A Backpropagation Neural Network Design Using Adder-Only Arithmetic

Vicente Mahoney, *Student Member IEEE*, Itamar Elhanany, *Senior Member IEEE*
Department of Electrical Engineering & Computer Science
The University of Tennessee
Knoxville, TN 37996-2100

***Abstract*— Feedforward and recurrent neural networks have enjoyed great popularity in the field of machine learning. Hardware implementation of these networks, often using a backpropagation algorithm for learning, has many advantages over software implementation, the main advantage being speedup due to taking advantage of the inherent parallelism of neural networks. However, tradeoffs are often introduced with respect to speed, area, precision, and other factors. Moreover, many of the calculations involve multiplications, which can be costly and complex to implement in custom hardware. This paper proposes a piecewise linear approximation architecture that can be used to replace multiplications with a series of shifts and additions. This, combined with the same approximation for the nonlinear activation function and its derivative, results in a neural network realization in which all arithmetic and function computations are carried out using adders only.**

I. Introduction

For several decades, artificial neural networks (ANNs) have been extensively studied with particular applications to the field of machine learning. Designed to mimic some of the inherent properties of the mammal nervous system, their widespread appeal is in their adaptivity, parallelizability and ability to model many different kinds of systems, including highly nonlinear ones. One of the most popular and perhaps simplest structures for implementing ANNs is the feedforward neural network, where information passes between multiple layers of neurons in only one direction. Recurrent neural networks take this concept one step further and allow for information to pass in the other direction, creating feedback that can improve learning in dynamic settings. With both types of neural networks, the most popular method of learning is through backpropagation, where error between a target output and the output from the network is propagated back through the network, as means of updating its parameters (or weights).

ANNs have been implemented extensively both in hardware and in software, each offering advantages and disadvantages. Hardware implementation offers the advantage of exploiting the inherent parallelism of neural networks. ANNs can be considered as large collections of multiplications, additions, and nonlinear functions, and because for the most part the calculations are independent of each other, they are prime candidates for being done in parallel on a device such as a field programmable gate array (FPGA). However, while speed may be gained, hardware has its own limitations. Silicon area is not unbounded on an FPGA, and thus the architecture of the implementation becomes crucial. Trade-offs must often be made between parameters such as area and precision [1].

Although digital hardware implementation of feedforward neural networks has been studied for decades, fundamentally they have involved combinations of additions, multiplications, and nonlinear activation functions. Because multiplications and activation functions are slower and more complex to implement than additions, they can act as bottlenecks. Various techniques for implementing activation functions have been discussed, with some [2], [3] utilizing a piecewise linear (PWL) approximation using powers of two as slope values. Powers of two are used so that approximations can be done using only shifts and adds, avoiding costly multiplications. However, multiplications are still present in calculating the activations at each neuron and in the backpropagation algorithm.

In this paper, we propose a digital hardware architecture that eliminates the need for multiplications altogether, allowing all operations to be carried out using shifts and additions. The design takes advantage of one of the rules of logarithms, which states that $xy = a^{\log_a x + \log_a y}$. If e is used as the base, then a multiplication can be replaced by three function executions; two of $\ln x$ and one of e^x. Although using look-up tables would be the quickest and most accurate methods of calculating these two functions, it has been shown [4] that for fixed point numbers, a minimum range-precision of 16 bits must be used for a typical backpropagation algorithm to converge, and thusly, look-up table methods for even the minimum precision would require too many entries (2^{16}) per function to implement. Therefore, a PWL method using powers of two as slopes can be used to approximate the natural logarithm and exponential functions. These, in addition to similar PWL approximations of the activation function and its derivative, allow for a design that uses only shifts and additions for arithmetic calculations. Although some precision is sacrificed in the approximation, substantial speed and logic volume gains are attained.

II. Piecewise Linear Approximating Architecture

The PWL implementation used in this design is based on the one used in the digital companding architecture discussed in [5], and is depicted in Figure 1. Each function range is divided into S segments which take the form of $k_i x + c_i$, where $i = 1, 2, ..., S$, k_i are powers of two or combinations

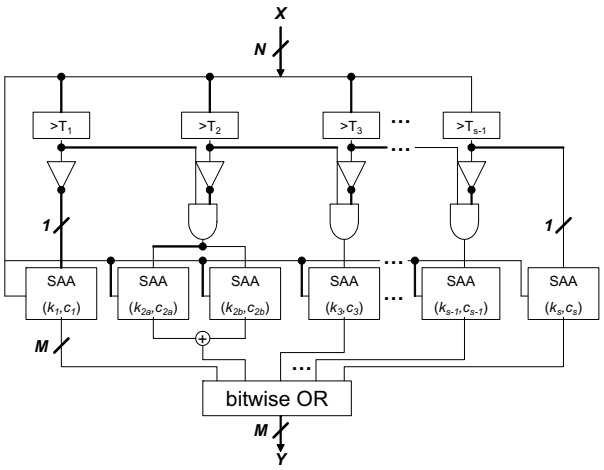

Fig. 1. The architecture of the PWL approximator. The second segment requires a combination of two slopes, such that two shift-and-add (SAA) operations are summed together.

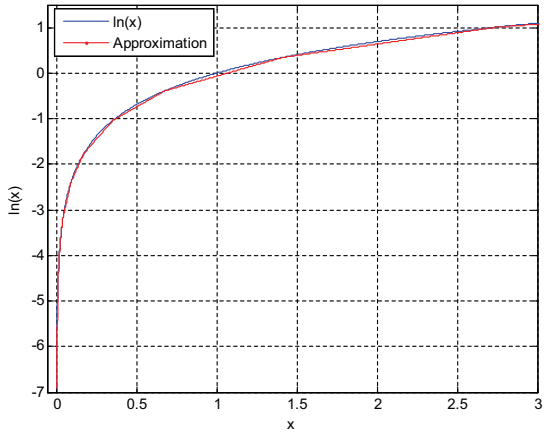

Fig. 2. Comparison of the natural logarithm function and its PWL approximation. The range is limited to the likely range of input values.

of powers of two, and c_i are constants. Therefore, each function calculation involves one shift and one addition. The approximating algorithm takes the input x and determines which section it fits in. This is accomplished in hardware by using $S-1$ separate comparator circuits, each with a predetermined threshold T_i corresponding to a section in the PWL architecture, where $i = 1, 2, ..., S-1$. The input x is passed through each of the circuits in parallel, and the comparison $x > T_i$ is made. The output from the comparator is passed through an inverter which is combined with the output of the preceding circuit and fed through an AND gate, except for the first and last segments. The structure of the logic section ensures that at most one signal from the series of gates will be set to '1', while the others are all set to '0'.

These signals feed into the next stage of the architecture, where they act as enable bits for a series of shift-and-add (SAA) blocks. Each block corresponds to a segment of the PWL approximation, and contains its related slope, k, and constant, c. The input x is passed to the SAA blocks, where they all perform the one shift and one add, using k and c. These operations can be done in parallel, as they are all identical and independent. Because only one of the enable signals is set, the results can all be passed through a bitwise OR to produce only the proper value. In the case that a section's slope is defined by a combination of powers of two, the result is obtained by $k_{i1}x + k_{i2}x + \ c_i$, and thus requires an extra shift and addition. The results of both SAA blocks are summed together. This causes an extra delay in producing the final output; however, the delay of a single extra adder is small relative to that introduced by a multiplier circuit.

III. Efficient Backpropagation Implementation

A version of the Elman network, first described in [6], is used for the design as an example of a basic recurrent neural network. While not as effective as other recurrent network designs, the Elman net is simple in structure and serves as a good test of the effects of approximation. The design is a standard feedforward network with one hidden layer, with the outputs of the hidden layer being copied to an additional hidden layer and delayed by a single time step. These delay units are called context units. The context units are then fed back to the input layer and treated as normal inputs to the network. The feedforward nature of the network is unchanged, and therefore backpropagation can be used to train the network.

The four functions to be approximated are the natural log, exponential, the activation function, and its derivative. No one activation function is standard; methods of approximation described in [2] and [3] use the sigmoid function, but the design discussed in this paper uses the hyperbolic tangent function ($\tanh x$). The derivative of the hyperbolic tangent is $1 - \tanh^2 x = \operatorname{sech}^2 x$, where $\operatorname{sech} x$ is the hyperbolic secant function. Although the derivative of the hyperbolic tangent can be calculated from the hyperbolic tangent itself, the calculation involves squaring the result and therefore a separate PWL approximation must be made.

Sections in the PWL approximation for each function were determined by fitting lines with slopes equal to powers of two to the graphs of the functions. Ranges for the functions were limited to those appropriate in the context of each function. The inputs to the $\ln x$ function are the same as the inputs to each multiplication, which for the learning applications used in this paper are generally between -3 and 3. Therefore the sections were concentrated to approximate numbers in this range. The inputs to the e^x function are the outputs from the $\ln x$ function, which are generally numbers less than 0, so segments were constructed to closely approximate that range. For $|x| > 4$, $\tanh x \approx 1$ and so approximation is focused on the range $|x| < 4$. Its derivative has a similar dynamic range.

Because the nonlinear activation function acts as a squashing function and is only involved in activations for hidden layer neurons, it can be more coarsely approximated while not

sacrificing much accuracy in the overall algorithm. However, because multiplications are more central to the overall algorithm, the $\ln x$ and e^x functions require more accuracy, and therefore more PWL segments. Specifically, the exponential function requires a great deal of accuracy, since the multiplication operation is replaced by $e^{\ln x + \ln y}$, resulting in the exponential becoming an approximation of two approximations. The degree of accuracy required for each function had to be determined using trial and error, because it was found that with rough approximations, the learning algorithm does not converge. The final natural log approximation contains 10 sections, the hyperbolic tangent and its derivative contain 6 each, and the exponential approximation contains 20 sections, with 5 of them having slopes corresponding to combinations of 2 powers of two. The comparison of the natural log to its approximation is shown in Figure 2. Having extra segments does not add extra steps to the computation of each function though, given that the comparisons are all done in parallel, such that only storage and gate count are sacrificed.

IV. Computational Complexity

As a result of the approximations, $\ln x$, $\tanh x$ and $\operatorname{sech} x$ each require one shift and one addition, while e^x requires at most two shifts and two additions. Therefore, each multiplication can be replaced by at most four additions and four shifts. To determine the amount of adders needed per neuron, given a feedforward network with one hidden layer, it is easiest to look at each stage of the algorithm. For the feedforward stage, assuming n inputs, each neuron in the original algorithm requires n multiplications, one for each weight-input combination, $n-1$ additions to sum them together, and one addition each to compute the activation function and its derivative. Therefore, the approximated algorithm requires at most $4n + (n-1) + 2 = 5n + 1$ additions to compute the activation of each neuron. This can be accomplished in VLSI by using a separate set of adders for each input activation and computing them in parallel, or by using a single set of adders and computing the activations sequentially, which could remove the n factor and cut the required adders to 6. This method may be preferable if chip area is at a premium, since with today's technology, additions can be performed very quickly.

The second stage of the algorithm, in which back propagation of the error signal takes place, requires 1 multiplication for the output connection and n multiplications for the input connections. Therefore, each neuron requires at most $4n + 1$ additions for this stage. As with the feedforward stage, a single set of adders can be used to perform all the additions for the input connections, cutting down the required number of adders to 5. The third stage of the algorithm, updating the weights, requires n multiplications, one for each input, to calculate weight changes for each connection, assuming that the learning rate, α, is set to a power of two which suggests that multiplying by α involves only a shift operation. Using approximation reduces the requirement to $4n$ additions that can be performed in parallel or sequentially. In all, the algorithm involves $13n + 2$ additions per neuron, whereas the original algorithm requires $3n + 1$ multiplications and $n - 1$ additions per neuron.

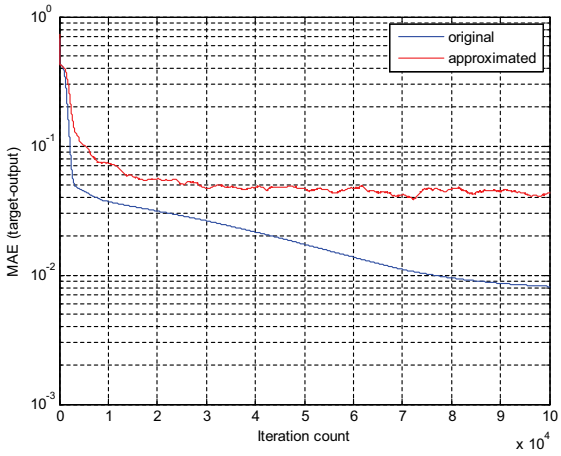

Fig. 3. Plot of MAE vs. iteration count for the feedforward network. The MAE value for each iteration was obtained from an average of 16 data points along the function to be learned.

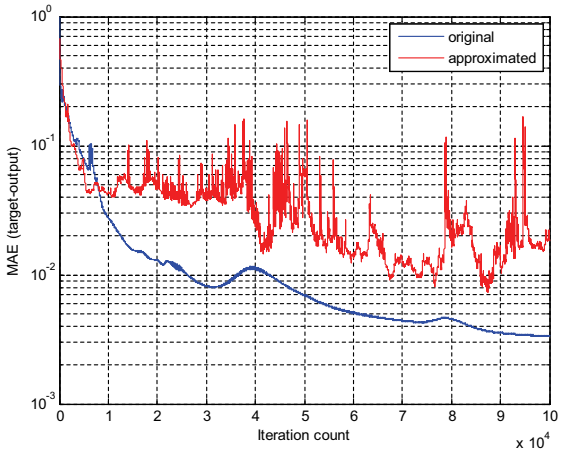

Fig. 4. Plot of MAE vs. iteration count for the recurrent network. The MAE value for each iteration was obtained from an average of 16 data points along the function to be learned.

V. Simulation Results

In order to evaluate the effects of the approximations, the algorithm was simulated using MATLAB. Two separate network structures were analyzed - one using recurrency and the other being a pure feedforward network. Both networks were constructed to have one input, one output, and one hidden layer with 12 neurons. The recurrent network used context units in the style of the Elman net, with the outputs of four neurons from the hidden layer being fed back to the input layer during the next time step, to introduce context of state.

Two different tasks were given to test the learning capabilities of the networks. The feedforward network was given 16

points of a linear input x while the output target was set as $\sin x$. The recurrent network was given the task of a frequency doubler, with the input being 16 points from the function $\sin x$, and the target output value being $\sin 2x$. For each network, 10 simulations of 10^4 iterations each were run without any approximations, with an iteration being the complete cycle of feedforward, backpropagation, and weight update steps for each of the 16 input values. The metric used to measure the performance of the networks is the mean absolute error (MAE), which is given by $MAE = \frac{1}{n}\sum_{i=1}^{n}|f_i - y_i|$, where f_i is the target output value, y_i is the actual output value, and n is 16 in this case. The MAE was measured once following each iteration.

To implement the approximations, separate functions were created, each containing slopes, constants, and range values for the segments to be used in the PWL approximating technique discussed in the previous sections. Slopes were kept at powers of two or combinations of powers of two. Multiplications and activation functions in the network algorithm were then replaced, and 10 more simulations of 10^4 iterations each were run, keeping everything else the same as in the original algorithm.

The plots of the performance for the approximated and original algorithms for the feedforward and recurrent networks are shown in Figures 3 and 4, respectively. As expected, the original algorithm has less error than the approximated algorithm; however, the error difference is less than an order of magnitude, and both algorithms converge at similar rates. The graphs of the actual output response of the networks are shown in Figure 5. They show that the output of the approximated network is very close to that of the original. One other result of note is that for the recurrent network, the approximated algorithm is overall less stable. This is likely be a consequence of feedback error introduces as a result of the approximation error. However, the algorithm does appear to converge in the long run, with the error being not far from that of the original algorithm, as evidenced by the graphs of the output responses.

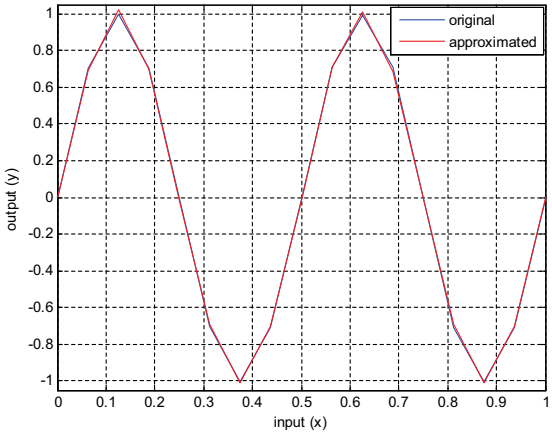

Fig. 5. Plots of the outputs of both the original and approximated recurrent neural networks. The input is $\sin x$, while the output is $\sin 2x$. Output values are the average of 10 runs for each network.

VI. Conclusions

This paper presents an easily scalable, efficient architecture for implementing backpropagation neural networks using adders only arithmetic. By using PWL approximation, multiplications can be reduced to shift-and-add operations, which offers increased flexibility in implementation and more control in the balance between speed, area, and precision. Performance has been shown through software simulations that learning algorithms still converge, and although error is greater than that of non-approximated algorithms, an architecture using only adders can run at high speeds while consuming moderate logic resources.

References

[1] M. Moussa, S. Areibi, and K. Nichols, "On the arithmetic precision for implementing back-propagation networks on FPGA: a case study," in *FPGA Implementations of Neural Networks*, A. R. Omondi and J.C. Rajapakse, Eds. The Netherlands: Springer, 2006. pp. 37-61.

[2] P. Murtagh and A. C. Tsoi, "Implementation issues of sigmoid function and its derivative for VLSI digital neural networks," *IEE Proceedings-E,* vol. 139, no. 3, May, pp. 207-214, 1992.

[3] K. Basterretxea, J. M. Tarela, and I. del Campo, "Approximation of sigmoid function and the derivative for hardware implementation of artificial neurons," *IEE Proc.-Circuits Devices Syst.*, vol. 151, no. 1, February, pp. 18-24, 2003.

[4] J. L. Holt and T. E. Baker, "Back propagation simulations using limited precision calculations," in *Proceedings, International Joint Conference on Neural Networks (IJCNN-91),* vol. 2, pp. 121-126, 1991

[5] O. Arazi and I. Elhanany, "A scalable architecture for high-speed digital companding," in *IEEE 48th Midwest Symposium on Circuits and Systems*, August 2005

[6] J. L. Elman, "Finding structure in time," *Cognitive Science*, no. 14, pp. 179-211, 1990.