Report Number: WUCS-94-15

1994-01-01

Strategies for the Parallel Training of Simple Recurrent Neural Networks

Peter J. McCann and Barry L. Kalman

Two concurrent implementations of the method of conjugate gradients for training Elman networks are discussed. The parallelism is obtained in the computation of the error gradient and the method is therefore applicable to any gradient descent training technique for this form of network. The experimental results were obtained on a Sun Sparc Center 2000 multiprocessor. The Sparc 2000 is a shared memory machine well suited to coarse-grained distributed computations, but the concurrency could be extended to other architectures as well.

Strategies for the Parallel Training of Simple Recurrent Neural Networks

Peter J. McCann and Barry L. Kalman

WUCS-94-15

June, 1994

Strategies for the Parallel Training of Simple Recurrent Neural Networks

Peter J. McCann and Barry L. Kalman*

Abstract

Two concurrent implementations of the method of conjugate gradients for training Elman networks are discussed. The parallelism is obtained in the computation of the error gradient and the method is therefore applicable to any gradient descent training technique for this form of network. The experimental results were obtained on a Sun Sparc Center 2000 multiprocessor. The Sparc 2000 is a shared memory machine well suited to coarse-grained distributed computations, but the concurrency could be extended to other architectures as well.

1 Introduction

There has been some work done in the area of parallel neural network training algorithms [7], [10], but very little of it has focused on the training of recurrent networks. It takes an exceptionally large amount of computer time to train these types of networks because of the added complexity of the derivative calculations. In this work, we focus on one type of recurrent network, Elman's Simple Recurrent Network [3], and we present two ways to distribute the gradient computation.

Our first parallel algorithm distributes the network over processing elements. It distributes the most computationally intense part of the gradient calculation during each pattern presentation. As such, it requires a synchronization step for every term of a summation over the training patterns.

Our second parallel algorithm duplicates the network over processing elements, and distributes the gradient computation at a higher level. Fewer synchronization steps are required and more speedup can be obtained. This method requires more duplication of variables, however. It is also not universally applicable in that it depends on the number of independent sequences in the training data.

*The authors are with the Department of Computer Science, Washington University, Campus Box 1045, St. Louis, Missouri 63130-4899. Peter J. McCann is pjm3@cs.wustl.edu . Barry L. Kalman is barry@cs.wustl.edu.

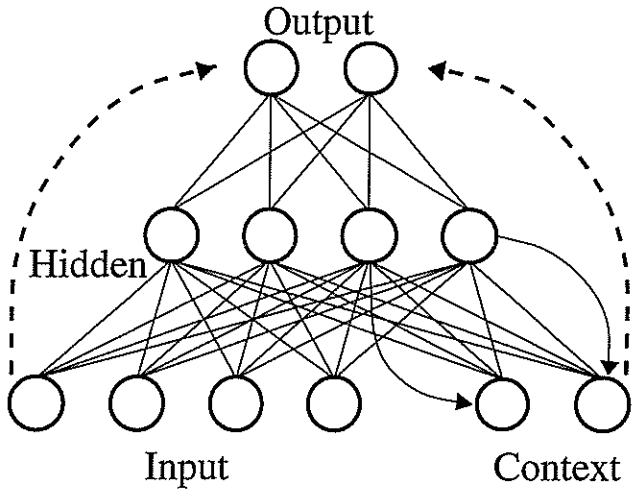

Figure 1: An Elman simple recurrent neural network. Some of the hidden unit activations are copied back to the context layer after each pattern presentation. The dashed lines represent the skip connections from the input and context layers to the output layer.

Figure 1 shows an Elman SRN. This is a partially recurrent neural network capable of learning sequence information. The context units hold copies of the hidden unit activations from the previous pattern presentation, and therefore the output of the network can depend not only on the current input but also on the entire input history. This type of network has found many applications in language processing, time series prediction, and other problems that require a network to maintain an internal state over some period of time.

Our network architecture includes "skip connections" that bypass the hidden layer. These weights fully and directly connect the input and context units to the output units. It has been determined experimentally that these connections allow for faster network convergence. They provide an alternate set of parameters for the linearly separable, or perceptron, portion of the problem. See [4] for a more complete discussion of the rationale for these connections.

Some notational conventions:

t_{po}	The target of output unit o when the network is presented with pattern p.
a_{po}	The activation of unit o when the network is presented with pattern p.
b_o	The bias value of unit o.
w_{ij}	The weight from unit i to unit j.

$\text{hid}(f)$	The hidden unit from which feedback unit f was copied.
$\mathcal{I}$	The set of all input units.
$\mathcal{F}$	The set of all feedback (context) units.
$\mathcal{H}$	The set of all hidden units.
$\mathcal{O}$	The set of all output units.
P	The number of input patterns.

For our error function, we choose a scaled, squared difference of the targets from the actual activations, defined as:

$$\Phi = \sum_{p=0}^{P-1} \sum_{o \in \mathcal{O}} \frac{(t_{po} - a_{po})^2}{1 - a_{po}^2}$$

which is tailored to a hyperbolic tangent squashing function. While the results presented here are applicable to any choice of error function, we should point out that the typical squared error function is not very well suited to variables on a finite domain such as those produced by the output of a neural network. See [6] for a complete treatment of the choice of error function.

For calculating the gradient, we need to take a derivative of our error function with respect to each of the parameters of the network. Taking Y to be some weight or bias in the network, we have

$$\begin{aligned} \frac{\partial \Phi}{\partial Y} &= \sum_{p=0}^{P-1} \sum_{o \in \mathcal{O}} \frac{2}{(1 - a_{po}^2)^2} (t_{po} - a_{po}) \\ &\quad ((t_{po} - a_{po}) a_{po} - (1 - a_{po}^2)) \frac{\partial a_{po}}{\partial Y}. \end{aligned} \tag{1}$$

All of the above terms except for $\frac{\partial a_{po}}{\partial Y}$ are easy to calculate.

Our sigmoidal function is the hyperbolic tangent with a $\frac{3}{2}$ coefficient on the sum of inputs. See [5] for a detailed derivation of this coefficient. With the knowledge that $a_{po} = \tanh(\frac{3}{2}X)$, where X is the weighted sum of all the inputs to unit o at time p, we can calculate this derivative:

$$\begin{aligned} \frac{\partial a_{po}}{\partial Y} &= \frac{3}{2}(1 - a_{po}^2)\Big[\sum_{i \in \mathcal{I}} a_{pi} \frac{\partial w_{io}}{\partial Y} + \sum_{f \in \mathcal{F}} (w_{fo} \frac{\partial a_{pf}}{\partial Y} + a_{pf} \frac{\partial w_{fo}}{\partial Y}) \\ &\quad + \sum_{r \in \mathcal{H}} (w_{ro} \frac{\partial a_{pr}}{\partial Y} + a_{pr} \frac{\partial w_{ro}}{\partial Y}) + \frac{\partial b_o}{\partial Y} \Big] . \end{aligned} \tag{2}$$

Similarly, we can calculate $\frac{\partial a_{pr}}{\partial Y}$, for $r \in \mathcal{H}$ to be:

$$\frac{\partial a_{pr}}{\partial Y} = \frac{3}{2}(1 - a_{pr}^2)\Big[\sum_{i \in \mathcal{I}} a_{pi} \frac{\partial w_{ir}}{\partial Y} + \sum_{f \in \mathcal{F}} (w_{fr} \frac{\partial a_{pf}}{\partial Y} + a_{pf} \frac{\partial w_{fr}}{\partial Y}) + \frac{\partial b_r}{\partial Y} \Big] . \quad (3)$$

Note that the terms such as $\frac{\partial a_{pf}}{\partial Y}$ for $f \in \mathcal{F}$ in the above sums cannot be ignored. If p is not the first pattern in some sequence, these are correctly given by $\frac{\partial a_{(p-1)h}}{\partial Y}$, where $h = \text{hid}(f)$ is the hidden unit that was copied back into unit f. This is because unit f's value is copied directly from unit h from the previous pattern presentation, and changing parameter Y will change this value. For this reason, we need to keep all of these partials in memory from pattern to pattern. If we are to evaluate derivatives due to pattern p, we need information from pattern $p-1$. In other words, we are copying back not only the activations of some hidden units from pattern $p-1$, but also the derivatives of the error function with respect to the weights connected to those hidden units.

At the beginning of every input sequence, we zero the activations of the context units. This choice seems logical in that we would like the network to be in a "quiescent state" before any input is presented to it, but it is in no way dictated by the above equations. Similarly, the equations tell us nothing about how to initialize the $\frac{\partial a_{pf}}{\partial Y}$ values for the first pattern in a sequence. Again, zero seems a good choice for these terms, and experiment has shown that this choice works as well as any other.

Note that these zeroing events partition our sequence of P input patterns into subsequences $s_0, s_1, \ldots, s_n$, where the terms corresponding to each subsequence can potentially be evaluated independently. This will be the source of the concurrency for the second parallel algorithm discussed below, but for now we return our attention to the derivative terms due to weights connected to the hidden layer.

If Y is a weight w_{xo}, where o is any output unit and x is any other unit, then the activations of hidden and feedback units do not depend on this weight. Equations 2 and 3 are then vastly simplified, and are easy to calculate. We have found, however, that over eighty percent of the total computation time, not just of the gradient calculation time, is spent calculating derivatives with respect to parameters such as w_{xh}, which are the weights from input or pseudo-input layers to the hidden layer, and b_h, the biases of the hidden layer units. Taking $o \in \mathcal{O}$, $r \in \mathcal{H}$, and $i \in (\mathcal{I} \cup \mathcal{F})$ we can define

$$\begin{aligned}
\delta_{po} &= 3\Big[\ (t_{po} - a_{po}) - \frac{(t_{po} - a_{po})^2}{1 - a_{po}^2} a_{po}\Big] \\
\delta_{pr} &= \frac{3}{2}(1 - a_{pr}^2) \\
\gamma_{pihr} &= \begin{cases} a_{pi} & \text{if } r = h \ \wedge\ i < \|(\mathcal{I} \cup \mathcal{F})\| \\ 1. & \text{if } r = h \ \wedge\ i = \|(\mathcal{I} \cup \mathcal{F})\| \\ 0. & \text{otherwise.} \end{cases}
\end{aligned}$$

The definition of γ_{pihr} abuses our notation somewhat, but it lends some clarity to the structure of equation 4. We can think of the last weight connected to each hidden unit as a bias for that unit. That is, w_{ih} is defined as b_h when $i = \|(\mathcal{I} \cup \mathcal{F})\|$ if our indices start from 0. This lets us write for $i \in (\mathcal{I} \cup \mathcal{F})$ and $h \in \mathcal{H}$:

$$\begin{aligned}
\frac{\partial \Phi}{\partial w_{ih}} = \sum_{p=0}^{P-1} \sum_{o \in \mathcal{O}} \delta_{po} \Big[\sum_{f \in \mathcal{F}} w_{fo} \frac{\partial a_{pf}}{\partial w_{ih}} \\
+ \sum_{r \in \mathcal{H}} w_{ro} \delta_{pr} \Big[\sum_{f \in \mathcal{F}} w_{fr} \frac{\partial a_{pf}}{\partial w_{ih}} + \gamma_{pihr} \Big] \Big]. \qquad (4)
\end{aligned}$$

2 Concurrency

2.1 Network Partition

The following pseudo-code implements the inner sums from equation 4. It is placed inside a loop over all the patterns, in which each pattern is presented in order and all of the δ_{px} values are calculated. Another portion of the loop should handle the calculations of terms like $\frac{\partial \Phi}{\partial w_{xo}}$. The following code is the interesting part, because we are looking for concurrency that we can easily take advantage of:

```
for h ∈ ℋ
    for i ∈ (ℐ ∪ ℱ)
        { If i = ||(ℐ ∪ ℱ)|| then w_ih is b_h }
        if (w_ih = b_h)  q_exc ← 1.
        else             q_exc ← a_pi
        for r ∈ ℋ
            if (r = h)  x ← q_exc
            else        x ← 0.
            for f ∈ ℱ
                x ← x + w_fr ∂a_pf/∂w_ih
            rof {f}
            dbuf[r] ← x · δ_pr
        rof {r}
        { At this point, dbuf[r] contains ∂a_pr/∂w_ih }
        for o ∈ 𝒪
            x ← 0.
            for f ∈ ℱ
                x ← x + w_fo · ∂a_pf/∂w_ih
            rof {f}
            for r ∈ ℋ
                x ← x + w_ro · dbuf[r]
            rof {r}
            ∂Φ/∂w_ih ← ∂Φ/∂w_ih + x · δ_po
        rof {o}
        for r ∈ ℋ
            ∂a_pr/∂w_ih ← dbuf[r]
        rof {r}
    rof {i}
rof {h}
```

Here $dbuf[\cdot]$ represents an array that holds one floating point number for each of the hidden units in the network. It is used to store the values of the derivatives

of the hidden activations with respect to weight w_{ih} during the presentation of pattern p. This buffer is copied back by the last **for** loop into the storage area for these derivatives, which will be used as $\frac{\partial a_{(p+1)I}}{\partial w_{ih}}$ during the calculations for the next input pattern. Actually, we only need to buffer those derivatives which will be copied back, but for the sake of simplicity we keep all of them separate until the end of the outer loop, as shown above.

Note that we can run each iteration of the loop completely independently. We need only duplicate the local variables q_{exc}, x, and $dbuf[\cdot]$ for each thread of execution. All of the data required by each process can be read from shared memory, and the results of the calculation, the $\frac{\partial \Phi}{\partial w_{ih}}$ values, are all written to separate locations, so there will be no memory contention.

In order to obtain the most performance from a concurrent implementation, the loop with the widest possible scope is the one that should be run in parallel. This is because the overhead introduced by the concurrency will be distributed over a larger and more complex computation, and therefore performance will hopefully not degrade as much as it would in an extremely fine-grained parallelism. One may ask, then, why this particular loop was chosen for concurrent execution, when it appears at least two nesting levels down from the high level computation of a gradient descent method. Our reasoning is three-fold: first, this was a fairly easy code transformation to make. We had only to allocate hidden units to each thread of execution and thereby change the bounds on some loops. Also, this method of obtaining concurrent execution is more feasible than many others. We had only to duplicate a few scalars and one (very small) vector. Finally, a complexity analysis reveals that this portion of code is especially time consuming. Experimentally it was found that over eighty percent of the total computation time was spent in this loop. This allows us to achieve high overall speedups from the concurrency obtained in this small, isolated portion of code.

Note that the complexity of the calculation above is $\Omega(\|\mathcal{F}\|^4)$ because there are at least as many hidden units as feedback units. This means that the time bottleneck this loop already presents for typically sized networks will get even worse as work proceeds towards larger and larger architectures. Taking advantage of the natural opportunities for concurrency will hopefully alleviate this problem.

2.2 Training Sequence Partition

Our second algorithm is usually the more efficient. Its parallelism has a larger scope than the first algorithm, and it therefore includes more of the sequential computation and provides more work for each processor.

We can define $\text{len}(i)$ to be the number of patterns in sequence s_i. We want to schedule the sequences on different processors so that the computation is as load balanced as possible. This means, if we have R processors, we need to

produce an assignment $0 \leq a_i < R$ for each of the sequences s_i such that

$$\max_{0 \leq r < R} \left(\sum_{a_i = r} \text{len}(i) \right) \tag{5}$$

is as small as possible. The problem of finding an assignment of jobs to processors so that 5 is minimized is known to be NP-complete [2], and so we use a variant of the First-Fit Decreasing-Height (FFDH) heuristic analyzed in [1]. We statically allocate the sequences to processors according to the following algorithm: First, sort the sequences in order of non-increasing length. Then, assign each sequence s_i in turn to the least loaded processor, starting with the longest sequence, and adjust the load of that processor upwards by $\text{len}(i)$.

It is straightforward to show that this algorithm performs at least as well as the FFDH algorithm. It is proven in [1] that the length of a schedule resulting from FFDH will have length at most

$$\frac{(R+1)}{R} \text{OPT} + \max_i(\text{len}(i))$$

where OPT represents the length of the optimal schedule.

After sequences have been assigned to processors, we need to make some additional modifications to the sequential code. Since each processor will be doing independent forward propagation, we will need a separate copy of the network activations for each job. Since each processor will be computing a local sum of the gradient components, we will need a separate copy of all the $\frac{\partial \Phi}{\partial Y}$ variables for each job. However, the results are to be computed using only one set of weights, and so all of the w_{ij} and b_i values can be shared.

Note that certain implementations of second order methods may require a line search along the descent direction indicated by the gradient in order to find a minimum in that direction. Our conjugate gradient trainer uses such a search, and we have found that a derivative-free line search involving only evaluations of Φ is the most efficient. The above partitioning of input patterns can be used to perform this forward propagation as well.

3 Results

Our conjugate-gradient trainer is implemented using the available C libraries for multi-threaded execution on the Sun Sparc Center 2000 multiprocessor. There are currently eighteen 40 megahertz Sparc 10 processors on the system. The Sparc 2000 has a shared memory architecture with two high bandwidth packet buses. Each processor maintains its own local cache of main memory, and each processor monitors bus transactions and invalidates portions of its cache when appropriate.

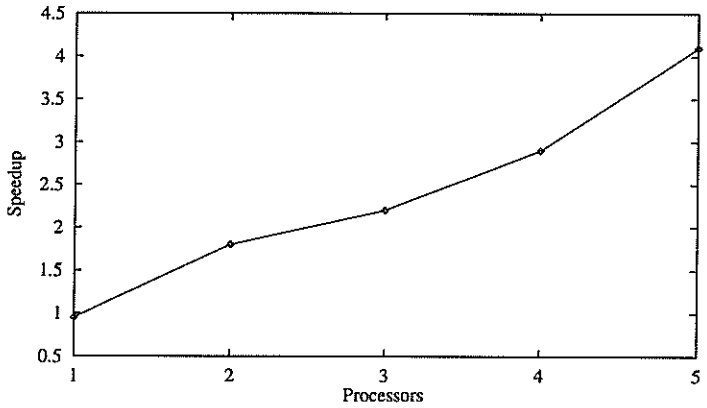

Figure 2: Speedup as a function of processors used to train a (240+10)-10-3 network.

3.1 Network Partition

The speedup values in figures 2 and 3 represent ratios of total real time elapsed during the execution of the loop from section 2.1. The value reported for each processor configuration is the time taken by the sequential version divided by the time taken by the parallel version. These values do not include the part of the code that is still performed sequentially, only the loop from section 2.1 was timed.

The network being trained in figure 2 is a (240+10)-10-3 network, meaning 240 inputs, 10 feedback or pseudo-input units, 10 hidden units, and 3 output units. The problem was to identify the type (classical, talk, or rock-and-roll) of a radio station from digital audio recordings. The sequential version of our conjugate gradient trainer took 22 hours to train on this problem. The five processor version took eight.

The concurrent algorithm was also timed on a natural language parsing problem involving a (34+24)-24-39 network. Figure 3 shows the speedups obtained. The simplicity of the division of labor shows through in the data point for seven processors. The twenty-four hidden units were allocated as evenly as possible over seven processors, and one of the leftovers was given to each of the first three. This means that some processors had a workload of four hidden units, which is the same as the workload for each in the six processor version. The additional overhead of the extra processor is enough to degrade performance. A more load-balanced computation is possible if partial calculations for the hidden units are allowed to proceed on separate processors, i.e., by partitioning the input units as well, but this was not attempted.

To implement this algorithm on a message passing architecture, one would need to broadcast the appropriate weights matrices at every step.

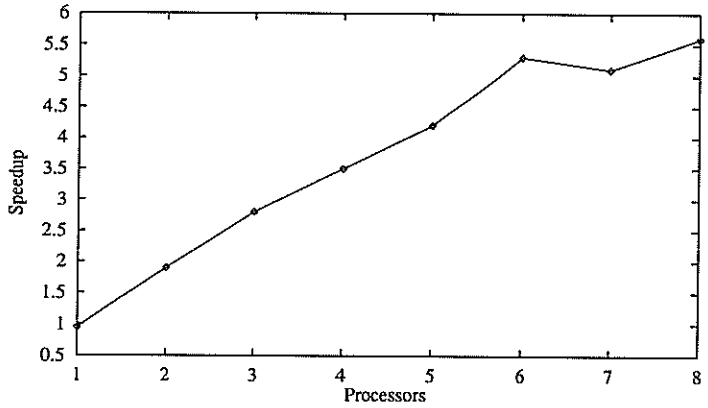

Figure 3: Speedup as a function of processors used to train a (34+24)-24-39 network.

3.2 Training Sequence Partition

The second parallel algorithm was also implemented on the Sparc 2000. This involved the duplication of all the program variables involved in the sum over the patterns, as well as an additional portion of code to sum these duplicated variables into their corresponding global variables. A message-passing implementation would, in addition, require duplication and update of the w_{ij} and b_i values in the local memory of each node.

The test case used was a recurrent version of the NETtalk experiment performed by Sejnowski and Rosenberg [8]. We used an architecture consisting of 103 input units, 20 hidden layer units, 10 feedback units, 2 units in a second hidden layer, and 27 outputs. The use of feedback in this network allowed us to eliminate the "look-behind" portion of the input layer while still achieving results comparable to [8]. This problem was especially amenable to this form of parallelism because it involved a large number of short training sequences, i.e., one sequence for each training word. The training corpus consisted of 4535 patterns organized into 821 sequences.

Figure 4 shows the speedups obtained for a derivative calculation epoch. Figure 5 shows the speedups obtained for a forward propagation epoch. The speedup curves, although they follow a general upward trend until peaking out when synchronization overhead starts to dominate the computation, exhibit behavior that is not readily explainable. The data points for some test cases seem to be much slower than the overall trend would predict. These effects remained constant over several runs, and so are probably not due to background machine load or measurement error. More likely, they are the effects of the limited bus bandwidth and cache utilization that are not easily modeled. Also, the timings of the linesearch calculations seem to be subject to much more of

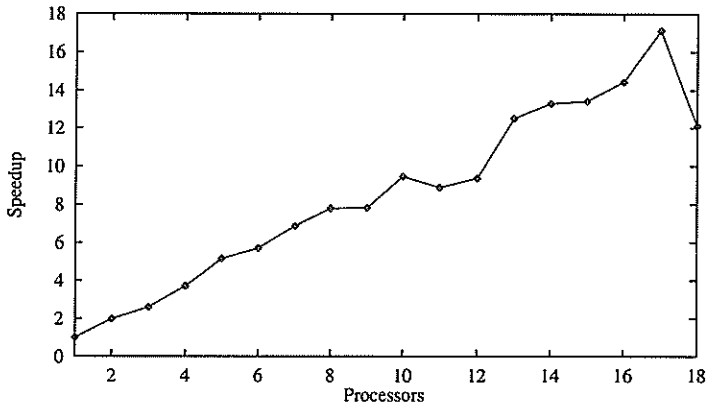

Figure 4: Speedup of a derivative, or back-propagation, epoch as function of processors used.

this deviation, due to the small size of the computation.

A derivative calculation takes about 417 seconds when run on one processor. A forward propagation epoch takes about 11 seconds. Typically, about ten forward propagation epochs will be performed along the gradient direction calculated during each derivative epoch. This is because we do an iterative line search for the error minimum along this direction. Our software allows for scheduling the training patterns onto different numbers of processors for each of these tasks, so that we can take advantage of the best point on each of these curves to achieve maximal speedup. Using 17 processors for each derivative epoch and 6 processors for each line search epoch gives us an overall speedup of 8.96, which includes all computation, not just the derivative and forward evaluations discussed here.

Typically, a single training run will require hundreds of epochs. The overall speedup presented here therefore represents significant savings in time over the sequential version. By reducing the turnaround time, a greater number of network architectures can be investigated, and connectionist research can be more effective.

4 Conclusion

Our first optimization focused on the possibilities for concurrency in one portion of the derivative calculation. This allowed us to achieve some speedup, but it was limited by the size of the network architecture used, and did not allow concurrent computation of the forward propagation step. Our current trainer, although it was more difficult to implement, allows us to partition the set of inputs, which is typically large. This lets us use the available hardware more

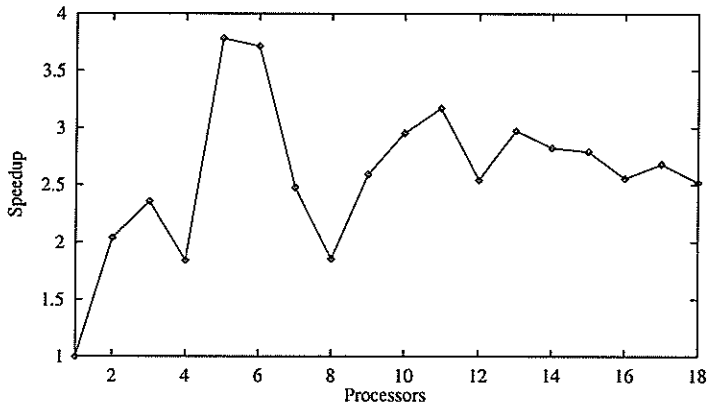

Figure 5: Speedup of a line search evaluation, or forward propagation, epoch as function of processors used.

efficiently.

While the training of recurrent networks, even of simple ones, introduces myriad new complexities over feed-forward network training, our algorithm contains opportunities for concurrency. These opportunities can be taken advantage of after a careful and thorough study of the data dependencies involved. Reducing the real time elapsed during a training run is of great benefit to those undertaking connectionist research projects. It means that more experiments can be conducted in less time than with sequential methods.

5 Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. IRI-9201987. Thanks to Dr. Mark Franklin and the Washington University Computer and Communications Research Center for the use of their Sparc Center 2000 multiprocessor. The Sparc Center 2000 was purchased in part with funds from NSF CISE Instrumentation Grant 9022560.

References

[1] Coffman, E.G., Garey, M.R., Johnson, D.S, and Tarjan, R.E. (1980). Performance Bounds for Level-Oriented Two-Dimensional Packing Algorithms. *SIAM Journal on Computers* 9, 808-826.

[2] Coffman, E.G., ed. (1976). *Computer and Job Shop Scheduling Theory*, New York: John Wiley.

[3] Elman, J.L. (1990). Finding Structure in Time. *Cognitive Science* **14**, 179-211.

[4] Kalman, B.L., and Kwasny, Stan C. (1993). TRAINREC: A System for Training Feedforward and Simple Recurrent Networks Efficiently and Correctly. Technical Report WUCS-93-26, St. Louis: Department of Computer Science, Washington University.

[5] Kalman, B.L., and Kwasny, Stan C. (1992). Why Tanh: Choosing a Sigmoidal Function. *Proceedings of the International Joint Conference on Neural Networks* (Baltimore 1992), vol. IV, 578-581. New York: IEEE.

[6] Kalman, B.L., and Kwasny, Stan C. (1991). A Superior Error Function for Training Neural Nets. *Proceedings of the International Joint Conference on Neural Networks* (Seattle 1991), vol. II, 49-52. New York: IEEE.

[7] Kramer, A., and Sangiovanni-Vincentelli, A. (1989). Efficient Parallel Learning Algorithms for Neural Networks. *Advances in Neural Information Processing Systems 1*, ed. D.S. Touretzky, 40-48. San Mateo: Morgan Kaufman.

[8] Sejnowski, T.J., and Rosenberg, C.R. (1987). Parallel Networks that Learn to Pronounce English Text. *Complex Systems* **1**, 145-168.

[9] Steck, J. E., McMillin, B., Krishnamurthy, K., and Leninger, G.G. (1993). Parallel Implementation of a Recursive Least-Squares Neural Network Training Method on the Intel iPSC/2. *Journal of Parallel and Distributed Processing* **18**, 89-93.

[10] Wu, C.H., and Tsai, J.H. (1992). Concurrent Asynchronous Learning Algorithms for Massively Parallel Recurrent Neural Networks. *Journal of Parallel and Distributed Computing* **14**, 345-353.