Hardware-efficient on-line learning through pipelined truncated-error backpropagation in binary-state networks
Hesham Mostafa\textsuperscript{1}, Bruno Pedroni\textsuperscript{2}, Sadique Sheik\textsuperscript{3}, Gert Cauwenberghs\textsuperscript{1,2,3}
\textsuperscript{1}Institute for Neural Computation,\textsuperscript{2}Department of Bioengineering, \textsuperscript{3}BioCircuits Institute
UC San Diego, California, USA
E-mail: \{hmmostafa,bpedroni,ssheik,gert\}@ucsd.edu

Abstract
Artificial neural networks (ANNs) trained using backpropagation are powerful learning architectures that have achieved state-of-the-art performance in various benchmarks. Significant effort has been devoted to developing custom silicon devices to accelerate inference in ANNs. Accelerating the training phase, however, has attracted relatively little attention. In this paper, we describe a hardware-efficient on-line learning technique for feedforward multi-layer ANNs that is based on pipelined backpropagation. Learning is performed in parallel with inference in the forward pass, removing the need for an explicit backward pass and requiring no extra weight lookup. This saves 50\% of the weight lookups needed by a standard online implementation of backpropagation. By using binary state variables in the feedforward network and ternary errors in truncated-error backpropagation, the need for any multiplications in the forward and backward passes is removed, and memory requirements for the pipelining are drastically reduced. For proof-of-concept validation, we demonstrate on-line learning of MNIST handwritten digit classification on a Spartan 6 FPGA interfacing with an external 1Gb DDR2 DRAM, that shows small degradation in test error performance compared to an equivalently sized ANN trained off-line using standard back-propagation and exact errors. Our results highlight an attractive synergy between pipelined backpropagation and binary-state networks in substantially reducing computation and memory requirements, making pipelined on-line learning practical in deep networks.

1 Introduction
The immense success of artificial neural networks (ANNs) is largely due to the use of efficient training methods that can successfully update the network weights in order to minimize the training cost function \cite{1}. Backpropagation \cite{2}, or gradient descent in multi-layer networks, has become the training method of choice as it provides a conceptually clear approach to minimizing the cost function that works very well in practice. Training ANNs using backpropagation, however, is still a computationally demanding task as successful learning requires several presentations of the training data to allow the backpropagation algorithm to slowly adjust the network parameters. ANN accelerators developed for deployment in low-power systems therefore typically do not implement the lengthy and power-hungry training phase and only implement the computationally cheaper forward/inference pass \cite{3–8}. These ANN accelerators can thus only implement pre-trained networks with fixed parameters. While this approach is appropriate for ANNs that process data from sources whose statistics are known beforehand and from which large amounts of training data have been gathered in the past in order to pre-train the network, it is inappropriate in situations where the device has to interact with unexpected or new sources of data and has to build its own classification or inference model on the fly.

Typical approaches for implementing large-scale ANN accelerators with learning capabilities \cite{9,10} are based on ANNs with smooth activation functions and high precision weights and neuron values. This necessitates the use of Multiply and Accumulate (MAC) operations as well as hardware implementation of activation functions such as the hyperbolic tangent and logistic sigmoid. This makes the hardware implementation of the accelerator costly in terms of logic resources and memory. We make use of recent developments that show that ANNs with binary neurons nearly match the performance of ANNs with smooth activation functions \cite{11}. In a binary $(-1, +1)$ neural network, the implementation of the activation function reduces to a comparator and the forward pass involves no multiplications. However, the backward pass in which the error from the top layer is backpropagated to deeper layers still involves multiplications. To avoid these multiplications, we describe an approximation to the backpropagation algorithm that truncates the error signal to a ternary $(-1, 0, +1)$ variable. This yields a training algorithm that performs well in practice and that does not require any multiplications.

State of the art ANNs typically have millions of parameters. Optimizing the movement of these
parameters between memory and the computational elements is a key step to improve the power-efficiency and speed of ANN accelerators. This is especially true if the network parameters or weights are stored off-chip as off-chip memory traffic could easily become the bottleneck limiting the accelerator speed. A straightforward implementation of online backpropagation on custom hardware would typically need to look up each network weight twice, once during the forward pass and once during the backward pass. The backward pass lookup is needed so that the weight can be used to backpropagate the errors and for the updated weight then to be written back into memory. Pipelined backpropagation is a technique that can be used to eliminate the extra weight lookup in the backward pass [12]. In pipelined backpropagation, the backward learning pass used to push the errors from the network output to the network input and to update the weights is performed in parallel with the forward inference pass. This is achieved by maintaining a history of the network state and using this history to update the weights based on delayed errors. The length of this history, however, grows with the network depth. Previously proposed hardware implementations of pipelined backpropagation [13–15] therefore incur large memory overheads as the network depth increases. We show that the network history can be compactly represented in binary-state networks (BSNs) which drastically reduces the memory overhead needed to implement pipelined backpropagation, making it a viable option when training deep networks.

In section 2.1, we provide some background on the backpropagation algorithm and the architecture of BSNs. In section 2.2, we describe our version of pipelined backpropagation, the learning algorithm, and the hardware architecture implementing approximate pipelined backpropagation in BSNs. We present experimental results demonstrating the learning architecture embedded on an FPGA platform in section 3 and present our conclusions and directions for future work in section 4.

2 Materials and Methods

2.1 Background

2.1.1 Feedforward Networks and Backpropagation

Multi-layer fully-connected feedforward neural networks are typically arranged in layers. In a network with \( L \) hidden layers, the activation vector, \( h_i \), for hidden layer \( i \) is given by

\[
h_i = \sigma(W_i h_{i-1} + b_i) \quad i = 1, ..., L
\]

where \( W_i \) is the weight matrix connecting neurons in layer \( i-1 \) to neurons in layer \( i \) and \( b_i \) is the bias vector for layer \( i \). \( \sigma \) is a non-linear activation function that is applied element-wise on the argument vector. \( x \equiv h_0 \) is the input layer. The top layer in the network aggregates input from the last hidden layer to yield the network output. We use the vector \( z \) to denote the top layer activity. The forward pass computation from network input to network output for a network with two hidden layer is depicted along the upward arrow in Fig. 1.

Network training involves updating the weights \( W_i \) and biases \( b_i \) so as to minimize an error cost function \( E \). The cost function quantifies the error in the network output and it is minimized when the network output approaches the desired output for the presented input. This cost function is almost always differentiable with respect to the network output. To minimize the cost function, gradient descent could thus be applied to move the network parameters along the negative direction of the the cost function gradient with respect to the parameters. This yields the backpropagation algorithm which is illustrated in Fig. 1 for the two-hidden layer network. To simplify the figure, the biases have been omitted. A layer bias vector can be implemented by having an extra neuron in the layer below that has no input and whose output is fixed at 1.

During the backward pass depicted in Fig. 1, the error derivative at the top layer, \( \frac{dE}{dz} \), is pushed down through the network. The backward pass needs access to two sets of vectors: the activation vectors (\( x \), \( h_1 \), and \( h_2 \) in Fig. 1) and the vectors containing the derivative of the activation function in the hidden layers at the operating point of the network during the forward pass (\( \sigma'(W_1x) \) and \( \sigma'(W_2h_1) \) in Fig. 1). The first set of vectors is needed to update the weights through a cross product with the error vectors and the second set of vectors (the derivative vectors) are needed to push the error from the top layer down through the layer stack.
2.1.2 Binary-State Networks

One of the surprising properties of neural networks is the resiliency of the backpropagation procedure to various deviations from the ideal algorithm depicted in Fig. 1. One hardware-oriented modification of the ideal backpropagation algorithm aims at training networks with binary activation functions [11] where each neuron’s output is either \(-1\) or \(1\) (or, equivalently, either 0 or 1). This considerably simplifies the forward pass through the hidden layers as the multiplications are replaced by sign change circuits. The neuron’s activation function, \(\sigma_b\), is given by:

![Diagram of the backpropagation algorithm in a feedforward network with two hidden layers.](image)

\[
\sigma_b(x) = \begin{cases} 
1 & \text{if } x \geq 0 \\
-1 & \text{otherwise.}
\end{cases}
\]  

The derivative of \(\sigma_b\), however, is zero almost everywhere which would stop errors from backpropagating. A virtual derivative which was found to work well in practice is the saturating straight-through estimator [11, 16]:

\[
\sigma'_b(x) = \begin{cases} 
1 & \text{if } -1 \leq x \leq 1 \\
0 & \text{otherwise.}
\end{cases}
\]

If the input layer activity is also binarized, then the entire forward pass from input layer to output layer is free from multiplications. The backward pass, however, still involves multiplications when pushing the error down by one layer (the \(W_3^T e_z\) and \(W_2^T e_2\) operations in Fig. 1). In section 2.2, we describe how these multiplications can be avoided.

A closely related development for reducing the computational and memory requirements of ANNs aims at training ANNs with low-precision weights [17–20]. During training, a high-precision version of the weights is typically stored and updated based on the errors calculated in the network. During the forward and backward passes, a low-precision quantized version of the weights is used. After training, the network operates using the low-precision weights. Since high-precision weights are still maintained during training, this approach can not reduce the ANN memory requirements during training. However, the use of quantized weights reduces the complexity of the logic used to compute the forward and backward passes.

Footnote: The latter choice (0, 1), which is equivalent to the bipolar binary representation \((-1, 1)\) through a linear transformation \(h \leftarrow 2h - 1\), is preferable in application settings that benefit from greater sparsity in neural activity. Throughout the following except where noted otherwise, the bipolar format \((-1, 1)\) is assumed, while the implementation in Sec. 2.2 supports both formats.
2.2 Learning Algorithm and Hardware Architecture

2.2.1 Learning Algorithm

Our goal is to use binary fully connected feedforward networks, also known as multi-layer perceptrons, to solve classification tasks. The top layer has as many neurons as the number of classification classes. We impose a cost function on the activity of the top layer neurons that is minimized when the activity of the top layer neuron corresponding to the correct input class is the highest among the top layer neurons. Note that the top layer neurons are not binary. Two of the most popular cost functions used in classification settings are the cross entropy loss and the square hinge loss. Cross entropy loss, however, involves exponentials and logarithms while the square hinge loss requires multipliers to implement the squaring operations. Therefore we use a simpler loss function, the hinge loss. Let $z$ be the vector of top layer activity and $z[i]$ be the $i^{th}$ element of $z$. If $p$ is the index of the correct class and $C$ is the number of classes (also the length of $z$), then the hinge loss is given by:

$$E_{hl} = \sum_{i=1}^{C} \max(0, z[i] + H - z[p])$$  \hspace{1cm} (4)$$

where $H \geq 0$ is the hinge hyper-parameter. The gradient of $E_{hl}$ with respect to $z$ is thus given by:

$$\frac{dE_{hl}}{dz[i]} = \begin{cases} \vartheta(z[i] + H - z[p]) & i \neq p \\ -\sum_{j=1}^{C} \vartheta(z[j] + H - z[p]) & i = p \end{cases}$$  \hspace{1cm} (5)$$

with the Heaviside operator $\vartheta(\cdot)$ defined as:

$$\vartheta(x) = \begin{cases} 1 & \text{if } x > 0 \\ -1 & \text{if } x < 0 \\ 0 & \text{otherwise.} \end{cases}$$  \hspace{1cm} (6)$$

The gradient $e_z = \frac{dE_{hl}}{dz}$ can thus be efficiently computed using comparators and adders. Note that all elements of $e_z$ are in the range $[-(C-1), 1]$.  

As shown in the example in Fig. 1, the backward pass in a binary-state network (BSN) involves multiplications in order to compute $W_3^T e_z$ and $W_2^T e_2$. Since at most one element in $e_z$ can have a value other than zero or one, and the absolute value of this element is at most $(C - 1)$, we can use repeated additions across at most $(C - 1)$ cycles to calculate the product of this element with a weight. Computing the second term, however, is more challenging as $e_2$ can take a broad range of values. Thus, we modify the backpropagation scheme illustrated in Fig. 3 so that all errors below the top layer are truncated to -1, 0, or 1. In our running example in Fig. 1, $e_2$ and $e_1$ are modified so that

$$e_2 = \text{sgn}(h'_2 \circ W_3^T e_z)$$  \hspace{1cm} (7)$$

$$e_1 = \text{sgn}(h'_1 \circ W_2^T e_2)$$

where the signum operator $\text{sgn}(\cdot)$ is defined as:

$$\text{sgn}(x) = \begin{cases} 1 & \text{if } x > 0 \\ -1 & \text{if } x < 0 \\ 0 & \text{otherwise.} \end{cases}$$  \hspace{1cm} (8)$$

The $\text{sgn}$ operation only yields zero if its argument is exactly zero. The forward pass of course only involves multiplications with binary values.  

In the hardware architecture described in the next section, we use limited precision fixed point weights during training and testing. We are thus interested in quantifying the effects of both limited precision weights, and error ternarization on the network performance. We use a fully connected network with 2 hidden layers that is trained on the MNIST dataset. The MNIST dataset contains 70,000 28×28 grayscale images of handwritten digits [21]. The training set of 60,000 labeled digits was used for training, and testing was done using the remaining 10,000 digits. No validation set was used. All grayscale images were
first binarized to two intensity values. The 784 neurons in the input layer thus have a binary output. In all trials we used standard stochastic gradient descent with a mini-batch size of 100.

Figure 2a shows the effect of error ternarization (Errors coming from the top layer are not ternarized) and limited precision (8-bit) weights on the network performance. The four lines depict the evolution of the test error in the 4 combinations of exact/ternary errors and 8-bit fixed-point/32-bit floating point weights. In all four cases, the network’s error on the training set reached zero. In the 8-bit case, we used a learning rate of one which is the smallest possible learning rate. In the 32-bit (full-precision) case, we used a real-valued exponentially decaying learning rate. The networks severely overfits on the training data which explains why using low-precision weights and error ternarization barely affect test accuracy.

To combat overfitting, we applied dropout [22] to the output of each layer (including the input layer). The training results are shown in Figure 2b. Networks with full-precision weights clearly outperform networks with 8-bit weights once the networks are regularized using dropout. Error ternarization slightly degrades accuracy in the regularized networks. However, the performance loss is small compared to the loss incurred when switching to 8-bit weights. In the 8-bit weights case, a weight update can not be smaller than $2^{-8}$ of the full weight range. Networks with 8-bit weights thus have a large effective learning rate. Small learning rates, however, are instrumental in allowing neural networks to gradually accumulate information from the entire training set. To achieve a small effective learning rate with large minimum weight updates, we tried using stochastic weight updates where the update of each individual weight is committed to memory with a probability $p_{commit}$. Weights thus change more slowly since weight updates are stochastically discarded. Performance of the network with 8-bit weights and two different commit probabilities, $p_{commit} = 0.5$ and $p_{commit} = 0.125$, is shown in Fig. 2c. Stochastically discarding weight updates did not appreciably improve network performance.

In order to approach the performance of networks with 32-bit floating point weights, we turn to networks with 16-bit fixed-point weights. We used a learning rate of 16. Using this learning rate, the size of the smallest weight update is thus $2^{-12}$ of the full weight range. The performance of networks with 16-bit weights is shown in Fig. 2d. Networks with 16-bit weights significantly outperform networks with 8-bit weights and their performance comes very close to that of full precision networks. Error ternarization slightly degrades accuracy in this case as well. In the hardware architecture described in the next section, we use 16-bit fixed-point weights as they represent a good trade-off point between network performance and hardware complexity.
2.2.2 Pipelined backpropagation

A straightforward implementation of backpropagation would carry out the forward pass (upward arrows in Fig. 1) followed by the backward pass (downward arrows in Fig. 1). That is because the computation in the backward pass depends on the error at the top layer and this error is only available at the end of the forward pass. Carrying out the forward and backward passes in a strict sequence, however, has the disadvantage that each network weight will have to be fetched twice: once during the forward pass to calculate the target neuron’s activation, and once during the backward pass so that the new weight value can be calculated by incrementing/decrementing the current value before writing the new weight value to memory.

Pipelined backpropagation addresses this problem by reducing the number of redundant weight fetches and removing the need for an explicit backward pass. We reuse the notation and 2-hidden layer example from Fig. 1. Let $x^{(i)}$ and $\text{target}^{(i)}$ be the $i^{th}$ input pattern and target presented to the network respectively, and $h_1^{(i)}$, $h_1'^{(i)}$, $h_2^{(i)}$, $h_2'^{(i)}$, $z^{(i)}$ and $e_z^{(i)}$ be the network state after processing the input $x^{(i)}$ and $\text{target}^{(i)}$ using the latest network weights. As weights are fetched in order to propagate $x^{(i)}$ upward through the network, the network does not yet have access to the weight updates associated with input pattern $x^{(i)}$. However, it can have access to the network states associated with previously presented
patterns and use these network states to carry out delayed weight updates associated with previously presented patterns.

Pipelined backpropagation is illustrated in Fig. 3 for the same example network from Fig. 1. During PASS 1, the first input and target, \( x^{(1)} \) and \( target^{(1)} \), are presented to the network and a complete forward pass is carried out, including the calculation of the top-level error \( e_{z}^{(1)} \). No weight updates are carried out during PASS 1. The network maintains a history of its state during PASS 1 when processing the second input in PASS 2. During PASS 2, as the network is fetching the weights in \( W_3 \) in order to calculate \( z^{(2)} \), it can use the values from the previous pass \( h_2^{(1)} \), \( h_2^{(1)} \), and \( e_z^{(1)} \) to carry out a delayed weight update for \( W_3 \) and delayed calculation of the error at the second hidden layer, \( e_2^{(1)} \), from \( e_z^{(1)} \). Similarly during PASS 3, as the network is fetching the weights in \( W_2 \) the error \( e_2^{(1)} \) is available and the network can update \( W_2 \) as well as push down the error by one layer to obtain \( e_1^{(1)} \). It is only during PASS 4 that all the weight updates associated with the first example, \( x^{(1)} \) can be completed using \( x^{(1)} \) and \( e_1^{(1)} \). The pipeline is now full and during all subsequent passes, each weight that is fetched to compute the forward pass for the current input will also be updated using an old network state and the delayed errors that are gradually pushed down from the top layer. This is illustrated in PASS 5. Note that each layer can discard its old error (shown in red) as soon as it has enough information to calculate the new error (shown in green). That is because the old error has already been used by the preceding layer and is no longer needed. The two errors in each layer thus never have to be simultaneously stored.

Due to the delayed weight updates, pipelined backpropagation does not yield the exact same results as standard backpropagation. For example, in PASS 3, the input \( x^{(3)} \) sees the initial values of \( W_1 \) and \( W_2 \) but the updated value of \( W_3 \). In standard backpropagation, the weights are all updated after a training example or a training minibatch. In pipelined backpropagation, the weights in higher layers are updated based on more recent input compared to the weights in deeper layers. For large datasets such as MNIST, these slight differences in the timing of the weight updates over the course of long training epochs have negligible impact on performance in practice, as we observed with our FPGA implementation.

It is clear from Fig. 3 that pipelined backpropagation incurs extra memory overhead in order to store old network states that are needed for carrying out the delayed weight updates. Let \( N_{inp} \), \( N_{hid} \), and \( N_{err} \) be the number of bits needed to store the activity of a neuron in the input layer, the activity (binary values of the activation and its derivative) of a neuron in the hidden layer, and the backpropagated error, respectively. In a network with \( L \) hidden layers, the extra memory needed in the input layer to carry out the pipelined backpropagation scheme shown in Fig. 3 is \((L + 1) \times N_{inp} \) bits per neuron. This is the extra memory compared to implementing backpropagation without pipelining. In the hidden layers, the extra memory requirements are largest in the deepest layer as this is the hidden layer which has to wait the longest to get the backpropagated error and requires \( L \times N_{hid} + N_{err} \) extra bits per neuron. The extra memory requirement per neuron successively decreases by \( N_{hid} \) bits for each layer above the deepest layer.

When using smooth activation functions, the derivative of the activation value does not need to be explicitly stored as it can be inferred from the activation value itself. However, for BSNs, the derivative needs to be stored as it can not be inferred from the neuron’s binary value. Thus the neuron’s activity can be compactly represented as just 2 bits, as the neuron’s output and the gradient of the activation function are both binary (Equations 2 and 3). Since we also use dropout while training the BSN, we need an extra bit to indicate whether the neuron was dropped in the forward pass. Bringing the total number of bits needed to store the neuron’s state to 3: \( N_{hid} = 3 \), and \( N_{inp} = 2 \) if the input variables are also binary. Note that we do not need to store the derivative information for the input layer neurons. Implementations of conventional networks on custom hardware often use 16 bits of precision for the activation values, i.e., 16 bits are needed to store the neuron state \( N_{hid} = 16 \) if we assume rectified linear units (ReLUs) [23] are used. The memory overhead of implementing pipelined backpropagation in BSNs is thus \( \sim 5.3x \) smaller compared to conventional networks. The reduction in memory overhead becomes significant when implementing deep networks that incur a larger memory overhead to support pipelined backpropagation.

### 2.2.3 Hardware Architecture

We developed a proof-of-concept hardware architecture to illustrate the viability and classification accuracy of the proposed pipelined backpropagation scheme illustrated in Fig. 3. This proof-of-concept architecture targets an FPGA platform where the weights are externally stored in DRAM. The archi-
Figure 3. Illustration of pipelined backpropagation for the two-layer network of Fig. 1, showing network history and storage requirements. The upward arrows indicate the order in which the weight updates are carried out, in the same order of weight lookups in the forward pass. Each additional layer in the network incurs for each lower layer additional delay in the error computation and the weight updates, which also requires additional history of the hidden unit states to be stored. Previous errors (shown in red) are overwritten by the newly backpropagated errors (shown in green) at the end of each pass. Hence only one error value is simultaneously stored per layer.

The proposed architecture is shown in Fig. 4a. We use 16-bit signed fixed-point weights. The neurons are distributed across sixteen cores where each core implements 256 neurons. Each core can only contain neurons belonging to one layer. Multiple cores can be assigned to the same layer. Each core communicates with a central controller. The states of the 256 neurons in a core are stored in internal memory that is local to the core. Each neuron has a 15-bit history field divided into 5 3-bit slots which can store the neuron’s state (binary output, binary derivative, and dropout state) for up to 5 passes in the past, a 32-bit accumulator field used to both accumulate forward propagating input and backward propagating error, and a 2-bit field used to store the ternary error at that neuron. Each core receives a 1-bit dropout signal from the Pseudo Random Number Generator (PRNG). The PRNG is implemented using two counter-propagating linear feedback shift registers with differing feedback length following the scheme in [24]. The dropout signal from the PRNG decides whether the currently updating neuron should be dropped for the current input. The central controller sends the update signal to each core in succession. When a core receives the update signal, it sequentially updates the states of its 256 neurons. A neuron update involves the following steps:

1. The neuron compares its local accumulator value to zero to decide its binary output value. It also decides the value of the binary virtual gradient by checking whether the accumulator value is in the range $[-2^{16}, 2^{16}]$ (corresponding to the range $[-1, 1]$ in Eq. 3). The neuron shifts the new binary value, virtual gradient, and dropout state into the 15-bit history field. The oldest 3-bit state in the history field is thus discarded. The neuron then resets the accumulator to zero.

2. The neuron communicates its new binary output value to the controller together with its dropout state. The neuron also communicates its delayed output value and delayed dropout state from $K$ passes in the past. These delayed quantities are fetched from the history field. $K$ is different for neurons in different layers and is larger for neurons in deeper layers. $K$ is the same for neurons in the same core as each core contains neurons from the same layer so $K$ is stored in a central core
Figure 4. (a) Block diagram of the FPGA architecture implementing pipelined backpropagation. Sixteen neuron cores with 256 neurons each are sequentially updated to realize the pipelined backpropagation scheme illustrated in Fig. 3. Each core uses $256 \times 49 = 12,544$ bits of internal memory to store the states of the 256 neurons. A Pseudo Random Number Generator (PRNG) supplies the dropout signal to the core. The probability that the dropout signal is asserted is controlled by a configurable register in the PRNG. (b) Layout of weights in the external memory.

3. If (and only if) the current dropout state of the source (updating) neuron is deasserted: The controller fetches the updating neuron’s outgoing weights from the external memory. For each fetched weight, it multiplies the weight by the source (updating) neuron’s current binary value and dispatches the result to the target neuron. The target neuron updates its accumulator using the incoming data and outputs its current error value.

4. If (and only if) the delayed-input dropout state of the source (updating) neuron is deasserted: The controller reads the error from the target neuron. If the error is ternary, the controller multiplies this error by the weight and sends the result to the updating (source) neuron. The neuron accumulates the incoming value into the accumulator field. If the error is not ternary, i.e., it is coming from a top layer neuron, then it will have an absolute value of at most $C - 1$. Denote this error by $e_z[i]$. Instead of multiplying the weight by the error, the controller dispatches the weight with the appropriate sign $e_z[i]$ times to the target neuron in order to implement multiplications through repeated additions. The controller multiplies the delayed output of the updating neuron (which is binary) by the delayed error from the target neuron to calculate the weight update, then writes the updated weight to memory if the weight update is non-zero.

5. After all the outgoing weights of the updating neuron have been processed, the updating neuron fetches the binary gradient value from $K$ passes in the past from the history field. This binary gradient is then multiplied by the accumulator value (which now contains the weighted sum of the backpropagated delayed errors from all the target neurons) to obtain the new high precision error in the neuron. This error is directly ternarized according to Eq. 7 and stored in the error field. The accumulator is then reset.

Note that the neuron accumulator which is used to accumulate the incoming neuron input coming from the layer below is also reused for accumulating the errors coming from the layer above. Neurons in the output layer are special. Updating these neurons is done internally in a special neuron core (core 15) based on the current input label. The core spends $C$ cycles to obtain the current classification result by finding the neuron with the maximum value among the $C$ output neurons. During these $C$ cycles, it also calculates the new top layer errors using Eq. 5.

The input layer neurons are implemented the same way as the hidden layer neurons. At the beginning of each pass, the accumulators of the input neurons are set to $+1/-1$ to encode the binary-valued input vector. This is wasteful in terms of memory resources as the extra 31 bits of the accumulator and the 2-bit error field are not needed for the input neurons but it leads to a more uniform implementation. A pass consists of setting the input neuron values then updating all the cores in succession which implements the pipelined backpropagation scheme depicted in the example in Fig. 3. For the first few initial passes, the
backpropagation pipeline will not be full (PASS 1 and PASS 2 in Fig. 3 for example). Special registers in the controller handle this initial phase by only committing weight updates calculated using valid error and delayed neuron output values. Since the network has to be updated from bottom to top (see Fig. 3), neurons belonging to one layer should occupy a core with a lower index than neurons belonging to a higher layer.

Each core has a flag that indicates how the binary value of the neuron should be interpreted: either as -1/1 or 0/1. This flag is communicated to the controller and influences how the target neurons and the weights are updated. In the 0/1 interpretation, no weight is sent to the target neuron if the updating (source) neuron value is low (as opposed to sending the negative of the weight in the -1/1 interpretation). Moreover no weight update is carried out if the delayed neuron value is low as multiplying this delayed value by the target neuron’s error would yield zero. The virtual gradient calculated using Eq. 3 is independent of the binary interpretation.

Figure 4b shows the structure of the data in the external memory. To fetch a neuron’s outgoing weights and targets, the neuron’s address is used as in index in the green region to fetch the location of the neuron’s outgoing weights list, and to fetch the number of target neurons and the address of the first target neuron. Since a neuron always targets a consecutive set of neurons, only the starting neuron and the number of targeted neurons are needed. A neuron’s weights list is an ordered list of 16-bit weights specifying the outgoing weights to all the target neurons. While full-indexing in weight lookup could be used to provide greater flexibility in sparse reconfigurable synaptic connectivity [25], the implemented lookup scheme is more compact, incurring only a small memory overhead (the green region) when storing the weights. In our particular implementation, this overhead is 64 bits per neuron.

3 Results

We implemented the proposed architecture on a Spartan6-LX150 FPGA. The external memory is a DDR2 memory. Each core stores the states of its 256 neurons in two 9-Kbit block rams. Due to the little parallelism in the architecture, the FPGA implementation takes up a small fraction of the FPGA resources: 3585 slice registers (1%), 7425 slice LUTs (8%), and 30 9-Kbits block ram elements (5%). The FPGA core operating frequency is 78 MHz. We configured the FPGA to implement a 2-hidden layer network with 600 neurons in each hidden layer, and to train the network on the MNIST dataset. The input layer of 784 neurons occupied 4 neuron cores and was configured as an unsigned binary (0,1) layer. Each hidden layer occupied 3 neuron cores and was configured as a bipolar binary (-1,+1) layer. The output layer was implemented on a special core (core 15) designed to calculate the error based on the hinge loss from Eq. 4. The weights in the DDR2 memory were initialized using the initialization scheme in [26]. The MNIST images were binarized and stored in the DDR2 memory together with their labels. Each MNIST image/label pair takes up 784 + 4 = 788 bits. The controller fetches the training images/labels sequentially from memory to configure the input layer (using the pixel values) and the output layer (using the label). Evaluation of the test set is also done on FPGA after switching off learning.

Exponentially decaying learning rates allow neural networks to converge to good solutions faster. We implemented a similar scheme in order to accelerate learning: the learning rate which is equivalent to the the smallest weight increment/decrement magnitude was halved every 10 epochs. The FPGA trained the network for 50 epochs. The minimum weight increment/decrement magnitude started at 128 in the first 10 epochs and ended at 8 in the last 10 epochs. The FPGA implements a dropout probability of 0.2 between all layers. Figure 5a shows the test set errors after each of the 50 training epochs. The network reaches a test set error of 1.89%.

Figure 5b shows the post-training distribution of weights in each of the three weight matrices in the network. Only a small percentage of weights are clipped at the boundaries: 0.016% in the input weights $W_1 (x \rightarrow h_1)$, 0.0011% in the intermediate weights $W_2 (h_1 \rightarrow h_2)$, and 1.1% in the output weights $W_3 (h_2 \rightarrow z)$. The distribution of the output weights is markedly more skewed towards the weight limits compared to the other two weight matrices. This could be due to the fact that these weights are the only weights that can be incremented/decremented by more than the learning rate for each training example because they see the non-ternarized error from the top layer whose magnitude could be as large as $C - 1 = 9$. Weights in the other two weight matrices always see a ternarized backpropagated error so they can not change by more than the learning rate for each training example.
Figure 5. (a) Test set errors on the MNIST dataset across 50 training epochs. Training and testing were conducted on the FPGA using a 784-600-600-10 network. The error figures in the legend refer to the final error figures after epoch 50. (b) Histogram of weights in each of the three inter-layer weight matrices $W_1$, $W_2$ and $W_3$ in the network after training. Weight clipping is particularly apparent in the output weights $W_3$ ($h_2 \rightarrow z$).

4 Conclusions and Discussion

We presented a scheme for the efficient implementation of pipelined backpropagation to train multi-layer feedforward networks. Due to the use of binary-state networks, the scheme is highly efficient in terms of logic and memory resources. We developed a proof-of-concept hardware architecture and implemented the architecture on FPGA to validate the proposed approach for pipelined training of BSNs. Due to error ternarization, the core operation in the forward and backward passes is fixed-point addition/subtraction. This is an equivalent operation to the synaptic operation (SynOp) which is the basic operation in spiking neuromorphic systems. SynOps can be up to two orders of magnitude more efficient than conventional MAC operations [27, 28]. Perhaps the biggest advantage of pipelined backpropagation is that it reduces the number of weight fetches compared to sequential forward and backward passes. In state of the art networks where the weights are too many to fit into the accelerator memory, reducing the off-chip weight traffic can lead to significant energy and performance gains. Moreover, no reverse lookup of weights is needed and weights can be stored in a way that only optimizes lookups using the source neuron address.

Pipelined backpropagation enables layer-level parallelism. A layer can begin processing a new input vector from the layer below as soon as it has finished processing the previous input vector and fetching the weighted errors from the layer above. We did not implement layer-level parallelism as it would require each layer to have access to its own weight memory in order to operate independently from the other layers. One of the main shortcomings of pipelined backpropagation is the extra memory needed to store the delayed errors and network state. Through the use of binary activation functions in the forward inference pass and ternary errors in the backward learning pass, this extra memory is kept to a minimum which makes pipelined backpropagation a feasible option for training deep networks.

The performance on the MNIST dataset was adequate but not state of the art. A natural extension of the presented architecture and algorithm would be the implementation of convolutional feedforward networks which achieve superior performance on learning tasks with a spatial structure such as vision-related tasks. When training BSNs, or binarized neural networks [11], normalization techniques such as batch-normalization [29] are used to center and normalize the variance of the input to each neuron. This is particularly important for BSNs due to the hard non-linearity used and the fact that the gradient only flows back when the input to the neuron is around zero. This is clearly useful when learning static datasets using mini-batches. In an online setting with continuous learning (effective minibatch size of 1) and continuously changing inputs, it is unclear how normalization should be applied. An online normalization technique has to take into account that the input statistics could change quickly during online learning in a real-world environment, hence a normalization technique with weak history dependence is preferred [30]. A hardware-efficient normalization technique that can be applied online is thus a clear next step to allow the proposed architecture to train deep networks online.
Even though limited precision weights and binary activations intrinsically regularize the training process by reducing the degrees of freedom in the network, this regularization effect was clearly not enough in the MNIST dataset case. An additional regularization technique, dropout, was used to reduce overfitting. Dropout is efficiently implemented by gating the weight lookup and update operations, leading to higher throughput owing to lower average memory access footprint per training sample. However, dropout also requires more training samples to reach convergence. In an online setting, dropout would require higher rates of processing the streaming data to iterate training on each input sample multiple times with different random dropout instances.

Spiking neural networks are an alternative network paradigm that is similar in many respects to BSNs in terms of required hardware resources; both types of networks can be built using adders and comparators and require no multipliers. Spiking networks have been used to solve various classification tasks [31–35]. Even though they require very similar computational resources, the energy, memory access patterns, and time needed to carry out the inference/forward pass in spiking networks and BSNs can be significantly different. Spiking networks are often used in the rate-based mode where the output value of a spiking neuron is encoded in its average firing rate. Multiple lookups of the same weights are thus needed to dispatch multiple spikes from the same neuron which could significantly raise energy consumption. The weight lookups are also more irregular compared to BSNs since neurons spike in an asynchronous manner. This reduces the ability to pipeline memory accesses. Spiking networks are dynamical systems which are emulated using time-stepped dynamics in digital implementations [27, 36] or using native analog dynamics [37–39] in analog/mixed-signal implementations. The dynamic nature of spiking networks results in an irregular computational load as there could be intervals where the network is quiescent and intervals where many neurons spike simultaneously or in rapid succession. This makes it difficult to consistently achieve optimal utilization of computational and memory access resources, unlike BSNs where data movement and computations are much more predictable.

The computational and weight lookup overhead in a BSN is roughly equivalent to that of a spiking network where each neuron spikes exactly once. The forward/inference pass in spiking networks can potentially be more efficient than a BSN if, on average, a neuron generates less than one spike. This can be achieved even in a rate-based network by making a classification decision based only on the first spike from the output layer [34, 40]. Classification accuracy, however, will typically suffer. Alternatively, the spiking network could be explicitly trained to optimize spike times so that an accurate classification decision is made using only one spike from the output layer [41, 42]. Spiking networks also have a decisive advantage when processing dynamic and sparse event-based data such as the event trains coming from neuromorphic sensors [43–45] as the networks can scale their spiking activity in response to the dynamically changing input event stream. The spiking network could thus effectively shut down during intervals when there is no input activity, saving power. Similar sparsity-induced power savings in the synchronous setting [6] may be obtained with BSNs using unsigned (0,1) rather than bipolar (−1, +1) binary neural activations.

One of the main advantages of BSNs is that they are effectively trainable using backpropagation. Training of spiking networks is often done indirectly in an offline manner by first training a conventional ANN then mapping the weights to the spiking network [31–33]. Recently, several approaches based on approximations to backpropagation have been proposed that can allow online training of spiking networks [34, 40]. These approaches, however, are based on spiking networks with rate coding, which typically require more memory accesses and longer processing time for each training pattern compared to BSNs. An alternative training approach based on exact backpropagation and temporal coding in spiking networks [41] has been shown to lead to highly sparse spiking activity during training and inference, and could potentially be more energetically efficient than training BSNs using the approach presented in this paper.

In summary, BSNs have the attractively low computational overhead of spiking networks, while still being efficiently trainable using backpropagation. They do not incur the increased computational and weight lookup overhead of rate-based spiking networks. Through the use of the approximate pipelined backpropagation scheme outlined in this paper, BSNs can be trained using significantly reduced weight lookup overhead while incurring a modest overhead in the neuron complexity.

**References**

1. Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.
2. David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by\nbak-propagating errors. *Nature*, 323(6088):533–536, 1986.

3. S. Himavathi, D. Anitha, and A. Muthuramalingam. Feedforward neural network implementation\nin FPGA using layer multiplexing for effective resource utilization. *IEEE Transactions on Neural\nNetworks*, 18(3):880–888, 2007.

4. Lukas Cavigelli, David Gschwend, Christoph Mayer, Samuel Willi, Beat Muheim, and Luca Benini.\nOrigami: A convolutional network accelerator. In *Proc. 25th Great Lakes Symposium on VLSI*,\npages 199–204. ACM, 2015.

5. Yu-Hsin Chen, Tushar Krishna, Joel Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 262–263. IEEE, 2016.

6. Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. EIE: Efficient inference engine on compressed deep neural network. In *Proc. 43rd International Symposium on Computer Architecture (ISCA)*, pages 243–254. IEEE Press, 2016.

7. Arash Ardakani, François Leduc-Primeau, Naoya Onizawa, Takahiro Hanyu, and W.J. Gross. VLSI implementation of deep neural networks using integral stochastic computing. In *Turbo Codes and Iterative Information Processing (ISTC), 2016 9th International Symposium on*, pages 216–220. IEEE, 2016.

8. Alessandro Aimar, Hesham Mostafa, Enrico Calabrese, Antonio Rios-Navarro, Ricardo Tapiador-Morales, Iulia-Alexandra Lungu, Moritz B. Milde, Federico Corradi, Alejandro Linares-Barranco, Shih-Chii Liu, and Tobi Delbruck. Nullhop: A flexible convolutional neural network accelerator based on sparse representations of feature maps. *arXiv preprint arXiv:1706.01406*, 2017.

9. Alexander Gomperts, Abhisek Ukil, and Franz Zurfüh. Development and implementation of parameterized FPGA-based general purpose neural networks for online applications. *IEEE Transactions on Industrial Informatics*, 7(1):78–89, 2011.

10. Francisco Ortega-Zamorano, J.M. Jerez, D.U. Muñoz, R.M. Luque-Baena, and Leonardo Franco. Efficient implementation of the backpropagation algorithm in FPGAs and microcontrollers. *IEEE transactions on neural networks and learning systems*, 27(9):1840–1850, 2016.

11. Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Advances in Neural Information Processing Systems*, pages 4107–4115, 2016.

12. Alain Petrowski, Gerard Dreyfus, and Claude Girault. Performance analysis of a pipelined backpropagation parallel algorithm. *IEEE Transactions on Neural Networks*, 4(6):970–981, 1993.

13. R.G. Gironés, R.C. Palero, J.C. Boluda, and A.S. Cortés. FPGA implementation of a pipelined on-line backpropagation. *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology*, 40(2):189–213, 2005.

14. Mohammed Bahoura and Chan-Wang Park. FPGA-implementation of high-speed mlp neural network. In *Electronics, Circuits and Systems (ICECS), 2011 18th IEEE International Conference on*, pages 426–429. IEEE, 2011.

15. Antony Savich, Medhat Moussa, and Shawki Areibi. A scalable pipelined architecture for real-time computation of MLP-BP neural networks. *Microprocessors and Microsystems*, 36(2):138–150, 2012.

16. Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

17. Evangelos Stromatias, Daniel Neil, Michael Pfeiffer, Francesco Galluppi, Steve B Furber, and Shih-Chii Liu. Robustness of spiking deep belief networks to noise and reduced bit precision of neuro-inspired hardware platforms. *Frontiers in neuroscience*, 9, 2015.
18. Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems*, pages 3123–3131, 2015.

19. Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

20. Chenzhuo Zhu, Song Han, Huizi Mao, and W.J. Dally. Trained ternary quantization. *arXiv preprint arXiv:1612.01064*, 2016.

21. Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

22. Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

23. Vinod Nair and G.E. Hinton. Rectified linear units improve restricted Boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814, 2010.

24. Gert Cauwenberghs. An analog VLSI recurrent neural network learning a continuous-time trajectory. *IEEE Transactions on Neural Networks*, 7(2):346–361, 1996.

25. Jongkil Park, Theodore Yu, Siddharth Joshi, Christoph Maier, and Gert Cauwenberghs. Hierarchical address event routing for reconfigurable large-scale neuromorphic systems. *IEEE Transactions on Neural Networks and Learning Systems*, 2017.

26. Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Aistats*, volume 9, pages 249–256, 2010.

27. P.A. Merolla, J.V. Arthur, Rodrigo Alvarez-Icaza, A.S. Cassidy, Jun Sawada, Filipp Akopyan, B.L. Jackson, Nabil Imam, Chen Guo, Yutaka Nakamura, et al. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–673, 2014.

28. E.O. Neftci, B.U. Pedroni, Siddharth Joshi, Maruan Al-Shedivat, and Gert Cauwenberghs. Unsupervised learning in synaptic sampling machines. *arXiv preprint arXiv:1511.04484*, 2015.

29. Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

30. J.L. Ba, J.R. Kiros, and G.E. Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

31. P. O’Connor, D. Neil, S.-C. Liu, T. Delbruck, and M. Pfeiffer. Real-time classification and sensor fusion with a spiking deep belief network. *Frontiers in Neuroscience*, 7(178), 2013.

32. Peter U Diehl, Daniel Neil, Jonathan Binas, Matthew Cook, Shih-Chii Liu, and Michael Pfeiffer. Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing. In *International Joint Conference on Neural Networks (IJCNN)*, 2015.

33. Yongqiang Cao, Yang Chen, and Deepak Khosla. Spiking deep convolutional neural networks for energy-efficient object recognition. *International Journal of Computer Vision*, 113(1):54–66, 2015.

34. Jun Haeng Lee, Tobi Delbruck, and Michael Pfeiffer. Training deep spiking neural networks using backpropagation. *Frontiers in Neuroscience*, 10, 2016.

35. S.K. Esser, P.A. Merolla, J.V. Arthur, A.S. Cassidy, Rathinakumar Appuswamy, Alexander Andreopoulos, D.J. Berg, J.L. McKinstry, Timothy Melano, D.R. Barch, et al. Convolutional networks for fast, energy-efficient neuromorphic computing. *Proceedings of the National Academy of Sciences*, 113(41), 2016.
36. M.M. Khan, D.R. Lester, L.A. Plana, A. Rast, X. Jin, E. Painkras, and S.B. Furber. SpiNNaker: Mapping neural networks onto a massively-parallel chip multiprocessor. In *IEEE International Joint Conference on Neural Networks, 2008. IJCNN 2008. (IEEE World Congress on Computational Intelligence)*, pages 2849–2856, June 2008.

37. Ben Varkey Benjamin, Peiran Gao, Emmett McQuinn, Swadesh Choudhary, Anand R Chandrasekaran, J Bussat, R Alvarez-Icaza, JV Arthur, PA Merolla, and K Boahen. Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations. *Proceedings of the IEEE*, 102(5):699–716, 2014.

38. Jongkil Park, Sohmyung Ha, Theodore Yu, Emre Neftci, and Gert Cauwenberghs. A 65k-neuron 73-Mevents/s 22-pJ/event asynchronous micro-pipelined integrate-and-fire array transceiver. In *Biomedical Circuits and Systems Conference (BioCAS), 2014 IEEE*, pages 675–678. IEEE, 2014.

39. Ning Qiao, Hesham Mostafa, Federico Corradi, Marc Osswald, Fabio Stefanini, Dora Sumislawska, and Giacomo Indiveri. A re-configurable on-line learning spiking neuromorphic processor comprising 256 neurons and 128k synapses. *Frontiers in Neuroscience*, 9(141), 2015.

40. Emre Neftci, Charles Augustine, Sonnath Paul, and Georgios Detorakis. Neuromorphic deep learning machines. *arXiv preprint arXiv:1612.05596*, 2016.

41. Hesham Mostafa. Supervised learning based on temporal coding in spiking neural networks. *arXiv preprint arXiv:1606.08165*, 2016.

42. Hesham Mostafa, Bruno Pedroni, Sadique Sheik, and Gert Cauwenberghs. Fast classification using sparsely active spiking networks. In *International Symposium on Circuits and Systems,(ISCAS), 2017*. IEEE, 2017.

43. P. Lichtsteiner, C. Posch, and T. Delbruck. A 128x128 120 dB 30 mW asynchronous vision sensor that responds to relative intensity change. In *2006 IEEE ISSCC Digest of Technical Papers*, pages 508–509. IEEE, Feb 2006.

44. S.C. Liu, A. van Schaik, B.A. Minch, and T. Delbruck. Event-based 64-channel binaural silicon cochlea with Q enhancement mechanisms. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 2027–2030. IEEE, 2010.

45. S.-C. Liu and T. Delbruck. Neuromorphic sensory systems. *Current Opinion in Neurobiology*, 20(3):288–295, 2010.