6-1993

Signal Processing with Neural Networks

Pero Smrzlic

Western Michigan University

Follow this and additional works at: http://scholarworks.wmich.edu/masters_theses

Part of the Computer Sciences Commons, Mathematics Commons, and the Neurology Commons

Recommended Citation

http://scholarworks.wmich.edu/masters_theses/822

This Masters Thesis - Open Access is brought to you for free and open access by the Graduate College at ScholarWorks at WMU. It has been accepted for inclusion in Master's Theses by an authorized administrator of ScholarWorks at WMU. For more information, please contact maira.bundza@wmich.edu.
SIGNAL PROCESSING WITH NEURAL NETWORKS

by

Pero Smrzlic

A Thesis
Submitted to the
Faculty of The Graduate College
in partial fulfillment of the
requirements for the
Degree of Master of Science
Department of Computer Science

Western Michigan University
Kalamazoo, Michigan
June 1993
SIGNAL PROCESSING WITH NEURAL NETWORKS

Pero Smrzlic, M.S.
Western Michigan University, 1993

In this study, we introduce the Adaptive Back Propagation (ABP) learning algorithm, computationally superior to the standard Back Propagation. The ABP is based on the new activation function, with its corresponding adaptive learning parameter. By using the combination of computer simulations and analysis in the domain of activation function, the Method of One Hidden layer was developed for the effective utilization of units in one-hidden layer networks. A parallel version of the ABP was designed and implemented on a nCUBE-2 supercomputer with 128 processors.

The simulation results suggested a strong correlation between frequency of signals and the role of hidden units. The way patterns are presented during the training process has a significant impact on the obtained results. The results show that the architectures with the perfect distribution of units can be almost perfectly parallelized.
ACKNOWLEDGMENTS

I would like to thank Professor Robert Trenary, my advisor and committee chairman, for his counsel and the time he spent in seeing me through all facets of writing this thesis.

I would like to thank Professor John Kapenga for his valuable suggestions.

I offer my deep appreciation to Professor Frank Severance for his encouragement and for offering many helpful suggestions.

I owe Dr. Mike Peikert for allowing me the use of the ECG data that is a part of Automatic Defibrillator Study conducted by Kalamazoo Center for Medical Studies.

Finally, I would like to thank Dr. Charles Barr, Director of the Center for Applied Medical Informatics, for his encouragement, and support in my research.

Pero Smrzlic
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Signal processing with neural networks

Smrzlic, Pero, M.S.
Western Michigan University, 1993
# TABLE OF CONTENTS

ACKNOWLEDGMENTS ................................................................................... ii
LIST OF TABLES ................................................................................................ v
LIST OF FIGURES .............................................................................................. vi

CHAPTER

I. INTRODUCTION .................................................................................... 1
   Neural Network Architectures .................................................................. 1
   The Three Types of Commonly Used Neural Networks ......................... 6
      Definitions ......................................................................................... 7
   The Delta Rule or Widrow-Hoff Rule With a Teacher .............................. 10
   The Hopfield Minimum-Energy Rule ..................................................... 11
   The Boltzmann Learning Algorithm .................................................... 12
   Back Propagation Learning Rule ............................................................ 13
   Applications of Back Propagation in Signal Processing ......................... 18
   Signal Terminology ............................................................................. 19
   A Preview ............................................................................................. 19

II. ADAPTIVE BACK PROPAGATION .......................................................... 21
   Overview ............................................................................................ 21
   Motivation for Modifying Back Propagation ........................................ 22
   Adaptive Activation Function .............................................................. 24
   Adaptive Back Propagation Algorithm ................................................ 31
   Summary .............................................................................................. 35

III. APPLICATION OF ADAPTIVE BACK PROPAGATION ............................ 36
   Introduction .......................................................................................... 36
## Table of Contents—Continued

### CHAPTER

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation Description</td>
<td>37</td>
</tr>
<tr>
<td>Random Versus Sequential Pattern Presentation</td>
<td>38</td>
</tr>
<tr>
<td>Simulation Results</td>
<td>40</td>
</tr>
<tr>
<td>The Method of One Hidden Layer</td>
<td>45</td>
</tr>
<tr>
<td>ECG Analysis</td>
<td>49</td>
</tr>
<tr>
<td>Summary</td>
<td>55</td>
</tr>
</tbody>
</table>

### IV. PARALLEL PROCESSING AND BACK PROPAGATION                           | 56   |
| Adaptive Back Propagation in Parallel                                   | 57   |
| Implementation on a Hypercube                                          | 60   |
| Summary                                                                | 66   |

### V. CONCLUSION                                                          | 67   |

### APPENDICES

A. Gaussian Activation Function                                           | 69   |
B. Source Code in C for Adaptive Back Propagation on nCUBE-2...           | 71   |

### BIBLIOGRAPHY                                                          | 92   |
LIST OF TABLES

1. XOR Function .......................................................................................... 28
2. Calculated XOR Function With One Adaptive Unit ............................. 29
3. OR Function ............................................................................................ 30
4. Calculated OR Function With One Adaptive Unit ............................. 30
LIST OF FIGURES

1. A Simplified Model of a Neuron ............................................................. 1
2. A Simple Perceptron ................................................................................ 5
3. A Schematic Representation of wij in a Feed-Forward Network .............. 10
4. A Hopfield Network With Four Units ................................................... 12
5. A Boltzmann Machine With Two Hidden Units (3 and 4) ..................... 12
6. A Two-Layer Feed-Forward Network ................................................... 16
7. A Sigmoidal Activated Unit for T=0.9, k=1, and k=3 ........................... 23
8. Bias Effect on the Adaptive Unit ............................................................. 25
9. Interval Width and the Adaptive Unit ..................................................... 26
10. XOR Solution With the Adaptive Unit. F11=0.8 & F10=0.2 ................. 27
11. OR Solution With the Adaptive Unit. F11=0.75 & F10=0.5 .................... 28
12. A Two-Layer Feed-Forward Network, P=2 ........................................ 33
13. The Response of 1-4-1 Network During Sequential Pattern Presentation for Signal s(x) ......................................................................................... 39
14. The Actual Response of 1-4-1 Network for Signal s(x) ......................... 39
15. Output of 1-4-1 Feed-Forward Network Using Adaptive Back Propagation for Signal s(x) .............................................................................. 41
16. Activation of Unit One in the Hidden Layer of 1-4-1 Network, and Signal s(x) ......................................................................................... 42
17. Activation of Unit Two in the Hidden Layer of 1-4-1 Network, and Signal s(x) ......................................................................................... 42
18. Activation of Unit Three in the Hidden Layer of 1-4-1 Network, and Signal s(x) ......................................................................................... 43
19. Activation of Unit Four in the Hidden Layer of 1-4-1 Network, and Signal s(x) ......................................................................................... 43

vi
List of Figures--Continued

20. Activation of the Hidden Units in Adaptive 1-4-1 Network, the Output of the Network, and Signal s(x).............................. 44
21. Sigmoidal vs. Adaptive Activation Function for 1-4-4-1 Network, and Signal s(x)................................................................. 44
22. Feed-Forward Network (1-4-1) With One Hidden Layer.......... 45
23. A Simplified Case of the Inverted Activation of Hidden Units by the Output Unit With the Adaptive Activation Function........ 46
24. Sine Function sin(5x).................................................................... 47
25. Normalized Five Second Data Set Ecg-1.......................................... 50
26. Activation of Adaptive 1-12-1 Network for Ecg-1......................... 51
27. Ecg-1 and 1-12-1 Adaptive Network.............................................. 51
28. Ecg-1 and Hidden Layer of 1-12-1 Adaptive Network.................. 52
29. Ecg-1 and 1-6-1 Adaptive Network.................................................. 52
30. Ecg-1 and 1-8-1 Adaptive Network.................................................. 53
31. 1-6-1, and 1-8-1 Adaptive Networks.............................................. 53
32. A Data Set of 1000 Points (Ecg-2).................................................. 54
33. Activation of 1-11-1 Adaptive Network.......................................... 54
34. Ecg-2 and 1-11-1 Adaptive Network............................................... 55
35. A Three-Dimensional Hypercube.................................................... 60
36. The Communication Tree on a Three-Dimensional Hypercube Using Vertical Distribution......................................................... 61
37. Time Performance of the Parallel ABP on a 128-128-128-128 Network................................................................. 63
38. Time Performance of the Parallel ABP on a 128-128-128 Network... 64
39. Time Performance of the Parallel ABP on a 32-128-32 Network...... 65

vii

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
List of Figures—Continued

40. Time Performance of the Parallel ABP on the 128-128-128-128, 128-128-128, and 32-128-32 Networks................................................................. 65
CHAPTER I

INTRODUCTION

Neural Network Architectures

Neural cells, or neurons, represent the basic unit of the brain structure. One of the earliest computational models of the neuron was suggested by McCulloch and Pitts (1943). In this model, the neuron is considered as a triggering device with a threshold. When the sum of the input signals exceeds this threshold, an output with the value "true" is obtained; otherwise, the output is "false."

Figure 1. A Simplified Model of a Neuron.

Figure 1 illustrates a simplified neuron model; a weight \( w_i \) represents the connection strength from another neuron which has activation \( x_i \), where \( x_i \) is 1 (true) or 0 (false). \( \theta \) represents the threshold or bias. The activation function \( f \) is a step function which takes value 1 if the input variable is
positive, and takes value 0 if the input variable is not positive. The output of the neuron or unit is denoted with $I$, such that

$$I = f(w_1x_1 + w_2x_2 + w_3x_3, -\theta) = f\left(\sum_{i=0}^{n} w_ix_i\right)$$

with $w_0$ representing the bias $-\theta$ with the corresponding input $x_0$ set to 1. The McCulloch and Pitts model is highly simplified, yet it can be used to explain many neural functions using Boolean algebra. Each neuron has one output called an axon with many branches. The neurons are interconnected via special formations called synapses. Synapses are the points along the axon at which the communication with other neurons is achieved. Dendrites are connected to join other neurons at synapses, forming neural networks. The input of a neuron is obtained via dendrites. Thus, weights can be interpreted as dendrites.

The network-synaptic layer presents the basis for a "connectionist" theory of memory and intelligence as opposed to the classical structural theories of Artificial Intelligence. Many models have been developed within the frame of connectionist theory.

Today's research in neural computations is largely motivated by the possibility of making artificial computing networks. Yet, as the term "neural network" implies, some research is aimed at modeling networks of real neurons in the brain. The field of neural networks is also known as neurocomputation, associative networks, collective computation, and connectionism. The analytical superiority of the human brain over a digital computer is the chief motivation for studying neural computation. Artificial computation is inspired by the results of neuroscience and cognitive psychology. However, brain modeling as a separate area of study is
concerned with the models of real brains as opposed to artificial computational neural networks.

There are many brain characteristics which should be considered in the formulation of different models as suggested by Rumelhart and McClelland (1987). There are a few that were taken most seriously and which have most affected our thinking. These are discussed below.

**Neurons are slow.** Neurons are much slower than conventional (electronic) computational components. It seems unlikely that a single neuron computes a function much more complex than a single instruction in a digital computer. Thus, the mechanisms of mind are most likely best understood as resulting from the cooperative activity of very many relatively simple processing units operating in parallel.

**There is a very large number of neurons.** Another important aspect of brain-style processing is the very large number of processing units involved. Moreover, each neuron is an active processing unit. This suggests parallelism on a very large scale.

**Neurons receive inputs from a large number of other neurons.** This suggests that human computation does not involve a kind of logic circuit, but that it involves a kind of statistical process in which the single units do not make decisions, but in which decisions are the product of the cooperative action of many somewhat independent processing units. This degree of connectivity is very large in contrast to the number of immediate neighbors of processors in current parallel computers.

**Learning involves modifying connections.** Another key assumption of the models which derive from our understanding of learning mechanisms
in the brain is that the knowledge is in the connections rather than in the units themselves.

Connections in the brain have a topological structure. In general it seems that nearby regions in one part of the brain map onto nearby regions in another part of the brain. Moreover, there is a general symmetry of connections. This is the defining feature of interactive models; i.e., if there is a connection from one unit (region of the brain) to another, there is usually connection in the reverse direction.

Distributed, not central control. It seems that all regions of brain work together, influencing one another, and each region contributes to the overall performance of the task.

Relaxation is the dominant mode of computation. All of the features discussed above lead us to believe that the primary mode of computation in the brain is best understood as a kind of relaxation system in which the computation proceeds by iteratively seeking to satisfy a large number of weak constraints. The system can be thought of more as settling into a solution than calculating a solution.

Around 1960, researchers were focusing on the problem of collective computation by the networks on interrelated units and on the problem of how to weight the effect of the input on the output of the processing element. They concentrated on networks called perceptrons (see Figure 2), in which the units were organized into layers with feed-forward connections between one layer and the next. For the simplest class of perceptrons without any intermediate layers (original definition), Rosenblat (1962) was able to prove...
the convergence of a learning algorithm, a way to change the weights iteratively, so that a desired computation was performed.

Figure 2. A Simple Perceptron.

Perceptrons have largely been used to analyze neural networks. A given perceptron may contain several association layers. The associative units represent points where the flow of information through the network is controlled. Multilevel perceptrons are also called parallel distributed processing systems (PDP). In PDP systems, the programs and the data patterns themselves are not stored. Rather, what is stored is the connection strengths between units that allow these patterns to be recreated. Learning or building the knowledge structure in PDP systems involves modifying the patterns of interconnectivity.
In supervised learning (learning with a teacher) the network has its output, \( O = [O_1, O_2, O_3, ..., O_m] \) which is equivalent to the activation of the last layer units denoted with \( I^P \) where
\[
I^P = [I_1^P, I_2^P, I_3^P, ..., I_m^P],
\]
and there are \( P \) layers, compared with known correct answers,
\[
\zeta = [\zeta_1, \zeta_2, \zeta_3, ..., \zeta_m],
\]
and receives feedback about any errors. The layer that receives input is denoted as the 0\(^{th} \) layer, and the task of its units is to propagate signals forward without any computation. In unsupervised learning the network must discover for itself interesting categories or features in the input data. The networks usually have separate inputs and outputs, and assume that there is a training set of correct input-output pairs as examples. We use \( \xi^\mu \) to denote input of the \( \mu \)\(^{th} \) pattern,
\[
\xi^\mu = [\xi_1^\mu, \xi_2^\mu, \xi_3^\mu, ..., \xi_n^\mu]
\]
and \( \zeta^\mu \) to denote output of the \( \mu \)\(^{th} \) pattern
\[
\zeta^\mu = [\zeta_1^\mu, \zeta_2^\mu, \zeta_3^\mu, ..., \zeta_m^\mu],
\]
where \( \mu = 1, 2, 3, ..., M \). The network output is compared to the correct output, and the connection strengths \( w_{ij} \) are changed to minimize the difference.

The Three Types of Commonly Used Neural Networks

There are several major architectures driving the design and application of artificial neural networks. These include Hopfield Networks (Hopfield, 1982), Kohonen Networks (Kohonen, 1987), Boltzmann Machines (Hinton and Sejnowski, 1986), and Back Propagation (Rumelhart, Hinton, and Williams, 1986). Many extensions and modifications have been developed.
based on these models. Different learning rules to modify the pattern of connectivity as a function of experience have been developed. These include: the Hopfield minimum-energy rule, the Delta Rule with a Teacher, and the Boltzmann Learning Algorithm. In order to explain these three networks we first provide some definitions.

Definitions

Adaptive coefficient. Weighting value associated with each input to a processing element. It gates or weights the effect of that input on the output of the processing element. Adaptive coefficients can be self-adjusting; that is their values can be self-modified in response to external input. The process of self-adjusting is called learning.

Artificial Intelligence (AI). The study to make computers more useful by performing tasks that, until recently, only humans could perform. AI addresses the methods and concepts of symbolic inference by a computer.

Distributed Memory. The independent memory of each processor in parallel systems. This allows each processor to work on a small portion of the overall computational problem thus distributing the load.

Energy Function. In many fields there is a state function that decreases during dynamic evolution, or that must be minimized to find a stable or optimum state. The most general name, from the theory of optimization, is cost function or objective function. For neural networks in general an energy function exists if the connection strengths are symmetric, i.e. \( w_{ij} = w_{ji} \).

Feed-Forward Networks. A layered architecture where no units in the same layer are connected, and only the connections between two successive
layers exist. Each unit in a layer receives the input from all the units in the preceding layer. The input layer consists of units whose only role is to feed input patterns into the rest of the network. After this come one or more intermediate layers of units, often called hidden layers. The output layer follows hidden layers or the input layer, and the activation of units in the output layer represents the result of the computation.

**Hidden Unit.** See Feed-Forward Networks.

**Hypercube.** A communication network for connecting a collection of microcomputers (nodes) together. The dimension, \(d\), of the hypercube represents the number of directly connected nodes. For example in a five-dimensional cube, each node is connected to its five nearest neighbors. Such a cube would have \(2^5 = 32\) total nodes. Nodes communicate by passing a message.

**Input Data.** Any item used to drive the state of the network. It can be a binary or gray-scale value that describes the external process on which network is operating.

**Input Pattern.** A collection of input data items that is sent to the neurocomputer to act as the external stimulus to the network.

**Learning Rule.** The equation specifying how the adaptive coefficients or weights are self-modified. Usually a first order, ordinary nonlinear differential or difference equation.

**Machine Learning.** Research that seeks to create computer programs that can learn from experience. It includes formation of general rules, adjustment of coefficients of decision function, and the discovery of heuristic rules.
Neural Network. A cognitive information processing structure based on models of brain function. In a more formal engineering context, a highly parallel dynamic system with the topology of a directed graph that can carry out information processing by its state response to continuous or initial input.

Parallel Processing. The operation of a computer in which several programs or several copies of the same program are executed concurrently as opposed to serially. Also the data could be distributed between a large number of independent processors that can function on different data simultaneously.

Perceptron. A network made up of three types of units: sensory units, associative units, and response units. The sensory units correspond to the initial sensory input. The associative units represent points where the flow of information through the network is controlled. The response units represent the output information from the network.

Processing Unit. The fundamental computational unit in a neural network. A network consists of a large collection of highly interconnected processing units. A processing unit is composed of a number of input values from other processing units. These are weighted by a set of adaptive coefficients and then used to generate a single output value that branches to form input to other processing units.

Weight. An adaptive coefficient that can be adjusted in response to external input.
The Delta Rule or Widrow-Hoff Rule With a Teacher

The learning procedure proposed by the Delta Rule involves the presentation of a set of pairs of input and output patterns. The input vector is used to produce its own output vector. If there is no difference between the desired output and the actual output, no learning takes place. Otherwise the weights are changed to reduce the difference (see Figure 3). If we use $w_{ij}(t+1)$ to denote the updated value of weight $w_{ij}$, and $w_{ij}(t)$ as the old value of $w_{ij}$, then $\Delta w_{ij}(t+1)$ represents the change such that

$$w_{ij}(t+1)=w_{ij}(t)+\Delta w_{ij}(t+1).$$

![Figure 3. A Schematic Representation of $w_{ij}$ in a Feed-Forward Network.](image)

The rule for changing weights is given by

$$\Delta w_{ij}=\eta[\xi_i - O_i]$$

i.e., the amount of learning is proportional to the difference between the actual activation achieved and the target activation $\zeta_j$ provided by the teacher, where $I_j$ is the output from the unit $j$, connected to the input of unit $i$ through the weight $w_{ij}$. The state of activation $\zeta_i$ is a function $f$ of the old state $\xi_i$ and of the net input. The net input is usually the weighted sum of all inputs to the unit. The output $I_i$ is a function $f$ of the activation state. The useful, frequently used $f$ function is the threshold function.
while a frequently used activation function is a sigmoid function

\[ f(x) = \begin{cases} 
1, & x \geq 0 \\
0, & x < 0 
\end{cases} \]

\[ f(\text{net}) = \frac{1}{1 + e^{-\text{net}}}, \]

where net is the weighted sum of all inputs to the unit. Using this learning rule, the system can learn to associate arbitrary input/output pairs and in this way can learn to compute arbitrary input/output mappings.

**The Hopfield Minimum-Energy Rule**

Hopfield introduced an interesting kind of network in which the units were always in one of two states: +1 or -1 (see Figure 4). Hopfield showed that if the units are symmetrically connected (i.e. \( w_{ij} = w_{ji}, w_{ii} = 0 \)) and if they are updated one at a time, each update reduces the value of a cost function which he called "energy" because of the analogy with physical systems. The updating rule is to switch each unit into whichever of its two states yields the lower total energy. \( S_i \) is used to denote a current state of the \( i^{\text{th}} \) unit. The new state of the \( i^{\text{th}} \) unit is given by

\[ S_i = \text{sgn}(\sum_j w_{ij}S_j) \]

where the sign function \( \text{sgn}(x) \) takes 1 for \( x \geq 0 \), and -1 for \( x < 0 \). The updating process is usually carried out by random selection of units (asynchronously).

The global "energy" of the system is defined as

\[ E = -\sum_{i \neq j} w_{ij}S_iS_j + \sum_i \theta_i S_i \]
where \( S_i \) is the state of the \( i \)th unit (-1 or +1), and \( \theta_i \) is a threshold. Parallel networks of this kind have been used to access content addressable memories. The goal is to have the network retrieve the "closest" (in terms of Hamming distance) pattern among the stored patterns, for a given input pattern. The Hopfield networks are also used to solve some of the optimization problems (e.g. the traveling salesman problem).

Figure 4. A Hopfield Network With Four Units.

**The Boltzmann Learning Algorithm**

The Boltzmann algorithm is designed for a machine with symmetrical connections. If we include hidden units in the Hopfield networks we get a Boltzmann machine (see Figure 5). Hidden units cannot be used as input or output units.

Figure 5. A Boltzmann Machine With Two Hidden Units (3 and 4).
The threshold in a Boltzmann machine is probabilistic:

\[ p_i(\Delta E_i) = \frac{1}{1 + e^{-\Delta E_i/T}} \]

where \( p_i \) is the probability for the \( i \)th unit to be in state 1; \( T \) is a parameter used to define the behavior of the activation function. The total input to the unit is

\[ \Delta E_i = \sum_j w_{ij} s_j. \]

The learning is supervised: the input units are exposed to a particular pattern, while the network relaxes into a state of low energy in which the output units have the correct values. Due to the symmetry, the energy gradient with respect to \( w_{ij} \) depends only on the behavior of the \( i \)th and \( j \)th units and not on the whole network. This fact helps in updating input, output, and hidden units.

**Back Propagation Learning Rule**

Various people have developed algorithms which work quite well for adjusting the weights connecting units in successive layers of multi-layer perceptrons. Many of these are variations of a strategy known as back propagation. It was first developed by Werbos (1974) and was shown able to solve many problems which the simple one-layer perceptrons could not. Much current activity is centered on back propagation and its extensions.

The back propagation algorithm is a method for changing the weights \( w_{ij} \) in any feed-forward network to learn a training set of input-output pairs of vectors \( \{\xi^\mu, \zeta^\mu\} \). The underlying strategy is simply gradient descent, as described below.
We define an error measure or cost function by

\[ E[w] = \frac{1}{2} \sum_{\mu,i} (\zeta_i^\mu - I_i^\mu)^2 = \frac{1}{2} \sum_{\mu,i} (\zeta_i^\mu - f(\sum_k w_k \xi_k^\mu))^2 \]

where \( w \) represents a matrix of weights

\[
\begin{bmatrix}
w_{10} & w_{11} & \cdots & w_{1n} \\
w_{20} & w_{21} & \cdots & w_{2n} \\
\vdots & \vdots & \ddots & \vdots \\
w_{m0} & w_{m1} & \cdots & w_{mn}
\end{bmatrix}
\]

This is smaller the better our \( w_{ik} \)'s are; \( E \) is always non-negative, and goes to a minimum as we approach a solution. Note that this cost function depends only on the weights \( w_{ik} \) and the problem patterns.

Given our error measure \( E[w] \), we can improve on a set of \( w_{ik} \)'s by moving "downhill" on the surface it defines in \( w \) space. Specifically, the usual gradient descent algorithm suggests changing each \( w_{ik} \) by an amount \( \Delta w_{ik} \) proportional to the gradient of \( E \) at the present location:

\[
\Delta w_{ik} = -\eta \frac{\partial E}{\partial w_{ik}}
\]

The cost function is just a quadratic form in the weights. In the subspace spanned by the patterns the surface is a parabolic bowl with a single minimum. Assuming that the pattern vectors are linearly independent, the minimum is at

\[
\frac{\partial E}{\partial w_{ij}} = 0.
\]
The gradient descent rule produces changes in the weight vectors
\[ w_i = [w_{i0}, w_{i1}, w_{i2}, \ldots, w_{in}] \]
only in the directions of the pattern vectors \( \xi^\mu \). Within the pattern subspace the gradient descent rule necessarily decreases the error if \( \eta \) is small enough, because it takes us downhill gradient direction. Thus, with enough iterations we approach the bottom of the valley arbitrarily closely, from any starting point. At the bottom of the valley is a minimum of the function \( E \). We say that \( E \) has a local minimum at a point \( w = w' \) if there exists a neighborhood \( N \) of \( w' \) such that \( E(w) \geq E(w') \) for each \( w \) belonging to \( N \). If there exists \( w' \) such that \( N \) coincides with the domain of \( E \), we say that \( E \) has the global minimum at \( w' \). Different methods have been developed to overcome the possibility of getting stuck in a local minimum, one of which is the method of momentum. Momentum or inertia is assigned to each weight \( w_{ij} \) in the network, so that it tends to change in the direction of the average downhill force that it feels. It is a fraction of the last weight update. Momentum is used as an accelerator of the training process. If we denote momentum with \( \rho \), then the momentum term is given by

\[ \rho \Delta w_{ij}(t), \text{ where } w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t+1) + \rho \Delta w_{ij}(t). \]

To illustrate the back propagation algorithm we consider a two-layer network such as that illustrated by Figure 6. Output units are denoted by \( O_k \), hidden units by \( H_i \), and the input terminals by \( \xi_{k}^\mu \), where \( \mu \) denotes \( \mu \)th pattern. Note that for a two-layer feed-forward network \( H_i \) is equivalent to \( \Gamma^1_i \) and \( O_i \) is equivalent to \( \Gamma^2_i \). There are connections \( w_{jk} \) from the inputs to the hidden units and \( w_{ij} \) from the hidden units to the output units. Note that the index \( i \) always refers to an output unit, \( j \) to a hidden unit, and \( k \) to an
input terminal. The inputs are always set to particular values. Input \( k \) is set to \( \xi_k^\mu \) where pattern \( \mu \) is being presented. The \( \xi_k^\mu \) can be binary (0/1), bipolar (±1), or continuous-valued.

![Diagram of a Two-Layer Feed-Forward Network](image)

Figure 6. A Two-Layer Feed-Forward Network.

Given pattern \( \mu \), hidden unit \( j \) receives a net input

\[ net_j^\mu = \sum_k w_{jk} \xi_k^\mu \]

where the thresholds are represented by \( w_{jk} \) with the corresponding input equal to 1, and produces hidden layer output

\[ H_j^\mu = f(\sum_k w_{jk} \xi_k^\mu). \]

It is normal to use a sigmoid function for the activation function \( f(\text{net}) \). Output unit \( i \) thus receives
\[ net_i^\mu = \sum_j w_{ij} H_j^\mu \]

and produces for the final output

\[ O_i = f(net_i^\mu) = f(\sum_j w_{ij} H_j^\mu) \]

The usual error measure or cost function

\[ E[w] = \frac{1}{2} \sum_{\mu i} (\xi_i^\mu - O_i^\mu)^2 \]

now becomes

\[ E[w] = \frac{1}{2} \sum_{\mu i} (\xi_i^\mu - f(\sum_j w_{ij} f(\sum_k w_{ik} \zeta_k^\mu)))^2 \]

This is a continuous, differentiable function of every weight, so we can use a gradient descent algorithm to make appropriate weight changes; i.e., modify weights to minimize our cost function. The activation function must be differentiable, and we normally want to saturate at both extremes.

Back propagation has been much studied in the past few years, and many extensions and modifications have been considered. The basic algorithm given above is exceedingly slow to converge in a multi-layer network, and many variations have been suggested to make it faster. Other goals have included avoidance of local minima, and improvement of generalization ability. There are many parameters one can consider varying within the general back propagation framework, including the architecture (number of layers, number of units per layer), the size and the nature of the training set, and the update rule.
The idea of Back Propagation appears to be widely used and accepted as the main representative model for Artificial Neural Networks. The principle is intuitive and it is relatively easy to implement. Despite learning complexity and serious drawbacks, Back Propagation has been applied in many scientific fields, including signal processing. The quasi-periodic and aperiodic continuous time domain signals are very difficult to analyze, but usually very important class of signals (e.g. Electrocardiogram signals; ECG).

Applications of Back Propagation in Signal Processing

Artificial neural networks are recognized as very effective tools for solving complex problems in Signal Processing. The Hopfield Network was used as an underlying model of the implementation which demonstrated that the electron trapping materials developed by Quantex can be used as erasable optical masks to develop high density modifiable interconnections (Jutamulia, Storti, Lindmayer, and Seiderman, 1991).

Neural networks based on Back Propagation have been used to analyze a complex simulated radar environment that contains noisy radar signals from different transmitters. The networks were used to classify, describe, and identify radar signals from different transmitters. An attempt was made to create a small data base by the neural network. It was envisioned as a potential transmitter identification tool (Anderson, Gately, Penz, and Collins, 1990).

Back propagation techniques have also been used to design an optically controlled neural network for nonlinear signal mapping. Neural network predictors have been used successfully in signal encoding and noise
reduction in linear systems and back propagation makes it possible to apply
the same methods to nonlinear signals (Rietman, Frye, and Wong, 1991).

Signal Terminology

The domain of a signal is the set of independent variables over which
the signal is defined. Usually, it is the time, over which we consider the
signals. The time domain may be discrete or continuous. Signals for which
there are values at a countable number of points (instances) are in the discrete
time domain. If a signal exists at any point in time we say that the signal
exists in the continuous time domain. Sampling is a process of measuring
signal values in the continuous time domain at the specific, usually regularly
spaced, points. The sampling interval is an interval between two points of the
domain where the signal was measured.

There are several different types of signal forms based on periodicity.
If a signal $s(t)$ exists at all times $t$, and there exists a $T$ such that $s(t)=s(t+T)$,
then the signal $s(t)$ is periodic, with period $T$. Quasi-periodic signals could
consist of a summation of harmonically unrelated periodic wave forms
(Shiavi, 1991). A signal is transient if its magnitude goes to zero after a short
period of time.

A Preview

The purpose of this study is to introduce a new variation of back
propagation which we call the Adaptive Back Propagation (ABP) learning
algorithm. It will be seen that it is computationally superior to the standard
Back Propagation algorithm.
Chapter II describes the ABP. The idea, here, is to introduce the adaptive activation function, characterized with its adaptive learning parameter. We illustrate the Back Propagation algorithm by considering a two-layer network.

Chapter III applies the ABP algorithm to simulations of ECG signals, where the results suggest a strong correlation between frequency of the signal and the role of hidden units.

Chapter IV constructs the parallel version of the ABP implemented on a nCUBE-2 supercomputer.
CHAPTER II

ADAPTIVE BACK PROPAGATION

Overview

There are still many difficulties associated with using Back Propagation during the training process. A method of choosing appropriate values of the learning rate and momentum for a particular problem is not known. The constant learning parameters in a complex problem space can either create oscillations if the parameters are relatively small, or get stuck in a local minima if the parameters are relatively large. Some proposed ideas to improve performance of Back Propagation include new hybrid architectures (Hartman, Keeler, and Kowalski, 1990), similar learning architectures with a new activation function (Moody and Darken, 1989), a new learning rate concept (Jacobs, 1988), and a new momentum concept (Minai and Williams, 1992).

The Delta-Bar-Delta algorithm (Jacobs, 1988) introduced a new approach in using the learning rate parameter. Each weight in the feedforward architecture has an associated learning rate parameter. It decomposes the learning process along the individual weight directions. However, the algorithm can make the search jump wildly and an individual learning rate can increase dramatically, creating an oscillation that is very difficult to stabilize. Back Propagation with Adaptive Decoupled Momentum (Minai and Williams, 1992) was introduced to address the problems of the Delta-Bar-Delta. It defines an individual momentum that is varied just like the learning
rate. Also, a ceiling is defined which would prevent momentum from an uncontrolled growth.

A combined learning procedure with Localized Receptive Fields (Moody and Darken, 1988) uses a two-step approach: self-organization of the individual units, and supervised fine-tuning. The more successful version, compared to the standard back propagation, needs more data points and uses Gaussian function (see Appendix A) as the activation function to achieve a satisfiable generalization. The proposed ideas of Radial Basis Function networks (Bishop, 1991), and similar hybrid architectures (Hartman, Keeler, and Kowalski, 1990) explore the idea of locally tuned units, the idea motivated by the overlapping receptive fields in a real nervous system. The complexity of such architectures makes them less desirable.

Motivation for Modifying Back Propagation

Numerous attempts to address the main drawbacks of Back Propagation have resulted in improved speed of convergence, avoidance of a local minima, modified basic architecture, new activation functions, and network analysis aimed towards understanding an individual unit's role in the collective computation of the network. However, some of the drawbacks that seem impossible to completely overcome, such as being stuck in a local minima, have inspired researchers to design hybrid neural network architectures.

The sigmoidal activation function used in the feed-forward neural networks exhibits threshold behavior but can be differentiated, which is necessary to generalize the Perceptron learning rule. The threshold behavior
makes the units less discriminating; i.e., the interval on which a unit activation is not considered either 1 or 0 is usually fixed during learning. Since the sigmoid is a continuous function we can define a threshold $T$ for which the unit is considered set (activation is 1). If we have a sigmoid, defined as $f(x) = \frac{1}{1+e^{-kx}}$, where $x$ is the network input to the unit with no bias, and parameter $k$ is the arbitrarily chosen constant, then constant $k$ defines the point where the unit reaches $T$. When $k$ gets bigger the interval $[-a, a]$, where $a$ is the network input for which unit reaches $T$, gets smaller (see Figure 7 for $k=1$, and $k=3$, where $T=0.9$).

![Sigmoidal Activated Unit](image)

**Figure 7.** A Sigmoidal Activated Unit for $T=0.9$, $k=1$, and $k=3$.

In this work a rational function, a new activation function, is proposed to improve performance of feed-forward neural networks. The Back Propagation method is also modified to accommodate an additional parameter per computational unit.
An efficient parallelization of the feed-forward networks with a high number of computational units would improve the network learning time and make them more attractive to complex signal processing, and other application areas. In this work a parallel Back Propagation algorithm for feed-forward neural networks on a hypercube is also designed, implemented and analyzed.

**Adaptive Activation Function**

A new activation function $f(x) = (1 + (\beta x)^2)^{-1}$ with its associated adaptive parameter $\beta$ is presented as a computationally more powerful function. The motivational principle of natural filtering is hidden in the parameter $\beta$. A computational unit is receptive in a certain range of the independent variable, regardless of the strength of influence outside the interval. Compared to the standard sigmoidal or hyperbolic tangent activation functions the existence of an activation interval is more discriminating than the threshold definition. For continuous activation functions we define a threshold value $F_l$ needed to construct the mappings for which the image vector components take value 0 or 1. In such cases the network input is determined by the inverse of an activation function. If we increase the network input more than necessary to produce $F_l$, the unit with sigmoidal activation function will always produce value $f(x)$ such that $F_l \leq f(x) < 1$. The unit with a new activation function $f(x) = (1 + (\beta x)^2)^{-1}$ will produce value $F_l \leq f(x) \leq 1$ only for a certain interval of the network input. If we change the network input value, so that it is no longer inside the defined network input interval, the unit will produce $f(x)$ such that $f(x) < F_l$. It is no longer important how big the network input is,
rather what is its actual value. Thus, an increase in the network input does not guarantee that the unit will stay active and insensitive to the input like the sigmoidal units.

By using a bias in a standard way (i.e., by introducing an additional weight with the corresponding input set to 1) the interval can be defined anywhere in the domain of the independent variable (see Figure 8). The width of an interval $y$, which is modified by varying $\beta$, defines a unit's sensitivity, or filtering capability (see Figure 9), and is defined as

$$y(f, \beta) = 2 \sqrt{\frac{1-f}{\beta^2 f}}$$

where $f$ is a given activation (e.g. $F_1$).

Figure 8. Bias Effect on the Adaptive Unit.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
If $y$ gets smaller the unit becomes more sensitive to the network input. It becomes computationally difficult to produce network input that will activate such a unit, especially in the context of the massive neural network architectures. Thus, the whole network becomes more discriminating as opposed to the network with sigmoidal activated units.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{interval_width_adaptive_unit.png}
\caption{Interval Width and the Adaptive Unit.}
\end{figure}

An interval defines a sub-domain of the independent variable for which $f(x) \geq F_1$, where $0 \leq F_1 \leq 1$. Let us consider an example where we want to observe the effect of parameter $\beta$ on the interval width $y$ for the unit's activation greater than 0.1

\[ f(x) = (1 + (\beta x)^2)^{-1} = (1 + \beta^2 x^2)^{-1} \text{ where } f(x) \geq 0.1 \]

For $|\beta| = 1$, we have the activation function $f(x) = (1 + x^2)^{-1}$, i.e. the width of the interval $y$ is 6, and for no bias $a = 3$, $-a \leq x \leq a$; any network
input x in the specified interval will cause the unit's activation to be greater than 0.1.

For $|\beta| < 1$, the width of the interval $y$ increases, i.e. $a$ gets bigger; for $|\beta| = 0.8$, and for no bias $a = 3.75$, $y = 7.5$, thus if $-a \leq x \leq a$ the unit's activation will be greater than 0.1.

![Graph of the adaptive activation function](image)

Figure 10. XOR Solution With the Adaptive Unit. $F_{11}=0.8 \& F_{10}=0.2$.

For $|\beta| > 1$, the width of the interval $y$ decreases, i.e. $a$ gets smaller; for $|\beta| = 1.5$, and no bias $a = 2$, $y = 4$, thus if $-a \leq x \leq a$ the unit's activation will be greater than 0.1.

The adaptive activation function is a rational function which is easier to evaluate than any exponential function. It is differentiable which allows us to apply Back Propagation, and it reaches its upper bound (unlike the sigmoid which approaches its upper bound only in the limit).
Let us consider the simplest case of the N-input parity function studied by Minsky and Papert (1969), i.e. the two-input XOR function (see Table 1).

Table 1

<table>
<thead>
<tr>
<th>XOR Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Figure 11. OR Solution With the Adaptive Unit. $F_{11}=0.75$ & $F_{10}=0.5$. 

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
It is well known that a simple perceptron cannot represent this mapping. However, it is easy to find the parameters for only one unit, activated by our adaptive \( f(x) \), that will do successful mapping. One way would be to define both weights to be 1, and the bias -1. In order to determine \( b \) we need to define the threshold values such that output is equal 1 if \( f(x) > F_{11} \), and output is equal 0 if \( f(x) < F_{10} \). The definition of interval depends on the level of discrimination we want to achieve. For example, the output is 1 if \( f(x) \geq 0.8 \), and the output is 0 if \( f(x) \leq 0.2 \). Let us define 0 when \( f(x) \leq 0.2 \), and 1 when \( f(x) \geq 0.8 \) then for \( \beta = 2 \) (see Figure 10) the computed output is given in Table 2.

Table 2
Calculated XOR Function With One Adaptive Unit

<table>
<thead>
<tr>
<th>( \xi_1 )</th>
<th>( \xi_2 )</th>
<th>( I_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0.2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0.2</td>
</tr>
</tbody>
</table>

Thus, we can obtain a very accurate solution to the XOR problem. It has to be noted that a solution for the corresponding OR problem (see Table 3) cannot be achieved with the same level of discrimination. The threshold values used to define 0 and 1 cannot be determined such that the difference \( F_{11} - F_{10} \) is arbitrarily big. This is a result of the bell-shape filtering property.
that \( f(x) \) exhibits. If we define 0 when \( f(x) \leq 0.5 \), and 1 when \( f(x) \geq 0.75 \), then for \( \beta = 11 \) (see Figure 11), the bias equal to -0.1, and both weights equal 0.05 the computed output is given in Table 4. In terms of Boolean Algebra we get a more powerful XOR function at the expense of a decreased ability to simulate the OR function.

Table 3

<table>
<thead>
<tr>
<th>( \xi_1 )</th>
<th>( \xi_2 )</th>
<th>( \zeta_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4

Calculated OR Function With One Adaptive Unit

<table>
<thead>
<tr>
<th>( \xi_1 )</th>
<th>( \xi_2 )</th>
<th>( I_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0.45</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0.77</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0.77</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0.77</td>
</tr>
</tbody>
</table>
Adaptive Back Propagation Algorithm

Based on the gradient descent rule, Adaptive Back Propagation is derived to utilize learning the adaptive parameter $\beta$ corresponding to the activation function during the training process. The modification proposed by this study does not include contemporary improvements of the standard Back Propagation, where the standard means constant learning rate $\eta$ and constant momentum $\rho$. It is an intention of this work to show that the basic algorithm with a new activation function results in a superior performance.

The gradient descent rule was used to derive a learning rule which will provide the learning of the newly introduced parameter $\beta$. By traveling in the direction opposite to the gradient, one descends fastest down the surface defined by objective $f(x)$.

$$v_{\text{min}} = -\frac{\nabla f}{\|\nabla f\|}$$

Rumelhart, Hinton, and Williams (1986) showed that

$$\Delta w_i = -\frac{\partial E}{\partial w_i},$$

where $E$ is objective error function defined before. Furthermore,

$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial \text{net}_i} \frac{\partial \text{net}_i}{\partial w_i},$$

Knowing that

$$E = \frac{1}{2} \sum_i (\zeta_i - l_i)^2,$$
we define the so-called unit deltas \( \delta_1 \) and \( \delta_2 \). The unit delta \( \delta_2 \) is introduced to derive a learning rule for newly defined parameter \( \beta \). \( f' \) denotes the first derivative of function \( f \).

\[
\delta_1 = -\frac{\partial E}{\partial net_i},
\]

\[
\delta_2 = -\frac{\partial E}{\partial \beta_i},
\]

for \( f(\beta_i net_i) \), and

\[
E_i = \frac{1}{2}(\zeta_i - f(\beta_i net_i))^2,
\]

\[
\delta_1 = -(\zeta_i - f(\beta_i net_i))(\beta_i net_i)/(1) f'(\beta_i net_i) \beta_i
\]

\[
= \beta_i f'(\beta_i net_i)(\zeta_i - f(\beta_i net_i))
\]

\[
\delta_2 = -(\zeta_i - f(\beta_i net_i))(\beta_i net_i)/(1) f'(\beta_i net_i) net_i
\]

\[
= net_i f'(\beta_i net_i)(\zeta_i - f(\beta_i net_i))
\]

As in the "standard" back propagation rule we use the chain rule to derive \( \delta_1 \), and \( \delta_2 \) for the preceding layers. This provides a recursive method for computing the deltas for all units in the network, and consequently the weight updates. \( I_i = f(net_i) \) is used to denote activation of the \( i \)th unit.

Therefore, to compute the deltas of the hidden units we use

\[
\delta_i = -\frac{\partial E}{\partial net_i} = -\frac{\partial E}{\partial I_i} \frac{\partial I_i}{\partial net_i} = -(\sum_k \frac{\partial E}{\partial net_k} \frac{\partial net_k}{\partial I_i}) f'(net_i),
\]
which yields

$$
\delta_{i1} = f'(\beta_i \text{net}_i) \beta_i \sum_k \delta_{ki} w_{ki}
$$

By using the same method we can compute $\delta_2$

$$
\delta_{i2} = f'(\beta_i \text{net}_i) \text{net}_i \sum_k \delta_{k2} w_{ki}.
$$

This study develops the Adaptive Back Propagation Algorithm the details of which follow: A network with $P$ layers (see Figure 12), with $I^p_i$ representing the output of the $i^{th}$ unit in the $p^{th}$ layer, and $w_{ij}^p$ the link from $I^{p-1}_j$ to $I^p_i$ is observed. $\beta_i^p$ determines the width of the interval of the $i^{th}$ unit in the $p^{th}$ layer, and $\xi_{i \mu}$ the $i^{th}$ input of the $\mu^{th}$ pattern.

![Figure 12. A Two-Layer Feed-Forward Network, $P=2$.](image-url)
The desired output of the $\mu^{th}$ pattern is represented by $\zeta_{ik}^\mu$. The biases were used as an additional weight to the unit, such that input of the line is always 1. There are two learning rates involved $\eta_1$ (for the weights), and $\eta_2$ (for the width). Error function $E$, an error measure, is defined as

$$E[w] = \frac{1}{2} \sum_{\mu,j} (\zeta_j^\mu - I_j^\mu)^2$$

The error function $E$ can be any other differentiable function that is minimized when its arguments are equal, where the arguments are calculated outputs and desired ones over all patterns.

The detailed procedure, inspired by Hertz, and Palmer (1991), consists of the following steps:

1. Initialize the weights and interval widths (for each unit) to random values

2. Pick a pattern $\zeta_{ik}^\mu$ and apply it to the input layer ($p=0$) such that

   $$I_k^0 = \zeta_k^\mu$$

3. Propagate signal forwards by using the rule

   $$I_i^p = f(\beta_i^p net_i^p) = f(\sum_j w_{ij}^p I_j^{p-1})$$

for all $i,p$, where $p=P-h$, $h=0,1,2,...,P-1$

4. Compute the deltas for the output layer

   $$\delta_{i1}^p = f'(\beta_i^{p,net_i^p})[\zeta_i^\mu - I_i^p]\beta_i^p = \frac{-2net_i^p[\zeta_i^\mu - I_i^p]}{(1+(\beta_i^{p,net_i^p})^2)^2}$$

   $$\delta_{i2}^p = f'(\beta_i^{p,net_i^p})[\zeta_i^\mu - I_i^p]net_i^p = \frac{-2\beta_i^{p,net_i^p}[\zeta_i^\mu - I_i^p]}{(1+(\beta_i^{p,net_i^p})^2)^2}$$

by comparing the calculated outputs $I_i^p$ with the expected ones $\zeta_{i1}^\mu$ for the
pattern \( \mu \) being considered

5. Compute the deltas for the preceding layers

\[
\delta_{f_1}^{p-1} = f'(\beta_i^{p-1}net_{i}^{p-1})\beta_i^{p-1}\sum_j w_{ji}^p \delta_j^p = \frac{-2net_{i}^{p-1}}{(1+(\beta_i^{p-1}net_{i}^{p-1})^2)^2} \sum_j w_{ji}^p \delta_j^p
\]

\[
\delta_{f_2}^{p-1} = f'(\beta_i^{p-1}net_{i}^{p-1})net_{i}^{p-1}\sum_j w_{ji}^p \delta_j^p = \frac{-2\beta_i^{p-1}}{(1+(\beta_i^{p-1}net_{i}^{p-1})^2)^2} \sum_j w_{ji}^p \delta_j^p
\]

for \( p=P-h, \ h=0,1,2,...,P-1 \)

6. Use

\[
\Delta w_j^p = \eta_i \delta_i^{p-1}
\]

\[
\Delta \beta_i^p = \eta_i \delta_i^p
\]

to do updating

\[
w_j = w_j + \Delta w_j
\]

\[
\beta_i = \beta_i + \Delta \beta_i
\]

7. Repeat the procedure for the next pattern \( \mu \)

Summary

In this chapter we introduced the Adaptive Back Propagation (ABP) learning algorithm, and a new activation function was proposed to improve performance of feed-forward neural networks. Some features of the associated adaptive parameter \( \beta \) were discussed. The outline of the algorithm was presented.

The next chapter will show how the ABP algorithm is applied to experimentally generated signals, and ECG signals. The role of the hidden layer units will be discussed.
CHAPTER III

APPLICATION OF ADAPTIVE BACK PROPAGATION

Introduction

The objective of this study is to determine the relationship between the size of the network and the particular patterns present in the training set (sampling signal) by taking the full advantage of the computationally more powerful activation function that uses newly introduced adaptive parameters $\beta_i$. By using the combination of computer simulations and analysis for a limited domain of activation functions and the adaptive architecture, some concrete numerical results were obtained which provide insight into process of fitting the data. This insight suggests an interesting avenue of research in the analysis of the task of learning a signal function with a neural network (Back Propagation).

Two back propagation simulators were used in implementing a proposed learning rules. Simulator-A is a sequential implementation in Turbo Pascal 6.0 programming language, for IBM PC computers under DOS operating system. IBM PC with Intel 80486/33 central processing unit, and DOS 5.0 was used to run the simulator.

Simulator-B is an implementation of the proposed parallel algorithm for adaptive back propagation. The simulator is written in the C programming language for an nCUBE-2 supercomputer, under the Vertex operating system. The user interfaces for both simulators allowed us to easily change the network architectures, initial conditions, and learning parameters.
Simulation Description

The testing data for Adaptive Back Propagation was generated by sampling signal $s(x)$ defined as

$$s(x) = \sin(x)\cos(2x) + \sin(5x)\cos(1.5x)\sin(x)$$

over the interval $[-6.5, 6.5]$ with the sampling interval of 0.1 (see Figure 12). The wave form is chosen to represent a class of signals that are difficult to decompose.

Two different feed-forward architectures are selected to illustrate the results obtained by using Adaptive Back Propagation: one with the 4 units in the hidden layer (1-4-1), and the other with the 8 units in the hidden layer (1-8-1). As described, the networks consist of only one hidden layer. The weights were initialized to random values from $[-0.5, 0.5]$, and the units' interval widths to random values from [0.9, 1.1]. Both learning rates were fixed at the beginning of the training process at $\eta_1 = 0.01, \eta_2 = 0.005$ and a momentum of 0.05. The small initial values contributed to the large training time: 50000 iterations on average. At the same time, it allowed us to run the training process without obligation to dynamically adjust the training parameters.

Several feed-forward architectures with the standard Back Propagation were compared with the adaptive architectures. Two of the networks had exactly the same structure as the adaptive counterparts. The other two networks had two hidden layers with 4 (1-4-4-1), and 8 (1-8-8-1) units in each layer. The training process involved training without any intervention, and training with occasional intervention to change the learning parameters. The training with intervention represents a method of avoiding local minima. For
various initial weight values the networks were trained 60000 iterations on average.

The signal data for both axes was normalized onto the interval [0, 1]. The training process for all compared networks was done by random pattern presentation.

**Random Versus Sequential Pattern Presentation**

A sequential pattern presentation is a way of presenting the patterns to the network in the same order in which they were loaded, or stored. In every iteration during the training process the patterns are presented in the same order. A random pattern presentation is a way of presenting the patterns to the network in the random order. In a single iteration each pattern has to be presented exactly once.

The way patterns are presented during the training process has a significant impact on the obtained results. Even though different learning environments could show different signs, the possible impact of sequential pattern presentation schemes will be the same. Therefore, a well defined training environment is presented. However, all of the previously presented results in the conducted experiments were obtained by training the neural network architectures with a random pattern presentation.

In order to compare the difference between two methods of pattern presentation the signal data defined in the simulation description was used. The initial values of the parameters were initialized to a small random values [-0.5, 0.5], and learning rate was set to 0.1 with the momentum 0.5. The pattern presentation was set to sequential.
Figure 13. The Response of 1-4-1 Network During Sequential Pattern Presentation for Signal \( s(x) \).

Figure 14. The Actual Response of 1-4-1 Network for Signal \( s(x) \).
After performing less than a hundred iterations the error measure showed that the patterns were learned almost perfectly. The network response for each pattern was recorded and Figure 13 shows the response. Nevertheless, when the parameters of the network were separately tested a totally different result was discovered (see Figure 14). The poor fit is a result of local tuning. Since the parameters of the network were modified after each pattern was presented, the network was able to dynamically adjust its parameters from a presented pattern to the next one. The amount of adjustment depends on learning rate, momentum, and the degree of similarity between patterns. Thus, the network did not learn any relationship between input and output parameters, rather it got trapped in the very localized optimization process.

Simulation Results

Adaptive Back Propagation proved to be computationally superior compared to standard Back Propagation. The error, number of units, and training time versus mapping ability were each compared. In all cases the standard Back Propagation produced a generally average amplitude over the entire domain. On the graphs, this property is represented by a nearly horizontal line over the domain.

For the small number of units the adaptive network 1-4-1 showed a meaningful performance. A certain number of components were mapped almost perfectly, while other values were averaged over the intervals inside the domain. In Figure 15 the dotted line represents the activation of the trained network, and the other one signal $s(x)$. 

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Figure 15. Output of 1-4-1 Feed-Forward Network Using Adaptive Back Propagation for Signal $s(x)$.

The detailed analysis of the hidden layer units offered an explanation of their role in the computation. Each one of four hidden units was centered at one of the distinct components. The frequency of the selected components was also matched. In Figures 16, 17, 18, and 19 the dotted line represents activation of one of the hidden units, whereas the full line shows signal $s(x)$. In Figure 18 the activation of all hidden units, the produced output, and signal $s(x)$ are shown. The adaptive network 1-4-4-1 converged apparently selecting different components than the adaptive network 1-4-1 (see Figure 21). The tendency to localize certain signal components with the distinctive role of hidden units strongly suggests correlation between a frequency of the signal and activation of the hidden units.
Figure 16. Activation of Unit One in the Hidden Layer of 1-4-1 Network, and Signal s(x).

Figure 17. Activation of Unit Two in the Hidden Layer of 1-4-1 Network, and Signal s(x).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Figure 18. Activation of Unit Three in the Hidden Layer of 1-4-1 Network, and Signal $s(x)$.

Figure 19. Activation of Unit Four in the Hidden Layer of 1-4-1 Network, and Signal $s(x)$. 

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Figure 20. Activation of the Hidden Units in Adaptive 1-4-1 Network, the Output of the Network, and Signal $s(x)$.

Figure 21. Sigmoidal vs. Adaptive Activation Function for 1-4-4-1 Network, and Signal $s(x)$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
In the following section as a consequence of the numerical results a method is defined to improve the network performance.

**The Method of One Hidden Layer**

The simulation results obtained suggests a strong correlation between signal frequency of the signal and the role of hidden units. The architectures with one hidden layer were considered. One part of the analysis focused on choosing the appropriate initial parameters which lead to an optimal solution in a small number of iterations. There are no substantial guidelines on how to choose the initial parameters for feed-forward networks with back propagation learning rule; however, it is recommended that the weights should be set to some small random values. Considering the wide range of possible initial configurations (parameters, initial weights) and network architectures that can be constructed to learn the mappings, it is practically impossible to choose initial parameter values well. If the training process starts "too far" from the optimal solution in the search space, it can either become stuck in a local minima, or it will require a very large number of iterations to converge.

![Feed-Forward Network (1-4-1) With One Hidden Layer.](Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.)
Modifications of the learning parameters during the training are almost inevitable. In the presence of a high frequency signal, the learning process becomes very sensitive. In order to achieve meaningful learning, the standard back propagation architectures have to have a large number of units, and more than one hidden layer.

Let us consider the architecture with only one hidden layer, as shown in Figure 22. Let each unit in a hidden layer "select" one of the signal components. Here, a single output unit does the summation of the hidden layer units' responses to a presented signal.

Moreover, it behaves as an inverter, because of the form of activation function. In Figure 23 the simplified case of the activations of one hidden unit

Figure 23. A Simplified Case of the Inverted Activation of Hidden Units by the Output Unit With the Adaptive Activation Function.
and one output unit (dotted line) illustrates the inverter behavior. Note that the output of the hidden unit is the input of the output unit.

A set of guidelines was established in order to effectively initialize the network parameters. Since the data is normalized, and the activation of a unit could be at most 1, the initial value for parameter $\beta$ is computed by solving $0.5 = (1 + (\beta x)^2)^{-1}$ for $\beta$. The data is normalized on the interval $[0,1]$ by performing the transformation

$$z_i = \frac{x_i - \min}{\max - \min}$$

where $z_i$ is the normalized data point; min and max are the minimum and maximum values of the normalized domain. The activation of 0.5 was arbitrarily chosen to compute the interval widths, because it seems to best reflect a given frequency.

Figure 24. Sine Function $\sin(5x)$.  

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
The motivation for choosing 0.5 is the fact that the average amplitude of an ideal signal (sine function) is 0.5 when normalized on the interval [0,1]. Moreover, in such a case the period could be best read off (see Figure 24 for the unnormalized sine function \( f(x) = \sin(5x) \), where the x-axis corresponds to the 0.5 line after the normalization).

For a given frequency we can easily compute the period by \( 1/f_r \), where \( f_r \) is the frequency. The argument of the activation function is set to \( T/2 \), where \( T \) is the period of the signal, because the activation function is symmetric with the interval width \( y=2x \). Thus, since

\[
f(x) = \frac{1}{1+(\beta x)^2}
\]

it follows

\[
\beta^2 = \frac{1-f(x)}{f(x)x^2} = \frac{1}{x^2}, \text{ which yields } \beta = \frac{1}{x} = \frac{2}{T} = 2f_r.
\]

Solving this equation for \( \beta \) allows us to choose the initial value for parameter \( \beta \). If the sum of the units' responses in the hidden layer is increased, the response of the output unit gets smaller unless the weights to the hidden units are proportionally decreased. The weights of the output unit are initialized to the small random values on the order \( 1/f_r \). Since the domain is normalized, the weights of the hidden units are initialized to the random values of the order of the sampling rate. The role of biases in the hidden layer is to allow the units to position themselves in the domain, and filter out a certain signal component. The biases should be initialized to the random values between 0 and 1, and should be multiplied by the sampling rate as a factor. This scaling-up is done because the initial dispersion of the location of the intervals of activity for the hidden units is correlated to the sampling rate.
The number of hidden units depends on the type of the expected signal; i.e., pulses, sine-like continuous wave form. However, for a known duration of the sampled signal $t_d$, in seconds, and the given frequency, the total number of units $U$

$$U = 4t_d f,$$

has provided good performance. The learning rates $\eta_1$, $\eta_2$, and momentums $\rho_1$, $\rho_2$ should be set to small values proportional to the frequency of the signal.

**ECG Analysis**

ECG data was used to test the method of one hidden layer. The data was provided by Kalamazoo Center for Medical Studies, as a part of ongoing Automatic Defibrillator Study. The data was collected by automatic defibrillator HS-1000 from the patients in a cardiac arrest. The sampling rate was 100 samples per second with an amplitude between -2.5mV and +2.5mV. The output was discretized with 256 distinct values, where 0 represented -2.5mV, and 255 represented +2.5mV. On average, the reviewed cases had 60 seconds of recorded data.

Because of its irregularity, the ECG data from a cardiac arrest was selected to test the proposed method. The data from 25, out of 152 available cases, was used to create sets between 500 and 1000 points. Even though not all of the recorded data for a given case was usable, 11 data sets with 1000 points, and 18 data sets with 500 points were generated.

The resulting mappings closely resembled the original data. The small learning rates ($\eta_1=0.005$, $\eta_1=0.005$) and momentums ($\rho_1=0.1$, $\rho_1=0.05$)
allowed careful adjustment of parameters. Since the initial parameters were in the vicinity of a "good" local minimum, the chances of being trapped were substantially decreased. In fact, 85% of the training tests were obtained in less than 8000 iterations with RMS < 0.001. For the cases when an initial frequency was considerably different, the networks tended to average the signal, which resulted in a poor mapping.

Two cases with 500 and 1000 data points are presented in the following figures. Figure 25 shows a 5 second data set Ecg-1. They indicate the superiority in goodness-of-fit for the Adaptive Back Propagation when compared to conventional back propagation. Also, they verify the efficacy of the algorithm proposed in the Method of One Hidden Layer.

![Normalized Five Second Data Set Ecg-1](image)

Figure 25. Normalized Five Second Data Set Ecg-1.
In the Figures 26, 27, and 28, we have the activation of a 1-12-1 network with the Adaptive Back Propagation.

Figure 26. Activation of Adaptive 1-12-1 Network for Ecg-1.

Figure 27. Ecg-1 and 1-12-1 Adaptive Network.
In the Figures 29, 30, and 31 we have the results of an attempt to learn Ecg-1 with 6, and 8 units in the hidden layer.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Figure 30. Ecg-1 and 1-8-1 Adaptive Network.

Figure 31. 1-6-1, and 1-8-1 Adaptive Networks.

In Figure 32 we have a data set of 1000 points, Ecg-2.
Figure 32. A Data Set of 1000 Points (Ecg-2).

An attempted learning of Ecg-2 is presented in Figures 33 and 34.

Figure 33. Activation of 1-11-1 Adaptive Network.
Summary

In this chapter we presented how the ABP algorithm is used in generated simulations and applied to ECG signals. Two simulators were defined, and several feed-forward architectures with the standard Back Propagation were compared with the ABP. The architectures with one hidden layer were analyzed, and the method of one hidden layer was developed and tested. The difference between random and sequential pattern presentation was also discussed.

Chapter IV will describe the parallel version of the ABP implementation on a nCUBE-2 supercomputer.
CHAPTER IV

PARALLEL PROCESSING AND BACK PROPAGATION

One of the main drawbacks of using back propagation with relatively large neural network architectures is slow convergence and large training time. Nevertheless, only massive neural systems will be able to take full advantage of its connectionist nature when solving complex tasks. The scaling factor or ability to use neural network architecture with millions of computational units is one of the main obstacles for proving the superiority of connectionist models over the classical AI methods. As the feed-forward architectures grow, in terms of number of units and number of layers, they become computationally extensive and almost impossible to handle by means of any sequential computer. The process of propagating error backwards and updating the parameters is the most expensive part of the algorithm, since $\delta_{1}$ and $\delta_{2}$ have to be computed for each unit in the network, before updating the corresponding parameters.

Over the past few years, research efforts have been directed toward developing algorithms to simulate neural networks on parallel computers. The interest has been in issues such as the time and space complexities of the algorithms, the inter-processor communication requirements, and the routing of data in the machine. The parallel machine is seen by many researchers as the perfect "neural engine" because of its fine grain structure.

In order to create a more powerful simulation we designed a parallel version of the Adaptive Back Propagation. With such a tool one can explore
larger architectures as well as the effect of scaling factor on connectionist models.

Adaptive Back Propagation in Parallel

There are several ways the algorithm can be parallelized. One of the ideas is to use what we call horizontal distribution in which each layer of the network is assigned to a separate processor. The learning procedure has to be modified to propagate error after all patterns are presented. This is necessary to create a pipeline and achieve a significant speedup. Otherwise, if we had to modify parameters after each presented pattern the algorithm would execute slower than the serial version, because of the generated communications. Usually, the network architectures have a small number of layers, and a large number of units per layer. A simulation of such architectures on a supercomputer with a large number of processors (more than four) would not take the advantage of the available computational power. Almost always one would have to develop some sort of a hybrid scheme in order to balance the work.

We decided on doing a vertical parallelization that is independent of the underlying parallel computer topology. The required communication among processors is high, making the performance of the algorithm very sensitive to the underlying topology. In that context, the performance analysis for a hypercube will follow in the implementation section.

In the vertical parallelization, each processor has the same number of layers. The units in each layer are distributed among processors such that each processor gets the same number of units. Each unit keeps its parameters,
the weights of the connections to the preceding layer, the unit's bias, and activation interval $\beta_i$. A simulation proceeds iteratively, where processors exchange information on each layer in a single iteration. As in the previous case, the parameters could be updated either after each pattern is presented or after all patterns are presented. In either case we can distinguish two phases: (1) network response, and (2) network update. During the network response phase the network output is calculated for a presented pattern. The network update phase uses the learning rule to update network parameters backwards, from the output layer.

As noted before, the number of layers in the feed-forward architecture includes input as well as the output layer. Thus, a network with three layers has one input, one output, and one hidden layer. In what follows we present the parallel algorithm in the form of an outline:

1. Distribute the network architecture
   1.1 Repeat steps 1.2-1.4 for each layer in the network
   1.2 Divide the number of units equally among processors
   1.3 Assign to the processors the corresponding initial values of network parameters (weights, biases, etc.)
   1.4 On each processor build the corresponding network architecture
2. Repeat steps 3-6 for each iteration, or until error is less than specified
3. Repeat steps 4-6 for each presented pattern
4. Calculate network response
   4.1 Repeat steps 4.2-4.3 for each layer (except input layer)
4.2 Each processor calculates its corresponding units' responses by means of the described learning procedure.

4.3 Each processor performs multinode accumulate on the units' responses.

5. Update the network parameters

5.1 Each processor calculates its units' deltas for the output layer.

5.2 Repeat steps 5.3-5.5 for each hidden layer.

5.3 Calculate partial units' deltas on each processor as described in the learning procedure (the weights are locally present, and the response of the preceding layer has been obtained in step 4).

5.4 Create a global summation by performing multinode accumulate, or single node accumulate followed by single node broadcast, or any other high-performance scheme.

5.5 Each processor updates its own parameters.

6. Each processor calculates its own response error, and a single processor is designated to perform a single node accumulate, followed by a single node broadcast (to make information about the error available to others).

In the steps 5.3 and 5.4 we need to define the partial sums used to calculate hidden layer's deltas $\delta_{1}^{p}$ and $\delta_{2}^{p}$. If we want to calculate a single unit delta $\delta_{ij}^{p}$ in layer $p$, we need to know all connections of that unit with layer $p+1$. Since we distributed the units in layer $p+1$, all the needed connections are also distributed. However, each processor has the response of all the units in the network (step 4.3), including the units in layer $p$. Therefore,
a partial sum for all units in layer p could be computed in parallel. By adding the same components of the summands from different processors we obtain the necessary sum to calculate deltas for layer p.

Implementation on a Hypercube

A parallel version of the adaptive back propagation was implemented on a nCUBE-2 supercomputer with 128 processors. Figure 35 shows a three-dimensional hypercube. The network update phase was performed after each pattern was presented. Since the available resources allowed so, the patterns were distributed to all processors, which decreased the amount of communications for one layer. Special attention was made to the required communications for each layer.

![Figure 35. A Three-Dimensional Hypercube.](image)

During the network response phase, each processor had to accumulate the responses of the other processors' units for the specified layer, in order to compute the activation of the next layer. This process was done by forming a tree-like structure on the hypercube (see Figure 36). The numbers associated with the links (0, 1, 2) illustrate the steps of output propagation on a three-
dimensional hypercube. After the second step, processor 0 has the output for the specified layer. The output is communicated to all processors by using the same scheme in reverse direction.

![Communication Tree](image)

Figure 36. The Communication Tree on a Three-Dimensional Hypercube Using Vertical Distribution.

The role of each processor in the communication tree is determined before the computation starts, locally on each processor. The idea was to reach the root of the tree where the needed information would be complete, then to propagate the same message to all nodes in the tree. All links in the tree are direct, i.e. only neighbors are connected.

After the process begins, each processor holds the output of its units for a given layer. If the dimension of the hypercube is \( d \), then we need \( d \) steps to reach the root of the tree, because the mapping of the tree in the hypercube is done by bisecting the hypercube. It is known that bisection of the \( d \) dimensional hypercube gives us two \( d-1 \) dimensional hypercubes. At each step, a child sends its response to the parent. After collecting the response from one of its children, the parent augments its response with the one
received. When the root is reached a complete response record is available. The responses of all units in a given layer are at one place. Then, the root sends the responses to all of its children. The responses are distributed the same way they were accumulated until all processors get the necessary information.

The number of generating messages in this scheme is at most \(2N-2\), where \(N\) represents the number of processors in the hypercube, because the implementation considers the possibility that some processors do not have any units in the given layer. This is important in the cases where the number of units per layer is not the same for all layers, and a number of units in the given layer is less than a number of processors. The total number of steps needed to complete the operation is \(2\lg(N)\). If the number of layers is \(P\), the total number of steps of communication to compute the network response is \(2(P-2)\lg(N)\). This is the actual cost of the operation if we assume that sending a message from one processor to another is more important by itself, rather than the actual size of the message. Nevertheless, the maximum size of a message is proportional to the maximum number of units among the hidden layers.

In the network update phase we use the similar scheme as in the network response phase. The way to handle communications remains the same, with the modifications in the performed operations at each step in the processors. The messages that get passed among the processors are all of the same size, proportional to the number of units in the preceding layer. Once the message arrives from a child to the parent, a parent processor does the summation of the deltas it holds and the deltas that it just received. If we use
L to denote the maximum number of units that a layer holds in the given architecture, then the cost of such operation is at most 2L (we have additional parameter $\beta_j$). The cost of complete operation is at most $2 \log(N)$ messages. We also include a computational delay equal to $2L \log(N)$, because the processors do not communicate until the partial summation is done. The total cost of updating the network for a given pattern is at most $2(P-2) \log(N) + C_d$, where $C_d = 2L(P-2) \log(N)$ is a computational delay.

![Graph](image)

Figure 37. Time Performance of the Parallel ABP on a 128-128-128-128 Network.

Several architectures of various sizes were used to assess the performance of our implementation (128-128-128, 128-128-128-128, 32-128-32). The patterns were generated randomly on the interval $[0,1]$. We chose the number of iterations that allowed us to measure the performance, because the running time was directly proportional ($k=0.97$) to the number of iterations. A feed-forward neural network has a perfect distribution if it has the same
number of units in each layer. Such architectures showed a very good parallelization due to the optimal work balance of processors. In the Figure 37, we have the architecture of 4 layers with 128 units in each layer. On the x-axis is a dimension of the used hypercube; y-axis represents the time needed to complete a training process such that 1 equals the time to complete a task using only one processor. Figure 38 shows the performance for the architecture of three layers with 128 units in each layer.

![Figure 38](image)

Figure 38. Time Performance of the Parallel ABP on a 128-128-128 Network.

For the architectures with a big difference in the number of units among layers (see Figure 39), the performance in the terms of speedup was not comparable to those with the perfect distribution. Figure 40 illustrates the performance of all three networks. If a layer $p$ has more units than the layer $p-1$, then the process of computing response for layer $p$ is faster compared to the situation where layer $p-1$ has the same number of units as layer $p$. 

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
However, the units in layer $p$ have to communicate their responses the same way, and spend the same amount of time.

Figure 39. Time Performance of the Parallel ABP on a 32-128-32 Network.

Figure 40. Time Performance of the Parallel ABP on the 128-128-128-128, 128-128-128, and 32-128-32 Networks.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Summary

This chapter gives the outline of a parallel implementation of the ABP algorithm, based on the method established in Chapter II. The comparative time performance on the several architectures of various sizes is discussed.
CHAPTER V

CONCLUSION

The Adaptive Back Propagation algorithm developed in this study results in a superior performance when compared to conventional back propagation. A new activation function with the corresponding adaptive parameter is a computationally more powerful function compared to the standard sigmoidal or hyperbolic tangent activation function. By using the combination of computer simulations and analysis some concrete numerical results were obtained providing insight into the process of fitting the data by the means of the ABP learning algorithm. The number of units in the Adaptive Back Propagation and training time versus mapping ability were compared to those of the standard Back Propagation. The simulation results suggested a strong correlation between frequency and the role of hidden units. The role of biases in the hidden layer is to allow the units to position themselves in the domain, and filter out certain signal components. The Method of One-Hidden layer was developed for the effective utilization of units in one-hidden layer networks. The number of hidden units depends on the type of the expected signal. The way patterns are presented during the training process has a significant impact on the obtained results.

The feed-forward neural networks with the perfect distribution of units can be parallelized much better than the architectures with a big difference in the number of units among layers (in terms of speedup).

67
Future work will include more extensive analysis of the utilization of hidden units as presented in the Method of One Hidden Layer. However, it was demonstrated that many of the issues can be effectively analyzed in a simple test domain. The proposed parallel Adaptive Back Propagation will be improved by decreasing the amount of required communications among processors, thus making a more efficient computation possible.
Appendix A

Gaussian Activation Function
\[ G(z) = \frac{1}{\sqrt{2\pi}} e^{-\frac{z^2}{2}}, \quad -\infty < z < \infty \]
Appendix B

Source Code in C for Adaptive Back Propagation on nCUBE-2
/*
  Pero Smrzlic (March, 1993)

  Part of the requirements for Master's Thesis in
  Computer Science, WMU - Winter 1993

  Parallel implementation of Adaptive Back Propagation learning
  algorithm for Feed-Forward Neural Networks (128-node nCUBE-2).
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/types.h>
#include <time.h>
#define BLOCK -1
#define Derivative(x) ((-2.0*x)/SQR(1.0+SQR(x)))
#define PACKET_SIZE 15000
#define Response(x) (1.0/(1.0+SQR(x)))
#define SQR(x) (x)*(x)

struct unit {
  double response, derivative,
           delta;
  double bheta, bheta_derivative,
           bheta_delta, bheta_Delta;
  double bias,
           bias_Delta;
  double *weights,
          *weights_Delta;
};

struct alpha_bheta {
  double alpha;
  double bheta;
};

struct pattern {
  double *input;
  double *output;
};

struct unit **network;
struct pattern *data;
struct alpha_bheta *layer_delta;
double **layer_resp;

double bheta_lr, bheta_mom,
epsilon, rms,
unit_lr, unit_mom;

long iterations=0, step=500;

int **cube_load,
*structure;

int cube_dim, cube_node,
layers, neurons=0, node_inv,
pattern_offset=0, points,
*recv, *send;

/* message types */
int act_acc=15, cfg_type=9,
deltas_type_acc=31, deltas_type_bcst=27,
eps_bcst=23, layer_prm_type=100,
patt_type=25, recv_n_node=BLOCK,
resp_type_acc=13, resp_type_bcst=17,
rqt_act_out=14, rqt_prm_out=22,
struct_type=8;

char out_prm_flag=0,
test_net_flag=0;

/*
Section I: Utility Routines
 */
int inverse(x_node)
int x_node;
{
    int j, num=0;
for (j=0; j<cube_dim; j++)
    num 1 = (((l«j)& x_node)»j)«(cube_dim-j-1);
return num;
}

void push(st, sp, x)
int *st, *sp, x;
{
    st[*sp] = x;
    (*sp)++;
}

int pop(st, sp)
int *st, *sp;
{
    (*sp)--;
    return st[*sp];
}

void partition(storage, stndx, init_s, init_v, comp_v)
int *storage, *stndx, init_s, init_v, comp_v;
{
    int sp=0, *stack, x;

    stack = (int *)malloc(init_s*sizeof(int));
    push(stack, &sp, init_v);
    while (sp>0)
        if (stack[sp-1]>comp_v) {
            x = pop(stack, &sp);
            push(stack, &sp, x-(x>>1));
            push(stack, &sp, x>>1);
        } else {
            x = pop(stack, &sp);
            storage[*stndx] = x;
            (*stndx)++;
        }
    free(stack);
}

int unit_offset(x_node, x, pp)
int x_node, x, *pp;
{
int offset=0, step, x_inv;

x_inv = inverse(x_node);
step = ((2<<x)-1)&x_inv;
while (step>0) {
    offset += pp[x_inv-step];
    step--;
}
return offset;

void collect_rms()
{
    double rms_rcvd=0.0;
    int dr=0, nr_ret, nw_ret;
    /* quasi - accumulate code for RMS */
    while (dr<cube_dim) {
        if (recv[dr] != BLOCK) {
            nr_ret = nread(&rms_rcvd, sizeof(double), &recv[dr],
                            &eps_bcst, 0);
            rms += rms_rcvd;
        } else if (send[dr] != BLOCK)
            nw_ret = nwrite(&rms, sizeof(double), send[dr],
                             eps_bcst, 0);
        dr++;
    }
    rms = sqrt(rms)/(double)points;
}

/*
Section II: Initialization
() init_rs: communication-tree mapping
() init_cfg: configuring the network
() init_prm: loading the parameters
() init_patt: loading the patterns
() init: init control
++++++++++++++++++++++++++
*/
void init_rs(cdim, cnode)
int cdim, cnode;
{
    char sendflag = 1;
int i, hypsum, hypcube, x;

recv = (int *)malloc( cdims);hypcube *sizeof(int));
send = (int *)malloc( cdims);hypcube *sizeof(int));

x = hypcube = 1 << cdims;
hypsum = cnode;
for (i = 0; i < cdims; i++) {
    x >>= 1;
    hypsum += x;
    if (hypsum < hypcube) {
        recv[i] = cnode+x;
        send[i] = BLOCK;
    } else if (sendflag) {
        send[i] = hypsum % hypcube;
        recv[i] = BLOCK; sendflag = 0;
    } else send[i] = recv[i] = BLOCK;
}

// establishing neighbour links (receiver, sender-list) */

x = cdims-1;
while (x > 0 && send[x] == BLOCK)
    x --;
if (x >= 0)
    recv_n_node = send[x];

void init_cfg(char *cfg_filename)
{  
double *cfg_record[7];
FILE *cfg_file;
int i, j, l_ndx, l_units, n_nodes, nr_ret, nw_ret,
    pp_comp, pp_n, src=0;

if (cube_node == 0) {
    cfg_file = fopen(cfg_filename, "r");
    fscanf(cfg_file, "%d", &layers);
    structure = (int *)malloc(layers*sizeof(int));
    for (i = 0; i < layers; i++) {
        fscanf(cfg_file, "%d %d", &l_ndx, &l_units);
        structure[l_ndx] = l_units;
    }
    fscanf(cfg_file, "%d", &points);

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
for (i=2; i<7; i++)
    fscanf(cfg_file, "%lf", &cfg_record[i]);
fclose(cfg_file);
if (cube_dim>0) {
    cfg_record[0] = (double)layers;
    cfg_record[1] = (double)points;
    nw_ret = nwrite(cfg_record, 7*sizeof(double), 0xffff,
        struct_type, 0);
    nw_ret = nwrite(structure, layers*sizeof(int), 0xffff,
        struct_type, 0);
}
} else {
    nr_ret = nread(cfg_record, 7*sizeof(double), &src, &cfg_type, 0);
    layers = (int)cfg_record[0];
    points = (int)cfg_record[1];
    structure = (int *)malloc(layers*sizeof(int));
    nr_ret = nread(structure, layers*sizeof(int), &src, &struct_type, 0);
}
unit_lr = cfg_record[2];
unit_mom = cfg_record[3];
bheta_lr = cfg_record[4];
bheta_mom = cfg_record[5];
epsilon = cfg_record[6];

/* init cube_load[] */
n_nodes = 1<<cube_dim;
cube_load = (int **)malloc(layers*sizeof(int *));
for (i=0; i<layers; i++) {
    neurons += structure[i];
    cube_load[i] = (int *)malloc(n_nodes*sizeof(int));
    if (i==0) {
        for (j=0; j<n_nodes; j++)
            cube_load[i][j] = structure[0];
    } else {
        pp_comp = structure[i]>>cube_dim;
        if ((pp_comp<<cube_dim)<structure[i])
            pp_comp++;
        pp_n = 0;
        partition(cube_load[i], &pp_n, cube_dim+2, structure[i],
            pp_comp);
        for (j=pp_n; j<n_nodes; j++)
            cube_load[i][j] = 0;
/* init network[] */
network = (struct unit **)malloc(layers*sizeof(struct unit *));

/* init response[] */
layer_resp = (double **)malloc(layers*sizeof(double *));
for (i=0; i<layers; i++)
    layer_resp[i] = (double *)malloc(structure[i]*sizeof(double));

void init_prm(prm_filename)
char *prm_filename;
{
    double *j_layer, *my_layer;
    FILE *prm_file;
    int i, j, j_inv, j_prms, k, layer_prms, my_prms, n_nodes,
        nr_ret, nw_ret, offset, src=0;

    n_nodes = 1<<cube_dim;
    layer_prms = layer_prm_type;
    if (cube_node==0)
        prm_file = fopen(prm_filename, "r");
    for (i=1; i<layers; i++) {
        my_prms = cube_load[i][node_inv]*(structure[i-1]+2);
        my_layer = (double *)malloc(my_prms*sizeof(double));
        if (cube_node==0) {
            for (k=0; k<my_prms; k++)
                fscanf(prm_file, "%lf", &my_layer[k]);
            for (j=1; j<n_nodes; j++) {
                j_inv = inverse(j);
                j_prms = cube_load[i][j]*(structure[i-1]+2);
                if (j_prms>0) {
                    j_layer = (double *)malloc (j_prms*sizeof
                                            (double));
                    for (k=0; k<j_prms; k++)
                        fscanf(prm_file, "%lf", &j_layer[k]);
                    nw_ret = nwrite(j_layer, j_prms*sizeof
                                (double), j_inv, layer_prms, 0);
                    free(j_layer);
                }
            }
        }
    }
}
else if (my_prms>0)
    nr_ret = nread(my_layer, my_prms*sizeof(double), &src,
                    &layer_prms, 0);

network[i] = (struct unit *) malloc (cube_load[i][node_inv]*
                             sizeof(struct unit));

for (j=0; j<cube_load[i][node_inv]; j++) {
    offset = j*(structure[i-l]+2);
    network[i][j].btheta = my_layer[offset++];
    network[i][j].bias = my_layer[offset++];
    network[i][j].weights = (double *) malloc(structure[i-l]*
                                           sizeof(double));
    network[i][j].weights_Delta = (double *) malloc
                                   (structure[i-l]*sizeof(double));
    for (k=0; k<structure[i-l]; k++) {
        network[i][j].weights[k] = my_layer[offset+k];
        network[i][j].weights_Delta[k] = 0.0;
    }
    network[i][j].bias_Delta = network[i][j].btheta_Delta = 0.0;
}
free(my_layer); layer_prms++;
}

if (cube_node==0)
    fclose(prm_file);

void init_patt(patt_filename)
char *patt_filename;
{
    double *buffer;
    FILE *patt_file;
    int buff_elem, buff_points, buff_ptr, elem_read=0, i, j, k,
        msg_cnt, msg_points, n_packets, nw_ret, patt_id=0,
        patt_size, src=0, tot_elem;

    patt_size = structure[0]+structure[layers-1];
    n_packets = PACKET_SIZE/(patt_size*sizeof(double));
    if (n_packets>points)
        buff_points = points;
    else buff_points = n_packets;
    buff_elem = buff_points*patt_size;
    msg_cnt = points/buff_points;
    if ((points%buff_points)!=0)
        msg_cnt++;
    }
data = (struct pattern *)malloc(points*sizeof(struct pattern));
buffer = (double *)malloc(buff_elem*sizeof(double));
tot_elem = points*patt_size;
if (cube_node==0)
    patt_file = fopen(patt_filename, "r");
for (i=0; i<msg_cnt; i++) {
    if (cube_node==0) {
        for (j=0; j<buff_elem && elem_read<tot_elem; j++,
            elem_read++)
            fscanf(patt_file, "%lf", &buffer[j]);
        msg_points = j*sizeof(double);
        if (cube_dim>0)
            nw_ret = nwrite(buffer, msg_points, 0xffff,
                patt_type, 0);
    } else msg_points = nread(buffer, buff_elem*sizeof(double),
            &src, &patt_type, 0);
        msg_points /= (patt_size*sizeof(double));
    for (j=buff_ptr=0; j<msg_points; j++, patt_id++) {
        data[patt_id].input = (double *)malloc(structure[0]*sizeof(double));
        data[patt_id].output = (double *)malloc(structure[layers-1]*sizeof(double));
        for (k=0; k<structure[0]; k++, buff_ptr++)
            data[patt_id].input[k] = buffer[buff_ptr];
        for (k=0; k<structure[layers-1]; k++, buff_ptr++)
            data[patt_id].output[k] = buffer[buff_ptr];
    }
}
free(buffer);
if (cube_node==0)
    fclose(patt_file);}

void init(group_name)
char *group_name;
{
    char *filename;
    int i=0, max=0;

cube_dim = cubedim();
cube_node = mynode();
node_inv = inverse(cube_node);
init_rs(cube_dim, cube_node);
filename = malloc(strlen(group_name)+6);
strcpy(filename, group_name);
strcat(filename, ".CFG");
init_cfg(filename);

strcpy(filename, group_name);
strcat(filename, ".PRM");
init_prm(filename);

strcpy(filename, group_name);
strcat(filename, ".PATT");
init_patt(filename);
free(filename);

if (cube_dim>0)
    pattern_offset = unit_offset(cube_node, cube_dim-1,
        cube_load[layers-1]);
for (i=0; i<layers; i++)
    if (structure[i]>max)
        max = structure[i];
layer_delta = (struct alpha_bheta *)malloc(max*sizeof(struct
    alpha_bheta));

Section III: Output/Results
()
out_prm: output trained parameters
()
test_net: test the network output for a given pattern
+++++++++++++++++++++++++++++++++++*/

void out_prm(out_filename)
char *out_filename;
{
    char *request;
double *j_layer, *my_layer;
FILE *out_file;
int i, j, j_inv, j_prms, k, layer_prms, my_prms, n_nodes,
    nr_ret, nw_ret, offset, src=0;

    n_nodes = 1<<cube_dim;
layer_prms = layer_prm_type;
if (cube_node==0)
out_file = fopen(out_filename, "w");
for (i=1; i<layers; i++) {
    my_prms = cube_load[i][node_inv]*(structure[i-1]+2);
    my_layer = (double *)malloc(my_prms*sizeof(double));
    for (j=0; j<cube_load[i][node_inv]; j++) {
        offset = j*(structure[i-1]+2);
        my_layer[offset] = network[i][j].beta;
        my_layer[offset + 1] = network[i][j].bias;
        for (k=0; k<structure[i-1]; k++)
            my_layer[offset+k] = network[i][j].weights[k];
    }
    if (cube_node==0) {
        for (k=0; k<my_prms; k++) {
            fprintf(out_file, "%lf", my_layer[k]);
            if ((k%(structure[i-1]+2))!=0)
                fprintf(out_file, "\n");
        }
    }
    for (j=1; j<n_nodes; j++) {
        j_inv = inverse(j);
        j_prms = cube_load[i][j]*(structure[i-1]+2);
        if (j_prms>0) {
            j_layer = (double *)malloc(j_prms*
                sizeof(double));
            nw_ret = nwrite(request, 0, j_inv,
                rqt_prm_out, 0);
            nr_ret = nread(j_layer, j_prms*sizeof
                (double), &j_inv, &layer_prms, 0);
            for (k=0; k<j_prms; k++) {
                fprintf(out_file, "%lf", j_layer[k]);
                if ((k%(structure[i-1]+2))!=0)
                    fprintf(out_file, "\n");
            }
            free(j_layer);
        }
    }
} else if (my_prms>0) {
    nr_ret = nread(request, 0, &src, &rqt_prm_out, 0);
    nw_ret = nwrite(my_layer, my_prms*sizeof(double), src,
        layer_prms, 0);
}
free(my_layer); layer_prms++;
if (cube_node==0)
fclose(out_file);

void test_net(out_filename)
char *out_filename;
{
char *request;
double *buffer, **responder;
FILE *out_file;
int buff_points, buff_ptr, elem_read=0, i, j, j_inv, j_prms, k,
msg_cnt, msg_points, n, n_end, n_nodes, n_packets,
rn_ret, nw_ret, patt_id=0, src=0;
void net_response();

n_nodes = 1<<cube_dim;
n_packets = PACKET_SIZE/(structure[layers-1]*sizeof(double));
if (n_packets>points)
  buff_points = points;
else buff_points = n_packets;
msg_cnt = points/buff_points;
if ((points%buff_points)!=0)
  msg_cnt++;
buffer = (double *)malloc(buff_points*structure[layers-1]*
sizeof(double));
if (cube_node==0) {
  out_file = fopen(out_filename, "w");
  responder = (double **)malloc(n_nodes*sizeof(double *));
  responder[0] = buffer;
}
for (i=0; i<msg_cnt; i++) {
  buff_ptr = 0;
  for (j=0; j<buff_points && elem_read<points; j++,
  elem_read++) {
    net_response(patt_id);
    for (k=0; k<cube_load[layers-1][node_inv]; k++,
      buff_ptr++)
      buffer[buff_ptr] = network[layers-1][k].response;
    patt_id++;
  }
  msg_points = j;
  if (cube_node==0) {
    for (j=1; j<n_nodes; j++) {
      j_inv = inverse(j);
    }
  }
}
void exchange_response(buffer, buff_init, buff_limit)
{ double *buffer;
int buff_init, buff_limit;

  int dr=0, nr_ret, nw_ret;
/* quasi - accumulate code for LAYER RESPONSE */
while (dr<cube_dim) {
    if (recv[dr]!=BLOCK && buff_init<buff_limit) {
        nr_ret = nread(&buffer[buff_init], neurons*
        sizeof(double), &recv[dr], &resp_type_acc, 0);
        buff_init += (nr_ret/sizeof(double));
    } else if (send[dr]!=BLOCK && buff_init>0)
        nw_ret = nwrite(buffer, buff_init*sizeof(double),
                        send[dr], resp_type_acc, 0);
    dr++;
}

if (recv_n_node != BLOCK) {
    buff_init = nread(buffer, neurons*sizeof(double), &recv_n_node,
                      &resp_type_bcst, 0);
    buff_init /= sizeof(double);
}

/* quasi - broadcast code for LAYER RESPONSE */
dr = cube_dim-1;
while (dr>=0) {
    if (recv[dr] != BLOCK)
        nw_ret = nwrite(buffer, buff_init*sizeof(double), recv[dr],
                        resp_type_bcst, 0);
    dr--;
}

void exchange_delta(layer_id, deltas)
    int layer_id;
    struct alpha_bheta *deltas;
{
    int dr=0, i, nr_ret, nw_ret, snd_ndx, snd_size;
    struct alpha_bheta *rcvd_delta;

    rcvd_delta = (struct alpha_bheta *) malloc (structure[layer_id]*
              sizeof(struct alpha_bheta));
    if (cube_load[layer_id+1][node_inv]<1)
        snd_size = 0;
    else snd_size = structure[layer_id];
    /* quasi - accumulate code for LAYER DELTAS */
    while (dr<cube_dim) {
        if (recv[dr] != BLOCK) {
nr_ret = nread(rcvd_delta, structure[layer_id]*
sizeof(struct alpha_bheta), &recv[dr],
&deltas_type_acc, 0);
if (nr_ret>0) {
    snd_size = structure[layer_id];
    for (i=0; i<snd_size; i++) {
        deltas[i].alpha += rcvd_delta[i].alpha;
        deltas[i].bheta += rcvd_delta[i].bheta;
    }
}
else if (send[dr] != BLOCK)
    nw_ret = nwrite(deltas, snd_size*sizeof(struct
    alpha_bheta), send[dr], deltas_type_acc, 0);
    dr++;
}
if (recv_n_node != BLOCK) {
    nr_ret = nread(deltas, structure[layer_id]*sizeof (struct
    alpha_bheta), &recv_n_node, &deltas_type_bcst, 0);
    snd_size = nr_ret/sizeof(struct alpha_bheta);
} else snd_size = structure[layer_id];
/* quasi - broadcast code for LAYER DELTAS */
    dr = cube_dim-1;
while (dr>=0) {
    if (recv[dr] != BLOCK) {
        snd_ndx = unit_offset(recv[dr], dr, cube_load[layer_id]);
        snd_size -= snd_ndx;
        nw_ret = nwrite(&deltas[snd_ndx], snd_size*sizeof(struct
        alpha_bheta), recv[dr], deltas_type_bcst, 0);
    }
    dr--;
}
free(rcvd_delta);

void net_response(pattern_id)
int pattern_id;
{
    double h;
    int i, j, k;
    /* init layer 0; globally */
    for (i=0; i<structure[0]; i++)
layer_resp[0][i] = data[pattern_id].input[i];

/* propagate signal forward */
for (i=1; i<layers; i++) {
    /* node calculation */
    for (j=0; j<cube_load[i][node_inv]; j++) {
        h = network[i][j].bias;
        for (k=0; k<structure[i-1]; k++)
            h += layer_resp[i-1][k]*network[i][j].weights[k];
        network[i][j].response = layer_resp[i][j] =
            Response(h*network[i][j].btheta);
        network[i][j].derivative =
            Derivative(h*network[i][j].btheta);
        network[i][j].btheta_derivative =
            network[i][j].derivative*h;
        network[i][j].derivative *= network[i][j].btheta;
    }
    /* layer response distribution */
    if (i<(layers-1))
        exchange_response(layer_resp[i], cube_load[i][node_inv],
                           structure[i]);
}

void layer_update(layer_id)
int layer_id;
{
    double delta_change;
    int i, j;

    for (i=0; i<cube_load[layer_id][node_inv]; i++) {
        for (j=0; j<structure[layer_id-1]; j++) {
            delta_change = unit_lr*network[layer_id][i].delta*
                layer_resp[layer_id-1][j];
            network[layer_id][i].weights[j] += delta_change+
                unit_mom*network[layer_id][i].weights_Delta[j];
            network[layer_id][i].weights_Delta[j] = delta_change;
        }
        delta_change = unit_lr*network[layer_id][i].delta;
        network[layer_id][i].bias += delta_change+unit_mom*
            network[layer_id][i].bias_Delta;
        network[layer_id][i].bias_Delta = delta_change;
        delta_change = btheta_lr*network[layer_id][i].btheta_delta;
        network[layer_id][i].btheta += delta_change+btheta_mom*
void back_propagation(pattern_id)
{
    int pattern_id;
    double difference;
    int i, j, k;

    /* deltas for the last layer */
    for (i=0; i<cube_load[layers-1][node_inv]; i++) {
        difference = data[pattern_id].output[pattern_offset+i] -
                      network[layers-1][i].response;
        network[layers-1][i].delta = network[layers-1][i]
            .derivative*difference;
        network[layers-1][i].bheta_delta = network[layers-1][i]
            .bheta_derivative*difference;
        rms += 0.5*SQR(difference);
    }

    /* deltas for the preceding layers */
    for (i=layers-1; i>1; i--) {
        for (k=0; k<structure[i-1]; k++) {
            layer_delta[k].alpha = layer_delta[k].bheta = 0.0;
            for (j=0; j<cube_load[i][node_inv]; j++) {
                layer_delta[k].alpha += network[i][j].weights[k] *
                      network[i][j].delta;
                layer_delta[k].bheta += network[i][j].weights[k] *
                      network[i][j].bheta_delta;
            }
        }
        exchange_delta(i-1, layer_delta);
        for (j=0; j<cube_load[i-1][node_inv]; j++) {
            network[i-1][j].delta = network[i-1][j].derivative *
                layer_delta[j].alpha;
            network[i-1][j].bheta_delta = network[i-1][j]
                .bheta_derivative*layer_delta[j].bheta;
        }
        layer_update(i);
    }

    layer_update(1);
}
void simulate()
{
    char cont=1;
    int i, nr_ret, nw_ret, patt_cnt, *patt_ndx, rnd_val, src=0;
    long cnt=0, seed_val=19;

    patt_ndx = (int *)malloc(points*sizeof(int));
    for (i=0; i<points; i++)
        patt_ndx[i] = -1;
    srand48(seed_val);
    while (cont && cnt<iterations) {
        patt_cnt = 0;
        for (i=0; i<points; i++)
            rnd_val = (int)(drand48()*(double)points);
            if (patt_ndx[rnd_val]<0) {
                patt_ndx[rnd_val] = patt_cnt;
                patt_cnt++;
        }
        i = points-1;
        while (i>=0 && patt_cnt<points) {
            if (patt_ndx[i]<0) {
                patt_ndx[i] = patt_cnt;
                patt_cnt++;
            }
            i--;
    }
    rms = 0.0;
    for (i=0; i<points; i++)
        net_response(patt_ndx[i]);
        back_propagation(patt_ndx[i]);
        patt_ndx[i] = -1;
    }
    if (((cnt+1)%step)==0 || cnt==(iterations-1)) {
        collect_rms();
        if (cube_node==0) {
            cont = rms>epsilon;
            nw_ret = nwrite(&cont, 1, 0xffff, eps_bcst, 0);
            printf("+++ %d\tRMS=%lf
", cnt, rms);
        } else nr_ret = nread(&cont, 1, &src, &eps_bcst, 0);
    }
}
cnt++;  
}
free(pattndx);

main(argc, argv)
int argc;
char **argv;
{
char group_flag=1, *group_name, *filename, status=0;
int i;
time_t cpu_time_i, cpu_time_o, cpu_time_r, cpu_time_s, 
cpu_time_t, junk;

for (i=1; i<argc && !status; i++) {
    if (strcmp("-f", argv[i])==0) {
        if (i+1 < argc) {
            group_name = (char *)malloc(strlen(argv[i+1])+1);
            strcpy(group_name, argv[i+1]);
            group_flag=0;
        } else status=1;
        i++;
    } else if (strcmp("-i", argv[i])==0) {
        if (i+1 < argc) {
            sscanf(argv[i+1], "%ld", &iterations);
        else status=1;
        i++;
    } else if (strcmp("-s", argv[i])==0) {
        if (i+1 < argc) {
            sscanf(argv[i+1], "%ld", &step);
        else status=1;
        i++;
    } else if (strcmp("-t", argv[i])==0) test_net_flag=1;
else if (strcmp("-p", argv[i])==0) out_prm_flag=1;
else status=1;
    }

if (status || argc<2 || group_flag) {
    if (mynode()==0)
        printf("usage: %s -f <group> {-i <iterations>} {-s <step> }
{t} {-p}\n", argv[0]);
    exit(1);
}
if (mynode()==0) {
    printf("nnCUBE 2: Neural Networks/Back Propagation 
(March, 1993)\n\n");
    printf("GROUP=%s\n", group_name);
    printf("ITERATIONS=%ld, ITER-STEP=%ld, TEST=%d, 
OUTPUT=%d\n\n", iterations, step,
            test_net_flag, out_prm_flag);
}

    cpu_time_s = time(&junk);
init(group_name);
    cpu_time_i = time(&junk);
simulate();
    cpu_time_r = time(&junk);

filename = (char *)malloc(strlen(group_name)+9);
if (out_prm_flag) {
    strcpy(filename, group_name);
    strcat(filename, ".PRM.run");
    out_prm(filename);
}
    cpu_time_o = time(&junk);
if (test_net_flag) {
    strcpy(filename, group_name);
    strcat(filename, ".TEST");
    test_net(filename);
}
    cpu_time_t = time(&junk);
if (cube_node == 0) {
    printf("- TIME (init) %ld sec\n", cpu_time_i-cpu_time_s);
    printf("- TIME (CPU:|%d node(s)) %ld sec\n", 1<<cube_dim, 
cpu_time_r-cpu_time_i);
    printf("- TIME (output parameters) %ld sec\n", cpu_time_o- 
cpu_time_r);
    printf("- TIME (test section) %ld sec\n", cpu_time_t-
cpu_time_o);
    printf("- TIME (total) %ld sec\n\n", cpu_time_t-cpu_time_s);
}

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


