# Modifications to Artificial Neural Networks Models for Digital Hardware Implementation

*Tom Baker and Dan Hammerstrom*

Department of Computer Science and Engineering

Oregon Graduate Center

Technical Report No. CS/E 88-035

December, 1988

# Modifications to Artificial Neural Networks Models for Digital Hardware Implementation

Tom Baker and Dan Hammerstrom[1]

Dept. of Computer Science and Engineering, Oregon Graduate Center

*Abstract—Before artificial neural network applications become common there must be specialized hardware that will allow large networks to be run in real time inexpensively. It is uncertain how large networks will do when constrained to implementations on architectures of current technology. Some tradeoffs must be made when the network models are implemented efficiently. One popular artificial neural network model is the back propagation algorithm [1]. The back propagation model promises to be a powerful and flexible learning model. This paper discusses the effects on performance when the model is modified for efficient hardware implementation. The modifications to the back propagation model includes limited precision computation, limited communication between layers, accumulation of weight changes, and the introduction of noise into the weight modification process.*

## 1. Introduction

Recent research has shown that artificial neural networks (ANNs) are a promising solution to many applications that are difficult for conventional computers. The parallel distributed processing models are a radical departure from previous artificial intelligence solutions. Instead of processing symbolic data, many ANN algorithms use massively parallel neuron-like elements to provide problem solutions. Because the ANN models are based on biologically inspired models, they promise to do tasks that are natural for humans but difficult for more traditional computers.

Although the ANN models are biologically inspired, they are not accurate simulations of real neural processes. Simulating the bio-electrical responses of neurons takes a prohibitive amount of CPU time. Instead of reverse engineering the brain, the ANN models try to achieve similar results with current computer technology.

[1]This work was supported under SRC grant 86-10-097.

The ANN research community needs architectures that implement the ANN models faster than what is currently available. Simulating ANN's on existing architectures is slow because of the large number of computations required by the models. Research is bounded by the time limits that current architectures place on ANN simulations. Researchers currently cannot simulate more than a few thousand neurons because the execution time of larger networks is too long. Many believe that the models will not be useful in real applications until large networks are implemented ( $> 10^4$ neurons). Therefor, specialized hardware must be created that will execute ANN models in real time before many useful applications can be developed.

One common ANN model is the back propagation algorithm made popular by Rumelhart, Hinton and Williams [1]. This paper discusses the effects of the back-propagation model when implemented with current technology. There are many tradeoffs that must be made when an algorithm is mapped to a hardware architecture. We describe what important concessions must be made and what the performance effects are.

## 2. Model Architecture

The goal of the Cognitive Architecture Project (CAP) at the Oregon Graduate Center is to create wafer scale neurocomputers. There are many architectural issues that must be resolved before our goal can be realized. This paper will discuss how an ANN model performs when implemented on a VLSI architecture.

It is not the purpose of the research reported here to design a computer architecture that will run ANN models. However, there must be a general target architecture to simulate. Although we only present the results of one type of ANN model, we assume a more general architecture. The target architecture is not specifically for back-propagation algorithms, but for executing ANN models in general. The model architecture must be general enough to emulate different types of ANN algorithms.

To have fast execution of the ANN models, the algorithms must be mapped to a more efficient implementation. Current technology cannot emulate fully connected communication and high precision computation in low cost and at the speeds required for real time execution. Simpler processors can be made smaller, and more processors can be implemented with a limited amount of resources.

For this paper, we assume that the target architecture consists of many simple parallel processors. The ANN network will be mapped onto the target architecture with one artificial neuron per processor node (PN). We have simulated a digital computer, but the same results would apply to an analog processor as well. Each PN must have its own local memory to store weight data. Sharing a common memory bus between PNs would not provide the performance that is required. Each of the PNs will be a simple processor that executes its functions in a pipelined fashion with one result per clock cycle. We realize that this performance is may not be realizable in real hardware, but it does give us a theoretical basis for comparison. Analysing the actual method of communication is outside the scope of this paper, knowing the exact type of inter-processor communication un-necessary for this research. Other research within the CAP group has found workable solutions to the problems of communication between processors in a massively parallel architecture [2] [3].

## 3. The Back Propagation Algorithm

The back propagation model is a feed forward network with $N$ layers ($N \geq 3$). The first layer is the input layer, and layer $N$ is the output layer. The layers between the input and output layers are called the hidden layers. Each node in layer $n$ ($1 < n \leq N$) is connected to every node in layer $n-1$. The activation value ($o_i$) for each node in layer $n$ is calculated as

$$o_i = \frac{1}{1 + e^{-\Sigma o_j w_{ij}}} \tag{1}$$

where $n$ is the number of input nodes, $o_j$ is the value of node $j$ in layer $n-1$, and $w_{ij}$ is the connection weight between the input node $j$ and node $i$.

The back-propagation model is a supervised learning algorithm that requires that the desired output is known. Learning occurs by using the weight update equation

$$\Delta w_{ij}(t) = \eta \delta_j o_i + \alpha \Delta w_{ij}(t-1) \tag{2}$$

where $\eta$ is the learning rate, $\delta$ is the error gradient, $\alpha$ is the momentum rate, and $\Delta w_{ij}(t-1)$ is the previous weight change. The error gradient for each output node ($o_i$) in layer $N$ is

$$\delta_i = (t_i - o_i)o_i(1 - o_i) \tag{3}$$

where $t_i$ is the desired value for the node. The error gradient for each hidden node in layer $n$ is

$$\delta_i = \Sigma \delta_k w_{ik} o_i(1-o_i) \tag{4}$$

where $\delta_k$ is the error gradient of the node in layer $n+1$ that is connected by the weight $w_{ik}$.

The network was trained on a character recognition application. The training set was a set of characters from eleven different alphabetical fonts taken from the Apple Macintosh font set, there were 286 total training pairs. The inputs were a twelve by twelve array that was a pixel representation of a single character. The position of the character was justified to the top left. There were twenty-six outputs, one output for each character in the alphabet. The network was expected to match the bit map input to the correct output category. There was one hidden layer with thirty hidden nodes. This application is a non-trivial problem that tests the networks' learning capacity and generalization abilities. We feel that it is a good problem to test the performance of the modifications that will be made to the algorithm. We have simulated other applications with similar results.

## 4. Modifications

The back propagation model was not designed for efficient hardware implementation. The current mathematical model is not concerned with physical realization. Many simulations use double precision calculations that are expensive to implement inexpensively in a massively parallel architecture. Architectures that use limited precision calculations can be made smaller and faster, and they can be made using analog computation elements. The algorithm also is not concerned about the problem of how many processors would communicate asynchronously with each other. Too much communication between processors causes a bottleneck in the performance of any parallel architecture.

The research that this paper describes answers the question of what the effect that certain modifications have to the performance of the back propagation algorithm. These modifications are inspired by our desire for inexpensive silicon implementations. We discuss some of the areas that prevent the model from being implemented in hardware. We have also presented some possible solutions to the problems that limit fast execution. For each possible solution, a simulation has been made to determine the effects on the performance and effectiveness of the resulting ANN. From the simulation results, decisions can be made whether the solutions are effective or not.

There are many tradeoffs that must be made when deviating from the theoretical basis of the ANN models. The important concept is what the effect is on network performance when modifications are made. The learning algorithm sometimes ceases to work effectively, and understanding the reasons for failure can be important when designing hardware. The results of our modifications to the back propagation algorithm are not always completely successful, but there is often an improvement in one aspect of the execution of the algorithm.

### 4.1. Limited Precision

Floating point processing units are too expensive and require more silicon area than is feasible for a highly parallel architecture. For a cost effective solution, the PNs should have an integer multiplier and adder. As we mentioned before, the PNs could also be an analog processor. The important issue is how much information the back propagation algorithm requires for the activation values and weights.

Fixed point computation can be used for limited precision architectures. The activation value of equation 1 can be computed by summing the products of the inputs $o_j$ and the weights $w_{ij}$, and using a table lookup for the squishing function. By using a table lookup, the binary point of the results can be automatically justified. Because the activation value of each node is always positive and less than one, the binary point will always be to the right of the significant bits. Our simulations have shown that eight bits of precision are enough for the activation values of the outputs.

The weights however, commonly grow larger than one and can have both positive and negative values. So the binary point for the weights must be placed within the significant bits, and there must be one bit to represent the sign of the value. We use three bits to the left of the binary point, which limits the weight values to less than eight. The weights of floating point simulations sometimes grow larger than eight, but weights greater than eight are not required. The simulation results of limited precision calculations are shown in Table 1. Each number in Table 1 is the average number of input presentations for the application to converge for five identical networks with different initial weights. A floating point simulation is given for comparison.

The limited precision simulation results show that sixteen total bits of precision (one sign bit, three bits to the left of the binary point and twelve bits to the right) are adequate. The network can learn with less precision, but the learning algorithm does not work as effectively. Our simulations have shown that there is a limit of twelve bits of weight precision required by the algorithm. The twelve bit limit is because the individual weight updates in equation 2 are small quantities. The algorithm requires weight updates in the bit range greater than eight bits to the right of the binary point ($|\Delta w_{ij}| < 0.004$), and elimination of these bits would inhibit learning. A floating point representation may be able to learn successfully with fewer than twelve bits in the mantissa, but floating point units are too expensive for massively parallel architectures.

The limited precision network required fewer input presentations to learn the training set. We feel that the improvement is caused by the fact that the weights are limited to a maximum value ($|w_{ij}| \leq 8$). Large weight values can cause the network to learn more slowly, because a single large weight may have

Floating Point vs. Integer

| Algorithm Modification | Input Presentations |
|---|---|
| Floating Point | 36,490 |
| 16 Bit Integer | 21,340 |

Table 1

too much influence over the result of the activation value of the node. It is unclear what the optimal upper bound for weight values should be for all applications. We have found that three bits to the left of the binary point is adequate for this application. We feel that the limited precision results are application dependent, and will not always learn with fewer input presentations.

## 4.2. Sign/Threshold Propagation

The interprocessor communication required to execute equation 4 is a serious problem. The information that must be sent between processors in all the other equations can be broadcast ($O(n)$ communication) from the originating node to all of the other nodes. The hidden nodes, however, must receive a unique weight value from each of the output nodes requiring point to point communication ($O(n^2)$ communication). Anderson [4] discovered that the network learning performance increased when the sign of the weight was used in equation 4 instead of the actual weight value. Using Anderson's results leads to a useful modification of the basic algorithm. If the sign of the weights for the connections between the hidden nodes and the output nodes are kept in the local memory of the hidden nodes, as well as the output nodes, then the value only has to be propagated when the sign of the weight changes. The output nodes can then broadcast their $\delta$ values, and send a point to point message when the sign of a weight changes. The communication overhead reduces dramatically when this modification is used. Table 2 shows that there is a performance increase when the propagation of weights is reduced. We have made an estimate of the total clock cycles that the target architecture requires to learn the application. Even though the network takes more input presentations to learn the data, there is less execution time.

As the simulation results in table 2 suggest, the network does not always learn with fewer input presentations with Anderson's modification. The algorithm seems to require more weight resolution than simply using the sign of the weight. If we use the entire value of the weight and propagate the weight only when it changes by a certain limit, then we can use greater weight precision

Propagation of Sign of the Weights

| Algorithm Modification | Input Presentations | Execution Cycles |
|---|---|---|
| Integer Standard | 21,340 | 60.82M |
| Use Sign of Weight | 27,510 | 56.26M |

Table 2

and also use reduced communication. Table 3 shows the results of simulations that propagate the weights only when the weight differed by a certain threshold from the weight that is stored in the hidden nodes' local memory. A threshold of 0.1 has a significant increase in performance, while using a threshold of 0.3 has reduced performance. The performance increase of using reduced weight propagation is application specific, and may not be beneficial in all circumstances. The additional cost of requiring more local memory for the hidden nodes must also be considered. The OGC CAP group is doing further research to reduce the communication for propagation of error values.

### 4.3. Sum Weight Changes

Accumulating the weight changes of equation 2 before updating the weights can have a positive effect on the number of input presentations that are required for the network to learn. Adjusting the weights for every input presentation can create a discontinuous path through the error surface. When the weight changes are accumulated, the path of the gradient descent algorithm tends to be smoother. The weight change accumulation is similar to the data partitioning method used by Pomerleau, et. al. [5] when they optimized the back propagation algorithm for execution on the Warp system. Weight change accumulation can also improve the execution time of the algorithm. The only value that needs to be accumulated from equation 2 is the $\delta_i o_j$ term. Multiplying the learning constant ($\eta$) and adding the momentum ($\alpha \Delta w_{ij}$) can be delayed until the weights are actually updated. The communication of error back propagation will also be reduced because the $\delta_i$ values in equation 4 are not sent on every input presentation. The simulation results in Table 4 suggest that the number of input presentations needed to learn the training set are not always reduced when accumulating the weight changes, but there is a consistent improvement in execution time.

Propagation of Weight After Threshold Change

| Algorithm Modification | Input Presentations | Execution Cycles |
|---|---|---|
| Integer Standard | 21,340 | 60.82M |
| Threshold 0.1 | 21,050 | 43.06M |
| Threshold 0.3 | 38,780 | 79.31M |

Table 3

Accumulated Weight Change

| Algorithm Modification | Input Presentations | Execution Cycles |
|---|---|---|
| Integer Standard | 21,340 | 60.82M |
| Accumulate 2 Inputs | 22,650 | 39.26M |
| Accumulate 3 Inputs | 18,480 | 25.14M |
| Accumulate 4 Inputs | 20,190 | 23.71M |
| Accumulate 5 Inputs | 25,340 | 26.93M |
| Accumulate 6 Inputs | 18,670 | 18.54M |
| Accumulate 7 Inputs | 21,790 | 20.38M |
| Accumulate 8 Inputs | 21,790 | 19.51M |
| Accumulate 9 Inputs | 24,250 | 20.96M |
| Accumulate 10 Inputs | 24,540 | 20.75M |

Table 4

### 4.4. Noise

An effect of using integer computation is that the weight space is quantized. The weights can only be adjusted in discrete increments. It is possible for a weight to get stuck on one value if the weight changes are not big enough to bypass local minima. Adding random noise to the weights can help smooth the weight space. We added noise to the weight space by randomly setting the lower $n$ bits of each weight. It is counter-intuitive to think that any computation can be improved by noise, but the simulation results in Table 5 suggest that the number of input presentations can be reduced with the addition of noise. The extra computation of adding the noise does increase the execution time of the algorithm, even though the network converges in fewer input presentations. It is possible, however, to design hardware that would introduce noise into the weight space with no cost of execution time.

## 5. Conclusions

The results in this paper should be considered cautiously. The performance improvement (or degradation) is somewhat dependent on the training set, and the algorithm modifications may not have the same results over all applications. The target architecture may also have been over-simplified, and the estimate of the execution time of the algorithm may not be accurate. There are many trade-offs that must be considered when the back-propagation algorithm is modified. The results in this paper should help a computer architect realize the effects of the algorithm modifications when designing hardware for

Effect of Noise on Learning.

| Algorithm Modification | Input Presentations | Execution Cycles |
|---|---|---|
| Integer Standard | 21,340 | 60.82M |
| 1 Random Bit | 18,250 | 66.72M |
| 2 Random Bits | 17,330 | 63.38M |
| 3 Random Bits | 18,880 | 69.02M |
| 4 Random Bits | 25,800 | 94.34M |

Table 5

ANN applications.

We have had similar results with other applications, such as a network that learned the coordinate transformation for a two axis robot arm. The desired position was input, and the axis positions were expected as an output. As mentioned in the previous paragraph the results are application specific, but the effects of the algorithm modifications were consistent. More research must be done to determine the effects of the algorithm modifications when large networks are used.

Our results show that:

- Reduced precision computation can be used successfully for the back-propagation algorithm.
- The communication between processors can be reduced when propagating the weights.
- Accumulating the weight changes can improve execution time of the algorithm.
- Noise can have a positive effect on the learning algorithm.

## 6. References

[1] Rumelhart, Hinton, Williams (1986). Learning Internal Representations by Error Propagation. In D.E. Rumelhart & J.L. McClelland's (Eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition (Vol. 1)* (pp. 318-362). Cambridge, Mass.: MIT Press.

[2] Bailey, Hammerstrom (1988). Why VLSI Implementations of Associative VLCNs Require Connection Multiplexing. *International Conference on Neural Networks.* San Diego, CA.

[3] Bahr, Bailey, Baker, Hammerstrom, Jagla, Johnson, Mates, May, McCartor, Means, Rudnick (1988). The OGC Cognitive Architecture Project: Silicon Implementation of Connectionist/Neural Networks. *NorthCon 88.* Seattle, WA.

[4] CA Anderson, Ph.D. Dissertation, *Learning and Problem Solving with Multilayer Connectionist Systems,* Amherst, MA, Sept. 1986.

[5] Pomerleau, Gusciora, Touretsky and Kung (1988). Neural Network Simulation at Warp Speed: How We Got 17 Million Connections Per Second. *International Conference on Neural Networks.* San Diego, CA.