RL-TR-92-306
Final Technical Report
December 1992

COMMUNICATIONS NEURAL NET PROCESSORS

Carnegie Mellon University

H.T. Kung, PI

Rome Laboratory
Air Force Materiel Command
Griffiss Air Force Base, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-92-306 has been reviewed and is approved for publication.

APPROVED: Richard N. Smith

RICHARD N. SMITH
Project Engineer

FOR THE COMMANDER: John A. Graniero

JOHN A. GRANIERO
Chief Scientist
Command, Control and Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL(C3BA) Griffiss AFB, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1992	Final - - - - - -

4. TITLE AND SUBTITLE
COMMUNICATIONS NEURAL NET PROCESSORS

5. FUNDING NUMBERS
C - F30602-88-D-0028
Task C-1-2404
PE - 62702F
PR - 4519
TA - 42
WU - P6

6. AUTHOR(S)
H.T. Kung, PI

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Carnegie Mellon University
School of Computer Science
Pittsburgh PA 15213

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rome Laboratory (C3BA)
525 Brooks Road
Griffiss AFB NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RL-TR-92-306

11. SUPPLEMENTARY NOTES
Rome Laboratory Project Engineer: Richard N. Smith/C3BA/(315)330-3091.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This report covers several neural net algorithms. The report describes the rationale for their developments and the details of their operation. The report also addresses the efficient implementation of the algorithms on the i WARP procession. Optimum architectures are also discussed.

14. SUBJECT TERMS
Neural Nets Architectures, Neural Net Processors, Neural Net Algorithms

15. NUMBER OF PAGES 24

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

1 OVERVIEW

The artificial neural network is a powerful new technology with applications in fields such as speech understanding, signal processing, image classification, control, and time-series extrapolation. This technology is built upon two elements: a massively parallel network of very simple processing elements and a learning algorithm that modifies the network's behavior in response to a set of training examples or experiences.

Research on learning algorithms is complementary to work on faster hardware for the simulation of neural networks: any performance gains from improved hardware are multiplied by the performance gains from better algorithms. By combining new high-speed learning algorithms with a fast, massively parallel, scalable machine like the iWarp, we can significantly extend our ability to attack difficult learning problems with large training sets.

A research group at Carnegie Mellon University (CMU) has made considerable progress on developing improved algorithms such as Quickprop [1] and Cascade-Correlation [2]. In addition, CMU has a strong research background in parallel computing and has worked closely with Intel in the development and commercialization of the iWarp parallel array computer. These two strengths were combined in this project. The project had two major tasks:

(1) Implementing advanced neural-net learning algorithms (Quickprop and Cascade-Correlation) on the high-speed iWarp processor.

Although work on this task was delayed by problems with late delivery and instability of the iWarp hardware and system software, we now have both Quickprop and Cascade-Correlation running on iWarp, and the inner loops have been tuned to take full advantage of the machine's speed. We have also developed prototype interface software that makes it possible to use the iWarp as a high-speed neural-net server from any machine on our local area network. However, the interface software requires further development and testing before it will be suitable for routine use.

(2) Improving the accuracy of the Cascade-Correlation algorithm on problems that require analog or continuous-valued outputs. Many problems in the area of communications and signal processing require analog outputs.

Our analysis of Cascade-Correlation performance on continuous-output problems showed that hidden units were being built that over-corrected for the remaining error. A new version of the algorithm, called Cascade 2, was developed to overcome this problem. In addition to improving performance on problems with real-valued outputs, Cascade 2 has unexpectedly shown improved accuracy on a number of pattern-classification problems with binary outputs, though with a small increase in training time. Testing of this algorithm continues, both by our group at CMU and by external colleagues working on a variety of real-world problems. A paper on this new algorithm will appear as soon as this testing is complete. A general release of public-domain code implementing the C2 algorithm will occur at that time.

2 NEURAL NETWORK LEARNING ALGORITHMS

Many algorithms and learning architectures have been proposed over the years. Of these, the back-propagation learning algorithm, operating on a layered feed-forward network of fixed topology, has been the most widely used and successful combination.

Despite its success in certain applications, the standard backprop algorithm suffers from some serious problems that limit the size, complexity, and type of problems to which it can be applied. The most obvious problem is the long time required for learning. If the network is to converge reliably to a good solution, the weights can only be adjusted by a small amount after the presentation of each training example, so even a simple problem might require thousands of iterations over a training set that may contain thousands of examples. The learning time for back-propagation scales up poorly as the size and complexity of the problem or the number of hidden layers in the network is increased.

Learning speed is not the only problem with backprop. It can be hard for the user to find a set of training parameters that will allow the network to converge reliably. The network topology (the number of layers, the number of nodes in each, and the pattern of connectivity) is chosen before the start of training. The selection of a topology for a given problem usually is a matter of guesswork. If the net is too small, it will fail to learn the training examples; if the net is too large, it will successfully memorize the training examples but will generalize poorly when given new inputs not in the training set.

3 QUICKPROP ALGORITHM

Our early investigations into the performance problems of backprop quickly revealed what we call the "step-size problem". The standard back-propagation method computes $\frac{\partial E}{\partial w}$, the partial first derivative of the overall error function with respect to each weight in the network. Given these derivatives, we can perform a gradient descent in weight space, reducing the error with each step. It is straightforward to show that if we take infinitesimal steps down the gradient vector, recomputing the gradient after each step, we will eventually reach a local minimum of the error function. Experience has shown that in most situations this local minimum will be a global minimum as well, or at least a reasonably good solution to the problem at hand.

In a practical learning system, however, we do not want to take infinitesimal steps. For fast learning, we want to take the largest steps that we can. Unfortunately, if we choose a step size that is too large, the network will not reliably converge to a good solution. In order to choose a reasonable step size, we need to know not just the slope of the error function, but something about its second-order derivative---its curvature---in the vicinity of the current point in weight space. This information is not available in the standard back-propagation algorithm.

The Quickprop algorithm computes the $\frac{\partial E}{\partial w}$ values just as in standard backprop, but instead

of simple gradient descent, Quickprop uses a second-order method, related to Newton's method, to update the weights. Quickprop's weight-update procedure depends on two approximations: first, that changes in one weight have relatively little effect on the error gradient observed at other weights; second, that the error function with respect to each weight is locally quadratic.

For each weight, Quickprop keeps a copy of $\frac{\partial E}{\partial w(t-1)}$, the slope computed during the previous training cycle, as well as $\frac{\partial E}{\partial w(t)}$, the current slope. It also retains $\Delta w(t-1)$, the change it made in this weight on the last update cycle. For each weight, independently, the two slopes and the step between them are used to define a parabola; we then jump to the minimum point of this curve. Because of the approximations noted above, this new point will probably not be precisely the minimum that we are seeking, but it will serve as a very good next step in an iterative algorithm. In practice, some complications must be added to the simplified algorithm presented here; see [1] for details.

On the learning benchmarks we have tried, Quickprop consistently out-performs other backprop-like algorithms, often by a factor of 10 or more. For example, Quickprop can solve the simple 2-2-1 XOR problem in a median time of 19 epochs; standard back-propagation typically requires 250 epochs, and many researchers report even slower times. (An epoch is defined as a single pass through the complete set of training examples.) In the few cases in which Quickprop has been run on the same problems as conjugate gradient methods, Quickprop has generally performed better. However, the conjugate-gradient methods may converge faster in very high-dimensional search spaces.

Quickprop is very simple to implement. It is identical to back-propagation with momentum except in the weight-update step, and that requires only that one additional value (the previous slope) be kept for each weight. All the information Quickprop needs to update each link is available locally; there is no additional need for communication among units.

4 CASCADE-CORRELATION ALGORITHM

A second, and more serious, source of inefficiency in back-propagation learning is what we call the "moving target problem". Suppose we have a problem that requires ten hidden units for its solution. This means that there are ten distinct jobs that must be performed by units in the hidden layer. Each hidden unit must evolve during learning into a feature detector that will play one of these roles, and each node must ultimately choose a different role from all the others.

This task is very difficult if all the hidden units are evolving at the same time. There is no central authority assigning units to roles. There is no communication between the units, except for their mutual and ever-changing effect on the global error. Under these conditions, it can take a very long time for all ten jobs to be filled. Instead of moving quickly and directly to assume some useful role in the network, a unit must engage in a complex game of musical chairs with all the other units. The situation is even worse in networks with multiple hidden layers because the error

signals back-propagating through the network are mixed together and weakened as they pass through successive, ever-changing layers of units.

The Cascade-Correlation (or "Cascor") algorithm [2] avoids the moving-target problem by adding new hidden units to the network one at a time. Instead of a moving target, each unit sees a fixed error surface. It can move quickly and directly to occupy some role that will be useful in reducing the error. The unit is then locked in place, and subsequent units will work only on the remaining components of the error. This "greedy" learning may sometimes overlook an optimal solution in which the units must cooperate in some intricate way, but it will find a reasonably good solution very fast. Algorithms such as Cascor, which build up an appropriate network structure in the course of learning, are called "constructive" learning algorithms.

The "cascade" in Cascade-Correlation refers to the network architecture shown in Figure 1. It begins with some inputs and one or more output units, but with no hidden units. The number of inputs and outputs is dictated by the problem and by the I/O representation the experimenter has chosen. Every input is connected to every output unit by a connection with an adjustable weight. There is also a "bias" input, permanently set to +1.

The learning algorithm begins by training the direct input-output connections as well as possible over the entire training set. In our simulations, we use Quickprop to adjust the output weights, since it is faster than simple gradient descent. At some point, this training will approach an asymptote. When no significant error reduction has occurred after a certain number of training cycles, we run the network one last time over the entire training set to measure the error. If we are satisfied with the network's performance, we stop; if not, there must be some residual error that we want to reduce further. This can only be done by creating a new hidden unit.

To create a new hidden unit, we begin with a pool of candidate units. Each of these receives a weighted connection from each of the network's external inputs and from any pre-existing hidden units. The candidates' outputs are not yet connected to the active network. Each candidate starts with a different random set of initial weights, so the candidates explore different parts of the weight-space in parallel. We run a number of passes over the set of training examples, adjusting each candidate's input weights so as to maximize the correlation between the candidate unit's output value and the residual error seen at the outputs of the active net. Once again, we use Quickprop for the actual weight adjustment.

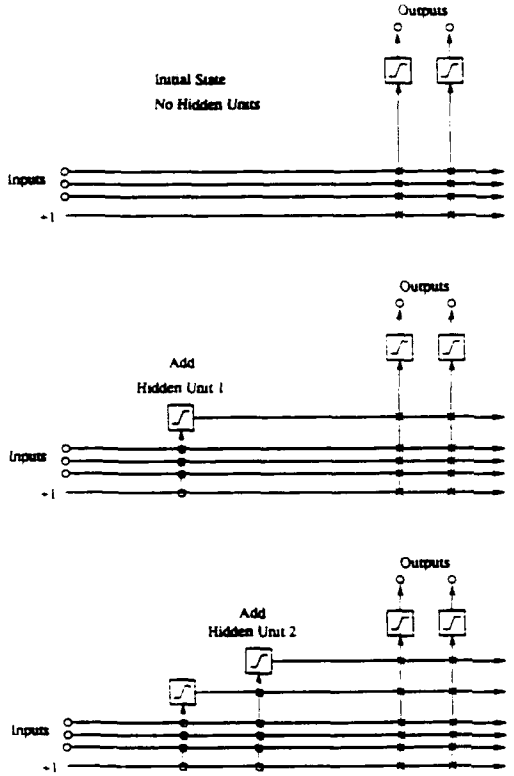

Figure 1 The Cascade Architecture, initial state and after adding two hidden units. The vertical lines sum all incoming activation. Boxed connections are frozen. X connections are trained repeatedly.

When the correlation scores stop improving, we choose the candidate unit with the best score and add it to the active network. (This is called "tenure".) At this point, the unit's incoming weights are frozen, so it becomes a permanent, unchanging feature detector in the network. Its output is now connected to all of the network's output units, and we repeat the output-training phase with this new unit in place. We should be able to do somewhat better than before. Since the new hidden unit was able to correlate with some part of the remaining error, it should now be able to cancel some of it. We repeat this cycle of output training, candidate training, and installation of a new hidden unit until the error is acceptably small (or until we give up).

We use a pool of candidates because some units may become stuck in uninteresting parts of the weight-space. We select only the unit that best matches the residual error. For most of our experiments we have used a pool-size of 8 candidates. Larger pools take longer to train but sometimes result in networks with fewer hidden units.

Since the candidate units receive connections from all pre-existing hidden units, each new hidden unit is in effect a new layer in the network. This allows us to create complex, high-order

feature detectors, but without the cost of back-propagating errors through multiple layers of network. As the net becomes larger, this may lead to the creation of more connections than are necessary to solve the problem. We propose to investigate several strategies for eliminating these excess connections in the course of training.

Cascor eliminates the need for the user to guess in advance the network's size, depth, and topology. A reasonably small (though not necessarily minimal) network is built automatically. Because a hidden-unit feature detector, once built, is never altered or cannibalized, the network can be trained incrementally. A large data set can be broken up into smaller "lessons", and feature-building will be cumulative.

Because of its greedy learning strategy, and because it only trains one layer of weights at any given time, Cascor learns very quickly. Speedups of 10x to 100x over backprop are common in problems of moderate size. In addition, Cascor scales up well. Our experience has been that the number of epochs required to train a network is almost linear (maybe N log N) in the number of hidden units (and layers) that are ultimately needed. Backprop scales up poorly as problems require more units and layers.

5 CASCADE 2 ALGORITHM

Cascor works very well for classification problems and other problems with binary-valued outputs. The inputs can be either binary or continuous. However, we have determined that it performs less well on problems such as function approximation that require continuous-valued outputs.

As part of the work under this contract, we studied this problem. We determined that the correlation measure (really a covariance) used in training the candidate units tends to overshoot small errors and drive the candidate unit into positive or negative saturation. We then developed a new learning algorithm called "Cascade 2" or "C2" that replaces the covariance measure with one that minimizes the sum-squared difference between unit's scaled output value and the residual error for the current training case. In effect, each candidate unit acts as a single-unit hidden layer, with its own trainable input and output weights. As before, when quiescence is reached in this training phase, the best scoring is added to the network with its input weights frozen.

Cascade 2 works much better than the original Cascade-Correlation algorithm on continuous-output problems, while retaining the speed and other desirable properties of the original algorithm. One popular benchmark for time-series prediction with a neural net has been prediction of future values for the Mackey-Glass equation. The sum-squared error of such predictions using Cascade 2 is typically 4 to 8 times smaller with C2 than with Cascor.

To our surprise, we have discovered that C2 gives improved performance over Cascor on some binary-output problems as well. For example, on the two-spirals benchmark, C2 can easily find solutions with 10 or 11 hidden units, while Cascor typically requires 13 to 15. Cascor can

learn the 10-bit parity problem after training on just 256 of the 1024 possible 10-bit patterns, but it typically gets 10% errors on the examples not in the training set; C2 can solve the same task with only 2% error.

We are in the process of testing the Cascade 2 algorithm on some larger training sets before preparing a tech report and submitting a journal paper on this work. Public-domain code implementing this algorithm (versions in both C and Common Lisp) will be released at the same time as the tech report. Once the C2 algorithm is published, it will be a simple matter to modify the iWarp-cased simulator to include the C2 algorithm as an option.

6 THE iWARP SIMULATION

It is fine to have new algorithms that improve the learning speed of neural networks by a factor of 10-100 or more, but to get full value from these new algorithms they must be implemented on the fastest available machines. We chose to use the iWarp for a number of reasons:

- At 20 MFLOPS, each iWarp cell is comparable in performance to the best available DSP chips, while offering much better inter-cell communication than conventional DSP chips can provide.
- Intel is marketing iWarp as a product, so the neural net tools we develop on this system will be readily available to other researchers as an off-the-shelf, high-performance solution for neural-net development. If iWarp is successful, Intel may produce faster versions in the future.
- Many tasks involve the use of standard DSP techniques on a raw signal, followed by a neural network to interpret and classify the result. The iWarp machine excels at DSP applications, and will already be present in some laboratories that want to use our neural-net tools.
- While the typical iWarp array employs 64 cells in an 8x8 array, it is possible to build larger or smaller systems to fit a given application. We use the iWarp cells as a 1-D systolic chain, so the same software can easily be re-configured to run on any iWarp configuration.
- The Nectar project at CMU is developing a gigabit network that will be able to link together multiple iWarp arrays. Gigabit Nectar is expected to be operational late this year and at that time it will be possible to string together several iWarp arrays into one large neural-net simulator.
- Since the iWarp architecture and some of its software was designed by people at Carnegie Mellon, we have local expertise in using this chip. In addition, the iWarp group at CMU is interested in developing applications, such as neural nets, that demonstrate the power and usefulness of the iWarp machine.

The big disadvantage of using the iWarp at this time is that it is a new and largely untried machine. This caused a number of problems and delays.

There are several ways in which the inner loops can be divided up to run on an N-processor parallel machine like iWarp:

1. Divide a large neural network into N equal-sized pieces, one on each of the N processors, and run the full data set through this distributed network. This scheme has the disadvantage that it may be impossible to divide a net into N equal chunks without an excessive number of inter-processor connections.

2. Each processor's local memory holds an identical copy of the full network, plus 1/N of the training examples. During an epoch, each processor runs the full net on its local training set. The dE/dw values are collected across all processors before the weights are updated, so all of the copies of the full network remain identical.

3. As in 2, each processor runs the full network on 1/N of the training cases. However, the weights are not stored redundantly in each processor. A single copy of the weight set is kept and it is circulated to all processors using the systolic data paths. As each weight comes by, it is used in the forward-propagation step and then is passed on to the next processor in a 1-D chain. During the back-propagation step, the weights are circulated and used in reverse order, and each weight is followed by a "bucket" that accumulates the dE/dw values associated with that weight. This is the scheme used successfully in the original Warp simulator.

4. Each processor runs the full net on the full training set, but we can run N different trials at once, each with a different set of random starting weights. This mode is only useful if we want to run many trials, perhaps because it is important to find a really good solution.

After some study, we decided to follow plan 3 in both our quickprop and cascade-correlation simulators.

Our plan was to put only the performance-critical inner loops of the neural-net simulator on the iWarp itself. The outer loops of the algorithm run on the iWarp's host machine (a Sun 4 running SunOS). The setup, display, and analysis tools run in the friendly software environment of the user's own workstation, connected to the host over a local area network.

We use the shared Andrew File System (AFS) for communication of large chunks of data between the user's machine and the iWarp host. These chunks include network topology files, weight files, parameter sets, and files holding input and output data for training and test cases. A TCP/IP connection is used to carry simple commands and responses between the user's machine and iWarp host. At present, the commands sent over this connection include the following:

Read/Write Topology file from/to AFS.

Read/Write Weights file from/to AFS.

Read/Write Parameter file from/to AFS.

Read/Write Input-data file from/to AFS.

Read/Write Output-data file from/to AFS.

Use specified range of data values as training set.

Use specified range of data values as test set.

Write Unit Values file to AFS.

Initialize Net, Randomizing Weights.

Execute N training epochs, then report on error.

Execute N training epochs, but stop early if success criterion is satisfied.

Run the test set, then report on error.

For ease in interfacing, the AFS files are stored in ASCII format, but they are converted into integers and floating-point values when read by the iWarp host machine. In the future, we plan to convert these file formats to use 32-bit integers and IEEE floating-point rather than ASCII. This will speed up loading and dumping of data sets, though this may create some problems in compatibility between different machines on the network.

At present, logging in to all the necessary machines and setting up all of the necessary protections and permissions is a tedious and error-prone business. These problems should soon be eliminated by improvements to the system software on the iWarp hosts. In any event, it is not a problem for users willing to sit at the iWarp host machine and work there.

We have not yet completed the user-interface code and display code that will run on the user's own workstation. We felt that it was important to finish the iWarp part of the project first, and then the communication links. These tasks were only finished in the last two weeks. We believe that it will be a simple matter to complete the user-interface code, since we can take an existing workstation-based neural-net simulator and replace its inner loops with calls to iWarp.

7 PERFORMANCE MEASUREMENTS

The performance of hardware implementations of backprop (and of closely related algorithms such as quickprop) is typically quoted in Connections Per Second (CPS) or Million Connections per Second (MCPS). Unfortunately, there are many different ways of calculating CPS figures, so numbers quoted by different authors are not always directly comparable. We compute CPS as $\frac{(eps \times Npat \times Ncon)}{t}$, eps is the number of training epochs run, $Npat$ is the number of training patterns per epoch, $Ncon$ is the number of weighted connections in the network, and t is the total time required.

This measure therefore reflects the time required for both the forward and backward passes through the network, plus the time spent in weight updates and other overheads that fall outside the inner loops of the program. For very large problems with many training cases and many connections running to each unit, the overheads become negligible, and we see only the time required

to cross each connection once during the forward pass and once during the backward pass. This is called the "asymptotic MCPS".

For comparison purposes, most Unix workstations run backprop simulations at less than 1 MCPS. A commercial "neurocomputer" board advertised by SAIC (actually just a single DSP chip and some memory that can be plugged into a PC) is rated at 2.8 MCPS. The backprop simulator implemented at CMU on the old Warp machine (10 processors, 100 MFLOPS total) achieved a peak rate of 20 MCPS. A backprop simulator implemented on the experimental GF11 machine at IBM (rated at 11 GFLOPS) achieved a peak rate of over 1000 MCPS, though this machine was never stable enough to make it out of the lab or to be used for anything serious.

The iWarp is rated at 20 MFLOPS per cell. A typical iWarp array is 8x8, or 64 cells in all, for a total rating of 1.28 GFLOPS. Our simulation programs rely heavily on the "combined" instruction mode of the iWarp, which allows a dot product step to be computed in a single two-cycle instruction of 100 nsec. In this time, the iWarp processor must receive a floating-point value from one of its systolic neighbors, do a 32-bit floating multiply-accumulate calculation, tick and check a counter, and pass a value on to the next processor in the chain.

On a chip in which this combined instruction is operating at full speed, our program requires one combined instruction per weight in the quickprop forward pass, and two such instructions for the backward pass: 300 nsec in all for these innermost-loop operations. For very large problems, this would allow us to approach a limit of 3.33 MCPS (as defined above) per processor, or 213 MCPS for a 64-cell array. For smaller problems, overheads become significant and consume an increasing fraction of this performance.

We have implemented and tested the quickprop and cascor simulation code on an iWarp machine with Intel's latest (c-step) iWarp chips. The code runs properly, but these chips contain a timing problem that prevents us from achieving the peak asymptotic rate calculated above. The forward pass must use two instructions per weight rather than 1, so the asymptotic MCPS is reduced to 2.5, and the theoretical peak rate of a 64-processor iWarp using these chips is reduced to 160 MCPS. We expect that Intel will eventually fix this timing problem.

Our iWarp machines currently have only .5M bytes per cell. This memory limitation prevents us from running networks with really large training sets. However, we have measured nets with up to 15,441 weights and 64,000 training cases, and have observed that for such nets we reach almost 2.1 MCPS per processor. We have developed an accurate model that can be used for predicting the performance of larger nets. This shows that for much larger nets and training sets, we can expect a smooth convergence toward the current limit of 2.5 MCPS per cell.

Figure 2 shows the measured MCPS per processor as a function of training set size for a network with a fixed 384-40-1 topology (15441 weights total). Figure 3 shows the performance for a 64 cell iWarp system.

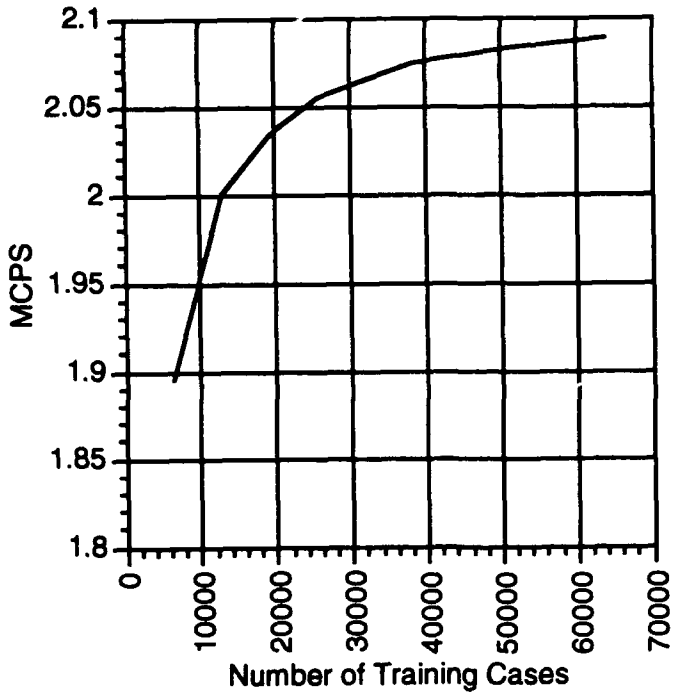
2.1
2.05
2
1.95
1.9
1.85
1.8
MCPS
0
10000
20000
30000
40000
50000
60000
70000
Number of Training Cases

Figure 2

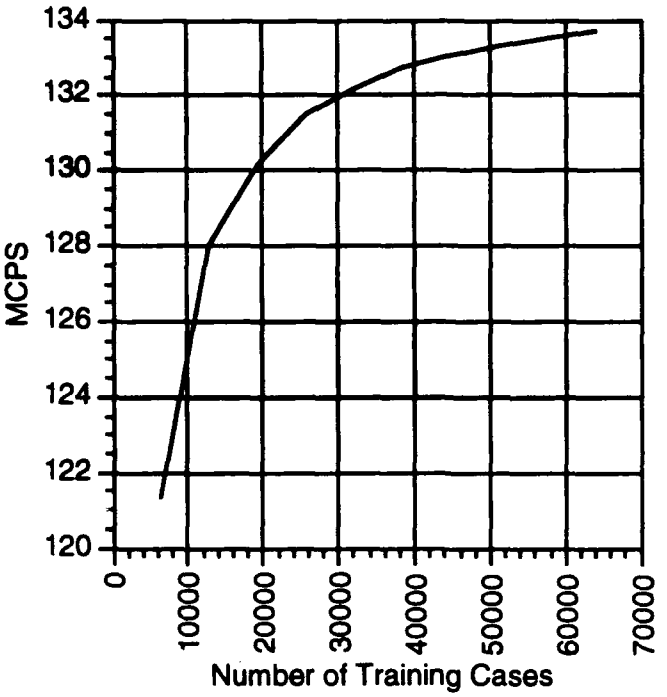

Figure 3

Figure 4 shows the measured MCPS per processor as a function of network size for a network with a fixed set of 64,000 training cases. Figure 5 shows the performance for a 64 cell system.

Table 1 summarizes typical test results. All measurements are for a single training epoch on a 64-processor C-step machine. The * indicates estimates from our performance model; all other results are measured times on the actual machine. It can be observed that as the problem size increases, the overheads become less significant and performance approaches the theoretical maximum as expected.

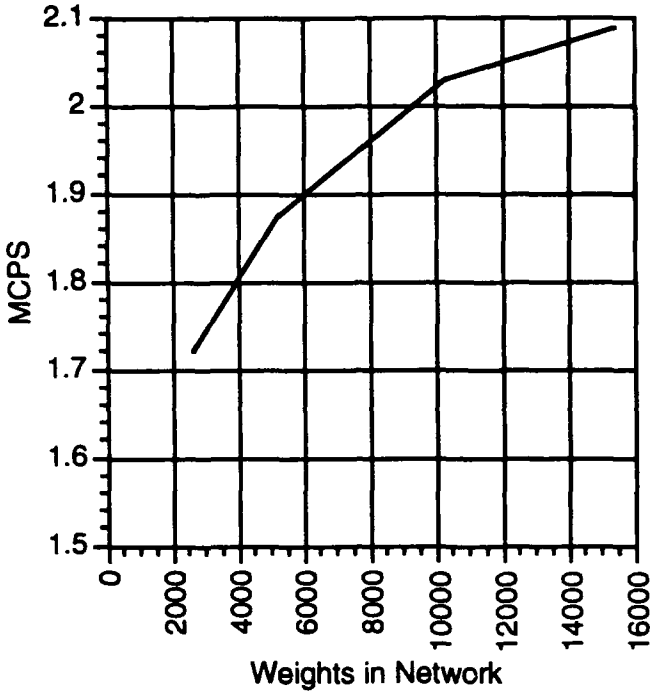
2.1
2
1.9
1.8
1.7
1.6
1.5
MCPS
0
2000
4000
6000
8000
10000
12000
14000
16000
Weights in Network

Figure 4

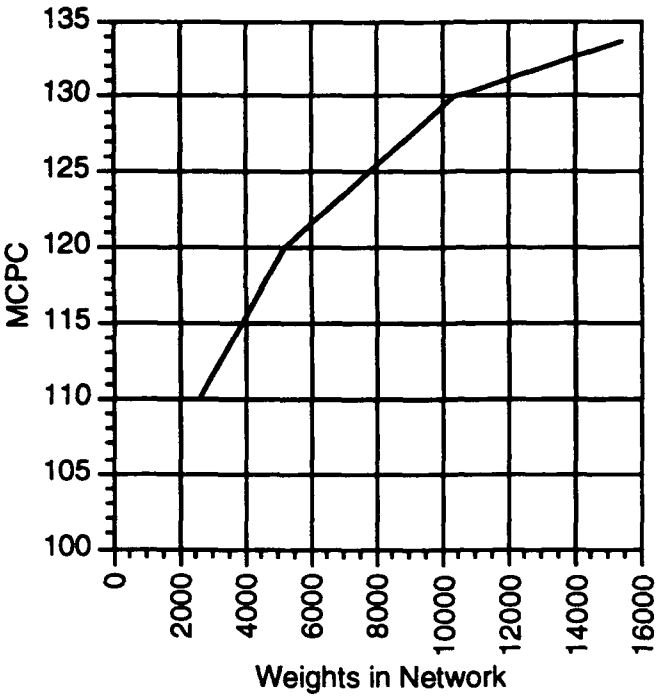
135
130
125
120
115
110
105
100
MCPC
0
2000
4000
6000
8000
10000
12000
14000
16000
Weights in Network

Figure 5

Inputs	Hidden Units	Outputs	Epochs	Weights	Cases	Total Time in 50 nsec. cycles	MCPS per Cell	MCPS for 64 Cell System	
128	20	1	1	2,601	6,400	3.27E+06	1.592	102	
					12,800	6.28E+06	1.656	106	
					19,200	9.27E+06	1.683	108	
					25,600	1.22E+07	1.702	109	
					32,000	1.52E+07	1.706	109	
					38,400	1.82E+07	1.713	110	
					44,800	2.12E+07	1.716	110	
					51,200	2.42E+07	1.717	110	
					57,600	2.72E+07	1.722	110	
					64,000	3.02E+07	1.722	110	
384	20	1	1	7,721	6,400	8.82E+06	1.751	112	
					12,800	1.68E+07	1.836	118	
					19,200	2.48E+07	1.866	119	
					25,600	3.28E+07	1.885	121	
					32,000	4.08E+07	1.893	121	
					38,400	4.88E+07	1.900	122	
					44,800	5.68E+07	1.904	122	
					51,200	6.47E+07	1.908	122	
					57,600	7.28E+07	1.910	122	
					64,000	8.07E+07	1.913	122	
1152	1000	1	1	1,154,001	6,400	9.62E+09	2.400	154	*
9000	5000	1	1	45,000,000	576,000	3.26E+12	2.487	159	*

*Estimates from a performance model with future compiler upgrade

Table 1 Typical Performance Results

8 Conclusions

The iWarp parallel array has proven to be a good machine for neural net simulations. Even with the timing problem in the current c-step component, we can approach 160 MCPS on a 64 cell array. If the bug is corrected, 3.33 MCPS per cell or over 200 MCPS for a 64 cell system should be expected. In both cases, the performance is the theoretical maximum for the machine. We also fully expect performance to scale linearly with the size of the array. Thus, a 256 cell iWarp (which has been built) with corrected c-step components should achieve 850 MCPS performance. This is very attractive performance for a relatively inexpensive and general purpose parallel machine with a rated performance of 20 MFLOPS per cell.

9 REFERENCES

Fahlman, S. E. (1988) "Faster-Learning Variations on Back-Propagation: An Empirical Study" in Proceedings of the 1988 Connectionist Models Summer School, Morgan Kaufmann.

Fahlman, S. E. and Lebiere, C. (1990) "The Cascade-Correlation Learning Architecture" in D. S. Touretzky (ed.), Advances in Neural Information Processing Systems 2, Morgan Kaufmann.

D. A. Pomerleau, G. L. Gusciora, D. S. Touretzky, and H. T. Kung (1988) "Neural network simulation at Warp speed: How we got 17 million connections per second", in Proceedings of ICNN-88, San Diego.

*U.S. GOVERNMENT PRINTING OFFICE 1993-710-093-60184

MISSION

OF

ROME LABORATORY

Rome Laboratory plans and executes an interdisciplinary program in research, development, test, and technology transition in support of Air Force Command, Control, Communications and Intelligence (C^3I) activities for all Air Force platforms. It also executes selected acquisition programs in several areas of expertise. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. In addition, Rome Laboratory's technology supports other AFSC Product Divisions, the Air Force user community, and other DOD and non-DOD agencies. Rome Laboratory maintains technical competence and research programs in areas including, but not limited to, communications, command and control, battle management, intelligence information processing, computational sciences and software producibility, wide area surveillance/sensors, signal processing, solid state sciences, photonics, electromagnetic technology, superconductivity, and electronic reliability/maintainability and testability.