IMPLEMENTATION OF THE BACKPROPAGATION ALGORITHM ON iPSC / 2 HYPERCUBE MULTICOMPUTER SYSTEM

A THESIS SUBMITTED TO THE
DEPARTMENT OF
COMPUTER ENGINEERING AND INFORMATION SCIENCE
AND
THE INSTITUTE OF ENGINEERING AND SCIENCES
OF
BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

BY
Deniz Ercoşkun
December 1990

IMPLEMENTATION OF THE BACKPROPAGATION ALGORITHM ON iPSC/2 HYPERCUBE MULTICOMPUTER SYSTEM

A THESIS SUBMITTED TO THE

DEPARTMENT OF

COMPUTER ENGINEERING AND INFORMATION SCIENCE

AND

THE INSTITUTE OF ENGINEERING AND SCIENCES

OF

BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Deniz Ercoşkun

December 1990

Deniz Ercoşkun

tarafından [illegible]

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assist. Prof. Dr. Kemal Oflazer(Principal Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Varol Akman

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assist. Prof. Dr. Cevdet Aykanat

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assist. Prof. Dr. Uğur Halıcı

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Dr. Neşe Yalabık

Approved for the Institute of Engineering and Sciences:

Prof. Dr. Mehmet Baray
Director of Institute of Engineering and Sciences

ABSTRACT

IMPLEMENTATION OF THE BACKPROPAGATION ALGORITHM ON iPSC/2 HYPERCUBE MULTICOMPUTER SYSTEM

Deniz Ercoşkun
M.S. in Computer Engineering and Information Science
Supervisor: Assist. Prof. Dr. Kemal Oflazer
December 1990

Backpropagation is a supervised learning procedure for a class of artificial neural networks. It has recently been widely used in training such neural networks to perform relatively nontrivial tasks like text-to-speech conversion or autonomous land vehicle control. However, the slow rate of convergence of the basic backpropagation algorithm has limited its application to rather small networks since the computational requirements grow significantly as the network size grows. This thesis work presents a parallel implementation of the backpropagation learning algorithm on a hypercube multicomputer system. The main motivation for this implementation is the construction of a parallel training and simulation utility for such networks, so that larger neural network applications can be experimented with.

ÖZET

GERİ YANSITMA ALGORİTMASININ iPSC/2 HYPERCUBE PARALEL İŞLEMCİSİNDE GERÇEKLEŞTİRİLMESİ

Deniz Ercoşkun
Bilgisayar ve Enformatik Mühendisliği Yüksek Lisans
Tez Yöneticisi: Y. Doç. Dr. Kemal Oflazer
Tarih 1990

Geri yansıtma, bazı yapay sinir ağı modelleri için geliştirilmiş bir öğrenme algoritmasıdır. Bu algoritma, özellikle bu tip sinir ağı modellerinin eğitilmesinde kullanılmaktadır. Temel geri yansıtma algoritmasının yavaş yakınsaması bu algoritmanın kullanımını küçük sinir ağlarıyla sınırlandırmıştır. Bu tez çalışmasında geri yasıtma algoritması hypercube paralel işlemcisinde gerçekleştirilmiş ve bir dizi yapay sinir ağına uygulanmıştır. Bu çalışmanın diğer bir amacı, büyük yapay sinir ağları için bir simulasyon ve öğretim ortamı geliştirilmesidir.

ACKNOWLEDGMENT

I thank my advisor, Assist. Prof. Dr. Kemal Oflazer for teaching me the meaning of the word "research," and providing perfect dosage of criticism and unfailing support throughout the thesis. I am thankful for his proper combination of justice and mercy on deadlines to ensure the completion of this thesis. I explicitly want to acknowledge the helpful contributions of Assist. Prof. Dr. Cevdet Aykanat during the last stages of my work. I owe debts of gratitude, to Elvan Göçmen, Zeliha Gökmenoğlu, Güliz Ercoşkun, Müjdat Pakkan for their help in shaping this thesis. Finally, I owe a measure of gratitude Bilkent University, who provided a very fruitful environment without which this work could not have been possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Backpropagation [22] is a supervised learning procedure for a class of artificial neural networks. It has recently been widely used in training such neural networks to perform relatively nontrivial tasks (e.g., text-to-speech conversion [27], autonomous land vehicle control [16]). However, the slow rate of convergence of the basic backpropagation algorithm coupled with the substantial computational requirements has limited its application to rather small networks. In this thesis, a parallel implementation of the backpropagation learning algorithm on an 8-processor Intel iPSC/2 hypercube multicomputer system is presented.

In computer science, there is a widespread feeling that conventional computers are not good models of the cognitive processes embodied in the brain. Tasks like arithmetic calculations and flawless memorization of large numbers of unrelated items are very easy for the computers and very hard for humans. However, humans are much better than computers at recognizing objects and their relationships, at natural language understanding, and at commonsense reasoning. In addition to all these, humans are experts at learning to do things through experience. The computational process in the brain is significantly different from those in the computers, and this causes the performance difference between humans and computers. The human brain consists of billions of comparatively slow neurons, that all compute in parallel, while conventional digital computers have fast processor units, which operate by executing in a sequential manner a sequence of very simple primitive steps [7].

The *Artificial Neural Network* model is a computational model proposed for emulating the functionality of human brains. This model consists of many

neuron-like computational elements operating in parallel, arranged in a structure similar to biological neural networks [13]. Learning, a property associated only with humans and animals, is a very important property of this model [4].

Artificial neural networks has been a popular research area for the last forthy years. During this period a number of different neural net models have been proposed, e.g., Hopfield networks [9], Boltzmann Machines [1], Kohonen's Self Organizing Feature Maps [13], Perceptrons [20].

The perceptron model was introduced in 1962 [20]. Later a special class of perceptrons multi-layer perceptrons attracted significant attention, mainly because of its simple structure, and potential capacity. Learning algorithm, called the *backpropagation learning algorithm* for a subclass of multi-layer perceptrons was proposed in 1986 to solve the learning problem of multi-layer perceptrons [23]. This thesis presents a tool for training and simulating backpropagation neural networks on an Intel's iPSC/2 hypercube multicomputer system. The main motivation for this implementation is the *development of a parallel training and simulation utility* for this class of networks so that larger neural network applications can be experimented with.

The remaining parts of this thesis are organized as follows. Chapter 2 surveys artificial neural networks. In Chapter 3, the two classes of perceptrons, namely *single-layer* and *multi-layer* perceptrons, and a subclass of multilayer perceptrons called *backpropagation neural networks* are explained in detail along with their network topologies and the applicable learning algorithms. Chapter 4 gives a detailed explanation of the standard backpropagation learning algorithm. Chapter 5 discusses the parallelism in the backpropagation algorithm and presents general information about the iPSC/2 system. Afterwards, the implementation of the standard backpropagation algorithm on the Intel iPSC/2 Hypercube multicomputer system is explained, along with the detailed explanation of two different approaches to the algorithm, *network partitioning* and *training set partitioning*. Finally, other parallel implementations of the backpropagation algorithm in literature along with the performance achieved with these implementations and the parallel platforms used are summarized. Chapter 6 describes our parallel backpropagation training and simulation tool, HYPERBP. Chapter 7 gives mathematical performance

models for calculating the theoretical elapsed times of network partitioning off-line, and training set partitioning off-line approaches. By applying the neural network problem parameters to the models, a user can estimate the elapsed times of these approaches, and can select the appropriate approach for a neural network problem without actually running it on the multicomputer. Chapter 8 discusses the five neural networks that are experimented with, using the HYPERBP. The problem descriptions, parameters, experimental results as well as the theoretical results calculated using the mathematical model, are given in this chapter. Conclusions are given in Chapter 9. Appendix A presents the detailed derivation of the standard backpropagation learning algorithm.

Chapter 2

Artificial Neural Networks

Studies for simulating computational processes in brain-like systems brought the need for models of computation that are appropriate for parallel systems composed of large numbers of interconnected, simple units. From these studies a model called *Artificial Neural Networks* is emerged. Artificial neural networks are referred by many different names in literature: Neural Nets, Connectionist Models, Parallel Distributed Processing (PDP) models, and Neuromorphic Systems [13]. All these models are composed of many computational elements operating in parallel and arranged in patterns similar to biological neural networks. Artificial neural networks have six major aspects [21]:

1. A set of processing units,
2. a state of activation,
3. a pattern of connectivity among units,
4. a propagation rule for propagating patterns of activities through network of connectivities,
5. an activation rule for combining the inputs impinging on a unit with the current state of the unit to produce a new level of activation for the unit, and
6. a learning rule whereby patterns of connectivity are modified by experience.

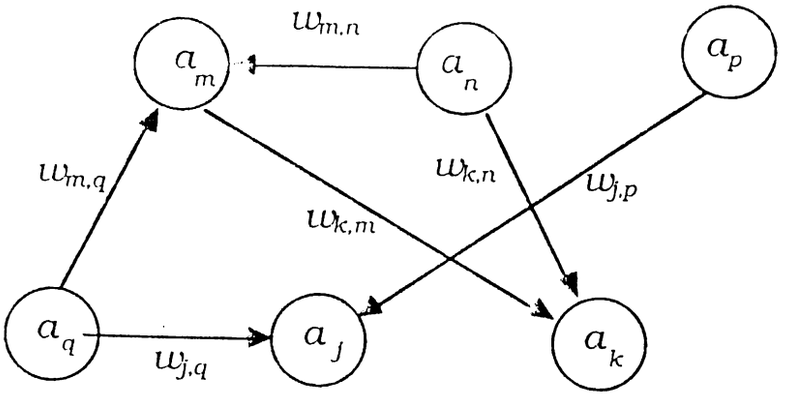


Figure 2.1: Artificial Neural Network Model

An artificial neural network consists of a set of processing units (see Figure 2). Every unit in the network has an activation value (denoted by a_i) at each point in time. The units within the neural network are connected to each other. The pattern of connectivity along with the neurons, determine what the response of the network to a specific input, will be. The output value of a unit is sent to other units in the system through this set of unidirectional connections (in Figure 2 the connections are represented as arrows between units). Associated with each connection, is the *weight* or the strength of the connection. The weight of a connection from unit i to unit j is represented as w_{ji}, and determines the amount of effect the source unit has on the destination unit. How the incoming inputs to a unit should be combined is determined by a propagation rule. The output values of the source units are combined with the corresponding weight values of the connections. Then, with the help of the propagation rule, the *net* input of the destination unit is calculated. The activation value of a unit is passed through a rule called activation rule to determine the new activation value of the unit. The artificial neural networks are modifiable, in the sense that the pattern of connectivity can change as a function of experience. Changing the processing or knowledge structure of an artificial neural network model involves changing the pattern of its connectivity. The rule which defines how weights or strengths of connections should be modified

through experience is defined by the learning rule of the network. This process is called as learning or training. Learning can either be supervised where an external agent indicates to the network what the correct response should be, or unsupervised where the network itself classifies the input patterns. (See [6] for a detailed overview of various learning methods.)

Artificial neural network models offer an alternative knowledge representation paradigm against conventional models. In neural network models the knowledge is represented in a distributed fashion by the strength of connections and interactions among numerous very simple processing units. Neural networks also offer fault tolerance. This property of artificial neural networks is mainly due to the distributed representation, since the behavior of the network is not seriously degraded when a (small) number of processing units fail. Another important property of the artificial neural networks over conventional models is that they can learn their behavior through a training or learning process.

Chapter 3

Perceptrons

3.1 Single Layer Perceptrons

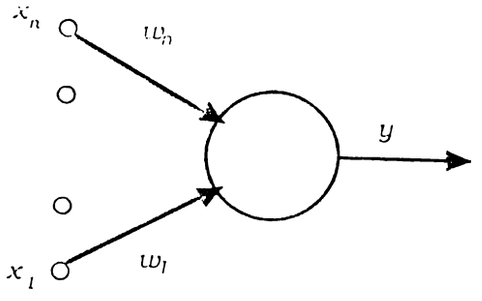
Perceptron is the name given to a class of simple artificial neural network structure [20, 25, 8]. A perceptron is a threshold logic unit with n inputs (see Figure 3.1). Associated with each input there is a real valued weight that plays a role analogous to the synaptic strength of inputs to a biological neuron. The input to the perceptron is an n-dimensional vector. The perceptron multiplies each component of the input vector with the associate weight and sums these products. The perceptron gives an output of 1 if this sum exceeds a threshold value θ and 0 otherwise. A more formal definition is given in Equation 3.1, where x_i is the activity on the i^{th} input line and w_i is the associated weight, and θ is the threshold.

Perceptrons can learn their expected behavior through a process called the *Perceptron Convergence Procedure* [20]. The procedure is as follows:

1. First, connection weights are initialized to small, random, non-zero values.

2. Then, the input with n continuous valued elements is applied to the input and the output is computed using equation 3.1. If the computed output value is different from the desired output value then connection weights are adapted using the error correction formula

$$\Delta\omega_j = \eta(t - o)i_j \tag{3.2}$$



$$y = f(\sum_i x_i w_i) \tag{3.1}$$

$$f(t) = \begin{cases} 1 & \text{if } t > \theta \\ 0 & \text{otherwise} \end{cases}$$

Figure 3.1: A Single-Layer Perceptron

where ω_j is the weight of the connection connecting the j^{th} input value to the perceptron, η is a constant which determines the learning rate of the procedure, t is the target (expected) value of the perceptron, o is the calculated value of the perceptron, i_j is the j^{th} input value.

3. *Step 2* is repeated until the network responds with the desired outputs to the given inputs.

The main result proven about perceptron convergence procedure is the following: Given an elementary perceptron, an input word W which consists of n continuous elements, and any classification $C(W)$ for which a solution exists, then beginning from an arbitrary initial state, perceptron convergence procedure will always yield a solution to $C(W)$ in finite time. Thus, if the given classification is separable (that is the classes fall on different sides of some hyperplane in n dimensional space), then the perceptron convergence procedure converges and positions the decision hyperplane between these classes [20, 13].

The results of a very careful mathematical analysis of the perceptron model was published in 1969 [14]. This careful analysis of conditions under which perceptron systems are capable of carrying out the required mapping of input

to output showed that in a large number of interesting cases, network models of this kind are not capable of solving the problem [14]. This idea was supported by many example problems, where perceptron models failed to find the solution. A classical example problem is the *exclusive-or*, (XOR) problem where the output is 0 if the inputs are same, and it is 1 if they are different. It has been shown that perceptron model fails to solve even such a simple problem [14] because the inputs of the problem is not separable. Thus, the single layer perceptron model can not solve the XOR problem, since there is no hyperplane (a line in this case) to separate the two classes of the problem.

3.2 Multilayer Perceptrons

Following these pessimistic results, an alternative model was suggested where the perceptron model was augmented with a layer of perceptron-like hidden units between input nodes and original perceptron layer. In such networks, there is always a recording (internal representation) of the input patterns in the hidden units. As a consequence, these hidden units can support any required mapping from input to output units. Thus, if we have the right connections from the input patterns to a large enough set of hidden units, we can always find a representation that will perform any mapping from input to output through these hidden units. This model given the name *multi-layer perceptrons* [14].

The multi-layer perceptron model fixed all the weaknesses of single-layer perceptrons. However, in spite of its strong features, the multi-layer perceptrons had a very important problem: They did not have a provably convergent learning procedure. Perceptron convergence procedure which is applicable to single-layer perceptrons, could not be extended to multi-layer perceptrons, because it could not adjust the weights between the perceptrons.

3.2.1 Backpropagation Networks

The pessimistic results about perceptrons actually halted the research on neural networks for about 20 years. In early 1980's, several solutions for the learning problem of multi-layer perceptrons were proposed, but none had the guarantee of convergence.

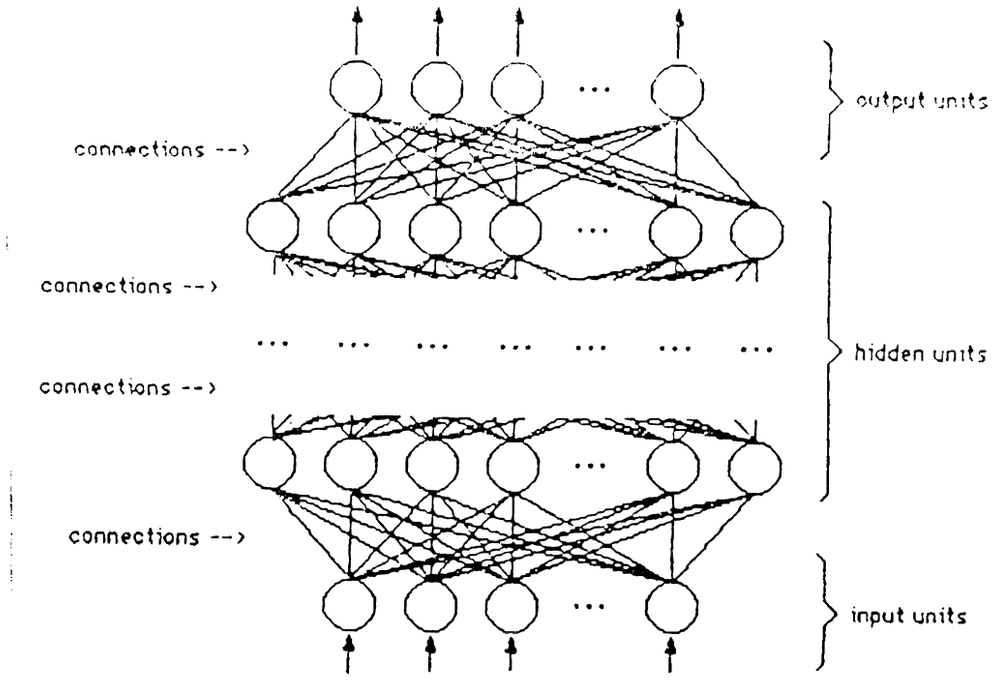


Figure 3.2: A Backpropagation Neural Network

In 1986, a learning procedure for a subclass of multi-layer perceptrons was proposed [24]. This procedure is applicable specifically to *fully connected, layered, feedforward* multi-layer perceptrons as shown in Figure 3.2. The procedure is named the *Backpropagation Learning Procedure*, and the corresponding subclass of multi-layer perceptrons is generally called as *Backpropagation Networks*. In backpropagation networks activations flow from the input layer to the output layer, through a series of hidden layers. Each unit in a layer is connected in the forward direction to each unit in the next higher layer. In backpropagation networks the activation values of the input layer is set in accordance with the sample input, then the rest of the units in the network find their activation values. The activation values of the units in the output layer determine the output of the network. The learning rule is called as the backpropagation rule [22], and is in fact a generalization of Perceptron Convergence Procedure. Backpropagation Learning Procedure involves only local computations. Although it does not guarantee to converge to the correct solution, in almost all applications it has done so. These properties of the learning algorithm, and the strong features of the multi-layer perceptron algorithm, made the multi-layer perceptrons a very popular neural network model.

Chapter 4

The Backpropagation Algorithm

The backpropagation algorithm is a supervised learning method for the class of networks described in Chapter 3. Starting with a given network structure and weights initialized to small random values, the backpropagation algorithm updates the weights between units as a given set input/output pattern associations – known as the *training set* – are presented to the network. Presentation of all the patterns in the training set to the network is known as an *epoch*. The backpropagation algorithm can be implemented as either an *on-line* or an *off-line* version. In the on-line version, the weights in the networks are adjusted after each input/output pair is presented, while in the off-line method the weights are adjusted after an epoch. The on-line method is described below (the off-line method is very similar).

In on-line training, each input/output presentation consists of two stages:

1. A *forward pass* during which inputs are presented to the input layer and the activation values for the hidden and the output layers are computed.
2. A *backward pass* during which the errors between the computed and expected outputs are propagated to the previous layers and the weights between the units are appropriately updated.

Training a backpropagation network typically takes many epochs. The network weights may never settle into stable values for certain problems as there is no corresponding convergence theorem for this algorithm. However, in practice almost all the networks have converged to configurations with reasonable performance.

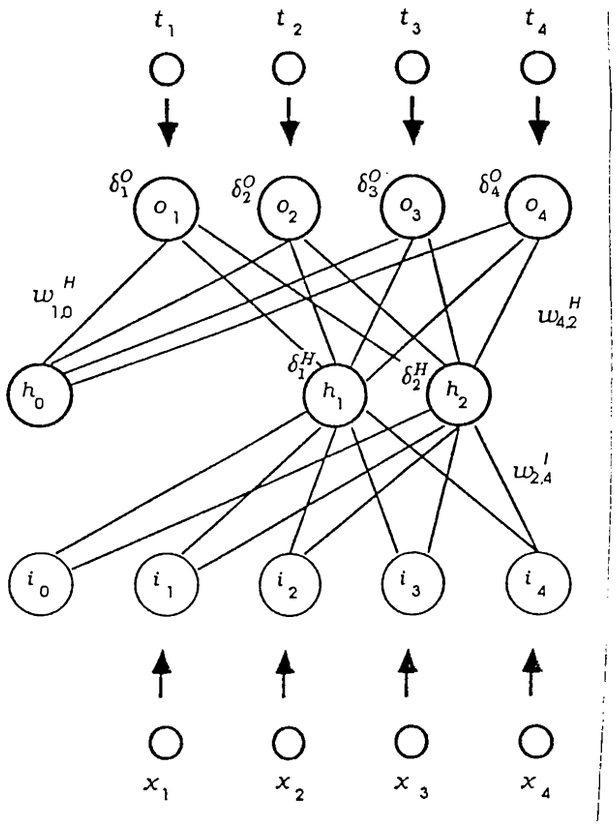


Figure 4.1: A Backpropagation Neural Network

The following is a formal description of the backpropagation algorithm for networks with a single hidden layer – extension to networks with multiple hidden layers is straightforward. Let N_i, N_h and N_o be the number of units at the input, hidden, and the output layers of a network respectively. We assume that the input and hidden layers employ an additional unit for thresholding purpose whose output is always a 1. The activation values will be denoted by i_j ($0 \leq j \leq N_i$, $i_0 = 1$ always) at the input layer, by h_j ($0 \leq j \leq N_h$, $h_0 = 1$ always) at the hidden layer, and by o_j ($1 \leq j \leq N_o$), at the output layer. Let the elements of the pattern vector be x_j ($1 \leq j \leq N_i$) and the corresponding expected output be t_j ($1 \leq j \leq N_o$) (see Figure 4). We initialize the input layer:

$$i_j = x_j \qquad 1 \leq j \leq N_i \tag{4.1}$$

and then propagate the activations from the input to the hidden layer with

$$h_j = F(\sum_{k=0}^{N_i} \omega_{j,k}^I \cdot i_k) \tag{4.2}$$

which is the weighted sum computation and application of activation function F. In backpropagation learning algorithm generally, the sigmoid function is used as the activation function. The sigmoid function is defined as

$$F(\alpha) = \frac{1}{1 + e^{-(\alpha)}} \tag{4.3}$$

Then, we propagate the activations from hidden layer to the output layer with

$$o_j = F(\sum_{k=0}^{N_h} \omega_{j,k}^H \cdot h_k) \tag{4.4}$$

Here $\omega_{j,k}^I$ denotes the weight between the k^{th} input unit and the j^{th} hidden unit, and $\omega_{j,k}^H$ denotes the weight between the k^{th} hidden unit and the j^{th} output unit. We then compute the errors at the output layer as

$$\delta_j^O = o_j(1 - o_j)(t_j - o_j) \qquad 1 \leq j \leq N_o \tag{4.5}$$

This definition of the error is related to the derivative of the activation function (see Appendix A, for the details of the mathematical derivation.) These errors are then propagated to the hidden layer as

$$\delta_j^H = h_j(1 - h_j) \sum_{k=1}^{N_o} \omega_{k,j}^H \delta_k^O \qquad 1 \leq j \leq N_h \ ^1 \tag{4.6}$$

The weights between the hidden and output layers are then updated with

$$\Delta\omega_{k,j}^H = \eta \delta_k^O h_j \qquad 1 \leq k \leq N_o, 1 \leq j \leq N_h, \tag{4.7}$$

and the weights between the input and hidden layers are updated with

$$\Delta\omega_{k,j}^I = \eta \delta_k^H i_j \qquad 1 \leq k \leq N_h, 1 \leq j \leq N_i \tag{4.8}$$

here η is a constant called the *learning rate.*

[1] In networks with more than one hidden layer, such errors are computed for every hidden layer.

The off-line version of the algorithm accumulates these $\Delta\omega_{k,j}$ values for all the patterns during an epoch and performs a weight update only at the end of the epoch with the accumulated changes. The computation of $\Delta\omega_{k,j}$ may also involve a momentum term in certain implementations of the algorithm ; this has been observed to improve the convergence rate. Using a momentum term α, the weight update rule for all layers is changed to

$$\Delta\omega_{k,j} = \eta\delta_k^O h_j \; + \; \alpha\Delta\omega_{k,j}^* \tag{4.9}$$

where $\Delta\omega_{k,j}^*$ is the previous $\Delta\omega_{k,j}$ value.

The procedure outlined above is repeated with every input/output pair in the training set and epochs are repeated as many number of times as necessary until satisfactory convergence is achieved (though as stated earlier there is no guarantee for convergence).

A closer look at the forward pass

The forward pass of the backpropagation computation is essentially a sequence of matrix – vector multiplications with intervening applications of the sigmoidal activation function to each element of the resulting vectors. For example, the outputs of the first hidden layer $\mathbf{H} = [h_1, h_2, \ldots, h_{N_h}]^T$ can be written as

$$\mathbf{H} = F(\mathbf{W^I} \cdot \mathbf{I}) \tag{4.10}$$

where $\mathbf{W^I}$ denotes the $N_h \times (N_i + 1)$ matrix[2] of weights between the input and the hidden layers, $\mathbf{I}$ denotes the input vector and F denotes the sigmoidal activation function applied to each element of the vector. For the output layer one can similarly write

$$\mathbf{O} = F(\mathbf{W^H} \cdot \mathbf{H'}) \tag{4.11}$$

where $\mathbf{W^H}$ denotes the $N_o \times (N_h + 1)$ weight matrix between the hidden and the output layers and $\mathbf{O} = [o_1, o_2, \ldots, o_{N_o}]^T$ is the output vector and $\mathbf{H'}$ is the same as $\mathbf{H}$ with h_0 prepended. It is therefore possible to use parallel algorithms for matrix-vector multiplication for the parallel processing platform available.

[2]Remember the extra threshold units.

A closer look at the backward pass

During the backward pass, the δ_j^O's computed at the output layer are propagated to the previous layers. A closer look at the definition of δ_j^H shows that the summation involved is actually computing an entry of the multiplication of a matrix $(\mathbf{W^H})^T$ – the transpose of the weight matrix $\mathbf{W^H}$ – and the vector $\delta^{\mathbf{O}} = [\delta_1^O, \delta_2^O, \ldots, \delta_{N_o}^O]$. Each such element computer is then multiplied by a scalar $(h_i(1 - h_i))$. Once $\delta^{\mathbf{H}}$ is computed, $\mathbf{\Delta W}$'s can be computed and the weight matrices can be updated.

Chapter 5

Parallel Implementation of Backpropagation Algorithm

5.1 Parallelism in Backpropagation

The backpropagation algorithm described in Chapter 4 offers opportunities for parallel processing in a number of levels. The on-line version of the algorithm is more limited than the off-line version in the ways parallel processing can be applied because of the necessity of updating the weights at every step. The following is a list of possible approaches to parallel processing of backpropagation algorithm, each with a different level of parallelism [15]:

1. Each unit at the hidden and output layers can perform the computation of the weighted sum (which is actually a dot-product computation) using a parallel scheme. For example, each multiplication can be performed in parallel and the results can be added with a tree-like structure in logarithmic number of steps. Here, the granularity of parallel computation is a single arithmetic operation like multiplication or addition.

2. Within each layer, all the units can compute their outputs in parallel, once the outputs of the previous level are available. Here, the granularity of parallel computation is the computation performed by a single unit.

3. With the off-line version where weight updates are performed once per epoch, all the patterns in the training set can potentially be applied to multiple copies of the network in parallel and the resulting errors

can be combined together. The granularity of parallel computation is a complete forward pass of the pattern through the network followed by the computation of changes in weight (but not the update of the weights).

Some or all of these parallel processing approaches can be used together depending on the available resources and the configuration of the parallel implementation platform.

5.2 The Hypercube Architecture

The iPSC/2[1] (Intel Personal Super Computer/2) is a distributed memory multiprocessor system. It consists of a set of *nodes* , and a front-end processor called *host* [10]. Each node is a processor, memory pair. Physical memory in each node is distinct from that of the host and other nodes. The nodes of the iPSC/2 system are connected in a *hypercube* topology. An n-dimensional hypercube consists of $k = 2^n$ vertices labeled from 0 to $2^n - 1$ and such that there is an edge between any two nodes if and only if the binary representations of their labels differ by precisely one bit [26].

System Overview

The iPSC/2 is the second generation hypercube supplied by Intel. An iPSC/2 system can have up to 128 nodes [10, 3].

The hypercube system is controlled from a host computer called System Resource Manager or SRM. The host computer is a PC/AT compatible computer having the following features [10, 3]:

- Intel 80386 *central processor* running at 16MHz,
- Intel 80387 *numeric processor* running at 16MHz,
- 8.5 Megabytes *memory*, and
- AT&T UNIX, Version V, Release 3.2, Version 2.1 *operating system*

[1] iPSC is a registered trademark of Intel Corporation

The hypercube computer at Bilkent University has 8 nodes. Each node has the following features [10, 3]:

- Intel 80386 *node processor* running at 16MHz,
- Intel 80387 *numeric coprocessor* running at 16MHz,
- 4 Megabytes *memory*, and
- NX/2 (*N*ode e*X*ecutive/2) *operating system*, providing message passing to communicate with the other nodes, and the host computer

The Communication System

In iPSC/2 hypercube multiprocessor system, communication between processors is achieved by *message passing*. Data is transferred from processor A to processor B by traveling across a sequence of nearest neighbor nodes starting with node A and ending with B [26]. The iPSC/2 hypercube system has a communication facility called Direct Connect Module (DCM)[2] for high speed message passing. DCM is used as a communication tool within nodes as well as between nodes and host. It supports peak data rates of 2.8 Megabytes/second[10]. At worst case, the hypercube interconnection network enables any one processor to send a message to another in at most a logarithmic number of hops (3 in our case).

5.3 Mapping Backpropagation to Hypercube

In this thesis, an on-line version and an off-line version of the backpropagation algorithm using the second approach of parallelism has been implemented. This method, where units within a single layer compute their outputs in parallel, is called the *Network Partitioning* method. Also off-line version of the algorithm using the third approach of parallelism has been implemented. This method, where the training set is partitioned among processors is called the *Training Set Partitioning* method.

[2]Direct Connect is a trademark of Intel Corporation.

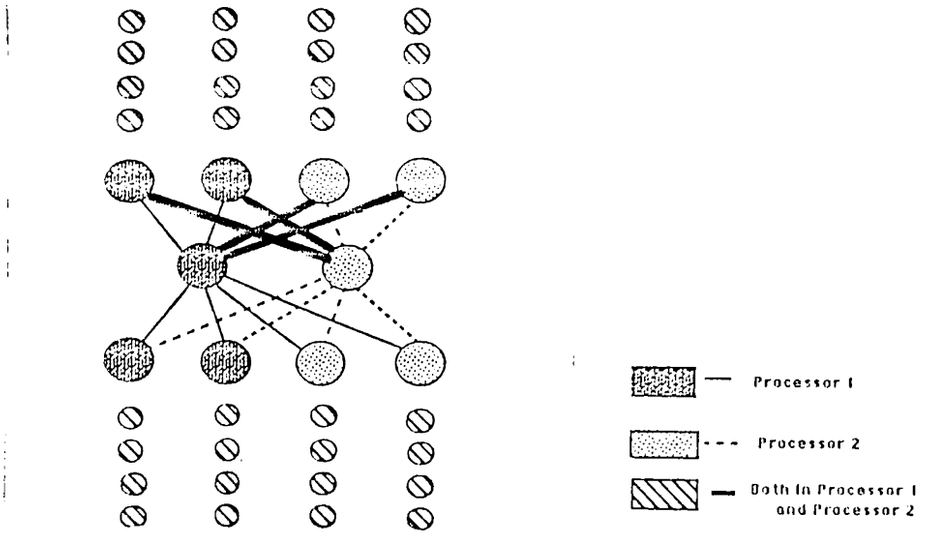


Figure 5.1: Mapping backpropagation to multiple processors using Network Partitioning

Network Partitioning

In network partitioning, assuming $P = 2^p$ processors available, the units in each hidden layer hidden layer and the output layer are partitioned so that (approximately) the same number of units from each layer are distributed to each processor. This is equivalent to partitioning the weight matrices into horizontal strips and assigning each strip to a processor. For example $\mathbf{W^I}$ matrix is partitioned so that the first N_h/P rows are assigned to the first processor, the second N_h/P rows are assigned to the second processor, and so on. The input vector $\mathbf{I}$ is available to all the processors. It can be seen that each processor assigned to a portion of the units from each layer needs to have access to the outputs of all the units in the prior layers. For instance, for computing the outputs of the first hidden layer, all the processors need to access all of the input vector $\mathbf{I}$. In a network with a single hidden layer, in order to compute the outputs of the output layer, all the processors need to access all of the output vector $\mathbf{H}$ of the hidden layer. Since a given processor computes only a portion of the layer outputs, the processors have to synchronize after each layer computation and then exchange their respective portions of the relevant output vectors by a global communication, so that all the processors get a copy of the output vector for the next stage of the computation (see Figure 5.1). In a network with multiple hidden layers a global communication would be necessary after each hidden layer computation in the forward pass. The total number of global communications performed during the forward pass of the

algorithm is equal to the number of layers in the network.

The forward pass is completed when each processor computes its portion of The network output vector O. After the error vector $\delta^{\mathbf{O}}$ is computed and passed to all the processors, the next step in the backward pass is the computation of the error vector for the (last) hidden layer for which the transpose of the weight matrix has to be multiplied with $\delta^{\mathbf{O}}$. Since the forward pass requires that the weight matrices be distributed to the processors as rows, multiplication with the transpose cannot readily be done. The necessary weight matrix values can be obtained in three different ways.

1. As a first attack to the problem, the weight values necessary at the backward pass, that are partitioned among the processors can be gathered by a number of communications.

2. The required weight information can be kept at the corresponding processors in addition to the horizontal matrix strips. This requires extra computations to maintain them.

3. Instead of keeping extra information, processors can calculate their partial δ^H values necessary for the calculation of the overall δ^H vector and by a global communication, and a following internal addition can calculate the overall δ^H vector.

We select the second approach for the implementation. The first method requires extra communication steps which significantly slows down the algorithm. The number of communications performed at the third approach is equal to the second one, but the information amount per communication increases by an amount proportional to $log_2 P$. Communication in the hypercube architecture being a bottleneck, we have opted to do some redundant computation instead of communication.

Instead of distributing just the horizontal strips from the weight matrices to the processors, we also distribute a vertical strip as shown by the shaded area in Figure 5.2. During the forward pass, each processor uses the horizontal strip, but during the backward pass it uses the vertical strip for multiplication with the transpose. Thus in the backward pass each processor computes the appropriate portion of $\delta^{\mathbf{H}}$ locally and then performs a global communication

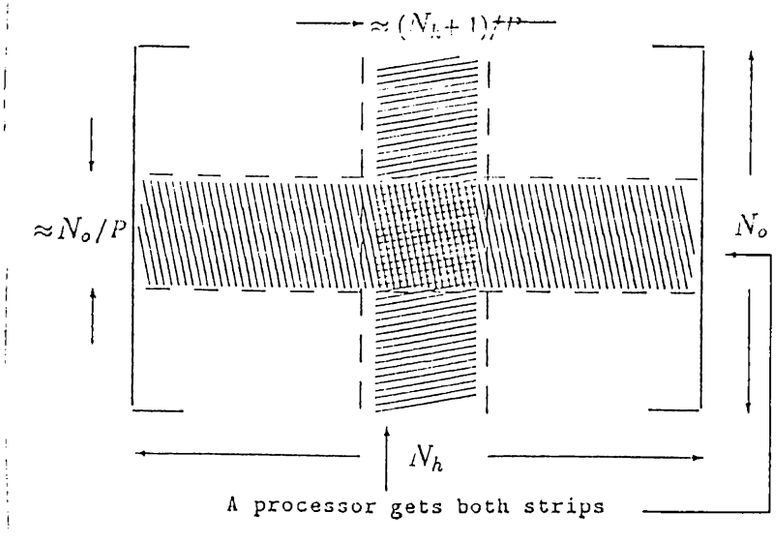


Figure 5.2: Partitioning the weight matrix $\mathbf{W^H}$ for forward and backward passes.

to obtain a copy of the whole $\delta^{\mathbf{H}}$ vector for the next stage of computation. The total number of communications performed at the backward pass is equal to the number of layers within the network. If there are more hidden layers, this procedure is repeated. After the computation of the error vectors, each processor computes the weight changes for the portions of the weight matrices assigned to it and then updates the weights in parallel. At this stage some of the computations are redundant since we are essentially keeping two copies of each weight matrix distributed across the processors and we are trading extra storage and redundant computation, for communication.

Training Set Partitioning

In training set partitioning, the training set of the neural network is partitioned among the processors so that nearly the same number of input output pairs are kept in all processors. In addition to a portion of the training set, each processor keeps a copy of entire network (both units and weights see Figure 5.3), and storage for accumulating the total weight change due to its own training set.

In this method, each processor performs the algorithm on its own training set. After all processors have finished calculating weight changes for all their input output pairs, they share their accumulated weight changes with the other

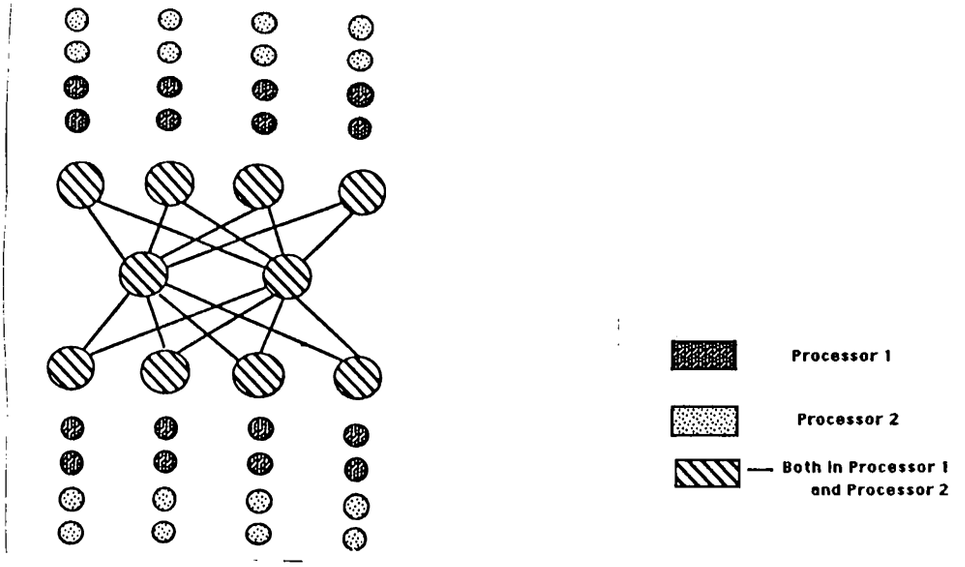


Figure 5.3: Mapping backpropagation to multiple processors using Training Set Partitioning

processors using the communication system. Finally, each processor updates its local weights with the resulting total weight change over whole training set.

In order to parallelize the backpropagation algorithm by partitioning the training set across the processors, it is assumed that the changes to weights due to each input output pattern are independent; that is, the training set can be applied in any order and yield the same result . Clearly, this condition is not satisfied by the on-line version of the algorithm, in which weight changes are calculated and applied after each pattern presentation. Consequently, this kind of parallelism is applicable to only the off-line version of the algorithm.

5.4 Other Parallel Implementations

There have been a number of implementations of backpropagation learning algorithm on a number of different platforms. These include implementation of the NETTalk system on the massively parallel Connection Machine [2], the simulator on the WARP array processor for a neural network for road recognition and autonomous land vehicle control [16, 17], the implementation of backpropagation on IBM's experimental GF11 system with 566 processors capable of delivering 11 gigaflops [28], an implementation of backpropagation on an array of Transputers [19]. Other massively parallel systolic VLSI emulators for artificial neural networks have also been proposed [18, 11].

The Connection Machine is a highly parallel computer configurable with between 16,384 to 65,536 processors. The implementation of NetTalk system on a Connection Machine with 16 K processors offers a speed up of 500 over a similar implementation of the backpropagation algorithm on VAX-11/780, and a speed up of 2 over an implementation on Cray 2 [2].

The WARP machine is a programmable systolic array composed of a linear array of 10 processors. Each processor is capable of performing a peak rate of 10 million floating point operations per second. The backpropagation simulator implemented on WARP machine is first implemented using the second type of parallelism (see Section 5.1) , then it is implemented using the third type of parallelism. It is experimented that second implementation of the algorithm performs much better than the first implementation. It is reported that the second implementation can update 17 million weights per second, and this is 6 to 7 times faster than a similar simulation running on Connection Machine [17].

The GF11 is an experimental SIMD machine consisting of 566 processors. Each processor is capable of performing 20 million floating point operations per second. The implementation of backpropagation simulator for GF11 System is planned to be used for continuous speech recognition. The simulator implemented for GF11 which uses the third type of parallelism approach, is estimated to deliver 50 to 100 times the performance of the WARP implementation [28].

Chapter 6

HYPERBP - The Parallel Backpropagation Simulator

The HYPERBP tool is a library of C functions for the iPSC/2 hypercube multicomputer system. Using these functions in an appropriate order, a user can create a backpropagation neural network, train it with the standard backpropagation learning algorithm, simulate it, and obtain the simulation results.

A user who wants to use the simulator HYPERBP has to write a C program and has to include the definition file namely *common.h* at the top of his program. This file includes necessary type definitions and constant declarations. The user must also include a header file depending on the parallel version to be used. If the user wants to use the training set partitioning method then *train_set_part.h* header file is to be included at the top of the program. If the network partitioning method is to be used then the *network_part.h* header file must be included. The user also has to write a block called *Backpropagation Block* in the program. This block consists of a set of C statements and calls to the simulator functions.

Backpropagation Simulation and Training Tool functions can be classified into seven categories:

1. Backpropagation Block Delimiter calls,
2. Neural Network Definition calls,
3. Training Parameter Setting calls,

4. Training Set Specification calls,

5. Neural Network Simulation calls,

6. Neural Network Training call, and

7. Information Retrieval calls.

6.1 Backpropagation Block Delimiter Calls

The backpropagation block begins by a *bp_begin* call and ends by a *bp_end* call.

bp_begin

The function call *bp_begin* marks the beginning of the backpropagation simulation block within the C code. It is the first function to be called in the backpropagation block. The function has the following form :

INTEGER[1] *bp_begin(n_of_prcs, name_of_cube)*
INTEGER n_of_prcs;
*char *name_of_cube;*

The *bp_begin* call takes two parameters. The first parameter, *n_of_prcs*, specifies the number of processors to be used during training , and simulation. The second parameter, *name_of_cube*, is the name to be given to the allocated cube.

After the execution of this call, a hypercube named *name_of_cube*, consisting of possibly *n_of_prcs* is allocated; the number of processors successfully allocated is returned as a result.

The number of processors allocated for a process has to be a power of 2. This is due to the requirements of the iPSC/2 hypercube multicomputer system (see page 17). Thus, if the value of parameter, *n_of_prcs* is not a power of two, this value is rounded to the next power of two, and then a cube consisting of

[1]The type INTEGER is defined as *long* in the common.h file

that many number of processors is allocated.
Example :

> A user can do a simulation with a cube named *deniz* with 4 processors, by the following call.
>
> *k=bp_begin(4,"deniz");*

If the number of processors available at that moment is 4 or greater, then the function will allocate a cube named "deniz" with 4 processors. Under such circumstances the value of variable k will be 4. But, if the number of processors available in the system at that moment is less than 4, then the largest possible cube will be allocated, and its number of processors will be returned as a result of the call.

bp_end

The function call *bp_end* marks the end of the backpropagation simulation block within the C code. It is the last function called at the end of backpropagation block. The function has the following form :

void bp_end()

6.2 Neural Network Definition Calls

In the backpropagation block, first the neural network to be used is declared. This declaration is surrounded by the *bp_def_begin* and *bp_def_end* calls.

bp_def_begin

The function call *bp_def_begin* marks the beginning of the neural network declaration within the backpropagation block. The function has the following form:

INTEGER bp_def_begin (n_of_layers)

INTEGER n_of_layers;

The *bp_def_begin* call takes one parameter. The parameter, *n_of_layers* specifies the number of layers in the network. This is *hiddenlayer(s)+outputlayer*.

After the execution of this call, the value of the parameter *n_of_layers* is returned if such a number of layers is acceptable, otherwise 0 is returned otherwise.

bp_input

The function *bp_input* defines the input layer of the network. It has the following form :

INTEGER bp_input (n_of_nodes, bias_flag)
INTEGER n_of_nodes;
BOOLEAN[2] *bias_flag;*

The *bp_input* call takes two parameters. The first parameter, *n_of_nodes* specifies the number of nodes to be used in the first layer, namely the input layer of the neural network. The second parameter, *bias_flag* specifies the existence of a dynamic biasing node in the input layer. If the bias_flag has a TRUE value, then an extra special node is augmented to the input layer. The activation value of this node is always one. Its weights on the connections to the upper layers are automatically updated like other ordinary nodes during training. These weights are used as a dynamic biasing factor during the calculation of the activation values in the upper layer.

After the execution of this call, the value of parameter *n_of_nodes* is returned, if there is no error, otherwise a 0 is returned.

bp_hidden

The function *bp_hidden* defines a hidden layer of the network. It has the following form :

[2]The type BOOLEAN is declared as *short* in the common.h file.

```
INTEGER bp_hidden(hl_index, n_of_nodes, afptr,
                  epfptr, bias_flag)
INTEGER hl_index;
INTEGER n_of_nodes;
ADDTYPE afptr;
ADDTYPE epfptr;
BOOLEAN bias_flag;
```

The *bp_hidden* call takes five parameters. The first parameter, *hl_index* specifies the hidden layer being defined, since, a neural network may have more than one hidden layer. All these layers must be defined one by one using the successive *bp_hidden* calls. The *hl_index* parameter distinguishes hidden layers, the hidden layer following the input layer has an index 1, while the hidden layer following it has an index 2, etc. The second parameter, *hl_index* specifies the number of nodes at the hl_index^{th} layer. The third parameter, *afptr* is a function pointer to a library function, which will be used for calculating the current hidden layer nodes activation values. When a *NIL* value is passed as the pointer value, the default C function is used to calculate the activation value. The default function is the standard sigmoid function. The fourth parameter, *epfptr* is a function pointer to a C function, which will be used for propagating the error values of the current hidden layer. When a *NIL* value is passed as the pointer value, the default C function is used as the error propagation function. The last parameter, *bias_flag* is a biasing flag parameter similar to the one in the *bp_input* function.

After the execution of this call, the value of parameter *n_of_nodes* is returned if there is no error. Otherwise a value of 0 is returned.

bp_output

The function *bp_output* defines the output layer of the network. It has the following form :

```
INTEGER bp_output(n_of_nodes, afptr)
```

INTEGER n_of_nodes;

ADDTYPE afptr;

The *bp_hidden* call takes two parameters. The first parameter, *n_of_nodes* specifies number of nodes in the output layer of the neural network. The second parameter, *afptr* is a pointer to a function, which will be used to calculate the activation values of the nodes in the output layer. When a *NIL* value is passed as the pointer value, the default C function is used to calculate activation value. The default function is the standard sigmoid function.

After the execution of this call, the value of parameter *n_of_nodes* is returned if there is no error. Otherwise a value of 0 is returned.

bp_def_end

bp_def_end marks the end of the neural network declaration within the backpropagation block. The function has the following form :

void bp_def_end()

After this call, a neural network with the specified properties is built.
EXAMPLE:

A user can define a neural network with 3 layers, by the following function calls.

bp_def_begin(3);
bp_input(6,TRUE);
bp_hidden(1, 10, NIL, NIL, TRUE);
bp_hidden(2, 3, NIL, NIL, TRUE);
bp_output(4, NIL);
bp_def_end();

There are 4 input nodes and an extra biasing node. The first hidden layer will have 10 nodes, and an extra biasing node. The default activation function will be used for these nodes, also at the backward

pass of the learning algorithm default error propagation functions will be used. The second hidden layer will have 3 nodes, and an extra biasing node. Also the default functions will be used for the nodes in this layer. The output layer will consist of 4 nodes, and these nodes will use the default activation function.

6.3 Training Parameters Setting Calls

The parameters to be used during the learning process are *momentum*, *learning rate*, and the *initialization range*. These parameters are set using the following calls.

bp_setmomentum

The function call *bp_setmomentum* sets the value of the momentum term to be used in the backpropagation learning algorithm.

void bp_setmomentum (momentum_value)

SIMTYPE[3] *momentum_value;*

The function has only one parameter. The value of parameter *momentum_value* is used to set the momentum term of the learning algorithm. If this function is not called within the backpropagation block, the default momentum value of 0.9 is used.

bp_setlearningrate

The function call *bp_setlearningrate* sets the value of the learning rate term to be used in the backpropagation learning algorithm.

void bp_setlearningrate(learning_rate_value)

SIMTYPE learning_rate_value;

[3]The type SIMTYPE symbolizes the type of the simulation going on, this value can be long or float.

The function has only one parameter. The parameter, *learning_rate_value*'s is used to set the learning rate term of the learning algorithm. If this function is not called within the backpropagation block the default learning rate value of 0.2 is used.

bp_randomrange

The function call *bp_randomrange* sets the upper and lower limit values to be used during weight matrix initializations, with random numbers. This weight randomization process is actually the first step of the backpropagation learning algorithm.

void bp_randomrange(lower, upper)

SIMTYPE lower, upper;

The function has two parameters. The first parameter, *lower*, is used as the lower limit, and the second parameter, *upper*, is used as the upper limit value during the random initialization. If this function is not called within the backpropagation block the default limits, 0 and 0.32767 are used.

6.4 Training Set Specification Calls

The training set of a neural network problem can be introduced to the simulation and training tool in two steps. First, the input pattern set is introduced by *bp_setinput* or *bp_setsinput* call, and then the target pattern set is introduced by the calls *bp_settarget* or *bp_setstarget.*

bp_setinput

The function *bp_setinput* defines the input patterns of the training set. The function has the following form :

INTEGER bp_setinput(b_add_input,n_of_patterns)

*SIMTYPE **b_add_input;*

INTEGER n_of_patterns;

The *bp_setinput* call takes two parameters. The first parameter, *b_add_input*, is a pointer to a matrix containing the input patterns of the training set. One input pattern resides in each row of the matrix. The number of columns of this matrix must be equal to the number of nodes at the input layer. The second parameter, *n_of_patterns* specifies the number of input patterns that is being defined. Thus, it specifies the training set size of the neural network. This call returns the value of *n_of_patterns* parameter if there is no error. Otherwise, it returns a value of 0.

bp_settarget

The function call *bp_settarget* introduces the target (in other words, destination) patterns of the training set. The function has the following form :

INTEGER bp_settarget(b_add_target,n_of_patterns)

*SIMTYPE **b_add_target;*

INTEGER n_of_patterns;

The *bp_settarget* call takes two parameters. The first parameter, *b_add_target*, is a pointer to a matrix containing the target patterns of the training set. One target pattern resides in each row of the matrix. The number of columns of this matrix must be equal to the number of nodes at the output layer. The second parameter, *n_of_patterns*, specifies the number of target patterns that is being defined. Thus, it specifies the training set size of the neural network. This value has to be consistent with the number of input patterns specified in the calls *bp_setinput* or *bp_setsinput*.

This call returns the value of *n_of_patterns* parameter if there is no error. Otherwise, it returns 0.

The training set of many neural network problems requires sparse input patterns and/or sparse target patterns. These sparse patterns consists of lot

of 0 values and very few 1 values. In order to introduce such patterns in an efficient way two special functions are implemented in HYPERBP; called, *bp_setsinput* and *bp_setstarget.*

While preparing the pattern array for the calls *bp_setsinput* and *bp_setstarget* the user has to place one pattern at each row of the array in a coded way. In each row there are $n + 1$ elements, where n is the number of 1 values within that pattern. The first element of the row holds the value n. The following n elements of the row specify the indices of the input unit to which the value 1 has to be applied.

bp_setsinput

The function *bp_setsinput* introduces the sparse input patterns of the training set. The function has the following form :

```
INTEGER bp_setsinput(b_add_input,n_of_patterns)
    unsigned char **b_add_sinput;
    INTEGER n_of_patterns;
```

The *bp_setsinput* call takes two parameters. The first parameter, *b_add_sinput*, is a pointer to a matrix containing the input patterns of the training set in a coded form as explained above. One input pattern resides in each row of the matrix. The second parameter, *n_of_patterns*, specifies the number of input patterns that is being defined. Thus, it specifies the training set size of the neural network.

This call returns, the value of *n_of_patterns* parameter if there is no error. Otherwise, it returns 0.

In a backpropagation block there can be a *bp_setinput* or *bp_setsinput* call, but not both.

bp_setstarget

The function *bp_setstarget* introduces the sparse target patterns.

```
INTEGER bp_setstarget(b_add_starget,n_of_patterns)
    unsigned char **b_add_starget;
    INTEGER n_of_patterns;
```

The *bp_setstarget* call takes two parameters. The first parameter, *b_add_target*, is a pointer to a matrix containing the target patterns of the training set in a coded form as explained at page 33. One target pattern resides in each row of the matrix. The second parameter, *n_of_patterns*, specifies the number of target patterns that is being defined. Thus, it specifies the training set size of the neural network. This value has to be consistent with the number of input patterns specified in the calls *bp_setinput* or *bp_setsinput*.

This call returns the value of *n_of_patterns* parameter if there is no error. Otherwise, it returns a value of 0.

In a backpropagation block there can be a *bp_settarget* or *bp_setstarget* call but not both.

6.5 Neural Network Simulation Calls

The functions *bp_forward* and *bp_sforward* perform a forward pass over the defined neural network with the given input pattern.

void bp_forward

bp_forward simulates the current neural network. It actually performs a forward pass of the backpropagation learning algorithm with the specified input vector. The function has the following form :

```
void bp_forward(an_input_pattern)
    ADDTYPE an_input_pattern;
```

The function *bp_forward* takes only one parameter. The parameter, *an_input_pattern* is a pointer to an input vector. This input vector is applied to the input layer and a forward pass is performed over the neural network.

void bp_sforward

The *bp_sforward* simulates the current neural network. It actually performs a forward pass of the backpropagation learning algorithm, with the specified sparse input vector. The function has the following form :

```
void bp_sforward(an_sinput_pattern)
    unsigned char an_input_pattern;
```

The function *bp_sforward* takes only one parameter. The parameter, *an_sinput_pattern* is a pointer to an input vector which is coded as explained at page 33. This input vector is forwarded to the input layer and a forward pass is performed over the neural network.

6.6 Neural Network Training Call

The function call *bp_learn* performs the forward and backward passes of the backpropagation learning algorithm.

bp_learn

bp_learn performs a specified number of epochs of the backpropagation algorithm to the defined network.

```
long bp_learn(n_of_patterns, n_of_epochs, online_flag)
    INTEGER n_of_patterns, n_of_epochs;
    BOOLEAN online_flag;
```

The function takes two parameters. The first parameter, *n_of_epochs*, specifies the number of epochs to be performed. The second parameter, *online_flag*, specifies the version of the program to be used. If this parameter has the value TRUE then the on-line version of the algorithm is performed. Otherwise, the off-line version of the algorithm is performed. But note that, with training set

partitioning approach this parameter has no effect, as only the off-line version of the algorithm is possible with training set partitioning approach.

After the execution of this call, the function returns the elapsed time for this call.

EXAMPLE:

> User can train the previously defined neural network with the on-line version of the backpropagation algorithm, for 500 epochs by the following call.
>
> *time_elapsed=bp_learn(500,TRUE);*

With this call the network is trained using the previously set training set, for 500 epochs, and the elapsed time of this training process is returned at the end.

6.7 Information Retrieval Calls

The current state of the neural network can be learned by the user with the help of the functions *bp_getstates* and *bp_getweights.*

bp_getstates

The function call *bp_getstates* puts the current activation values of the nodes at a specific layer into a vector.

INTEGER bp_getstates(layer_index,output_add)

INTEGER layer_index;

ADDTYPE output_add;

The function takes two parameters. The first parameter, *layer_index*, specifies a layer. The second parameter, *output_add*, specifies a pointer to a vector. As a result of *bp_getstates*, the activation values of the nodes at the specified layer is put into the specified vector.

The function returns the value of parameter *layer_index* if there is no error during the execution of the call. Otherwise, it returns a 0.
Example:

> User can put the current activation values of the node at the input layer to the vector, whose beginning address is *output_vec* by the following call.
>
> *bp_getstates(1,output_vec);*

bp_getweights

The function call *bp_getweights* puts the weight values between two layers into a matrix.

BOOLEAN bp_getweights(layer_index,final_layer)

INTEGER layer_index;

ADD_TYPE output_add;

The function takes two parameters. The first parameter, *layer_index*, specifies the layer where the weighted connections originate. The second parameter, *output_add*, specifies a pointer to a matrix. As a result of *bp_getweights*, the weights of the connections originating from the specified layer is put in to the matrix specified by the pointer *output_add.*

The function returns the value of parameter *layer_index* if there is no error during the execution of the call. Otherwise, it returns 0.
Example:

> The user can get the weights of the connections originating from the first hidden layer to a matrix whose beginning address is *output_vec* by issuing the following call.
>
> *bp_getweights(2,output_vec);*

INPUT				OUTPUT			
1	0	0	0	1	0	0	0
0	1	0	0	0	1	0	0
0	0	1	0	0	0	1	0
0	0	0	1	0	0	0	1

Table 6.1: The table illustrating 4-2-4 Encoding Decoding Problem

6.8 Sample Program 4-2-4 Encoding/Decoding Problem

4-2-4 encoding/decoding problem is a well-known neural network Problem. A two-layer Backpropagation Network is used to solve this problem. The network consists of one input layer, one hidden layer, and an output layer. The input layer has 4 nodes, to receive four bits of the input pattern. The hidden layer has 2 nodes. At the output layer, there are four nodes representing four bits of the output pattern.

The 4-2-4 encoding/decoding network is trained using 4 input-output pairs. In the Table 6.1, the input output set used to train the network is given.

A sample C code written to create, train and simulate this problem using the HYPERBP is given bellow.

```
00  #include "common.h"
01  #include "train_set_part.h"
02  main(argc, argv)
03          int argc;
04          char *argv[];

05          {
06          SIMTYPE input[4][4],
07                      target[4][4],
08                      output[4][4];
09          INTEGER i, j;
10          INTEGER NProc, NEpoch, tmp;
11          long et;
```

```
12      if (argc != 3)
13      {
14          printf("Needs two arguments \n");
15          exit(1);
16      }

17      if ((tmp = sscanf(argv[1], "%d", &NProc)) != 1)
18      {
19          printf("Could not parse first arg . Bye \n");
20          exit(1);
21      }

22      if ((tmp = sscanf(argv[2], "%d", &NEpoch)) != 1)
23      {
24          printf("Could not parse second arg . Bye \n");
25          exit(1);
26      }

27      printf("%d processors - %d Epochs \n", NProc, NEpoch);


/* Set the input vectors */
28      input[0][0]=1.0;input[0][1]=0.0;input[0][2]=0.0;input[0][3]=0.0;
29      input[1][0]=0.0;input[1][1]=1.0;input[1][2]=0.0;input[1][3]=0.0;
30      input[2][0]=0.0;input[2][1]=0.0;input[2][2]=1.0;input[2][3]=0.0;
31      input[3][0]=0.0;input[3][1]=0.0;input[3][2]=0.0;input[3][3]=1.0;


/* Set the output vectors */
32      target[0][0]=1.0;target[0][1]=0.0;target[0][2]=0.0;target[0][3]=0.0;
34      target[1][0]=0.0;target[1][1]=1.0;target[1][2]=0.0;target[1][3]=0.0;
35      target[2][0]=0.0;target[2][1]=0.0;target[2][2]=1.0;target[2][3]=0.0;
36      target[3][0]=0.0;target[3][1]=0.0;target[3][2]=0.0;target[3][3]=1.0;


/* Backpropagation Block */
```

```
37          bp_begin(NProc, "Cube1");

38              bp_def_begin(3);
39                  bp_input(4, TRUE);
40                  bp_hidden(1, 2, NIL, NIL, TRUE);
41                  bp_output(4, NIL, NIL);
42              bp_def_end();


/* Neural Network definition block */
43              bp_setlearningrate(0.9);
44              bp_setmomentum(0.4);
45              bp_randomrange(-0.5, 0.5);
46              bp_setinput(input, 4);
47              bp_settarget(target, 4);

48              et = bp_learn(4, NEpoch, TRUE);
49              printf("\n***** ELAPSED TIME of LEARN : %d \n ", et);

50              for (i = 0; i < 4; i++)
51                  {
52                      bp_forward(input[i]);
53                      bp_getstates(3, output);
54                      printf("Output For Input %d \n", i);

55                      for (j = 1; j < 5; j++)
                            printf("%.2f ", output[j]);
56                      printf(" \n ");
57                  }
58
59              printf(" \n ");

60          bp_end();
61  }
```

The first two statements of the given program include the two necessary

files to use HYPERBP. The file *common.h* contains the type and constant definitions. The file named *train_set_part.h* contains the C function codes used by the HYPERBP in training set partitioning method. Between lines 06 to 11 the variables to be used are defined.

The backpropagation block resides between lines 37 to 60. At the first statement of the block (line 37) the number of processors to be used during simulation and training is specified. The neural network definition block comes next (lines 38 to 42). The network defined is a 2 layer network, with one hidden, and one output layer. The input contains 4 nodes plus an extra biasing node. The hidden layer contains 2 nodes plus a biasing node, and the nodes in this layer use the standard activation function and the standard propagation function. The output layer has 4 nodes, and those nodes also use the standard functions during forward and backward passes.

The value of the learning rate parameter is specified to be 0.9 at line 43. The value of the momentum term is specified to be 0.4 at line 44. The weight initialization range is specified to be within -0.5, 0.5 at line 45. At lines 46 and 47 the training set of the network is introduced to HYPERBP.

At line 48 the backpropagation learning algorithm is performed over the declared network *NEpoch* times using the training set partitioning parallelism method. Then the elapsed time returned from the function is printed.

Afterwards, the whole training set is simulated over the network, and the results of this simulation is printed. The *bp_forward* call is used for simulation. The *bp_getstates* call is used to obtain the activation values of the output layer nodes after simulation.

Chapter 7

Performance Models for the Simulator

In this chapter performance models of the implemented parallel backpropagation learning algorithms that have been developed are presented . These models are helpful to the users of HYPERBP tool. Using these models with the parameter of network, and the training set size of the problem, a user can estimate the execution time of a training epoch for a problem without actually implementing it on the hypercube multicomputer system. One can then select the most appropriate version for a problem.

A performance model has been developed corresponding to the two parallel methods, namely, network partitioning off-line and training set partitioning off-line methods. Taking the C codes of the simulator as a base, these models have been derived. A performance model has not been developed for network partitioning on-line method, as there is not an alternative implementation for the on-line version.

7.1 Parameters of the Mathematical Model

The parameters used in the mathematical model is divided into two main classes :

1. System Parameters,
2. Problem Parameters,
   (a) Network Parameters,

(b) Training Set Parameter.

7.1.1 System Parameters

T^+ : The amount of CPU time it takes, to perform a floating point addition on the iPSC/2 hypercube multicomputer system.

$$T^+ = 0.00558 \ msec. \ [3]$$

T^* : The amount of CPU time it takes, to perform a floating point multiplication on the iPSC/2 hypercube multiprocessor system.

$$T^* = 0.00664 \ msec. \ [3]$$

T^S : The amount of CPU time it takes, to compute the sigmoid function of a floating point number, on iPSC/2 hypercube multiprocessor system. The sigmoid function is computed using the following formula (see also Chapter 4)

$$SIGMOID(x) = \frac{1}{1+e^{-x}}. \tag{7.1}$$

The value of SIGMOID function is very close to 0 when the value of x is less than -15.0, and it is very close to 1 when the value of x is greater than 15.0. These properties of the function is taken into consideration in the implementation. The corresponding C function immediately returns 0 if x is less than -15.0 and returns 1 if x is greater than 15.0. If none of these conditions holds then the sigmoid function is calculated using Equation 7.1. This saves time while calculating the sigmoid values of big and small numbers. In the performance model, the amount of CPU time it takes, to perform sigmoid function is taken to be

$$T^S = 0.091 \ msec.$$

which has been experimentally determined for random values of x within a large range.

p	a_p if $0 < N \leq 50$	a_p if $50 < N \leq 100$	a_p if $100 < N \leq 200$	a_p if $N > 200$
2	1.2882	0.5900	0.5900	0.5900
4	1.1343	1.5702	2.1979	2.8111
8	1.6907	2.1562	2.7331	3.5888

Table 7.1: a_p values used in the calculation of $T^{gcol_p}(N)$

p	b_p if $0 < N \leq 50$	b_p if $50 < N \leq 100$	b_p if $100 < N \leq 200$	b_p if $N > 200$
2	0.0075	0.0077	0.0077	0.0077
4	0.0079	0.0108	0.0097	0.0079
8	0.0085	0.0102	0.0098	0.0097

Table 7.2: b_p values used in the calculation of $T^{gcol_p}(N)$

T^L : The amount of CPU time, to perform a loop (loop overhead).

$$T^L = 0.0013 \ msec.$$

This value has been determined experimentally.

$T^{gcol_p}(N)$: The Global Concatenation Operation, concatenates parts of a floating point vector of length N which is distributed among the p nodes of the cube. After the execution of this call the whole vector of length N is in all processors of the cube.

$$T^{gcol_p}(N) \simeq a_p + b_p N \ \ msec$$

The a_p, and b_p values are determined for $p = 2, 4, 8$ experimentally, and are given in Tables 7.1, and 7.2.

$T^{gsum_p}(N)$: The Global Sum Operation, calculates the sum of the components of a floating point vector of size N across all nodes. After the execution of this call the result is returned in the vector to every node.

p	c_p	d_p
2	0.9533	0.0121
4	1.8120	0.0259
8	2.6213	0.0395

Table 7.3: c_p, d_p values used in the calculation of $T^{gsum_p}(N)$

$$T^{gsum_p}(N) \simeq c_p + d_p N \quad msec.$$

c_p, and d_p values have been experimentally determined for $p = 2, 4,$ and 8, and are given in Table 7.3.

7.1.2 Problem Parameters

Network Parameters

p : The number of processor to be used.

n : The total number of layers in the neural network (excluding the input layer).

W : The total number of weights in the neural network.

W^I : The total number of weights between the input layer and the first hidden layer.

N : The total number of nodes in the neural network except input nodes.

N_i : The number of nodes at the layer i. i ranges from 0 (*input*) to n (*output*) layer.

EXAMPLE:

N_n: Number of nodes at the output layer.

N_{n-1}: Number of nodes at the last hidden layer.

Training Set Parameter

S : The Training Set size. The number of input and target pairs comprising the training set.

7.2 Model for Network Partitioning Off-line

The total time of an epoch using network partitioning off-line version of the algorithm can be computed as shown in Equation 7.3, using the parameters introduced at Section 7.1.

In the following equations:

$Time_t^{NP}$ is the total time to perform a whole epoch of the back-propagation algorithm.

$Time_f^{NP}$ is the time to perform the single forward pass of the back-propagation learning algorithm.

$Time_b^{NP}$ is the time to perform the single backward pass of the backpropagation learning algorithm.

$$Time_t^{NP} \simeq S\left(Time_f^{NP} + Time_b^{NP}\right) + \overbrace{\underbrace{\sum_1^n \left(\left\lceil \frac{N_i}{p} \right\rceil N_{i-1}\right)(T^+ + T^L)}_{2} + \underbrace{\left(\left\lceil \frac{N_{i-1}}{p} \right\rceil N_i - \left\lceil \frac{N_i}{p} \right\rceil \left\lceil \frac{N_{i-1}}{p} \right\rceil\right)(T^+ + T^L)}_{3}}^{1} \tag{7.2}$$

1: Update the weight matrix.

2: Update the horizontal strips of the weight matrix.

3: Update the vertical strips of the weight matrix.

$$Time_f^{NP} \simeq \sum_{i=1}^{n}\left(\left\lceil \frac{N_i}{p} \right\rceil (N_{i-1}+1)\left(T^+ + T^* + T^L\right) + \left\lceil \frac{N_i}{p} \right\rceil T^S\right) \tag{7.3}$$

$$+ \sum_{i=1}^{n-1} T^{gcol_p}(N_i)$$

$$Time_b^{NP} \simeq \left\lceil \frac{N_n}{p} \right\rceil \left(2T^+ + 2T^*\right) + T^{gcol_p}(N_n) \tag{7.4}$$

$$+ \sum_{i=1}^{n-1} \left(\left\lceil \frac{N_i}{p} \right\rceil \left(\left(T^+ + 2T^*\right) + N_{i+1} \left(T^+ + T^* + T^L\right) \right) + T^{gcol_p}(N_i) \right)$$

$$+ \sum_{i=1}^{n} \left(\left\lceil \frac{N_i}{p} \right\rceil N_{i-1} + \left\lceil \frac{N_{i-1}}{p} \right\rceil N_i - \left\lceil \frac{N_i}{p} \right\rceil \left\lceil \frac{N_{i-1}}{p} \right\rceil \right) \left(T^+ + 3T^* + T^L\right)$$

7.3 Model for Training Set Partitioning Off-line

The total time of an epoch using training set partitioning off-line version of the algorithm can be computed as shown in Equation 7.6, using the parameters introduced at Section 7.1.

In the following equations :

$Time_t^{TP}$ is the total time to perform a whole epoch of the back-propagation algorithm.

$Time_f^{TP}$ is the time to perform the single forward pass of the back-propagation learning algorithm.

$Time_b^{TP}$ is the time to perform the single backward pass of the backpropagation algorithm.

$$Time_t^{TP} \simeq \left\lceil \frac{S}{p} \right\rceil \left(Time_f^{TP} + Time_b^{TP}\right) \tag{7.5}$$

$$+ WT^+ + \sum_{i=0}^{n-1} T^{gsum_p}(N_i * N_{i+1})$$

$$Time_f^{TP} \simeq \sum_{i=1}^{n} \left(N_i \left((N_{i-1}+1) \left(T^+ + T^* + T^L \right) \right) + N_i T^S \right) \qquad (7.6)$$

$$\simeq W \left(T^+ + T^* + T^L \right) + N \left(T^+ + T^* + T^S \right)$$

$$\begin{aligned} Time_b^{TP} &\simeq N_n \left(2T^+ + 2T^* \right) \\ &+ \sum_{i=1}^{n-1} \left(N_i \left(\left(T^+ + 2T^* \right) + N_{i+1} \left(T^+ + T^* + T^L \right) \right) \right) \\ &+ \sum_{i=1}^{n} \left(N_i (N_{i-1}+1) \right) \left(T^+ + 3T^* + T^L \right) \end{aligned}$$

$$\begin{aligned} &\simeq T^+ + N \left(3T^+ + 5T^* \right) + W \left(2T^+ + 4T^* + T^L \right) \\ &- W^I \left(T^+ + T^* + T^L \right) \end{aligned}$$

Chapter 8

Experiments

A number of well-known neural network problems have been implemented on HYPERBP. This chapter explains these benchmark neural network problems that are solved with the help of HYPERBP tool along with the experimental results and the calculated expected time values using the performance models in Chapter 7. These neural network problems are :

1. 8 Bit Parity problem
2. Digit problem
3. Two Spirals problem
4. NETTalk

8.1 8 Bit Parity Problem

8.1.1 Problem Description

In the *parity* problem, the required output is 1 if the input pattern contains an odd number of 1s, 0 otherwise.

8.1.2 Backpropagation Approach

The parity problem can easily be solved by conventional methods, but it is a very difficult task for backpropagation neural network, because the most similar patterns (those which differ by a single bit) require different answers. The XOR problem which cannot be solved by single layer perceptrons is a parity problem, with input patterns of size two.

Parity problems with patterns sizes ranging from size 4 to 8 have been solved using HYPERBP. In this section, the backpropagation neural network solution to 8 bit parity problem is explained.

Network Parameters

A two-layer backpropagation network is used to solve this problem. The network consists of a set of input nodes, one hidden layer and an output layer.

There are 8 input nodes to receive the 8 input values. The hidden layer has 8 nodes. At the output layer, there is only one node.

Problem Parameters

During training, learning rate term with a value of 0.9 and momentum with a value of 0.4 is used. At the initialization step of the algorithm, the weights of the network are set to random values within the range 0 and 0.32767.

Training Set

The network is trained using 256 input-output pairs. A part of the training set is given in Figure 8.1.

Results

Table 8.1 summarizes the results for the training 8 Bit Parity problem with 1, 2, 4, and 8 processors, with network partitioning on-line, network partitioning

Pair No.	Input Patterns								Target Patterns
1	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	1	1
3	0	0	0	0	0	0	1	0	1
4	0	0	0	0	0	0	1	1	0
.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.
253	1	1	1	1	1	1	0	0	0
254	1	1	1	1	1	1	0	1	0
255	1	1	1	1	1	1	1	0	0
256	1	1	1	1	1	1	1	1	1

Figure 8.1: A subset of the training set of *8 Bit Parity Problem*

off-line, and training set partitioning off-line approaches. The table at the top shows the results from the on-line version with network partitioning (ONL), the table at the middle shows the results from the off-line version with network partitioning (OFL-NP), and the table at the bottom shows the results from the off-line version with training set partitioning (OFL-TP). In each table, the number of processors used for that simulation is in the first column. Expected time calculated, using the formulas in chapter 7, in seconds, is in the second column. The on-line table does not contain expected time column as the is not a performance model applicable for the on-line version. Actual amount of time the simulation of one epoch takes, in seconds, is in the third column. Speed ups calculated using actual time values is in the last column. Speed up is defined as the ratio T_1^a/T_P^a where T_P^a is the actual time taken by executing the program with P processors, and T_1^a is the actual time taken by executing the program on a single node with no parallel processing overhead.

ONL

# Proc	Actual Time	Speed Up
1	1.622	1.00
2	1.562	1.04
4	1.654	0.98
8	1.900	0.85

OFL-NP

# Proc	Expected Time	Actual Time	Speed Up
1	NA[1]	1.090	1.00
2	1.185	1.369	0.79
4	1.341	1.495	0.72
8	1.618	1.780	0.61

OFL-TP

# Proc	Expected Time	Actual Time	Speed Up
1	NA	1.090	1.00
2	0.610	0.553	1.97
4	0.310	0.289	3.77
8	0.160	0.151	7.22

Table 8.1: Results for 8 Bit Parity Problem (Times are per epoch and in seconds)

8.2 Digit Recognition Problem

8.2.1 Problem Description

Digit Recognition is a synthetic network that we have constructed as a sample input for our benchmarks. It is not intended to perform any significant recognition.

In this problem, the 10 by 6 retina containing the image of a digit is introduced to the system, and the expected output of the system is the canonical order of the input digit among 10 possible digits.

[1]Not Applicable.

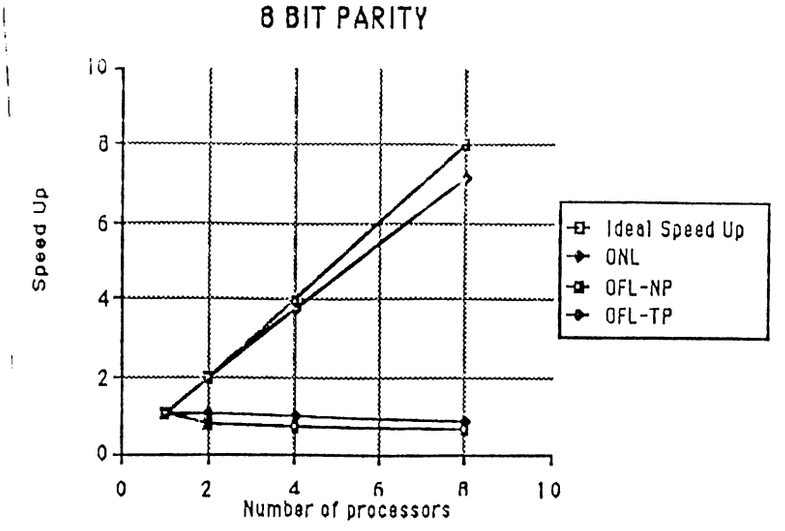


Figure 8.2: Number of processors versus speed up graph for 8 bit parity problem

8.2.2 Backpropagation Approach

Network Parameters

A two-layer backpropagation network is used for this problem. The network consists of one hidden layer and an output layer.

There are 60 input nodes, to receive the 10 by 6 image. The hidden layer has 60 nodes. The output layer has 10 nodes corresponding to 10 digits.

Problem Parameters

During training, a learning rate term with a value of 0.9 and a momentum of 0.5 are used [5]. At the initialization step of the algorithm, the weights of the network are set to random values within the range 0.0 and 0.32767.

The network is trained using 10 input-output pairs.

Results

Table 8.2 summarizes the results for the training digit recognition problem.

ONL

# Proc	Actual Time	Speed Up
1	2.591	1.00
2	1.750	1.48
4	1.042	2.49
8	0.642	4.04

OFL-NP

# Proc	Expected Time	Actual Time	Speed Up
1	NA	2.009	1.00
2	1.274	1.438	1.39
4	0.756	0.857	2.34
8	0.474	0.523	3.84

OFL-TP

# Proc	Expected Time	Actual Time	Speed Up
1	NA	2.009	1.00
2	1.085	1.098	1.83
4	0.731	0.833	2.41
8	0.584	0.747	2.69

Table 8.2: Results for digit recognition network (Times are per epoch and in seconds)

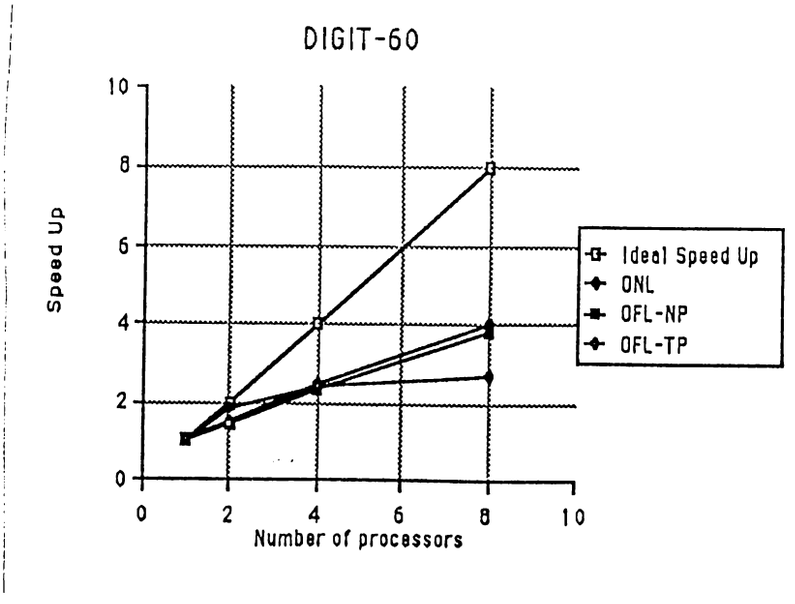


Figure 8.3: Number of processors versus speed up graph for digit recognition problem

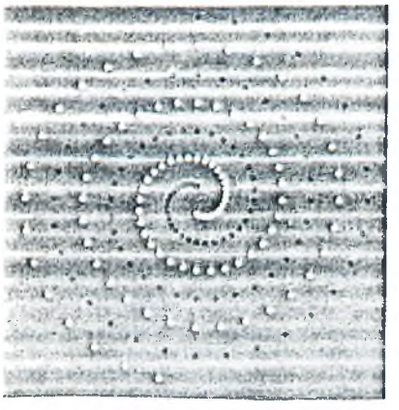


Figure 8.4: Training points for two-spirals problem

8.3 The Two Spirals Problem

8.3.1 Problem Description

The *Two-spirals* problem was originally posed by Wieland [12]. In this problem, there are two distinct spirals made up of points lying on the unit square in *x-y plane*. These spirals coil three times around the origin and around one another (See Figure 8.4). One of the spirals consists of white points while the other one consists of black points. The task is to discriminate between two sets of points on two intertwined spirals.

Although this problem can easily be solved by conventional methods, such as table-lookup, it is a hard task for backpropagation networks. Besides, as there is no way to draw a single straight line in x-y plane so that all black dots end up on one side while all the white dots are on the other side (not linearly separable), this problem cannot be solved using single layer perceptrons.

8.3.2 Backpropagation Approach

In backpropagation approach, the network is trained so that it will respond with a value 1 at the output node if the point is on white spiral, and respond with a value 0 if it is on black spiral. After training, the x and y coordinate

```
main()
 {
    int i;
    double x, y, angle, radius;

    for (i=0; i<=96; i++)
      {
         angle = i * M_PI/16.0;
         radius = 6.5 * (104 - i) / 104.0;
         x = radius * sin(angle);
         y = radius * cos(angle);
      }
 }
```

Figure 8.5: Fragment of C code which generates the training set of two-spirals problem

values of a point, on one of the spirals, is presented to the input layer of the network and corresponding response of the network is received from the output node.

Network Parameters

A three-layer Backpropagation Network is used to solve this problem. The network consists of two hidden layers and an output layer.

There are 2 input nodes, to receive the x and y coordinates of the input point. The first hidden layer has 20 nodes and the second hidden layer has 10 nodes. At the output layer, there is only one node.

Problem Parameters

During training, a learning rate term of 0.1 and a momentum 0.7 are used. At the initialization step of the algorithm, the weights of the network are set to random values within the range -0.5 and 0.5.

ONL

# Proc	Actual Time	Speed Up
1	4.616	1.00
2	3.606	1.28
4	3.063	1.51
8	3.009	1.53

OFL-NP

# Proc	Expected Time	Actual Time	Speed Up
1	NA	3.079	1.00
2	2.656	2.798	1.10
4	2.438	2.528	1.22
8	2.627	2.647	1.16

OFL-TP

# Proc	Expected Time	Actual Time	Speed Up
1	NA	3.079	1.00
2	1.878	1.589	1.94
4	0.965	0.828	3.72
8	0.490	0.461	6.68

Table 8.3: Results for two-spirals network (Times are per epoch and in seconds)

Training Set

The network is trained using 194 input-output pairs. The fragment of C code in Figure 8.5, supplied by Wieland is used to generate two sets of input points, each with 97 members (three complete revolutions at 32 points per revolution, plus end points). Each point is represented by two floating point numbers (the x and y coordinate values of the point). The corresponding output value is 0 for one set and 1 for the other set.

Results

Table 8.3 summarizes the results for the training two-spirals problem with 1, 2, 4, and 8 processors.

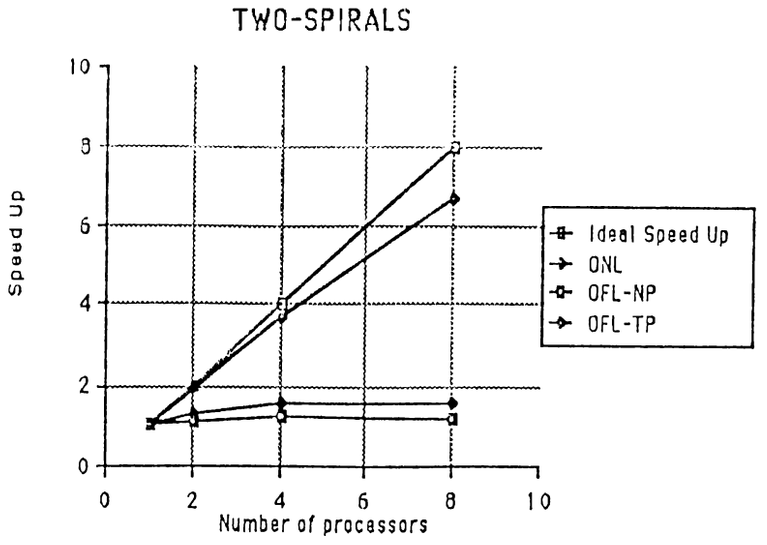


Figure 8.6: Number of processors versus speed up graph for two-spirals problem

8.4 NETTalk

8.4.1 Problem Description

Pronunciation of English words is very inconsistent. Although it follows some rules, these rules are very weak and contains many exceptions as well as many qualifications. These properties of English pronunciation makes it a suitable research area for neural network models.

In 1986 a feed forward neural network to converting English text to speech signals is proposed [27]. The neural network called *NETTalk* accepts a string of characters forming English Text as input and maps this input to the corresponding string of phonemes. This output is eventually forwarded to a speech synthesizer.

8.4.2 Backpropagation Approach

A two layer feed-forward neural network is proposed to solve this problem. The network is trained using both the Boltzmann machine learning algorithm, and backpropagation learning algorithm. It has been concluded that both learning algorithms are appropriate for this problem, but with backpropagation learning

algorithm, the network learns faster.

Network Parameters

The original NETTalk neural network is implemented using HYPERBP developed, with the original training parameters. The network is a three layer backpropagation neural network.

The input to the network is a seven letter string from a word, and the network responds with the phonemic output corresponding to the central letter of the given string. The six surrounding letters are a partial context which guides the pronunciation of the middle letter.

A letter in the input string is represented by one active unit within 29 (26 letters + 3 special symbols) units. Thus, the input layer consists of 29 * 7 = 203 units to represent the 7 character window. The network uses 120 hidden units and 26 units at the output layer represent the phonemes in a distributed fashion with multiple simultaneously active units: 21 output units representing various articulatory features, and 5 units encoding stress and syllable boundaries.

Problem Parameters

During training, a learning rate term with a value of 0.1 and a momentum of 0.7 are used. At the initialization step of the algorithm, the weights of the network are set to random values within the range -0.5 and 0.5.

Training Set

The network is trained using a list of 1000 English words. This set is a random subset of the 20,008 words list used by Sejnowski and Rosenberg. The words are presented to the network through a 7 character long sliding window. Extra characters (spaces and word boundaries) are augmented to the back and front of the word to keep the window full. After every presentation of a word a weight update takes place. So an epoch of the problem (presentation of 1000 words) actually composed of 1000 smaller sub-epochs (presentation of a word).

Pair No.	Input Patterns							Target Patterns	
1	!	!	[	c	o	m	p	k	>
2	!	[	c	o	m	p	e	a	2
3	[	c	o	m	p	e	t	m	<
4	c	o	m	p	e	t	i	p	>
5	o	m	p	e	t	i	t	x	0
6	m	p	e	t	i	t	i	t	>
7	p	e	t	i	t	i	o	I	1
8	e	t	i	t	i	o	n	S	<
9	t	i	t	i	o	n	]	-	0
10	i	t	i	o	n	]	!	x	<
11	t	i	o	n	]	!	!	n	<

! denotes the space character

[denotes the word boundary (front)

] denotes the word boundary (back)

Figure 8.7: Presentation of word *competition* to NETtalk Neural Network

For example, the word *competition* with the articulatory features *kampxtIS-xn* and with the voicing and vowel sounds > 2 <> 0 > 1 < 0 << (for further information see [27]) is presented to the network by 11 input, target patterns as shown in Figure 8.7.

The network successfully converged after 20 presentations of the 1000 word set.

Results

Table 8.4 summarizes the results for the training NETtalk with 1, 2, 4, and 8 processors. The expected time calculation for NETTalk is a little complicated as an epoch of the problem consists of 1000 word presentations, and a presentation of a word itself is a sub-epoch, whose training set size is depending on the word length. In order to calculate the expected time of an epoch, first the 1000 sub-epochs have to be calculated, then they must be added, which is really a tedious job. Because of this deficiency, the expected times for NETTalk are not calculated.

ONL

# Proc	Actual Time	Speed Up
1	9421.976	1.00
2	6417.409	1.47
4	3670.992	2.57
8	2006.281	4.70

OFL-NP

# Proc	Actual Time	Speed Up
1	9114.591	1.00
2	6276.361	1.45
4	3592.464	2.54
8	1884.161	4.84

OFL-TP

# Proc	Actual Time	Speed Up
1	9114.591	1.00
2	6383.399	1.43
4	5495.305	1.66
8	5554.188	1.64

Table 8.4: Results of NETTalk problem (Times are per epoch and in seconds)

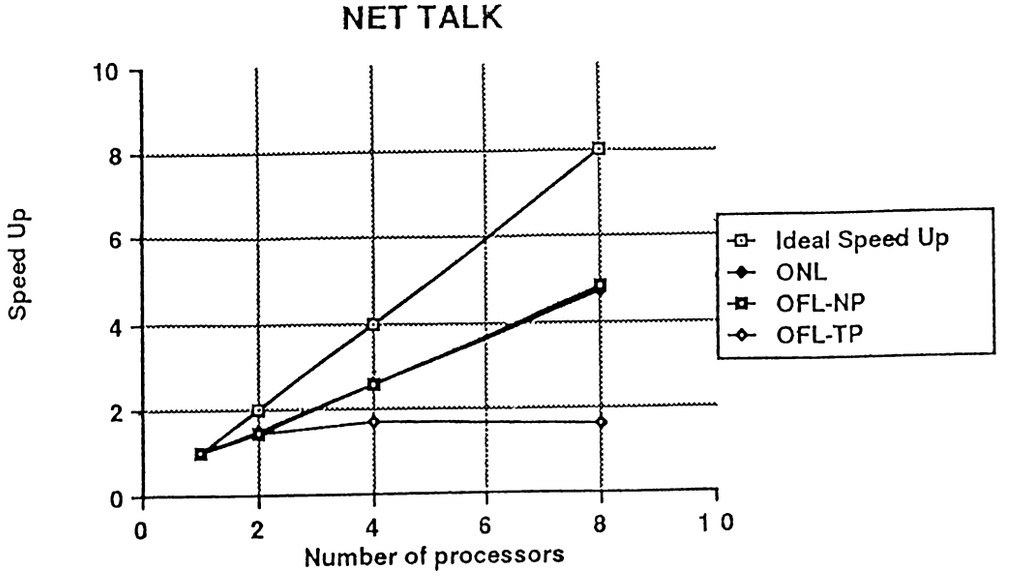


Figure 8.8: Number of processors versus speed up graph for NETTalk problem

The experimental results showed that, the speed up achieved with the 8 bit parity problem with training set partitioning method (7.22 with 8 processors) is much better than the speed up achieved with network partitioning method (0.61 with 8 processors). Similarly, the speed up achieved with the two spirals problem with training set partitioning method (6.68 with 8 processors) is much better than the speed up achieved with network partitioning method (1.16 with 8 processors). On the other hand, the speed up achieved with digit problem with network partitioning method (3.84 with 8 processors) is better than the speed up achieved with training set partitioning method (2.69 with 8 processors). Similarly, the speed up achieved with NETTalk with network partitioning method (4.84 with 8 processors) is much better than the speed up achieved with training set partitioning method (1.64 with 8 processors). These results shows that, in general, training set partitioning method is suitable for problems which have small networks with relatively large training sets, while network partitioning method is more suitable for solving problems with large networks having relatively small training sets.

Chapter 9

Conclusions

A parallel simulator executing on iPSC/2 hypercube multicomputer system has been developed for simulating and training a class of multi-layer perceptrons. The backpropagation learning algorithm is used for training these neural networks. Besides parallel simulation and training, the developed system enables a user to define a backpropagation neural network and to control network parameters (learning rate, momentum, random range for initialization, etc.).

The parallel implementation enables the user to use either the on-line or the off-line version of the algorithm. There are two choices for the off-line version. Training set partitioning or network partitioning can be used depending on the structure of the network and the size of the training set. In this thesis a mathematical performance model of the implemented off-line algorithms has been developed. This mathematical model helps the user to select the most appropriate off-line algorithm for a problem among the two possible implementations.

The experimental results as well as the theoretic results calculated using the mathematical model explained in Chapter 7, showed that :

- The off-line version with training set partitioning gives substantial speed-up for training sets with large numbers of patterns but performs badly for large neural networks with relatively small training set.
- Although the off-line version with network partitioning does not catch up with the speed-up of the training set partitioning, it performs considerably well with large neural networks with small training sets.

The internal communication is costly in iPSC/2 hypercube multicomputer system due to the high set up cost per communication. The off-line version of the backpropagation algorithm with network partitioning method requires frequent communication, once at each layer. These communication requirements of the network partitioning method significantly degrades the performance of the implementation. The communication is the main limitation of this method. The experimental maximum speed up with this method is 4.84 with 8 processors with the NETTalk.

The off-line version of the algorithm with training set partitioning method requires communication once in every epoch. As the communication requirements of this method are few, the high communication cost in iPSC/2 hypercube multicomputer system does not have much significance at the overall performance of the implementation. The experimental maximum speed up achieved with this approach is 7.22 with 8 processors with the 8 bit parity problem. As a result, the off-line version of the training set partitioning method mapped more neatly to the iPSC/2 architecture than the network partitioning method.

The network partitioning method implemented in this thesis requires redundant local computations by an amount proportional to W/P (where W is the number of weights within the neural network, and P is the number of processors) to avoid extra global communications. The network partitioning method could also be implemented making no redundant local computations. In such an implementation the number of global communications would remain same, but the amount of communication per global communication would increase by an amount proportional to $log_2 P$. This implementation can be an interesting topic for a further study.

Appendix A

Derivation of the Backpropagation Algorithm

A.1 Backpropagation Rule

Backpropagation algorithm tries to minimize the square of the differences between the actual and the desired output values summed over the output units and all pairs of input-output vectors. It makes a *gradient descent* in sum squared error surface. As the backpropagation neural network contains hidden layers, the error surface corresponding to the network is not concave upwards. Then, there is the danger of getting stuck into a local minima. But, it has been shown by various examples that, in wide variety of tasks the backpropagation algorithm successfully finds the best set of weights. It rarely gets stuck into a local minima.

Backpropagation algorithm is derived from backpropagation rule posed by Rumelhart [22, 23]. According to the Backpropagation Rule :

- The weight on each line should be changed by an amount proportional to the product of an error signal, δ, available to the unit along that line and the output of the unit sending activation along that line. This can be formulated as :

$$\Delta_p w_{ji} = \eta \ \delta_{pj} \ o_{pi} \tag{A.1}$$

In Equation A.1, $\Delta_p w_{ji}$ represents the amount of weight change at the connection between unit i to unit j presentation of pattern p, η is the learning rate, which is a constant controlling the learning rate of the

algorithm, δ_{pj} is the error value coming from unit j after presentation of pattern p, and o_{pi} is the output value of unit i after presentation of pattern p to the network.

- δ_{pj} value in Equation A.1 can be computed in a recursive manner using the following equations.
  - The process starts with the output units. An output unit j calculates its δ_{pj} value using Equation A.2.

$$\delta_{pj} = f'_j(net_{pj})\ (d_{pj} - o_{pj}) \tag{A.2}$$

    In Equation A.2, d_{pj} is the jth element of destination vector and $f'_j(net_{pj})$ is the derivative of activation function f_j which maps the total input of unit j to its output value.
  - If unit j is not an output unit, then its δ_{pj} value is calculated using Equation A.3.

$$\delta_{pj} = f'_j(net_{pj}) \sum_k \delta_{pk} w_{kj} \tag{A.3}$$

    As specific destination is not given for hidden units, the δ_{pj} values of these units are determined recursively, by means of the δ values of units to which they are directly connected, and the weight of the connection between them.

A.2 The Derivation of Backpropagation Rule

In the backpropagation algorithm, units calculate their net input value using Equation A.4.

$$net_{pj} = \sum_i o_{pi}\ w_{ji} \tag{A.4}$$

If unit i is an input unit then $o_{pi} = i_{pi}$.

After, calculating its net input value, unit j calculates its output value by passing its net_{pj} value from the activation function[1] f_j as in Equation A.5.

$$o_{pj} = f_j(net_{pj}) \tag{A.5}$$

The error measure following the presentation of pattern p is

$$E_p = \frac{1}{2}\sum_j (d_{pj} - o_{pj})^2 \tag{A.6}$$

The total error measure is

$$E = \sum_p Ep \tag{A.7}$$

- The derivative of error measure with respect to each weight is taken to be proportional to weight change of the backpropagation rule , with a negative constant of proportionality, in order to make a gradient descent search. This derivative can be written as the product of two partial derivatives using the chain rule as :

$$\frac{\partial E_p}{\partial w_{ji}} = \frac{\partial E_p}{\partial net_{pj}} \frac{\partial net_{pj}}{\partial w_{ji}} \tag{A.8}$$

Using Equation A.4, we can rewrite Equation A.8 as

$$\begin{aligned} \frac{\partial E_p}{\partial w_{ji}} &= \frac{\partial E_p}{\partial net_{pj}} \frac{\partial}{\partial w_{ji}} \sum_k w_{jk} o_{pk} \\ &= \frac{\partial E_p}{\partial net_{pj}} o_{pi} \end{aligned} \tag{A.9}$$

If we define δ_{pj} as

[1]Activation function f must be a differentiable and nondecreasing function so that the backpropagation rule can make a gradient descent.

$$\delta_{pj} = -\frac{\partial E_p}{\partial net_{pj}} \tag{A.10}$$

then, we can rewrite Equation A.8 as

$$-\frac{\partial E_p}{\partial w_{ji}} = \delta_{pj} o_{pi} \tag{A.11}$$

So a gradient descent can be made in surface E using weight change formula given in Equation A.1.

- To compute δ_{pj} we can apply the chain rule to Equation A.10

$$\begin{aligned} \delta_{pj} &= -\frac{\partial E_p}{\partial net_{pj}} \\ &= -\frac{\partial E_p}{\partial o_{pj}} \frac{\partial o_{pj}}{\partial net_{pj}} \end{aligned} \tag{A.12}$$

We can compute $\partial o_{pj} \,/\, \partial net_{pj}$ using Equation A.5 as

$$\frac{\partial o_{pj}}{\partial net_{pj}} = f_j'(net_{pj}) \tag{A.13}$$

So, Equation A.12 can be rewritten as

$$\delta_{pj} = -\frac{\partial E_p}{\partial net_{pj}} f_j'(net_{pj}) \tag{A.14}$$

  - If the unit computing the error term is an output unit then $\partial E_p / \partial net_{pj}$ can be computed using Equation A.6 as

$$\frac{\partial E_p}{\partial o_{pj}} = -(d_{pj} - o_{pj}) \tag{A.15}$$

    Combining Equations A.14 and A.15, we can compute δ_{pj} value of an output unit by

$$\delta_{pj} = f_j'(net_{pj})(d_{pj} - o_{pj}) \tag{A.16}$$

– If the unit is not an output unit then we can rewrite $\partial E_p / \partial o_{pj}$ as

$$\begin{aligned}
\frac{\partial E_p}{\partial o_{pj}} &= \sum_k \frac{\partial E_p}{\partial net_{pk}} \cdot \frac{\partial net_{pk}}{\partial o_{pj}} \\
&= \sum_k \frac{\partial E_p}{\partial net_{pk}} \frac{\partial}{\partial o_{pj}} \sum_i w_{ki} o_{pi} \qquad \text{from Equation A.4} \\
&= \sum_k \frac{\partial E_p}{\partial net_{pk}} w_{kj} \\
&= \sum_k \delta_{pk} w_{kj}
\end{aligned} \tag{A.17}$$

Combining Equations A.14 and A.17, we can compute δ_{pj} value of units other than output units by

$$\delta_{pj} = f_j'(net_{pj}) \sum_k \delta_{pk} w_{kj} \tag{A.18}$$

References

[1] D. H. Ackley, G. E. Hinton, and T. J. Sejnowski. A learning algorithm for Boltzmann machines. *Cognitive Science*, 9(1):147 - 169, 1985.

[2] G. Blelloch and C. R. Rosenberg. Network learning on the Connection Machine. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, 1987.

[3] L. Bomans and D. Roose. Benchmarking the ipsc/2 hypercube multiprocessor. *Concurrency : Practice and Experience*, 1(1):3 – 18, September 1989.

[4] J. D. Cowan and D. H. Sharp. Neural nets and artificial intelligence. *Daedalus's special AI issue*, 117(1):85 – 120, Winter 1988.

[5] S. E. Fahlman. An empirical study of learning speed in back-propagation networks. Technical Report CMU–CS–88–162, School of Computer Science, Carnegie Mellon University, June 1988.

[6] G. E. Hinton. Connectionist learning procedures. *Artificial Intelligence*, 1988.

[7] G. E. Hinton and J. A. Anderson. *Parallel Models of Associative Memory.* Lawrence Erlbaum Associates, Publishers, 1981.

[8] G. E. Hinton, J. L. McClelland, and D. E. Rumelhart. Distributed representations. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing*, volume 1, chapter 3, pages 77 – 109. MIT Press, 1986.

[9] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of National Academy of Sciences*, 79:2554 – 2558, 1982.

[10] Intel Corporation. *iPSC/2 User's Guide*, 3 edition, March 1989.

[11] S. Y. Kung and J. N. Hwang. Parallel architectures for artificial neural nets. In *Proceedings of IEEE International Conference on Neural Networks*, volume 2, pages 165 - 172, 1988.

[12] K. J. Lang and M. J. Witbrock. Learning to tell two spirals apart. In D. S. T. G. E. Hinton and T. J. Sejnowski, editors, *Proceedings of the 1988 Connectionist Models Summer School.* ,San Mateo, CA: Morgan Kaufmann Publishers, 1988.

[13] R. P. Lippmann. An introduction to computing with neural nets. *IEEE ASSP Magazine*, pages 4 - 22, April 1987.

[14] M. Minsky and S. Papert. *Perceptrons.* MIT Press, 1969.

[15] K. Oflazer and D. Ercoşkun. Implementation of backpropagation learning algorithm on a hypercube parallel processor system. In A. E. Harmancı and E. Gelenbe, editors, *Proceedings of the Fifth International Symposium on Computer and Information Science*, volume 1, pages 347 - 356, 1990.

[16] D. A. Pomerleau. ALVINN: An autonomous land vehicle in a neural network. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 1. Morgan Kaufman, 1989.

[17] D. A. Pomerleau, G. L. Gusciora, D. S. Touretzky, and H. T. Kung. Neural network simulation at Warp speed: How we got 17 million connections per second. In *Proceedings of IEEE International Conference on Neural Networks*, volume 2, pages II–143 - II–150, 1988.

[18] U. Ramacher and J. Beichter. Systolic architectures for fast emulation of artificial neural networks. In *Proceedings of International Conference on Systolic Arrays*, 1989.

[19] G. D. Richards. Implementation of Back-Propagation on a Transputer. Edinburgh Preprint, 1989.

[20] F. Rosenblatt. *Priciples of Neurodynamics.* Spartan, 1962.

[21] D. E. Rumelhart, G. E. Hinton, and J. L. McClelland. A general framework for parallel distributed processing. In D. E. Rumelhart and J. L.

McClelland, editors, *Parallel Distributed Processing*, volume 1, chapter 2, pages 15 – 76. MIT Press, 1986.

[22] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing*, volume 1, chapter 8, pages 318 – 364. MIT Press, 1986.

[23] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing*, volume 1. MIT Press, 1986.

[24] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533 – 536, 1986.

[25] D. E. Rumelhart and D. Zipser. Feature discovery by competitive learning. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing*, volume 1, chapter 5, pages 151 – 193. MIT Press, 1986.

[26] Y. Saad and M. H. Schultz. Topological properties of hypercubes. *IEEE Transactions on Computers*, 37(7):867 – 872, July 1988.

[27] T. J. Sejnowski and C. R. Rosenberg. Parallel networks that learn to pronounce English text. *Complex Systems*, 1:145 – 168, 1987.

[28] M. Witbrock and M. Zagha. An implementation of Back-propagation on GF11, a large SIMD parallel computer. Technical Report CMU–CS–89–208, School of Computer Science, Carnegie Mellon University, December 1989.