program is downloaded to the DPU and executed entirely by the DPU. The PU array is activated by special parallel commands. Serial instructions are executed by the DPU control processor which usually is a small processor with execution rates comparable to those of a microcomputer.

The DPU architecture has a certain inherent synchronicity since a single processor controls instruction decoding and PU intercommunications. The control processor will broadcast an instruction to all active PUs and will not execute the next instruction in the code until all PUs have completed the previous command. The PUs are made up of simple processors with small local memories (Figure 3.5).

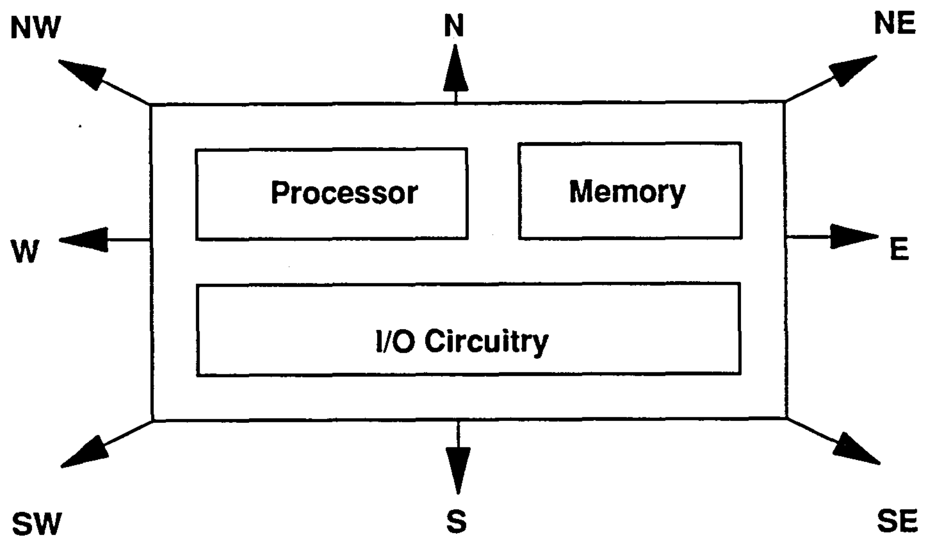

Figure 3.5 Processing unit block diagram

The PU array performance is limited by the speed of its processors. Individual PUs, which possess very small processors, do not execute instructions as fast as more complex single processor computers. The advantages to SIMD computing are realized by breaking a problem down into simple calculations and distributing these among the many PUs. Therefore, the architecture of the PU array and how the instruction code distributes a problem on the array are critical to the processing performance reaped from a SIMD machine.

3.2.2 Distributed Array of Processors

A massively parallel computer consists of a large interconnected matrix (e.g., 128x128) of processors, often referred to as a distributed array of processors. The matrix is controlled by a single high-speed master control processor (MCP). The DAP is typically connected in a two-dimensional mesh of north, south, east, and west communication pathways. Often northeast, northwest, southeast, and southwest communication pathways are incorporated into the interconnection mesh. Each processing unit (PU) consists of a processor, local memory, and input/output controller (Figure 3.5). The MCP broadcasts decoded instructions to all "active" (receiving and executing code) PUs, and the PUs in turn execute that code on data stored in their local memory or registers. The next broadcasted instruction is not executed until all PUs have completed the previous instruction (Almasi and Gottlieb, 1989).

Since the individual PUs typically have small 1-, 2-, or 4-bit processors, floating point operations and complex functions require considerable processing time. This is an important consideration when developing parallel code. While distributing the program on the PU array can greatly improve execution times, that improvement is somewhat tempered by the fact that the PUs have small processors which are individually slow. This is particularly true of floating point operations. If floating point calculations can be done via integer operations then the PU array can be better utilized, and thus execute code more efficiently.

3.2.3 SIMD Programming

Most parallel languages are based around a high-level serial language, such as FORTRAN or "C". All of the high-level language commands are retained and additional parallel extensions are used to identify which commands are to be executed on the PU array. Parallel declarations are used to distinguish PU variables from MCP variables. The MCP executes all serial instructions, and decodes and broadcasts parallel instructions to all active PUs.

Communications between PUs can be direct, or indirect. Both types of communication are controlled by the MCP. The difference in communication schemes lies in the path by which information travels, and the speed with which it is exchanged. Direct communications are through high-speed interconnections, and can only be carried out between PUs that lie on directional pathways (e.g., N, S, E, W) (Figure 3.5). Indirect communications are required when interacting PUs are not connected on a directional pathway. In these cases, information is transmitted from the PU to the MCP, and then the MCP channels the information to the target PU. When outlining a program it is important to know that direct intercommunications are substantially faster than indirect communications. Therefore, when distributing a program on the array of processors, positioning the majority of interacting PUs so as to take advantage of the high-speed pathways is vital to producing fast code.

Although there is no one preferred method to parallel programming, most programmers will have a good working knowledge of the serial language upon which the parallel language is based. They will also have a serial mind set to programming unless they have had previous parallel programming experience. For those without previous parallel experience, it is often much easier to program exclusively with serial commands, and develop and debug a working model of the code in serial before introducing parallel commands. In this way parallel instructions can be systematically incorporated into a working model of the code, and errors, such as unfamiliar syntactical conventions, can be readily isolated. This technique allows for the working model of the program to be adapted until an acceptable execution rate is achieved.

3.3 Single Photon Emission Computed Tomography (SPECT)

Emission Computed Tomography (ECT) utilizes the emitted radiation of a conventional nuclear medical radionuclide to produce a diagnostic three-dimensional image, reconstructed from a series of two-dimensional planar projection images. The ability to produce three-dimensional images from two-dimensional planar views of an object was mathematically demonstrated by Radon in 1917. Kuhl and Edwards initially investigated emission tomography in 1963. They superimposed rectilinear scans to produce tomographic images. These early images were blurred by extraneous noise, and physical photon effects. In the early 1970's, better reconstruction algorithms, originally developed for x-ray computed tomography, were adapted for ECT.

ECT has since developed in two directions, each of which is defined by the type of radiation emitted by the tracer used: positron emission computed tomography (PET), and single photon emission computed tomography (SPECT). PET is based on the detection of a pair of photons released 180° opposite each other when a emitted positron interacts with an electron. Commonly used radionuclides in PET imaging are F-18 and C-11 (Macovski, 1983). While PET can be used for some very unique diagnostic applications, the short half-life of the positron emitters requires that these tracers be produced at the time of use. This means that a high cost cyclotron particle accelerator must be on hand to produce the radiopharmaceuticals. SPECT on the other hand, relies on the detection of gamma ray emitting tracers, such as Tc-99m and Ti-201 (Jaszczak, 1988). Since these tracers have longer half-lifes, an onsite cyclotron is not required. For this reason a SPECT system is far less expensive to implement into the clinical environment.

3.3.1 SPECT Data Collection

A variety of radiopharmaceuticals are used in SPECT imaging. Different radionuclides localize better in certain regions of the body and are selected according to the target organ to be imaged (Jaszczak, 1988). The technicium isotope, Tc-99m, is an example of a widely used radionuclide. The advantages of using Tc-99m are threefold: it does not require a cyclotron to produce, it has a short 6-hour half-life so the patient is exposed to the radiation for a brief period of time, and it has a gamma ray emission energy of 140 kev, which is a low energy level similar to the energy used in x-rays, and so has little effect on cells and tissues (Macovski, 1983).

The detecting element of the photons emitted by the radionuclide is a sodium iodine crystal (NaI(TI)), which emits visible light photons after exposure to gamma photons (Cho and Ra, 1984). A photomultiplier tube (PMT) converts the light photons to electrical pulses which are manipulated to produce an image that can be recorded and displayed (Figure 3.6). The Anger camera (Jaszczak, 1988), or gamma ray scintillation camera, resolves the problem of limited resolution that is inherent in cameras composed of an array of discrete detectors (Figure 3.6). The Anger camera (Figure 3.7) channels the gamma rays through a multiple hole collimator on to a large NaI(TI) crystal. Instead of having a PMT for each collimator hole, there is a small array of PMTs which are exposed to light from the scintillation crystal. The magnitude of the signals produced by each PMT can be used to pinpoint where a gamma ray was

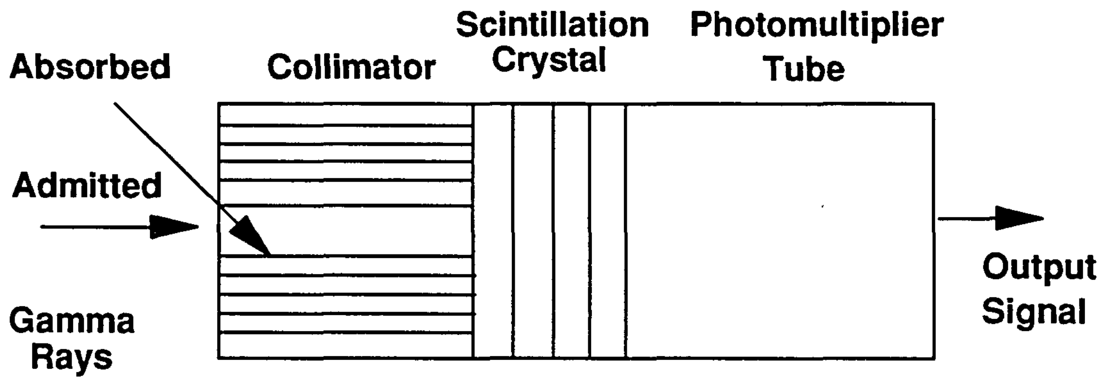

Figure 3.6 Discrete gamma ray detector element

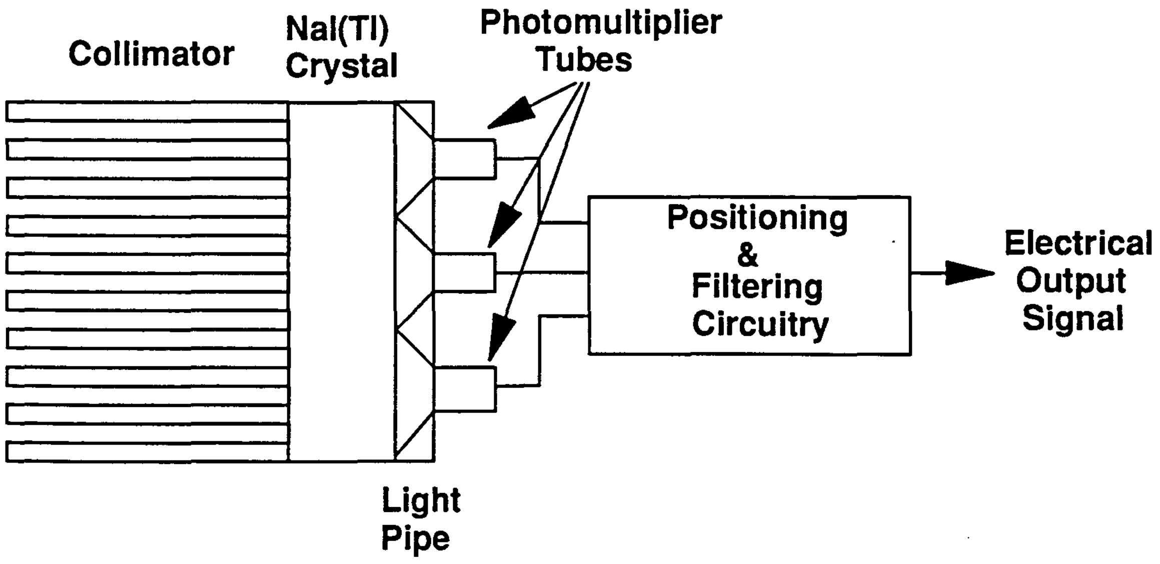

Figure 3.7 Anger camera diagram

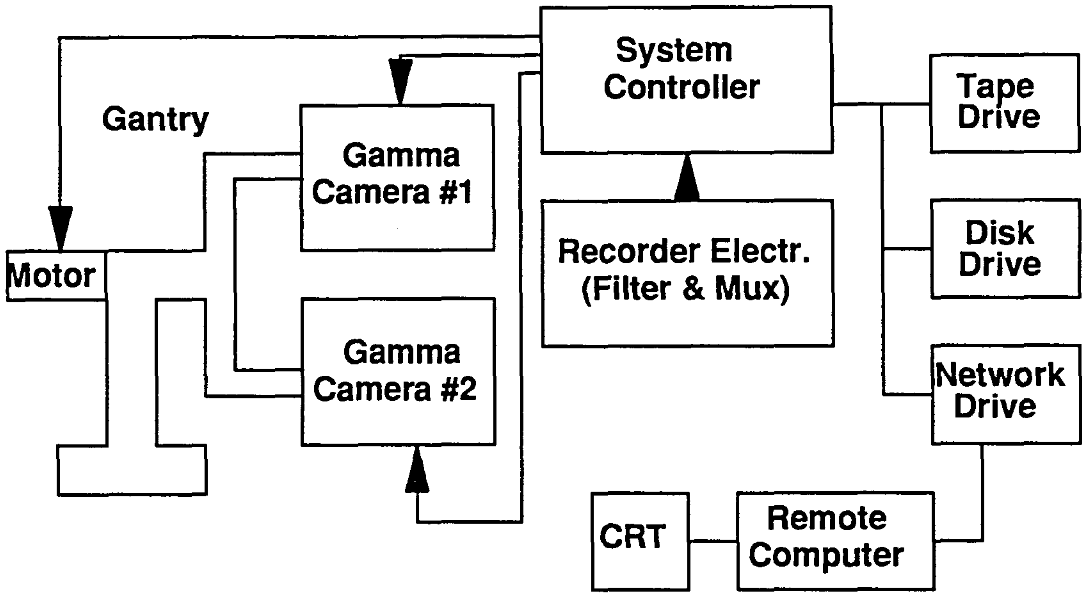

Figure 3.8 SPECT system block diagram

absorbed. This not only simplifies the camera, but it also allows a high resolution image to be produced with a limited number of detectors (Jaszczak, 1988).

Unlike radiology where the scintillation image is recorded on film, the visible photons detected by the PMTs are converted to electrical signals and the image corrected for attenuation and scatter effects, if desired (Larsson, 1980). Compton scattering, in which a photon colliding with a free or bound electron is deflected at a new angle and at a lower energy level, can cause significant distortion that often appears as fog in the image (Macovski, 1983). Some of the scatter is absorbed by the collimator material, typically lead, since the photons are not travelling parallel to the channels (Figure 3.7). The scattered photons that do reach the detector may be eliminated by a filter circuit in the recorder (Figure 3.8). After conversion of photon events to electrical signals, some filtering of scattered photons is possible since they may have energy levels well below that of the energy level of the isotope administered. Likewise, multiple events have a higher than expected energy and can be identified and discarded by the recorder. The recorder circuitry determines the position of each event and eliminates photons outside the expected energy range. Once these planar images are recorded the SPECT system generates three-dimensional image slices of the scanned area using a reconstruction algorithm.

3.3.2 Reconstruction Algorithms

The gamma ray camera is typically mounted on a gantry so that equally spaced incremental planar images can be taken over a 180° rotation, or 360° in some systems, about the patient. These multiple views of the target organ are necessary to provide enough information to reconstruct image slices of the target area. Simple backprojection methods were initially used by Kuhl and Edwards to perform ECT reconstructions (Kuhl and Edwards, 1968). These methods, however, produced very blurred images that provided limited information. Since then, filtering prior to backprojection reconstruction or after reconstruction, termed filtered backprojection, has proven to produce clearer tomographic images. Filtered backprojection algorithms are predominantly used in clinical medical imaging. Despite this wide acceptance of filtered backprojection, it does suffer from a variety of noise effects, such as scatter and attenuation, which create a high percentage of statistical uncertainty in the reconstructed images.

Filtered backprojection encompasses a wide range of reconstruction techniques. The majority of these methods can be classified as either analytic or algebraic.

Analytical algorithms perform a spatial or frequency filtering of the planar data prior to backprojection, or after backprojection. Algebraic algorithms use iterative techniques in which the projection data sums are adjusted until the difference between calculated values and experimental values fall within a predetermined acceptance range.

Each filtered backprojection method has its advantages. Comparisons between analytical and algebraic techniques show that while algebraic methods are easier to implement, the analytical methods' abilities to handle incomplete and noisy gamma camera images make them the preferred backprojection techniques (Larsson, 1980). Additional sources of error, such as systematic errors, artifacts, and physical photon transport effects (scatter and attenuation), can to some extent be compensated for, but the compensation methods greatly increase reconstruction time.

Techniques, that perform reconstruction with much less statistical error than filtered backprojection, have been developed but require extensive computational time. Once such method, called the Monte Carlo method, improves on the accuracy of the reconstructed slices by modeling the physics of SPECT imaging and uses this information in conjunction with a standard reconstruction algorithm to correct for scatter and attenuation (Gordon and Herman, 1974; Shepp and Vardi, 1982). However, while such techniques give a reconstructed image a more desirable degree of error, the reconstruction time required makes their use in clinical situations impractical.

4. LITERATURE REVIEW

Backpropagation neural networks have found wide application in a variety of areas: speech and pattern recognition (Carpenter et al., 1991; Fukushima, 1990; Kim and Lee, 1991; Hiraiwa et al., 1990; Tom and Tenorio, 1991), image compression and enhancement (Garg and Floyd, 1991; Mougeot et al., 1991), and control systems (Abbas and Chizeck, 1991; Kuperstein, 1991). A survey of the literature revealed only the article by E. B. Bartlett and this author applying neural networks to medical image reconstruction (Appendix I).

While image reconstruction is a new field for neural network application, ANNs have found a wide range of other uses in medical imaging. Considerable research into the diagnosis and detection of physiological abnormalities in medical images using ANNs has been done. These studies primarily utilize the pattern recognition abilities of ANNs as image classifiers (Hudgins et al., 1991) and as aides in medical diagnosis, such as vascular necrosis determination from a magnetic resonance image (MRI) (Manduca et al., 1991), and in detection of organ lesions and nodes, such as from x-ray scans (Garg and Floyd, 1991) and in PET images (Kippenhan et al., 1991).

Simulating a backpropagation neural network on a parallel machine has recently attracted considerable interest (Brown et al., 1989; Koikkalainen and Oja, 1991; Schiffmann and Mecklenburg, 1990). There is no one best implementation of a neural network on the PU array of a SIMD parallel machine. Two typical implementation schemes are: (i) to dedicate a PU to each node and each interconnection in the neural network (Hicklin and Demuth, 1988), and (ii) to dedicate one node and one interconnection from each layer of the neural network to a single PU (Yoon et al., 1990). The higher degrees of processing efficiency are primarily dependant on utilizing the entire PU array and reducing the number of floating point operations whenever possible. Integer operations can be quickly executed by the small processors in the PU array, but floating point operations and complex function calls are detrimental to the PU arrays performance. An efficient backpropagation ANN simulated on a parallel machine can execute two to three orders of magnitude faster than an ANN simulated by a serial machine (Kamgar-Parsi et al., 1988). Additionally, the development of a integer-based backpropagation algorithm would make a SIMD ANN simulation even more efficient.

A great number of three-dimensional reconstruction algorithms have been developed to improve on the standard backprojection algorithm without incurring an excessive cost in time. Spatial filtered and frequency filtered backprojection techniques (Larsson, 1980) are used in the majority of clinical applications. Techniques which have less statistical error in their reconstructions, such as a Monte Carlo algorithm (Shepp and Vardi, 1982), would be better in clinical situations than filtered backprojection algorithms if it were not for the time required for reconstruction. The Monte Carlo technique involves a randomized searching scheme to determine the best reconstruction image. The time required to randomly search the domain of possible reconstruction patterns is computational expensive even for a parallel machine.

5. MATERIALS AND METHODS

5.1 SPECT Data Description

The projection data used to train the neural network and make reconstruction error measurements was generated from a clinically reconstructed SPECT image of a human brain. Data manipulation routines were required to extract 8x8 sections from image slices, project the 8x8 slices into eight 8-quadrant planes, normalize output data for network training, and convert 8x8 images into a displayable format. The "C" code developed to execute these conversions and data manipulations is listed, along with functional descriptions, in Appendix II.

Each 8x8 section used for ANN training was taken from a clinical 64x64 SPECT image slice. The planar inputs were generated by projecting each image slice into eight 8-quadrant planes. Each plane was rotated 22.5° from the previous plane around the image, giving eight incremental views covering 180° around the image (Figure 5.1). Each of the eight quadrants of each planar view represented the summation of the intensity values projected from the 8x8 section.

Two sets of training data were used to demonstrate SPECT reconstruction. The first set consisted of a single datum. The objective of training a single image was to demonstrate a neural network's ability to memorize a SPECT image. The second data set consisted of two parallel sections separated by one slice from the clinical SPECT

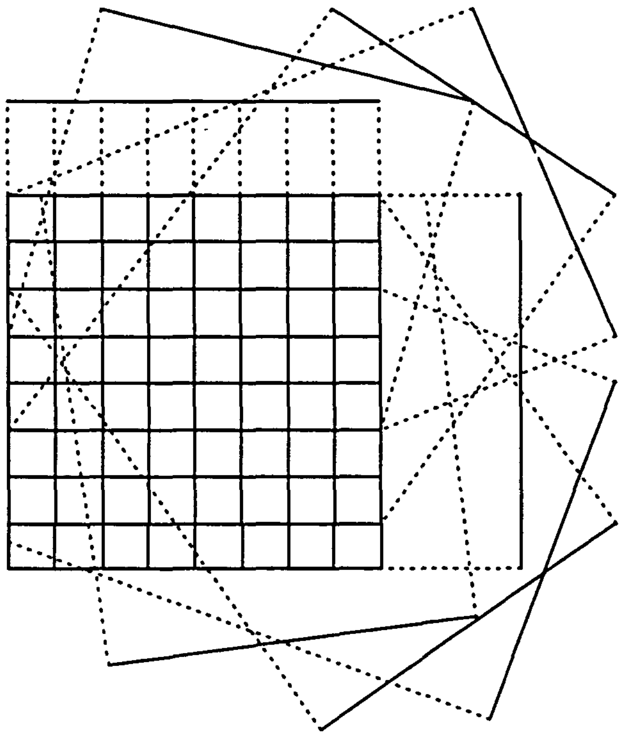

Figure 5.1 Eight planar projections from 8x8 SPECT section

image. The objective here was not only to show the ANN's ability to recall more than one image, but also to determine its ability to generalize the relationship between the planar and reconstructed data. This was achieved by testing the network's ability to reconstruct the untrained center image slice.

5.2 Backpropagation ANN Implementation

The fully-interconnected backpropagation neural networks were simulated on a serial computer and a parallel computer. The networks were composed of 64 input nodes to accommodate the eight 8-quadrant projection planes and 64 output nodes to produce the 8x8 image reconstructions. Various hidden layer architectures were implemented and trained both for parallel-to-serial processing time comparisons, and for determining the optimal architecture for learning the reconstruction relationship. The architectures which converged the fastest and had the smaller RMS errors in reproducing the slices are identified in the results chapter of this text. The feedforward and backpropagation equations were implemented as cited in the background chapter of this text. A learning rate of 0.3 (equation 3.4) and a momentum term of 0.0 (equation 3.7) were used in each neural network simulation.

The serial computer (VAXStation 3520) neural network simulation program was coded in "C" and is listed, along with a functional description, in Appendix III. The parallel system (MasPar MP-1) neural network simulation program was coded in "MasPar Parallel Language" (MPL), a parallel derivative of "C". MPL possesses all the "C" commands and libraries, as well as a small set of parallel specific-extensions and commands. The parallel code is listed, along with a functional description, in Appendix IV.

All output values in the training sets were normalized to a 0.1 to 0.9 range. This was done so that the maximum and minimum output values of the sigmoidal activation function from each node (equation 3.2) were in the active region of the function and not in the saturated region near 0 and 1 (Figure 3.1). This technique limits all the PE outputs to the active region of the sigmoidal function, which gives the neural network better generalizing capabilities. It also allows the ANN to be more readily trained, since driving the activation functions of the PEs into the saturated regions takes many more iterations of the training set.

5.3 MasPar MP-1 SIMD Parallel Computer

5.3.1 MP-1 Architecture

The MasPar MP-1 is a single instruction multiple data (SIMD), massively parallel machine. The MP-1 is composed of a 128x128 interconnected array of 4-bit processors (Figure 5.2). The MP-1's performance is dependant on how readily the problem at hand can be distributed among its 16,384 processors. Access to the PE array, program development, and program debugging are the functions of the front-end system (Figure 5.2), which in this case is a VAXStation 3520. The MP-1, commonly referred to as the data processing unit (DPU), consists of an arithmetic control unit (ACU) which decodes and broadcasts instructions to the PU array and executes all serial commands, and a PU array which is a 128x128 torodial mesh of 4-bit processors and memory elements (Figure 5.2).

5.3.2 MPL Extensions and Commands

MPL is based on Kernighan and Ritchie "C", first edition (Kernighan and Ritchie, 1978). MPL provides low-level access to the DPU. A detailed knowledge of the DPU is necessary to properly control processing on the PU array. Control of the processing units in the array is via predefined variables, user-defined parallel variables, PU-to-ACU communications, and PU-to-PU communications.

Both singular and parallel variables are defined in the MPL library **"mpl.h"**.

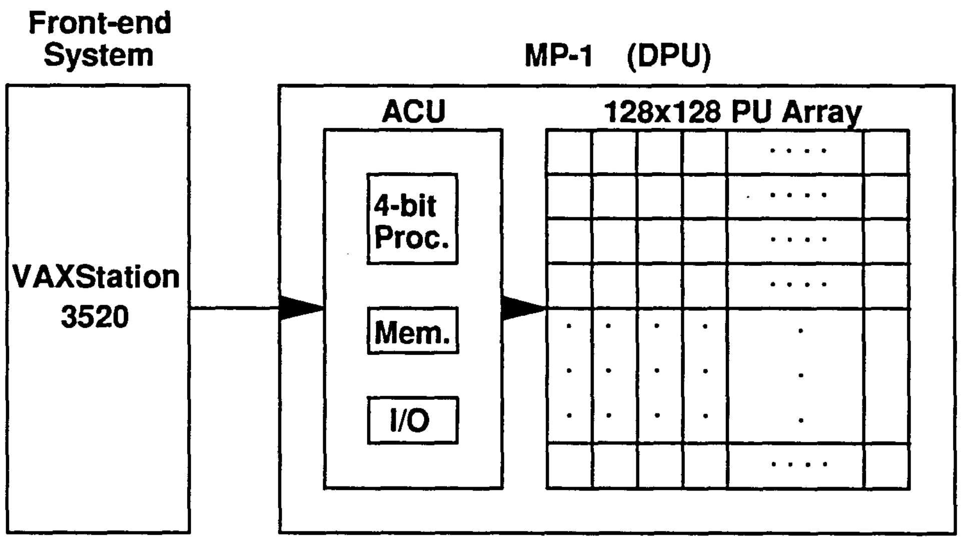

Figure 5.2 MasPar MP-1 system block diagram

Variables **nproc**, **nxproc**, and **nyproc** define the total number of PUs in the array, the number of PUs in the x-direction of the array, and the number of PUs in the y-direction of the array, respectively. Likewise, singular variables **lnproc**, **lnxproc**, and **lnyproc** contain the number of bits required to represent the values in **nproc**, **nxproc**, and **nyproc**. These values are useful in masking out processing units, which can be used to change the size of the active PU array during code execution.

User-defined parallel variables are variables which can be declared locally in each PU. Singular variables can be declared in the ACU only. The syntax for a parallel declaration is distinguishable from that of a singular declaration by the addition of the **plural** extension.

```
plural float x;
```

Each PU allocates a memory location for the variable, in this case **x**. In a SIMD parallel computer each PU executes the same code, but on different data. This is accomplished in the MP-1 by having variables with identical names allocated in all PUs, but with different data values.

PU-to-ACU communications are via the **proc** instruction.

```
proc[x][y].val = acu_val;
```

This example of the **proc** command sets the variable **val** in the PU located at coordinates **x, y** equal to the ACU variable **acu_val**.

```
acu_val = proc[x][y].val;
```

Likewise, this example of the **proc** command sets the ACU variable **acu_val** equal to the value of plural variable **val** in PU **x, y**.

PU-to-PU communications can be carried out via an **xnet** command or the **router** command. The **xnet** commands use high-speed communications pathways to transfer data directly from one PU to another. To use **xnet** commands the communicating PUs must lie on one of the eight directional pathways: north, south, east, west, northeast, northwest, southeast, southwest (Figure 3.5). The twenty-four different **xnet** commands are listed in Table 5.1. **xnetp** and **xnetc** differ from the basic PU-to-PU **xnet** commands. The **xnetp** (pipeline) commands are faster, but require all PUs in between the communicating PUs to be inactive. On the other hand, **xnetc** (copy) commands change the value of the variable being transferred, and in all the PUs in between, to the value of the variable being broadcasted.

Table 5.1 : Listing of all MPL xnet commands.

Plain Access	Copy	Pipeline
xnetN	xnetcN	xnetpN
xnetS	xnetcS	xnetpS
xnetE	xnetcE	xnetpE
xnetW	xnetcW	xnetpW
xnetNE	xnetcNE	xnetpNE
xnetNW	xnetcNW	xnetpNW
xnetSE	xnetcSE	xnetpSE
xnetSW	xnetcSW	xnerpSW

```
xnetNW[5].val = new_val;
```

This **xnet** command example sets the variable **val** in all PUs, that are five PUs away from all active PUs in the northwest direction, equal to the value of **new_val** in each respective PU.

The **router** command is not as fast as the **xnet** commands since all data is sent to the ACU and then forwarded to the receiving PU. However, the **router** command is not restricted to any pathways and can be used to exchange data between PUs regardless of their location on the PU array.

```
router[3].val = new_val;
```

This **router** command example instructs each PU, whose **iproc** value is three greater than an active PU, to set its variable **val** equal to value of **new_val** from the associated PU.

5.3.3 Neural Network Distribution on the PU Array

The architecture implemented to determine the feasibility of SPECT reconstruction by an ANN required 64 inputs for the eight planar projections and 64 outputs to produce the 8x8 reconstructed image slice. The optimal number of PEs in the hidden layer was determined by training a number of networks with various hidden layer sizes. The neural networks were distributed on the processor array by dedicating a processor to each node and each interconnection in the network (Figure 5.3). The nodes of the network were aligned on the array so as to best utilize the high speed north-south, east-west communication pathways. This distribution of the network allowed data to propagate from layer to layer more efficiently than it could through indirect pathways. This implementation scheme limits the maximum ANN architecture to no more than 128 input and output nodes combined, and to no more than a 128 nodes in the hidden layer. This was an acceptable utilization of the PU array for 8x8 image reconstructions, but could not be used for larger reconstructions, such as full 64x64 SPECT images.

I_1	$w_{1,1,1}$	$w_{1,1,2}$	$w_{1,1,3}$	. . .	$w_{1,1,j}$		
I_2	$w_{1,2,1}$	$w_{1,2,2}$	$w_{1,2,3}$	. . .	$w_{1,2,j}$		
I_3	$w_{1,3,1}$	$w_{1,3,2}$	$w_{1,3,3}$	. . .	$w_{1,3,j}$		
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$		
I_{64}	H_1/w_1	H_2/w_2	H_3/w_3	. . .	H_j/w_j		
	$w_{2,1,1}$	$w_{2,2,1}$	$w_{2,3,1}$	. . .	$w_{2,j,1}$	O_1	
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
	$w_{2,1,63}$	$w_{2,2,63}$	$w_{2,3,63}$	. . .	$w_{2,j,63}$	O_{63}	
	$w_{2,1,64}$	$w_{2,2,64}$	$w_{2,3,64}$	. . .	$w_{2,j,64}$	O_{64}	

Figure 5.3 ANN distribution on 128x128 PU array.

6. RESULTS

The objectives of this research were two-fold: (i) simulate a backpropagation neural network on a SIMD parallel computer and evaluate its performance, and (ii) determine the feasibility of SPECT image reconstruction from planar images via a backpropagation neural network simulated on the parallel computer.

6.1 Parallel Vs. Serial Processing Rates

The efficiency of the parallel backpropagation code, listed in Appendix IV, to the serial code, listed in Appendix III, was determined by measuring the execution time required to train each backpropagation network architecture on a single datum training set for 1000 iterations. Five different ANN architectures consisting of 64 input-nodes, 64 output-nodes, and 8, 16, 24, 32, and 40 hidden-nodes were simulated with both the serial and parallel codes. The execution times on the MP-1, the parallel machine, and the VAXStation 3520, the serial machine, are listed in Table 6.1. The results in Table 6.1 show that the processing rates on the VAXStation 3520 increased at a nearly linear rate of approximately 55 seconds for each additional 8-hidden nodes. Conversely, each incrementally larger hidden node architecture had comparatively small increases in processing time on the MP-1. Table 6.1 shows that smaller architectures, below

64x8x64, execute faster on the serial machine than on the parallel machine. On the other hand, as the architectures get larger the processing rate on the parallel machine becomes significantly smaller than that of the serial machine.

Table 6.1: Serial vs. parallel processing rates

ANN Architecture 1000 iterations of a single datum training set (time)					
	64x8x64	64x16x64	64x24x64	64x32x64	64x40x64
VAXStation 3520 (serial)	62 s	135 s	176 s	225 s	277 s
MasPar MP-1 (parallel)	62 s	66 s	72 s	75 s	77 s

6.2 SPECT Image Reconstruction

The feasibility of SPECT reconstruction by a backpropagation neural network was evaluated via two training tests; (i) training different ANNs on a single 8x8 SPECT section to determine each network's ability to recall a known image, and (ii) training different networks on two 8x8 sections and then determining each network's ability to recall an untrained 8x8 section.

6.2.1 Single Image Reconstruction

A number of different neural network architectures were trained on a single SPECT image. Each network architecture was trained to an RMS error (equation 3.3) of zero (single-precision 16-bit floating point), except for the architecture with only one hidden node. All other architectures reproduced the SPECT images exactly. This precise recall accuracy was achieved in less than 100 training iterations for all trainable architectures, except for 2, 3, 4, and 5 hidden-node architectures which required as many as 200 iterations to reach an RMS error of zero. Table 6.2 lists the architectures tested, the number of iterations required to reach minimum RMS error, and the minimum RMS error achieved by each network architecture.

The one hidden-node architecture was the only network that did not converge to an RMS error of zero. Different random number values for the interconnects were tried, but the network always converged to the same RMS error (0.020802). Since the input and output layers were large, it is not surprising that the one hidden node could not process all the data it received from the input layer.

6.2.2 Untrained Image Reconstruction

The next task was to determine a neural network's ability to generalize the training set in order to accurately produce novel images. This was tested by training each ANN on two 8x8 parallel images from the clinical SPECT image, which were separated by one image slice, and then generating the untrained 8x8 reconstructed

Table 6.2 : Minimum RMS error obtained from single image training set.

# Hidden Nodes	# Training Iterations	RMS Error
1	3000	0.020802
2	110	0.000000
3	175	0.000000
4	110	0.000000
5	170	0.000000
6	< 100	0.000000
7	< 100	0.000000
8	< 100	0.000000
9	< 100	0.000000
10	< 100	0.000000
11	< 100	0.000000
12	< 100	0.000000
13	< 100	0.000000
14	< 100	0.000000

center slice given its planar images as inputs to the network. The full 64x64 reconstructed SPECT image, from which the untrained 8x8 section was taken, is shown in Figure 6.1. The actual untrained 8x8 SPECT image section, from which the eight 8-quadrant planar projections were calculated, and the ANN generated 8x8 section are shown in Figure 6.2 and Figure 6.3, respectively. The network that produced the image in Figure 6.3 had a 64x8x64 architecture and was trained for 6000 iterations on the two-image training set. The network achieved an RMS error of 0.001928 on the two-image training set. The generated output image had an RMS error of 0.001231. This means that the trained network reproduced the untrained image better than it could produce either image in the training set. It is most likely that the generalizing characteristics of the neural network, induced by the sigmoidal activation functions, learned a nonlinear average of the two training set images allowing the network to reproduce the untrained middle image with better accuracy than either training image. Table 6.3 lists various architectures and RMS errors achieved with the two-image training set.

Table 6.3 Minimum RMS error obtained from two-image training set

#Hidden Nodes	# Training Iterations	RMS Error
8	6000	0.001928
10	10000	0.002240
16	7000	0.002156
20	2000	0.002946

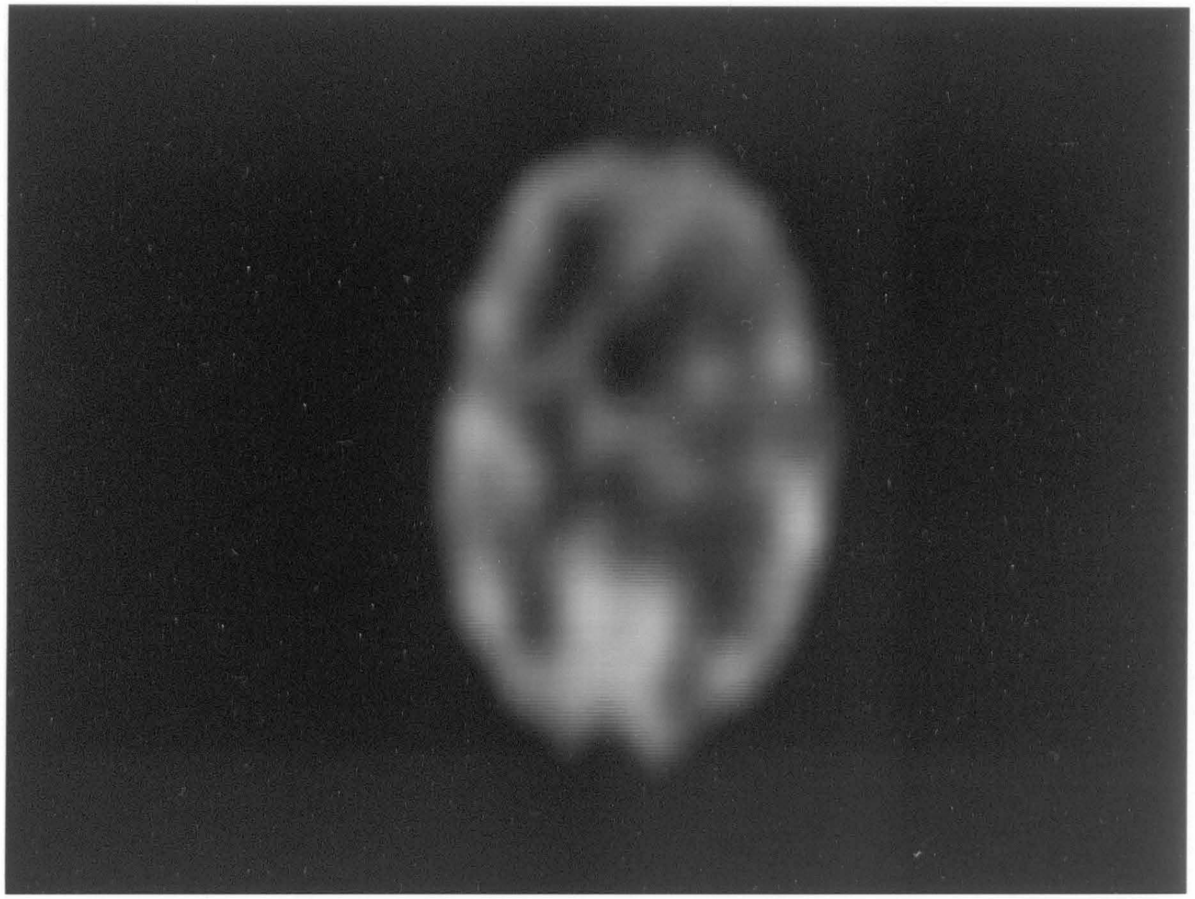

Figure 6.1 Conventionally reconstructed 64x64 SPECT image slice

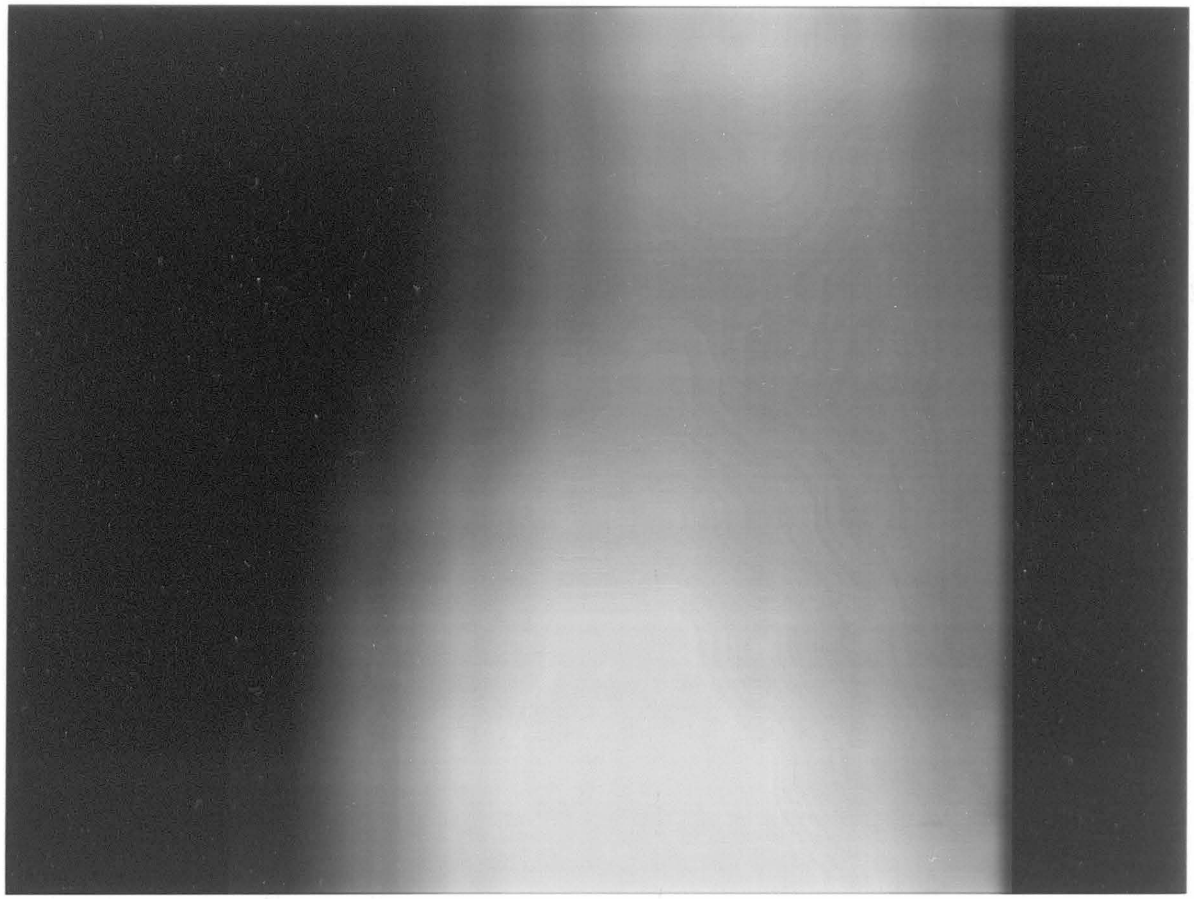

Figure 6.2 8x8 section of conventionally reconstructed SPECT image

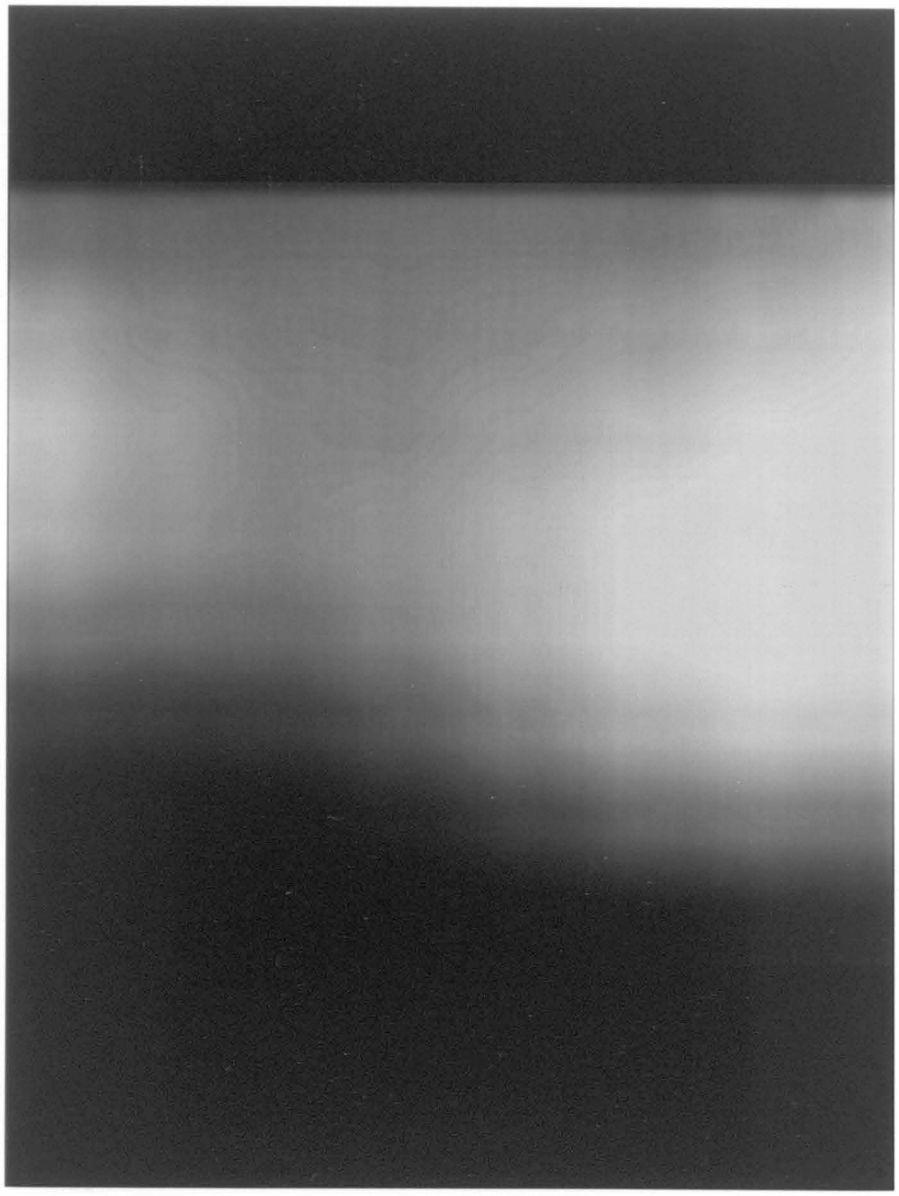

Figure 6.3 Ann reconstructed 8x8 section (figure 6.2) from untrained data

7. DISCUSSION AND CONCLUSIONS

Each neural network's ability to memorize a SPECT image, and to generalize between two SPECT image slices to reconstruct an untrained center image, shows that full SPECT image reconstruction via an ANN is feasible. Although statistical comparisons of reconstructed images to the original images were made, the most important measure of the ANN's image reconstruction ability is a visual one. This is particularly true of SPECT information collection, which is through visual interpretation. It can be seen in Figure 6.2 and Figure 6.3 that the anatomical features and the gamma ray intensities of the original SPECT section are present in the image reconstructed by the ANN.

The parallel simulation of the ANNs on the MP-1 demonstrated the advantages of training very large networks on a parallel machine. Smaller networks will typically train faster on a serial machine since high-speed 16- and 32-bit processors can handle smaller networks better than a few 4-bit processors on a parallel machine. These results demonstrate that larger architectures cannot be trained in a reasonable amount of time on a serial computer, and that an ANN simulated and trained on a parallel computer will be required if full 64x64 SPECT reconstruction via an ANN is to be achieved.

The next step in developing SPECT reconstruction with an ANN will be to train the network on every other slice from all the slices of a SPECT brain image. This will be important in determining if the ANN can generalize over a large range of images and accurately reproduce untrained slices. From this point the same approach taken for 8x8 image reconstruction will be applied to full 64x64 SPECT images. This will require a modification to the parallel implementation algorithm of the ANN. Although the processing times achieved with the MP-1 are encouraging, many high-speed serial machines could more efficiently train the architectures used in this paper. Superior training rates are attainable through better utilization of the PU array and perhaps through the development of an integer-based backpropagation algorithm. The parallel ANN simulation code developed for this study can only handle 64 input-nodes and 64 output-nodes. Therefore, a new algorithm will be necessary for full SPECT image reconstruction, and for what may eventually be a more efficient SPECT reconstruction technique.

The primary constraint to accurate ECT image reconstruction involves the presence of noise, systematic errors, artifacts, and physical photon transport effects such as scatter and attenuation (Budinger, 1983; Jaszczak. 1988). Improved SPECT reconstruction via an ANN may be achieved by compensating for these problems in the training set. One way in which the accuracy of reconstruction can be improved is by modeling the physics of the problems in conjunction with the reconstruction algorithm used to produce the ANN training set. The Monte Carlo reconstruction method can be used to simulate attenuation, scatter, and other effects so that the ANN can be trained to correct for these undesired artifacts (Gordon and Herman, 1974; Shepp and Vardi, 1982). ANN SPECT reconstruction would then offer the advantages associated with the Monte Carlo method, but without the inherent computational cost. Ultimately, ANN SPECT reconstruction would require training just one network on simulated data. The network could then be used for all SPECT reconstructions, regardless of the SPECT system used, the target organ imaged, or the radiopharmaceutical administered.

REFERENCES

Abbas, J., and H. Chizeck, "A Neural Network Controller for Functional Neuromuscular Stimulation Systems," in *Proc. of the Annual Int. Conf. of the IEEE - Engineering in Medicine and Biology Society*, vol. XIII, 1991, pp. 1456-1457.

Almasi, G., and A. Gottlieb, *Highly Parallel Computing.* Redwood City : The Benjamin/Cummings Publishing Company, Inc., 1989.

Bartlett, E., and A. Basu, "A Dynamic Node Architecture Scheme for Backpropagation Neural Networks," in *Intelligent Engineering Systems through Artificial Neural Networks.* New York : ASME Press, 1991, pp. 101-106.

Brown, J., M. Garber, and S. Venable, "Artificial Neural Network on a SIMD Architecture," in *Proc. of the 2nd Symp. on the Frontiers of Massively Parallel Computation*, 1989, pp. 43-47.

Bryson, A., and Y. Ho, *Applied Optimal Control.* New York: Blaisdell, 1969.

Budinger, T., "Current Status and Limitations of Single Photon Emission Imaging," in *Diagnostic Imaging in Medicine.* Boston : Martinus Nijhoff Publishers, 1983, pp. 278-298.

Carpenter, G., S. Grossberg, and J. Reynolds, "A Self Organizing ARTMAP Neural Architecture for Supervised Learning and Pattern Recognition," in *Neural Networks Theory and Applications.* San Diego : Academic Press, Inc., 1991, pp. 43-80.

Caudill, M., "Neural Networks Primer : Part III," *AI Expert*, pp. 53-59, June 1988a.

Caudill, M., "Neural Networks Primer : Part IV," *AI Expert*, pp. 61-67, August 1988b.

Cho, Z., and J. Ra, "Methods and Algorithms Toward 3-D Volume Image Reconstruction with Projections," in *Lecture Notes in Medical Informatics.* New York : Springer-Verlag, 1984, pp. 1-39.

Desrochers, G., *Principles of Parallel and Multiprocessing.* New York : McGraw-Hill Book Company, 1987.

Flanders, P., D. Hunt, S. Reddaway, and D. Parkinson, "Efficient High Speed Computing with the Distributed Array Processor," in *Massively Parallel Computing with the DAP.* London : Pittman Publishing, 1990, pp. 23-36.

Fukushima, K., "Neural Network Models for Visual Pattern Recognition," in *Parallel Processing in Neural Systems and Computers.* North-Holland : Elsevier Science Publishers, 1990, pp. 351-356.

Garg, S., and C. Floyd, "The Training of an Artificial Neural Network for a Detection Task in Medical Images," in *Proc. of the Annual Int. Conf. of the IEEE - Engineering in Medicine and Biology Society*, vol. XIII, 1991, pp. 1407-1408.

Gordon, R., and G. Herman, "Three-Dimensional Reconstruction from Projections : A Review of Algorithms," *Int. Review of Cytology*, vol. 38, pp. 111-151, 1974.

Hecht-Nielsen, R., "Theory of the Backpropagation Neural Network," in *International Joint Conference on Neural Networks (IJCNN)*, vol. 1, 1989, pp. 593-605.

Heistermann, J. "Learning in Neural Nets by Genetic Algorithms," in *Parallel Processing in Neural Systems and Computers.* North-Holland : Elsevier Science Publishers, 1990, pp. 165-168.

Hicklin, J., and H. Demuth, "Modeling Neural Networks on the MPP," in *Proc. of the 2nd Symp. on the Frontiers of Massively Parallel Computation*, 1988, pp. 39-42.

Hiraiwa, A., K. Shimohara, and Y. Tokunaga, "EEG Topography Recognition by Neural Networks," *IEEE Engineering in Medicine and Biology*, vol. 9, no. 3, pp. 39-42, September 1990.

Hudgins, B., P. Parker, and R. Scott, "A Neural Network Classifier for Multifunction Myoelectric Control," in *Proc. of the Annual Int. Conf. of the IEEE - Engineering in Medicine and Biology Society*, vol. XIII, 1991, pp. 1454-1455.

Jaszczak, R., "Tomographic Radiopharmaceutical Imaging," *Proceedings of the IEEE*, vol. 76, no. 9, pp. 1079-1094, September 1988.

Kamgar-Parsi, B., J. Gualtieri, and J. Devaney, "How to Cluster in Parallel with Neural Networks," in *Proc. of the 2nd Symp. on the Frontiers of Massively Parallel Computation*, 1988, pp.31-38.

Kernighan, B., and D. Ritchie, *The C Programming Language.* Englewood Cliffs : Prentice Hall, 1978.

Kim, E., and Y. Lee, "Handwritten Langual Recognition Using a Modified Neocognitron," *Neural Networks*, vol. 4, no. 6, pp. 743-750, 1991.

Kippenhan, J., W. Barker, S. Pascal, R. Duara, and J. Nagel, "Optimization and Evaluation of a Neural-Network Classifier for PET Scans of Memory-Disorder Subjects," in *Proc. of the Annual Int. Conf. of the IEEE - Engineering in Medicine and Biology Society*, vol. XIII, 1991, pp. 1472-1473.

Koikkalainen, P., and E. Oja, "The CARELIA Simulator : A Development and Specification Environment for Neural Networks," in *Advances in Control Networks and Large-Scale Parallel-Distributed Processing Models - Volume 1.* New Jersey : Ablex Publishing Corporation, 1991, pp. 242-272.

Kuhl, D., and R. Edwards, "Image Separation Radioisotope Scanning," *Radiology*, vol. 80, pp. 653-662, 1963.

Kuhl, D., and R. Edwards, "Reorganizing Data from Transverse Section Scans of the Brain Using Digital Processing," *Radiology*, vol. 91, pp. 926-935, 1968.

Kuperstein, M., "INFANT Neural Controller for Adaptive Sensory-Motor Coordination," *Neural Networks*, vol. 4, no. 2, pp. 131-145, 1991.

Larsson, S., "Gamma Camera Emission Tomography," *Acta Radiologica*, Supplementum 363, Stockholm, 1980.

Lippmann, R., "An Introduction to Computing with Neural Nets," *IEEE ASAP Magazine,* pp. 4-22, April 1987.

Macovski, A., *Medical Imaging Systems.* Englewood Cliffs : Prentice-Hall, Inc., 1983.

Manduca, A., P. Christy, and R. Ehman, "Neural Network Diagnosis of Avascular Necrosis from Magnetic Resonance Images," in *Proc. of the Annual Int. Conf. of the IEEE - Engineering in Medicine and Biology Society*, vol. XIII, 1991, pp. 1429-1431.

Mougeot, M., R. Azencott, and B. Angeniol, "Image Compression with Backpropagation : Improvement of the Visual Restoration Using Different Cost Functions," *Neural Networks*, vol. 4, no. 3, pp. 467-476, 1991.

Parker, D., "A Comparison of Algorithms for Neuron-Like Cells," in *Proc. of the 2nd Annual Conf. of Neural Networks for Computing*, vol. 151, 1986, pp. 327-332.

Radon, J. "Uber die Bestimmung von Funktionen durch ihre Integralwerte langs gewisser Mannigfaltigkeiten," *Berichte Uber Die Verhandlungen,* pp. 262-277, 1917.

Rumelhart, D., G. Hinton, and R. Williams, "Learning Representations by Back-Propagating Errors," *Nature*, vol. 323, pp. 533-536, October 9, 1986.

Rumelhart, D., and J. McClelland, *Parallel Distributed Processing : Explorations in the Microstructure of Cognition - Vol. I,II, & III.* Cambridge : M.I.T Press, 1986&7.

Schiffmann, W., and K. Mecklenburg, "Genetic Generation of Backpropagation Trained Neural Networks," in *Parallel Processing in Neural Systems and Computers.* North-Holland : Elsevier Science Publishers, 1990, pp. 205-208.

Shepp, L., and Y. Vardi, "Maximum Likelihood Reconstruction for Emission Tomography," *IEEE Transactions on Medical Imaging*, vol. MI-1, pp. 113-122, 1982.

Tom, M., and F. Tenorio, "Short Utterance Recognition Using a Network with Minimum Training," *Neural Networks*, vol. 4, no. 6, pp. 711-722, 1991.

Werbos, P., "Building and Understanding Adaptive Systems: A Statistical/Numerical Approach to Factory Automation and Brain Research," *IEEE Trans. on Systems, Man, and Cybernetics*, vol. SMC-17, no. 1, pp. 7-20, January/February 1987.

Yoon, H., J. Nang, and S. Maeng, "A Distributed Backpropagation Algorithm of Neural Networks on Distributed-Memory Multiprocessors," in *Proc. of the 3rd Symp. on the Frontiers of Massively Parallel Computation*, 1990, pp. 358-363.

ACKNOWLEDGEMENTS

I would like to express my gratitude to Dr. Eric Bartlett for sharing his wisdom, support, and guidance throughout this research; as well as giving me the opportunity to pursue research in a field I truly enjoy. Likewise, I would like to thank Dr. David Carlson and Dr. Satish Upda for their help and service on my program of study committee. Special thanks to Mike Carter and the Scalable Computing Laboratory, at Ames Laboratories, for access to and assistance with the MasPar parallel computer.

I would also like to thank my mother, Rita, and brother, Tim, for their support in my decision to return to college, Wm. Daniel Mundt for his help and friendship without which I could not have returned to college, and to the McDonnell Aircraft Company for the impetuous to return to college and pursue an enjoyable, worthwhile career.

Finally, thanks to the faculty, staff, and students in the Biomedical Engineering Program for their help and support in getting me through the program.

APPENDIX I

"SPECT RECONSTRUCTION USING A BACKPROPAGATION NEURAL NETWORK IMPLEMENTED ON A MASSIVELY PARALLEL SIMD COMPUTER"

John P. Kerr and Eric B. Bartlett

Publish Date : 14 June 1992
IEEE Computer-Based Medical Systems Conference
Durham, North Carolina

SPECT RECONSTRUCTION USING A BACKPROPAGATION NEURAL NETWORK IMPLEMENTED ON A MASSIVELY PARALLEL SIMD COMPUTER

John P. Kerr and Eric B. Bartlett
Biomedical Engineering Program
Iowa State University
Ames, IA 50011

Abstract

In this paper, the feasibility of reconstructing a single photon emission computed tomography (SPECT) image via the parallel implementation of a backpropagation neural network is shown. The MasPar, MP-1 is a single instruction multiple data (SIMD) massively parallel machine. It is composed of a 128x128 array of 4-bit processors. The neural network is distributed on the array by dedicating a processor to each node and each interconnection of the network. An 8x8 SPECT image slice section is projected into eight planes. It is shown that based on the projections, the neural network can produce the original SPECT slice image exactly. Likewise, when trained on two parallel slices, separated by one slice, the neural network is able to reproduce the center, untrained image to an RMS error of 0.001928.

Introduction

In recent years, artificial neural networks (ANNs) have been the subject of extensive theory, implementation and applications research. Spawned by the ever-increasing processing power of computers, ANNs have proven to be useful in applications for which conventional techniques have had difficulty. Such applications include pattern and speech recognition, and image enhancement.

One area in which the image enhancement capabilities of neural networks may be applied is nuclear medical emission computed tomography (ECT). ECT utilizes the radiation emitted by a medical radionuclide to produce a three-dimensional image. This image is reconstructed from a series of two-dimensional projections. Reconstruction is typically achieved through a computationally expensive filtered backprojection algorithm [1]. Although this method provides useful diagnostic information, it does have several limitations that create high statistical uncertainty in the reconstructed image [2,3]. Neural networks on the other hand, have the capability of handling many of the causes of these uncertainties, including attenuation and scatter effects. However, the training time required to simulate a reconstruction ANN large enough to handle useful images (i.e., 64x64) may not be practical. Training a network fully, often requires presenting the entire training set several thousands of times. Therefore, the time in which the ANN can be trained is an important consideration in ECT reconstruction.

Although much success has been achieved with neural networks, the applicability of ANN's to large-scale problems has been limited. Implemented primarily through simulation on digital serial computers, the size of the neural network and hence, the size of the problem that can be evaluated is limited by the processing speed of the implementing computer. The architecture of a multi-layer neural network has a natural parallel structure. One way to utilize this architecture and improve processing time is to simulate the ANN on a parallel machine.

The objective of this paper is to demonstrate the feasibility of single photon emission computed tomography (SPECT) image reconstruction via a backpropagation neural network [4], implemented on the MasPar MP-1 parallel computer.

SPECT Data Set

Small sections of conventionally reconstructed SPECT images were used as the training set for demonstrating neural network reconstruction capabilities. Each 8x8 section used for training the ANN was taken from a clinical 64x64 SPECT image slice. The planar inputs were generated by projecting each image slice into eight 8-quadrant planes. Each plane was rotated 22.5° from the previous plane around the image, giving eight incremental views covering 180° around the image. Each of the eight quadrants of each planar view is a summation of the intensity values projected from the 8x8 section.

Two sets of training data were used to demonstrate SPECT reconstruction. The first set consisted of a single datum. The objective of training a single image was to demonstrate the neural networks ability to memorize a SPECT image. The second data set consisted of two parallel sections separated by one slice, taken from a clinical SPECT image. The objective here was not only to show the ANN's ability to recall more than one image, but also to determine its ability to generalize the relationship between the planar and reconstructed data. This was achieved by testing the networks ability to reconstruct an image slice not used in the training set.

Multi-layer Neural Network on a Distributed Array of Processors

The MasPar MP-1 is a single instruction multiple data (SIMD), massively parallel machine. Composed of a 128x128 interconnected array of 4-bit processors, the performance of the MP-1 is dependant on how readily the problem at hand can be distributed among the 16,384 processors. The backpropagation architecture can utilize this parallel processing power, by executing the functions of all processing elements (PEs) in each layer simultaneously.

The architecture implemented for determining the feasibility of SPECT reconstruction by an ANN, required 64 inputs, for the eight planar projections, and 64 outputs, to produce the 8x8 reconstructed image slice. The optimal number of PEs in the hidden layer was determined by training a number of networks with various hidden layer sizes. The neural network was distributed on the processor array by dedicating a processor to each node and each interconnection in the network, figure 1. These processors are aligned on the array so as to best utilize the high speed north-south, east-west communication pathways. This distribution of the network allows data to propagate from layer to layer more efficiently than less direct pathways. The training time required for large architectures was significantly less on the MP-1 than on the VAXStation 3520 serial computer, to which serial and parallel training rate comparisons were made, Table I.

I_1	w_{111}	w_{112}	w_{113}	. . .	w_{11j}			
I_2	w_{121}	w_{122}	w_{123}	. . .	w_{12j}			
I_3	w_{131}	w_{132}	w_{133}	. . .	w_{13j}			
⋮	⋮	⋮	⋮	⋮	⋮			
I_{64}	H_1/w_1	H_2/w_2	H_3/w_3	. . .	H_j/w_j			
	w_{211}	w_{221}	w_{231}		w_{2j1}	O_1		
	⋮	⋮	⋮	⋮	⋮	⋮		
	w_{2163}	w_{2263}	w_{2363}	. . .	w_{2j63}	O_{63}		
	w_{2164}	w_{2264}	w_{2364}	. . .	w_{2j64}	O_{64}		

Figure 1 ANN Distribution on 128x128 PE Array.

Table I Serial Vs. Parallel Processing Rates

ANN Architecture 1000 iterations of a single datum training set (time)					
	64x8x64	64x16x64	64x24x64	64x32x64	64x40x64
VAXStation 3520 (serial)	62 s	135 s	176 s	225 s	277 s
MasPar MP-1 (parallel)	62 s	66 s	72 s	75 s	77 s

Single Image Memorization

A number of different neural network architectures were trained on a single SPECT image. Each network architecture trained to an RMS error of zero (single-precision), except for an architecture with only two hidden nodes. All other architectures reproduced the SPECT images exactly. This precise recall accuracy was achieved in less than 100 training iterations, except for 1-, 3-, 4-, and 5- hidden node architectures which required as many as 200 iterations to reach an RMS error of zero. Overall memorization of a single SPECT image was not a problem for the backpropagation neural networks.

ANN Generalization of Multiple SPECT images

The next task was to determine the neural networks ability to generalize the training set in order to accurately produce novel images. This was addressed by training an ANN on two 8x8 parallel image slice sections which were separated by one slice. The network was trained on these two images and achieved an RMS error of 0.001928. Then the untrained middle image slice was fed forward through the neural network. The output image generated had an RMS error of 0.001231. The full 64x64 reconstructed SPECT image from which the untrained 8x8 section was taken, is shown in figure 2. The actual SPECT image section and the ANN generated SPECT image section are shown in figures 3 and 4. The network that produced the image in figure 4, had a 64x8x64 architecture and was trained for 6000 iterations on the two image training set.

Discussion and Concluding Remarks

The neural networks ability to memorize a SPECT image and to generalize between two slice images to reconstruct the center, untrained image shows that full SPECT image reconstruction via an ANN is feasible. Although a statistical comparison of the reconstructed image to the original image was made, the most important measure of the ANN's image reconstruction ability is a visual one. This is particularly true of SPECT information collection which is through visual interpretation. It can be seen in figures 3 and 4 that the anatomical features of the original SPECT section are present in the image reconstructed by the ANN.

The next step in developing SPECT reconstruction with an ANN will be to train the network on multiple slices, every other one, from all the slices of a SPECT brain image. This will be important in determining if the ANN can generalize over a large range of images and accurately reproduce the untrained slices.

From that point the same approach taken for 8x8 image reconstruction will be applied to full 64x64 SPECT images. This will require a modification to the parallel implementation of the ANN. Although the processing times achieved with the MP-1 are encouraging, many high-speed serial machines could more efficiently train the architectures used in this paper. Superior training rates are attainable through better utilization of the PE array. An improved parallel backpropagation implementation algorithm has been proposed [5]. A new algorithm will be necessary for full SPECT image reconstruction, and for what may eventually be a more efficient SPECT reconstruction technique.

The primary constraint to accurate ECT image reconstruction involves the presence of noise, systematic errors, artifacts, and physical photon transport effects such as scatter and attenuation [2, 3]. Improved SPECT reconstruction via an ANN may be achieved by compensating for these problems in the training set. One way in which the accuracy of reconstruction can be improved is by modeling the physics of these problems in conjunction with the reconstruction algorithm used to produce the ANN training set. The Monte Carlo method can be used to simulate attenuation, scatter, and other effects so that the ANN can be trained to correct for these undesired artifacts [6, 7]. ANN SPECT reconstruction would then offer the advantages associated with the Monte Carlo method, but without the inherent computational cost.

References

[1] Z. Cho and J. Ra, "Methods and algorithms Toward 3-D Volume Image Reconstruction with Projections", Lecture Notes in Medical Informatics. New York : Springer-Verlag, pp. 1-39 : 1984.

[2] R. Jaszczak, "Tomographic Radiopharmaceutical Imaging", *Proceedings of the IEEE,* vol. 76(9), pp. 1079-1094, September 1988.

[3] T. Budinger, "Current Status and Limitations of Single Photon Emission Imaging", Diagnostic Imaging in Medicine. Boston : Martinus Nijhoff Publishers, pp. 278-298: 1983.

[4] R. Hecht-Nielsen, "Theory of the Backpropagation Neural Network", *International Joint Conference on Neural Networks (IJCNN),* Vol. 1, pp. 593-605, 1989.

[5] H. Yoon, J. Nang and S, Maeng, "A Distributed Backpropagation Algorithm of Neural Networks on Distributed-Memory Multiprocessors", *Proc. of the 3rd Symp. on the Frontiers of Massively Parallel Computation,* pp. 358-363, 1990.

[6] R. Gordon and G. Herman, "Three-dimensional Reconstruction from Projections: A Review of Algorithms", Int. Rev. of Cytol., vol. 38, pp. 111-151, 1974.

[7] L. Shepp and Y. Vardi, "Maximum Likelihood Reconstruction for Emission Tomography", I.E.E.E. Trans. Med. Imagaing, vol. MI-1, pp. 113-122, 1982.

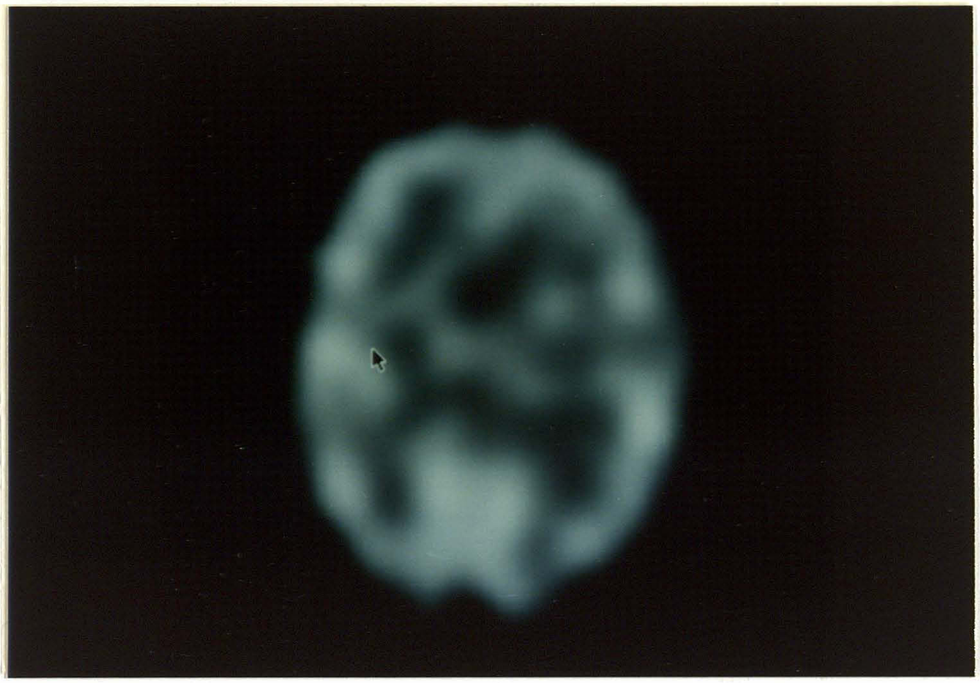

Figure 2 Reconstructed 64x64 SPECT image from which the 8x8 untrained section was taken (denoted by arrow).

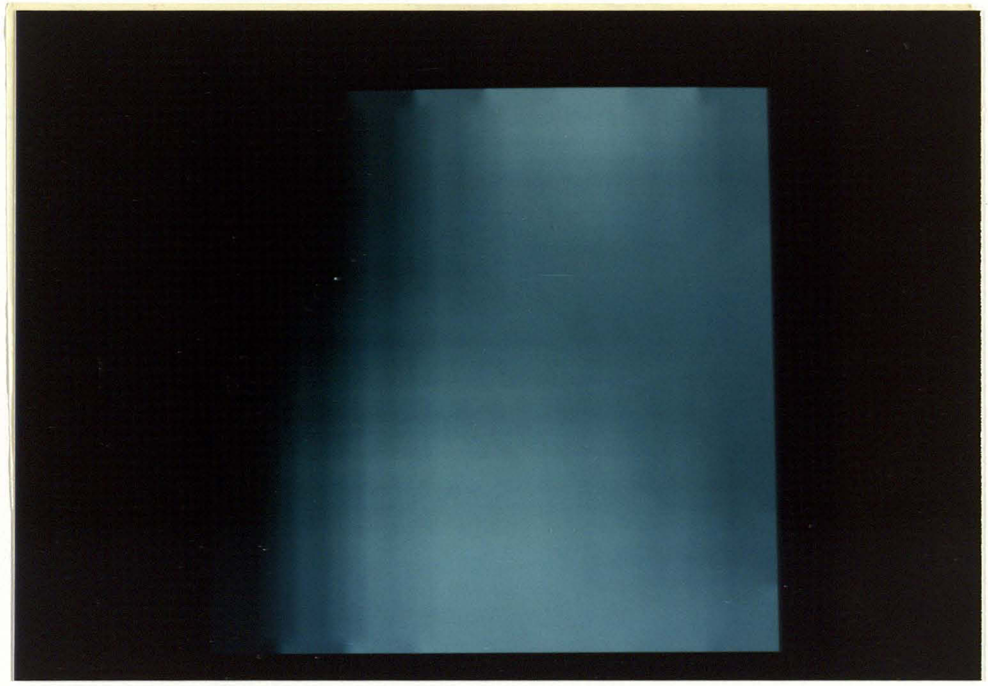

Figure 3 Conventionally reconstructed 8x8 SPECT section.

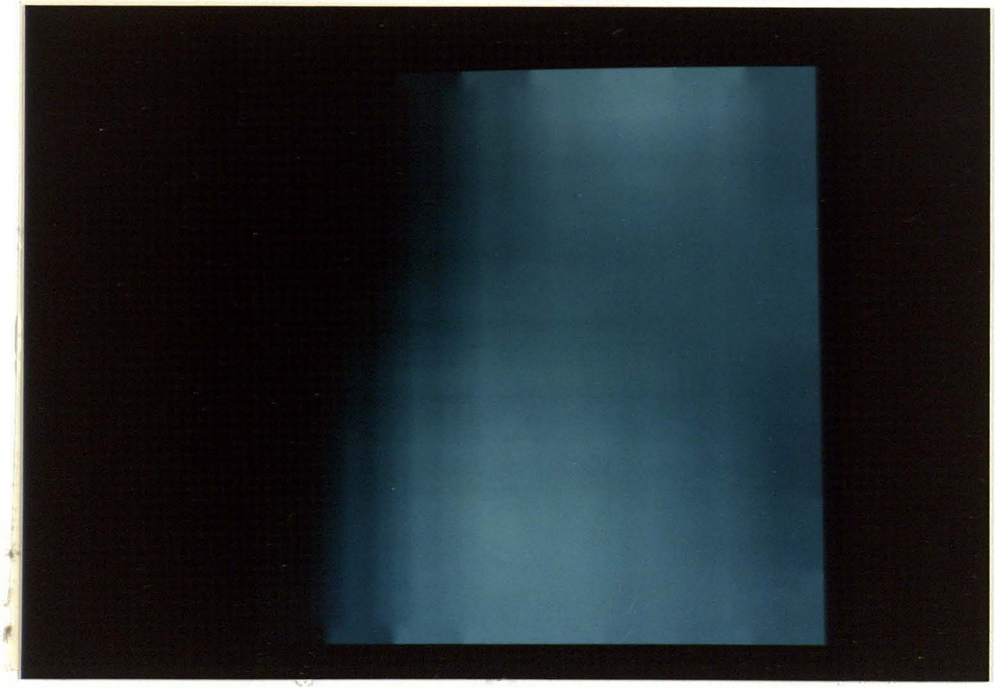

Figure 4 ANN reconstructed 8x8 SPECT section.

APPENDIX II

SPECT DATA CONVERSION AND MANIPULATION ROUTINES "C" CODE

....

This code, written in "C" language, was written to manipulate the data of a complete, conventionally reconstucted SPECT image file. These data manipulation routines are required to extract reconstructed slices form the image, extract 8x8 sections from a slice image, project the slices into eight 8-quadrant planes, normalize output data for network training, and convert 8x8 images into a displayable format. The rest of this page list the declarations section of the code.

....

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

#define NAME_SIZE 12
#define MAX_COL 64
#define MAX_ROW 64

void file_data();
void compress_data();
void retrieve();
void project_data();
void calc_data();
void convert_data();
void format_data();
void gen_data();

int x_num, y_num, proj, cmpress, num;
char inname[NAME_SIZE], outname[NAME_SIZE];
char sel;
float in_data[MAX_ROW][MAX_COL];
float id[MAX_ROW][MAX_COL];
FILE *ap;
```

....

The main program makes the primary function calls to the data manipulation routines. The routines selectable are listed under the **while{}**loop in the main program.

....

```
main()
{
        while (1){
                printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
                printf("   SPECT data handling procedures\n");
                printf("          1)  Retrieve record\n");
                printf("          2)  Compress file\n");
                printf("          3)  Project file into eight planes\n");
                printf("          4)  Convert binary file to ANN format\n");
                printf("          5)  Merge projection files\n");
                printf("          6)  Generate image formatted file\n");
                printf("          7)  Exit\n\n");
                printf(" Enter selection -> ");
                while ((sel = getchar()) = = '\n');
                switch(sel){
                        case '1' : retrieve();
                                   break;
                        case '2' : file_data();
                                   compress_data();
                                   break;
                        case '3' : project_data();
                                   break;
                        case '4' : convert_data();
                                   break;
                        case '5' : format_data();
                                   break;
                        case '6' : gen_data();
                                   break;
                        default :  exit(0);
                }
        }
}
```

....

The function **file_data()** reads in SPECT file information and the desired section size to be extracted.

....

```
void file_data()
{
        int i, j;
        FILE *fp;

        printf(" Input data filename : ");
        scanf("%s", inname);
        if ((fp = fopen(inname, "r")) = = NULL){
                printf("        Input file not found\n");
                exit(0);
        }
        printf(" Enter number of data points in each row -> ");
        scanf("%d", &x_num);
        printf(" Enter number of data points in each column -> ");
        scanf("%d", &y_num);
        printf(" Enter the compression factor -> ");
        scanf("%d", &cmpress);
        for (i = 0; i < y_num; + +i)
                for (j = 0; j < x_num; + +j)
                        fscanf(fp, "%f ", &in_data[i][j]);
}
```

....

The procedure **compress_data()** compresses 64x64 slices into a user selectable size.

....

```
void compress_data()
{
        int i, j, k, l;
        float tmp_data[MAX_ROW][MAX_COL];
        char cpname[NAME_SIZE];
        FILE *cp;

        x_num = x_num/cmpress;
        y_num = y_num/cmpress;
        printf(" Enter compressed image filename : ");
        scanf("%s", cpname);
        cp = fopen(cpname, "w");
        for (i = 0; i < y_num; ++i)
                for (j = 0; j < x_num; ++j){
                        for (k = 1; k < cmpress+1; ++k)
                                for (l = 1; l < cmpress+1; ++l)
                                        tmp_data[i][j] +=
                                        in_data[i*cmpress+k-1][j*cmpress+l-1];
                        in_data[i][j] = tmp_data[i][j]/(cmpress*cmpress);
                        fprintf(cp, "%9.6f ", in_data[i][j]);
                }
}
```

....

The procedure **retrieve()** reads a 64 x 64 record from the SPECT file and stores that in a separate file so further data manipulation can be done without destroying the original SPECT file.

....

```
void retrieve()
{
        int i, j, k, l, rec, tpp;
        char dt[MAX_COL], dy[8];
        char recname[NAME_SIZE], opname[NAME_SIZE];
        short dat[64];
        FILE *op;
        FILE *rp;

        printf(" Enter filename in which record is located : ");
        scanf("%s", opname);
        if ((op = fopen(opname, "r")) == NULL){
                printf("      Input file not found\n");
                exit(0);
        }
        printf(" Enter filename to store record : ");
        scanf("%s", recname);
        rp = fopen(recname, "w");
        printf(" Enter the number of the record -> ");
        scanf("%d", &rec);
        for (i = 0; i < rec-1; ++i)
                for (j = 0; j < 64; ++j)
                        fread(dat,2,64,op);
        printf(" Enter '1' for an 8x8 image, '2' otherwise -> ");
        scanf("%d", &rec);
        for (j = 0; j < 64; ++j){
                fread(dat,2,64,op);
                if (rec == 1){
                        if ((j > 23) && (j < 32)){
                                for (l = 16; l < 24; ++l){
                                        if (dat[l] < 0)
                                                dat[l] = 0;
                        tpp = dat[l];
                        dy[l-16] = tpp;
                    }
                    fwrite(dy,1,8,rp);
                }
            }
            else{
                for (k = 0; k < 64; ++k){
                    if (dat[k] < 0)
                        dat[k] = 0;
                    tpp = dat[k];
                    dt[k] = tpp;
                }
                fwrite(dt,1,64,rp);
            }
        }
        fclose(recname);
    }
```

....

The procedure **project_data()** projects an 8x8 image section into eight 8-quadrant planes 180° about the 8x8 section.

....

```
void project_data()
{
        int i, j, numb;
        float tmpin[8][8];
        char projname[NAME_SIZE], netname[NAME_SIZE];
        FILE *pf;

        numb = 1;
        printf(" Enter the 8x8 input filename : ");
        scanf("%s", projname);
        if ((pf = fopen(projname,"r")) = = NULL){
                printf("      Input file not found\n");
                exit(0);
        }
        printf(" Enter network formatted projection filename : ");
        scanf("%s", netname);
        ap = fopen(netname,"w");
        for (i = 0; i < 8; + +i)
                for (j = 0; j < 8; + +j){
                        fscanf(pf, "%f\t", &id[i][j]);
                        id[i][j] = ((id[i][j] / 128.0) * 0.80) + 0.10;
                }
        fprintf(ap, "%d\n", numb);
        calc_data();
        for (i = 0; i < 8; + +i)
                for (j = 0; j < 8; + +j)
                        tmpin[i][j] = id[j][7-i];
        for (i = 0; i < 8; + +i)
                for (j = 0; j < 8; + +j)
                        id[i][j] = tmpin[i][j];
        calc_data();
        for (i = 0; i < 8; + +i)
                for (j = 0; j < 8; + +j)
                fprintf(ap, "%9.6f\t", id[7-j][i]);
        fprintf(ap, "\n");
        fclose(netname);
}
```

....

The procedure **calc_data()** is a routine called bt **project_data()** to calculate four 8-quadrant planar projections from a 8x8 image. The procedure **project_data()** then rotates the section 90° and calls **calc_data()** again to calculate the other four planar projections.

....

```
void calc_data()
{
        int i, j;
        float prt[8];

        /*              Projection No. 0                */
        for (i = 0; i < 8; ++i){
                prt[i] = 0.0;
                for (j = 0; j < 8; ++j)
                        prt[i] += id[j][i];
        }
        for (i = 0; i < 8; ++i)
                fprintf(ap, "%9.6f\t", prt[i]);
        /*              Projection No. 1                */
        prt[0] = 0.23445*id[0][2] + 0.83975*id[0][1] + 0.00819*id[0][0]
                + 0.78835*id[1][1] + 0.29404*id[1][0] + 0.37415*id[2][1]
                + 0.70824*id[2][0] + 0.10789*id[3][1] + 0.92623*id[3][0]
                + 0.54572*id[4][0] + 0.13900*id[5][0];
        prt[1] = 0.31684*id[0][3] + 0.76555*id[0][2] + 0.00731*id[1][3]
                + 0.86343*id[1][2] + 0.21165*id[1][1] + 0.45654*id[2][2]
                + 0.62585*id[2][1] + 0.06691*id[3][2] + 0.94171*id[3][1]
                + 0.07377*id[3][0] + 0.62811*id[4][1] + 0.45428*id[4][0]
                + 0.14111*id[5][1] + 0.85140*id[5][0] + 0.79962*id[6][0]
                + 0.38547*id[7][0];
        prt[2] = 0.39923*id[0][4] + 0.68316*id[0][3] + 0.03099*id[1][4]
                + 0.91483*id[1][3] + 0.13657*id[1][2] + 0.53893*id[2][3]
                + 0.54346*id[2][2] + 0.12472*id[3][3] + 0.92490*id[3][2]
                + 0.03279*id[3][1] + 0.71050*id[4][2] + 0.37189*id[4][1]
                + 0.21390*id[5][2] + 0.85889*id[5][1] + 0.00960*id[5][0]
                + 0.00960*id[6][2] + 0.87241*id[6][1] + 0.20038*id[6][0]
                + 0.46786*id[7][1] + 0.61453*id[7][0];
```

```
prt[3] = 0.48162*id[0][5] + 0.60077*id[0][4] + 0.07107*id[1][5]
        + 0.93346*id[1][4] + 0.07786*id[1][3] + 0.62132*id[2][4]
        + 0.46107*id[2][3] + 0.20711*id[3][4] + 0.87528*id[3][3]
        + 0.00819*id[3][2] + 0.87528*id[4][3] + 0.28950*id[4][2]
        + 0.29629*id[5][3] + 0.78610*id[5][2] + 0.03553*id[6][3]
        + 0.91933*id[6][2] + 0.12753*id[6][1] + 0.55025*id[7][2]
        + 0.53214*id[7][1];
prt[7] = 0.23445*id[7][5] + 0.83975*id[7][6] + 0.00819*id[7][7]
        + 0.78835*id[6][6] + 0.29404*id[6][7] + 0.37415*id[5][6]
        + 0.70824*id[5][7] + 0.10789*id[4][6] + 0.92623*id[4][7]
        + 0.54572*id[3][7] + 0.13900*id[2][7];
prt[6] = 0.31684*id[7][4] + 0.76555*id[7][5] + 0.00731*id[6][4]
        + 0.86343*id[6][5] + 0.21165*id[6][6] + 0.45654*id[5][5]
        + 0.62585*id[5][6] + 0.06691*id[4][5] + 0.94171*id[4][6]
        + 0.07377*id[4][7] + 0.62811*id[3][6] + 0.45428*id[3][7]
        + 0.14111*id[2][6] + 0.85140*id[2][7] + 0.79962*id[1][7]
        + 0.38547*id[0][7];
prt[5] = 0.39923*id[7][3] + 0.68316*id[7][4] + 0.03099*id[6][3]
        + 0.91483*id[6][4] + 0.13657*id[6][5] + 0.53893*id[5][4]
        + 0.54346*id[5][5] + 0.12472*id[4][4] + 0.92490*id[4][5]
        + 0.03279*id[4][6] + 0.71050*id[3][5] + 0.37189*id[3][6]
        + 0.21390*id[2][5] + 0.85889*id[2][6] + 0.00960*id[2][7]
        + 0.00960*id[1][5] + 0.87241*id[1][6] + 0.20038*id[1][7]
        + 0.46786*id[0][6] + 0.61453*id[0][7];
prt[4] = 0.48162*id[7][2] + 0.60077*id[7][3] + 0.07107*id[6][2]
        + 0.93346*id[6][3] + 0.07786*id[6][4] + 0.62132*id[5][3]
        + 0.46107*id[5][4] + 0.20711*id[4][3] + 0.87528*id[4][4]
        + 0.00819*id[4][5] + 0.87528*id[4][4] + 0.28950*id[3][5]
        + 0.29629*id[2][4] + 0.78610*id[2][5] + 0.03553*id[1][4]
        + 0.91933*id[1][5] + 0.12753*id[1][6] + 0.55025*id[0][5]
        + 0.53214*id[0][6];
for (i = 0; i < 8; ++i)
        fprintf(ap, "%9.6f\t", prt[i]);
/*                  Projection No. 2                    */
prt[0] = 0.91169*id[0][2] + 0.28680*id[0][3] + 0.21574*id[0][1]
        + 0.28680*id[1][2] + 0.91169*id[1][1] + 0.28680*id[2][1]
        + 0.21574*id[1][0] + 0.91169*id[2][0] + 0.28680*id[3][0];
prt[1] = 0.01472*id[0][5] + 0.65685*id[0][4] + 0.71320*id[0][3]
```

```
          + 0.01472*id[1][4] + 0.65685*id[1][3] + 0.71320*id[1][2]
          + 0.01472*id[2][3] + 0.65685*id[2][2] + 0.02944*id[0][2]
          + 0.02944*id[1][1] + 0.71320*id[2][1] + 0.01472*id[3][2]
          + 0.02944*id[2][0] + 0.65685*id[3][1] + 0.71320*id[3][0]
          + 0.01472*id[4][1] + 0.65685*id[4][0] + 0.01472*id[5][0];
prt[2] = 0.17157*id[0][6] + 0.89949*id[0][5] + 0.34315*id[0][4]
          + 0.17157*id[1][5] + 0.89949*id[1][4] + 0.34315*id[1][3]
          + 0.17157*id[2][4] + 0.89949*id[2][3] + 0.34315*id[2][2]
          + 0.17157*id[3][3] + 0.89949*id[3][2] + 0.34315*id[3][1]
          + 0.17157*id[4][2] + 0.89949*id[4][1] + 0.34315*id[4][0]
          + 0.17157*id[5][1] + 0.89949*id[5][0] + 0.17157*id[6][0];
prt[3] = 0.5*id[0][7] + 0.82843*id[0][6] + 0.08579*id[0][5]
          + 0.5*id[1][6] + 0.82843*id[1][5] + 0.08579*id[1][4]
          + 0.5*id[2][5] + 0.82843*id[2][4] + 0.08579*id[2][3]
          + 0.5*id[3][4] + 0.82843*id[3][3] + 0.08579*id[3][2]
          + 0.5*id[4][3] + 0.82843*id[4][2] + 0.08579*id[4][1]
          + 0.5*id[5][2] + 0.82843*id[5][1] + 0.08579*id[5][0]
          + 0.5*id[6][1] + 0.82843*id[6][0] + 0.5*id[7][0];
prt[7] = 0.91169*id[7][5] + 0.28680*id[7][4] + 0.21574*id[7][6]
          + 0.28680*id[6][5] + 0.91169*id[6][6] + 0.28680*id[5][6]
          + 0.21574*id[6][7] + 0.91169*id[5][7] + 0.28680*id[4][7];
prt[6] = 0.01472*id[7][2] + 0.65685*id[7][3] + 0.71320*id[7][4]
          + 0.01472*id[6][3] + 0.65685*id[6][4] + 0.71320*id[6][5]
          + 0.01472*id[5][4] + 0.65685*id[5][5] + 0.02944*id[7][5]
          + 0.02944*id[6][6] + 0.71320*id[5][6] + 0.01472*id[4][5]
          + 0.02944*id[5][7] + 0.65685*id[4][6] + 0.71320*id[4][7]
          + 0.01472*id[3][6] + 0.65685*id[3][7] + 0.01472*id[2][7];
prt[5] = 0.17157*id[7][1] + 0.89949*id[7][2] + 0.34315*id[7][3]
          + 0.17157*id[6][2] + 0.89949*id[6][3] + 0.34315*id[6][4]
          + 0.17157*id[5][3] + 0.89949*id[5][4] + 0.34315*id[5][5]
          + 0.17157*id[4][4] + 0.89949*id[4][5] + 0.34315*id[4][6]
          + 0.17157*id[3][5] + 0.89949*id[3][6] + 0.34315*id[3][7]
          + 0.17157*id[2][6] + 0.89949*id[2][7] + 0.17157*id[1][7];
prt[4] = 0.5*id[7][0] + 0.82843*id[7][1] + 0.08579*id[7][2]
          + 0.5*id[6][1] + 0.82843*id[6][2] + 0.08579*id[6][3]
          + 0.5*id[5][2] + 0.82843*id[5][3] + 0.08579*id[5][4]
          + 0.5*id[4][3] + 0.82843*id[4][4] + 0.08579*id[4][5]
          + 0.5*id[3][4] + 0.82843*id[3][5] + 0.08579*id[3][6]
```

```
        + 0.5*id[2][5] + 0.82843*id[2][6] + 0.08579*id[2][7]
        + 0.5*id[1][6] + 0.82843*id[1][7] + 0.5*id[0][7];
for (i = 0; i < 8; ++i)
        fprintf(ap, "%9.6f\t", prt[i]);
/*              Projection No. 3              */
prt[0] = 0.23445*id[2][0] + 0.83975*id[1][0] + 0.00819*id[0][0]
        + 0.78835*id[1][1] + 0.29404*id[0][1] + 0.37415*id[1][2]
        + 0.70824*id[0][2] + 0.10789*id[1][3] + 0.92623*id[0][3]
        + 0.54572*id[0][4] + 0.13900*id[0][5];
prt[1] = 0.31684*id[3][0] + 0.76555*id[2][0] + 0.00731*id[3][1]
        + 0.86343*id[2][1] + 0.21165*id[1][1] + 0.45654*id[2][2]
        + 0.62585*id[1][2] + 0.06691*id[2][3] + 0.94171*id[1][3]
        + 0.07377*id[0][3] + 0.62811*id[1][4] + 0.45428*id[0][4]
        + 0.14111*id[1][5] + 0.85140*id[0][5] + 0.79962*id[0][6]
        + 0.38547*id[0][7];
prt[2] = 0.39923*id[4][0] + 0.68316*id[3][0] + 0.03099*id[4][1]
        + 0.91483*id[3][1] + 0.13657*id[2][1] + 0.53893*id[3][2]
        + 0.54346*id[2][2] + 0.12472*id[3][3] + 0.92490*id[2][3]
        + 0.03279*id[1][3] + 0.71050*id[2][4] + 0.37189*id[1][4]
        + 0.21390*id[2][5] + 0.85889*id[1][5] + 0.00960*id[0][5]
        + 0.00960*id[2][6] + 0.87241*id[1][6] + 0.20038*id[0][6]
        + 0.46786*id[1][7] + 0.61453*id[0][7];
prt[3] = 0.48162*id[5][0] + 0.60077*id[4][0] + 0.07107*id[5][1]
        + 0.93346*id[4][1] + 0.07786*id[3][1] + 0.62132*id[4][2]
        + 0.46107*id[3][2] + 0.20711*id[4][3] + 0.87528*id[3][3]
        + 0.00819*id[2][3] + 0.87528*id[3][4] + 0.28950*id[2][4]
        + 0.29629*id[3][5] + 0.78610*id[2][5] + 0.03553*id[3][6]
        + 0.91933*id[2][6] + 0.12753*id[1][6] + 0.55025*id[2][7]
        + 0.53214*id[1][7];
prt[7] = 0.23445*id[5][7] + 0.83975*id[6][7] + 0.00819*id[7][7]
        + 0.78835*id[6][6] + 0.29404*id[7][6] + 0.37415*id[6][5]
        + 0.70824*id[7][5] + 0.10789*id[6][4] + 0.92623*id[7][4]
        + 0.54572*id[7][3] + 0.13900*id[7][2];
prt[6] = 0.31684*id[4][7] + 0.76555*id[5][7] + 0.00731*id[4][6]
        + 0.86343*id[5][6] + 0.21165*id[6][6] + 0.45654*id[5][5]
        + 0.62585*id[6][5] + 0.06691*id[5][4] + 0.94171*id[6][4]
        + 0.07377*id[7][4] + 0.62811*id[6][3] + 0.45428*id[7][3]
        + 0.14111*id[6][2] + 0.85140*id[7][2] + 0.79962*id[7][1]
```

```
                + 0.38547*id[7][0];
        prt[5] = 0.39923*id[3][7] + 0.68316*id[4][7] + 0.03099*id[3][6]
                + 0.91483*id[4][6] + 0.13657*id[5][6] + 0.53893*id[4][5]
                + 0.54346*id[5][5] + 0.12472*id[4][4] + 0.92490*id[5][4]
                + 0.03279*id[6][4] + 0.71050*id[5][3] + 0.37189*id[6][3]
                + 0.21390*id[5][2] + 0.85889*id[6][2] + 0.00960*id[7][2]
                + 0.00960*id[5][1] + 0.87241*id[6][1] + 0.20038*id[7][1]
                + 0.46786*id[6][0] + 0.61453*id[7][0];
        prt[4] = 0.48162*id[2][7] + 0.60077*id[3][7] + 0.07107*id[2][6]
                + 0.93346*id[3][6] + 0.07786*id[4][6] + 0.62132*id[3][5]
                + 0.46107*id[4][5] + 0.20711*id[3][4] + 0.87528*id[4][4]
                + 0.00819*id[5][4] + 0.87528*id[4][4] + 0.28950*id[5][3]
                + 0.29629*id[4][2] + 0.78610*id[5][2] + 0.03553*id[4][1]
                + 0.91933*id[5][1] + 0.12753*id[6][1] + 0.55025*id[5][0]
                + 0.53214*id[6][0];
        for (i = 0; i < 8; ++i)
                fprintf(ap, "%9.6f\t", prt[i]);
}
```

....

The procedure **convert_data()** converts a binary file generated from the original SPECT data file to the ANN output format.

....

```
void convert_data()
{
        int i;
        float bnum;
        char bdat[64];
        char binname[NAME_SIZE], anname[NAME_SIZE];
        FILE *bp;
        FILE *np;

        printf(" Enter binary filename : ");
        scanf("%s", binname);
        if ((bp = fopen(binname, "r")) == NULL){
                printf("        Binary file not found\n");
                exit(0);
        }
        printf(" Enter ANN output filename : ");
        scanf("%s", anname);
        np = fopen(anname, "w");
        fread(bdat,1,64,bp);
        for (i = 0; i < 64; ++i){
                bnum = bdat[i] * 1.0;
                fprintf(np, "%f\t", bnum);

        }
        fclose(binname);
}
```

....

The procedure **format_data()** takes ASCII files generated in the ANN format and merges them to create the ANN training data file.

....

```
void format_data()
{
        int i, innt;
        float fdat;
        char bin[64];
        char anname[NAME_SIZE], binname[NAME_SIZE];
        FILE *np;
        FILE *bp;

        printf(" Enter input projection filename : ");
        scanf("%s", anname);
        if ((np = fopen(anname, "r")) == NULL){
                printf("        Input file not found\n");
                exit(0);
        }
        printf(" Enter merged to image output filename : ");
        scanf("%s", binname);
        if ((bp = fopen(binname, "a")) == NULL){
                printf("        Merge file not found\n");
                exit(0);
        }
        fscanf(np, "%d\n", innt);
        for (i = 0; i < 128; ++i){
                fscanf(np, "%f\t", &fdat);
                fprintf(bp, "%f\t", fdat);
                }
        fprintf(bp, "\n");
}
```

....

The procedure **gen_data()** converts an ANN format to single byte binary format file which can then be displayed on a screen via xwindows.

....

```
void gen_data()
{
        int i, inum;
        float fdat;
        char bin[64];
        char anname[NAME_SIZE], binname[NAME_SIZE];
        FILE *np;
        FILE *bp;

        printf(" Enter ANN filename : ");
        scanf("%s", anname);
        if ((np = fopen(anname, "r")) == NULL){
                printf("        Input file not found\n");
                exit(0);
        }
        printf(" Enter image formatted output filename : ");
        scanf("%s", binname);
        bp = fopen(binname, "w");
        for (i = 0; i < 64; ++i){
                fscanf(np, "%f\t", &fdat);
                inum = ((((fdat - 0.1)/0.8)*128) + 0.5)*2;
                bin[i] = inum;
                printf("%4d", inum);
        }
        fwrite(bin,1,64,bp);
}
```

APPENDIX III

SERIAL BACKPROPAGATION ANN SIMULATION "C" CODE

....

This code, written in "C" language, simulates a three-layer (input, hidden, output) backpropagation neural network. The size of the network is limited by the constant MAX_INPUT. A network with a layer containing more nodes than MAX_INPUT cannot simulated with this code. The rest of this and the next page list the declarations section of the code. The training set for a network is read into matrix variable **in_data[][]**. Variable **wt_val[][][]** handles the current interconnection weight values, and **tmp_wt[][][]** stores previous weight values during backpropagation updating of the interconnection weights. **net[][]** stores the most recent ouputs from all the nodes in the network.

....

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

#define MAX_LAYERS 5
#define MAX_INPUT 200
#define MAX_DATA 200
#define NAME_SIZE 12

void getline();
int net_data();
void weight_set();
void fwd_prop();
void bck_prop();
float error();
void prog_int();
void save_net();
void recall_wgts();
int test_net();

int i, cnt, cum, innum, dat_num, layers;
int lyr[MAX_LAYERS];
char name[NAME_SIZE], wgtfile[NAME_SIZE];
float in_data[MAX_DATA][MAX_INPUT];
float wt_val[MAX_LAYERS][MAX_INPUT][MAX_INPUT];
float tmp_wt[MAX_LAYERS][MAX_INPUT][MAX_INPUT];
float net[MAX_INPUT][MAX_INPUT];
float d[MAX_INPUT];
float errsum, errval, errmin;
double sqrt();
double exp();
double fmod();
FILE *np;
```

....

The main program makes all the primary function calls. The **while{}** loop contains the backpropagation training calls. The loop is externally interrupted when the variable **cnt**, which stores the number of training iterations, reaches the number of iterations requested by the user.

....

```
main()
{
        errmin = 1.0;
        errval = 0.0;
        cum = 1;
        cnt = 0;
        getline();
        if (net_data()){
                printf("\n  Enter weight filename -> ");
                scanf("%s", wgtfile);
                if ((np = fopen(wgtfile, "r")) = = NULL)
                        weight_set();
                  else
                        recall_wgts();
                while (1){
                        errsum = 0.0;
                        for (innum = 0; innum < dat_num; + +innum){
                                fwd_prop();
                                bck_prop();
                                for (i = 0; i < lyr[layers-1]; + +i)
                                        errsum + = d[i] * d[i];
                        }
                        if ((error() < errval) | | (cnt = = 0))
                                prog_int();
                }
        }
}
```

....

The function **getline()** reads in the ANN data filename from keyboard and stores it in character array **name[]**.

....

```
void getline()
{
        int c, i;

        printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
        printf("      BACK-PROPAGATION ARTIFICIAL NEURAL NETWORK\n");
        printf("\n                    Input data filename : ");
        scanf("%s", name);
}
```

....

The function **net_data()** reads in the network training set size and training set data from file stored in **name[]**. It also reads in the network size from the keyboard.

....

```
int net_data()
{
        FILE *fp;

        int c, i, j, tst;

        if ((fp = fopen(name, "r")) = = NULL){
                printf("Input File Not Found\n\f");
                return 0;
        }
        printf("\n\nEnter the number of layers in the neural network -> ");
        scanf("%d", &layers);
        for (i = 0; i < layers; + +i){
                printf("    Input the number of nodes in layer%3d  ", i+1);
                scanf("%d", &lyr[i]);
        }
        fscanf(fp,"%3d\n", &dat_num);
        for (i = 0; i < dat_num; + +i){
                for (j = 0; j < (lyr[0] + lyr[layers-1]); + +j)
                        fscanf(fp,"%f\t", &in_data[i][j]);
                fscanf(fp, "\n");
        }
        return 1;
}
```

....

The function **weight_set()** generates random weight values and stores them in the floating point array **wt_val[][][]**.

....

```
void weight_set()
{
        float wgt;
        int i, j, k;
        double seed;

        seed = 37591.0;
        for (i=0; i<layers-1; ++i)
                for (j = 0; j < lyr[i+1]; ++j)
                        for (k = 0; k < lyr[i]; ++k){
                                seed = fmod((seed*7141+54773), 259200.0);
                                wt_val[i][k][j] = seed/259200.0 - 0.5;
                        }
}
```

....

The function fwd_prop() feeds the training data forward through the network by summing interconnection values and calculating node outputs via the sigmoidal activation function.

....

```
void fwd_prop()
{
        int i, j, k, l, m;
        float tmp_net[MAX_INPUT][MAX_INPUT];

        for (j = 0; j < lyr[0]; ++j){
                net[0][j] = in_data[innum][j];
                if (cnt > 100){
                        printf("%6.3f  ", net[0][j]);
                }
        }
        for (k = 1; k < layers; ++k)
                for (l = 0; l < lyr[k]; ++l){
                        tmp_net[k][l] = 0.0;
                        for (m = 0; m < lyr[k-1]; ++m){
                                tmp_net[k][l] += net[k-1][m]*wt_val[k-1][m][l];
                        }
                        net[k][l] = 1.0 / (1.0 + exp(-tmp_net[k][l]));
                        if ((cnt > 100) && (k == layers-1)){
                                printf("%9.6f,%9.6f\n", net[k][l], tmp_net[k][l]);
                        }
                }
}
```

....

The function **bck_prop()** backpropagates the RMS error, or "cost", of the training set through the network adjusting the interconnection weight values per the Delta Rule and the Generalized Delta Rule.

....

```
void bck_prop()
{
        int i, j, k;
        float sum, adj;

        for (j = 0; j < lyr[layers-1]; ++j){
                d[j] = in_data[innum][j+lyr[0]] - net[layers-1][j];
                for (k = 0; k < lyr[layers-2]; ++k){
                        tmp_wt[layers-2][k][j] = wt_val[layers-2][k][j];
                        wt_val[layers-2][k][j]
                                += d[j] * net[layers-2][k] * 0.3;
                }
        }
        for (i = layers-2; i > 0; --i)
                for (j = 0; j < lyr[i]; ++j){
                        sum = 0.0;
                        for (k = 0; k < lyr[layers-1]; ++k)
                                sum += d[k] * tmp_wt[i][j][k];
                        adj = sum * net[i][j] * (1 - net[i][j]);
                        for (k = 0; k < lyr[i-1]; ++k){
                                tmp_wt[i-1][k][j] = wt_val[i-1][k][j];
                                wt_val[i-1][k][j] += adj * net[i-1][k] * 0.3;
                        }
                }
}
```

```
..............................................................................................................................
        The function error() calculates the "cost" of the most recent pass of the training
        set through the network, and returns the floating point error value back to the
        calling routine.
..............................................................................................................................

float error()
{
        float tot;

        cnt += 1;
        tot = (1.0/(dat_num*lyr[layers-1]))*sqrt(errsum);
        if (tot < errmin)
                errmin = tot;
        if (cnt > 101){
                printf("%5d,%9.6f,%9.6f\n", cum, tot, errmin);
                cnt = 0;
                cum += 1;
        }
        errsum = 0.0;
        return tot;
}
```

....

The function **prog_int()** is called when training is halted or when the program is first run so as to determine if training is to continue, weight values saved, untrained data tested, or the program to be halted.

....

```
void prog_int()
{
        char j;

        printf("\n      1)  Save & quit\n");
        printf("        2)  Quit\n");
        printf("        3)  Continue training network\n");
        printf("        4)  Run untrained data through network\n");
        printf("  Enter selection -> ");
        while ((j = getchar()) = = '\n');
        switch(j){
                case '1' : save_net();
                case '3' : errval = errval/10.0;
                           printf ("\n  Continuing %6.5f\n", errval);
                           break;
                case '4' : if (!test_net())
                              exit(0);
                default  : exit(0);
        }
}
```

....

The function **save_net()** saves the interconnection weights of the network to the file in character variable **wgtfile.**

....

```
void save_net()
{
	FILE *fp;
	int i, j, k;

	fp = fopen(wgtfile, "w");
	fprintf(fp, "%3d\t", layers);
	for (i = 0; i < layers; + + i)
		fprintf(fp, "%3d\t", lyr[i]);
	fprintf(fp, "\n");
	for (i =0; i < layers-1; + +i)
		for (j = 0; j < lyr[i+1]; + +j){
			for (k = 0; k < lyr[i]; + +k)
				fprintf(fp, "%9.6f\t", wt_val[i][k][j]);
			fprintf(fp, "\n");
		}
	printf("\n   Weights saved in file %s", wgtfile);
	printf("\n");
	exit(0);
}
```

....

The function **recall_wgts()** retrieves interconnection weights of a previuosly trained network from filename **wgtfile** if it exists in the directory.

....

```
void recall_wgts()
{
        int i, j, k, tmp_layers;
        int tmp_lyr[MAX_LAYERS];

        fscanf(np, "%d\t", &tmp_layers);
        for (i = 0; i < tmp_layers; ++i){
                fscanf(np, "%d\t", &tmp_lyr[i]);
                printf("%6d\n", tmp_lyr[i]);}
        fscanf(np, "\n");
        for (i = 0; i < tmp_layers-1; ++i)
                for (j = 0; j < tmp_lyr[i+1]; ++j){
                        for (k = 0; k < tmp_lyr[i]; ++k)
                                fscanf(np, "%f\t", &wt_val[i][k][j]);
                        fscanf(np, "\n");
                }
}
```

....

The function **test_net()** runs untrained data through the neural network to determine it's ability to produce accurate answers for the untrained inputs. The untrained inputs and the correct outputs are stored in filename in character variable **netfile.**

....

```
int test_net()
{
        int i, j, numb;
        float costsum;
        char netfile[NAME_SIZE];
        FILE *fp;

        printf(" Enter file name of untrained data -> ");
        scanf("%s", netfile);
        if ((fp = fopen(netfile, "r")) == NULL){
                printf("\n  Input file Not Found\n");
                return 0;
        }
        fscanf(fp, "%d\n", &numb);
        for (i = 0; i < numb; ++i){
                for (j = 0; j < (lyr[0] + lyr[layers-1]); ++j)
                        fscanf(fp, "%f\t", &in_data[i][j]);
                fscanf(fp, "\n");
        }
        for (innum = 0; innum < numb; ++innum){
                cnt = 101;
                costsum = 0.0;
                fwd_prop();
                for (j = 0; j < lyr[layers-1]; ++j)
                        costsum +=
                           in_data[innum][j+lyr[0]] - net[layers-1][j];
                printf("  Difference = %9.6f\n", costsum);
        }
        return 1;
}
```

APPENDIX IV

PARALLEL BACKPROPAGATION ANN SIMULATION "MPL" CODE

....

This code, written in "MPL" language, simulates a three-layer (input, hidden, output) backpropagation neural network. The size of the network is limited by the constant MAX_INPUT, which is also the limiting size of the PU array. A network with a layer containing more nodes than MAX_INPUT cannot simulated with this code. The rest of this and the next page list the declarations section of the code. The training set for a network is read into singular variable **in_data[][]**. Plural variable **wt_val** handles the current interconnection weight values, and **tmp_wt** stores previous weight values during backpropagation updating of the interconnection weights. Plural variable **net** stores the most recent ouputs from all the nodes in the network.

....

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

#define MAX_LAYERS 5
#define MAX_INPUT 128
#define MAX_DATA 150
#define NAME_SIZE 12

void getline();
int net_data();
void weight_set();
void fwd_prop();
void bck_prop();
float error();
void prog_int();
void save_net();
void recall_wgts();
int test_net();

int i, cnt, cum, innum, dat_num, layers, iter;
int lyr[MAX_LAYERS];
char name[NAME_SIZE], wgtfile[NAME_SIZE];
float in_data[MAX_DATA][MAX_INPUT];
```

```
plural float wt_val;
plural float tmp_wt;
plural float net, neti;
plural float tmp_net;
plural float diff;
float ert, errsum, errval, errmin;
double sqrt();
double fmod();
FILE *np;
```

....

The main program makes all the primary function calls. The **while{}** loop contains the backpropagation training calls. The loop is externally interrupted when the variable **cnt**, which stores the number of training iterations, reaches the number of iterations requested by the user.

....

```
main()
{
        errmin = 1.0;
        errval = 0.0;
        cum = 1;
        cnt = -1;
        iter = 1000;
        getline();
        if (net_data()){
                printf("\n  Enter weight filename -> ");
                scanf("%s", wgtfile);
                if ((np = fopen(wgtfile, "r")) = = NULL)
                        weight_set();
                  else
                        recall_wgts();
                while (1){
                        errsum = 0.0;
                        if (cnt = = -1){
                                prog_int();
                                printf(" Enter number of iterations -> ");
                                scanf("%d", &iter);
                                cnt + = 1;
                        }
                        for (innum = 0; innum < dat_num; + +innum){
                                fwd_prop();
                                bck_prop();
                                for (i = 0; i < lyr[layers-1]; + +i){
                                        ert = proc[lyr[0]+i][lyr[1]+1].diff;
                                        errsum + = ert * ert;
                                }
                        }
                        if ((error() < errval) || (cnt == 0)){
                                prog_int();
                                printf(" Enter number of iterations -> ");
                                scanf("%d", &iter);
                        }
                }
        }
}
```

....

The function **getline()** reads in the ANN data filename from keyboard and stores it in character array **name[]**.

....

```
void getline()
{
        int c, i;

        printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
        printf("            BACK-PROPAGATION ARTIFICIAL NEURAL
NETWORK\n");
        printf("\n                Input data filename : ");
        scanf("%s", name);
}
```

....

The function **net_data()** reads in the network training set size and training set data from file stored in **name[]**. It also reads in the network size from the keyboard.

....

```
int net_data()
{
        FILE *fp;

        int c, i, j, tst;

        if ((fp = fopen(name, "r")) == NULL){
                printf("Input File Not Found\n\f");
                return 0;
        }
        printf("\n\nEnter the number of layers in the neural network -> ");
        scanf("%d", &layers);
        for (i = 0; i < layers; ++i){
                printf("    Input the number of nodes in layer%3d  ", i+1);
                scanf("%d", &lyr[i]);
        }
        fscanf(fp,"%3d\n", &dat_num);
        for (i = 0; i < dat_num; ++i){
                for (j = 0; j < (lyr[0] + lyr[layers-1]); ++j)
                        fscanf(fp,"%f\t", &in_data[i][j]);
                fscanf(fp, "\n");
        }
        return 1;
}
```

....

The function **weight_set()** generates random weight values and stores them in the floating point plural variable **wt_val.**

....

```
void weight_set()
{
        int i, j;
        double seed;

        seed = 37591.0;
        for (i = 0; i < (lyr[0] + lyr[2]); ++i)
                for (j = 0; j < (lyr[1] + 1); ++j){
                        seed = fmod((seed*7141+54773), 259200.0);
                        proc[i][j].wt_val = (seed/259200.0);
                }
}
```

....

The function **fwd_prop()** feeds the training data forward through the network by summing interconnection values from dedicated PUs and calculating node outputs via the sigmoidal activation function.

....

```
void fwd_prop()
{
        int i, j, k, l, m;

        tmp_net = 0.0;
        for (i = 0; i < lyr[0]; ++i)
                proc[i][0].net = in_data[innum][i];
        if ((ixproc == 0) & (iyproc < (lyr[0])))
                xnetcE[lyr[1]].net = net;
        if ((ixproc > 0) & (ixproc < lyr[1]+1) & (iyproc < lyr[0]))
                tmp_net = wt_val * net;
        if ((iyproc == (lyr[0]-1)) & (ixproc > 0) & (ixproc < (lyr[1]+1))){
                neti = net;
                for (k = 1; k < lyr[0]; ++k)
                        tmp_net += xnetN[k].tmp_net;
                net = 1.0/(1.0 + fp_exp(-tmp_net));
                xnetcS[lyr[2]].net = net;
        }
        if ((iyproc > lyr[0]-1) & (iyproc < lyr[0]+lyr[2]) & (ixproc<lyr[1]+1))
                tmp_net = wt_val * net;
        if ((ixproc==(lyr[1]+1)) & (iyproc>(lyr[0]-1))
           & (iyproc<(lyr[0]+lyr[2]))){
                for (m = 1; m < lyr[1]+1; ++m)
                        tmp_net += xnetW[m].tmp_net;
                net = 1.0/(1.0 + fp_exp(-tmp_net));
        }
        if (cnt == iter){
                for (i = 0; i < lyr[0]; ++i)
                        if (i > lyr[0] - 5)
                                printf("%9.6f,%9.6f, %9.6f\n", proc[i][0].net,
                                        proc[lyr[0]+i][lyr[1]+1].tmp_net,
                                        proc[lyr[0]+i][lyr[1]+1].net);
        }
}
```

....

The function **bck_prop()** backpropagates the RMS error, or "cost", of the training set through the network on the PU array adjusting the interconnection weight values per the Delta Rule and the Generalized Delta Rule.

....

```
void bck_prop()
{
        int i, j, k, l;
        plural float sum;
        plural float adj;


        tmp_wt = wt_val;
        for (i = 0; i < lyr[layers-1]; ++i)
                proc[lyr[0]+i][lyr[1]+1].diff = in_data[innum][lyr[0]+i];
        if ((ixproc == (lyr[1]+1)) & (iyproc > (lyr[0]-1))
           & (iyproc < (lyr[0]+lyr[layers-1]))){
                diff = diff - net;
                for (j = 1; j < lyr[layers-2]+1; ++j)
                        xnetW[j].wt_val += diff * 0.3 * xnetW[j].net;
                xnetcW[lyr[1]].diff = diff;
        }
        if ((iyproc == lyr[0]-1) & (ixproc > 0) & (ixproc < (lyr[1]+1))){
                sum = 0.0;
                for (k = 1; k < lyr[layers-1]+1; ++k)
                        sum += xnetS[k].diff * xnetS[k].tmp_wt;
                adj = 0.3 * sum * net * (1 - net);
                wt_val += adj * neti;
                for (l = 1; l < lyr[0]; ++l)
                        xnetN[l].wt_val += adj * xnetN[l].net;
        }
}
```

....

The function **error()** calculates the "cost" of the most recent pass of the training set through the network, and returns the floating point error value back to the calling routine.

....

```
float error()
{
        float tot;

        cnt + = 1;
        tot = (1.0/(dat_num*lyr[layers-1]))*sqrt(errsum);
        if (tot < errmin)
                errmin = tot;
        if (cnt > (iter+1)){
                printf("%5d,%9.6f,%9.6f\n", cum, tot, errmin);
                cnt = 0;
                cum + = 1;
        }
        errsum = 0.0;
        return tot;
}
```

....

The function **prog_int()** is called when training is halted or when the program is first run so as to determine if training is to continue, weight values saved, untrained data tested, or the program to be halted.

....

```
void prog_int()
{
        char j;

        printf("\n      1)  Save & quit\n");
        printf("        2)  Quit\n");
        printf("        3)  Continue training network\n");
        printf("        4)  Run untrained data through network\n");
        printf("  Enter selection -> ");
        while ((j = getchar()) = = '\n');
        switch(j){
                case '1' : save_net();
                           exit(0);
                case '3' : errval = errval/10.0;
                           printf ("\n  Continuing %6.5f\n", errval);
                           break;
                case '4' : test_net();
                           break;
                default  : exit(0);
        }
}
```

....

The function **save_net()** saves the interconnection weights of the network to the file in character variable **wgtfile.**

....

```
void save_net()
{
        FILE *fp;
        int i, j, k;

        fp = fopen(wgtfile, "w");
        fprintf(fp, "%3d\t", layers);
        for (i = 0; i < layers; + + i)
                fprintf(fp, "%3d\t", lyr[i]);
        fprintf(fp, "\n");
        for (i =0; i < (lyr[0] + lyr[2]); + +i){
                        for (k = 0; k < lyr[1]; + +k)
                                fprintf(fp, "%9.6f\t", proc[i][k+1].wt_val);
                        fprintf(fp, "\n");
                }
        printf("\n   Weights saved in file %s", wgtfile);
        printf("\n");
        exit(0);
}
```

....

The function **recall_wgts()** retrieves interconnection weights of a previuosly trained network from filename **wgtfile** if it exists in the directory.

....

```
void recall_wgts()
{
        int i, j, k, tmp_layers;
        int tmp_lyr[MAX_LAYERS];
        float wtv;

        fscanf(np, "%d\t", &tmp_layers);
        for (i = 0; i < tmp_layers; ++i)
                fscanf(np, "%d\t", &tmp_lyr[i]);
        fscanf(np, "\n");
        for (i = 0; i < (tmp_lyr[0] + tmp_lyr[2]); ++i){
                        for (k = 0; k < tmp_lyr[1]; ++k){
                                fscanf(np, "%f\t", &wtv);
                                proc[i][k+1].wt_val = wtv;
                        }
                        fscanf(np, "\n");
                }
}
```

....

The function **test_net()** runs untrained data through the neural network to determine it's ability to produce accurate answers for the untrained inputs. The untrained inputs and the correct outputs are stored in filename in character variable **netfile.**

....

```
int test_net()
{
        int i, j, numb;
        float costsum, cst;
        char netfile[NAME_SIZE], annfile[NAME_SIZE];
        FILE *fp;
        FILE *ap;

        printf(" Enter file name of untrained data -> ");
        scanf("%s", netfile);
        if ((fp = fopen(netfile, "r")) == NULL){
                printf("\n  Input file Not Found\n");
                exit(0);
        }
        printf(" Enter filename for network output : ");
        scanf("%s", annfile);
        ap = fopen(annfile, "w");
        fscanf(fp, "%d\n", &numb);
        for (i = 0; i < numb; ++i){
                for (j = 0; j < (lyr[0] + lyr[layers-1]); ++j)
                        fscanf(fp, "%f\t", &in_data[i][j]);
                fscanf(fp, "\n");
        }
        for (innum = 0; innum < numb; ++innum){
                cnt = iter + 1;
                costsum = 0.0;
                cst = 0.0;
                fwd_prop();
                for (j = 0; j < lyr[layers-1]; ++j){
                        fprintf(ap, "%f\t", proc[lyr[0]+j][lyr[1]+1].net);
                        costsum =
                        in_data[innum][j+lyr[0]]
                             - proc[lyr[0]+j][lyr[1]+1].net;
                    cst += costsum * costsum;
          }
          cst = (1.0/(numb*lyr[layers-1]))*sqrt(cst);
          printf("  RMS error = %9.6f\n", cst);
     }
     return 1;
}
```