Implementation of Neural Network on Parameterized FPGA

Alexander Gomperts
Student, Technical University of Eindhoven, The Netherlands
E-mail: alexgomp@gmail.com

Abhisek Ukil and **Franz Zurfluh**
ABB Corporate Research, Baden-Daettwil, Switzerland
E-mail: (abhisek.ukil, franz.zurfluh)@ch.abb.com

Abstract

Artificial neural networks (ANNs, or simply NNs) are inspired by biological nervous systems and consist of simple processing units (artificial neurons) that are interconnected by weighted connections. Neural networks can be "trained" to solve problems that are difficult to solve by conventional computer algorithms. This paper presents the development and implementation of a generalized back-propagation multi-layer perceptron (MLP) neural network architecture described in very high speed hardware description language (VHDL). The development of hardware platforms has been complicated by the high hardware cost and quantity of the arithmetic operations required in an online MLP, i.e., one used to solve real-time problems. The challenge is thus to find an architecture that minimizes hardware costs while maximizing performance, accuracy, and parameterization. The paper describes herein a platform that offers a high degree of parameterization while maintaining performance comparable to other hardware based MLP implementations.

Introduction

Artificial neural networks (ANNs) present an unconventional computational model characterized by densely interconnected simple adaptive nodes. From this model stem several desirable traits uncommon in traditional computational models; most notably, an ANN's ability to learn and generalize upon being provided examples. Given these traits, an ANN is well suited for a range of problems that are challenging for other computational models like pattern recognition, prediction, or optimization (Basheer and Hajmeer 2000; Paliwal and Kumar 2008; Widrow, Rumelhart, and Lehr 1994).

An ANN's ability to learn and solve problems relies in part on the structural characteristics of that network. Those characteristics include the number of layers in a network, the number of neurons per layer, and the activation functions of those neurons, among others. There remains a lack of a reliable means for determining the optimal set of network characteristics for a given application. Lacking any defined heuristic, the potential for fast prototyping to enable the search for an optimal network setup becomes an important consideration for any given platform.

Numerous implementations of ANNs already exist (Schrauwen et al. 2008; Mead and Mahowald 1988; Theeten et al. 1990; Liu and Liang 2005), but most of them in software on sequential processors (Paliwal and Kumar 2008). Software implementations can be quickly constructed, adapted, and tested for a wide range of applications. However, in some cases the use of hardware architectures matching the parallel structure of ANNs is desirable to optimize performance or reduce the cost of the implementation, particularly for applications demanding high performance (Ienne, Cornu, and Kuhn 1996; Omondi and Rajapakse 2002). Unfortunately, hardware platforms suffer from several unique disadvantages such as difficulties in achieving high data precision with relation to hardware cost, the high hardware cost of the necessary calculations, and the inflexibility of the platform as compared to software.

In our work we aimed to address some of these disadvantages by developing and implementing a field programmable gate array (FPGA) based architecture of a parameterized online neural network. Exploiting the reconfigurability of FPGAs, we are able to perform fast prototyping of hardware based ANNs find optimal application specific configurations.

Background Information

Previous Work

Many ANNs have already been implemented on FPGAs. The vast majority are static implementations for specific offline applications. In these cases the purpose of using an FPGA is generally to gain performance advantages through dedicated hardware and parallelism. Far fewer are examples of FPGA based ANNs that make use of the reconfigurability of FPGAs.

FAST (Flexible Adaptable Size Topology) (Sanchez 1996) is an FPGA based ANN that utilizes run-time reconfiguration to dynamically change its size. In this way FAST is able to skirt the problem of determining a valid network topology for the given application *a priori*. Run-time reconfiguration is achieved by initially mapping all possible connections and components on the FPGA, then only activating the necessary connections and components once they are needed. FAST is an adaptation of a Kohonen type neural

network and so has a significantly different architecture than our multi-layer perceptron (MLP) network. The result is a network with the ability to organize itself based correlations in the input, and thereby reveal those correlations.

Izeboudjen *et al.* presented an implementation of an FPGA based MLP with back-propagation in (Izeboudjen et al. 2007). Like ours, their design is flexible, allowing for the adjustment of network data width and precision as well as the possibility to copy and paste or remove neurons to generate new network architectures. Some important differences exist in terms of which network characteristics are parameterized as well as differences in the implementation approach.

Platform

Our development platform is the Xilinx Virtex-5 SX50T FPGA (Xilinx 2007). While our design is not directed exclusively at this platform and is designed to be portable across multiple FPGA platforms, we will mention some of the characteristics of the Virtex-5 important to the design and performance of our system.

This model of the Virtex-5 contains 4,080 configurable logic blocks (CLBs), the basic logical units in Xilinx FPGAs. Each CLB holds 8 logic function generators (in lookup tables), 8 storage elements, a number of multiplexers, and carry logic. Relative to the time in which this paper is written, this is considered a large FPGA; large enough to test a range of online neural networks of varying size, and likely too large and costly to be considered for most commercial applications.

Arithmetic is handled using CLBs containing DSP48E slices. Of particular note is that a single DSP48E slice can be used to implement one of two of the most common and costly operations in ANNs: either two's complement multiplication or a single multiply-accumulate (MACC) stage. Our model of the Virtex-5 holds 288 DSP48E slices.

Artificial Neural Networks

Artificial neural networks are characterized by their densely interconnected neurons, which are implemented as simple adaptive processing elements (PEs). MLPs (Fig. 1) are layered fully connected feed-forward networks. That is, all PEs (Fig. 2) in two consecutive layers are connected to one another in the forward direction. Data is presented via the input layer through which the current input vector enters the network but no computation is made. The input layer is followed by any number of hidden layers, followed by an output layer, the output of which is the output of the network.

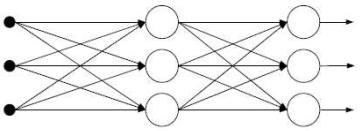

Figure 1: Multi-layer Perceptron model.

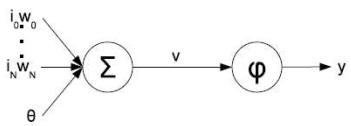

Figure 2: Processing element.

During the network's forward pass each PE computes its output y_k from the input $\vec{i}_k$ it receives from each PE in the preceding layer as shown here:

$$y_k = \varphi_k \sum_j w_{kj} i_{kj} + \theta_k \quad (1)$$

where φ_k is the squashing function of PE k whose role is to constrain the value of the local field,

$$v_k = \sum_j w_{kj} i_{kj} + \theta_k \quad (2)$$

w_{kj} is the weight of the synapse connecting neuron k to neuron j in the previous layer, and θ_k is the bias of neuron k. Equation 1 is computed sequentially by layer from the first hidden layer which receives its input from the input layer to the output layer, producing one output vector corresponding to one input vector.

Back-Propagation Algorithm

The back-propagation learning algorithm (Rumelhart, Hinton, and Williams 1986) allows us to compute the error of a network at the output then propagate that error backwards to the hidden layers of the network adjusting the weights of the neurons responsible for the error. The network uses the error to adjust the weights in an effort to let the output y_j approach the desired output d_j.

Back-propagation minimizes the overall network error by calculating an error gradient for each neuron from which a weight change $\triangle w_{ji}$ is computed for each synapse of the neuron. The error gradient is then recalculated and propagated backwards to the previous layer until weight changes have been calculated for all layers from output to the first hidden layer.

The weight correction for a synaptic weight connecting neuron *i* to neuron *j* mandated by back-propagation is defined by the delta rule:

$$\Delta w_{ji} = \eta \delta_j y_i \quad (3)$$

where η is the learning rate parameter, δ_j is the local gradient of neuron *j*, and y_i is the output of neuron *i* in the previous layer.

Calculation of the error gradient can be divided into two cases: for neurons in the output layer and for neurons in the hidden layers. This is an important distinction because we must be careful to account for the effect that changing the output of one neuron will have on subsequent neurons. For output neurons the standard definition of the local gradient applies.

$$\delta_j = e_j \varphi_j{}'(v_j) \quad (4)$$

For neurons in a hidden layer we must account for the local gradients already computed for neurons in the following layers up to the output layer. The new term will replace the calculated error *e* since, because hidden neurons are not visible from outside of the network, it is impossible to calculate an error for them. So, we add a term that accounts for the previously calculated local gradients:

$$\delta_j = \varphi_j{}'(v_j) \sum_k \delta_k w_{kj} \tag{5}$$

where *j* is the hidden neuron whose new weight we are calculating, and *k* is an index for each neuron in the next layer connected to *j*.

As we can see from (4) and (5), we are required to differentiate the activation function φ_j with respect to its own argument, the induced local field v_j. In order for this to be possible, the activation function must of course be differentiable. This means that we cannot use non-continuous activation functions in a back-propagation based network. Two continuous, nonlinear activation functions commonly used in back-propagation networks are the logistic function:

$$\varphi(v_j) = \frac{1}{1 + e^{-av_j}} \tag{6}$$

and the hyperbolic tangent function:

$$\varphi(v_j) = \frac{e^{av_j} - e^{-av_j}}{e^{av_j} + e^{-av_j}} \tag{7}$$

Training is performed multiple times over all input vectors in the training set. Weights may be updated incrementally after each input vector is presented or cumulatively after the training set in its entirety has been presented (1 training epoch). This second approach, called batch learning, is an optimization of the back-propagation algorithm designed to improve convergence by preventing individual input vectors from causing the computed error gradient to proceed in the incorrect direction.

Hardware Implementation

Our design approach is characterized by the separation of simple modular functional components and more complex intelligent control oriented components. The functional units consist of signal processing operations (e.g. multipliers, adders, squashing function realizations, etc.) and storage components (e.g. RAM containing weights values, input buffers, etc.). Control components consist of state machines generated to match the needs of the network as configured. During design elaboration, functional components matching the provided parameters are automatically generated and connected, and the state machines of control components are tuned to match the given architecture.

Network components are generated in a top-down hierarchical fashion as shown in Fig. 3. Each parent is responsible for generating its children to match the parameters entered by the user prior to elaboration and synthesis.

Data Representation

Network data is represented using a signed fixed point notation. This is implemented in VHDL with the IEEE proposed fixed point package (Bishop). Fixed point notation

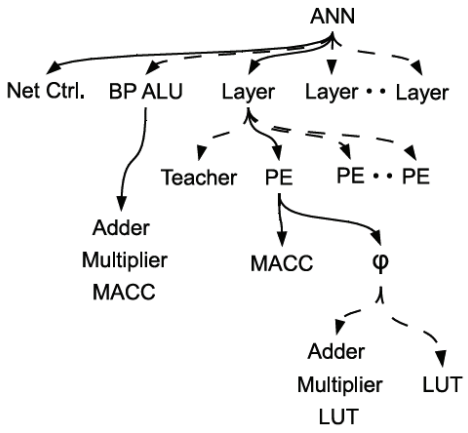

Figure 3: Block view of the hardware architecture. Solid arrows show which components are always generated. Dashed arrows show components that may or may not be generated depending on the given parameters.

serves as a compromise between traditional integer math and floating point notation. While of the available options, floating point notation offers the best precision for the number of bits used, floating point arithmetic is prohibitively costly in terms of hardware (Nichols, Moussa, and Areibi 2002). Conversely, integer math lacks the necessary precision. Using fixed point notation we are able to represent non integer values while maintaining comparable speeds and hardware costs to standard integer arithmetic.

Our network has a selectable data width which is subdivided into integer and fractional portions. For example a sign bit, 1 integer place, and 3 fractional places:

$$SI.FFF$$

This gives network data a precision of 2^{-F} (0.125 in our example) where F is the number of fraction bits and gives the data set D a maximum range of

$$-2^{N-1} \leq D \leq 2^{N-1} - 2^{-F} \tag{8}$$

($-2 \leq D \leq 1.875$ in our example) where $N = 1 + I + F$ is the total data width.

The choice of data width and precision has a direct impact on the width of the data paths in the network and the width of the necessary operators. The parameter thus acts as means to adjust the tradeoff between the size of the hardware footprint of the network and its accuracy.

Processing Element

An effort was made to keep the realization of a single PE as simple as possible. The motivation for this was that in our parallel hardware design many copies of this component would be generated for every network configuration. So, keeping this component small helps to minimize the size of the overall design. The control scheme was centralized external to the PE component to prevent the unnecessary duplication of functionality and complexity.

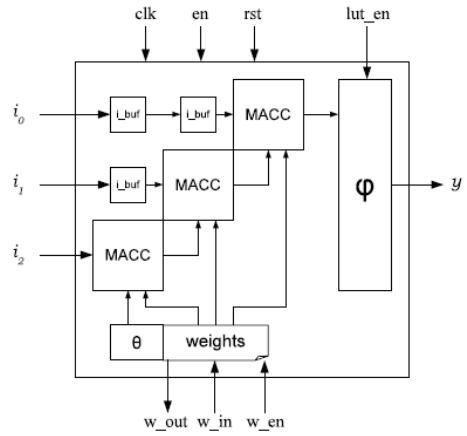

Figure 4: Functional blocks of the PE component.

In paring the design of the PE component down to its essential features we are left with a multiply-accumulate function with a width equal to the number of neurons in the previous layer, the selected squashing function implementation, and memory elements such as registers contained the synaptic weights and input buffers.

The PE, whose structure is shown in Fig. 4, is responsible for the full calculation of its output (1).

Squashing Function

The direct implementation of the preferred squashing function, a sigmoid function, presents a problem in hardware since both the division and exponentiation operations require an inordinate amount of time and hardware resources to compute. The only practical approach in hardware is to approximate the function (Tommiska 2003). But, in order for training to converge or for us to obtain accurate offline results, a minimum level of accuracy must be reached (Holt and Hwang 1993). More accurate approximations will result in faster, better convergences and a hence more accurate results. There has thus been a significant amount of research into how a sigmoid function can be efficiently be approximated in hardware while maintaining an acceptable degree of accuracy and the continuity required for reliable convergence in back-propagation learning (Tommiska 2003). To create a generalized design we must add one additional requirement for sigmoid function approximation, that the method of approximation must be valid for a variety of sigmoid functions.

Based on size and accuracy requirements to be met by the network we are generating, we may select one of two implementation styles for a sigmoid function that we have implemented in our generalized design: a uniform lookup table (LUT) or a LUT with linear interpolation.

Uniform Lookup Table Implementation

A uniform LUT implemented in block RAM may be used to approximate a function of any shape.

The LUT is addressed using the local field. The address is formed by taking the inverse of the sign bit of the local field and concatenating the most significant bits required to represent the highest input value of the function mapped onto the LUT down to the number of address bits of the LUT. All told, the computation requires one cycle and minimal hardware beyond that which is required to hold the table itself.

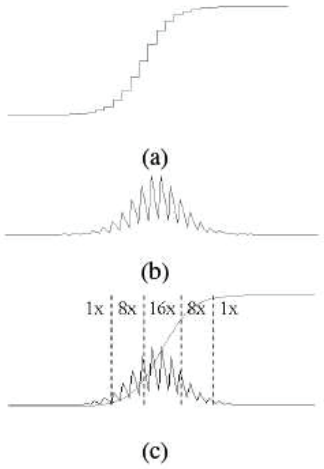

Figure 5: (a) Sigmoid function implemented on a uniform LUT. (b) Uniform LUT error distribution for a sigmoid function. (c) Example of partitioning for a variable resolution LUT.

The uniform LUT implementation, despite being popular in FPGA based ANNs and while efficient in terms of speed and size, presents a problem in terms of its size vs. accuracy tradeoff when it comes to modeling functions with steep slopes like a sigmoid function. As the slope increases so does the quantization error between LUT entries (Fig. 5). A common solution for this problem is the use of a LUT with variable resolution (Fig. 5c). That is, a LUT with higher resolution for portions of the function with steeper slopes. However, this is a solution that must be custom crafted for every given sigmoid and thus is not easily generalized as we would like.

To address the diminished accuracy of a uniform LUT while maintaining generalizability over a variety of functions, we incorporate linear interpolation alongside the LUT. This solution effectively draws a line between each point of a uniform LUT as in Fig. 6. The resulting activation function is

$$y = \frac{\mathrm{LUT}(d+1) - \mathrm{LUT}(d)}{2^{N-M}} q_e + \mathrm{LUT}(d) \qquad (9)$$

where LUT(d) is the value returned by a uniform LUT representing the target function for index d (done as described in section), N is the bit widths of the local field, M is the bit width of the LUT address bus, and q_e is the quantization error described by:

$$q_e = \varphi(v) - \mathrm{LUT}(d) \qquad (10)$$

The algorithm flow is controlled by a state machine inside the squashing function component enabled by the network controller. After receiving input, the result is registered at the output after 5 clock cycles.

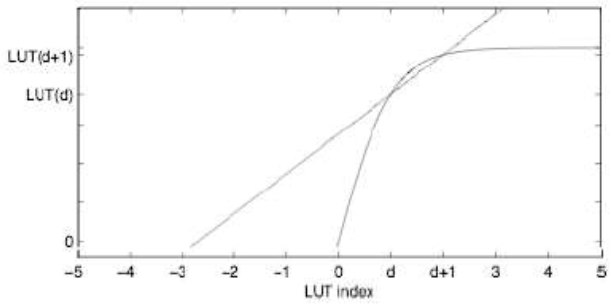

Figure 6: The hyperbolic tangent function is approximated by finding the nearest indices of the LUT corresponding to the local field, then drawing a line between the values referenced by those indices. The quantization error is then used to find the corresponding point on the line segment $\overline{\mathrm{LUT}(d)\mathrm{LUT}(d+1)}$.

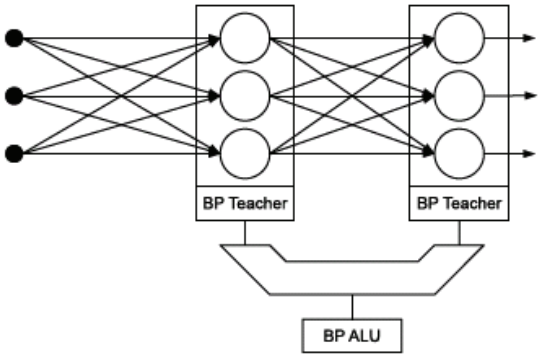

Figure 7: Back-propagation implementation.

Back-Propagation Computation

The back-propagation algorithm relies on the calculation of the network error at the output layer to estimate the error of neurons in the previous layers. The error is estimated and propagated backwards layer by layer until the first hidden layer is reached. It follows that the error estimation of any given layer except the output layer is dependent on the error calculation of its successor. Because of this, the training algorithm must be computed sequentially, layer by layer limiting parallelism to the neuron level.

The hardware implementation once again seeks to separate control and functional based components. Each layer contains its own back-propagation teacher component which is responsible for the control flow of the algorithm for its layer. Because the back-propagation algorithm is only being executed for one layer at a time we need only one set of the necessary arithmetic components.

Since the number of multipliers and adders, and the size of the MACC are dependent on the size of a given layer and its predecessor, we must compute and generate the worst case number of arithmetic components that are needed during elaboration of the given network design. The set of arithmetic components used in the back-propagation calculation are then packaged into the BP ALU component.

The BP ALU is accessed by each back-propagation teacher via an automatically generated multiplexer (Fig. 7) which is controlled by the network controller.

To optimize the performance of the training algorithm we begin execution during the forward pass of the network. In the forward pass we are able to compute two elements of the algorithm: $\eta\vec{i}$ once a given layer receives its input, and $\varphi'(v)$ once the layer's output has been computed. In the hidden layers, the results of these preprocessing steps are then saved until the error gradient reaches them in the backward pass. The output layer teacher continues immediately by calculating the output error and the local error gradient (4) for every neuron in the output. Once the error gradient has been calculated at the output layer, the final hidden layer may calculate its own error gradient (5) and pass that back.

Network Controller

The design of the network controller was strongly guided by the highly generalized design of the network and the initial decision to separate functional and control units. The decision to centralize control of the network was based on the goal of minimizing the size and complexity of components that must be generated multiple times. This is contrary to a distributed control mechanism made up of independent components capable of determining their own state and communicating that state to one another. This would be the more modular solution, but would also inflict a significant time and hardware penalty caused by the control overhead in the network, since control mechanisms would be repeated many times through the network. The centralized control scheme on the other hand, relies on the predictability of the timing and behavior of any generated network configuration.

Depending on the network to be generated, the network controller is created as either an online or offline network controller. Different implementations are necessary since in offline mode a pipelined network is generated and the online controller must include control for the computation of the back-propagation algorithm. Despite this, both controllers are implemented in the same manner.

The network controller is a Mealy state machine based on a counter indicating the number of clock cycles that have passed in the current iteration (in the case of an online network) or the total number of clock cycles passed (in the case of an offline network). For the value of the counter to have any meaning we must be able to pre-calculate the latency to reach milestones in the forward and back passes of the network. These milestones are calculated during elaboration of the design. Based on these milestones the state machine outputs a set of enable signals to control the flow of the network.

LUT with Linear Interpolation

To judge the accuracy of our approximation technique for sigmoid function using a LUT with linear interpolation we calculated the average error and worst case error of the technique for a range of network data precisions and LUT sizes. Using a uniform LUT as a control, we approximated the hyperbolic tangent with input ranging [-4:4]. Table 1 shows

Table 1: Worst case error of approximated hyperbolic tangent using a LUT with linear interpolation and a uniform LUT.

		LUT Size (log2)								
Lin-LUT		7	8	9	10	11	12	13	14	15
Fractional Precision	8	3.8E-04								
	9	3.8E-04	9.4E-05							
	10	3.8E-04	9.4E-05	2.3E-05						
	11	3.8E-04	9.4E-05	2.3E-05	5.9E-06					
	12	3.8E-04	9.4E-05	2.3E-05	7.9E-06	2.6E-06				
	13	3.8E-04	9.4E-05	2.3E-05	9.2E-06	3.9E-06	1.3E-06			
	14	3.8E-04	9.4E-05	2.3E-05	9.9E-06	4.6E-06	2.0E-06	6.6E-07		
	15	3.8E-04	9.4E-05	2.3E-05	1.0E-05	4.9E-06	2.3E-06	9.8E-07	3.3E-07	
	16	3.8E-04	9.4E-05	2.3E-05	1.0E-05	5.1E-06	2.5E-06	1.1E-06	4.9E-07	1.6E-07
Uni-LUT										
Fractional Precision	8	3.1E-02								
	9	4.7E-02	1.6E-02							
	10	5.5E-02	2.3E-02	7.8E-03						
	11	5.9E-02	2.7E-02	1.2E-02	3.9E-03					
	12	6.0E-02	2.9E-02	1.4E-02	5.9E-03	2.0E-03				
	13	6.1E-02	3.0E-02	1.5E-02	6.8E-03	2.9E-03	9.8E-04			
	14	6.2E-02	3.1E-02	1.5E-02	7.3E-03	3.4E-03	1.5E-03	4.9E-04		
	15	6.2E-02	3.1E-02	1.5E-02	7.6E-03	3.7E-03	1.7E-03	7.3E-04	2.4E-04	
	16	6.2E-02	3.1E-02	1.6E-02	7.7E-03	3.8E-03	1.8E-03	8.5E-04	3.7E-04	1.2E-04

Table 2: Average error of approximated hyperbolic tangent function using a LUT with linear interpolation and a uniform LUT.

		LUT Size (log2)								
Lin-LUT		7	8	9	10	11	12	13	14	15
Fractional Precision	8	6.1E-05								
	9	7.6E-05	1.5E-05							
	10	8.0E-05	1.9E-05	3.8E-06						
	11	8.1E-05	2.0E-05	4.8E-06	9.5E-07					
	12	8.2E-05	2.0E-05	5.0E-06	1.2E-06	2.4E-07				
	13	8.2E-05	2.0E-05	5.1E-06	1.3E-06	3.0E-07	6.0E-08			
	14	8.2E-05	2.0E-05	5.1E-06	1.3E-06	3.1E-07	7.5E-08	1.5E-08		
	15	8.2E-05	2.0E-05	5.1E-06	1.3E-06	3.2E-07	7.8E-08	1.9E-08	3.7E-09	
	16	8.2E-05	2.0E-05	5.1E-06	1.3E-06	3.2E-07	7.9E-08	2.0E-08	4.7E-09	9.3E-10
Uni-LUT										
Fractional Precision	8	3.9E-03								
	9	5.9E-03	2.0E-03							
	10	6.8E-03	2.9E-03	9.8E-04						
	11	7.3E-03	3.4E-03	1.5E-03	4.9E-04					
	12	7.6E-03	3.7E-03	1.7E-03	7.3E-04	2.4E-04				
	13	7.7E-03	3.8E-03	1.8E-03	8.5E-04	3.7E-04	1.2E-04			
	14	7.7E-03	3.8E-03	1.9E-03	9.1E-04	4.3E-04	1.8E-04	6.1E-05		
	15	7.8E-03	3.9E-03	1.9E-03	9.5E-04	4.6E-04	2.1E-04	9.1E-05	3.0E-05	
	16	7.8E-03	3.9E-03	1.9E-03	9.6E-04	4.7E-04	2.3E-04	1.1E-04	4.6E-05	1.5E-05

the worst case error using the two techniques and table 2 the average error. From the results we see that the LUT with linear interpolation provides a significant improvement in accuracy. For example, in a case with a network that uses 15-bit fractional precision, an 8192 element uniform LUT can be replaced by a 128 element LUT with linear interpolation and achieve a slightly better quality approximation on average.

Using the LUT with linear interpolation it becomes possible to reduce the error such that the magnitude of the error falls under the resolution of the network data precision. At this point we have reached a maximally precise approximation for the given network. Table 3 shows the resolution necessary to register the approximation error in the worst case for each set up in our tests. The boxed entries show the minimum LUT size for a given network data resolution to reach a maximally precise approximation.

Conclusion

In this paper we have presented the development and implementation of a parameterized FPGA based architecture for back-propagation MLPs. Our architecture makes native

Table 3: Bit resolution required to resolve the worst case approximation error.

		LUT Size (log2)								
Lin-LUT		7	8	9	10	11	12	13	14	15
Fractional Precision	8	-12								
	9	-12	-14							
	10	-12	-14	-16						
	11	-12	-14	-16	-18					
	12	-12	-14	-16	-17	-19				
	13	-12	-14	-16	-17	-18	-20			
	14	-12	-14	-16	-17	-18	-19	-21		
	15	-12	-14	-16	-17	-18	-19	-20	-22	
	16	-12	-14	-16	-17	-18	-19	-20	-21	-23
Uni-LUT										
Fractional Precision	8	-6								
	9	-5	-7							
	10	-5	-6	-8						
	11	-5	-6	-7	-9					
	12	-5	-6	-7	-8	-10				
	13	-5	-6	-7	-8	-9	-11			
	14	-5	-6	-7	-8	-9	-10	-12		
	15	-5	-6	-7	-8	-9	-10	-11	-13	
	16	-5	-6	-7	-8	-9	-10	-11	-12	-14

prototyping and design space exploration in hardware possible. Also presented was a new method for approximation of a sigmoid function in hardware. We showed that by applying a linear interpolation technique to a uniform LUT, we could significantly reduce the size of the necessary LUT while maintaining the same degree of accuracy at the cost of implementing one adder and one multiplier.

References

Basheer, I. A., and Hajmeer, M. 2000. Artificial neural networks: fundamentals, computing, design, and application. *Journal of Microbiological Methods* 43(1):3–31.

Bishop, D. Fixed point package. http://www.eda.org/vhdl-200x/vhdl-200x-ft/packages/fixed_pkg.vhd.

Holt, J., and Hwang, J. 1993. Finite precision error analysis of neural network hardware implementations. *Computers, IEEE Transactions on* 42(3):281–290.

Ienne, P.; Cornu, T.; and Kuhn, G. 1996. Special-purpose digital hardware for neural networks: An architectural survey. *The Journal of VLSI Signal Processing* 13(1):5–25.

Izeboudjen, N.; Farah, A.; Bessalah, H.; Bouridene, A.; and Chikhi, N. 2007. *Towards a Platform for FPGA Implementation of the MLP Based Back Propagation Algorithm*. 497–505.

Liu, J., and Liang, D. 2005. A survey of FPGA-Based hardware implementation of ANNs. In *Neural Networks and Brain, 2005. ICNN&B '05. International Conference on*, volume 2, 915–918.

Mead, C., and Mahowald, M. 1988. A silicon model of early visual processing. *Neural Networks* 1:97, 91.

Nichols, K.; Moussa, M.; and Areibi, S. 2002. Feasibility of Floating-Point arithmetic in FPGA based artificial neural networks. In *Proceedings of the 15th International Conference on Computer Applications in Industry and Engineering*.

Omondi, A., and Rajapakse, J. 2002. Neural networks in FPGAs. In *Neural Information Processing, 2002. ICONIP '02. Proceedings of the 9th International Conference on*, volume 2, 954–959 vol.2.

Paliwal, M., and Kumar, U. A. 2008. Neural networks and statistical techniques: A review of applications. *Expert Systems with Applications*.

Rumelhart, D. E.; Hinton, G. E.; and Williams, R. J. 1986. *Learning internal representations by error propagation*, volume 1. MIT Press. 318–362.

Sanchez, E. 1996. FPGA implementation of an adaptable-size neural network. *In Proceedings of the International Conference on Artificial Neural Networks ICANN96* 1112:383—388.

Schrauwen, B.; D'Haene, M.; Verstraeten, D.; and Campenhout, J. V. 2008. Compact hardware liquid state machines on FPGA for Real-Time speech recognition. *Neural Networks* 21(2-3):511–523.

Theeten, J. B.; Duranton, M.; Mauduit, N.; and Sirat, J. A. 1990. The LNeuro chip: A digital VLSI with on-chip learning mechanism. In *Proceedings of the International Conference on Neural Networks*, volume 1, 593–596.

Tommiska, M. 2003. Efficient digital implementation of the sigmoid function for reprogrammable logic. *IEE Proceedings Computers and Digital Techniques* 150(6):403–411.

Widrow, B.; Rumelhart, D. E.; and Lehr, M. A. 1994. Neural networks: applications in industry, business and science. *Communications of the ACM* 37(3):93–105.

Xilinx. 2007. Virtex-5 FPGA user guide.