Network Learning on the Connection Machine

Guy Blelloch
M.I.T Artificial Intelligence Laboratory
545 Technology Square
Cambridge, Massachusetts 02139

Charles R. Rosenberg
Cognitive Science Laboratory
Princeton University
Princeton, New Jersey 08542

Abstract

Connectionist networks are powerful techniques, inspired by the parallel architecture of the brain, for discovering intrinsic structures in data. However, they are not well suited for implementation on serial computers. In this paper, we discuss the first implementation of a connectionist learning algorithm, error back-propagation, on a fine-grained parallel computer, the Connection Machine. As an example of how the system can be used, we present a parallel implementation of NETtalk, a connectionist network that learns the mapping from English text to the pronunciation of that text. Currently, networks containing up to 16 million links can be simulated on the Connection Machine at speeds nearly twice that of the Cray-2. We found the major impediment to further speed-up to be the communications between processors, and not processor speed per se. We believe that the advantage for parallel computers will become even clearer as developments in parallel computing continue.

1 Introduction

Massively parallel, connectionist networks have undergone a rediscovery in artificial intelligence and cognitive science, and have already lead to broad application in many areas of artificial intelligence and cognitive simulation, including knowledge representation in semantic networks [8], speech recognition [4,17], dimensionality reduction [12,18], and backgammon [16]. However, connectionist networks are computationally intensive, and days or weeks are often required to train even moderate-sized networks using the fastest serial computers; The exploration of large networks consisting of more than a million links or connections is barely feasible given current technology.

One option being explored is the development of special-purpose hardware using VLSI technology [15,5,1]. Initial estimates of these so-called neuromorphic systems indicate that tremendous speed-up may be achievable, perhaps up to five to six orders of magnitude over a VAX780 implementation. However, one problem with special purpose hardware is that it does not allow one to explore different patterns of connectivity and different learning algorithms. Although neuromorphic systems will certainly have an important impact on the field, they may be limited as research tools.

A more flexible alternative is seen in the development of general purpose, fine-grained, parallel computers. The Connection Machine (CM) is a massively parallel computer consisting of up to 65,536 (2^{16}) one-bit processors arranged in a hypercube architecture [7]. In this paper we discuss the implementation of a connectionist network learning algorithm, back-propagation [11], on the Connection Machine. Currently, the Connection Machine offers a factor of 500 speedup over a previous implementation on a VAX780 and a factor of two speed-up over an implementation of a similar network on a Cray-2. Considering that parallel computing is only in it's infancy, we expect the speed-up to be much greater in the near future. Finally, we present an application in the domain of speech synthesis, called NETtalk [14], that uses our implementation of back-propagation on the Connection Machine.

[1]This research was supported by Thinking Machines Corporation, Cambridge, Mass. The authors wish to thank Terry Sejnowski, David Walts, and Craig Stan fill for their assistance.

2 Error Back-Propagation

Connectionist network models are dynamic systems composed of a large number of simple processing units connected via weighted links, where the state of any unit in the network depends on the states of the units to which this unit connects. The values of the links or connections determine how one unit affects another; The overall behavior of the system is modified by adjusting the values of these connections through the repeated application of a learning rule.

The back-propagation learning algorithm is an error-correcting learning procedure for multilayered network architectures. Information flows forward through the network from the input layer, through the intermediate, or hidden layer(s), to the output layer. The value of a unit is determined by first computing the weighted sum of the values of the units connected to this unit and then applying the logistic function, $1/1+e^{-z}$ to the result. This forward propagation rule is recursively applied to successively determine the unit values for each layer.

The goal of the learning procedure is to minimize the average squared error between the values of the output units and the correct pattern provided by a teacher. This is accomplished by first computing the error gradient for each unit on the output layer, which is proportional to the difference between the target value and the current output value. The error gradient is then recursively determined for layers from the output layer to the input, by computing the weighted sum of the errors at the previous layer. These error gradients, or deltas, are then used to update the weights[2].

Computationally the forward and backward propagation steps are very similar. Forward propagation consists of four basic steps: distributing the activation values of the units to their respective fan-out weights, multiplying the activations by the weight values, summing these values from the weights into the next layer of units, and applying the logistic function to this value. The backward propagation of error consists of four similar steps: distributing the error values of the units to their respective fan-in weights, multiplying the error by the weight values, summing these values from the weights into the previous layer of units, and evaluating the derivative of the logistic function. In addition to forward and

[2]The original source should be consulted for the details of back-propagation.

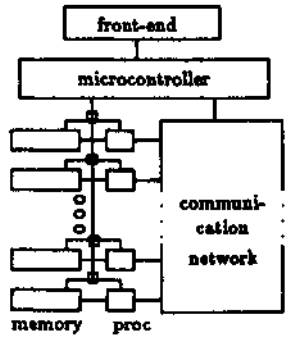

Figure 1: The Connection Machine.

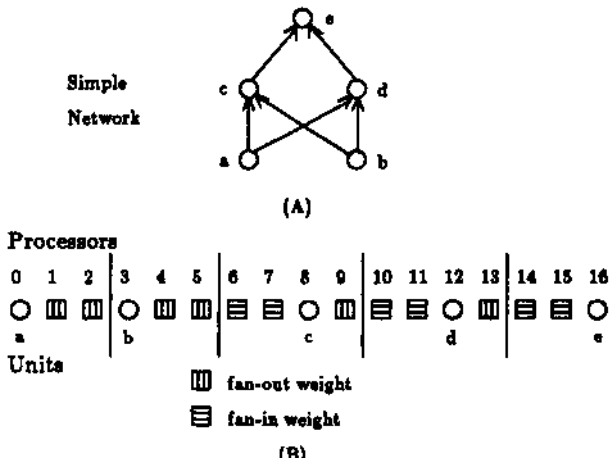

Figure 2: The Layout of Weights and Units of a Simple Network on the Connection Machine. (A) A simple two layer network. (B) The layout of the network on the processors of the Connection Machine.

backward propagation, the inputs and outputs must be clamped to the appropriate values. In the next section, we will show how each of these steps is executed on the Connection Machine.

3 The Connection Machine

The Connection Machine is a highly parallel computer with between 16,384 and 65,536 processors. Each processor has two single-bit arithmetic logic units (ALUs), and some local memory - currently 64K bits. In addition, every 32 processors shares a floating point unit. All the processors are controlled by a single instruction stream (SIMD) broadcast from a microcontroller. Figure 1 shows a block diagram of the Connection Machine. Processors can communicate using a few different techniques - the only two of concern in this paper are the *router* and the scan operations. The *router* operations allow any processor to write into the memory or read from the memory of any other processor. The *Sean* operations allow a quick[8] summation of many values from different processors into a single processor, or the copying of a single value to many processors [3].

In the implementation of back-propagation, we allocate one

[8]Usually fatter than a router cycle.

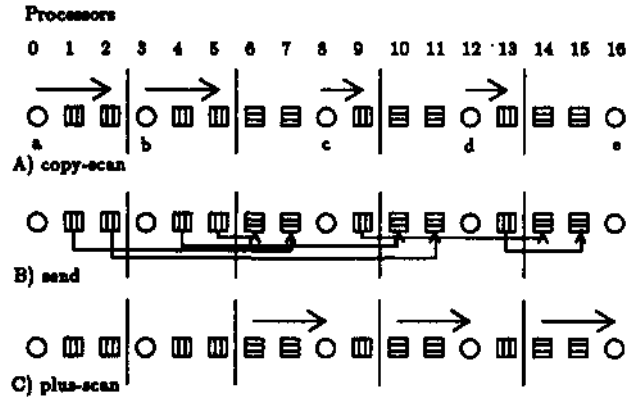

Figure 3: Forward Propagation.

processor for each unit and two processors for each weight[4]. The processor for each unit is immediately followed by all the processors for it's outgoing, or fan-out, weights, and immediately preceded by all of it's incoming, or fan-in, weights (see Figure 2). The beginning and ending of these contiguous segments of processors are marked by flags. A *$can* operation called segmented *copy-scan* is used to quickly copy a value from the beginning or end of a segment of contiguous processors to all the other processors in the segment and an operation called segmented *plus-scan* is used to quickly sum all the values in a segment of processors and leave the result in either the first or last processor of the segment. Thus our layout enables us to use the scan operations to distribute the unit values from the units to their output links and to sum the values from their input links.

The forward propagation step proceeds as follows. First, the activations of the input units are clamped according to the input text. The algorithm then distributes the activations to the weights using a *copy-scan* operation (step A in Figure 3), and then all weights multiply their weight by these activations. The result is sent from the output weights to the input weights of the unit in the next layer (step B) using the router. A *plus-scan* then sums these input values into the next layer of units (step C), and the logistic function is applied to the result at all the unit processors to determine the unit activations. This forward propagation step must be applied once for each layer of links in the network.

Once forward-propagation is completed, we determined the error at the output layer, and propagate this error backward. This error back-propagation consists of running *copy-scan* backwards, copying the deltas from the output units into their input weights, then sending the deltas from the input weights to the output weights of the units at the previous layer. The deltas can then be multiplied by the weight values, and summed in the reverse direction into the units at the previous layer. The derivative of the logistic function is then evaluated to determine the error of units at the previous layer. As in forward propagation, this step must be applied once for each layer of links.

The algorithm as described so far uses the processors inefficiently for two reasons. Firstly, we use two processors for each weight when only one of them is busy at a time, and secondly, when we are running one layer, the processors for all the other layers are idle. To overcome the first inefficiency and use one pro-

[4]Later in this paper we will show how the processors can be shared between units and weights, requiring only one processor per weight.

cessor per weight, we overlap the input and output weights. We also overlap the units with the weights. To overcome the second problem and keep the processors for each layer busy we pipeline the layers as follows. Given a set of n input vectors V_i ($0 < i < n$), and m layers l_j, pipelining consists of propagating the i^{th} input vector across the first layer, while propagating the previous input vector (v_{i-1}) across the second layer, v_{i_3} across the third layer, ... , and $Vi\text{-}_m$ across the last layer. We also interleave the back-propagation with the forward-propagation so that immediately after presenting v_i, to the input, we start back-propagation v_{i_m} backward from the output. The depth of the whole pipe for m layers is 2m.

This implementation has some important advantages over other possible implementations. Firstly, with pipelining, the implementation unwraps all the potential concurrency.[5] Since it is possible to simulate a concurrent computer with a serial computer but the opposite is not true, our method can be used efficiently on any machine ranging from a completely serial computer to a very fine grained parallel computer. If we did not expose all the concurrency, we would not utilize as many processors in a very fine grained computer. Secondly, the implementation works well with sparse connectivity. Methods based on dense matrix multiplies, such as some of the serial implementations, although faster for dense connectivity, are extremely inefficient with sparse connectivity. Thirdly, in our implementation, all processors are kept active even if different units have different fan-ins. This would not be true if we used one processor per unit and serially looped over the fan-ins of each unit — as one might be tempted to do in a more coarse grained parallel computer. Lastly, the time taken by each step is independent of the largest fan-in.

Networks with more links than physical processors can be simulated in the Connection Machine using an abstraction called the *virtual processor* (VP) [7]. A *virtual processor* is a slice of memory within a physical processor. Many such VPs can exist within each physical processor. Looping over the VPs in general causes a linear slow-down.

Similar layouts of static networks have been used to implement a rule based system [2], a SPICE circuit simulator and a maximum-flow algorithm.

4 CM-NETtalk

NETtalk is a connectionist network that uses back-propagation to learn the pronunciations of English words. We have implemented NETtalk on the Connection Machine, and present the results of our implementation here[6].

NETtalk is is composed of three layers of units, an input layer, an output layer, and an intermediate or hidden layer. Each unit in each layer is connected to each unit in the layer immediately above and/or below it. The representations at the input and output layers are fixed to be representations of letters and phonemes respectively. The representation at the hidden layer, on the other hand, is constructed automatically by back-propagation.

NETtalk uses a fixed-size input window of seven letters to allow the textual context of three letters on either side of the current letter to be taken account in the determination of that letter's pronunciation (see Figure 4). This window is progressively stepped through the text. At each step, the output of the network generates its guess for the pronunciation of the middle,

[5]This is not strictly true sinca we could get another factor of 3 by running the forward and backward propagation concurrently.

[6]interested readers should consult the original sources for details.

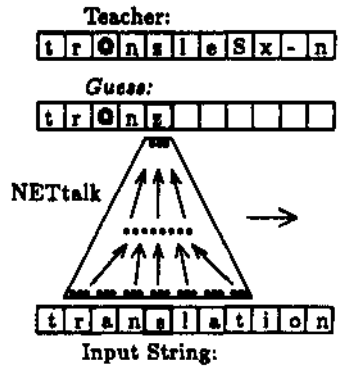

Figure 4: The Seven Letter Window Used by NETtalk.

or fourth, letter of the sequence of letters currently within the input window. This guess is compared to the correct pronunciation, and the values of the weights are iteratively adjusted using back-propagation to minimize this difference. Good pronunciations (95% of the phonemes correct) of a thousand-word corpus of high-frequency words are typically achieved after ten passes through the corpus.

We have experimented with a network consisting of 203 input units (7 letters with 29 units each), 60 hidden units and 26 output units. This required a total of 13826 links (processors) - 12180 in the first layer, 1560 in the second layer and 76 to the true units[7]. The learning rate was approximately the same as that achieved by a C implementation on various serial machines.

In the current implementation, using a 16,384 processor machine, the time required for each letter during the learning stage was 5 milliseconds. This includes the forward propagation, the backward propagation, the time necessary to clamp the input and output, and the time required to calculate the error. The time is broken up by the type of operation as follows:

- Scanning 30% - This includes two segmented *plus-scans* and two segmented *copy-scans.*
- Routing 40% - This includes two routing cycles.
- Arithmetic 20% - This includes seven multiplies and several additions, subtractions and comparisons.
- Clamping 5% - This is the time needed to get the characters to the input units and the expected phonemes to the output units.
- Other 5% - Mostly for moving values around.

With improvements in the coding of the implementation and in microcode, we expect that this time could be improved by a factor of three or more.

Table 5 shows comparative running times of the error back-propagation algorithm for several machines. On an existing implementation of back-propagation on a VAX 780, the same network required 650 microseconds per letter. This represents a 130 to 1 improvement in speed. On a 64K machine and larger networks, we could get a 500 to 1 improvement. This is about twice the speed of an implementation of the algorithm on a Cray-2, yet a Connection Machine costs a quarter of the price. Fanty [6] has

[7]True units are units that are always kept iln the active state. Their function is to allow the thresholds of the other units in the network to be modified in a simple way.

	MLPS	Relative Times
MicroVax	.008	1
Sun 3/75	.01	1.3
Vax 780	.027	3.4
Sun 160 with FPA	.034	4.2
Dec 8600	.06	7.5
Convex	.80	98
Cray-2	7	860
65,536 Processor CM	13	1600

Figure 5: Comparison of Running Times for Various Machines. MLPS stands for Millions of Links Per Second. Some of these times are from [10,9,13].

implemented a connectionist network using the BBN Butterfly, a coarse grained parallel computer with up to 128 processors, but because the type of networks he used were considerably different, we cannot compare the performances.

Using virtual processors, on the Connection Machine it is possible to simulate up to 16 million links in physical memory. With software currently being developed to use the Connection Machine's disks, the CM will be able to simulate many more than this.

5 Conclusions

We have discussed the first implementation of a connectionist learning network on a fine-grained parallel machine. Our experiments indicate that the Connection Machine can currently simulate rather large networks of over 65 thousand connections at speed over twice as fast as the most powerful serial machines such as the Cray-2. The method outlined here should generalize to any highly concurrent computer with a routing network and, with small modifications, can be used with many variations of connectionist network. Unlike neuromorphic, hardware-based systems, our method places no restrictions on the computations performed at the links or the units, nor on the topology of the network.

In our implementation we were able to keep all of the processors busy most of the time using a single instruction stream; multiple instruction streams do not seem to be necessary. Rather, communication was the bottleneck - at least on the current Connection Machine. Effort needs to be spent designing faster routing networks.

The lack of computational power was a major factor in the dissolution of the first wave of connectionism in the 1950's and 60's. Alternative, symbolic techniques were more successful in part because they better fit the computational resource available at the time. With the advent of fine-grained parallel computers, this situation is beginning to change; the exploration of large-scale connectionist networks is starting to become computationally feasible.

References

[1] Joshua Alspector and Robert B. Allen. *A Neuromorphic VLSI Learning System.* Technical Report, Bell Communications Research, 1987.

[2] Guy E. Blelloch. *AFL-1: A Programming Language for Massively Concurrent Computers.* Technical Report 918, Massachusetts Institute of Technology, November 1986.

[3] Guy E. Blelloch. The scan model of parallel computation. Proceedings Int. Conf. on Parallel Processing, August 1987.

[4] J. L. Ellman and D. Zipser. *Learning the Hidden Structure of Speech.* Technical Report ICS Report 8701, University of California at San Diego, Institute for Cognitive Science, 1987.

[5] H. P. Graf et al. VLSI implementation of a neural network memory with several hundreds of neurons. In *Proceedings of the Neural Networks for Computing Conference,* Snowbird, UT, 1986.

[6] M. Fanty. *A Connectionist Simulator for the BBN Butterfly Multiprocessor.* Technical Report Butterfly Project Report 2, University of Rochester, Comp. Sci. Dept., January 1986.

[7] William D. Hillis. *The Connection Machine.* MIT Press, Cambridge, Mass., 1985.

[8] G. E. Hinton. Learning distributed representations of concepts. In *Proceedings of the Cognitive Science Society,* pages 1-12, Erlbaum, 1986.

[9] K. Kukich. Private Communication, 1986. Bell Communications Research Corporation.

[10] James L. McClelland and Kevin Lang. Personal Communication, 1987. Carnegie-Mellon University.

[11] David E. Rumelhart, Geoffrey E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Vol. 1: Foundations,* pages 318-362, MIT Press, Cambridge, Mass., 1986.

[12] E. Saund. Abstraction and representation of continuous variables in connectionist networks. In *Proceedings of the Fifth National Conference on Artificial Intelligence,* pages 638-644, Morgan Kauffmann, 1986.

[13] Terrence J. Sejnowski. Personal Communication, 1987. John Hopkins University.

[14] Terrence J. Sejnowski and Charles R. Rosenberg. Parallel networks that learn to pronounce English text. *Complex Systems,* 1:145-168, 1987.

[15] M. Silviotti, M. Emerling, and C. Mead. A novel associative memory implemented using collective computation. In *Proceedings of the 1985 Chapel Hill Conference on Very Large Scale Integration,* page 329, 1985.

[16] G. Tesauro and T. J. Sejnowski. A parallel network that learns to play backgammon. 1987. in preparation.

[17] R. L. Watrous, L. Shastri, and A. H. Waibel. *Learned phonetic discrimination using connectionist networks.* Technical Report, University of Pennsylvania Department of Electrical Engineering and Computer Science, 1986.

[18] D. Zipser. *Programming networks to compute spatial functions.* Technical Report, University of California at San Diego, Institute for Cognitive Science, 1986.