Parallel and Distributed Systems for Constructive Neural Network Learning*

J. Fletcher Z. Obradović†

School of Electrical Engineering and Computer Science
Washington State University
Pullman WA 99164-2752

Abstract

A constructive learning algorithm dynamically creates a problem-specific neural network architecture rather than learning on a pre-specified architecture. We propose a parallel version of our recently presented constructive neural network learning algorithm. Parallelization provides a computational speedup by a factor of $O(t)$ where t is the number of training examples. Distributed and parallel implementations under p4 *using a network of workstations and a Touchstone DELTA are examined. Experimental results indicate that algorithm parallelization may result not only in improved computational time, but also in better prediction quality.*

1 Introduction

A *neural network* is a weighted graph of simple processing units (or *neurons*). The interconnection graph of a *feed-forward* network is acyclic with processing units arranged in multiple layers consisting of input, zero or more hidden, and output layers. All units in any layer are fully connected to the succeeding layer. Units compute an *activation function* of their weighted input sum. Here we consider *binary neural networks* where the activation function of each unit is of the form $g(x) : \mathrm{R} \rightarrow \{0, 1\}$,

$$g(x) = \begin{cases} 0 & \text{if } x < t \\ 1 & \text{if } x \geq t. \end{cases}$$

Traditional neural networks learning (e.g. back-propagation [12]) involves modification of the interconnection weights between neurons on a pre-specified network. Determining the network architecture is a challenging problem which currently requires an expensive trial-and-error process. In selecting an appropriate neural network topology for a classification problem, there are two opposing objectives. The network must be large enough to be able to adequately define the separating surface and should be small enough to generalize well [7]. Rather than learning on a pre-specified network topology, a *constructive algorithm* also learns the topology in a manner specific to the problem. The advantage of such constructive learning is that it automatically fits network size to the data without overspecializing which often yields better generalization. Examples include the tiling algorithm of Mézard and Nadal [9] and the cascade-correlation algorithm of Fahlman and Lebiere [4]. Our goal is to explore the use of distributed and parallel systems in constructively learning a single hidden layer binary neural network architecture. We argue that a parallel approach improves computational efficiency and generalization quality.

In a single hidden layer feed-forward binary neural network, each hidden unit with fan-in k is a representation of a k-1 dimensional hyperplane. The hyperplane corresponding to the hidden unit may be determined through solution of the equation system defined by k points on the hyperplane. Our work is inspired by a constructive algorithm proposed by Baum [1] where a sequence of oracle queries are used in conjunction with training examples to find these k points. Here the learner is allowed to ask an oracle for the correct class associated with arbitrary points in the problem domain in addition to using the training examples provided. The hyperplanes are sequentially determined by partitioning the problem domain space using training examples and queries. The hidden units of a single hidden layer feed-forward binary neural network and corresponding connections are then created from the hyperplanes. The connection weights from the hidden layer to the output layer are determined by an algorithm which separates the hidden layer represen-

*Research sponsored in part by the NSF Industry / University Cooperative Center for the Design of Analog-Digital ASICs (CDADIC) under grant NSF-CDADIC-90-1 and by Washington State University Research Grant 10C-3970-9966.

†Also affiliated with the Mathematical Institute, Belgrade, Yugoslavia.

0-8186-3900-8/93 $3.00 © 1993 IEEE

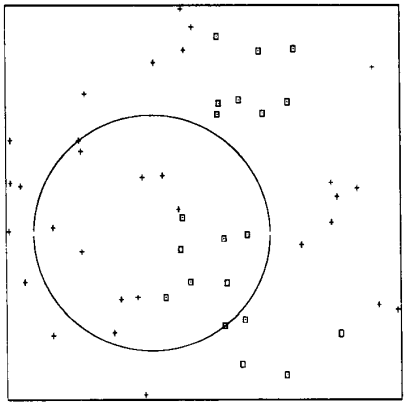

Figure 1: First unknown region

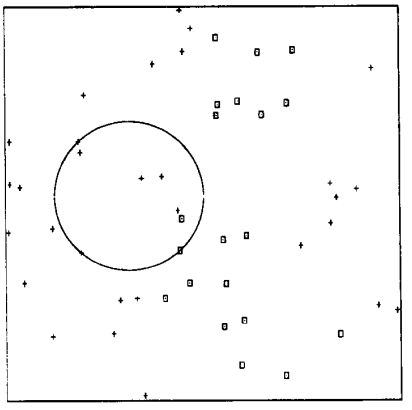

Figure 2: Next unknown region

tation of the problem by a single hyperplane (e.g. the perceptron algorithm [11]).

In Section 2 we describe our constructive learning algorithm which does not require oracle queries. In Section 3 a new parallel approach to this algorithm is explored with analysis and experimental results following in Section 4.

2 Sequential hidden layer construction

While Baum's algorithm is applicable where an oracle for the classification of any given point exists, in many cases such an oracle is not available or may be too expensive for practical use. In [5] we proposed a modification of Baum's algorithm which does not assume the availability of such an oracle and incrementally constructs the neural network from examples alone. In this modification, approximations of the points on the hyperplane are found by repeatedly interpolating between example points of the various classes T_1 and T_2 in the training set T. The interpolation begins by selecting positive and negative examples $m \in T_1$, $n \in T_2$. The unknown region between m and n is then searched for the nearest point $q \in T$ to the midpoint of m and n. The unknown region is defined as the the circle centered at the midpoint of m and n with a diameter of the distance between m and n, as shown in Figure 1. If q is found, the search is then repeated in the smaller unknown region between q and m or q and n respectively depending on whether q is positive or negative (Figure 2).

If no point from T is found in the current unknown region, its midpoint p^1 is the closest approximation to a point on the separating hyperplane. If p^1 is determined to be within a specified tolerance of an existing hyperplane, a new pair of points is selected and the search is repeated. The remaining points p^2 through p^k that define the hyperplane are found by taking a random vector from p^1 to a point $v \in T$ (Figure 3) and interpolating between either m and v or n and v to p^i based on the class of v. The interpolated points from T and the generated hyperplane are shown in Figure 4.

As in Baum's algorithm, the connection weights from the hidden layer to the output layer units must be computed once the hidden unit layer has been generated. In the modified algorithm, the hidden layer units are generated from examples alone, and so may not correspond to the optimal separating hyperplanes. As such, the hidden layer problem representation of the generated network with the same number of hidden units as in the minimal network may not be linearly separable. In order to account for this possibility, hidden units continue to be generated beyond the minimal architecture; for example, until the data is exhausted, a number of data points have been examined without generating a new hidden unit or a predetermined number of units have been created.

The pocket algorithm [6] is a single-layer neural network learning algorithm that finds the optimal separation under a given topology for problems that are not linearly separable. The algorithm keeps the best set of weights in the "pocket" while the perceptron is trained incrementally. A practical modification of the pocket learning algorithm is proposed in [10] which is faster and still has the same guarantee for convergence to the optimal separating hyperplane. This parallel dynamic algorithm is used to determine the output layer weights in the constructed network.

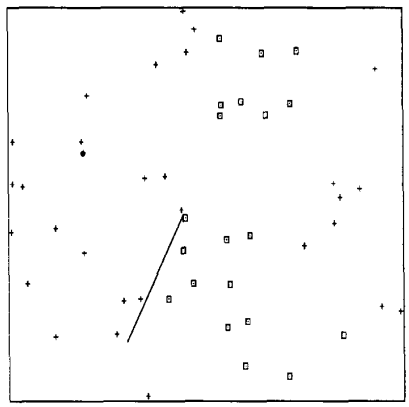

Figure 3: Random vector

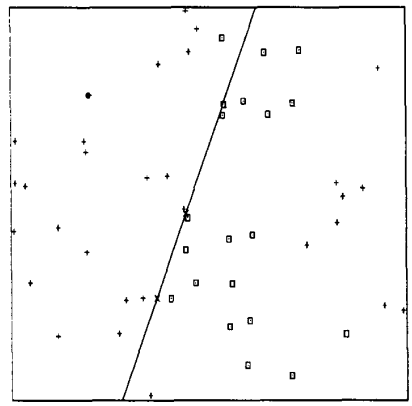

Figure 4: Separating hyperplane

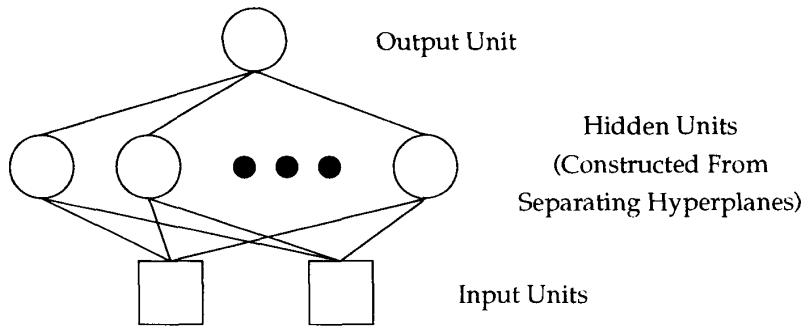

Figure 5: Hidden layer construction

3 Parallel hidden layer construction

While the sequential algorithm provides good generalization, significant computational resources are required. Here we propose a speedup by a parallelization that distributes the computational load across a number of processors. In order for parallelization to be efficient, an appropriate partitioning of the input space is required. This is accomplished by assigning the example points of one class evenly across the available processors. Given training set T of t examples belonging to classes T_1 and T_2 let t_1 and t_2 $(t_1 \geq t_2)$ be the number of examples in each class respectively. In a system with $P+1$ processors each of P slave processors $(1 \leq P \leq t_1)$ is assigned $\lceil t_1/P \rceil$ examples of class T_1. Each slave processor examines the input subspace formed by pairing its assigned examples of T_1 with all examples in T_2. A system with a balanced computational load is obtained by this partitioning of the initial pairs.

Figure 6 shows the proposed parallel architecture. Each processor may be either a workstation in a distributed environment or a processor on a parallel machine. One processor is responsible for the master process. This master process distributes the training data at initialization and creates neural network hidden layer units from the determined separating hyperplanes. All slave processors search for separating hyperplanes as described in Section 2 starting from the initial pairs in their assigned data partitions. When such a separating hyperplane is found, it is communicated to the master process. The master process then compares the hyperplane to those that currently exist. If it is not sufficiently similar to an existing hyperplane, a new hidden unit corresponding to the hyperplane is generated (Figure 5).

Hidden layer construction is completed when a predetermined number of hidden units have been generated, the input space has been exhaustively searched, or a number of initial pairs have been examined with-

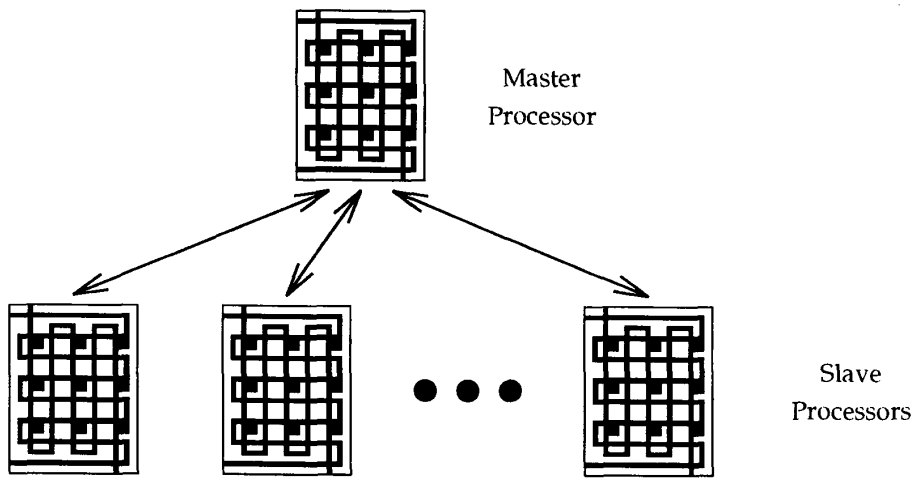

Figure 6: Parallel architecture

out determining a new separating hyperplane. Finally, the master process performs the relatively simple task of training the output layer weights as in the algorithm of Section 2.

4 Analysis and Experimental Results

The total running time of our algorithm depends primarily upon the time required to determine if a separating hyperplane can be constructed starting from a given pair of training examples. Search for a point on the hyperplane takes $O(\log t)$ interpolation steps since each interpolation removes at least half of the t training examples. In each interpolation step, finding the nearest training example to the center of an unknown region can be determined in time $O(\log t)$ through use of the *k-d* tree of Bentley [2]. Thus, the worst case time required to search for one point on the hyperplane is $O(\log^2 t)$. A hyperplane is defined by k points, and so the total time to determine if a hyperplane can be found starting from a given pair of training examples is $O(k \log^2 t)$.

In the sequential algorithm an exhaustive data partitioning starting from all $t_1 t_2$ training pairs of examples can be performed in worst case time of $O(k t_1 t_2 \log^2 t)$. In the parallel algorithm an initial overhead of $O(t)$ is required for data distribution. A minimal overhead of $O(k)$ is incurred for transfer of generated hyperplane data from the slave to the master processor. Since $t_1 + t_2 = t$ the worst case parallel time for an exhaustive data partitioning is thus $O((k t_1 t_2 \log^2 t)/P)$ where P is the number of slave processors.

In learning problems w.l.o.g. we can assume that both t_1 and t_2 are of order $\Theta(t)$ as both classes have to be well represented in the training set. With that assumption the worst case parallel computing time of the maximal distributed system ($P = \max(t_1, t_2)$) is $O(kt \log^2 t)$ compared to a sequential time of $(kt^2 \log^2 t)$. Algorithm parallelization thus provides a computational speedup by a factor of $O(t)$.

The algorithm was implemented using *p4* [3]. Developed at Argonne National Laboratory, *p4* supports parallel programming for both distributed environments and highly parallel computers. Two implementation platforms were used: a distributed system of 19 DECStations and a Touchstone DELTA. The Touchstone [8] is an Intel high-speed concurrent multicomputer, consisting of 576 nodes in a 19×36 mesh. Of these, 64 nodes were allocated for our experiments. Implementation under *p4* allowed the same code to be used for the Touchstone as for the DECStation network.

Experiments were performed using the MONK's problems [13] to compare the quality of generalization between the sequential and parallel implementations. The MONK's problems consist of three six-feature binary classification problems which represent specific challenges for standard machine-learning algorithms, such as the ability to learn data in disjunctive normal form, parity problems and performance in the presence of noise. To allow the random vector to search equally in each dimension, the input data is normalized to points on a hypersphere.

	Sequential		Distributed		Parallel	
Processors	1		19		64	
Accuracy	Train	Test	Train	Test	Train	Test
Problem 1	100.00	85.42	100.00	81.02	100.00	81.71
Problem 2	81.07	70.37	85.80	72.45	91.72	75.23
Problem 3	96.72	72.92	99.18	77.08	100.00	78.94

Table 1: Percentage Accuracy on the MONK's Problems

The generalization ability of the sequential and parallel implementations is compared in Table 1. It is interesting to note that in the more complex problems 2 and 3 the generalization of the parallel algorithm exceeds that of the sequential algorithm. This improvement may be due to the fact that as the number of processors increases, a greater diversity in the input space will be searched.

While these results are promising, the principles described here are being further evaluated on the large-scale problem of predicting protein structure.

5 Conclusions

Neural networks efficiency and prediction quality depends significantly on how we select network architecture, learning algorithm and initial set of weights. The constructive learning algorithm of Section 2 efficiently learns not just connection weights but also creates the required architecture. A parallel version proposed in Section 3 provides a significant speed-up in the construction of the hidden layer and a greater diversity in the input space searched, also resulting in improved generalization quality.

References

[1] E. B. Baum. Neural net algorithms that learn in polynomial time from examples and queries. *IEEE Transactions on Neural Networks*, 2(1):5–19, January 1991.

[2] J. L. Bentley. Multidimensional binary search tree used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[3] R. Butler and E. Lusk. *User's Guide to the p4 Parallel Programming System*, November 1992.

[4] S. Fahlman and C. Lebiere. The cascade-correlation learning architecture. In D. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2, pages 524–532, Denver 1989, 1990. Morgan Kaufmann, San Mateo.

[5] J. Fletcher and Z. Obradović. Creation of neural networks by hyperplane generation from examples alone. In *Neural Networks for Learning, Recognition and Control*, page 23, Boston, 1992.

[6] S. I. Gallant. Perceptron-based learning algorithms. *IEEE Transactions on Neural Networks*, 1(2):179–191, June 1990.

[7] S. Geman, E. Bienstock, and R. Doursat. Neural networks and the bias / variance dilemma. *Neural Computation*, 4(1):1–58, 1992.

[8] Intel Supercomputer Systems Division, Beaverton, OR. *Touchstone Delta System User's Guide*, October 1991.

[9] M. Mézard and J.-P. Nadal. Learning in feedforward layered networks: The tiling algorithm. *Journal of Physics A*, 22:2191–2204, 1989.

[10] Z. Obradović and R. Srikumar. Dynamic evaluation of a backup hypothesis. In *Neural Networks for Learning, Recognition and Control*, page 71, Boston, 1992.

[11] F. Rosenblatt. *Principles of Neurodynamics*. Spartan, New York, 1962.

[12] D. Rumelhart, G. Hinton, and R. Williams. Learning internal representations by error propagation. In D. Rumelhart and J. McClelland, editors, *Parallel Distributed Processing*, volume 1, chapter 8, pages 318–362. MIT Press, Cambridge, 1986.

[13] S. B. Thrun et al. The MONK's problems: A performance comparison of different learning algorithms. Technical Report CMU–CS–91–197, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1991.