2002 International Joint Conference on Neural Networks (IJCNN'02), Honolulu, Hawaii, May 12-17, 2002

On Neural Network Hardware and Programming Paradigms

DR. KARL MATHIA* and JEFF CLARK**

*PRI Automation, Inc.
455 North Bernardo Avenue
Mountain View, California 94043
kmathia@pria.com, www.pria.com

**Renaissance Networks
415 South McClintock Dr., Suite 5
Tempe, Arizona 85281
jeff@rni.net, www.rni.net

Abstract - Implementation guidelines and performance benchmarks for dedicated neural network hardware and software are proposed. An example of a dedicated neural network processor and the implementation of a popular neural network architecture and learning algorithm on this hardware is presented, emphasizing the specific architecture and programming paradigm. Finally, the performance of the implementation is benchmarked in a single-processor and multiple-processor environment against a supercomputer. Such implementations are aimed for neural network application requiring rapid on-line learning.

1 INTRODUCTION

It is well known that Artificial Neural Networks (ANNs) are well suited as computational tools for solving certain classes of complex problems, although software implementations on general-purpose computers can be too slow for time-critical applications. Sufficient performance has been demonstrated utilizing 'supercomputers,' but at considerable cost for both hardware and software. Here, dedicated ANN hardware and the ANN programming paradigm are proposed as an effective compromise of performance and cost, enabling a computational platform for time-critical neural network processing tasks, e.g. on-line learning for neurocontrol in robotics applications or complex patterns recognition. The usual hardware design questions apply, for example, digital or analog technology, fixed or floating point arithmetic.
The majority of ANN implementations today tend to be software simulators hosted on general-purpose computers. Some special neural network platforms were researched and published [1][2][8]. This paper briefly reviews the basic computational paradigm for ANNs, hardware and software realization, and performance benchmarks. Finally, an implementation and benchmark example with a commercial neural network processor illustrates these concepts.

2 COMPUTATIONAL PARADIGMS AND BENCHMARKS

General guidelines for implementing the computational paradigm of ANNs are proposed, including hardware, software, and benchmarks.

2.1 Computational Paradigm

ANNs represent a computational paradigm, whose characteristics include

- intrinsic parallelism,
- local processing in (artificial) neurons,
- distributed memory,
- learning and recall modes.

The computational paradigm of ANNs specifies an architecture of one or more parallel layers of 'artificial' neurons. A neuron output is a function of input, connection weights, and nonlinear transfer function. The scalar multiplication of a weight and input is often called a 'connection.' Knowledge is acquired by an ANN through a learning process and stored in the network's 'distributed memory,' i.e. the connection weights [2][6].

2.2 Hardware Considerations

The limited scope of the ANN computational paradigm, i.e. the small number of computational 'primitives', suggests advantages of hosting ANNs on dedicated Neural Network Hardware (NNH) to maximize performance at a given cost target. Connections and transfer functions are the most expensive primitives and should be emphasized: the number of weights grows exponentially with the number of neurons, leaving room for significant performance improvement

0-7803-7278-6/02/$10.00 ©2002 IEEE

Further parameters to the design of NNH include [1][2][8]:

- general-purpose platforms with a certain flexibility for various ANN paradigms, and
- dedicated NNH, specialized for the implementation of specific ANN architectures only,
- analog, digital, or hybrid technology,
- off-board or on-board learning,
- level of parallelism.

After ‘off-board’ learning the resulting connection weights are ‘frozen’ and downloaded to the NNH for rapid operation. One reason for this solution is the challenge of on-chip learning due to implementing complex learning algorithms on NNH with limited instruction sets.
The intrinsic parallelism of ANNs facilitates parallel processing as a key feature of NNH, i.e. the decomposition of neural processing into concurrently executed subprocesses. Parallel processing exploits concurrence in a computational process and comprises both parallel hardware and parallel software. The sophistication level of parallelism also depends on the software, as is discussed below.

2.3 Programming Paradigm and Language

An intriguing aspect o the ANN computational paradigm is to represent complex problems with a small number of primitives. The way we describe computations for computers is known as programming paradigm or programming model. It is a technique which supports writing code for a specific set of problems, using a programming language that is designed to describe that specific set of problems. A good language also includes standard libraries and programming tools like editor and compiler [3]. Computer code that reads well, is easy to maintain, and is easy to reuse is often loosely referred to as ‘good’ code.
In general one has to compromise between

- paradigm,
- powerful language, and
- runtime performance.

for a computer platform. An ANN language should therefore resemble the ANN paradigm, i.e. should match its description and definition. Typically, for ANNs a good compromise is not hard to find, due to the limited number of computational primitives and therefore possible match of language and hardware. Unfortunately, an understandable code can result in a more complex, thus less powerful language with degraded runtime performance.
For example, the phrase "compute the value at neuron N" could involve the computation of hundreds of inner product and transfer function, but it may be desired to calculate all connections with one command for all neurons in a layer. The ANN paradigm therefore provides the opportunity to design and leverage a powerful language that exploits the capabilities of the NNH for good run-time performance.
The mere availability of parallel hardware itself does not guarantee parallel processing. This directly corresponds to the sophistication of parallel software exploiting the parallel architecture in NNH. Parallel processing at the highest level is carried out by multiple programs, while at a medium level it is limited to concurrent tasks within a single program. This requires the decomposition of the problem at hand into simultaneously executable tasks and is ideal for neural processing. The lowest level refers to concurrence of multiple instructions, or even concurrence within an instruction [3][5].

2.4 Benchmarking

For artificial neural networks the benchmarking problem has rarely been addressed. Elapsed time is often measured for the most expensive neural operations, i.e. transfer functions and, in particular, connections [3][4][6][7]. While the number of transfer functions increases only linearly with network size, the number of connection weights increases exponentially and becomes the determining performance factor.
Connections-per-second (CPS) have been proposed as a benchmark for the computational speed of ANN software simulators, but without considering data format and the network scaling problem, i.e. performance as a function of the network size for digital ANNs [7]. Published CPS could refer to 1-bit connections, i.e. the least expensive connection. These issues are taken into account by the following benchmark [6].
A *Connection-byte-per-second* (CBS) is defined as

0-7803-7278-6/02/$10.00 ©2002 IEEE

$$CBS = bytes(w) * bytes(x) * CPS \quad (1)$$

The operato *bytes(.)* returns the word length of its argument in bytes. Equation 1 evaluates the computational cost of a 32-bit connection sixteen times as high as that of a 8-bit connection, i.e. 1 CBS = 4*4 CPS = 16 CPS.

3 NEURAL NETWORK PROCESSOR NNP®

The above concepts are illustrated with the commercially available Neural Network Processor (NNP®) by Accurate Automation Corp. [1][8]. Its design concept is reviewed, and the implementation of the popular error backpropagation algorithm and its benchmarked against a high performance workstation.

3.1 Architectur

The NNP® architecture provides features for fast processing of neural connections and transfer functions, and is well suited for feedforward ANN architectures. Up to eight NNPs can be integrated in a parallel multiprocessor environment, resulting in a multiple instruction stream/multiple data stream (MIMD) platform. The block diagram of a single NNP® is shown in Figure 1.

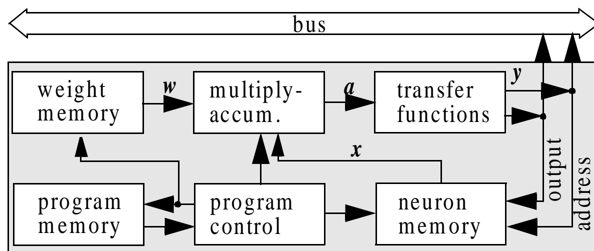


Figure 1. Block diagram of a singleNNP.

A multiply-accumulate unit, lookup tables for transfer functions, and 16-bit integer arithmetic emphasize the above feature. The figure illustrates dataflow of activation values *a*, neuron input ***x*** and weight value ***w***. The neuron output *y*, i.e. the transfer function, is not computed explicitly but is read from a pre-loaded lookup tabl . Weights are usually learned off-line, but we will shown how the NNP can be utilized for on-chip learning.

Efficiency is further increased by instruction pipelining, enabling the completion of one instruction per clock cycle. Thus the NNP is single instruction stream/multiple data stream (SIMD) processor. It is argued that instruction pipelining implements parallelism even on a SISD processor. Indeed, pipelined events in overlapped time periods are often referred to as temporal parallelism, optimizing the exploitation of a single processor. Parallelism through the replication of physical devices is referred to as spatial parallelism [3]. In this sense a single NNP with instruction pipelining implements temporal parallelism, i.e. parallelism at the third or lowest level. Spatial parallelism at the first or second level is realized by parallel NNPs, whose performance increases approximately linearly with the number of NNPs.

3.2 Programming Paradigm

The NNP programming paradigm emphasizes layered, well-structured, feedforward ANN architectures. Accordingly, its programming language comprises only nine powerful commands, tailored for the efficient computations of appropriate ANN primitives, in particular connections and nonlinear transfer function. For example ‘mula’ (multiply and accumulate) and ‘lbtf’ (compute transfer function). A sigmoidal neuron with three inputs and weights would be implemented using four commands:

```
mull x1,w1
mula x2,w2
mula x2,w2
lbtf c,sigmoid
```

The cornerstone of an NNP assembler program is the multiply-accumulate command, which multiplies one weight value with one input value and adds the product to the existing activation value. For larger yet well-structured ANNs we did not code the NNP language, but wrote a code generator in the C-programming language.

3.3 Example: Backpropagation Learning

This section describes the exercise of representing a complex problem (a learning process) with a small number of ANN primitives. We demonstrate how programming paradigm and language for dedicated NNH, even when tailored for off-board learning of feedforward ANN architectures, can be used for implementing rapid on-board learning. The simple XOR problem was used for demonstra-

0-7803-7278-6/02/$10.00 ©2002 IEEE

tion purposes, and the well-known multi-layer perceptron (MLP), error backpropagation were chosen as the ANN architecture and the learning algorithm, respectively [2][5]. Backpropagation (BP) learning is widely used, but is almost exclusively performed off-line with subsequent weight downloads. Our experimental results underline that real-time learning for a large class of problems is not only achievable but can also be financially affordable on dedicated NNH.

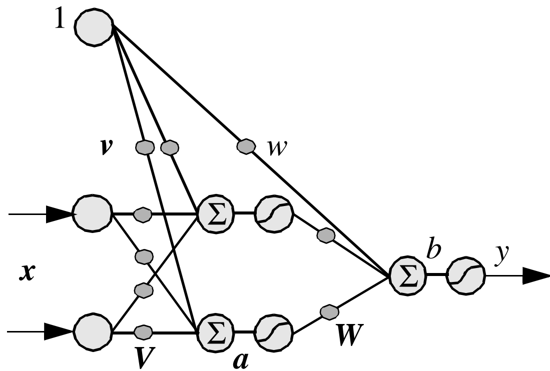


Figure 2. 2-2-1 Multi-Layer Perceptron (MLP).

The MLP used is shown in Figure 2. It has the classic 2-2-1 topology for solving the XOR problem. The summation and transfer function primitives per neuron are displayed separately. The connection weights are illustrated as dots between neurons. In matrix notation the MLP is defined by a m-to-n mapping $f:\mathfrak{R}^m \rightarrow \mathfrak{R}^n$ [5]. Her $m=2$, $n=1$, and the following notation applies:

- $\boldsymbol{x} \in \mathfrak{R}^{2x1}$ .. MLP input vector,
- $\boldsymbol{V} \in \mathfrak{R}^{2x2}$ .. weight matrix (hidden layer),
- $\boldsymbol{v} \in \mathfrak{R}^{2x1}$ .. bias vector (hidden layer),
- $\boldsymbol{a} \in \mathfrak{R}^{2x1}$ .. activation vector (hidden layer),
- $\boldsymbol{W} \in \mathfrak{R}^{1x2}$ .. weight matrix (output layer),
- $w \in \mathfrak{R}^{1x1}$ .. bias (output layer),
- $y \in \mathfrak{R}^{1x1}$ .. MLP output.

In this special case with only one MLP output v and y are scalars. The 2-to-1 mapping $y=f(\boldsymbol{x})$ realized by the MLP in Figure 2 is defined as follows. The sigmoid function, here $\sigma:\mathfrak{R}^2 \rightarrow \mathfrak{R}^2$, is a vector-valued function of same dimension as its argument:

$$f(\boldsymbol{x}) = \sigma(\boldsymbol{W}\,\boldsymbol{\sigma}(\boldsymbol{V}\,\boldsymbol{x} + \boldsymbol{v}) + w) \tag{2}$$

$$= \sigma(\boldsymbol{W}\,\boldsymbol{\sigma}(\boldsymbol{a}) + w) \tag{3}$$

$$= \sigma(b) \tag{4}$$

BP learning adjusts the connection weights by minimizing a scalar, quadratic, and multi-variate error function $E \in \mathfrak{R}$ using the gradient method [2][5]:

$$E(\boldsymbol{x}) = 1/2\,(\boldsymbol{d} - \mathbf{y}(\boldsymbol{x}))^T\,(\boldsymbol{d} - \mathbf{y}(\boldsymbol{x})) \tag{5}$$

$$= 1/2\,\boldsymbol{e}^T(\boldsymbol{x})\,\boldsymbol{e}(\boldsymbol{x}), \tag{6}$$

$$= 1/2\,[e(\boldsymbol{x})]^2\text{, for } n=1, \tag{7}$$

where $\boldsymbol{d} \in \mathfrak{R}^{nx1}$ is the desired output, and $\boldsymbol{e} \in \mathfrak{R}^{nx1}$ the output error for training. Weights are updated at each learning cycle k, using gradient and learning rate η:

$$\boldsymbol{W}_{k+1} = \boldsymbol{W}_k + \eta \cdot \frac{\partial E(\boldsymbol{x})}{\partial \boldsymbol{W}} = \boldsymbol{W}_k + \Delta \boldsymbol{W}_k \tag{8}$$

The partial derivatives of E with respect to the four weight matrices and vectors are obtained using the chain rule. Note that the derivative of $\sigma(b)$ is $\sigma(b)[1-\sigma(b)]$, or $y[1-y]$:

$$\frac{\partial E}{\partial \boldsymbol{W}} = \frac{\partial E}{\partial e} \cdot \frac{\partial e}{\partial y} \cdot \frac{\partial y}{\partial b} \cdot \frac{\partial b}{\partial \boldsymbol{W}} \tag{9}$$

$$= e \cdot (-1) \cdot [y(1-y)] \cdot \boldsymbol{\sigma}(\boldsymbol{a})^T, \tag{10}$$

$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial e} \cdot \frac{\partial e}{\partial y} \cdot \frac{\partial y}{\partial b} \cdot \frac{\partial b}{\partial w}, \tag{11}$$

$$= e \cdot (-1) \cdot [y(1-y)], \tag{12}$$

Partial derivatives for E with respect to $\boldsymbol{V}$ and $\boldsymbol{v}$ are derived accordingly.

We implemented the above BP learning on a single NNP and its host computer, as is illustrated in Figure 7 for a 2-2-1 MLP at the end of this paper. The MLP from Figure 2 is shown grey-shaded. The BP forward path on the NNP was written in assembler (using a code generator program), while the feedback path was written in C for the host computer. The NNP computes the gradients and delta weights (learning rate 1). The host updates weights and presents training data to the NNP. Figure 7 also shows that neurons are organized in parallel layers, as is dictated by the NNP's computational paradigm, even where a neuron does not contribute to the algorithm or serves as distribution node only.

0-7803-7278-6/02/$10.00 ©2002 IEEE

Note that only the computational primitives in Figure 3 are needed to realize the BP algorithm.

(Σ) ... summation node

... sigmoidal function

... derivative of sigmoid function

(Π) ... multiplication node

... linear distribution node

Figure 3. Computational primitives.

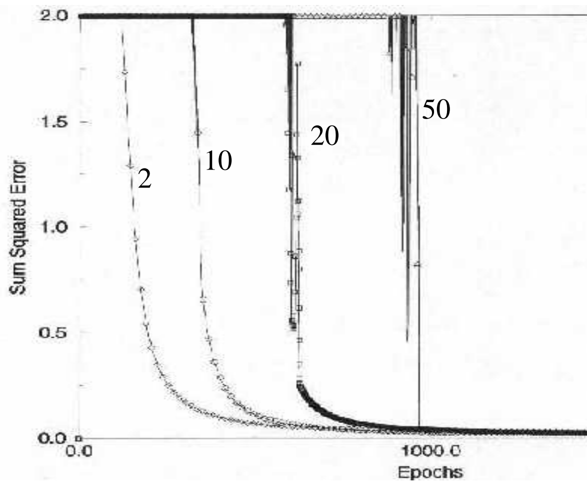


Figure 4. Learning curves of four MLPs.

Learning curves (sum of squared error over learning sets, or epochs) for MLPs with 2, 10, 20, and 50 neurons in the hidden layer are shown in Figure 4. Note the initial saturation, due to the numerical limit of the transfer function (16 bit) at the output neuron. Saturation also occurs for more than 10 hidden neurons.

3.4 Benchmarking

The realizable sustainable performance, not the theoretical peak performance, of the NNP in a single and parallel multiprocessor environment was compared against that of an Intel Paragon using the CBS benchmark: 1 to 4 parallel NNPs were compared against 1 to 128 parallel processor nodes (two RISC i860 processors each) on the Intel Paragon. The MLP architecture used above implemented on both the NNP (in assembler) and on the Paragon (in C).

The results are illustrated in Figure 5 (NNP) and in Figure 6 (Paragon): CBS are shown as a function of both the number of neurons and processors.

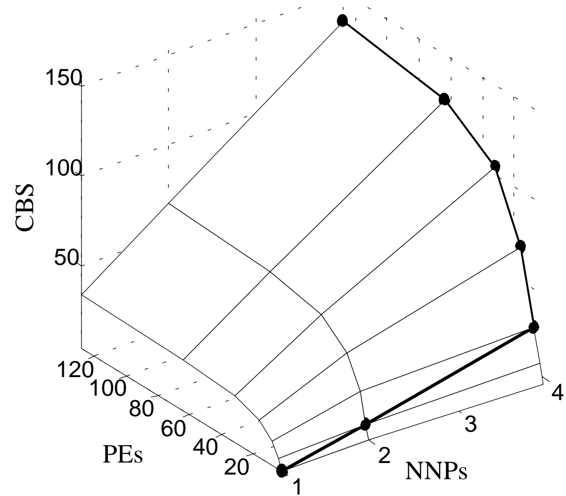


Figure 5. CBS benchmark for the NNP.

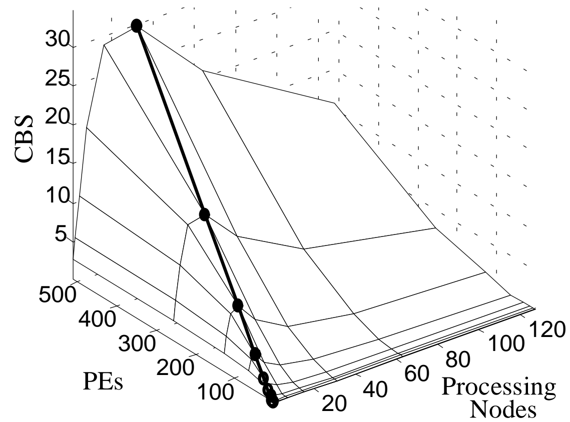


Figure 6. CBS benchmark for the Intel Paragon.

The performance optimum for a given number of neurons (or processing elements, PEs) is emphasized as a bold line with black dots (the actual data points tested). Note that the maximum CBS computed by the NNP outperforms that of the Paragon by a factor of approximately four (~150 over ~30). The measured optimal number of neurons per processor on either machine relates to the well-known load balancing problem on parallel machines [3]. For the NNP that number was 2, for the Paragon 32 neurons. Also note the linear performance increase with the number of parallel NNPs (per underlying computational paradigm), while for the Paragon the increasing computational overhead degrades performance beyond the optimal number of neurons per processor. For more details see [6].

0-7803-7278-6/02/$10.00 ©2002 IEEE

4 CONCLUSION

Implementation guidelines for dedicated neural network hardware and software were proposed. These guidelines were reviewed emphasized using a commercial dedicated ANN hardware/software platform. The 'backprop' learning was implemented and the NNH platform benchmarked. It was demonstrated that ANN learning at the performance of supercomputers cannot only be implemented on dedicated hardware, but also can be affordable.

5 REFERENCES

[1] Accurate Automation Corporation, *AAC Neural Network MIMD Processor, Technical Data Sheet* , Chattanooga, TN, 1995.

[2] S. Haykin, *Neural Networks - A Comprehensive Foundation*, IEEE Press/Macmillan College, New York, 1994.

[3] Hennesy J.L. and D.A. Patterson, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufman, San Mateo, California, 1993.

[4] E. van Keulen, S. Colak, H. Withagen, and H. Hegt, "Neural Network Hardware Performance Criteria," *Proc. IEEE Int. Conf. Neural Networks,* Orlando/Florida, Vol. 3, pp. 1885-1888, 1994.

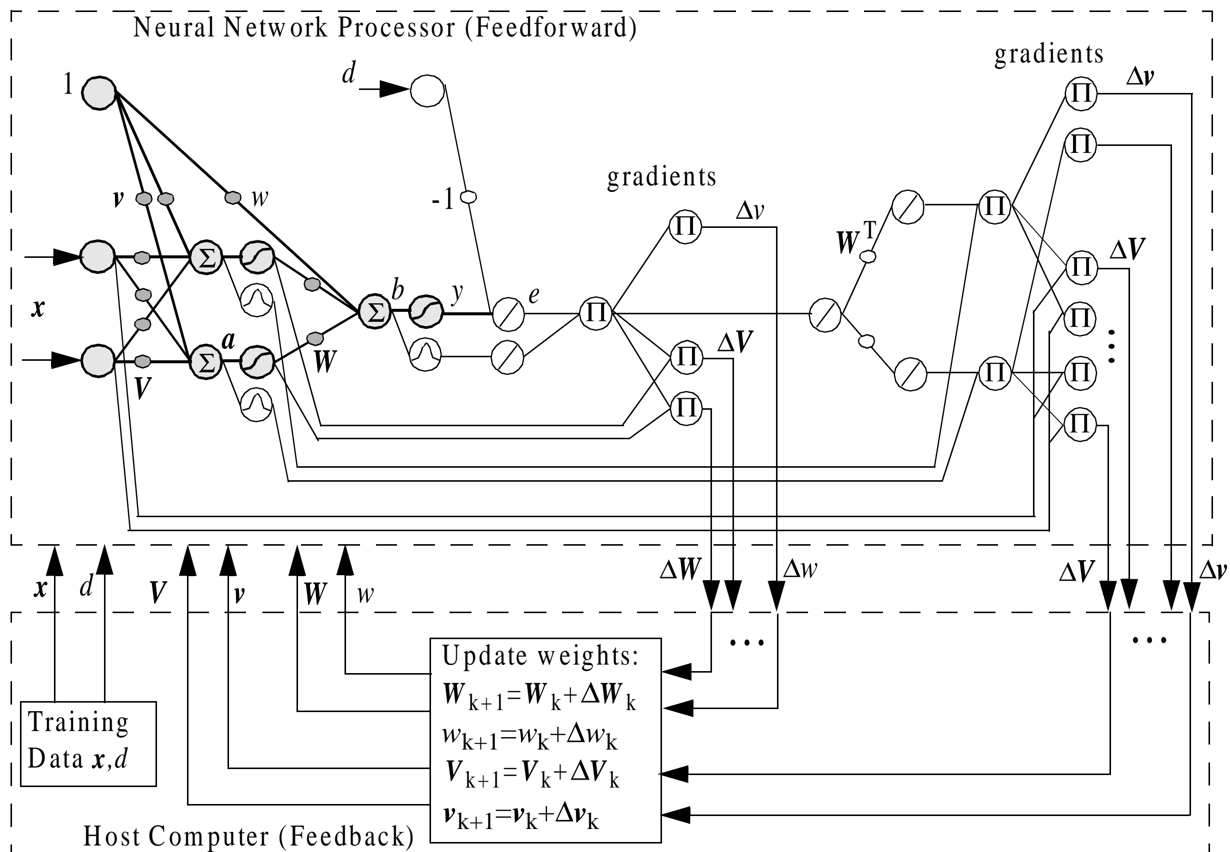
[5] K. Mathia, *Solutions of Linear Equations and a Class of Nonlinear Equations Using Recurrent Neural Networks*, Ph.D. Dissertation, Portland State University, Portland, Oregon, 1996.

[6] K. Mathia, J. Clark, B. Colbert, and R. Saeks, "Benchmarking an MIMD Neural Network Processor", *Proc. World Cong. Neural Networks '96*, San Diego/California, pp. 590-595, June 1995.

[7] G. Rogers, J. Solka, J. Ellis, and H. Szu, "A Neurocomputing Benchmark for Digital Computers," *Proc. SIMTEC-WNN '92,* Clear Lake, Texas, pp. 425-430, November 1992.

[8] R. Saeks, K. Priddy, K. Schnieder, and S. Stowell, "On the Design of an MIMD Neural Network Processor", *Proc. World Cong. Neural Networks '95*, San Diego/California, pp. 590-595, June 1995.

Figure 7. Illustration of the BP implementation.



0-7803-7278-6/02/$10.00 ©2002 IEEE