Journal of Parallel and Distributed Computing **60**, 677–689 (2000)
doi:10.1006/jpdc.2000.1634, available online at http://www.idealibrary.com on IDEAL®

Mapping of Neural Network Models onto Systolic Arrays

Sudipta Mahapatra

Department of Computer Science Engineering and Applications, Regional Engineering College, Rourkela 769 008, Orissa, India
E-mail: smahapatra@rec.ori.nic.in

and

Rabi N. Mahapatra

Department of Computer Science, Texas A&M University, College Station, Texas 77843-3112
E-mail: rabi@cs.tamu.edu

Received August 4, 1995; revised February 25, 1999; accepted January 28, 2000

This paper presents a mapping scheme for the proposed implementation of neural network models on systolic arrays. The mapping technique is illustrated on the multilayer perceptron with back-propagation learning. Dependency graphs have been given that represent the operations in the execution phases of the neural network model and later suitable algorithms are presented to realize the operations in a linear bidirectional systolic array. The speedup metric has been used to evaluate the performance of the proposed implementation.

Key Words: neural net models; dependency graph; mapping scheme; bidirectional systolic array; back-propagation algorithm.

1. INTRODUCTION

Artificial neural networks (neural nets) have emerged as a promising alternative for solving real world problems such as speech and patter recognition, radar signal tracking, and sonar target detection. They are able to satisfy the basic requirement of real world problems, i.e., high execution speed. But, for solving any problem, first a neural net has to be trained or the network weights have to be adjusted to correctly classify a set of example patterns, an operation that is highly computation intensive.

Uniprocessor simulation and the implementation on high speed hardware platforms are both inappropriate for neural network implementation as they are either very slow or rather inflexible. A compromise between these two is the implementation on programmable parallel computers [3–11]. And, the systolic array is one of the best parallel architectures for implementing neural nets. It can

0743-7315/00 $35.00

circumvent the communication problem encountered in neural net implementation due to the high degree of interconnection among the neurons.

A number of systolic algorithms are available for matrix–vector multiplication, the basic computation involved in the operation of a neural net. Using these, many systolic algorithms have been formulated for the implementation of neural nets [6–10]. In [7], Kung *et al.* have proposed a unified systolic architecture for the implementation of neural net models. It has been shown that the proper ordering of the elements of the weight matrix makes it possible to design a cascaded DG (dependency graph) for consecutive matrix–vector multiplication, which requires the directions of data movement at both the input and the output of the DG to be identical. Using this cascaded DG, the computations in both the recall and the learning iterations of a back-propagation algorithm have been mapped onto a ring systolic array. The same mapping strategy has been used in [8] for mapping the hidden Markov model (HMM) and the recursive back-propagation network (RBP) onto the ring systolic array. The main drawback of the above implementations is the presence of spiral (global) communication links. Thus, an important advantage of the systolic architecture, i.e., use of a locally communicative interconnection structure, is lost. In [10], a two-dimensional array is used to map the synaptic weights of individual weight layers in the neural net. By placing side by side the arrays corresponding to adjacent weight layers, both the recall and the learning phases of the back-propagation algorithm can be executed efficiently. But, as the directions of data movement at the output and the input of each array are different, this leads to a very nonuniform design. Again, a particular layout can only implement neural nets having identical structures. For neural nets that are structurally different, another layout would be necessary.

In the current work, mapping schemes have been devised for the proposed implementation of the multilayer perceptron with back-propagation learning (BP net) on systolic arrays. DGs are derived for implementing operations in both the recall and the learning phases of the back-propagation algorithm. All these DGs are free from global connections. Later, these DGs are mapped onto a linear bidirectional systolic array and algorithms have been presented for executing both the recall and the learning phases efficiently. The issue of partitioned mapping has also been addressed. Also, the P processor speedup of the proposed implementation has been estimated and is plotted against both the number of neurons and the number of processors.

Section 2, following the Introduction, briefly discusses the multilayer perceptron network and the back-propagation learning algorithm. The mapping strategy is outlined in Section 3. In Section 4, a partitioning scheme is presented for the efficient execution of large sized neural nets on smaller arrays. Finally, Section 5 concludes this paper with discussions on the speedup performance of the proposed implementation.

2. MULTILAYER PERCEPTRON AND BACK-PROPAGATION ALGORITHM

Multilayer perceptron is a popular neural net model widely used in pattern classification or pattern matching [1]. It is trained using the back-propagation

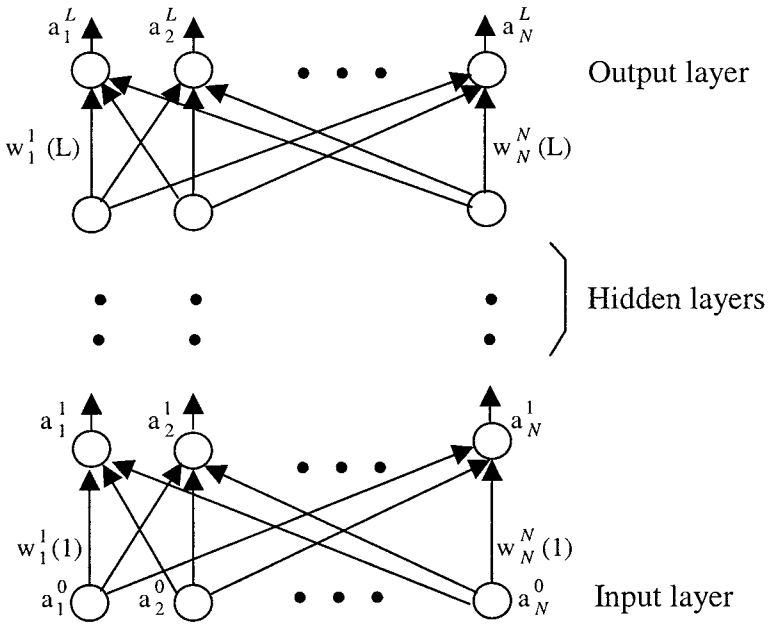

FIG. 1. Multilayer perception network.

learning algorithm [2]. Below, we discuss the structure and operation of the back-propagation algorithm in brief.

Multilayer Perceptron

The multilayer perceptron consists of multiple neurons arranged in the form of an input layer, a output layer, and one or more hidden layers. We assume that any two adjacent layers are fully interconnected with weighted connections. One such neural network of $L+1$ layers is shown in Fig. 1.

Back-Propagation Algorithm

The back-propagation algorithm operates in two distinct phases: (1) the forward pass or recall phase and (2) the backward pass or learning phase. The recall phase is used to compute the state values of the hidden and output layer neurons. In the learning phase, the error values computed for the output layer neurons are propagated backward to compute the error values of all the hidden layer neurons and to adjust their input weights.

The notations and conventions used in presenting the algorithm are as follows:

1. The multilayer perceptron is assumed to have $L+1$ layers, numbered from 0 to L with N^l neurons in layer l. The layer numbered 0 is the input layer, layer L is the output layer, and layers 1 to $L-1$ are the hidden layers.

2. The ith neuron in layer l, its state value, and its error value (δ-value) are given by n_i^l, a_i^l, and δ_i^l, respectively.

3. Each neuron n_i^l has a bias input θ_i^l.

4. u_i^l represents the weighted sum for n_i^l.

5. The input weight of n_i^l from the jth neuron of layer $l-1$ is denoted as $w_i^j(l)$.

6. t_i^l stands for the ith element of the target pattern.

7. f is the nonlinear activation function used in the algorithm.

Using these notations, the state value computation, δ-value calculation, and weight adaptation are carried out using Eqs. (1), (2), and (3), respectively.

$$\begin{aligned} u_i^l &= \sum_{j=1}^{N} w_i^j(l)\, a_j^{l-1} + \theta_i^l \\ a_i^l &= f(u_i^l) \end{aligned} \tag{1}$$

$$\begin{aligned} \delta_j^l &= (t_j - a_j^l)\, f^l(u_j^l), && \text{for} \quad l = L, \\ &= \left[\sum_{i=1}^{N} \delta_i^{l+1} w_i^j(l+1) \right] f^l(u_j^l), && 1 \leqslant l \leqslant L-1. \end{aligned} \tag{2}$$

$$\Delta w_i^j(l) = \eta \delta_i^l a_j^{l-1}, \qquad \text{where } \eta \text{ is a learning constant.} \tag{3}$$

The multilayer perceptron network starts from an arbitrary set of weights which is thereafter adjusted by repeatedly presenting it with a training set of input pattern–target pattern pairs and executing the recall and the learning phases till the network converges or the error is within an acceptable limit.

3. MAPPING OF THE BP NET

3.1. *Systolic Design for the Recall Phase*

The computations involved in the recall phase can be represented in the matrix form as follows:

$$\begin{aligned} U^l &= W^l A^{l-1} + \theta^l \\ A^l &= f(U^l). \end{aligned} \tag{4}$$

In the above equation, A^l and θ^l are vectors representing the state values and bias inputs of layer l neurons and W^l is a matrix representing the input weights of layer l.

A DG for computing the state values of layer l neurons from the state values of its preceding layer is shown in Fig. 2. Functions of the various nodes in this DG are illustrated in Fig. 3. It can be observed that all the nodes are functionally identical and differ only in the directions of data movement, which depend on the position of a node in the DG. The above DG can be mapped onto a linear systolic array in a straightforward manner. A projection can be taken in the vertical direction and the schedule hyperplanes can be chosen in a direction parallel to the horizontal. The mapping is such that the processor P_i stores the state values and all the input weights of neuron n_i^l for $1 \leqslant l \leqslant L$. This has been shown in Fig. 4 for mapping of a

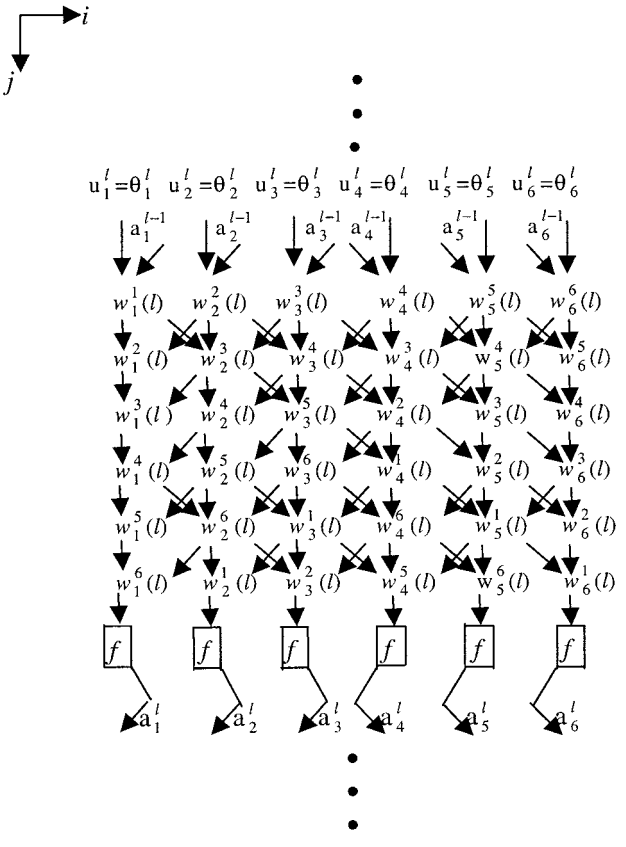

FIG. 2. DG for the recall phase.

BP net of six neurons per layer onto a six-processor array. It may be observed that the first and the last links of the first and the last processor, respectively, are shorted. Thus, for $i=1$, $P_{i-1}=P_i$ and similarly for $i=N$, $P_{i+1}=P_i$. The operations performed by a processor in the lth iteration are as given in Algorithm 1.

ALGORITHM 1. In the following algorithm it is assumed that the processors in the linear array are represented as $P_{f(k)}$, where $f(k)=k$, $1 \leqslant k \leqslant N/2$, represents the processors $P_1, P_2, ..., P_{N/2}$, and $P_{f(k)}$ for $f(k)=(N-k+1)$, $1 \leqslant k \leqslant N/2$, represents the processors $P_N, P_{N-1}, ..., P_{N/2+1}$. Using these notations, the algorithm for computing the state values of layer l neurons from the state values of layer $(l-1)$ is as follows:

1. Each processor P_i initializes a variable $u_i^l=\theta_i^l$.
2. P_i multiplies a_i^{l-1} with $w_i^j(l)$ and adds the product to u_i^l.
3. Afterwards, each processor performs the following operations

(a) In any cycle, processor $P_{f(k)}$ receives data a_j^{l-1} from $P_{f(k+1)}$ and updates: $u_i^l=u_i^l+w_i^j(l)\,a_j^{l-1}$. $P_{f(k)}$ then sends a_j^{l-1} to $P_{f(k-1)}$.

(b) In addition to the above,

(i) for (f_k-1) clock cycles, starting with the second, processor $P_{f(k)}$, $1 \leqslant k \leqslant N/2$, receives data a_n^{l-1} from $P_{f(k-1)}$ and forwards it to $P_{f(k+1)}$.

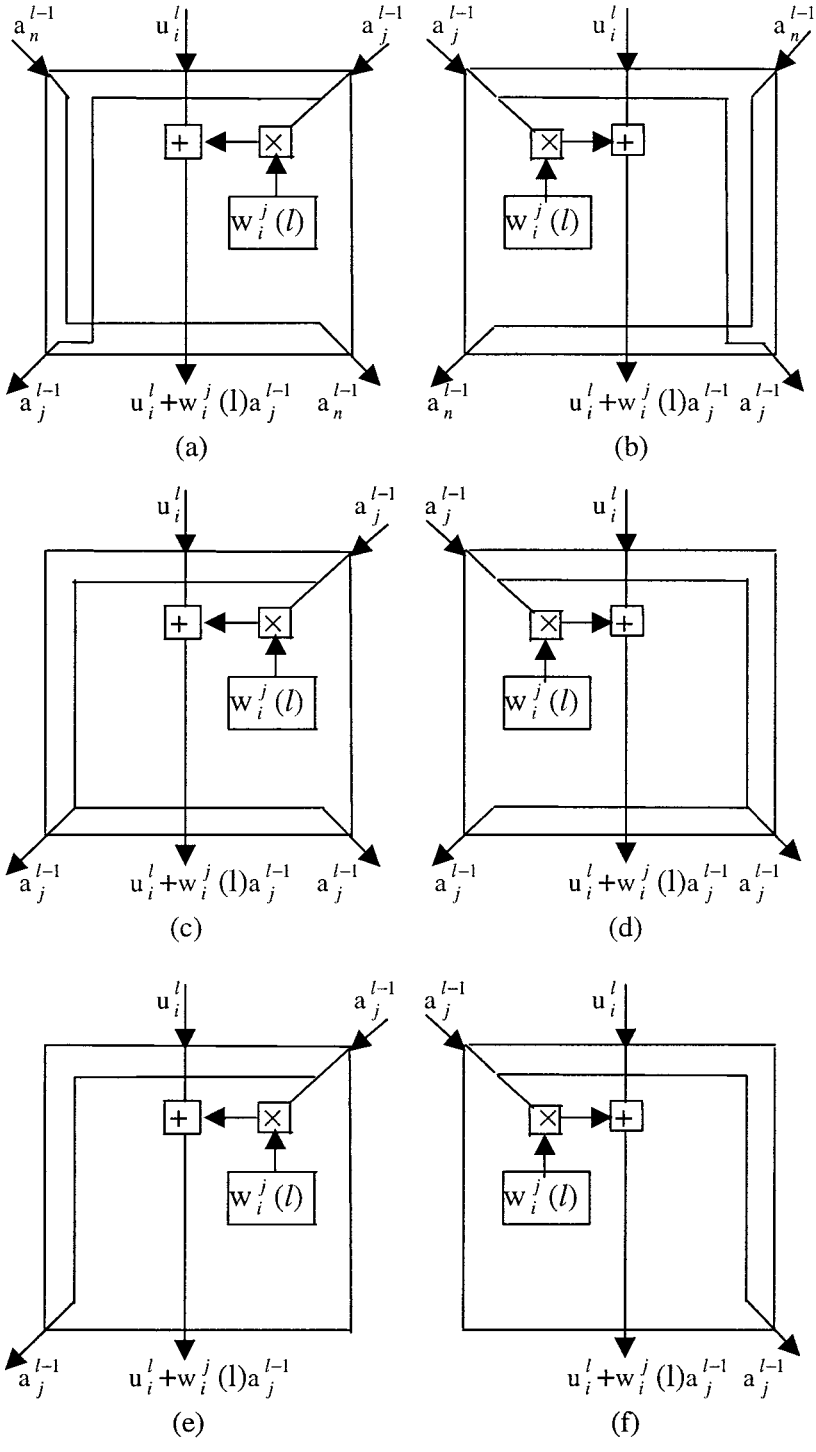

FIG. 3. Node functions in the DG of Fig. 2 for the node $N(k, m)$. (a) $2 \leqslant k \leqslant N/2$, $2 \leqslant m \leqslant k$, and $N/2+2 \leqslant m \leqslant N/2+k$. (b) $N/2+1 \leqslant k \leqslant N-1$, $2 \leqslant m \leqslant N-k+1$, and $N/2+2 \leqslant m \leqslant 3N/2-k+1$. (c) $1 \leqslant k \leqslant N/2$, $m=1$ and $m=N/2+1$. (d) $N/2+1 \leqslant k \leqslant N$, $m=1$ and $m=N/2+1$. (e) $1 \leqslant k \leqslant N/2$, $k+1 \leqslant m \leqslant N/2$ and $N/2+k+1 \leqslant N$. (f) $N/2+1 \leqslant k \leqslant N$, $N-k+2 \leqslant m \leqslant N/2$, and $3N/2-k \leqslant m \leqslant N$.

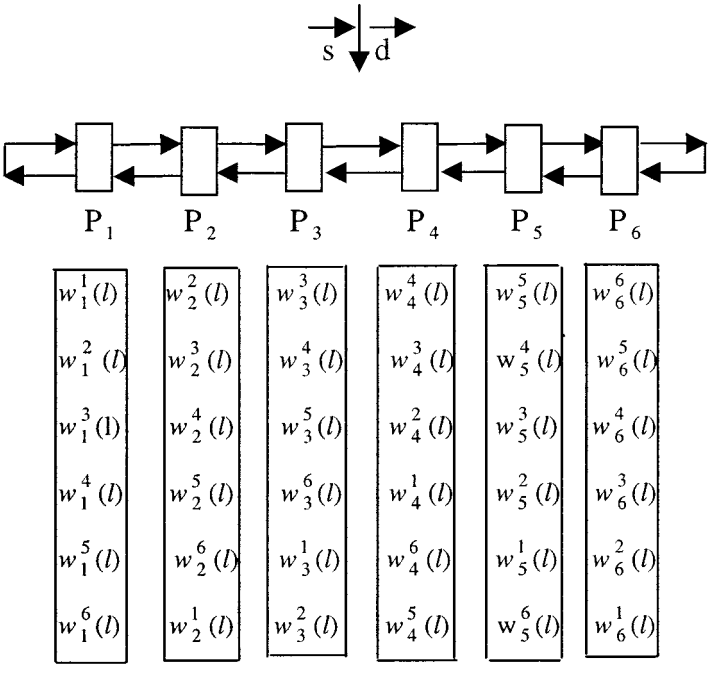

FIG. 4. Implementation in a six-processor array.

(ii) In the $(N/2+1)$th clock cycle, $P_{f(k)}$ sends the data a_j^{l-1} received from $P_{f(k+1)}$ in the previous cycle back to $P_{f(k+1)}$.

(iii) Again for (f_k-1) cycles, this time starting with the $(N/2+2)$th, processor $P_{f(k)}$, $1 \leqslant k \leqslant N/2$, receives data a_n^{l-1} from $P_{f(k-1)}$ and forwards it to $P_{f(k+1)}$.

4. After $(N-1)$ clock cycles, the ith processor would have accumulated the weighted sum $u_i^l = \sum_{j=1}^{N} w_i^j(l)\, a_j^{l-1}$ for the ith neuron. It then computes a_i^l by applying the Sigmoid function to this weighted sum.

This algorithm is executed repeatedly with increasing values of l till the state values of all the output layer neurons have been determined. It is assumed that the state value a_i^l after its evaluation is stored in the processor P_i.

3.2. *Systolic Design for the Learning Phase*

Calculation of the lower layer δ-values can be carried out using consecutive vector–matrix multiplication as shown below.

$$\delta^l = \delta^{l+1} W^{l+1} f^l(U^l). \tag{5}$$

Adaptation of the input weights of layer l, on the other hand, needs the evaluation of the outer product of two vectors as follows:

$$W^l = W^l + \eta \delta^l A^{l-1}. \tag{6}$$

In the above equations, δ^l is a vector representing the δ-values of layer l neurons.

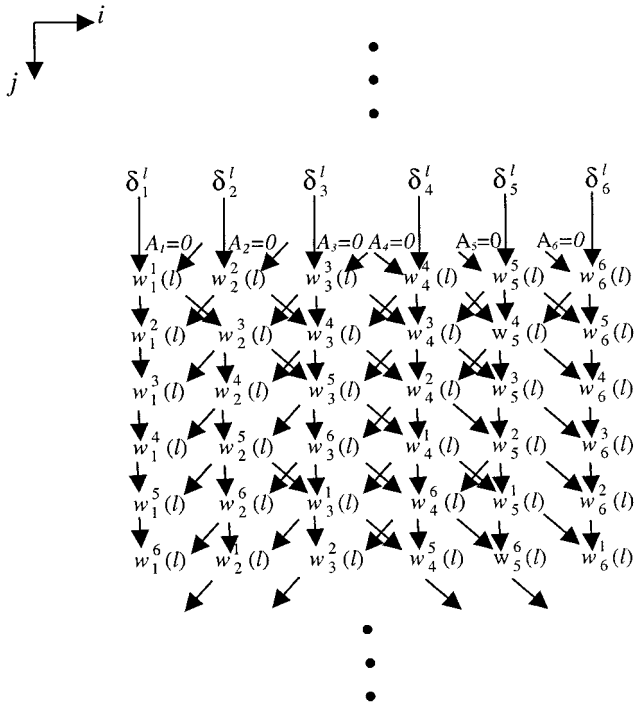

FIG. 5. DG for δ-value computation.

Calculation of δ-values. A DG for calculating the δ-values of layer l from the δ-values of its next higher layer has been shown in Fig. 5 and is found to be similar to the DG for representing the operations in the recall phase, but possesses a slightly different communication structure. The node functions are also similar and the same MAC operation is performed in each of the nodes with a different set of variables. When the learning phase starts, processor P_i, $1 \leqslant i \leqslant N$, will be having the state value a_i^L. This, together with t_i, is used to compute the value of δ_i^L and adjust the input weights of n_i^L. Afterward, Algorithm 2 is repeatedly executed with decreasing values of l, i.e., from $l = L$ to $l = 2$, in order to compute the δ-values of all the hidden layer neurons.

ALGORITHM 2. 1. Each processor initializes an accumulator $A_i = w_i^j(l)\, \delta_i^l$.

2. P_i also initializes two more accumulators A_i^1 and A_i^N to zeros and sends them respectively to P_{i-1} and P_{i+1}.

3. In each of the following $(N-1)$ cycles,

(i) Processor P_i for $1 \leqslant i \leqslant N/2$ receives the accumulator A_j^1 from P_{i+1} and updates: $A_j^1 = A_j^1 + w_i^j(l)\, \delta_i^l$. P_i then sends A_j^1 to P_{i-1}.

(ii) Processor P_i for $(N/2+1) \leqslant i \leqslant N$ receives the accumulator A_k^N from P_{i-1} and updates: $A_k^N = A_k^N + w_i^k(l)\, \delta_i^l$. P_i then sends A_k^N to P_{i+1}.

4. Additionally, in the $(N/2+1)$th cycle, processor P_i, $2 \leqslant i \leqslant N/2$, initializes a variable $A_j^N = 0$ and sends it to P_{i+1} and processor P_i, $(N/2+1) \leqslant i \leqslant (N-1)$, initializes a variable $A_k^1 = 0$ and sends it to P_{i-1}. In the above, j and k are indices of the accumulators received in the previous cycle from processors P_{i+1} and P_{i-1}, respectively.

The following data transfers are also necessary to ensure the successful execution of the algorithm:

(i) for $(i-1)$ cycles, starting with the second, processor P_i, $2 \leqslant i \leqslant N/2$, receives the data item A_k^n from P_{i-1} and simply forwards it to P_{i+1} and for $(N-i)$ cycles, starting with the second, P_i for $(N/2+1) \leqslant i \leqslant (N-1)$, receives the data item A_i from P_{i+1} and forwards it to P_{i-1}.

(ii) for $(i-2)$ cycles, starting with the $(N/2+2)$th, processor P_i for $3 \leqslant i \leqslant N/2$ receives the data item A_j^N from P_{i-1} and forwards it to P_{i+1} and for $(N-i-1)$ cycles, starting with the $(N/2+2)$th, the processor P_i for $(N/2+1) \leqslant i \leqslant (N-2)$ receives the data A_j^1 from P_{i+1} and forwards it to P_{i-1}.

5. In the next cycle, the processor P_i for $1 \leqslant i \leqslant (N/2-1)$ receives the accumulator A_i^1 from P_{i+1} and adds it to A_i. Similarly, in this cycle, processor P_i for $(N/2+2) \leqslant i \leqslant N$ receives A_i^N from P_{i-1} and adds it to A_i.

Now, three partial sums, A_j^1, A_j^N, and A_j, exist for the jth neuron in the processors P_1, P_N, and P_j, respectively. It takes an additional N cycles to collect and add these partial sums in the processor P_j to obtain the sum $\sum_{i=1}^{N} \delta_i^l w_i^j(l)$. As a_j^{l-1} is already available in P_j, it now proceeds to compute δ_j^{l-1}.

Weight adaptation. The DG for weight adaptation is nearly identical to that for the recall phase and can be derived from Eq. (3). The node functions are also similar. This DG can be processed in the same bidirectional systolic array to adjust the input weights of layer l.

4. PARTITIONING SCHEME

Choice of a suitable partitioning scheme is very important for the full utilization of the systolic array [12]. Below, it is shown that the proposed mapping scheme is useful even when a problem is to be mapped onto a smaller number of processors. In fact, for mapping a neural net of N neurons per layer onto a P processor array, the respective algorithms just have to be executed N/P times each for executing the recall and the learning phase operations. The data assignment for mapping a multilayer perceptron with 12 neurons per layer onto a 6-processor array is shown in Fig. 6. It may be observed from this figure that the processor P_i now stores the neuron $n_{i \bmod P}$. The input weights of these neurons are stored in the memory of P_i in a skewed manner. It is clear that now the respective algorithms have to be executed twice each in order to process the recall and learning phases of the back-propagation algorithm.

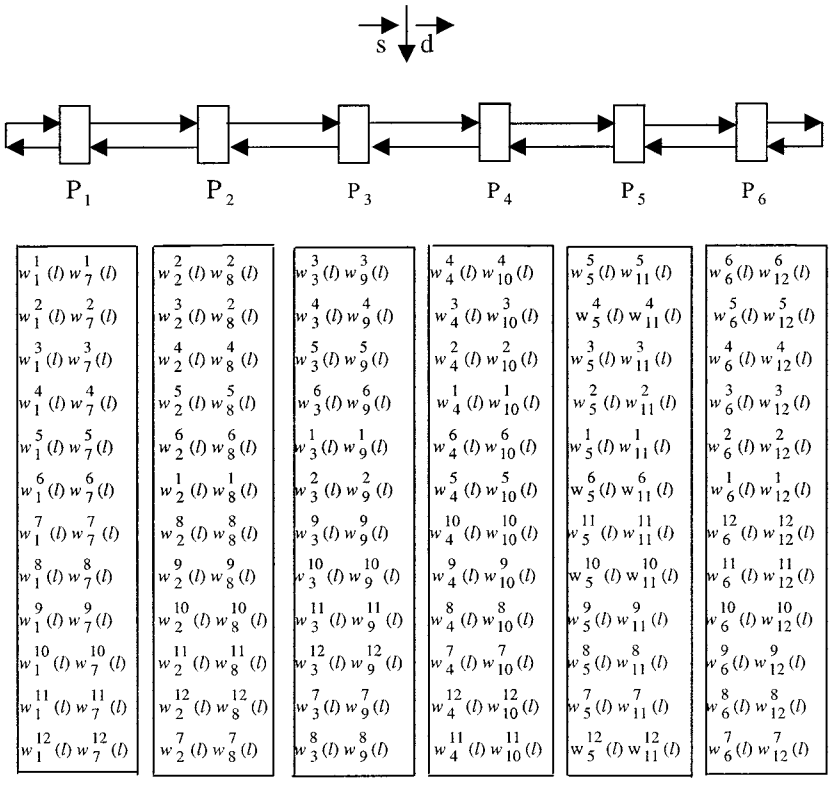

FIG. 6. Partitioned implementation of a BP net with 12 neurons per layer onto a six-processor array.

5. RESULTS AND DISCUSSION

This paper presents mapping schemes for the proposed implementation of artificial neural nets on systolic arrays. After deriving DGs for representing the operations in the different execution phases, steps are outlined to execute the back-propagation algorithm.

The performance of the proposed implementation has been evaluated by estimating the execution speedup of a linear array of T-805 transputers in executing the recall and the learning phases of a single layer of the BP net. The T-805 is a 32-bit microprocessor with four 20-Mbps serial bidirectional communication links. For implementing a neural net of n neurons per layer on a P processor array, the time for executing the back-propagation algorithm is given by: $T_P = t_r + t_\delta + t_w$, where $t_r = (n/P)(n(t_a + t_m) + t_s + CP)$, $t_\delta = (n/P)(n(t_a + t_m) + (t_a + 2t_m) + 2CP)$, and $t_w = (n/P)(n(t_a + 2t_m) + CP)$.

In the above t_a, t_m, and t_s are respectively the time to add two 32-bit numbers, the unit multiplication time, and the time to evaluate the Sigmoid function, the nonlinear activation function used in the algorithm. C is the communication time for data transfer to neighboring processors.

The uniprocessor time T_1 is obtained putting $P = 1$ and $C = 0$ in the relation for T_P. And, the P processor is obtained as: $S(P) = T_1/T_P$.

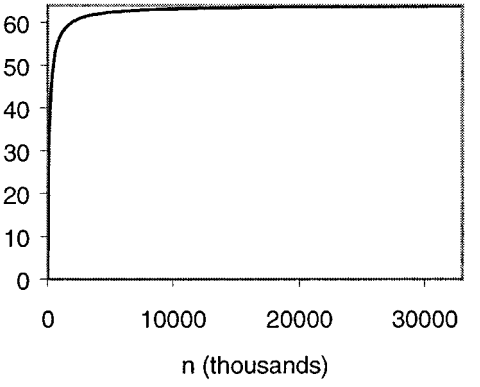

FIG. 7. Speedup versus the number of neurons per layer for a 64-processor array.

For the T-805, $t_a = 330$ ns and $t_m = 550$ ns [13]. The time to transfer a 32-bit number over the serial communication link is estimated to be 1.6 ms. Assuming the Sigmoid function to be evaluated using a range-reduction technique [5], t_s comes out to be 5.2 ms. Using the timing parameters given above, speedup of the proposed implementation has been estimated and is plotted for increasing values of n in Fig. 7. Figure 8, on the other hand, shows the variation of the expected speedup with the number of processors. Both the above figures prove that the use of the proposed mapping scheme would result in an efficient and scalable implementation.

It is observed from Algorithms 1 and 2 that the operations in the execution phases of the BP net are quite similar and involve the same communication pattern. This is because the basic computation in both is the multiplicative addition operation. In fact, this is the case with almost all of the existing neural network algorithms such as the Hopfield net, the HMM, the RBP, and the Kohonen feature map. Therefore, the proposed mapping scheme is actually valid for a wide class of neural net algorithms. In the implementation of the Hopfield net, it is possible for the net to learn as and when it is recalling a given input pattern [11]. It has also been shown that the above

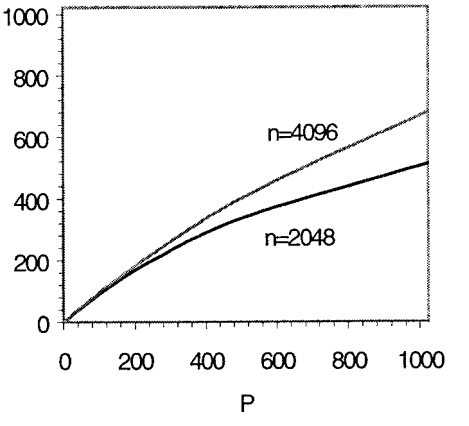

FIG. 8. Speedup versus the number of processors.

scheme is useful for the partitioned implementation of large sized neural nets in smaller processor arrays, a highly desirable feature.

In the above, only neuron level parallelism [3] has been used in the parallel implementation of neural nets. Still another possibility is present in the parallel implementation of the back-propagation algorithm, i.e., the pipelined execution of a multiple number of exemplar patterns in the layers of the neural net [11]. Systolic architectures can exploit this possibility quite efficiently due to their highly pipelined nature. The systolic array can execute the back-propagation algorithm in a pipelined manner with unity pipeline period [7]. This can greatly reduce the training time of multilayer neural nets.

The mapping scheme presented in this paper has been illustrated on uniformly structured neural networks only, for the ease of explanation. The techniques are equally valid even for nonuniformly structured networks with unequal number of neurons in the different layers. For such networks, the layer size can be made the same by adding dummy neurons with zero activation values in the layers having a smaller number of neurons. The processing proceeds in the same way as before, except that the weights between the real and dummy neurons are set to zeros. Of course, this may lead to a slight deterioration in the performance of the implementation. Finally, it is to be noted that although the proposed implementation uses a bidirectional systolic array, still, the locality property of interprocessor communication in systolic arrays is preserved, which is highly desirable for VLSI implementation [6]. The fact that the learning phase requires N additional cycles would not matter much once training is complete, because then the neural net has to execute the recall phase only for multiple input patterns that are to be classified.

REFERENCES

1. R. P. Lippman, An introduction to computing with neural nets, *IEEE ASSP Mag.* (April 1987), 4–22.
2. D. E. Rumelhart, G. E. Hinton, and R. J. Williams, Learning internal representations by error propagation, *in* "Parallel Distributed Processing," Chap. 8, MIT Press, Cambridge, MA, 1986.
3. A. Singer, Implementation of artificial neural networks on the connection machine, *Parallel Comput.* **14** (1990), 305–315.
4. H. Yoon, J. N. Hwang, and S. R. Maeng, Parallel simulation of multilayer neural networks on distributed memory multiprocessors, *Microprocess. Microprogr.* **29** (1990), 185–195.
5. S. Shams and K. W. Przytula, Implementation of multilayer neural networks on parallel programmable digital computers, *in* "Parallel Algorithms and Architectures for DSP Applications" (M. Bayoumi, Ed.), pp. 255–279, Kluwer Academic, Dordrecht/Norwell, MA, 1991.
6. S. Y. Kung, "VLSI Array Processors," Chap. 4, Prentice–Hall, Englewood Cliffs, NJ, 1988.
7. S. Y. Kung and J. N. Hwang, A unified systolic architecture for artificial neural networks, *J. Parallel Distrib. Comput.* **6** (1989), 358–387.
8. J. N. Hwang, J. A. Vlontzos, and S. Y. Kung, A systolic neural network architecture for hidden Markov models, *IEEE Trans. on ASSP* **32**, 12 (Dec. 1989), 1967–1979.
9. J. Chung, H. Yoon, and S. R. Maeng, A systolic array exploiting the inherent parallelism of artificial neural networks, *Microprocess. Microprogr.* **33** (1991/92), 145–159.
10. C. Lehmann, M. Viredaz, and F. Blayo, A generic systolic array building block for neural networks with on-chip learning, *IEEE Trans. Neural Networks* **4**, 3 (May 1993).

11. S. Mahapatra, "Some Studies on Mapping of Neural Network Models onto Parallel Architectures," Ph.D. thesis, I.I.T., Kharagpur, 1997.
12. J. J. Navarro, J. M. Labera, and M. Valero, Partitioning: An important step in mapping algorithms into systolic array processors, *IEEE Comput*. (July 1987), 77–89.
13. Transputer Education kit, Computer System Architects, Provo, Utah, 1990.