Handwritten Digit Recognition System on an FPGA

Jiong Si
Department of Electrical and Computer Engineering
University of Nevada, Las Vegas
Las Vegas, NV, U.S.A.
sij1@unlv.nevada.edu

Sarah L. Harris
Department of Electrical and Computer Engineering
University of Nevada, Las Vegas
Las Vegas, NV, U.S.A.
sarah.harris@unlv.edu

***Abstract*—This paper describes our implementation of a multilayer perceptron (MLP) learning network on a Cyclone IVE field programmable gate array (FPGA). The MLP uses MNIST data, the Modified National Institute of Standards and Technology database of handwritten digits, to train and test the design. Working with 8-bit precision, the FPGA design has similar accuracy and execution time as the 32-bit software solution but with 144 times slower clock frequency. With power consumption being proportional to frequency, the hardware solution provides power savings at no cost in accuracy or performance. Further reducing the precision from 8 to 4 bits only reduces accuracy from 89% to 78%, with area decreasing by 41%. Thus, the FPGA implementation of the MLP learning network offers a high-performance, low power alternative to traditional software methods.**

Keywords—MNIST; FPGA; MLP; Deep learning; Machine learning; Hardware acceleration

I. INTRODUCTION

Machine learning and deep learning algorithms and their applications are becoming increasingly prevalent. While these algorithms enhance system intelligence, they also have the disadvantages of being compute intensive and demanding in terms of power consumption and execution time. These algorithms are typically run on CPUs or GPUs, which are not specifically tuned to run machine learning algorithms. Several recent machine learning programs combine software algorithms with hardware specially designed for such algorithms. For example, AlphaGo Zero [1] is a computer program implemented on the Tensor Processing Unit (TPU) [2] designed by Google. A TPU is a domain-specific custom application specific integrated circuit (ASIC) designed to implement machine learning algorithms. The TPU is 15-30 times faster and 30-80 times more power efficient than modern GPUs and CPUs.

Field programmable gate arrays (FPGAs) offer similar advantages to ASICs such as Google's TPU by providing the ability to configure hardware specific to machine learning algorithms, including parallel execution. This hardware-specific design on an FPGA offers increased performance, lower power consumption, and decreased cost compared to a CPU implementation. It also offers the advantage over ASIC designs of increased flexibility and decreased design to implementation time.

Over the past decade or more, several groups have implemented handwritten digit recognition on an FPGA [3-6]. However, each of these designs use selected image features, instead of the whole image, to test the neural network. Additionally, these designs exhibit lower accuracy [3, 4] and larger neural networks that require more hardware [5]. Tay, et al [6] use 16-bit precision but their design shows an only 2% increase in accuracy over the 8-bit precision solution in this paper. The execution time during testing for each image is four times smaller than our system but runs at a clock speed that is eight times faster than our proposed design. This indicates an effective execution time of two times slower or approximately twice the power consumption. Huynh, et al use floating point arithmetic for calculations instead of fix point, which costs much more in hardware resources and execution time [7]. Another two solutions uses the on-chip soft-core processor and, thus, fails to take advantage of hardware acceleration [4, 8].

This paper describes our FPGA design of a multilayer perceptron (MLP) learning network used to train and test data from the Modified National Institute of Standards and Technology (MNIST) database of handwritten digits. After introducing the overall algorithm in the Methods section (Section II), we continue by describing both software and hardware (FPGA) implementations of the MLP algorithm. Section III describes the performance and area usage of the FPGA design as compared to the software implementation. We also show results for varying bits of precision. Finally, Section IV summarizes our findings and describes future work.

II. METHODS

This section describes the MLP algorithm including the MLP network architecture, MNIST dataset, and the sigmoid activation function. It also describes the MLP hardware design on the FPGA including the overall system architecture and the control and computation units.

A. MNIST Dataset

The MNIST dataset of handwritten digits used in this paper includes a training set of 55,000 images and their labels, and a testing set of 10,000 images and their labels. Figure 1 shows five image examples from the MNIST dataset with their labels shown above each image. Each image is 28 × 28 pixels (784 total pixels), as shown by the x and y labels on the images.

B. Multilayer Perceptron Network

The machine learning network we designed is a two-layer fully connected perceptron network that trains on the MNIST database of single handwritten digits ranging from 0 to 9. The goal of the network is to first train on a subset of the

978-1-5386-4649-6/18/$31.00 ©2018 IEEE

handwritten data and then predict values (from 0 to 9) for other test images of handwritten digits.

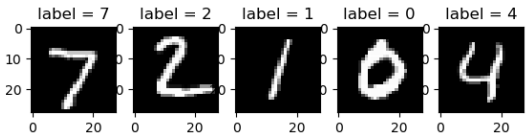

Fig. 1. Examples of digits in the MNIST dataset.

In a fully-connected two-layer MLP, the first layer consists of the inputs, the second layer is the output layer that sums the weighted inputs, and then goes through activation function to produce the outputs, the prediction that the input image was one of the ten possible digits.

The architecture of the MLP network is shown in Figure 2 and is described here. Because each MNIST image consists of 784 pixels, the input layer consists of 784 inputs, $x_0, x_1, \ldots x_{783}$. The second layer, also called the output layer, contains calculation nodes, or neurons, that sum the weighted inputs [9] and send their outputs through the activation function to produce the outputs. With 10 possible outputs (i.e., digits 0-9), the network has 10 neurons in the output layer, as shown in Figure 2.

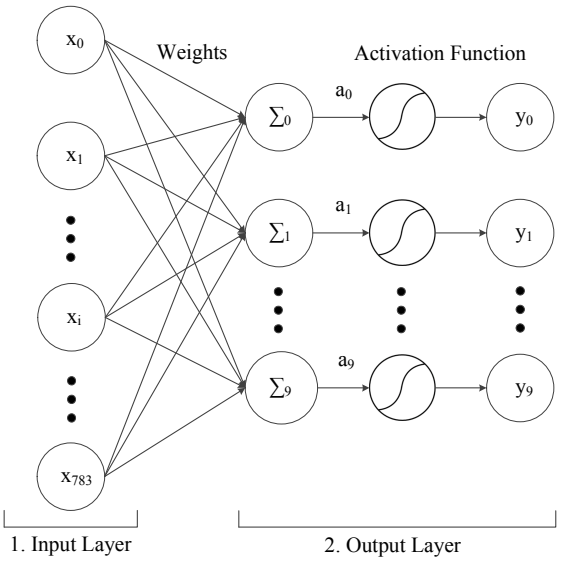

Fig. 2. Multilayer Perceptron Network.

The weights matrix consists of ten weights (one for each possible digit) for each of the 784 input pixels. So the system has 784 × 10 = 1784 weights: ($w_{0,0}$, $w_{0,1}$, $w_{0,2}$,…$w_{0,9}$, …$w_{783,0}$,… $w_{783,9}$). Each weight represents a synaptic weight (activating or inhibiting). For example, with 1 bit of precision, activating would be 1 and inhibiting 0. But with higher precision, the weight is not binary but instead has fractional values. So, with 8 bits of precision, the weights have 256 (2^8) fractional values between full activation (the maximum value) and complete inhibition (the minimum value).

Each of the 10 neurons in the output layer corresponds to a given digit (0-9) and that neuron sums the input pixels using the weights corresponding to that digit. The outputs of these summations, called $a_0, a_1, \ldots a_9$, are then transformed through an activation function to produce the final outputs. The final ten outputs of the MLP system are called $y_0, y_1, \ldots, y_9$. The outputs give the probability that an image is a given digit; y_0 gives the probability of the digit being 0, y_1 gives the probability of the digit being 1, and so on. For example, an output of $\{y_0, y_1, \ldots, y_9\}$ = {0.1, 0.2, **0.9**, 0.3, 0.1, 0.2, 0.3, 0.1, 0.2, 0.1} would predict that the handwritten image was of the digit 2.

The process described above is called testing or *forward propagation*. Information flows from inputs to outputs, as represented by the arrows in Figure 2. During the training process, called *backward propagation*, the MLP calculates the errors between the predicted output probabilities and the actual outputs, provided by the image labels (see Figure 1). During back propagation, these errors are used to update the synaptic weights of each neuron, as described in Section C.2.

C. Activation Function

An activation function inhibits low inputs and accentuates high inputs. In this way, activation functions reflect neuron behavior, where neurons require an input above some threshold to activate.

A common activation function, also used here, is the sigmoid function shown in equation (1).

$$f(x) = \frac{1}{1 + e^{-x}} \tag{1}$$

Figure 3 shows the sigmoid function with 8 bits of precision (and an approximation of that function which is discussed in Section IIID). The inputs of the function vary from -128 to 127 instead of 0 to 1 as is the case with 1-bit precision. An input of -128 indicates complete inhibition and an input of 127 indicates full activation. So, for example, an input of -128 to the sigmoid function would result in an output very close to 0; an input of 0 would result in an output of 16, which means half inhibition and half activation. The sigmoid function also rescales the output to only positive values in a range of 0 to 32, where 0 indicates inhibition and 32 is fully activated. So, as shown, the sigmoid function mimics neuron behavior by accentuating high values and minimizing low values, as desired.

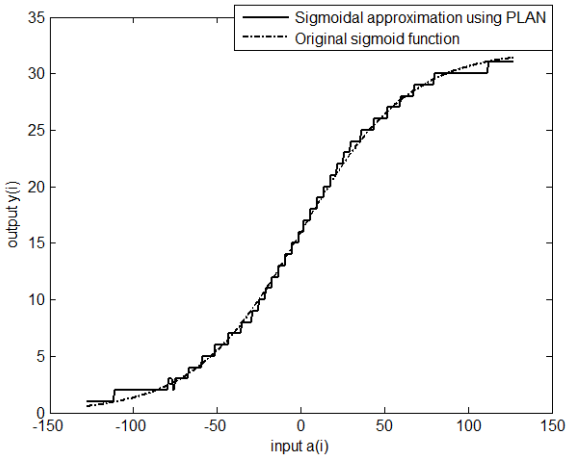

Fig. 3. Sigmoid function and its approximation using PLAN.

D. FPGA System

This section describes the MLP network we built in SystemVerilog [10] on a Cyclone IVE FPGA. The system diagram of the MLP network is shown in Figure 4 and is described in detail in this section. We implement the network using 8-, 6-, 5- and 4-bit precision for all inputs, outputs, and calculations.

The MLP hardware system (see Figure 4) consists of a UART communications module, Image/Label RAM, and a Controller that directs the Computation Unit. The system also outputs results to a 7-segment display. The UART module transmits all training and test images and their labels from the PC to the FPGA. The system then stores these data in the Image/Label RAM. After a single image and its label are transferred to the FPGA, the Controller module is triggered to start either training or testing, depending on whether the system is in backward or forward propagation mode (described further in Section C.1). As directed by the Controller module, the Computation Unit completes forward or backward propagation calculations, as described in Section C.2.

The Computation Unit reads the weights from the Weights RAM in testing mode and both reads and updates the weights in training mode. At startup, the Weights RAM is initialized to random values. During both training and testing, the Computation Unit reads the weights from the Weights RAM to perform the weighted sum of the inputs. During training, the Computation Unit then also calculates the errors between the calculated outputs and the actual values and updates the Weights RAM with calculated delta weights, as described in detail in Section C.2. During this backward propagation process, the weights are updated after each training image is processed.

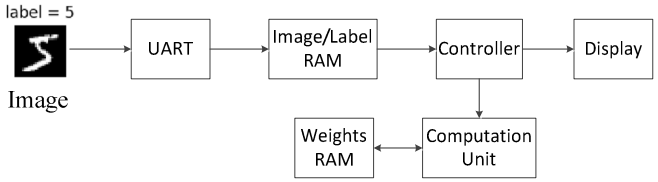

Fig. 4. FPGA system architecture.

To simplify the computation and, thus, decrease the area, execution time, and power consumption required by the sigmoid transformation, our system approximate the sigmoid function using a piecewise linear approximation of a nonlinear function (PLAN) [11]. Table I shows the output y as a function of the input a given several input ranges. So, for example, if the input is 16, the output would be 20. Thus, the PLAN approximation replaces the exponential function and division (see equation (1)) with multiplication and addition.

TABLE I. IMPLEMENTATION OF PLAN [11]

Output: y = F(a)	*Input Condition*
y = 32	a >= 160
y = 0.03125[a] * a + 27	76 =< a <160
y = 0.125[b] * a + 20	32 =< a < 76
y = 0.25[c] * a + 16	0 =< a < 32
y = 1 – y	a < 0

[a.] Right shift 5 bits; [b.] right shift 3 bits; [c.] right shift 2 bits.

The comparison of the sigmoid function and its approximation using PLAN are given in Figure 3. As shown, the approximation deviates only slightly from the original sigmoid function.

C.1 Controller

The Controller module manages the two main processes: forward propagation and backward propagation. It also displays the target and calculated outputs. Figure 5 shows the finite state machine (FSM) of the Controller. In the Idle state, the FSM waits for the start signal to assert. After the UART module receives and stores one image and its label into the Image/Data RAM, the UART module asserts the start signal, thus moving the FSM to the forward (Fwd) state and triggering the forward propagation process. After forward propagation computations are complete, if the system is in testing mode, the FSM displays the image label (target result) and predicted results (y_0,…y_9) on the 7-segment displays and then returns to the Idle state to continue processing test images. However, if the system is in training mode, the FSM moves from the forward propagation state (Fwd) to the backward propagation state (Back) to update the weights in the Weights RAM (see Section C.2). After backward propagation is complete, the FSM displays the image label and training results on the 7-segment displays and then moves to the Idle state to continue processing images, as was done in testing mode.

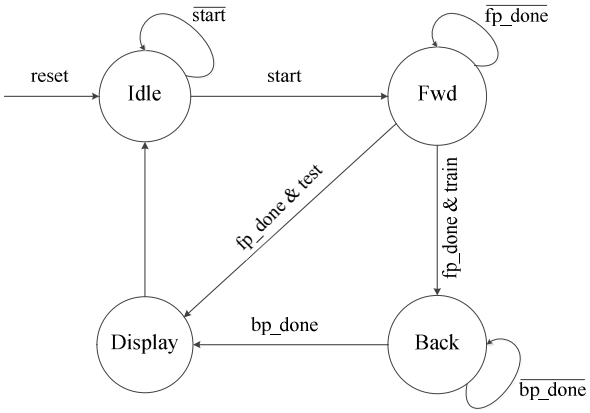

Fig. 5. Controller finite state machine (FSM).

C.2 Computation Unit

The Computation unit acts as the ten system neurons by performing the calculations of the output layer. This unit also updates the weights when the system is in training mode. After being triggered by the Controller moving to the forward propagation (Fwd) state, the Computation Unit calculates the weighted sums of each input (a_0-a_9, see equation (2)) and then passes these results through the activation function to produce the outputs (y_0-y_9, see equation (3)).

$$\begin{aligned} a_0 &= w_{0,0}x_0 + w_{0,1}x_1 + \cdots + w_{0,783}x_{783} \\ &\vdots \\ a_9 &= w_{9,0}x_0 + w_{9,1}x_1 + \cdots + w_{9,783}x_{783} \end{aligned} \tag{2}$$

$$y_0 = sigmoid(a_0)$$

$$y_9 = sigmoid(a_9) \quad (3)$$

If the system is in backward propagation mode, the system still computes the outputs (y_0 - y_9) in the forward state (Fwd) but then also proceeds to the backpropagation state (Back) to both compute the errors between the outputs and target results and update the weights. The error calculation for each digit i, where i is 0 to 9, is shown in equation (4). The values of the actual results, $target_i$, are obtained from the Label RAM.

$$error_i = target_i - y_i \quad (4)$$

These errors are then used to update the weights in the Weights RAM. However, the errors are first passed through the activation function and then multiplied by the input pixels before being used to update the weights. The errors pass through the sigmoid function by multiplying the error ($error_i = \Delta y_i$) by the gradient (df/dy) of the sigmoid function, $f(y)$. The gradient of the sigmoid function is given in equation (5).

$$f'(y) = f(y) * (1 - f(y)) \quad (5)$$

The change in weights (that will be multiplied by the inputs before being added to the weights) is then calculated as given in equation (6).

$$w_{change_i} = \Delta y_i * \frac{df}{dy} = error_i * f'(y_i) \quad (6)$$

This calculation can then be rewritten as:

$$w_{change_i} = error_i * f(y_i) * (1 - f(y_i)) \quad (7)$$

The MLP hardware system calculates the change for each weight by multiplying the input by the calculated weight change, as shown in equation (8), where $w_{delta_{i,j}}$ is the amount to change the weight, i is 0 to 9 for each of the digits, j is 0 to 783 for each of the input pixels, and x_j is the value of each input pixel in the training image.

$$w_{delta_{i,j}} = x_j * w_{change_i} \quad (8)$$

The weights used to process the next image are the updated weights, $w_{new_{i,j}}$, shown in equation (9).

$$w_{new_{i,j}} = w_{old_{i,j}} + w_{delta_{i,j}} \quad (9)$$

The calculations for forward and backward propagation are summarized in equations (10) and (11), respectively.

$$\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_9 \end{bmatrix} = f\left[\begin{pmatrix} w_{0,0}x_0 + w_{0,1}x_1 + \cdots + w_{0,783}x_{783} \\ w_{1,0}x_0 + w_{1,1}x_1 + \cdots + w_{1,783}x_{783} \\ \vdots \\ w_{9,0}x_0 + w_{9,1}x_1 + \cdots + w_{9,783}x_{783} \end{pmatrix}\right] \quad (10)$$

$$w_{new_{i,j}} = w_{old_{i,j}} + x_j * (error_i) * f(y_i) * [1 - f(y_i)] \quad (11)$$

III. Results and Discussion

This section compares the accuracy and execution time of the software solution with our proposed hardware solutions and discusses the FPGA area requirements with varying bits of precision.

A. Accuracy

The accuracy of the 8-, 6-, 5- and 4-bit hardware designs vary from 73-89% as compared to the 32-bit precision software implementation that achieves an accuracy of 68-89%, depending on the number of training images. Figure 6 summarizes these results. The horizontal axis shows the number of training images (up to 55,000), and the vertical axis represents the accuracy, the percentage of correct predictions using 10,000 test images. The 8-bit FPGA solution's accuracy reaches (and slightly exceeds) that of the 32-bit software solution when using the maximum number of training images (55,000). Moreover, it has double the convergence speed – the 8-bit FPGA solution requires only 20,000 training images to achieve the highest prediction accuracy, while the 32-bit software solution required 40,000 training images to converge.

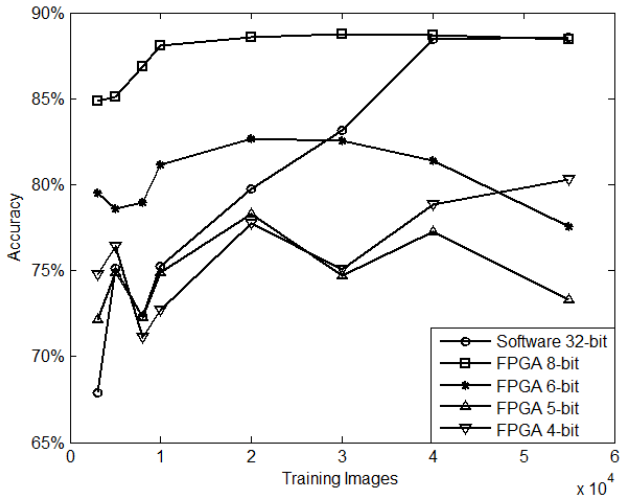

Fig. 6. Software solution accuracy vs. FPGA solutions accuracy.

Further reducing the bits of precision results in only 6-11% drops in accuracy (when using 55,000 training images), as shown in Figure 6 and summarized in Table II. Compared with the 8-bit hardware solution, the 6-bit solution's accuracy drops by only 6%, and the 5- and 4-bit solutions have accuracies that are only 9-11% lower than the 8-bit solution. So, reducing the precision by 50% from 8 bits to 4 bits results in an accuracy decrease of only 9%. Table II indicates accuracy is not affected when decreasing precision from 32 to 8 bits, but additional decreases in precision costs approximately 4% in accuracy per bit decrease in precision.

TABLE II. Accuracy

Implementation	*Average Accuracy*
32-bit software	89%
8-bit FPGA	89%
6-bit FPGA	83%
5-bit FPGA	78%
4-bit FPGA	80%

B. Performance

The performance differences, as measured by execution time, between software and hardware solutions are summarized in Table III. The clock frequency of the software solution is the CPU frequency (3.6 GHz), and the clock frequency of the FPGA designs (25 MHz) is the highest frequency at which the

designs can run on the Cyclone IVE FPGA. The execution time measured includes both training and testing time when using 55,000 training images and 10,000 test images. Execution time of both the software and hardware designs is almost identical: 3.7 seconds in software and 3.8 seconds for each FPGA solution.

TABLE III. PERFORMANCE

Solution	*Clock Frequency*	*Execution Time*	*Performance*
Software	3.6 GHz	3.7 seconds	1×
Hardware	25 MHz	3.8 seconds[d]	140×

[d] Includes calculation time for forward and backward propagation (but not UART transfer).

While the FPGA hardware designs complete the computations in essentially the same time as the software implementation, the clock frequency of software model is 144 times faster than that of the FPGA designs. Thus, to compare solutions running at the same frequency, the hardware solution running at 3.6 GHz would have a performance 140 times faster (144 × 3.7s/3.8s) than the software solution. On the other hand, keeping the lower frequency of the hardware design produces an approximate 140× decrease in power consumption for training and testing calculations, using the relationship that power is proportional to operating frequency. Thus, the FPGA hardware designs offer higher performance or lower power alternatives to the software implementation.

C. Area

The FPGA area requirements for the MLP hardware design using 4, 5, 6, and 8 bits of precision requires 20-34k logic elements (LEs). Table IV summarizes the area requirements of each hardware design as well as the percent use of the FPGA's total LEs. As expected, lower bit width solutions require fewer logic elements, with the 4-bit solution using about 40% less area than the 8-bit solution.

TABLE IV. AREA

FPGA Solution	*Logic elements (% use[e])*
8-bit width	34 k (29%)
6-bit width	26 k (23%)
5-bit width	22 k (19%)
4-bit width	20 k (17%)

[e] Intel's Cyclone IVE FPGA.

Figure 7 shows the area usage (relative to the 8-bit version) versus prediction accuracy for varying bit width solutions when using 55,000 training images and 10,000 test images. The 6-bit solution uses 24% fewer logic elements than the 8-bit solution with an only 6% accuracy drop (from 89% to 83%), as shown in Figure 7. The 5-bit solution saves another 11% of the logic elements compared with the 6-bit solution but only has a 5% accuracy drop (from 83% to 78%). The 4-bit design requires 6% fewer logic elements than the 5-bit design while maintaining accuracy similar to that design. Figure 7 shows the actual accuracy decrease with decreasing precision as well as a trend line. As the trend line shows, from 8-bit precision to 4-bit precision, the accuracy drops 11% (from 89% to 78%), and the area decreases by 41%. So area decreases at approximately 4% per percent decrease in accuracy.

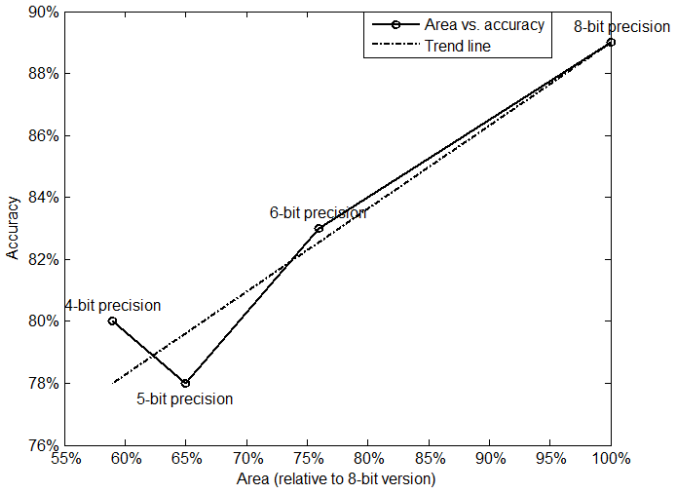

Fig. 7. Accuracy (percentage of correct predictions) vs. area (relative to 8-bit version) for the 4-, 5-, 6-, and 8-bit MLP FPGA designs.

Figure 8 shows data width versus area and a trend line for each FPGA MLP design. As depicted in the figure, the trend line shows that area grows at approximately 10% per bit of precision.

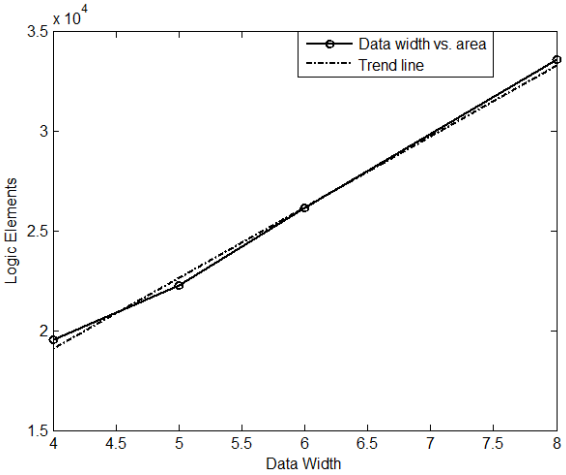

Fig. 8. Data width vs. area (number of FPGA logic elements).

IV. CONCLUSIONS AND FUTURE WORK

The FPGA hardware design of the MLP learning network presented here offers a high-performance, low power alternative to traditional software methods. The 8-bit hardware design performs with similar execution time (3.8 seconds) and prediction accuracy (89%) as the 32-bit software solution running at a clock speed 144 times greater than the hardware design (3.6 GHz vs. 25 MHz). This difference in clock frequency indicates that the hardware solution offers either lower power consumption or potential increased performance of 140 times, at no cost to accuracy, as compared to the software solution.

Furthermore, a reduction in precision from 32 to 8 bits results in no decrease in accuracy. Additional reductions in precision below 8 bits result in only small reductions in

accuracy (4% accuracy reduction per bit of reduced precision), moderate area decreases (10% decreased area per bit of precision), and a resulting area decrease higher than that of the accuracy decrease (4% decrease in area per percent decrease in accuracy).

Future work includes directly measuring power consumption of both the software and hardware solutions, decreasing bit widths even further, and designing FPGA hardware accelerated versions of other machine learning and deep learning algorithms including MLP models with more layers, convolutional neural network (CNN) [12], and generative adversarial nets (GAN) [13]. Pruning techniques such as pooling [14] and dropout [15] will also be applied to obtain smaller and faster hardware models. Beyond MNIST, FPGA solutions for more complex databases such as CIFAR [16] and ImageNet [17] will also be tested to compare performance, area, and power tradeoffs as compared to software-based solutions.

REFERENCES

[1] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of Go without human knowledge," Nature, vol. 550, no. 7676, pp. 354–359, 2017.

[2] N. P. Jouppi, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, C. Young, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, N. Patil, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Patterson, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, G. Agrawal, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, R. Bajwa, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, S. Bates, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, D. H. Yoon, S. Bhatia, and N. Boden, "In-Datacenter Performance Analysis of a Tensor Processing Unit," Proceedings of the 44th Annual International Symposium on Computer Architecture - ISCA '17, pp. 1–12, 2017.

[3] E. Bouvett, O. Casha, I. Grech, M. Cutajar, E. Gatt, and J. Micallef, "An FPGA embedded system architecture for handwritten symbol recognition," Proceedings of the Mediterranean Electrotechnical Conference - MELECON, pp. 653–656, 2012.

[4] A. Suyyagh and G. Abandah, "FPGA Parallel Recognition Engine for Handwritten Arabic Words," Journal of Signal Processing Systems, vol. 78, no. 2, pp. 163–170, 2013.

[5] M. Moradi, M. A. Pourmina, and F. Razzazi, "FPGA-Based farsi handwritten digit recognition system," International Journal of Simulation: Systems, Science and Technology, vol. 11, no. 2, pp. 17–22, 2010.

[6] V. Tay, "Design of Artificial Neural Network Architecture for Handwritten Digit Recognition on FPGA," no. November, 2016.

[7] T. V. Huynh, "Design space exploration for a single-FPGA handwritten digit recognition system," 2014 IEEE Fifth International Conference on Communications and Electronics (ICCE), pp. 291–296, 2014.

[8] L. B. Saldanha and C. Bobda, "An embedded system for handwritten digit recognition," Journal of Systems Architecture, vol. 61, no. 10, pp. 693–699, 2015.

[9] M. A. Nahmias, B. J. Shastri, A. N. Tait, and P. R. Prucnal, "A leaky integrate-and-fire laser neuron for ultrafast cognitive computing," IEEE Journal on Selected Topics in Quantum Electronics, vol. 19, no. 5, 2013.

[10] K. Kudrolli, S. Shah, and D. Park, "Handwritten Digit Classification on FPGA," 2015.

[11] H. Amin, K. M. Curtis, and B. R. Hayes-Gill, "Piecewise linear approximation applied to nonlinear function of a neural network," Circuits, Devices and Systems, IEE Proceedings -, vol. 144, no. 6, pp. 313–317, 1997.

[12] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," Advances In Neural Information Processing Systems, pp. 1–9, 2012.

[13] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative Adversarial Nets," Advances in Neural Information Processing Systems 27, pp. 2672–2680, 2014.

[14] K. He, X. Zhang, S. Ren, and J. Sun, "Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition," pp. 1–14, 2014.

[15] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," pp. 1–18, 2012.

[16] https://www.cs.toronto.edu/~kriz/cifar.html

[17] http://www.image-net.org/