Efficient Distribution of Back-Propagation Models on Parallel Architectures

Louis G. Ceci
Patrick Lynn
Phillip E. Gardner

CU-CS-409-88

University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Efficient Distribution of Back-Propagation Models on Parallel Architectures

Louis G. Ceci, Patrick Lynn, Phillip E. Gardner

CU-CS-409-88 September 1988

Department of Computer Science
Campus Box 430
University of Coloado
Boulder, Colorado 80309

Efficient Distribution of Back-Propagation Models on Parallel Architectures

Louis G. Ceci, Patrick Lynn, and Phillip E. Gardner

University of Colorado, Boulder

Abstract.

This paper looks at three different schemes for distributing a back-propagation model with three layers of trainable weights. We attempt to predict the relative speed-up each scheme can achieve and determine under which conditions a given scheme is optimal. The three schemes fall into two broad classes: distribution-by-pattern, in which the patterns to be learned are distributed over several processors, and distribution-by-layer, in which the network itself is distributed over several processors. Distribution-by-layer can be done two ways. The first, which we called the simple pipe, puts each layer of the network on a separate processor. Each processor in this scheme does both forward and backward passes. In the second scheme, we separated not only the layers but the forward and backwards passes as well. We called this scheme the cube-implmented ring because of the communications pathways it requires.

We divised mathematical models of the computation and communication costs of all three schemes and compared them to the computation cost of a single processor. Our models predict that, when there are fewer than 75 patterns to learn or fewer than 20 units per layer, the single processor is fastest; for all other conditions, one of the three other schemes is better.

Section 0. Introduction.

Connectionist networks are intrinsically appealing for modeling certain problems in artificial intelligence (AI). They have succeeded at pattern recognition and classification tasks, speech generation from printed text, and machine vision--difficult areas for most symbolic AI models [4]. Back-propagation models have been particularly useful. They "learn from experience," changing the weights on links between units so that the entire network maps input patterns to output patterns more accurately, even when the input patterns are incomplete or corrupted by noise. This fault-tolerant acquisition of behavior in connectionist networks mimics, sometimes surprisingly well, the process living creatures go through in acquiring new behaviors [8].

Unfortunately, back-propagation models are also quite slow, often taking hundreds, sometimes thousands of epochs--complete presentations of the input-output pattern set--before converging to a solution. Several proposals for speeding up the learning algorithm in back-propagation networks have been made, including using a momentum term and adaptively changing the learning rate (see [2]). This paper takes a different approach. We seek to speed up the back-propagation model by distributing the computational cost of the model over several processors, each running a different part of the model or training on some subset of the input-output pattern pairs.

Our investigation reveals there are three key issues that affect the speed of the back-propagation model: the size of the largest "layer" or vector of units in the model, the number of layers, and the size of the set of input-output pairs to be learned. We chose to limit our consideration to a network with three trainable weight matrices: a layer of input units, a layer of output units, and two layers of hidden units. We chose this architecture for the model because a network with two hidden layers can divide any decision space into arbitrarily complex regions [10]. It is therefore unlikely that a back-propagation model would need to have more than three weight matrices. To model vector size, we chose a worst-case scenario in which all the vectors are the same size: the output layer is the same size as the input layer and there is no compression of data in the hidden layers. We augmented each layer with a threshold unit, and connected all the units in one layer to the non-threshold units of the next layer in the network. A diagram of the resulting model is given in Figure 1.

Section 1. The Computational Cost of Back-Propagation Models with Three Train-

able Layers.

We begin our analysis by estimating, in terms of n, the largest vector in the model, the computational cost of running the three-layer back-propagation model on a single processor. To help clarify our discussion of the cost, we define six terms:

activation vector is the vector of values computed by taking the inner product of the input vector and each row of the weight matrix, and then passing the resulting products through an activation function. By our worst-case scenario, each activation vector is of size n. Activation vectors are symbolized by **a**.

input vector is the activation vector computed by a previous layer in the network plus a threshold unit. (In the case of the first layer of the network, the activation vector is presumably fetched from a table of input patterns.) Each input vector is therefore of size $n + 1$; they are symbolized by **i**.

weight matrix is the matrix holding the weights representing the strength of connections between each unit of the input vector and each unit of the outgoing activation vector. Each weight matrix is of size $n^2 + n$, and is symbolized by **W**.

delta vector is the vector of values that represent the error signal passed down from a higher layer. All delta vectors are size n and are symbolized by **d**.

change-of-weight matrix is a matrix of values that accumulates the changes in weights mandated by the error detected on each presentation of an input-output pair. Each change-of-weight matrix is size $n^2 + n$ and is symbolized by

ΔW.

a trainable layer consists of an input vector, a weight matrix, a delta vector, and a change-of-weight matrix. The input layer and the two hidden layers are trainable; the output layer, containing only the activation vector it receives from the second hidden layer and a table of target vectors, is not.

A typical trainable layer is schematically diagramed in Figure 2. This diagram shows the four tasks each trainable layer must perform and the resources it needs to complete them. Three of these tasks, indicated by solid arrows, must be done on each presentation of an input-output pattern pair; the fourth task, updating the weight matrix, is indicated with a dashed arrow and is done only at the end of an epoch (for reasons that will be given below). In greater detail, these tasks are:

1. Using the input vector and the weight matrix to compute the activation vector for the next higher layer. The inner product calculation costs $(n+1) * n$, and passing the results through the activation function costs an additional n operations, giving this task a total cost of $n^2 + 2n$.

2. Using the delta vector and the input vector to compute the change in weights, and storing those changes in the change-of-weight matrix. This requires $n^2 + n$ operations.

3. Using the activation vector, the delta vector, and the weight matrix to compute the delta vector for the next layer down in the network. Like the forward pass that computes activation, the back-propagation of error takes two steps. First, the inner product of the delta vector and each *column* of the weight matrix is computed. (The column correspond-

ing to the threshold weights is not used since the threshold unit does not propagate error.) The resulting product is multiplied by the first derivative of the element of the activation vector corresponding to the column of the weight matrix that was used in computing the inner product. The resulting cost for computing the delta vector is thus n^2+n.

Actually, only the trainable layers associated with the two hidden layers incur this cost. The input layer has no lower layer to pass error on to, and so skips this task. The output layer computes its error signal simply by subtracting the activation it receives from the second hidden layer from the appropriate target vector and multiplying the difference by the first derivative of the activation, a $2n$ operation.

4. After all the input-output pattern pairs have been presented, the weight changes stored in the Δ **W**s are added to the values in the **W**s. This updating process costs n^2+n operations.

If the weight changes calculated in task 2 were immediately added to the weight matrices, step 4 could be eliminated and the memory space for the three n^2+n Δ **W**s could be freed. But there are two reasons not to do this. First, as Rumelhart, Hinton, and Williams point out in their proof of the generalized delta rule, in order to guarantee gradient descent in the error space, the weight matrices used to compute the delta vectors must remain "uncorrupted" by weight changes until all the patterns have been presented [7]. Second, updating the weight matrices after each pattern presentation greatly increases the message traffic in some of the distribution schemes outlined below, thereby reducing their ability to speed up the network's over-all performance.

In summary, then, the total computational cost of a single processor running a three-

trainable layer back-propagation network through one epoch of m patterns is

$m(3(n^2+2n))$ forward passes

$m(3(n^2+n))$ change-of-weight calculations

$m(2n)$ first deltas

$m(2(n^2+n))$ hidden deltas

$3(n^2+n)$ updates

for a total of

$$m(8n^2+13n)+3(n^2+n). \tag{1}$$

Section 2. Analysis of Three Parallelizing Schemes.

We examined three schemes for improving on the computational cost of the three-trainable layer network. Two of them involve pipelining the model and the third involves distributing the number of patterns to be learned over several processors.

Pipelining the network involves splitting the network up and spreading its layers around on the available processors, giving each layer a processor of its own. The first pipelining scheme is conceptually diagramed in Figure 3 with dashed lines separating the different processors. The purpose of this scheme is to speed up processing time by pipelining the forward and backward passes. The input layer processor does not have to wait for the error signal to come back down the pipe before processing the next pattern. The same is true for the delta vectors. As soon as the last activation vector from the second hidden layer has reached the fourth processor, the output layer can begin passing delta vectors

back down the pipe. This scheme has the advantage of spreading the memory load across four processors: no one processor has to hold the whole model. We call this scheme the "simple pipe."

To analyze the differences between schemes, and in particular to be able to compare their relative costs in computation and communication, we have broken the pipeline schemes down into a number of steps. A step consists of all computations up to the moment when one processor must communicate with another. Each processor must wait until the processor calculating the lengthiest computation for that step has finished. This lenthiest computation is considered "visible," while faster, shorter computations going on in the other processors are considered "invisible." The total computational cost of the scheme is then calculated as the sum of the visible computations for each step. Of course, all computations are visible in the single processor scheme, since it never needs to communicate with another processor. The steps for the simple pipe and the cube-implemented ring (discussed below) are diagramed in Figure 4; a side-by-side comparison of the computational load for a single processor, the simple pipe and the cube-implemented ring are given in Figure 5.

Even with this "lock-step" restriction, the simple pipe is computationally faster than the single processor. First, while each first delta calculation is visible in the single processor version, only one is visible in the simple pipe (the rest are obscured by the computationally more intensive forward passes going on at the same time). Second, only one weight matrix update is counted at the end of the epoch because only the update done by the first hidden layer is visible. The entire computational cost for the simple pipe is

$$(m+2)(n^2+2n)+2n+(m+1)(2n^2+2n)+(n^2+n)\,. \tag{2}$$

By comparison, expression (1) cast in the same terms, is

$$3m(n^2+2n)+m(2n)+2m(2n^2+2n)+3(n^2+n)+m(n^2+n)\,. \tag{1a}$$

(The additional $m(n^2+n)$ term is to account for the change of weight calculations in the

input layer.) Both pipelining schemes take advantage of the fact that the input layer processor can continuously update its weight matrix because it doesn't use the matrix to compute a delta vector. Since (2) will always yield a smaller value than (1a), the simple pipe will always be *computationally* faster than the single processor.

However, there are communication costs incurred by the simple pipe, and the single processor has none. The simple pipe has to pass one message of size n for every step except the last. For m patterns, there will be $2m+4$ such message-passing steps. In general, the cost of sending a message can be represented by the expression

$$\alpha + \beta kn \,, \tag{3}$$

where α is the start-up cost of sending a message, β is the speed at which a single byte can be transmitted, k is the number of bytes in each element of the vector being passed between layers; and n is, as always, the size of the vector in number of elements [3], [6]. The communications cost of the simple pipe is

$$(2m+4)(\alpha + \beta kn)\,. \tag{4}$$

To make our comparisons concrete, we chose to estimate the actual time each scheme consumed in communication and calculations. For communication costs, we chose values for α, β, and k from published results on the Intel iPSC [6]; they are $\alpha = 170 \times 10^{-5}$ secs, $\beta = .283 \times 10^{-5}$ secs, and $k = 4$. On somewhat shakier grounds, we assumed that each operation figured into the computational cost of a scheme consisted of a single floating-point operation, and then used Intel's advertised computational speed for the iPSC of 2 MFLOPS. The results of plugging these values into the cost expressions for single processor, simple pipe, cube-implemented ring, and distribution of patterns over four processors is given in Figure 6. Even with the communications costs added in, the simple pipe outperforms the single processor when $n > 45$.

Despite its efficiency, the simple pipe has its shortcomings, as Figure 6 suggests. Having only one processor per layer imposes certain restrictions on the parallelism. For one,

there is uneven load balancing among the processors. The output layer processor is almost always idle, waiting for the second hidden layer's output. All the other processors do both forward and backward passes, which means that calculation of the change in weights has to wait until all the forward passes are through. Consequently, all processors have to hold copies of their input vectors until the appropriate delta vector gets passed on down the pipe, adding $m(n-1)$ in memory overhead.

We overcame these shortcoming by collapsing the output layer with the second hidden layer and splitting the forward and backwards passes between processors. The resulting distribution scheme is diagramed in Figure 7a. We call this scheme a "cube-implemented ring" because computation flows around the six processors as if they were in a ring, but the necessity of matching input vectors with appropriate delta vectors requires the kinds of communications links typically associated with a 3-dimensional hypercube. Figure 7b shows the cube-like structure more clearly.

The computation costs of the cube-implemented ring turn out to be less than the simple pipe, but the communication costs are higher. The computation cost is

$$3(n^2+2n)+2n+(m+1)(2n^2+2n)+(n^2+n)\,. \tag{5}$$

The communication cost during the processing of pattern pairs is based on the following assumptions. First, most processors must send two messages between computational steps. Processors I and H send the activation vectors they compute to processors H and H/O, respectively, then send their input vectors to processors i and h, respectively. At the same time, processor H/O sends processor h/o two vectors: the first delta calculated by H/O and the H/O input vector. (Processors h/o and h send the delta vectors they calculate to h sends i, respectively.) Second, even though input vectors are size $n+1$, all the messages sent are assumed to be size n because each receiving processor can easily add the threshold unit to its input vector without having it explicitly sent. Further, because most of these messages travel along separate paths, it is assumed that two or more

processors can send their messages without having them collide with each other. Consequently, the time taken between computational steps during pattern presentations is at most $2(\alpha + \beta kn)$.

This worst-case traffic cost occurs for every pattern pair, and continues for two steps after the last pattern is processed by I as it percolates around the pipe to H and then to H/O. For two more steps, the communications cost is $\alpha + \beta kn$ as processors h/o and h propagate the error of the last pattern to h and i. Finally, after each lower-case processor updates its weight matrix, it ships the new matrix off to its upper-case counterpart. Since, again, this takes place simultaneously on all three paths, the visible cost of shipping the updated matrices is $\alpha + \beta k(n^2 + n)$. Altogether, then, the communication cost of the cube-connected ring is

$$2(m+2)(\alpha + \beta kn) + 2(\alpha + \beta kn) + (\alpha + \beta k(n^2 + n)). \tag{6}$$

This rather awkward communication cost expression does not bode well for the cube-implemented ring. However, when $n > 40$ and $m > 30$, the cube-implemented ring is faster than the simple pipe, and the ring is faster than the pipe with as few as 25 patterns to learn if $n > 100$ (see Figure 6).

The third distribution scheme is perhaps the most immediately apparent from expression (1). By distributing the m patterns to be learned over p processors, the computation cost in (1) can be reduced to

$$\frac{m}{p}(8n^2 + 13n) + 3(n^2 + n). \tag{7}$$

In this scheme, which we call distribution by pattern, each processor has a complete version of the back-propagation model. One processor, the "master," is responsible for updating the weight matrices based on the change-of-weight matrices sent to it by the "slave" processors at the end of each epoch. Communication in this scheme is expensive, since it requires sending current (updated) weight matrices to each slave processor at the start of each epoch and change-of-weight matrices back to the master at the end. These

matrices are size n^2+n, there are three of them, and they must be sent to and from the p processors. The communication cost is thus

$$2p(\alpha+3\beta k(n^2+n)). \tag{8}$$

As in the other schemes, this communication cost assumes that processors don't share memory.

Figure 6 shows how the distribution-by-pattern scheme fares against the two pipelining schemes and the single processor when $p = 4$, the number of processors in our simulation. As might be expected, the principle strength of distribution-by-pattern shows up when m is large. However, Figure 6 also shows the overwhelming effect of the communication cost as n also grows large: eventually, the cube-implemented ring becomes more efficient, and even a single processor runs through the patterns faster than multiple processors when $m < 70$ and $n < 15$.

Section 3. Conclusions and Caveats.

Before reporting the results of our implementation of the distribution-by-pattern scheme, we should note that several features typical of actual back-propagation models and real parallel architectures are ignored or glossed over by our cost models.

First, the models assume an equal number of units for all layers, and, consequently, nearly square weight matrices. This is rarely the case in back-propagation models. Indeed, one application of the back-propagation model is to have hidden layers with significantly fewer units than either the input or the output layer. Such models learn internal representations of features present (but sometimes unknown) in the input [7]. Smaller hidden layer vectors can also be thought of as shrinking the bandwidth of the incoming data. Both the cube-implemented ring and the distribution-by-pattern scheme, whose communication costs include an n^2+n term, can expect to do better than our cost models predict since the weight matrices they ship around will not all be the maximum

size possible.

All three distribution schemes require some set-up. Distribution-by-pattern requires setting up the entire model on each slave processor, as well as sending each processor its share of the input and teaching patterns. Since all slaves need to set up identical copies of the network, a broadcast primitive can be used to communicate the model's parameters to the slaves, thus saving some time. That is not the case with either the simple pipe or the cube-implemented ring. In those schemes, only a portion of the model is set up on each processor and, save for the two hidden layer processors in the simple pipe, each processor is running a structurally different part of the network. It is hard to say which scheme will have the longest set-up time and how set-up time will affect the overall cost.

Further, none of the schemes takes particular account of the message-passing hardware or special facilities to speed message-passing. A factor which may increase the communications costs for the Intel hypercube is its use of an ethernet connection between processors; on such an architecture, only one message may be passed at a time. Consequently, time may be lost when collisions occur. On the other hand, the Intel hypercube allows the use of broadcast primitives. Not only would these speed the initial set-up of the distribution-by-pattern scheme, as mentioned above, but they could also be used when the updated weight matrices are sent to all slave processors. This would reduce the factor multiplying the communications cost in (8) from $2p$ to $2\log_2 p$.

Finally, we should note that of all the schemes, distribution-by-pattern is probably the easiest to code. (In fact, it is the only one we empirically tested.) Of course, different machines will allow different degrees of ease in parallel programming, but the simple pipe requires three different programs (the two hidden layers are the same), and the cube-implemented ring, six (one for each processor). Distribution-by-pattern, on the other hand, requires only two programs, master and slave, regardless of how many slave processors there are. Thus the software development costs of pipe and ring may well be

a real and limiting factor in their implementation.

Nevertheless, we believe our cost models are useful. They allow abstract comparisons to be made between distribution schemes, and the results of our simulation of a scaled-down version of the distribution-by-pattern scheme using only one hidden layer supports our analysis.

From our cost models and our simulation, we conclude that, for moderately high numbers of patterns to be learned, distribution-by-pattern is faster than a single processor, and for even modest numbers of nodes per layer, the simple pipe and the cube-implemented ring can usually outdo them both. Above 25 patterns, the simple pipe's need to wait until all patterns have been presented before back propagation begins turns out to be longer than the message-passing costs of the cube-implemented ring, which doesn't have to wait and is consequently faster.

There may be a way to combine the advantages of distribution-by-pattern and the other two distribution schemes. One can imagine a cube which has, at each vertex, the architecture to support either the simple pipe or the cube-implemented ring. Such an architecture would be a variation on the cube-connected cycles discussed by Preparata [5] with a cube instead of a ring at each vertex. This is, in fact, exactly the architecture of a 6-cube, like Intel's iPSC model d6. Either the simple pipe or the cube-implemented ring could be implemented on the 3-cubes that constitute the "corners" of the larger cube. The choice between simple pipe and cube-implemented ring would depend on how great a reduction in m is achieved by dividing the patterns among the eight 3-cubes. As Figure 6 indicates, a lower m per sub-cube may drop a simulation which was running faster on the cube-implemented ring into the range where a simple pipe is faster.

The only area of the graph in Figure 6 where a single processor does best is a narrow corner in the pattern-node plane. However, this is the corner where most students first

learn how to build connectionist AI models. To reach the truly complex models connectionists envision (and some are already building), either distribution schemes for parallel processors or special purpose processors, such as might be implemented on specially designed VLSI chips (see [9]) or in an optical computer (see [1]), are needed. We hope the distribution schemes described here will assist connectionist AI researchers in exploring the capabilities of truly large networks working on large sets of input-output pattern pairs.

Section 4. Acknowledgements.

Discussions of schemes for distributing the processing load of connectionist networks by the PDP Research Interest Group at the University of Colorado, Boulder, first sparked our interest in this topic. We are particularly indebted to Paul Smolensky, Dennis Heimbigner and Rod Brittner for insights into distribution schemes. The elaboration of the specific schemes, the mathematical cost models, and the analysis of data are, however, our own.

References.

[1] Anderson, D. Z., Lippincott, W. L., & Lee, J. N. Optical implementation of associative networks with versatile learning capabilities. *Applied Optics*, 26 (1987), 5039-5054.

[2] Chan, L-W., & Fallside, F. An adaptive training algorithm for back propagatin networks. Cambridge University Engineering Department, Tech. Rpt. F-INFENG/TR.2, 1987.

[3] Kadri, R. A discrete model of the neuron and its implications towards a concept of universal automata. *IEEE First International Conference on Neural Networks*, San Diego, June 1987, Vol. III, 149-156.

[4] MacLennan, B. J. Technology-independent design of neurocomputors: The universal field computer. *IEEE First International Conference on Neural Networks*, San Diego, June 1987, Vol. III, 39-49.

[5] Preparata, F. P., & Vuillemin, J. The cube-connected cycles: A versatile network for parallel computation. *Communications of the ACM*, 24 (3), May 1977, 458-473.

[6] Reed, D. A., & Grunwald, D. C. The performance of multicomputer interconnection networks. *Computer*, 20 (6), June 1987, 63-73.

[7] Rumelhart, D. E., Hinton, G. E., & Williams, R. J. Learning internal representations by error propagation. In Rumelhart, D. E., & McClelland, J. L., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Vol. 1, Foundations. MIT Press, 1986.

[8] Rumelhart, D. E., & McClelland, J. L. On learning the past tense of English verbs. In McClelland, J. L., & Rumelhart, D. E., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Vol. 2, Psychological and Biological Models. MIT Press, 1986.

[9] Smith, A. V. W., and Butler, Z. F. Bit-serial VLSI neural networks. *IEEE Conference on "Neural Information Processing Systems--Natural and Synthetic,"* Abstracts, Denver, November, 1987, p. 44.

[10] Weiland, A., & Leighton, R. Geometric analysis of neural network capability. *IEEE First International Conference on Neural Networks*, San Diego, June 1987, Vol. III, 385-392.

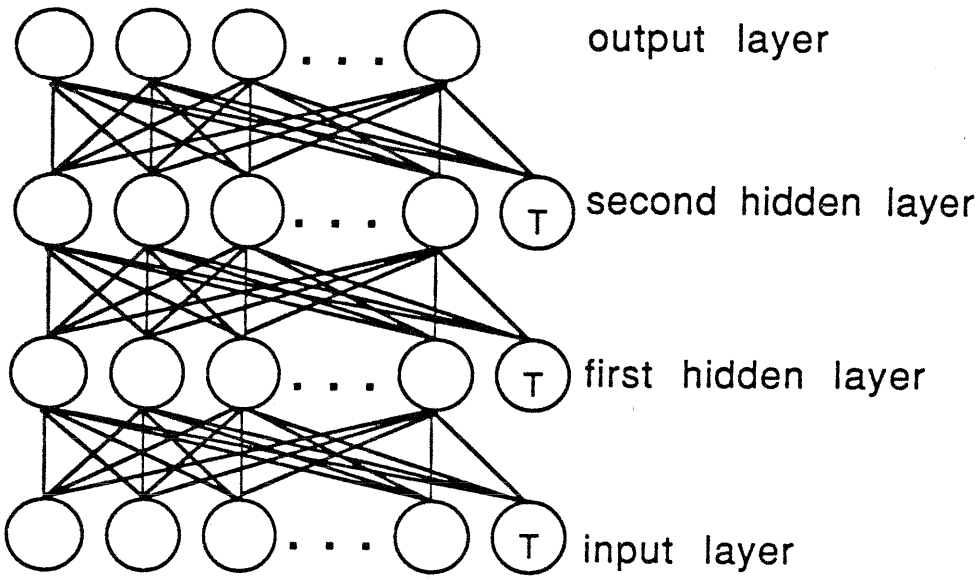

Figure 1. A Three-Trainable Layer Back-Propagation Model

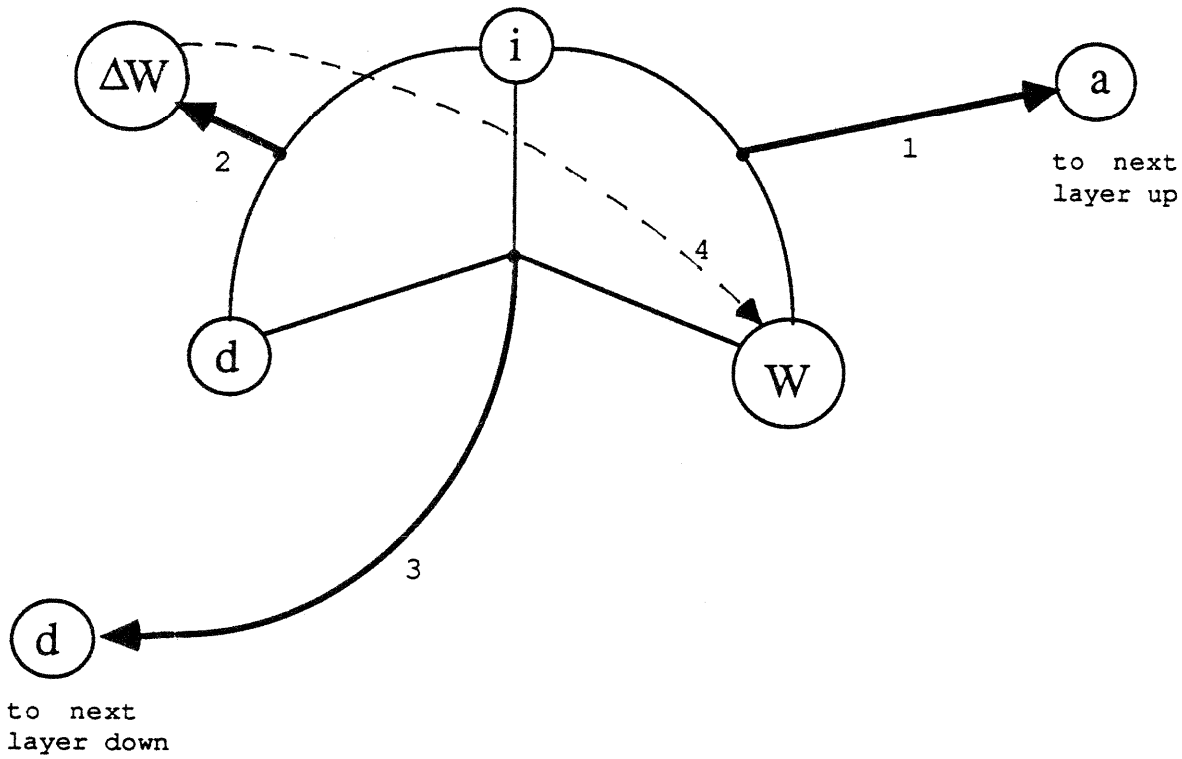

Figure 2. A Typical Trainable Layer

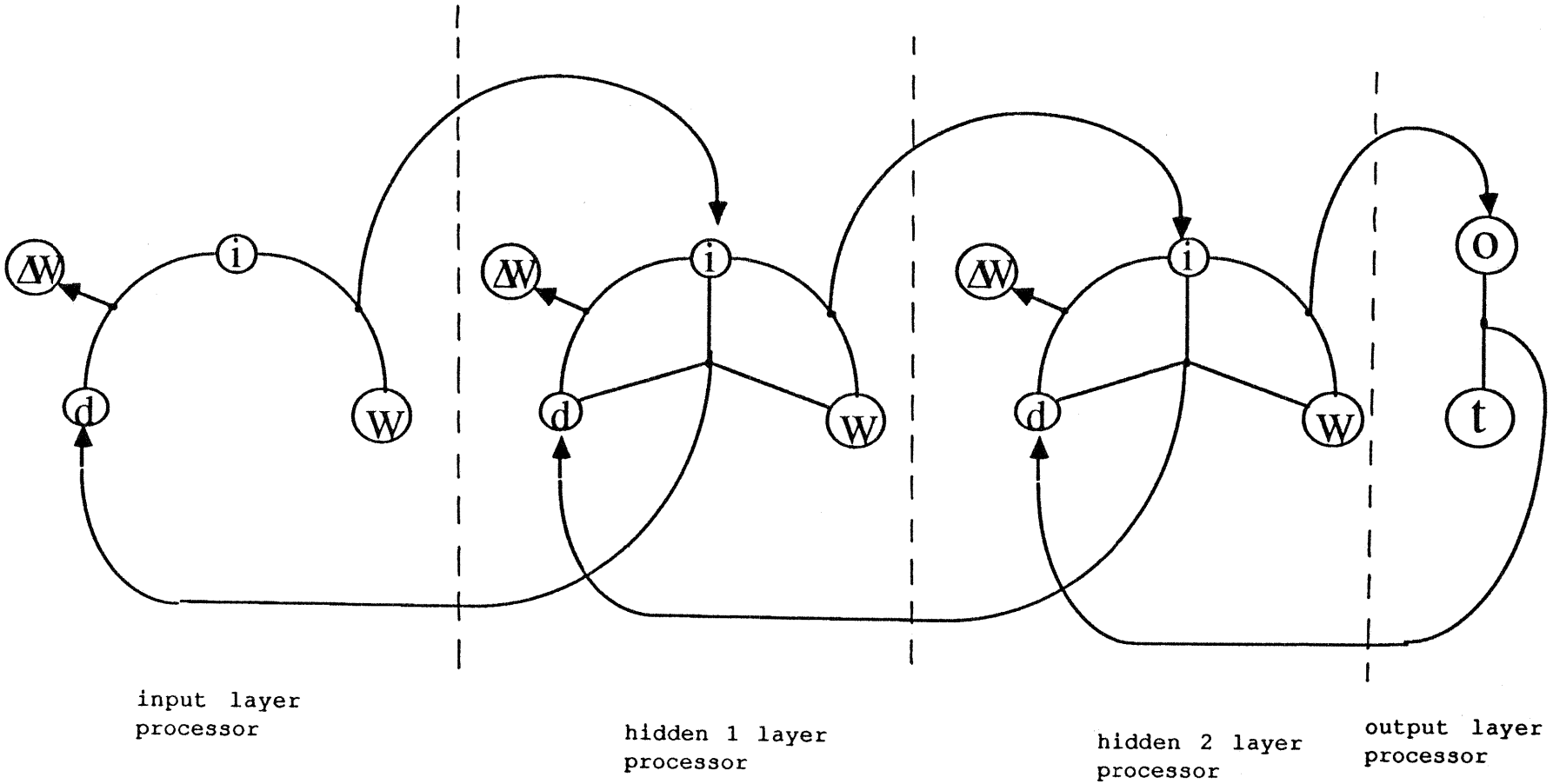

Figure 3. A Simple Pipelining Scheme

FIGURE FOUR: LOCK-STEP ANALYSIS OF PIPLINING

Step	Processor: I	H1	H2	O	Cost of Visible Computation
1.	a_1				
2.	a_2	a_1			
3.	a_3	a_2	a_1		
	⋮	a_3	a_2	e_1	(m+2)a
m.	a_m	⋮	a_3	e_2	
m+1.		a_m	⋮	e_3	
m+2.			a_m	⋮	
m+3.				e_m	e
m+4.			$w+d_1$		
m+5.		$w+d_1$	$w+d_2$		
m+6.	$+u_1$	$w+d_2$	$w+d_3$		
	$+u_2$	$w+d_3$	⋮		(m+1) w+d
2m+3.	$+u_3$	⋮	$w+d_m$		
2m+4	⋮	$w+d_m$	u		
2m+5.	$+u_m$	u			u

COMPUTATIONAL COST OF THE SIMPLE PIPE.

Step	Processor: I	H	H/O	h/o	h	i	Cost of Visible Computation
1.	a_1						2a
2.	a_2	a_1					
3.	a_3	a_2	a_1+e_1				a + e
.		a_3	a_2+e_2	$w+d_1$			
m.	a_m		a_3+e_3	$w+d_2$	$w+d_1$		
m+1.		a_m		$w+d_3$	$w+d_2$	$+u_1$	(m+1) w+d
m+2.			a_m+e_m		$w+d_3$	$+u_2$	
m+3.				$w+d_m$		$+u_3$	
m+4.				u	$w+d_m$		
m+5.					u	$+u_m$	u

COMPUTATIONAL COST OF THE CUBE-IMPLEMENTED RING.

Key: a = cost of calculating activation: $n^2 + 2n$
e = cost of calculating first delta: $2n$
w+d = cost of calculating weight changes and next delta: $2n^2 + 2n$
u = cost of updating matrix: $n^2 + n$
+u = cost of keeping a running updated matrix: $n^2 + n$

	Single Processor	Simple Pipe	Cube-Implemented Ring
tasks			
activation (a)	3m	m + 2	3
first delta (e)	m	1	1
weight changes and next delta (w + d)	2m	m + 1	m + 1
updates (u)	3	1	1
weight changes in input layer	m	0	0

A SIDE-BY-SIDE COMPARISON OF THE FACTORS MULTIPLYING THE COMPUTATIONAL COSTS OF VARIOUS TASKS IN THE SINGLE PROCESSOR, SIMPLE PIPE, AND CUBE-IMPLEMENTED RING.

Figure 5. Cost Comparisons.

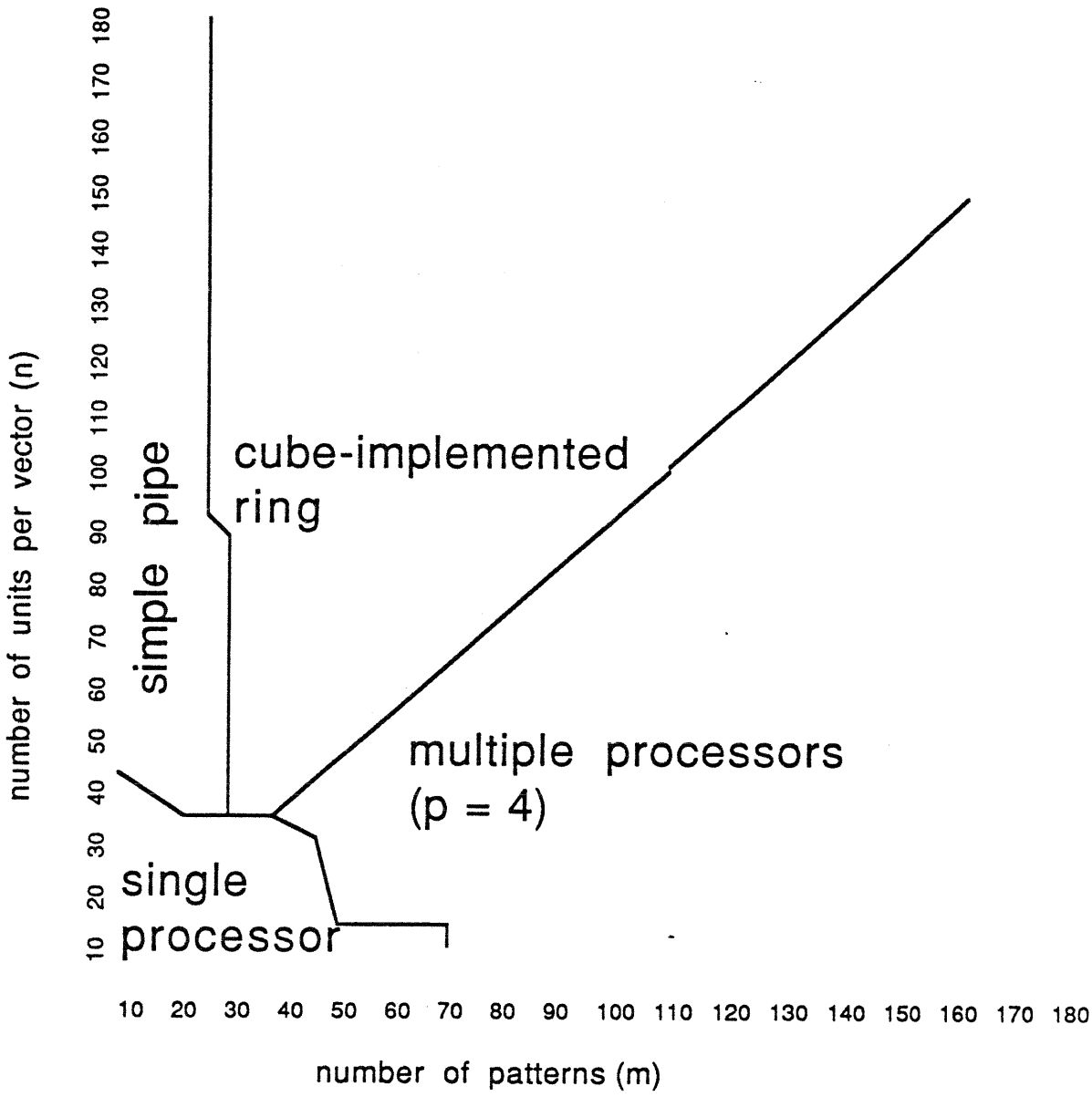

Figure 6. Areas of the Pattern-Units Plane Where the Different Schemes are Fastest

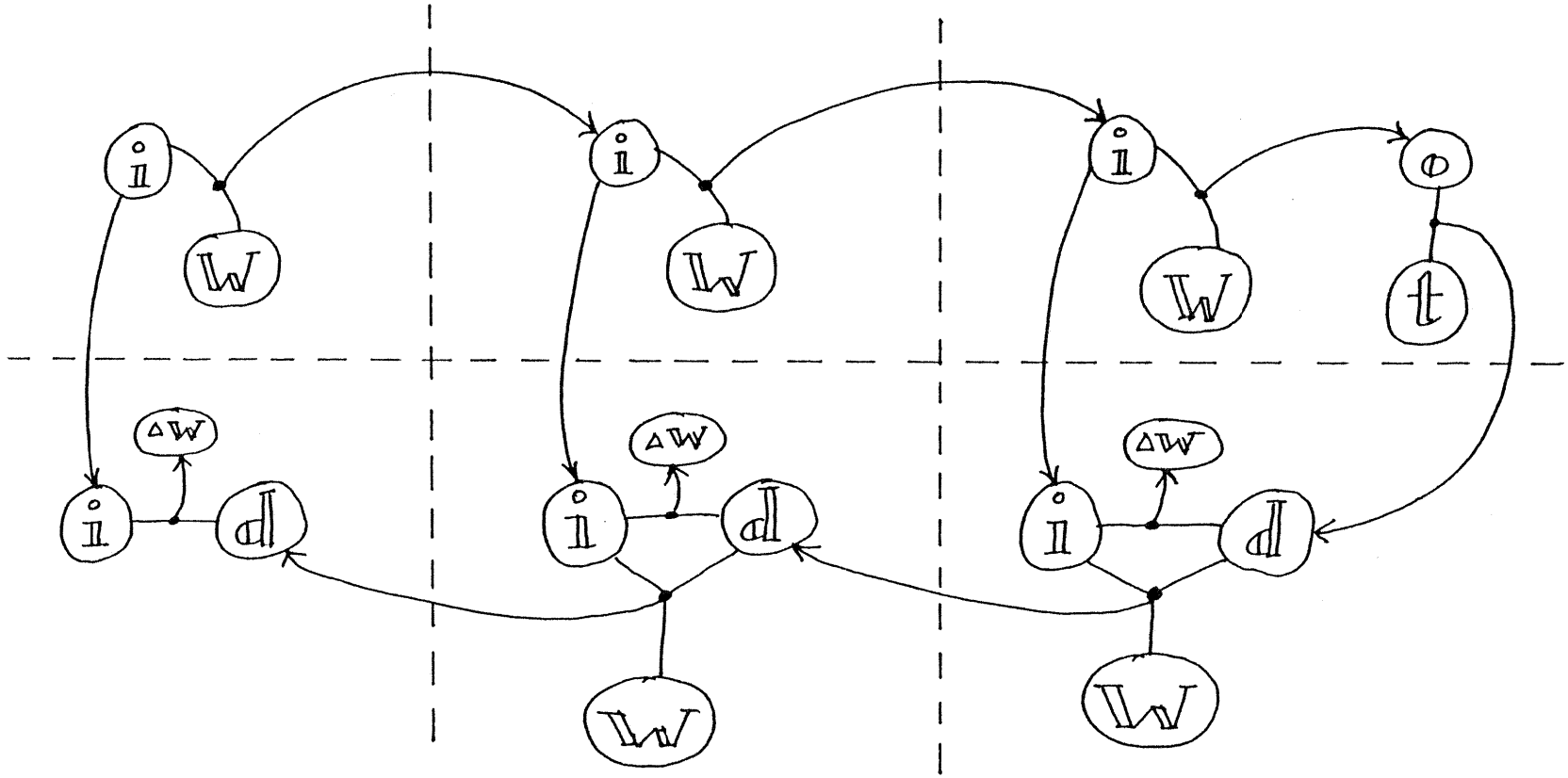

Figure 7a. Separating the Forward and Backward Flows: The Cube-Implemented Ring Scheme.

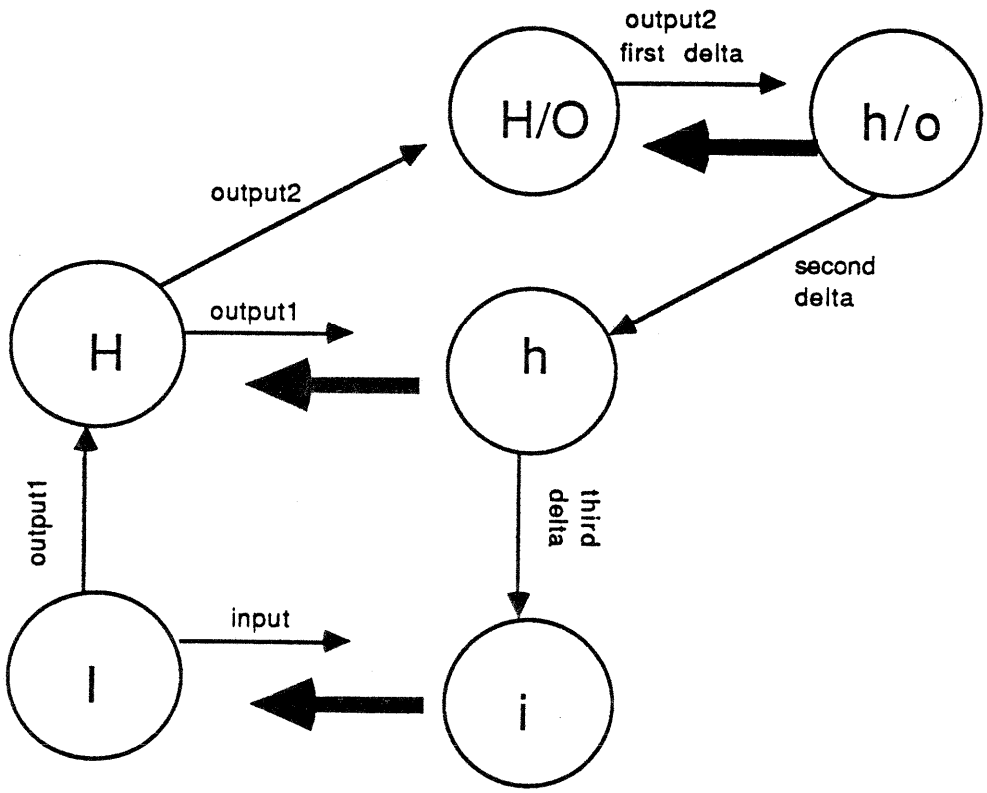

Figure 7b. Cube-Implemented Ring. Thick Arrows Indicate Weight Matrices Passed at End of Epoch.