A DISTRIBUTED SIMULATOR FOR NEURAL NETWORKS TRAINING

Sorin Babii* and Vladimir Creţu**

** Department of Computers, "Politehnica" University of Timişoara,*
Faculty of Automation and Computers, V. Pârvan 2, Timişoara, Romania
Phone: (40) 256-404059, Fax: (40) 256-403214,
E-Mail: sorin@cs.utt.ro, WWW: http://www.cs.utt.ro/~sorin
*** Department of Computers, "Politehnica" University of Timişoara,*
Faculty of Automation and Computers, V. Pârvan 2, Timişoara, Romania
Phone: (40) 256-403255, Fax: (40) 256-403214,
E-Mail: vcretu@cs.utt.ro, WWW: http://www.cs.utt.ro/~vcretu

Abstract: This paper presents a distributed simulator for neural networks – NetPar, and the results of several experiments in distributing the training phase of an artificial neural network. We developed and analyzed a distribution strategy for the back-propagation algorithm.

We describe a distributing procedure of the well-known algorithm of back-propagation, and implemented this algorithm on several networks of computers, which allowed us to evaluate and analyze the performances using the results of actual experiments. We were interested in the qualitative aspects, trying to understand the factors which determine the behavior of this distributed algorithm. We tried to emphasize some specific aspects to be considered when implementing such a parallel algorithm on a set of workstations, interconnected in a local area network. Also, we investigated the possibilities to exploit the computational resources of such a set of workstations.

Keywords: neural networks, back-propagation algorithm, parallel algorithms, distributed algorithms, computers network.

1. INTRODUCTION

The training phase for a neural network needs a considerable computational effort. There are millions of floating point multiplications, even for small size networks and applications. Moreover, the neural networks need a large amount of memory. These are the reasons for which working with neural networks is time-consuming activity, drastically limiting the size of the applications.

There are some ways to compensate these disadvantages. The first approach consists in reducing the size of the problem by pre-processing the input data, obtaining a reduced number of iterations needed for training, or even a smaller neural network. These changes are almost always problem-specific and this approach cannot be generalized for all kind of problems.

Another possibility is to enhance the performance of the back-propagation algorithm (see Rumelhart *et al*., 1986), either by modifying it according to a specific problem, either using results from numerical optimization theory, like applying the *conjugate gradient* method (see Johansson *et al*., 1991).

A third approach consists in accelerating the actual algorithms by hardware implementing them (using VLSI techniques or optical), or changing them to operate on a parallel architecture.

We used this last approach: to distribute the back-propagation training algorithm over a set of

computers in a local area network. Unlike other implementations, (see Pomerleau *et al.*, 1988; Pétrowski *et al.*, 1989; Singer 1990; Cosnard *et al.*, 1991; Pétrowski *et al.*, 1993; Tørresen, 1996; Tørresen and Tomita, 1998) which distribute the neural network in dedicated architectures with multiple processors, our algorithm was implemented in a simulator – NetPar, which distributes the neural network over a set of workstations. We will show that this is a promising approach, when considering the reduced cost of the equipment.

2. NETWORK PARTITIONING

When converting a sequential algorithm into a parallel one, we have to consider at least two strategies. Talking about neural networks, these two strategies are known as data partitioning, and network partitioning.

If the same algorithm is applied several times on different data sets, we can execute these applications concurrently, on different processors, with completely independent tasks. This strategy is known as *job parallelism* or *data partitioning*, or, sometimes — for obvious reasons — *training parallelism*.

Another approach is using the so-called geometric or spatial parallelism. In this approach the execution of the algorithm is parallelized over a single data set, by distributing the data over the processors so that all necessary data for a processor is stored locally or they are easily accessed from neighbor processors.

This strategy is also known (see Pomerleau *et al.*, 1988) as *network partitioning*, as the processing of a pattern can be parallelized by dividing the neural network, every processor having to deal with a small piece of the network.

2.1 Dividing the network.

If we want to distribute the units and weights over a set of processors, it cannot be done in too many ways. One possibility is to divide the network horizontally, in three parts, so that a processor will contain the units from a given layer. This is not a very good idea, since usually the network does not have too many layers, so we cannot use too many processors. Moreover, the forward and the backward steps are in fact sequential and only the units in a single layer evolving in parallel, making this parallelization scheme impossible.

The network can be divided vertically, so that the units in a layer are evenly distributed among processors. In this way, each processor will hold units from all the layers. There are two methods to distribute the weights:

1) All weights for the connections incoming into a unit are stored into the processor which simulates that unit.

2) All weights for the connections leaving a unit are stored into the processor which simulates that unit.

In fact, there is no difference between these two strategies, since in the first case the forward step is simple to parallelize while the backward step will be a little difficult. If we choose the second strategy, it will be the other way round.

We selected the first strategy. All the units are evenly distributed among processors, so that all processors store the same number of input, hidden and output units, although it is not necessary that the number of processors to divide the number of units in a layer. In this case, some processors will simulate one extra unit than others. To simplify, we will consider that the number of units in any layer is divided by the number of processors.

The weights in the network are distributed as in the first strategy: the processor which simulates a hidden or output unit will store the weights for the connections incoming to that unit, like in Fig. 1 – we represented the weights in the processor number 1.

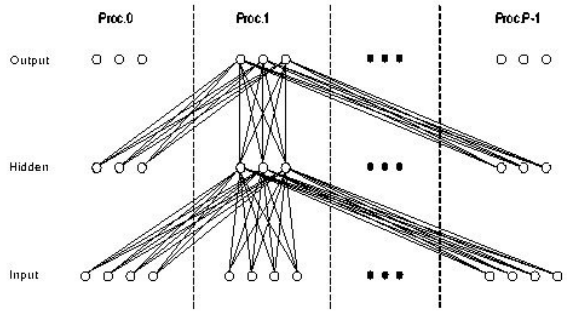

Fig. 1. Distribution of units and weights.

The processors are in a ring topology, which is easy to extend to include more elements, and can contain any number of processors. Maybe other topologies would be of interest, but this is another subject.

Even if each processors simulates only some input and hidden units, all the processors have to store the activation of all input and hidden units, since this is needed both in the forward step and the backward step of the back-propagation algorithm. By storing the activation value of all units we avoid some supplemental communications.

All the processors execute the same program, even there is an administrator and some slaves, but the administrator role differs only when there are input/output operations.

2.2 The Algorithm

The parallelization consists of seven steps, two steps for the forward propagation and five steps for the backward propagation.

The **first step** computes the activation of the hidden units. In fact there are two sub-steps, one for the distribution of input units activations, based on equation (1), and the second for the actual evaluation of the hidden units activations, as in equation (2). Since the values are needed again in step six, when the weight changes are computed, the entire vector of inputs unit activations is stored for later use, after it was received from the other processors.

$$net_j^H = \overline{a^I} * \overline{w_{i\to j}^H} = \sum_{i=0}^{NI-1} a_i^I w_{i\to j}^H, \forall\, j \in H_{[p]} \quad (1)$$

$$a_j^H = f(net_j^H), \forall\, j \in H_{[p]} \quad (2)$$

The **step two** computes the output units activations and is very similar to the first step. Again, we need to store the values of all hidden units activations.

$$net_k^O = \sum_{j=0}^{NH-1} a_j^H w_{j\to k}^O = \sum_{q=0}^{P-1} \overline{a_{[q]}^H} * \overline{w_{[q]\to k}^O}, \forall\, k \in O_{[p]} \quad (3)$$

The **third step** evaluates the output unit delta values, which can be done exclusively local.

$$\delta_k^O = (t_k - a_k^O) a_k^O (1 - a_k^O), \forall\, k \in O_{[p]} \quad (4)$$

In the **fourth step** we calculate the hidden units delta value. This step is very similar to step two, as it consists of a vectorial product. Still, we need weights stored on other processors. Since we used the first approach in distributing the weights, for this backward computations we had to implement a special communication procedure in order to collect all the values required for these calculi.

$$\delta_j^H = a_j^H (1 - a_j^H)(\overline{\delta^O} * \overline{w_{j\to}^O}) = a_j^H (1 - a_j^H) \sum_{k=0}^{NO-1} \delta_k^O w_{j\to k}^O, \forall\, j \in H_{[p]} \quad (5)$$

We rewrote equation (5) to emphasize the sums over processors:

$$\delta_j^H = a_j^H (1 - a_j^H) \sum_{k=0}^{NO-1} \delta_k^O w_{j\to k}^O = a_j^H (1 - a_j^H) \sum_{q=0}^{P-1} \overline{\delta_{[q]}^O} * w_{j\to[q]}^O, \forall\, j \in H_{[p]} \quad (6)$$

The last sum is over all processors. Each term can be computed only by the processor which stores the specified weights and delta values. As a result, if $j \in H_{[a]}$ is the index of a hidden unit simulated by processor *a*, processor *b* can compute the terms $\overline{\delta_{[b]}^O} * \overline{w_{j\to[b]}^O}$ needed by processor *a* when computing the values $\delta_{[a]}^H$. We developped a communication scheme which needs *P*-1 sequential steps, with *P* transmissions at each step.

The **fifth step** computes the weight changes between the hidden and the output layers; all the hidden units activations were stored in step two, so the processors can calculate all weight changes with local data exlusively.

$$\Delta w_{j\to k}^O(n+1) = \eta \delta_k^O a_j^H + \alpha \Delta w_{j\to k}^O(n) \quad (7)$$

The **step six** is similar to step five, except we compute the weight changes for the weights between the input and the hidden layers. Again, since all the input activations were stored locally in step one, this step can be done exclusively local.

$$\Delta w_{i\to j}^H(n+1) = \eta \delta_j^H a_i^I + \alpha \Delta w_{i\to j}^H(n) \quad (8)$$

Step seven is the actual changing of all the weights, and can be performed only with local data, without communication.

3. EXPERIMENTAL RESULTS

We run the parallelization on nets of different sizes with varying numbers of processors, in order to understand what factors determine the behavior of the algorithm. In order to measure the performance compared to the sequential verion, the nets are simply nets of convenient sizes, not nets for real applications, and no actual learning takes place.

We defined the *speedup* as the ratio of the execution time T_{seq} needed by the sequential algorithm and the execution time T_{par} for the parallel algorithm using *P* when both algorithms are applied to te same problem:

$$S(P) =_{\text{def}} \frac{T_{seq}}{T_{par}(P)} \quad (9)$$

We also define the efficiency as the ratio of the speedup obtained and the optimal one. If we assume that the maximal *S(P)* that can be obtained is *P*, we will use for efficiency the formula:

$$E(P) =_{\text{def}} \frac{S(P)}{P} = \frac{T_{seq}}{T_{par} \cdot P} \quad (10)$$

Obviously, the values for *E(P)* are between 0 and 1.

We tried to construct the algorithm so that a minimum amount of communication is performed.

One advantage of this algorithm is the low memory requirements. We distributed the weights, activations and delta values of the net uniformly among the processors, but the activation values of all input and hidden units are stored on all processors, because these values are used twice. The memory

requirement of each processor (number of double64) is:

$$NI + NH + PH + 2PO + 2\left((NI+1)\cdot PH + (NH+1)\cdot PO\right) \quad (11)$$

Where:

NI is the number of input units,
NH is the number of hidden units,
NO is the number of output units,
PI is the number of input units simulated by each processor
PH is the number of input units simulated by each processor
PO is the number of input units simulated by each processor

In the following we will present the results of the algorithm when executing on a network consisting of up to 16 SunBlade 150 workstations, in a 100MB/s network. Each graph presents the average over sets of 10 experiments for the same configuration.

3.1 Results of Varying the Net Size

The following graph presents the efficiency graph using the maximum number of workstations (16). Both the sequential and parallel simulators are executed on nets of exactly the same sizes.

Fig. 2 presents the results for the 16 SunBlades, with nets of sizes such that the number of units simulated by each of the 16 processors varies from 1 to 20 — nets of 16 input, hidden and output units, 32 input, hidden and output units and so forth.

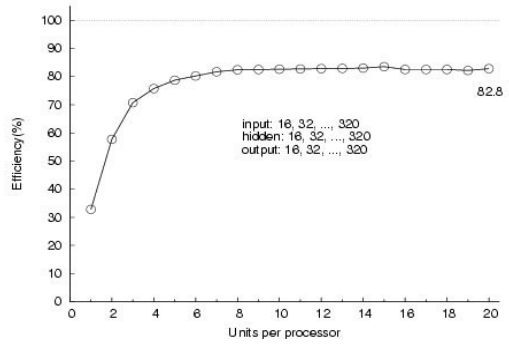

Fig. 2. Efficiency when varying the net size.

In this way, we simulated, from each layer 1, 2, and so on units.

We can see that when there is only one unit per processor in each of the three layers, the efficiency is very low, around 35%, and it raises when adding units. With around seven units per processor the efficiency stabilizes on a value of 80-85%, apparently the highest efficiency we can hope to obtain, even if we add more processors into the ring.

3.2 Results of Varying the Number of Processors

In the following experiments the size of the net is unchanged and the number of processor is varied. The size of the net was chosen such that the maximum number of processors (16) divides the number of units in each layer.

We used nets with 320 units in each layer. To note that it is almost not possible to run the program for 1 or 2 processors on a real application, due to the extreme demands of memory used to store the weights. The values in the graph from Fig. 3 for these processor numbers are based on a modified version which reuses memory. Of course, this version is not able to learn any task, but we have built it to get the execution times.

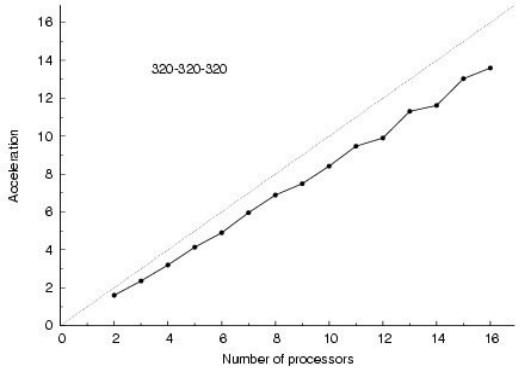

Fig. 3. Acceleration when varying the number of processors.

The graph shows the advantage of distributing the weights among the processors. With a net of such a size it is not possible to use a sequential program, unless the processor is equipped with more memory or uses an extensive memory swapping.

We can simulate very large nets when the number of processors is large.

3.3 Results of Scaling the Net with the Number of Processors

In the next experiments we scaled the net with the number of processors, such that the number of units from each layer per processor is the same, i.e. when using *P* processors we had nets of *P*-*P*-*P*, with one unit from each layer on a processor, 2*P*-2*P*-2*P*, with 2 units from each layer on a processor, and so forth with 3, 5 and 10 units per processor. The graphs in Fig. 4 show the efficiency for 2 to 16 SunBlades.

The graphs show, as expected, that simulating nets with few units per processor gives a low efficiency. With a larger number of units per processor, the efficiency is almost constant, with horizontal lines. If we assume that graphs continue this way for larger number of processors, we can specify if it is possible

to use more processors for a given task. The condition is to have enough units per processor; for one or two units per processor the efficiency is too low.

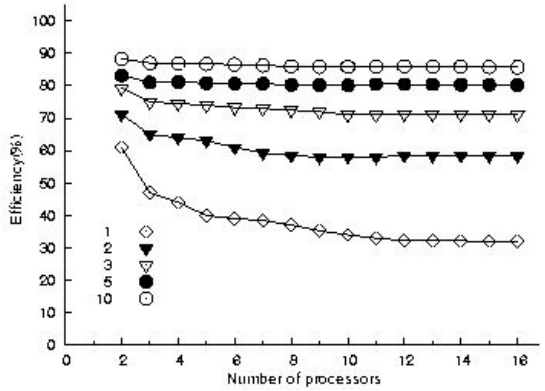

Fig. 4. Efficiency when scaling the net with the number of processors

4. CONCLUSIONS

We developed a distributed simulator for training with the back-propagation algorithm, which uses a partitioning of the network. We tried to reduce the communication to a level which has little effect on the algorithm. We demonstrated that it *is* possible to simulate a large neural network, which requires a large amount of memory.

A large number of processors can only be applied efficiently when large neural networks are simulated.

ACKNOWLEDGMENTS

We are thankful to Alcatel Romania, who kindly allowed us to perform the experiments on one of their network.

REFERENCES

Cosnard, M., and Mignot, J.C. and Paugam-Moisy, H. (1991). Implementations of Multilayer Neural Networks on Parallel Architectures, *2nd International Specialist Seminar on "Parallel Digital Processors"*, Lisbon.

Johansson, E.M. and Dowla, F.U. and Goodman, D.M. (1991). Back-propagation learning for multi-layer feed-forward neural networks using the conjugate gradient method. *International Journal of Neural Systems*, **vol. 2**, pp. 291-302.

Pétrowski, A. and Personnaz, L. and Dreyfus, G. and Girault, C. (1989). Parallel Implementations of Neural Network Simulations. In: *Hypercube and Distributed Computers*, (F. André & J.P. Verjus Eds.), Amsterdam: North Holland, pp. 205-218.

Pétrowski, A. and Dreyfus, G. and Girault, C. (1993). Performance analysis of a pipelined backpropagation algorithm. *IEEE Trans. on Neural Networks*, **vol. 4**, pp. 970--981.

Pomerleau, D.A. and Gusciora, D.L. and Touretzky, D.S. and Kung, H.T. (1988). Neural Network Simulation at Warp Speed: How We Got 17 Million Connections per Second. *IEEE Second International Conference on Neural Networks (2nd ICNN'88)*, IEEE, **vol.II**, pp.143-150.

Rumelhart, D. E., Hinton, G. E. and Williams, R. J. (1986). Learning Internal Representations by Error Propagation. In: *Parallel Distributed Processing — Explorations in the Microstructure of Cognition (*D. E. Rumelhart and J. L. McClelland (Eds.)), pp. 318-362, **Volume 1 — Foundations**. A Bradford Book, MIT Press.

Singer, A. (1990). Implementations of Artificial neural networks on the Connection Machine, *Parallel Computing*, **vol. 14**, pp. 305-315.

Tørresen, Jim (1996). Parallelization of Backpropagation Training for Feed-Forward Neural Networks. PhD. thesis.

Tørresen, Jim and Tomita, Shinji (1998). A Review of Parallel Implementations of Backpropagation Neural Networks. In: *Parallel Architectures for Artificial Neural Networks* (N. Sundararajan and P. Saratchandran (editors)) Chapter 2, IEEE CS Press. ISBN 0-8186-8399-6G.