N88-17221

DESIGN OF A NEURAL NETWORK SIMULATOR ON A TRANSPUTER ARRAY

Gary McIntire
Advanced Systems Engineering Dept. Ford Aerospace, Houston, TX.

James Villarreal, Paul Baffes, and **Monica Rua** Artificial Intelligence Section, NASA/Johnson Space Center, Houston, TX.

Abstract

A high-performance simulator is being built to support research with neural networks. All of our previous simulators have been special purpose and would only work with one or two types of neural networks. The primary design goal of this simulator is versatility; it should be able to simulate all known types of neural networks. Secondary goals, in order of importance, are high speed, large capacity, and ease of use. A brief summary of neural networks is presented herein which concentrates on the design constraints imposed. Major design issues are discussed together with analysis methods and the chosen solutions.

Although the system will be capable of running on most transputer architectures, it currently is being implemented on a 40-transputer system connected in a toroidal architecture. Predictions show a performance level nearly equivalent to that of a highly optimized simulator running on the SX-2 supercomputer.

Introduction

There are several ways to simulate large neural networks. Computationally speaking, some of the fastest are via optical computers and neural net integrated circuits (hardwired VLSI). However, both methods have some basic problems that make them unsuitable for our research in neural networks. Optical computers and hardwired VLSI are still under development, and it will be a few years before they will be commercially available. Even if they were available today, they would be generally unsuitable for our work because they are very difficult (usually impossible) to reconfigure programmably. A non-hardwired VLSI neural network chip does not exist today but probably will exist within a year or two. If done correctly, this would be ideal for our simulations. But the state of the art in reconfigurable simulators are supercomputers and parallel processors. We have a very fast simulator running on the SX-2 supercomputer (200 times faster than our VAX 11/780 simulator), but supercomputer CPU time is _very_ costly.

For the purposes of our research, several parallel processors were investigated including the Connection Machine, the BBN Butterfly, and the Ncube and Intel Hypercubes. The best performance for our needs was exhibited by the INMOS Transputer System.

Our previous neural network simulators have been specific to one particular type of algorithm and consequently would not work for other types of networks. With this simulator our primary goal is to be able to implement all types of networks. This will be more complicated but is deemed well worth the effort. When we are finished, it should be possible to implement a different kind of network in less than a day. The performance reduction will be less than 10 percent for this general-purpose capability.

Example networks

To have examples to work with, let us consider two very typical neural networks. The first is a three-layer feedforward network (fig. 1) that is to be trained with the generalized delta learning algorithm (also called back propagation). Rumelhart et. al.[5] describe this algorithm in detail. We will assume that every node in one layer is connected to every node in its adjacent layer. The network will be trained with a number of I/O pairs which are an encoding of the associations to be learned. The sequence of events to the algorithm can be described as follows. First, an input vector is placed into the input nodes. The weight values of the connections between the input layer and hidden layer are then multiplied by the output value of the corresponding input nodes and the result is propagated forward to the hidden layer. These products are collected and summed at each node of the hidden layer. Once all the nodes in the hidden layer have output values, this process is repeated to propagate signals from the hidden layer to the output layer. When the output layer values are computed, an error can be calculated by comparing the output vector with the desired output value of the I/O pair.

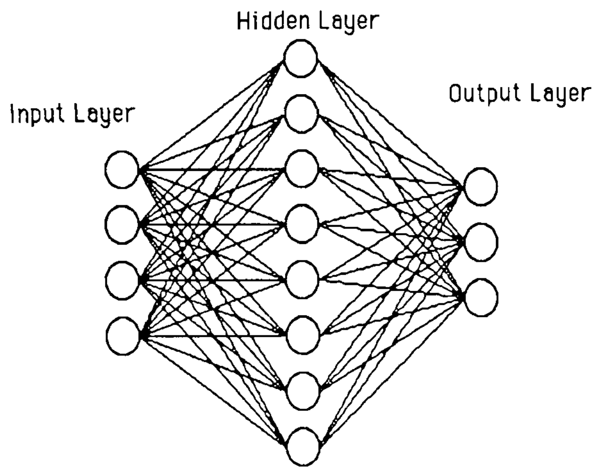

Figure 1. - Feedforward network.

With this error computed, the weights at each connection between the hidden and the output layer can be adjusted. Next, the error values are backpropagated from the output layer to the hidden layer. Finally, each weight can be adjusted for the connections between the input and hidden layers.

This entire sequence is repeated for each successive I/O pair. Note that this algorithm has a sequence of events to it. Other neural net algorithms do not; instead, every node and connection updates itself continuously[2,4]. Such algorithms can be viewed as a sequence of length one.

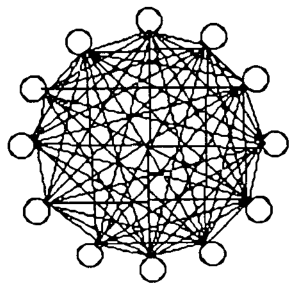

Figure 2. - Hopfield net.

The second network to be considered is the Hopfield network[1] shown in figure 2. It is an auto-associative memory in which every node is equivalent; i. e., they are all used as inputs and outputs. Every node is connected to every other node but is not connected to itself. The connections are bidirectional and symmetric which means that the weight on the connection from node i to node j is the same as the weight from node j to node i.

Other neural networks impose different constraints. For example, the connection scheme may be totally random, or it may be that every ninth node should be connected. The equations used are very often different. The order of sequencing forward propagation, back propagation, weight adjusting, loading inputs, etc., also may be different. Stochastic networks update their values probabilistically. Yet some generalizations can be made for almost all types of networks and these are what we have used as the basis of our simulator design.

The first generalization is that all network computations are local computations. In other words, the computations only involve a node and the nodes to which it is connected.

This is a generally accepted principal in the literature and some authors even use it in the definition of a neural network.

The second generalization is that all neural computations can be performed by applying functions to the state variables of a node and its neighbors. While this is a trivial restatement of Turing equivalence, the emphasis here is that this can be done in a computationally efficient manner.

The third generalization is that any computation performed by a node on its input signals can be decomposed into parallel computations that reduce multiple incoming signals to a single signal. This single signal is then sent to a node to be combined with single signals sent from the other parallel computations. This allows a node computation to be broken apart, and the partial results forwarded to the node which computes the final result. For example, a computation that is common to almost all networks is the dot product which is a sum of products. This can be decomposed into multiple sums of products whose results are forwarded and summed.

Memory Utilization

For maximum memory efficiency, the target design should have one memory cell of the minimum possible size for each state variable which the network must keep. These state variables consist of the values kept for the connections and nodes. For a connection in the two networks above, one memory cell per connection is needed since the only information associated with a connection is its weight. If the connectivity is like that of the above two networks (where everything in one layer in connected to everything in another layer), the connection information can be implicit. At the other extreme, where the connectivity is totally random, a additional pointer between nodes would have to be kept for each connection.

For nodes, a network like the Hopfield network only has to keep one variable: its output value. The size of this output value, however, can vary. Some networks work with 8-bit or 16-bit integers while others use floating point numbers. Other kinds of networks require additional state variables at the nodes and connections. Almost all networks include added representations to ease debugging tasks. Because of these differences in size, we have allowed the user to specify node variables and their types by defining structures in the C language to hold the state variable information. This allows all the flexibility of C (ie. integers, floats, doubles, bytes, bit fields, etc.).

CPU Utilization

Since previous simulations have shown that a neural net simulator spends almost all of its time processing connections (typically, there are many more connections than there are nodes), an examination of execution speed must focus on the calculations done for each connection. The operation common to almost all neural nets is some function of the dot product. This is

$$O_i = \mathbf{f}(\Sigma\, W_{ij} O_j)$$

where O_i is the output of the i th node, Oj is the output of the j th node, W_{ij} is the connecting weight, and **f** is some function applied to the dot product (note that the time spent executing the function **f** would be relatively small in any sizable network since there would be few nodes compared to the number of connections). Notice that the above computation can be thought of as a loop of multiply-accumulate operations. For each operation the computer must calculate two addresses, fetch the two referenced variables, multiply them together, and add that product to a local register. If the addresses were sequential, calculating a new address could be done by incrementing a register. Otherwise, a randomly connected network would require that the computer fetch a pointer which would be used as the address of O_j.

Without the extra pointer, we could imagine that the weight, W_{ij}, and O_j could be fetched in parallel from two separate memories and pushed into a pipeline where they would be multiplied and summed. Using today's electronic components, memory fetching would be the bottleneck with memory access times of 100 nanoseconds (ns) per fetch. Thus the execution rate of the fetch-multiply-store loop above would be on the order of 100 ns. If the weight and the output could not be fetched in parallel, the loop would take 200 ns. As a result the best can be hoped for, even with a custom made VLSI chip, is about 200 ns per cycle through the loop. Of course, this can be done in parallel with multiple chips. Said another way, this is 5 million connections per second per memory bank. Since economic and time constraints precluded the design of a VLSI chip, the best commercially available hardware was sought.

Transputer Architecture

A transputer[3] is a 32-bit, 10 million-instruction-per-second (MIPS), single-chip microcomputer manufactured by INMOS, Great Britain's leading semiconductor manufacturer. Transputers are designed to be components in large parallel processing systems and have hardware multitasking for sub-microsecond task switching times. Each transputer has four 10-Megabit per second, full duplex serial links. We purchased the INMOS transputer system ITEM 4000, a 40 transputer parallel processor with capabilities of 400 MIPS, 50 MFLOPS, and 10 Mbytes total local memory (256K per processor). An additional transputer plugs into an IBM personal computer (PC) advanced terminal (AT) with 2 Mbytes of memory. This transputer uses the PC AT as its I/O subsystem. Yet another transputer controls a 512x512x8 graphics board. The development system has both the C language and OCCAM, the parallel processing language of the transputer.

Decomposition of the Matrix

There are several ways to divide the nodes and connections of the network among the processors. One scheme is to copy the node variables to all processors. This makes allocation simpler, but it can use a lot of memory. We have chosen to decompose the connection matrix (fig. 3) into partitions that require only a subset of node variables.

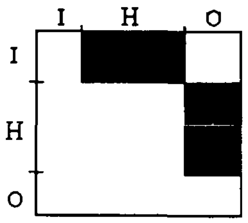

Figure 3. - Connection matrix.

The connection matrix is formed by having one row and one column for each node in the network. Assume that each row of the matrix represents a "from" node (the source of a connecting link) and each column a "to" node. For every connection, a mark is placed at the intersection of its from and to nodes. Figure 3 shows this for a three-layer feedforward network where all of the input nodes are connected to all of the hidden nodes and all of the hidden nodes are connected to all of the output nodes. The nodes and connections are then allocated to processors by sectioning the covered areas of this matrix into the same number of regions as processors. A processor associated with a region must have access to both the from nodes represented by the rows of its region and to the to nodes represented by the columns. When a node is unidirectionally transmitting a signal from one node to another, the state variables of the from node are not the same as the state variables of the to node (in a typical case, the from node must have a variable for its output value but the to node must have an accumulator for its dot product). Therefore, the state variables are separated into "exported values" and "partial results," and only the necessary variables are kept in a processor. Thus the memory allocation for copies of node variables required by a processor when receiving a region is equal to the height of the region (in number of nodes) times the number of bytes for the from node "exported" state variables plus the width of the region times the number of bytes for the to node "partial result" state variables. Let us assume that the number of bytes for from nodes is F, the number for to nodes is T, the number of nodes in the network is N and the number of processors we have is P . Let us next assume the matrix is fully covered, and that we have some number of processors that is a perfect square. If we leave enough space for every node in a processor the number of bytes allocated is $(NF + NT)P$. However, by partitioning the matrix, the number of bytes allocated is $(NF + NT)P / \sqrt{P}$ (again, assuming square regions). This is a factor of $\sqrt{P}$ savings. When the allocations are contiguous groups of nodes, node variable structures can be stored in an array with minimal overhead both in access time and memory. Other schemes would necessitate pointer or hash table overhead.

Load Balancing

A basic problem with most parallel processing schemes is balancing the load evenly so that all processors can be working most of the time. If one processor is slower than the rest, all of the other processors have to wait for it to finish before they can all synchronize and continue. Since the time of execution is proportional to the number of connections that a processor must process, our matrix decomposition scheme solves this if equal areas can be assigned to the processors. If the matrix is fully covered, this is easy. But when it is covered with many irregularly sized areas, it is more difficult. We are developing a heuristic method for doing this which we refer to as our load balancing algorithm. Because processes may be sequenced, as in the generalized delta algorithm, it is necessary to have a separate matrix for each asynchronous phase group of connections. An asynchronous phase group is defined as a set of connections where all processing can be done in parallel. In the generalized delta rule described above, there are two phase groups: input to hidden and hidden to output. The user specifies the phase when he defines the group.

Mapping Macros

To get a system up and running in a reasonable amount of time, the user is required to modify a few sections of program and recompile the source to create a network with his specifications. This code modification method offers total flexibility. The user specifies the network by calling functions. He also must specify the structures to hold the state variables of nodes and connections. To specify the equations to be applied, a "map-connections" macro is provided which expands to the actual code that goes in each of the slave processors. This macro handles all of the addressing and hands the user pointers to the connection variables and to its two adjacent node variables. The code that he provides can do whatever he wants to the state variables. It can propagate a signal forward, backward, or both ways. It could initialize the weights. It could save the weights to a file or recover them. A similar "map-nodes" macro is also provided. The macros create functions that are called with an argument of the node group or connection group to which the user wants the function applied. This macro approach expands to code that is 90 percent as run-time-efficient as can be handcoded. To execute the sequencing of the generalized delta rule the user would call a predefined function that loads the inputs. The user would then broadcast a message to all processors that would invoke his macro-defined function with an argument of the input to hidden connection group. This function executes in parallel in all of the slave processors. It first checks to be sure it has connections from the input to hidden connection group; if not it just responds "done" to the master. The user then broadcasts to invoke his function that was defined with "map nodes" to process the hidden nodes. The same is done from the hidden nodes to output nodes. Backpropagating is very similar but the user invokes different functions. He probably would name this routine "train_one_input_output_pair" and call it inside a loop to do all of his training. Likewise, the user might define a function called "output_of" that takes an input vector as argument, propagates it through the network, and returns an output vector. If the user used symbolic constants in his functions, he would only have to change constants such as Number_of_input_nodes, Number_of_hidden_nodes and Number_of_output_nodes to change their sizes. Even architectural changes can be made with small changes in the program (such as connecting all input nodes to all output nodes as well). Although this approach allows the total flexibility that many users want, others will dislike tinkering with the code. Ultimately, a much friendlier user interface will be provided as well. We are considering both a language and menu-driven graphics for specification of the network.

Program Structure

The system architecture we are using is master-slave. The master transputer in the IBM PC acts as an interpreter of the commands from the user interface. In turn, the master issues commands to the slaves whose sole task is to interpret commands from the master. The slaves merely look the command up in a table (index an array) and execute the function associated with that command (whose pointer is stored in the table). The argument to the function is a pointer to the buffer that holds the remainder of the command message which contains the arguments to the function.

Synchronization

Synchronization of the master and slave processes is accomplished by having the slaves respond to every command. When the master gets as many responses as there are processors, he can continue.

Communication

Since processors are only connected in a point to point fashion, a message between two non-adjacent processors must be relayed by intermediate processors. Our communications process inside each processor continually waits for a message to come in from any of the four input ports and, when one comes in, it reads the address in the message and looks in a table (indexes an array) to determine the appropriate output channel for retransmitting the message (a channel is a logical port; it may be a physical port or a location in memory[3]). Buffering is used in each processor to avoid deadlock and to smooth irregularities in transmission rates.

Even though it reduces the total amount of communication required, the method of decomposing the matrix means that some nodes are split across processors and that the partial results accumulated at several of the processors must be shipped to a central location (for this node) to be combined. This location is called the home processor of the node. Each processor is home for a roughly equal number of nodes that are in the same asynchronous processing phase. When a map_nodes function is called, it is applied in the home processor of the node with the variables which it contains. These are referred to as the static variables of the node. Since they do not have multiple copies, the static variables use little memory.

So the bulk of the communications results from processors sending partial results to the home nodes and from home nodes exporting these processed values to several processors. Analyzing the time this will take is very difficult, but there are two potentially limiting factors that we can analyze individually: the limit set by the serial links and the limit set by the CPU cycles required to buffer and relay messages. Surprisingly, the CPU time is the limiting factor. The analysis is as follows.

The number of processors that ship a message home and that the home node ships to can be seen by examining the sections of the connection matrix after the areas have been carved out. Let us assume that a fully covered matrix has been evenly partitioned into 5 rows by 8 columns and that each of the 40 processors gets 1 partition. From a home node processor's point of view, five processors must ship their partial results to the home node processor and this processor must combine these results and ship this new output value to eight processors (let us assume the home processor is not one of these). This means that there are 14 messages shipped for every node. It can be determined by enumeration that the average distance between processors in our array (fig. 4) is 3.5 links.

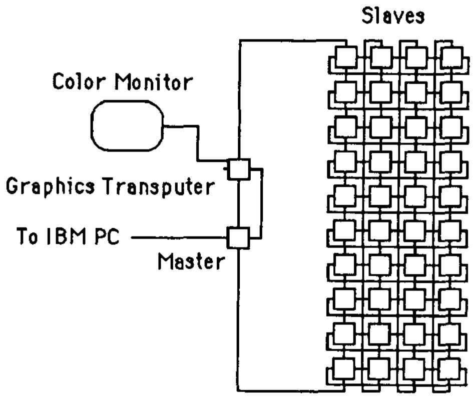

Figure 4. - Processor configuration

Each of these 14 messages must travel over 3.5 links, yielding 3.5 * (5 + 8) ≈ 46 transmissions per node. If each message is 12 bytes (96 bits) long, this is 96 * 46 = 4416 bits transmitted per node. Since each serial link is 10 million bits per second and there are 80 full duplex serial links in our network, there are 2 * 80 * 10million = 1600 million transmitted bits per second possible (This assumes that every bit slot is being used, which is not likely; but if adequate buffering is available and the CPU retransmit speed is sufficiently fast, this is almost the case (> 50 percent of this)). Dividing this by 4416 bits gives 362,318 nodes that could transmit each second. This means that the serial-link is limited to about 362,000 nodes per second. This could become the limiting factor in networks with large numbers of nodes and few connections per node; but, typically, there will only be a few thousand nodes in a network so other things will probably limit throughput before the serial links do.

To examine the CPU speed requirement, remember that there are 46 transmissions per node. For each of these, a processor must receive the message, buffer it, determine the port to which it should be sent, and then retransmit it. The transputer direct memory access (DMA) hardware handles the receiving and transmitting concurrently. All the CPU must do is copy the received message to a buffer, determine where to send it, and later copy it from the buffer. There are 46 transmissions per node, and a serial-link-limited rate of 362,000 nodes per second is 46 * 362,000 = 16,652,000 transmissions per second. When this is distributed over the 40 processors, each one must do 416,300 transmissions or one transmission every 2.4 microseconds (μs). Yet summing instruction execution times shows that the transputer will actually take about 10 μs to retransmit for a rate of 81,600 nodes per second. This is about four times slower than the serial link rate. So the speed of processing networks on this system is still CPU limited.

It takes about 2 μs to fetch operands and do a multiply accumulate on our 20 MHZ transputer. This is what must be done at each connection when propagating forward. Forty transputers can process 20 million connections per second. The time to propagate an input forward to get an output from a feedforward network can be estimated by (number_of_connections / 20 million) + (number_of_nodes / 81,600) since these two terms represent 90 percent of the processing in typical networks.

The 2 μs required for each fetch-multiply-accumulate loop is about 10 times slower than what would be possible with a handcrafted VLSI chip but is significantly faster than most other 32-bit microprocessors. This, along with the capability of packing several transputers in a very small space, makes a transputer array a very cost effective solution to neural network simulation with off the shelf components.

Conclusion

We have discussed the constraints imposed by neural networks on simulation. We have shown what is achievable in terms of memory efficiency and simulation speeds and have compared our design to this. We have discussed a technique for partitioning a neural network to minimize memory waste on a parallel machine. The program structure also was discussed. The communication network was analyzed to determine what the costs of communication are. The resulting design gives us a neural network simulator that has a performance level nearly equivalent to the highly optimized simulator we have running on the SX-2 supercomputer for a cost equivalent to 2 days of CPU time on that supercomputer.

References

1. Hopfield, J., "Neural Networks and Physical Systems with Emergent Collective Computational Capabilities", Proceedings of the National Academy of Science, USA, Vol 79, April, 1982, pp. 2554-2558.

2. Hopfield, J., "Neurons with Graded Response Have Collective Computational Properties Like Those of Two-State Neurons", Proceedings of the National Academy of Science, USA, Vol 81, May, 1984, pp. 3088-3092.

3. INMOS Transputer Reference Manual INMOS Ltd. 72-TRN-006-03, Bristol, UK 1987.

4. Grossberg, S. STUDIES OF MIND AND BRAIN: NEURAL PRINCIPALS OF LEARNING, PERCEPTION, DEVELOPMENT, COGNITION AND MOTOR CONTROL, Reidel Press, Boston, MA.

5. McClelland, J., and Rumelhart, D., PARALLEL DISTRIBUTED PROCESSING: EXPLORATIONS IN THE MICROSTRUCTURE OF COGNITION, MIT Press, Cambridge, MA, 1986.