Journal of Computer and Communications, 2020, 8, 251-277
https://www.scirp.org/journal/jcc
ISSN Online: 2327-5227
ISSN Print: 2327-5219

Neural Networks on an FPGA and Hardware-Friendly Activation Functions

Jiong Si, Sarah L. Harris*, Evangelos Yfantis

University of Nevada, Las Vegas, USA
Email: *sarah.harris@unlv.nevada.edu

How to cite this paper: Si, J., Harris, S.L. and Yfantis, E. (2020) Neural Networks on an FPGA and Hardware-Friendly Activation Functions. *Journal of Computer and Communications*, **8**, 251-277.
https://doi.org/10.4236/jcc.2020.812021

Received: December 1, 2020
Accepted: December 28, 2020
Published: December 31, 2020

Abstract

This paper describes our implementation of several neural networks built on a field programmable gate array (FPGA) and used to recognize a handwritten digit dataset—the Modified National Institute of Standards and Technology (MNIST) database. We also propose a novel hardware-friendly activation function called the dynamic Rectified Linear Unit (ReLU)—D-ReLU function that achieves higher performance than traditional activation functions at no cost to accuracy. We built a 2-layer online training multilayer perceptron (MLP) neural network on an FPGA with varying data widths. Reducing the data width from 8 to 4 bits only reduces prediction accuracy by 11%, but the FPGA area decreases by 41%. Compared to networks that use the sigmoid function, our proposed D-ReLU function uses 24 - 41% less area with no loss to prediction accuracy. Further reducing the data width of the 3-layer networks from 8 to 4 bits, the prediction accuracies only decrease by 3 - 5%, with area being reduced by 9 - 28%. Moreover, FPGA solutions have 29 times faster execution time, even despite running at a 60× lower clock rate. Thus, FPGA implementations of neural networks offer a high-performance, low power alternative to traditional software methods, and our novel D-ReLU activation function offers additional improvements to performance and power saving.

Keywords

Deep Learning, D-ReLU, Dynamic ReLU, FPGA, Hardware Acceleration, Activation Function

1. 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 de-

manding in terms of power consumption and execution time. 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 when running machine learning algorithms.

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, for example, to support the parallel execution required by machine learning algorithms. 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, low cost for a small number of units, and decreased design to implementation time.

One of the most compute-intensive steps of any machine learning algorithm is performing the activation function calculation. Commonly used activation functions, such as sigmoid and tangent functions, are highly compute-intensive. In contrast, the ReLU function is an activation function that requires relatively simple calculations [3]. As emphasized by Krizhevsky, *et al.* [4], the advantages of using the ReLU function include faster training speed, decreased saturation problems, smaller numbers of epochs, and usually fewer samples. However, the conventional ReLU activation function has the disadvantages of potentially causing a neural network to explode (*i.e.*, retain too much information) or die (*i.e.*, retain too little information) during learning calculations. Several recent papers attempt to design ReLU functions that address these disadvantages, including the Leaky ReLU [5], Parametric ReLU [6], and Randomized Leaky ReLU. In these functions, while the parameters are either fixed or modified during training, they are fixed during testing. In contrast, our proposed D-ReLU algorithm allows for parameter modification during both training and testing.

Currently, machine learning algorithms typically run on CPUs or GPUs, but these platforms have shortcomings, the most notable of which are fixed calculation resources and high power consumption. Systems built on an FPGA platform, however, offer algorithm-specific hardware and low power consumption [7] [8], are easy to pipeline architecturally [7] [9] and algorithmically [10], and allow for quantization and compression of weights [8] [11]. Several current FPGA implementations of MLP networks exist [10] [11] [12] [13], but they have some disadvantages, such as low prediction accuracy [12] [13], the use of floating point instead of fixed-point arithmetic [14], and large area requirements [15].

This paper describes our FPGA designs of MLP learning networks 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 2), we continue by describing the hardware (FPGA) implementations of the MLP algorithm. We also propose a novel dynamic ReLU (D-ReLU) activation function in this section to simplify calculation and decrease area requirements and power consumption of neural networks. We apply this algorithm to 2- and 3-layer MLP networks designed both in software and in hardware on an FPGA. Section 3 describes the performance and area usage of the FPGA design as compared to the software implementation. We also show results for varying bits of data width. Section 4 summarizes our findings and conclusions.

2. Methods

This section describes the MLP algorithm, dataset, and the network architectures. It then introduces the FPGA platforms, their functional units, and calculations. The section concludes by diving into the details of the activation function, It first describes the sigmoid function approximation method used to simplify computations and decrease FPGA area, and it then introduces the proposed dynamic ReLU function, that offers high prediction accuracy at low computation cost when compared to three other commonly used activation functions: the conventional ReLU function, the modified ReLU function, and the sigmoid function.

2.1. Two-Layer Online Training MLP Neural Network

The simplest 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, first, to train on a subset of the handwritten data and, second, to then predict values (from 0 to 9) for the remaining test images of handwritten digits.

2.1.1. MNIST Dataset

In this paper, we use the MNIST dataset (the Modified National Institute of Standards and Technology database of handwritten digits) to train and test the MLP networks. The MNIST dataset includes a training set of 60,000 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 in Figure 1.

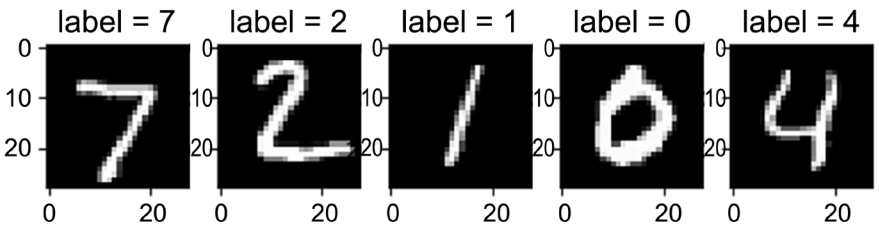

Figure 1. Examples of digits in the MNIST dataset.

We use a subset of 55,000 training images in this paper. In the process of model learning, we introduce a validation phase to minimize overfitting. Specifically, the 55,000 training images are divided into two sets: one set of 50,000 images are used as training data, the other set of 5,000 images are used as validation data. We use 100 iterations to find the optimal *learning rate*—in the range between 0.0 and 1.0, which is the amount that the weights are updated during training—based on the prediction accuracies in validation data, but the network is only trained by one epoch with the optimal learning rate. Finally, the trained network is used to process the test images.

2.1.2. Network Architecture

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 the 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, $x_0, x_1, \cdots, x_{783}$. The second layer, also called the output layer, contains calculation nodes, or neurons, that sum the weighted inputs [16] 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**.

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 = 7840 weights: ($w_{0,0}, w_{0,1}, w_{0,2}, \cdots, w_{0,9} \cdots, w_{783,0}, \cdots, w_{783,9}$). Each weight represents a synaptic weight (activating or inhibiting). For example, with one 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 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 $a_0, a_1, \cdots, a_9$, are then transformed through an activation function to produce the final outputs. The final ten outputs of the MLP system are called $y_0, y_1, \cdots, y_9$. The outputs give the probability that an image is a given digit; for example, 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 $\{y_0, y_1, \cdots, y_9\} = \{0.1, 0.2, 0.9, 0.3, 0.1, 0.2, 0.3, 0.1, 0.2, 0.1\}$ would predict that the handwritten image was the digit 2.

In this testing or *forward propagation* process described above, information flows from inputs to outputs, as represented by the arrows in **Figure 2**. During the training or *back propagation* process, 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

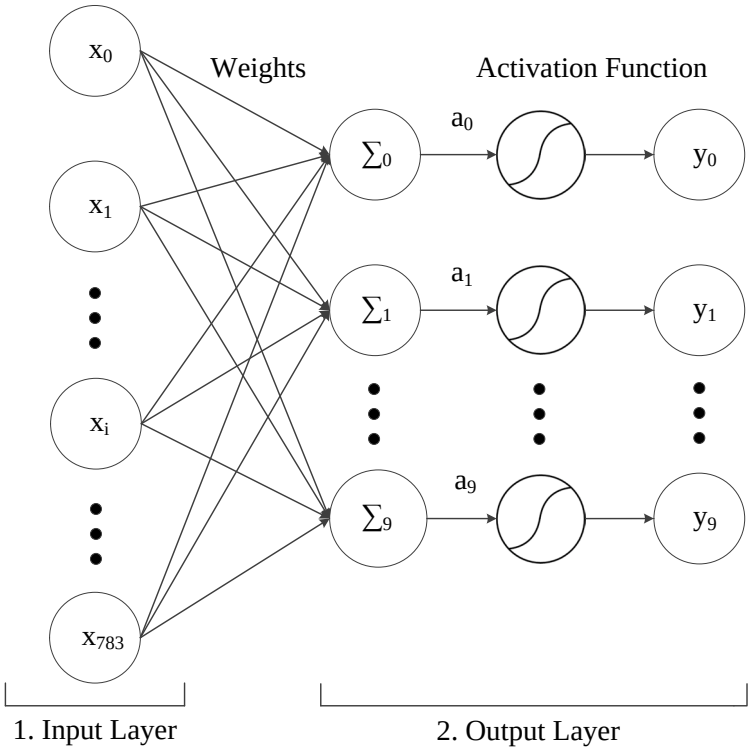

Figure 2. 2-layer Multilayer Perceptron Network.

the synaptic weights and biases of each neuron so that the network can classify the MNIST characters with high probability. After back propagation is used to train the network to correctly classify digits, and after verifying the algorithm using the validation dataset, then forward propagation is used to classify new data.

2.1.3. FPGA System

This section describes the MLP network we built using SystemVerilog [17] on a Cyclone IVE FPGA. The system diagram of the MLP network is shown in **Figure 3** and is described in detail in this section. We implemented the network using 4-, 5-, 6- and 8-bit precision for all inputs, outputs, and calculations.

The MLP hardware system (see **Figure 3**) 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 2.1.4). As directed by the Controller module, the Computation Unit completes forward or backward propagation calculations, as described in Section 2.1.5.

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

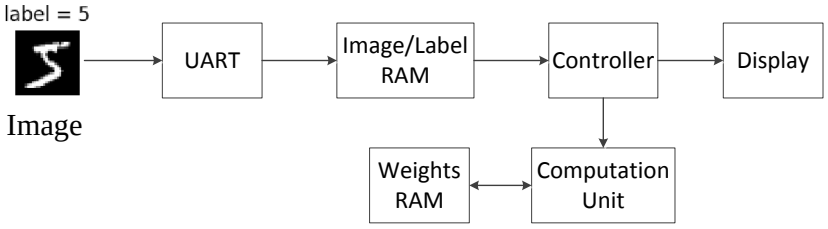

Figure 3. FPGA system architecture.

Weights RAM is initialized to hold 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 2.1.5. During this backward propagation process, the weights are updated after each training image is processed.

2.1.4. Controller

The Controller module manages the two main processes: forward and backward propagation. It also displays the target and calculated outputs. **Figure 4** 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 $(y_0, \cdots, y_9)$ 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 2.1.3). 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.

2.1.5. 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 (1)) and then passes these results through the activation function to produce the outputs (y_0 through y_9, see Equation (2)).

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

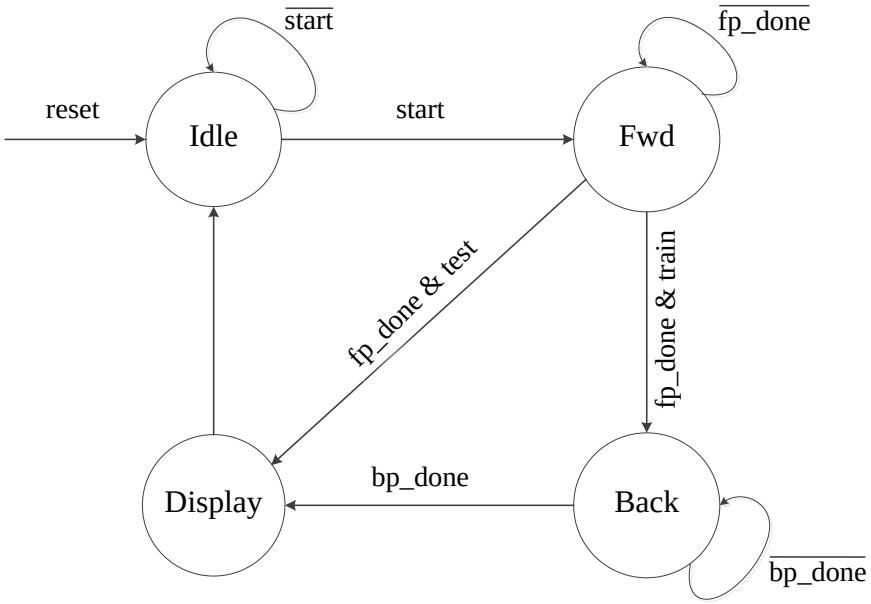

Figure 4. Controller finite state machine (FSM).

$$\begin{aligned} y_0 &= sigmoid\left(a_0\right) \\ &\vdots \\ y_9 &= sigmoid\left(a_9\right) \end{aligned} \tag{2}$$

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 back-propagation 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 (3). The values of the actual results, $target_i$, are obtained from the Label RAM.

$$error_i = target_i - y_i \tag{3}$$

We set the cost function for the training process as:

$$f\left(error_i\right) = \frac{1}{2}\left(error_i\right)^2 \tag{4}$$

This cost function of the errors is then used to update the weights in the Weights RAM. The derivative of the cost function with respect to output errors is:

$$\frac{\partial f_{cost\ function}}{\partial error_i} = error_i \tag{5}$$

However, the errors are first passed through the activation function and then multiplied by the input pixels before being used to update the weights. So, we need to calculate the derivative of the activation function $f\left(y\right)$:

$$\frac{\partial f_{sigmoid\ function}}{\partial y} = f\left(y\right) * \left(1 - f\left(y\right)\right) \tag{6}$$

The change in weights (that will be multiplied by the inputs before being added to the weights) is then calculated as given in Equation (7).

$$w_{change_i} = \frac{\partial f_{cost\ function}}{\partial error_i} * \frac{\partial f_{sigmoid\ function}}{\partial y_i} = error_i * f'(y_i) \tag{7}$$

This calculation can then be rewritten as:

$$w_{change_i} = error_i * f(y_i) * (1 - f(y_i)) \tag{8}$$

The MLP hardware system calculates the change for each weight by multiplying the input by the calculated weight change, as shown in Equation (9), 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 pixels in a single image, and x_j are the inputs to the MLP, that is, the value of each pixel in the training image.

$$w_{delta_{i,j}} = x_j * w_{change_i} \tag{9}$$

The weights used to process the next image are the updated weights, $w_{new_{i,j}}$, shown in Equation (10).

$$w_{new_{i,j}} = w_{old_{i,j}} + w_{delta_{i,j}} \tag{10}$$

The calculations for forward and backward propagation are summarized in Equation (11) and Equation (12), respectively.

$$\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_9 \end{bmatrix} = f\left[\begin{pmatrix} w_{0,0}x_0 + w_{0,1}x_1 + \cdots + w_{0,783}x_{783} \\ w_{1,0}x_0 + w_{1,1}x_1 + \cdots + w_{1,783}x_{783} \\ \vdots \\ w_{9,0}x_0 + w_{9,1}x_1 + \cdots + w_{9,783}x_{783} \end{pmatrix}\right] \tag{11}$$

$$w_{new_{i,j}} = w_{old_{i,j}} + x_j * (error_i) * f(y_i) * [1 - f(y_i)] \tag{12}$$

2.2. Three-Layer Offline Training MLP Neural Network

The 3-layer MLP network has higher prediction accuracy than the 2-layer MLP network described in Section 2.1. The additional layer in the 3-layer MLP network includes more trainable parameters. This added complexity enables increased prediction accuracy, but it also increases the circuit complexity and size. For this reason, we train the network in software and only implement the forward propagation path in hardware on an FPGA.

2.2.1. Network Architecture

The architecture of the 3-layer MLP network is shown in **Figure 5**. It is based on the structure of the 2-layer MLP network introduced in Section 2.1. In addition to input and output layers, the system includes one hidden layer, which contains 128 neurons. Because each MNIST image consists of 784 pixels, the input layer has 784 inputs, $x_0, x_1, \cdots, x_{783}$. The second layer, also called the hidden layer, contains 128 calculation nodes, or neurons, that sum the weighted inputs and send their outputs through the activation function to the output layer. Again, the 10 neurons in the output layer sum the weighted inputs from the hidden layer

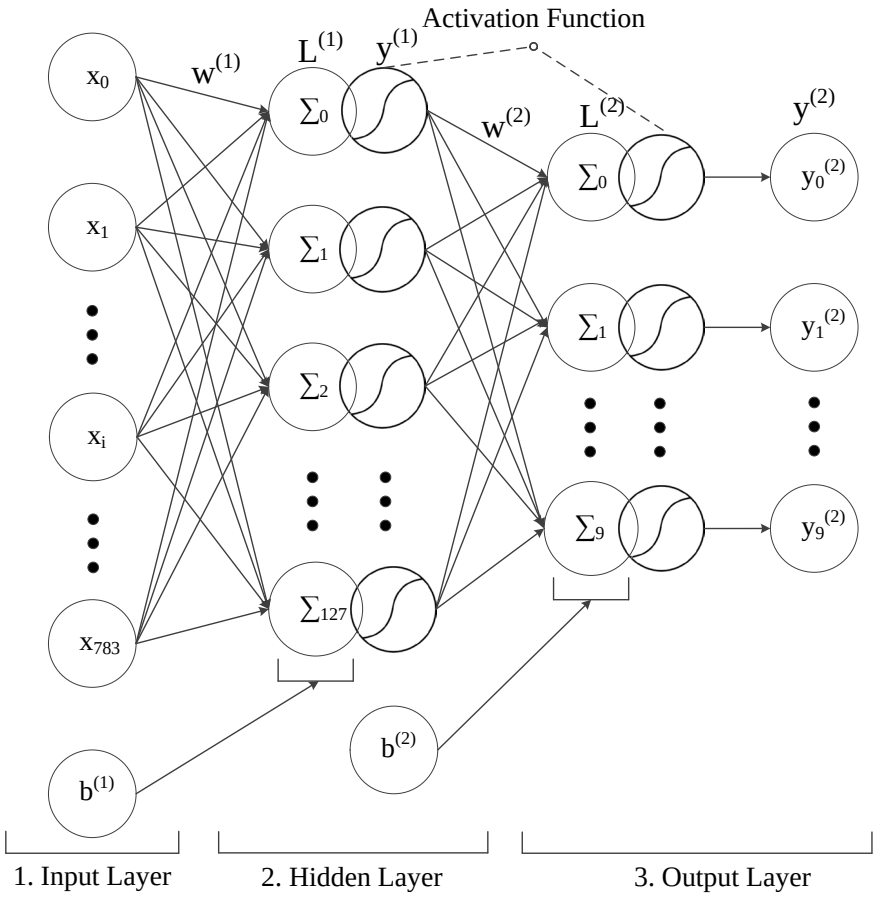

Figure 5. 3-layer Multilayer perceptron network.

and then pass them through the activation function to produce the outputs. With ten possible outputs (*i.e.*, digits 0 - 9), the network has ten neurons in the output layer, as shown in **Figure 5**. In addition to the weights between the two fully connected layers, we also add biases $b^{(1)}$ and $b^{(2)}$ for each non-output layer. As shown in equations (13) and (15), in the calculation of the following layer, weights control that how much any given layer input affects the output, and biases act as an offset.

2.2.2. FPGA System

The FPGA system architecture of the 3-layer neural network is similar to the architecture of the 2-layer neural network introduced in Section 2.1, but simpler.

As shown in **Figure 6**, we save weights obtained through training in software in the Weights RAM upon initialization of the system. After the system begins, it receives testing images and their labels from a PC through the UART. At the same time, the system reads the weights generated during training from the Weights RAM. After all the input data is loaded into memory on the FPGA, the data is passed through the layers and the predicted value is calculated. After each image is processed, the predicted digit and the image's label are shown on 7-segment displays on the FPGA board.

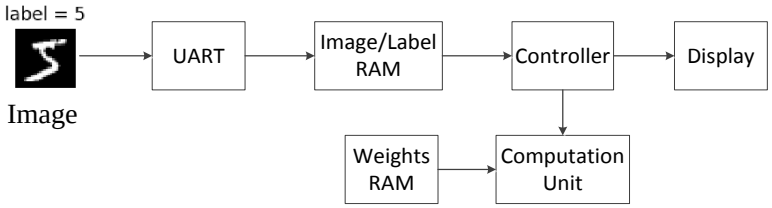

Figure 6. FPGA system architecture.

2.2.3. Controller

The controller module manages the dataflow during forward propagation and its main module is the finite state machine (FSM) defined by the state transition diagram in **Figure 7**. 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, the FSM displays the image label and predicted results $(y_0, y_1, \cdots, y_9)$ on the 7-segment displays and then returns to the Idle state to continue processing the next image.

2.2.4. Computation Unit

This section describes the calculations performed by the 3-layer MLP during both forward and backward propagation. Because the multiplication of layer input and weights, summation, and activation happen in each pair of fully connected layers, the 3-layer MLP's computation unit requires more calculations than the 2-layer MLP network. The 3-layer MLP network has 128 neurons in the hidden layer, and one bias ($b^{(1)}, b^{(2)}$) in each non-output layer. The computation unit calculates the weighted ($w_{0,0}^{(1)}$ through $w_{127,783}^{(1)}$) sums of each input ($L_0^{(1)}$ through $L_{127}^{(1)}$, see Equation (13)) and then passes these results through the activation function to produce the hidden layer ($y_0^{(1)}$ through $y_{127}^{(1)}$, see Equation (14)).

$$\begin{aligned} L_0^{(1)} &= w_{0,0}^{(1)} x_0 + w_{0,1}^{(1)} x_1 + \cdots + w_{0,783}^{(1)} x_{783} + b^{(1)} \\ &\vdots \\ L_{127}^{(1)} &= w_{127,0}^{(1)} x_0 + w_{127,1}^{(1)} x_1 + \cdots + w_{127,783}^{(1)} x_{783} + b^{(1)} \end{aligned} \tag{13}$$

$$\begin{aligned} y_0^{(1)} &= Sigmoid\left(L_0^{(1)}\right) \\ &\vdots \\ y_{127}^{(1)} &= Sigmoid\left(L_{127}^{(1)}\right) \end{aligned} \tag{14}$$

These results, $y_0^{(1)}$ through $y_{127}^{(1)}$, are then passed to the output layer and multiplied with another batch of weights ($w_{0,0}^{(2)}$ through $w_{9,127}^{(2)}$). Their sums ($L_0^{(2)}$ through $L_9^{(2)}$, see Equation (15)) are then passed through the activation function to produce the output layer ($y_0^{(2)}$ through $y_9^{(2)}$, see Equation (16)).

$$\begin{aligned} L_0^{(2)} &= w_{0,0}^{(2)} y_0^{(1)} + w_{0,1}^{(2)} y_1^{(1)} + \cdots + w_{0,127}^{(2)} y_{127}^{(1)} + b^{(2)} \\ &\vdots \\ L_9^{(2)} &= w_{9,0}^{(2)} y_0^{(1)} + w_{9,1}^{(2)} y_1^{(1)} + \cdots + w_{9,127}^{(2)} y_{127}^{(1)} + b^{(2)} \end{aligned} \tag{15}$$

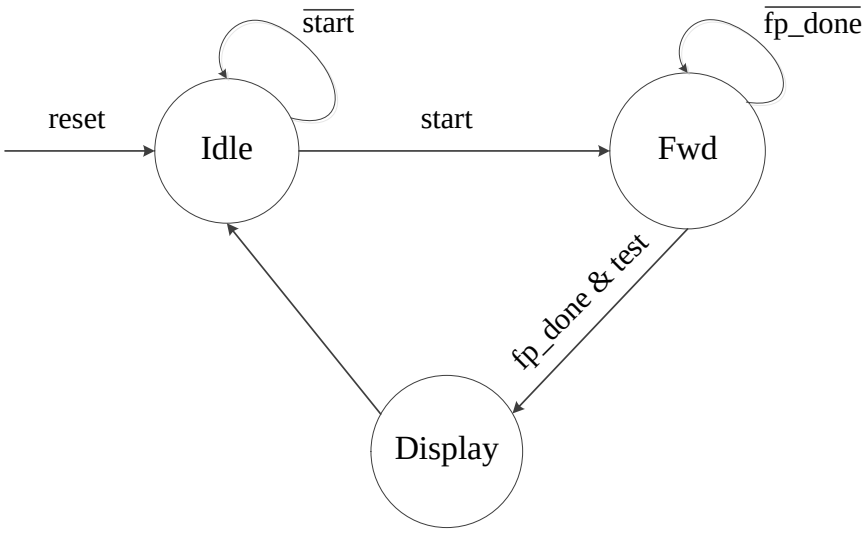

Figure 7. Controller finite state machine (FSM).

$$\begin{aligned} y_0^{(2)} &= Sigmoid\left(L_0^{(2)}\right) \\ &\vdots \\ y_9^{(2)} &= Sigmoid\left(L_9^{(2)}\right) \end{aligned} \tag{16}$$

For backward propagation, the error calculation for each digit i (where i is 0 to 9) is shown in Equation (17). The values of the actual results, $target_i$, are obtained from the Label RAM (as shown in **Figure 6**).

$$error_i^{(2)} = target_i - y_i^{(2)} \tag{17}$$

with the same theory introduced in A.5, the change of the hidden layer weights $w_{change_i}^{(2)}$ is then calculated as given in Equations (18) and (19).

$$w_{change_i}^{(2)} = \frac{\partial f_{cost\ function}}{\partial error_i^{(2)}} * \frac{\partial f_{sigmoid\ function}}{\partial y_i^{(2)}} = error_i^{(2)} * f'\left(y_i^{(2)}\right) \tag{18}$$

$$w_{change_i}^{(2)} = error_i^{(2)} * f\left(y_i^{(2)}\right) * \left(1 - f\left(y_i^{(2)}\right)\right) \tag{19}$$

Having calculated the layer weight changes and using the learning rate η, the new bias of the hidden layer, $b_{new}^{(2)}$, is calculated using Equation (20).

$$b_{new}^{(2)} = b_{old}^{(2)} + \eta * w_{change_i}^{(2)} \tag{20}$$

According to the chain rule, to back-propagate $w_{change_i}^{(2)}$ to the hidden layer weights, we need to multiply the derivative of the weights in Equation (15). In Equation (15), the hidden layer weights are multiplied with hidden layer inputs, therefore, we need to multiply the weight changes by the inputs of the hidden layer $y_j^{(1)}$, in order to get $w_{delta_{i,j}}^{(2)}$ (see Equation (21)).

$$w_{delta_{i,j}}^{(2)} = w_{change_i}^{(2)} * y_j^{(1)} \tag{21}$$

Then we can produce new hidden layer weights by adding the multiplication of weight deltas and learning rate (see Equation (22)).

$$w_{new_{i,j}}^{(2)} = w_{old_{i,j}}^{(2)} + \eta * w_{delta_{i,j}}^{(2)} \tag{22}$$

Similar to the error calculation in the output layer, we must also calculate the errors for the hidden layer neurons. But unlike the output layer, we cannot calculate these errors directly because we do not have a target, so we back-propagate them from the output layer. This is done by taking the errors from the output neurons and running them back through the weights to produce the hidden layer errors (see Equation (23), where m is 0 to 9 for each of the digits).

$$w_{change_m}^{(1)} = \left(error_i^{(2)} * f\left(y_i^{(2)}\right) * \left(1 - f\left(y_i^{(2)}\right)\right)\right) * w_{old_{i,j}}^{(2)} * f\left(y_m^{(1)}\right) * \left(1 - f\left(y_m^{(1)}\right)\right) \tag{23}$$

Having obtained the weight changes for $w^{(1)}$, we calculate the new bias $b_{new}^{(1)}$ and weights $w_{delta_{m,n}}^{(1)}$ for the input layer using the same method described for the hidden layer – see Equations (24) to (26), where n is 0 to 783 for each of the input pixels.

$$b_{new}^{(1)} = b_{old}^{(1)} + \eta * w_{change_i}^{(1)} \tag{24}$$

$$w_{delta_{m,n}}^{(1)} = w_{change_m}^{(1)} * x_n \tag{25}$$

$$w_{new_{m,n}}^{(1)} = w_{old_{m,n}}^{(1)} + \eta * w_{delta_{m,n}}^{(1)} \tag{26}$$

These biases and weights are updated each time the system processes a new training image.

2.3. Activation Function

Activation functions in neural networks produce the decision boundary of the neuron output, which decide whether the neuron should be activated or not. So an activation function inhibits low inputs and accentuates high inputs. Commonly used activation functions include sigmoid, ReLU, Softmas and tanh functions. In this way, activation functions reflect neuron behavior, where neurons require an input above some threshold to activate.

2.3.1. Sigmoid Function

A popular activation function is the sigmoid function shown in Equation (27) and Figure 8. As with other commonly used activation functions, such as Softmax and tangent functions, it is highly compute intensive. Along with the high prediction accuracy when using sigmoid function as the activation in MLP neural networks, another consideration is that it can be easily differentiated (see Equation (6)), which is required for the training process.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{27}$$

Figure 9 shows the sigmoid function with 9 bits of precision. The inputs of the function vary from −256 to 255 instead of 0 to 1 as would be the case with 1-bit precision. An input of −256 indicates complete inhibition and an input of 255 indicates full activation. So, for example, an input of −256 to the sigmoid

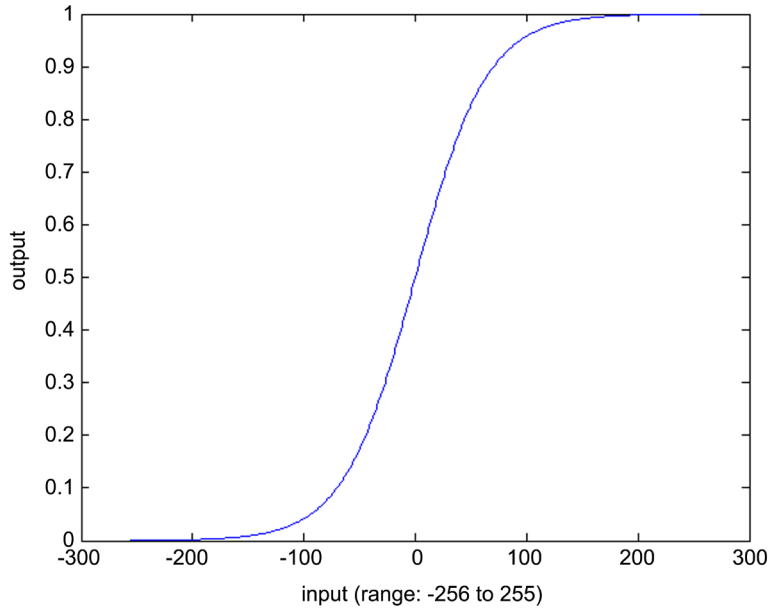

Figure 8. Sigmoid function.

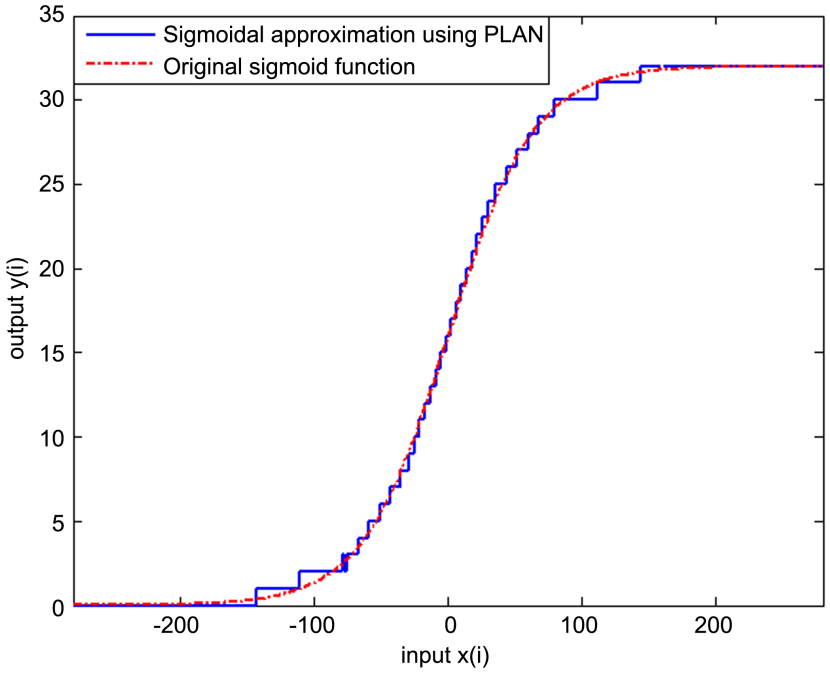

Figure 9. Sigmoid function and its approximation using PLAN.

function would result in an output very close to 0; an input of 0 would result in an output of 16, which means approximately 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.

Table 1 shows the output y as a function of the input a given several input

Table 1. Implementation of PLAN [18].

Output: $y = F(a)$	Input Value
$y' = 32$	$\|a\| \geq 160$
$y' = 0.03125^{a} * a + 27$	$76 \leq \|a\| < 160$
$y' = 0.125^{b} * a + 20$	$32 \leq \|a\| < 76$
$y' = 0.25^{c} * a + 16$	$0 \leq \|a\| < 32$
$y = y'$	$a \geq 0$
$y = 32 - y'$	$a < 0$

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

ranges. So, for example, if the input is 16, the output would be 20. Thus, the PLAN approximation replaces the exponential and division operations (see Equation (27)) with multiplication and addition.

2.3.2. D-ReLU Function

This section introduces our D-ReLU activation function and contrasts it with the most common activation functions used in MLP networks: the ReLU, leaky ReLU, parametric ReLU (P-ReLU), and Randomized Leaky ReLU functions.

The conventional ReLU function [3], which is shown in Equation (28) and **Figure 10**, is a piecewise function. It keeps the positive values unchanged, and outputs zero for negative inputs. Depending on the range of inputs, this behavior can be problematic. If most of the numbers in a batch are positive, too much redundant information is retained and this results in wider bit-width calculations or overflow for proceeding calculations. On the other hand, if most of the numbers are negative, zero is output for them and too much useful information is discarded. This would prohibit the network from learning and potentially even cause the network's death.

$$\begin{aligned} y &= 0, \ \text{if} \ x < 0 \\ y &= x, \ \text{if} \ x \geq 0 \end{aligned} \tag{28}$$

To address the problem of losing too much information, a few prior papers [5] [6] suggested modified ReLU functions, particularly the Leaky ReLU, Parametric ReLU, and Randomized Leaky ReLU functions. As shown in **Figure 11** and **Figure 12**. All of these modified ReLU functions tried to fix the negative input case. When the input is negative, instead of outputting zero, they would output the multiplication of the input with a smaller parameter. The difference among them is, this parameter could be fixed (Leaky ReLU), learnable (Parametric ReLU), or random (Randomized Leaky ReLU). This parameter was fixed in the testing stage in all three of these modified ReLU functions.

To alleviate the disadvantages in conventional ReLU and modified ReLU functions, we propose the dynamic ReLU (D-ReLU) function, with a dynamic threshold that changes in real-time. The two types of D-ReLU functions are the 1) middle D-ReLU—where the threshold is the middle of the range of numbers in each layer input; and 2) mean D-ReLU—where the threshold is the average of

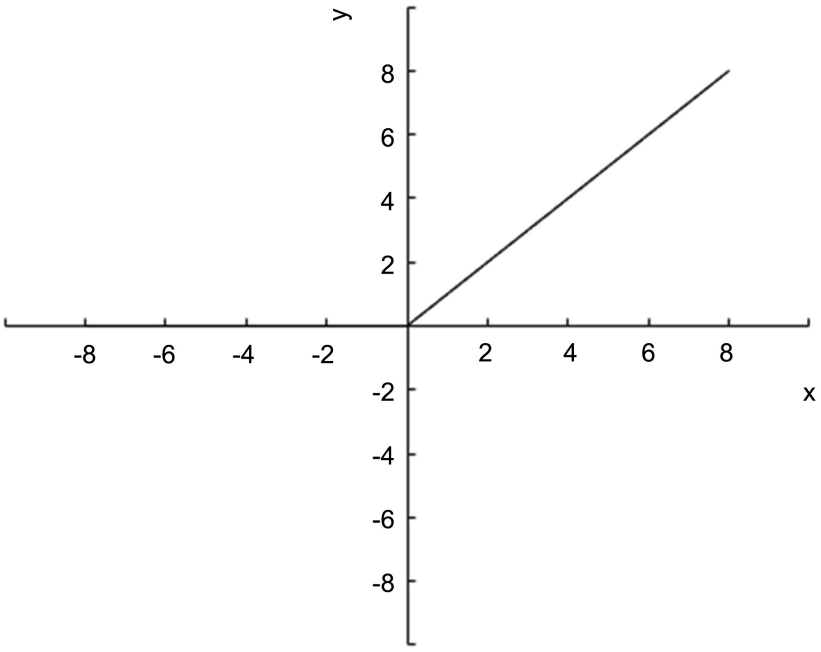

Figure 10. Conventional ReLU function.

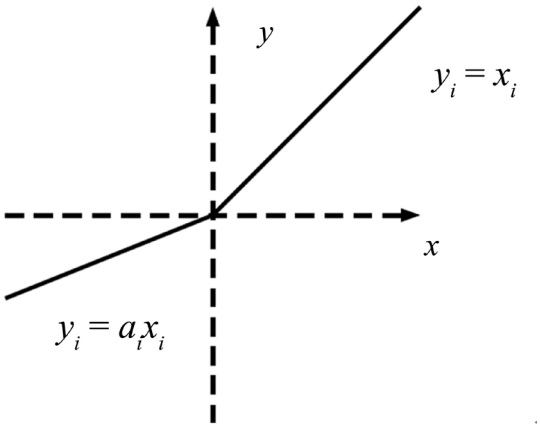

Figure 11. Leaky ReLU/Parametric ReLU function [5].

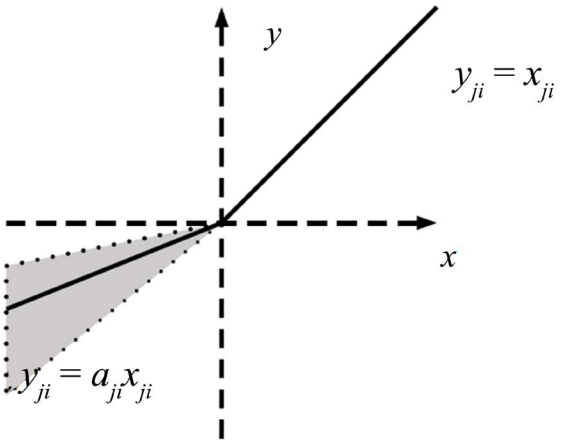

Figure 12. Randomized Leaky ReLU function [6].

the numbers in each layer input. For example, suppose the value range of the inputs is from −4 to 8, the calculated threshold would be $\frac{-4+8}{2}=2$ for the middle D-ReLU; The mean D-ReLU, on the other hand, uses the mean value of all of the inputs. In this way, the system could discard redundant information and preserve useful information dynamically, based on the pixel values of each image. Positive and negative threshold ReLU functions are shown in Figure 13.

We replace the sigmoid function from Section 2 with our novel D-ReLU function. During forward propagation, the threshold of the D-ReLU function changes dynamically. Before activating the outputs of the hidden layer and of the output layer, the D-ReLU function calculates the range of activation function inputs and sets the middle or mean value of the inputs as the current threshold. Each layer shares the same threshold at any given time, and this threshold changes when the network processes a new image—during training, validation, or testing.

2.3.3. Backward Propagation of the D-ReLU Function

While the calculations of backward propagation in the D-ReLU function are similar to those described for the sigmoid function in Section 2.2.4, in many cases the calculations are simpler, as described here.

Because the D-ReLU function is a piecewise function, we need to find the derivative of its two segments. As shown in Figure 13, when the input is smaller than the threshold, the output would always be the threshold value. So in this case, the derivative is 0. When the input is equal to or larger than the threshold, the slope of the function is 1, *i.e.*, the derivative is 1. In this way, the backward propagation calculations of the network change from those in Equation (18) and Equation (19) for the sigmoid function to Equation (29) and Equation (30) for the D-ReLU function.

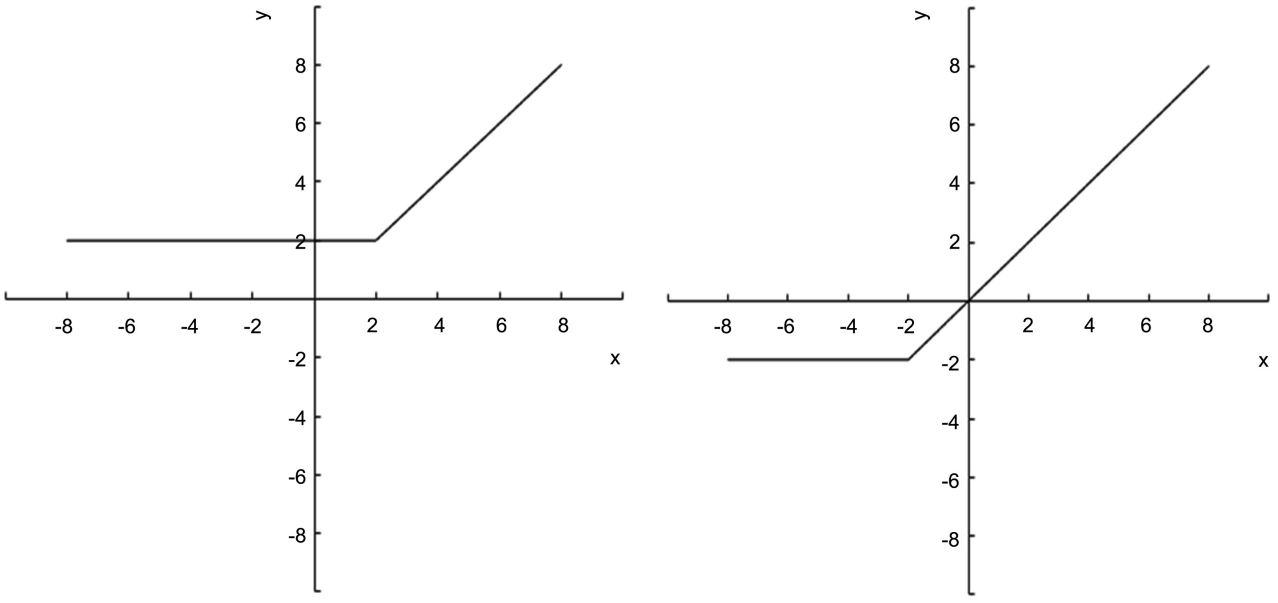

Figure 13. ReLU function with positive (left) and negative (right) thresholds.

$$f'(y) = 0, \text{ if } y < threshold$$
$$f'(y) = 1, \text{ if } y \geq threshold \quad (29)$$

$$w^{(2)}_{change_i} = 0, \text{ if } y_i < threshold$$
$$w^{(2)}_{change_i} = error^{(2)}_i, \text{ if } y_i \geq threshold \quad (30)$$

And the $w^{(1)}_{change}$ from Equation (23) becomes the much simpler Equation (31).

$$w^{(1)}_{change_m} = 0, \text{ if } y_i \text{ or } y_m < threshold$$
$$w^{(1)}_{change_m} = error^{(2)}_i * w^{(2)}_{old_{i,j}}, \text{ if } y_i \text{ or } y_m \geq threshold \quad (31)$$

As shown above, using the D-ReLU derivative during back propagation, instead of the more complex sigmoid derivative, means that we multiply with either 0 or 1. Thus, using the D-ReLU simplifies the calculations and reduces the required hardware and calculation time.

3. Results and Discussion

The experimental results described in this section show that neural networks implemented on an FPGA result in lower execution time and lower clock frequency without loss in prediction accuracy, which also indicates potential power saving. Moreover, by comparing different bit-widths of 2-layer MLP networks on FPGA, we can obtain different hardware resource saving solutions. At last, by applying the D-ReLU function, we can further improve these networks' performance on an FPGA.

3.1. Two-Layer Online Training MLP Neural Network

This section compares the prediction accuracy and execution time of the software solution with our proposed 2-layer Online Training MLP Neural Network and discusses the FPGA area requirements with varying bits of precision.

3.1.1. Prediction Accuracy

The prediction accuracy of the 4-, 5-, 6- and 8-bit hardware designs vary from 73 - 89% as compared to the 32-bit precision software implementation that achieves a prediction accuracy of 68-89%, depending on the number of training images. Figure 14 summarizes these results. The horizontal axis shows the number of training images (up to 55,000), and the vertical axis represents the prediction accuracy, the percentage of correct predictions using 10,000 test images. The 8-bit FPGA solution's prediction 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 requires 40,000 training images to converge.

Further reducing the bits of precision results in only 6% - 11% drops in prediction accuracy (when using 55,000 training images), as shown in Figure 14 and summarized in Table 2. Compared with the 8-bit hardware solution, the

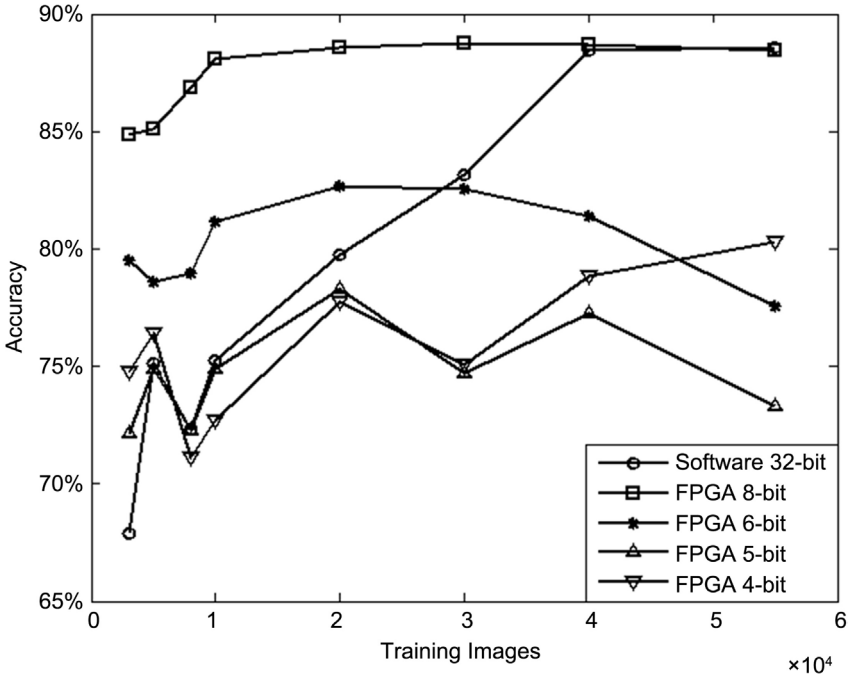

Figure 14. Software solution prediction accuracy vs. FPGA solutions prediction accuracy.

Table 2. Prediction accuracy.

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

6-bit solution's prediction 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 a prediction accuracy decrease of only 9%. Table 2 shows that accuracy is not affected when decreasing precision from 32 to 8 bits, but additional decreases in precision cost approximately 4% in prediction accuracy per bit decrease in precision.

3.1.2. Performance

The performance differences, as measured by execution time in hardware or software, are summarized in Table 3. 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 3. 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 the software implementation 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 × 3.7 s/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 when compared to the software implementation.

3.1.3. FPGA Area

The FPGA area requirements for the MLP hardware design using 4, 5, 6, and 8 bits of precision are 20 - 34 k logic elements (LEs). **Table 4** summarizes the FPGA 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.

Figure 15 shows the FPGA 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, and its accuracy only decreases by 6% (from 89% to 83%), as shown in **Figure 15**. The 5-bit solution saves another 11% of the logic elements compared with the 6-bit solution but only has a 5% prediction accuracy drop (from 83% to 78%). The 4-bit design requires 6% fewer logic elements than the 5-bit design while maintaining prediction accuracies similar to that design. **Figure 15** shows the actual prediction 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 prediction accuracy drops 11% (from 89% to 78%), and the area decreases by 41%. So area decreases at approximately 4% per percent decrease in prediction accuracy.

Figure 16 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.

However, in many cases, online training is not always necessary. For example, for image recognition on mobile phone, we just need to train the network on a server or cloud and the mobile phone need only perform the testing function, thus reducing the hardware and power requirements of the mobile phone. **Table 5**

Table 4. Fpga area of training and testing.

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

[e]Intel's Cyclone IVE FPGA.

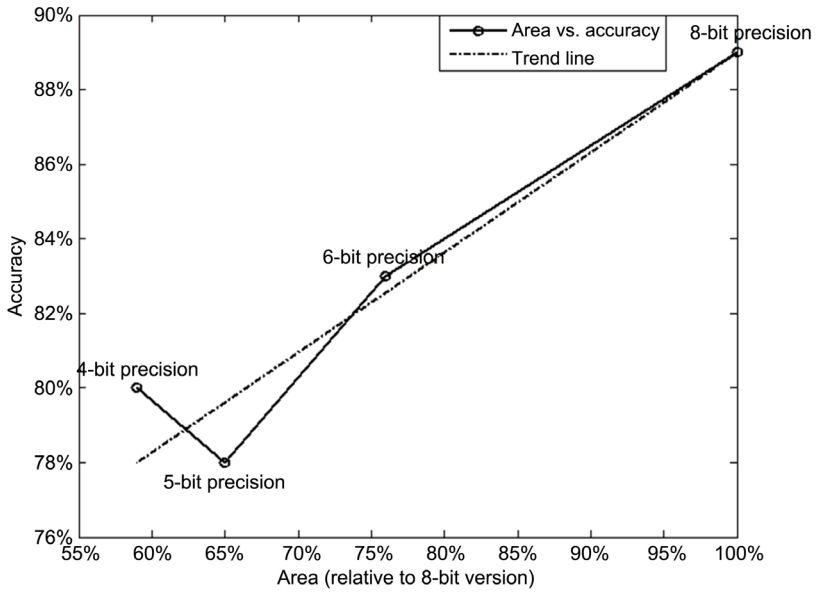

Figure 15. Prediction accuracy (percentage of correct predictions) vs. FPGA area (relative to 8-bit version) for the 4-, 5-, 6-, and 8-bit MLP FPGA designs.

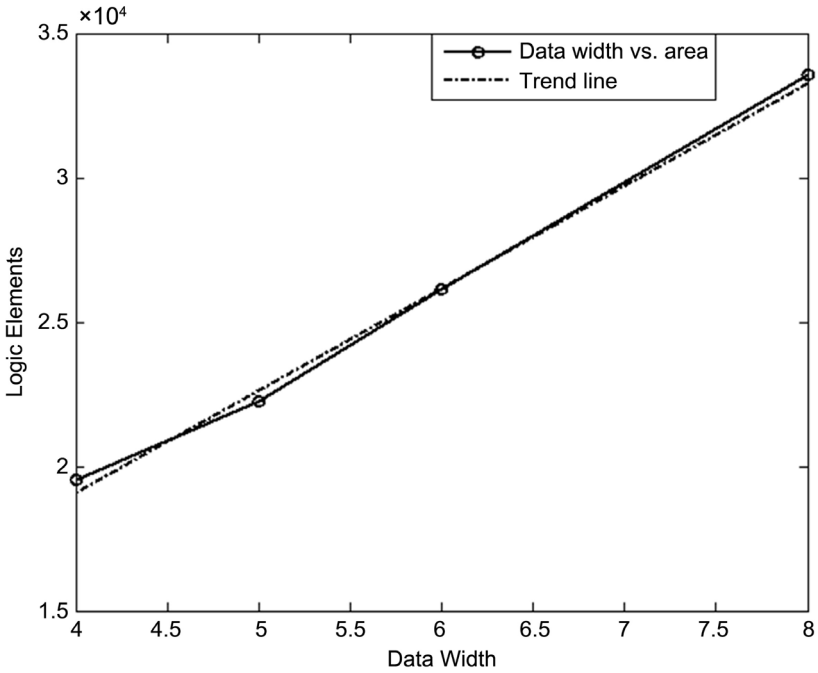

Figure 16. Data width vs. FPGA area (number of FPGA logic elements).

Table 5. FPGA area of testing (forward propagation).

FPGA Solution	Logic elements (% use[f])
8-bit	27 k (24%)
6-bit	21 k (18%)
5-bit	17 k (15%)
4-bit	14 k (12%)

[f]Intel's Cyclone IVE FPGA.

shows the FPGA area requirement of the 2-layer MLP network when working on different data widths after removing the training hardware. On average, including only the testing hardware requires 20% less hardware than including both testing and training hardware.

Additional advantages of offline training are described in Section 3.2.

3.2. Two- and Three-Layer Offline Training MLP Neural Networks

The advantages of the D-ReLU function over the sigmoid function are that it has lower execution time in software, requires less area in hardware, and is potentially more power-efficient with little loss in prediction accuracy, as described in detail in this section.

3.2.1. Prediction Accuracy

This section compares the prediction accuracy of networks using the sigmoid or D-ReLU activation function. Note that the designs are all trained in software and then tested in either software or hardware. The software designs use 32-bit floating point calculations, and the FPGA implementations use 8-bit fixed-point calculations. The system preprocesses the input image data and parameters as 8-bit integers before sending them to the FPGA.

As shown in **Table 6**, the 3-layer MLP network with the sigmoid activation function has higher prediction accuracy than the 2-layer MLP network we built in Section 2.1. The prediction accuracy increases by 6.5% by adding one more layer, adjusting the biases associated with each non-output layer, and adjusting the learning rate η. The prediction accuracy of the 3-layer MLP network with a sigmoid activation function achieves 95.5% prediction accuracy, which is 6.5% higher than the 2-layer MLP network in Section 2.1.

When working in software, the three 2-layer MLP networks achieve similar accuracies (~90%) across platforms and type of activation function. The accuracies of the 3-layer MLP network using the middle D-ReLU function achieves 92.9% prediction accuracy, only 2.6% less than the 95.5% of the network using the sigmoid activation function. The accuracies of the 3-layer MLP network with mean D-ReLU activation function are even slightly higher than the networks with sigmoid activation function.

When working on an FPGA, the prediction accuracies are not affected by the data and parameter preprocessing. The 2-layer MLP network has the same

Table 6. Prediction accuracy.

	2-layer MLP	3-layer MLP
w/ sigmoid in software	90.2%	95.5%
w/ middle D-ReLU in software	90.0%	92.9%
w/ mean D-ReLU in software	90.0%	95.8%
w/ sigmoid in hardware (on an FPGA)	90.3%	95.6%
w/ middle D-ReLU in hardware (on an FPGA)	89.6%	92.7%
w/ mean D-ReLU in hardware (on an FPGA)	89.6%	96.0%

prediction accuracy (~90%) whether implemented in software or built using an FPGA. The 3-layer MLP networks achieve similar prediction accuracies for both hardware and software implementations: ~95% when using the sigmoid activation function and ~93% and ~96% when using the D-ReLU activation function.

3.2.2. Performance

This section compares the execution time and power consumption of the MLP networks across activation functions (sigmoid vs. D-ReLU) and across platforms (software vs. hardware).

First, we consider only the software implementations of the networks. As expected, networks using the D-ReLU activation function have faster execution times than those using sigmoid activation functions. As discussed in Section 2.3, the calculations of the D-ReLU function are much simpler than the sigmoid function, so it requires less computation, and thus less execution time. As shown in Table 7 and Table 8, compared with the systems using the sigmoid activation functions, the 2-layer MLP network with the D-ReLU function executes 14% faster, and the 3-layer MLP network with the D-ReLU function is 57% faster. Note that the execution time accounts for forward propagation time only because training is completed in software for all systems.

When built in hardware on an FPGA, the activation function used does not affect execution time, but all four MLP networks show potential for lower power consumption in the FPGA design over the software implementation due to faster execution times and lower system clock frequency. As shown in Table 7 and Table 8, the execution time of the 2-layer MLP network is 145 and 125 times less, respectively when using sigmoid and D-ReLU functions, than their software solutions. Similarly, the execution time of the 3-layer MLP network is 66 and 29 times less, respectively when using sigmoid and D-ReLU functions, than their software solutions. Moreover, the 2-layer MLP networks' FPGA clock frequency is 14.4 times slower than the software frequency, and the 3-layer MLP networks' FPGA clock frequency is 60 times slower than the software clock frequency. Using the relationship that power is proportional to operating frequency, hardware designs require about 14.4× and 60× less power due to clock speed alone. When combining the effects of both decreased execution time and decreased clock frequency of the FPGA design over the software implementation, these four MLP

networks offer a potential of more than a 1700× decrease in power consumption when working on an FPGA. Thus, the FPGA hardware designs offer higher performance or lower power alternatives when compared to software implementations.

3.2.3. FPGA Area

The D-ReLU activation function requires less area and fewer computation cycles than the sigmoid activation function when built on an FPGA. In order to highlight the contribution of the two kinds of activation functions, we compare the FPGA area of each network's computation unit only. As shown in Table 9, 2- and 3-layer MLP networks using the D-ReLU function use 41% and 24% less area, respectively, compared with networks using the sigmoid activation function. Moreover, the D-ReLU function uses two fewer clock cycles to complete the calculations in each layer than the sigmoid activation function.

By reducing the data width from 8 bits to 6, 5, and 4 bits as we did in Section 2.1, we can also get a similar trend of trading off prediction accuracy for FPGA area. As shown in Table 10, if we reduce the precision from 8 to 4 bits for the two types of 3-layer neural networks, the prediction accuracy of the networks drop by only 2.7% and 4.8% respectively. At the same time, their FPGA area requirements reduced by 28% and 9% respectively.

Figure 17 and Figure 18 show the data width versus area and a trend line for

Table 7. 2-layer MLP execution time and power.

	Clock Frequency	Execution Time	Power Saving
w/sigmoid in SW	3.6 GHz	2.9 seconds	1.0×
w/D-ReLU in SW	3.6 GHz	2.5 seconds	1.2×
w/sigmoid in HW	250 MHz	0.02 seconds	2088×
w/D-ReLU in HW	250 MHz	0.02 seconds	1800×

Table 8. 3-layer MLP execution time and power saving.

	Clock Frequency	Execution Time	Power Saving
w/sigmoid in SW	3.6 GHz	9.9 seconds	1.0×
w/D-ReLU in SW	3.6 GHz	4.3 seconds	2.3×
w/sigmoid in HW	60 MHz	0.15 seconds	3960×
w/D-ReLU in HW	60 MHz	0.15 seconds	1740×

Table 9. FPGA area.

FPGA Solution	Logic Elements[g]	
	2-layer MLP	3-layer MLP
w/sigmoid	308	564
w/D-ReLU	183	428

[g]Intel's Cyclone IVE FPGA.

Table 10. FPGA area.

FPGA Solution	Bit Precision	Logic Elements[h]	Prediction accuracy
3-layer network w/sigmoid	8-bit	564	95.6%
	6-bit	495	95.1%
	5-bit	459	94.7%
	4-bit	404	92.9%
3-layer network w/D-ReLU	8-bit	428	92.7%
	6-bit	416	90.3%
	5-bit	407	88.6%
	4-bit	386	87.9%

[h]Intel's Cyclone IVE FPGA.

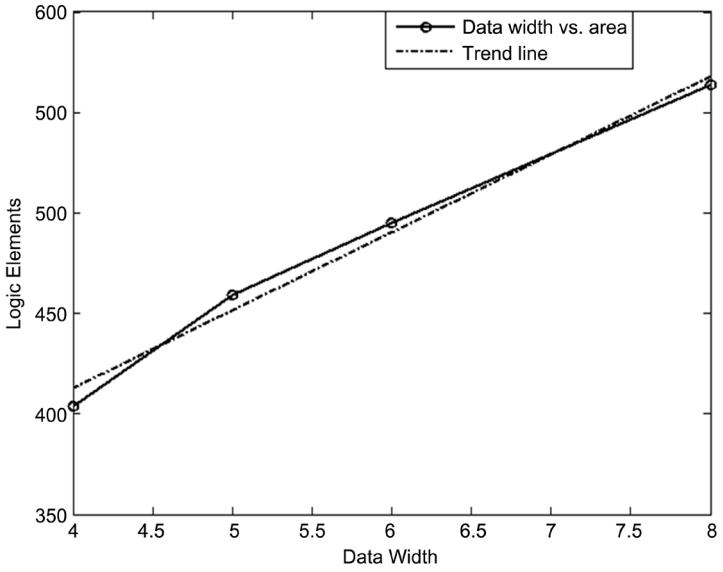

Figure 17. Data width vs. FPGA area of 3-layer network with sigmoid.

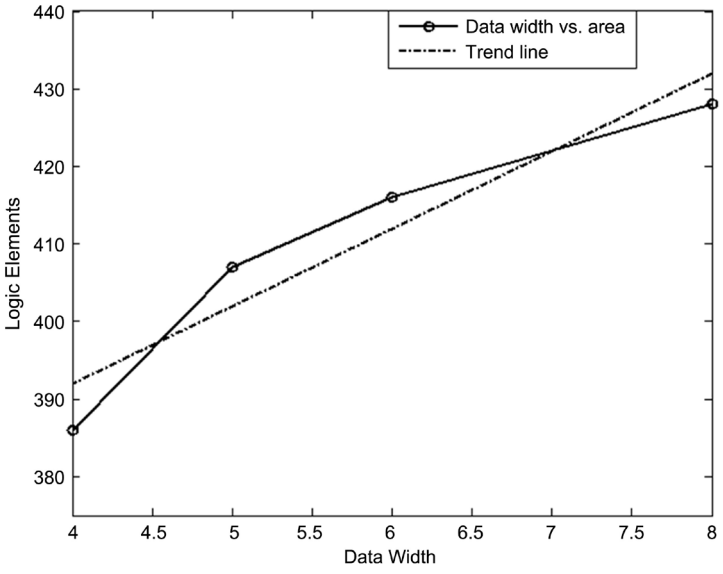

Figure 18. Data width vs. FPGA area of 3-layer network with D-ReLU.

each network implemented on an FPGA. As depicted by the trend lines, the FPGA area of the 3-layer network with sigmoid function grows by ~9% per bit of precision, and the FPGA area of the 3-layer network with the D-ReLU function grows by ~3% per bit of precision.

4. Conclusions

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 of the 2-layer online training MLP neural network 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 144 times, at no cost to prediction accuracy, as compared to the software solution. Furthermore, a reduction in precision from 32 to 8 bits results in no decrease in prediction accuracy. Additional reductions in precision below 8 bits result in only small reductions in prediction accuracy (4% prediction accuracy reduction per bit of reduced precision), moderate area decreases (10% decreased area per bit of precision), and a resulting area decrease that falls off more quickly than the decrease in prediction accuracy (4% decrease in area per percent decrease in prediction accuracy).

The D-ReLU activation function proposed in this paper offers a more flexible and accurate algorithm than the traditional ReLU function. It also results in a faster, more power-efficient design when compared to the software implementation without incurring loss in prediction accuracy. Compared with networks using sigmoid activation functions, networks using the proposed D-ReLU activation function are 57% faster during the testing phase and use 41% less FPGA area. Furthermore, if we reduce the bit precision of the 3-layer neural networks, their prediction accuracies drop by only 2.7% and 4.8% with 28% and 9% FPGA area savings. Moreover, because they operate at a lower clock frequency and require less execution time, FPGA solutions of MLP networks offer a low power alternative to traditional software methods. In this paper, MLP networks implemented on an FPGA offered the potential of being 1700× more power efficient than comparable software solutions.

In all, FPGA solutions of neural networks provide fast, power efficient, low-cost and portable alternatives to the software solutions, which could be integrated into low-power portable devices in the future. Moreover, FPGA solutions offer flexible, customizable, and secure solutions that protect privacy, which could be integrated into customized FPGA/ASIC SoC systems with ease.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., *et al.* (2017) Mastering the Game of Go without Human Knowledge. *Nature*, **550**, 354-359. https://doi.org/10.1038/nature24270

[2] Jouppi, N.P., Borchers, A., Boyle, R., Cantin, P., Chao, C., Clark, C., Coriell, J., *et al.* (2017) In-Datacenter Performance Analysis of a Tensor Processing Unit. *Proceedings of the* 44*th Annual International Symposium on Computer Architecture*, Toronto, June, 2017, 1-12. https://doi.org/10.1145/3079856.3080246

[3] Wikipedia (2020) Rectifier (Neural Networks). https://en.wikipedia.org/wiki/Rectifier_(neural_networks)

[4] Krizhevsky, A., Sutskever, I. and Hinton, G.E. (2012) ImageNet Classification with Deep Convolutional Neural Networks. *Communications of the ACM*, **60**, 84-90.

[5] Maas, A.L., Hannun, A.Y. and Ng, A.Y. (2013) Rectifier Nonlinearities Improve Neural Network Acoustic Models. *Proceedings of the* 30*th International Conference on Machine Learning*, Atlanta, 16-21 June 2013, 6.

[6] He, K.M., Zhang, X.Y., Ren, S.Q. and Sun, J. (2015) Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. 2015 *IEEE International Conference on Computer Vision*, Santiago, 7-13 December 2015, 1026-1034. https://doi.org/10.1109/ICCV.2015.123

[7] Zhang, C., Li, P., Sun, G.Y., Guan, Y.J., Xiao, B.J. and Cong, J. (2015) Optimizing FPGA-Based Accelerator Design for Deep Convolutional Neural Networks. *Proceedings of the* 2015 *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Monterey, February 2015, 161-170. https://doi.org/10.1145/2684746.2689060

[8] Qiu, J.T., Wang, J., Yao, S., Guo, K.Y., Li, B.X., Zhou, E.J., Yu, J.C., Tang, T.Q., *et al.* (2016) Going Deeper with Embedded FPGA Platform for Convolutional Neural Network. 2016 *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Monterey, February 2016, 26-35. https://doi.org/10.1145/2847263.2847265

[9] Li, H.M., Fan, X.T., Jiao, L., Cao, W., Zhou, X.G. and Wang, L.L. (2016) A High Performance FPGA-Based Accelerator for Large-Scale Convolutional Neural Networks. 2016 26*th International Conference on Field Programmable Logic and Applications*, Lausanne, 29 August-2 September 2016, 1-9. https://doi.org/10.1109/FPL.2016.7577308

[10] Lu, L.Q., Liang, Y., Xiao, Q.C. and Yan, S.G. (2017) Evaluating Fast Algorithms for Convolutional Neural Networks on FPGAs. *IEEE* 25*th Annual International Symposium on Field-Programmable Custom Computing Machines*, Napa, 30 April-2 May 2017, 101-108. https://doi.org/10.1109/FCCM.2017.64

[11] Han, S., Mao, H. and Dally, W.J. (2015) Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *International Conference on Learning Representations* 2016, San Juan, 2-4 May 2016, 1-14.

[12] Bouvett, E., Casha, O., Grech, I., Cutajar, M., Gatt, E. and Micallef, J. (2012) An FPGA Embedded System Architecture for Handwritten Symbol Recognition. 2012 16*th IEEE Mediterranean Electrotechnical Conference*, Yasmine, 25-28 March 2012, 653-656. https://doi.org/10.1109/MELCON.2012.6196516

[13] Suyyagh, A. and Abandah, G. (2013) FPGA Parallel Recognition Engine for Handwritten Arabic Words. *Journal of Signal Processing Systems*, **78**, 163-170. https://doi.org/10.1007/s11265-013-0848-x

[14] Huynh, T. (2014) Design Space Exploration for a Single-FPGA Handwritten Digit Recognition System. *2014 IEEE* 5*th International Conference on Communications and Electronics*, Danang, 30 July-1 August 2014, 291-296. https://doi.org/10.1109/CCE.2014.6916717

[15] Moradi, M., Pourmina, M.A. and Razzazi, F. (2010) FPGA-Based Farsi Handwritten Digit Recognition System. *International Journal of Simulation*: *Systems*, *Science and Technology*, **11**, 17-22.

[16] Nahmias, M.A., Shastri, B.J., Tait, A.N. and Prucnal, P.R. (2013) A Leaky Integrate-and-Fire Laser Neuron for Ultrafast Cognitive Computing. *IEEE Journal on Selected Topics in Quantum Electronics*, **19**, 1-12. https://doi.org/10.1109/JSTQE.2013.2257700

[17] Kudrolli, K., Shah, S. and Park, D. (2015) Handwritten Digit Classification on FPGA.

[18] Amin, H., Curtis, K.M. and Hayes-Gill, B.R. (1997) Piecewise Linear Approximation Applied to Nonlinear Function of a Neural Network. *IEE Proceedings—Circuits*, *Devices and Systems*, **144**, 313-317. http://dx.doi.org/10.1049/ip-cds:19971587