ReBNet: Residual Binarized Neural Network

Mohammad Ghasemzadeh, Mohammad Samragh, and Farinaz Koushanfar
Department of Electrical and Computer Engineering, University of California San Diego
{*mghasemzadeh, msamragh, farinaz*}*@ucsd.edu*

***Abstract*—This paper proposes ReBNet, an end-to-end framework for training reconfigurable binary neural networks on software and developing efficient accelerators for execution on FPGA. Binary neural networks offer an intriguing opportunity for deploying large-scale deep learning models on resource-constrained devices. Binarization reduces the memory footprint and replaces the power-hungry matrix-multiplication with light-weight XnorPopcount operations. However, binary networks suffer from a degraded accuracy compared to their fixed-point counterparts. We show that the state-of-the-art methods for optimizing binary networks accuracy, significantly increase the implementation cost and complexity. To compensate for the degraded accuracy while adhering to the simplicity of binary networks, we devise the first reconfigurable scheme that can adjust the classification accuracy based on the application. Our proposition improves the classification accuracy by representing features with *multiple* levels of residual binarization. Unlike previous methods, our approach does not exacerbate the area cost of the hardware accelerator. Instead, it provides a tradeoff between throughput and accuracy while the area overhead of multi-level binarization is negligible.**

***Keywords*-Deep neural networks, Reconfigurable computing, Domain-customized computing, Binary neural network, Residual binarization.**

I. Introduction

Convolutional Neural Networks (CNNs) are widely used in a variety of machine learning applications, many of which are deployed on embedded devices [1], [2], [3]. With the swarm of emerging intelligent applications, development of real-time and low-power hardware accelerators is especially critical for resource-limited settings. A line of research has therefore been focused on the development of FPGA accelerators for execution of CNN applications [4], [5], [6]. Although the building blocks of CNNs are highly parallelizable, the high computational complexity and memory footprint of these models are barriers to efficient implementation.

A number of prior works have focused on reducing the computational complexity and memory footprint of CNNs by trimming the redundancies of the model prior to designing an accelerator. Examples of such optimization techniques include tensor decomposition [7], [8], [9], parameter quantization [10], [11], [12], sparse convolutions [13], [14], and training binary neural networks [15], [16].

Among the above optimization techniques, binary networks result in two particular benefits: (i) They reduce the memory footprint compared to models with fixed-point parameters; this is especially important since memory access plays an essential role in the execution of CNNs on resource-constrained platforms. (ii) Binary CNNs replace the power-hungry multiplications with simple XNOR operations [16], [17], significantly reducing the runtime and energy consumption. Consequent to the aforementioned benefits, the dataflow architecture of binary CNNs is remarkably simpler than their fixed-point counterparts.

There are several remaining challenges for training binary CNNs. First, the training phase of binary neural networks is often slow and the final classification accuracy is typically lower than the model with full-precision parameters. It has been shown that the loss of accuracy can be partially evaded by training binary networks that have wider layers [17]. Nevertheless, this method for accuracy enhancement diminishes the performance gains of binarization. In another effort, authors of XNOR-net [16] tried to improve the accuracy of the binary CNN using scaling factors that are computed by averaging the features during inference. This method, however, sacrifices the simplicity of the binary CNN accelerator by adding extra full-precision calculations to the computation flow of binary CNN layers. Similarly, the approach in [18] involves multiple rounds of computing the average absolute value of input activations which incurs an excessive computation cost during the inference phase. Ternary neural networks [19], [20] may surpass binary CNNs in terms of the inference accuracy; however, they are deprived of the benefits of simple XNOR operations.

We argue that a practical solution for binary CNNs should possess two main properties: (i) The accuracy of the binary model should be comparable to its full-precision counterpart. (ii) The proposed method to improve the accuracy should not compromise the low overhead and accelerator scalability benefits of binary networks. This paper proposes ReBNet, an end-to-end framework for training reconfigurable binary CNNs in software and developing efficient hardware accelerators for execution on FPGA. We introduce the novel concept of multi-level binary CNNs and design algorithms for learning such models. Building upon this idea, we design and implement a scalable FPGA accelerator for binary CNNs. The benefit of our approach over existing binarization methods is that the number of binarization levels can be adjusted for different applications without significant hardware modification.

In ReBNet, the weight parameters of CNN layers are represented with 1-level binarized values, while a multi-level residual binarization scheme is learned for the activation units. As such, the memory footprint of the *parameters* in ReBNet is the same as that of a single-level Binary CNN. We show that the accuracy of ReBNet can be improved by using 2 or 3 levels of residual binarization with a negligible area overhead. The design of ReBNet residual binarization is

compatible with the standard *XnorPopcount* operations; the underlying computations involving the feature vectors can be decomposed into a number of standard *XnorPopcount* operations. As such, ReBNet provides scalability for the design of binary CNNs. The contributions of this paper are summarized as follows:

- Proposing residual binarization as a reconfigurable dimension of binary CNNs. We devise an activation function with few scaling factors that are used for residual binarization. We also introduce a method for training the scaling factors.
- Development of an Application Programming Interface (API) for training multi-level binarized CNNs[1].
- Creation of a hardware library for implementation of different CNN layers using ReBNet methodology. The library allows users to configure the parallelism in each CNN layer using high-level parameters[1].
- Performing proof-of-concept evaluations on four benchmarks on three FPGA platforms.

II. Preliminaries

In this section, we outline the operations of binary CNNs and their hardware implementation. Specifically, we briefly describe the FPGA design proposed by [17]. Please refer to the original paper for a more detailed explanation.

A. Binary CNN Operations

Neural networks are composed of multiple convolution, fully-connected, activation, batch-normalization, and max-pooling layers. Binarization enables the use of a simpler equivalent for each layer as explained in this section.

Binary dot product: The computational complexity of neural networks is mostly owing to the convolution and fully-connected layers. Both layers can be broken into a number of dot products between input features and weight parameters. A dot product accumulates the element-wise products of a feature vector $\vec{x}$ and a weight vector $\vec{w}$:

$$dot(\vec{x},\vec{w}) = \sum_i \vec{x}[i] \times \vec{w}[i] \tag{1}$$

In case of binary CNNs, the elements of $\vec{x}$ and $\vec{w}$ are restricted to binary values $\pm\gamma_x$ and $\pm\gamma_w$, respectively. The dot product of these vectors can be efficiently computed using *XnorPopcount* operations as suggested in [15], [16]. Let $\vec{x} = \gamma_x\ \vec{s}_x$ and $\vec{w} = \gamma_w\ \vec{s}_w$, where $\{\gamma_x, \gamma_w\}$ are scalar values and $\{\vec{s}_x, \vec{s}_w\}$ are sign vectors whose elements are either $+1$ or -1. If we encode the sign values ($-1 \rightarrow 0$ and $+1 \rightarrow 1$), we obtain binary vectors $\{\vec{b}_x, \vec{b}_w\}$. The dot product between $\vec{x}$ and $\vec{w}$ can be computed as:

$$dot(\vec{w},\vec{x}) \ = \gamma_x\gamma_w\ dot(\vec{s}_x,\vec{s}_w) = \gamma_x\gamma_w\ XnorPopcount(\vec{b}_x,\vec{b}_w) \tag{2}$$

Figure 1 depicts the equivalence of $dot(\vec{s}_x,\vec{s}_w)$ and $XnorPopcount(\vec{b}_x,\vec{b}_w)$ using an example.

Binary activation: The binary activation function encodes an input y using a single bit based on the sign of y. Therefore, the hardware implementation only requires a comparator.

[1]Codes are available at https://github.com/mohaghasemzadeh/ReBNet

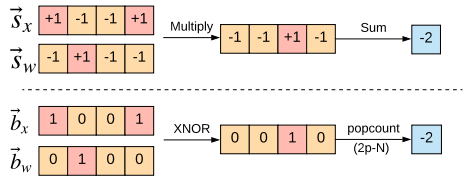


Figure 1: The equivalence of dot product (top) and *XnorPopcount* (bottom) operations. In the popcount operation, p is the number of set bits and N is the size of the vector.

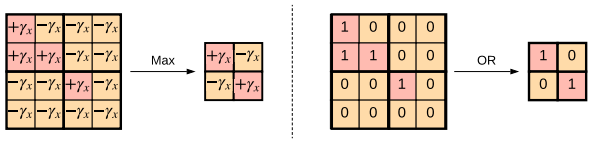


Figure 2: The equivalence of max-pooling operation over binarized features (left) and OR operation over encoded features (right).

Binary batch-normalization: It is often useful to normalize the result of the dot product $y = dot(\vec{x},\vec{w})$ before feeding it to the binary activation function described above. A batch-normalization layer converts each input y into $\alpha \times y - \beta$, where α and β are the parameters of the layer. Authors of [17] suggest combining batch-normalization and binary-activation layers into a single thresholding layer. The cascade of the two layers computes the following:

$$output = Sign(\alpha \times y - \beta) = Sign(y - \frac{\beta}{\alpha}) \tag{3}$$

Therefore, the combination of the two layers only requires a comparison with the threshold value $\frac{\beta}{\alpha}$.

Binary max-pooling: A max-pooling layer computes the maximum of features over sliding windows of the input feature map. The max-pooling operation can be performed by element-wise OR over the binary features. Figure 2 depicts the equivalence between max-pooling over fixed-point features $\vec{x} \in \{+\gamma, -\gamma\}^N$ and element-wise OR over binary features $\vec{b}_x \in \{0,1\}^N$.

B. Hardware Implementation and Parallelism

Matrix multiplication: Authors of [17] propose the flow diagram of Figure 3-(a) to implement different layers of binary CNNs on FPGA. The sliding window unit (SWU) scans the input feature maps of convolution layers and feeds appropriate values to the corresponding matrix vector threshold unit (MVTU). Both convolution and fully-connected layers are implemented using the MVTU, which realizes matrix-vector multiplication, batch normalization, and binary activation.

Parallelism: The MTVU offers two levels of parallelism for matrix-vector multiplication as depicted in Figure 3-(b,c). First, each MTVU has a number of processing elements (PEs) that compute multiple output neurons in parallel; each PE is responsible for a single dot product between two binarized vectors. Second, each PE breaks the corresponding

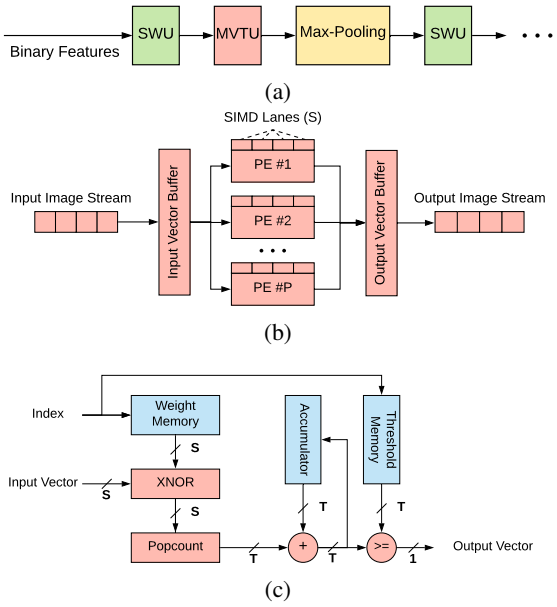


Figure 3: (a) Computation flow of FINN [17] accelerator. (b) An MVTU with "P" processing elements, each with SIMD-width of "S". (c) Architecture of a processing unit. "S" is the SIMD-width and "T" is the fixed-point bitwidth.

operation into a number of sub-operations, each of which performed on SIMD-width binary words.

III. Overview

The global flow of ReBNet API is presented in Figure 4. The user provides the CNN architecture to the software library, which trains the binary CNN with a specified number of residual binarization levels. She/he also provides the network description using our hardware library, along with the parallelism factors for the hardware accelerator. Based on these parallelism factors (PE-count and SIMD-width), the binary network parameters are re-aligned and stored appropriately to be loaded into the hardware accelerator. The bitfile is then generated using the hardware library.

In this section, we first describe residual binarization as a reconfigurable dimension in the design of binary CNNs. We next discuss the training methodology for residual binary networks. Finally, we elaborate on our hardware accelerator.

A. Residual Binarization

Imposing binary constraints on weights and activations of a neural network limits the model's ability to provide the inference accuracy that a floating-point or fixed-point counterpart would achieve. To address this issue, we propose a multi-level binarization scheme where the residual errors are sequentially binarized to increase the numerical precision of the approximation.

Multi-level residual binarization: Figure 5 presents the procedure to approximate a fixed-point input x with a multi-level binarized value e. For an M-level binarization scheme,

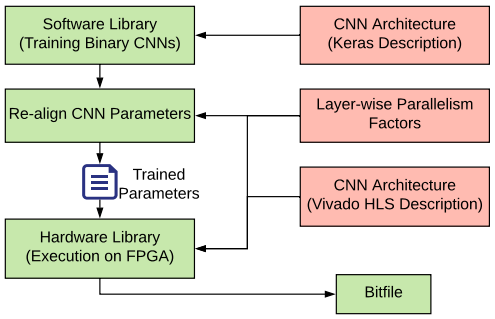


Figure 4: The global flow of ReBNet software training and hardware accelerator synthesis.

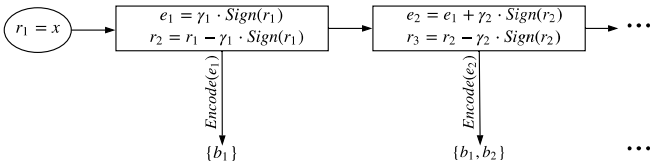


Figure 5: Schematic flow for computing an M-level residual binary approximate e_M and the corresponding encoded bits $\{b_1, b_2, \cdots, b_M\}$. As one goes deeper in levels, the estimation becomes more accurate. In this figure, we drop the subscript x from γ_{xi} and represent them as γ_i for simplicity.

there exist M scaling factors $\{\gamma_1, \gamma_2, \ldots, \gamma_M\}$. The first level of binarization takes input x and approximates it by either $+\gamma_1$ or $-\gamma_1$ based on the sign of x, then computes the residual error $r_2 = x - \gamma_1 \cdot Sign(x)$. The second level of binarization approximates r_2 by either $+\gamma_2$ or $-\gamma_2$ based on the sign of r_2, then computes the residual error $r_3 = r_2 - \gamma_2 \cdot Sign(r2)$. Repeating this process for M-times results in an M-level binarization for the input value. More formally, an input x can be approximated as $e = \sum_{i=1}^{M} \gamma_i \cdot Sign(r_i)$ with r_i being the i-th residual error. In ReBNet, the same set of scaling factors is used for all features corresponding to a certain CNN layer; therefore, the features can be encoded using M bits. Algorithm 1 presents the procedure for computing encoded bits $\{b_1, b_2, \cdots, b_n\}$.

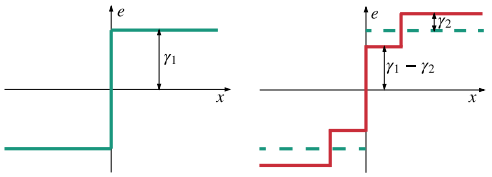
Algorithm 1 M-level residual encoding algorithm

inputs: $\gamma_1, \gamma_2, ..., \gamma_M$, x
outputs: $b_1, b_2, ..., b_M$

```
r ← x
for i = 1...M do
    b_i ← Binarize(Sign(r))
    r ← r − Sign(r) × γ_i
end for
```

Residual binary activation function: Similar to previous works which use the *Sign* function as the activation function, in this paper we use the residual binarization. The difference between our approach and the single-bit approach is illustrated in Figure 6. The activation function includes a set of scaling factors $\{\gamma_1, \gamma_2, \ldots, \gamma_M\}$ that should be learned during the training phase.

Multi-level XnorPopcount: In ReBNet, the dot product of



(a) (b)

Figure 6: Illustration of binarized activation function. (a) Conventional 1-level binarization. (b) Residual binarization with two levels. Note that the γ parameters are universal across all features (activations) of a particular layer.

an M-level residual-binarized feature vector $\vec{e}$ and a vector of binary weights $\vec{w}$ can be rendered using M subsequent *XnorPopcount* operations. Let $\vec{e} = \sum_{i=1}^{M} \gamma_{e_i} \vec{s}_{e_i}$ and $\vec{w} = \gamma_w \vec{s}_w$, where $\vec{s}_{e_i}$ denotes the i-th residual sign vector of the features and $\vec{s}_w$ is the sign vector of the weight vector. The dot product between $\vec{e}$ and $\vec{w}$ is computed as:

$$\begin{aligned} dot(\vec{w},\vec{e}) &= dot(\textstyle\sum_{i=1}^{M} \gamma_{e_i}\vec{s}_{e_i}, \gamma_w \vec{s}_w) \\ &= \textstyle\sum_{i=1}^{M} \gamma_{e_i}\gamma_w \; dot(\vec{s}_{e_i}, \vec{s}_w) \\ &= \textstyle\sum_{i=1}^{M} \gamma_{e_i}\gamma_w \; XnorPopcount(\vec{b}_{e_i}, \vec{b}_w), \end{aligned} \tag{4}$$

where $\{\vec{b}_{e_i}, \vec{b}_w\}$ are the binary encodings corresponding to $\{\vec{s}_{e_i}, \vec{s}_w\}$, respectively. Note that the subsequent *XnorPopcount* operations can be performed sequentially on the same hardware accelerator, providing a tradeoff between runtime and approximation accuracy.

B. Training residual binary CNNs

Training neural networks is generally performed in two steps. First, the output layer of the neural network is computed and a cost function is derived. This step is called forward propagation. In the second step, known as backward propagation, the gradient of the cost function with respect to the CNN parameters is computed and the parameters are updated accordingly to minimize the cost function.

For binary neural networks, the forward propagation step is performed using the binary approximations of the parameters and the features. In the backward propagation step, however, the full-precision parameters are updated. Once the network is trained, the full-precision parameters are binarized and used for efficient inference.

In this section, we derive the gradients for training residual binarized parameters in CNNs. Let $\mathscr{L}$ denote the cost function of the neural network. Consider a full-precision feature (weight) x approximated by a single binary value $e = \gamma \cdot Sign(x)$. The derivatives of the cost function with respect to γ is computed as follows:

$$\frac{\partial \mathscr{L}}{\partial \gamma} = \frac{\partial \mathscr{L}}{\partial e} \times \frac{\partial e}{\partial \gamma} = \frac{\partial \mathscr{L}}{\partial e} \times Sign(x) \tag{5}$$

Similarly for x:

$$\frac{\partial \mathscr{L}}{\partial x} = \frac{\partial \mathscr{L}}{\partial e} \times \frac{\partial e}{\partial Sign(x)} \times \frac{\partial Sign(x)}{\partial x} = \frac{\partial \mathscr{L}}{\partial e} \times \gamma \times 1_{|x|\leq 1} \tag{6}$$

where the derivative term $\frac{\partial Sign(x)}{\partial x}$ is approximated as suggested in [15]:

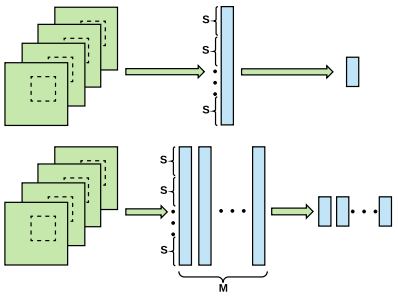


Figure 7: The baseline [17] sliding window unit (top), and our M-bit sliding window unit (bottom).

$$1_{|x|\leq 1} = \begin{cases} 1 & |x| \leq 1 \\ 0 & otherwise \end{cases} \tag{7}$$

In a multi-level binarization scheme, input x is approximated as $e = \sum_i \gamma_i \cdot Sign(r_i)$ with r_i denoting the $i-th$ residual error. The gradients can be computed similar to the derivatives above:

$$\begin{aligned} \frac{\partial \mathscr{L}}{\partial \gamma_i} &= \frac{\partial \mathscr{L}}{\partial e} \times Sign(r_i) \\ \frac{\partial \mathscr{L}}{\partial x} &= \frac{\partial \mathscr{L}}{\partial e} \times \textstyle\sum_i \gamma_i \times 1_{|r_i|\leq 1} \end{aligned} \tag{8}$$

Note that the training phase of ReBNet is performed on a floating point processor (e.g., CPU or GPU) and the parameters have full-precision values in this phase. After training, the binary approximates are loaded into the binary hardware accelerator.

C. Hardware Accelerator

ReBNet includes a high-level synthesis library for FPGA implementation of binary CNNs. The accelerator architecture is inspired by the one described in Figure 3 but provides a paradigm shift in term of overhead and scaling properties. Unlike the previous works that only accommodate single-bit binary CNNs, our accelerator offers a reconfigurable design that enjoys residual binarization with minimal hardware modification. In this section, we discuss different components of ReBNet accelerator and compare them with those of the baseline accelerator discussed in Section II-B.

Sliding window unit (SWU): Figure 7 depicts a high-level overview of the SWU. It slides the (dashed) windows through the feature maps, converts them into a binary vector, and splits the binary vector into S-bit words (S is the SIMD-width), which are sequentially sent to the MVTU. In the case of M-level residual binarization, the SWU still sends S-bit words but this time it transfers M such words sequentially. This approach enables a scalable design with a fixed SIMD-width; therefore, the reconfigurability of M incurs negligible hardware overhead while the runtime grows *linearly* with M.

Matrix vector threshold unit (MVTU): This unit is responsible for computing the neuron outputs in convolution and fully-connected layers. Similar to the baseline accelerator in Section II-B, our MVTU offers two levels of

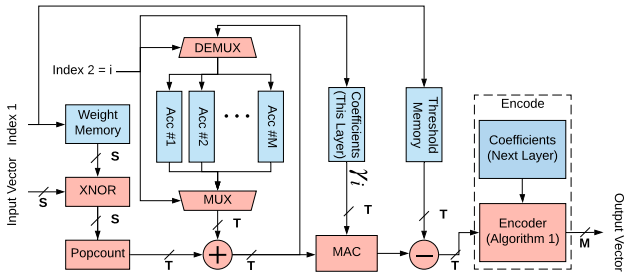


Figure 8: Architecture of processing element in ReBNet. "S" is the SIMD-width, "T" is the fixed-point bitwidth in computations, and "M" is the number of residual levels.

parallelism. The internal architecture of PE is shown in Figure 8. Compared to the baseline unit of Figure 3-(c), our PE maintains multiple accumulators to store the popcount values corresponding to each of the M residual binarization levels. Once all M popcounts are accumulated, they are multiplied by the corresponding scaling factors (γ_i) and summed together via the *MAC* unit. Batch normalization is implemented using the threshold memory. Finally, the encoder module computes the M-bit residual representation fed to the next layer based on Algorithm 1.

Max-pooling: The original 1-bit binarization allows us to implement max-pooling layers using simple OR operations. For M-level binarization, however, max-pooling layers should be implemented using comparators since performing Boolean OR over the binary encodings is no longer equivalent to performing max-pooling over the features. Nevertheless, the pooling operation can be performed over the encoded values directly. Assume full-precision values e_x and e_y, with M-bit binary encodings b_x and b_y, respectively. Considering ordered positive γ_i values (i.e. $\gamma_1 > \gamma_2 > \cdots > \gamma_l > 0$), one can conclude that if $b_x < b_y$ then $e_x < e_y$; therefore, instead of comparing the original values (e_x, e_y), we compare the binary encodings (b_x, b_y) which require small comparators.

IV. Experiments

We implement our Software API using Keras [21] library. Our hardware API is implemented in Vivado HLS and the synthesis reports (resource utilization and latency) for the FPGA accelerator are obtained using Vivado Design Suite [22]. We compare ReBNet with the prior art in terms of accuracy, FPGA resource utilization, execution (inference) time on the FPGA accelerator, and scalability. Proof-of-concept evaluations are performed for four datasets: CIFAR-10, SVHN, MNIST, and ILSVRC-2012 (Imagenet). Our hardware results are compared against the FINN design [17], which uses the training method of Binarynet [15]. The FINN paper only evaluates the first three applications. For the last dataset (Imagenet), we implemented the corresponding FINN accelerator using their open-source library. Throughout this section, M denotes the number of residual binarizations and $T = 24$ is the fixed-point bitwidth of features.

We implement a small network consisting only fully-connected layers for MNIST, which we call *Arch-1*. The CIFAR-10 and SVHN datasets are evaluated on a medium-sized convolutional neural network named *Arch-2*. Finally, the Imagenet dataset is evaluated on a relatively large network called *Arch-3*. Table I outlines the three neural network architectures and the corresponding parallelism factors for different layers. The parallelism factors only affect hardware performance and have no effect on the CNN accuracy. For *Arch-1* and *Arch-2*, we set the parallelism factors exactly the same as the baselines in the FINN design. For *Arch-3*, we configure the parallelism factors ourselves. Each of these architectures is implemented on a different FPGA evaluation board outlined in Table II.

Table I: Network architectures for evaluation benchmarks. $C64(P,S)$ denotes a convolution with 64 output channels; the kernel size of the convolution is 3×3 and the stride is 1 unless stated otherwise. $D512(P,S)$ means a fully-connected layer with 512 outputs. The numbers (P,S) represent the two parallelism factors (PE-count, SIMD-width) for the layers. $MP2$ stands for 2×2 max pooling, *BN* represents batch normalization. Residual Binarization is shown using *RB*.

Benchmark	CNN Architecture
MNIST	784 (input)- D256(16,64)- BN- RB- D256(32,16)- BN- RB- D256(16,32)- BN- RB- D10(16,4)- BN- Softmax
CIFAR-10 & SVHN	$3 \times 32 \times 32$ (input)- C64(16,3)- BN- RB- C64(32,32)- BN- RB- MP2- C128(16,32)- BN- RB- C128(16,32)- BN- RB- MP2- C256(4,32)- BN- RB- C256(1,32)- BN- RB- D512(1,4)- BN- RB- D512(1,8)- BN- RB- D10(4,1)- BN- Softmax
Imagenet	$3 \times 224 \times 224$ (input)- C96(32,33)*- BN- RB- MP3- C256(64,25)**- BN- RB- MP3- C384(64,54)- BN- RB- C384(64,36)- BN- RB- C256(64,36)- BN- RB- MP3- D4096(128,64)- BN- RB- D4096(128,64)- BN- RB- D1000(40,50)- BN- Softmax

*Stride is 4 for this layer, filter size is 11×11.
** Filter size is 5×5 for this layer.

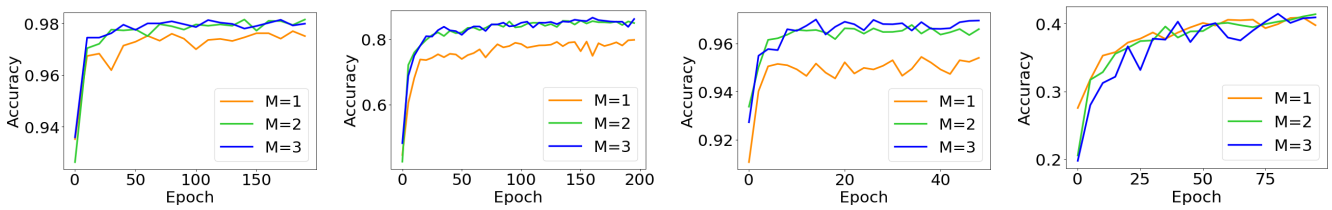
A. Accuracy and Throughput Tradeoff

The accuracy of a binary CNN depends on the network architecture and the number of training epochs. The training phase is performed on a computer, not the FPGA accelerator. However, we provide the learning curve of ReBNet with different numbers of residual levels to show that residual binarization increases the convergence speed of binary CNNs.

Figure 9 presents the learning curves (accuracy versus epoch) for the four applications. As can be seen, increasing M improves both the convergence speed and the final achievable accuracy. On the other hand, a higher M requires more computation time during the execution (after the accelerator is implemented on FPGA). Table III compares our accuracy and throughput with the conventional binarization method proposed in [15] and implemented (on FPGA) by the FINN design [17]. Even with 1 level of binarization, our method surpasses the FINN design in terms of accuracy, which is due to the trainable scaling factors of ReBNet. As can be seen, we can improve the accuracy of ReBNet by increasing

Table II: Platform details in terms of block ram (BRAM), DSP, flip-flop (FF), and look-up table (LUT) resources.

Application	Platform	BRAM	DSP	FF	LUT
Imagenet	Virtex VCU108	3456	768	1075200	537600
CIFAR-10 & SVHN	Zynq ZC702	280	220	106400	53200
MNIST	Spartan XC7S50	120	150	65200	32600



(a) MNIST (b) CIFAR-10 (c) SVHN (d) Imagenet

Figure 9: Top-1 accuracy of ReBNet for different benchmarks. M denotes the number of residual binarization levels. MNIST, CIFAR-10, SVHN, and Imagenet models are trained for 200, 200, 50, and 100 epochs, respectively.

Table III: Comparison of accuracy and throughput between the FINN design and ReBNet. The baseline accuracy for the first three datasets is reported by [17]. The authors of XNOR-net [16] implement Binarynet [15] and report the top-1 accuracy of 27.9% for Imagenet. As M grows, the accuracy of ReBNet is increased. Note that for CNN applications, even incremental accuracy gains are of importance [23]. The clock frequency of our designs is 200Mhz.

			MNIST	CIFAR10	SVHN	Imagenet
FINN		Accuracy (%)	95.83	80.1	94.9	27.9
		Throughput (samples/sec)	6.3×10^5	6×10^3	6×10^3	5.2×10^2
Ours	M=1	Accuracy (%)	97.87	80.59	95.46	40.89
		Throughput (samples/sec)	6.4×10^5	6×10^3	6×10^3	5.3×10^2
	M=2	Accuracy (%)	98.29	85.94	96.82	41.37
		Throughput (samples/sec)	3.3×10^5	3×10^3	3×10^3	2.6×10^2
	M=3	Accuracy (%)	98.25	86.98	97.00	41.43
		Throughput (samples/sec)	2.3×10^5	2×10^3	2×10^3	1.7×10^2

M, which, in turn, reduces the throughput. We evaluate the extra hardware cost of having $M \geq 2$ in the next section.

B. Hardware Evaluation and Reconfigurability

Figure 10-(a, b, c) compares the resource utilization of ReBNet with the baseline FINN design for each of the three network architectures. Indeed, compared to the less accurate FINN design, ReBNet has an added area cost mainly due to the design reconfigurability. Recall the PE design in Figure 8; The MAC block, the thresholding unit, and the encoder unit altogether require extra computation resources, which explains why ReBNet has extra DSP utilization compared to FINN. The BRAM, FF, and LUT utilizations of ReBNet increase with M because the PEs of the design would require more accumulators and a more complex control logic. Recall that the FINN design has lower accuracy compared to ReBNet (See Table III). In fact, If we were to enhance the accuracy of FINN (by training wider networks), the increase in resource utilizations would exceed the limitations of the board as discussed in Section IV-C.

Figure 10-(d) compares the latency (runtime) of ReBNet with FINN. The latency of our accelerator with $M = 1$ is the same as that of FINN; this is due to the fact that the runtime overhead of the extra modules of ReBNet, i.e., the MAC and encoder modules, is negligible compared to the *XnorPopcount* operations. Overall, the latency of ReBNet grows linearly with M since the number of *XnorPopcount* operations (for a single dot product) is equal to M (see Equation 4). As such, ReBNet offers a tradeoff between accuracy and latency.

C. Effect of CNN size

As we demonstrated before, ReBNet improves the accuracy using more residual binarization levels. Another method for enhancing the accuracy is to train a *wide* network as analyzed in [17]. In our notation, a reference network is widened by the scale S if all hidden layers are expanded by a factor of S, i.e., the widened network has S-times more output channels in convolution layers and S-times more output neurons in fully-connected layers. Consider the CIFAR-10 architecture *Arch-2* in Table I. In Figure 11, we depict the accuracy of the *Arch-2* network with $M = 1$ that is widened with different scales (variable S). The accuracies of the 2-level and 3-level binarization schemes correspond to the reference *Arch-2* with no widening ($S = 1$). It can be seen that the 1-level network should be widened by scales of 2.25 and 2.75 to achieve the same accuracy as the 2-level and 3-level networks, respectively.

In Figure 12, we compare the reference 2-level and 3-level residual binary networks with their widened 1-level counterparts that achieve the same accuracies. In this Figure, the resource utilization (i.e., BRAM, DSP, FF, and LUT) are normalized by the maximum resources available on the FPGA platform and the latency is normalized by that of the reference CNN (i.e., $S = 1$ and $M = 1$). It is worth noting that widening a network by a factor of S increases the computation and memory burden by a factor of S^2 while adding to the number of residual levels M incurs a linear overhead. As such, widening the network explodes the resource utilization and exceeds the platform constraints. In addition, the latency of the wide CNN increases quadratically with S whereas in ReBNet the latency increases linearly with M. Therefore, ReBNet achieves higher throughput and lower resource utilization for a certain classification accuracy.

D. Effect of Online Scaling Factors

Authors of XNOR-net [16] suggest computing the scaling factors during the execution. Here, we show that such computations add significant hardware cost to the binary CNN accelerator. We assume that the runtime and DSP utilization overhead of online scaling factor computation are negligible. This section only considers the excessive memory footprint of online scaling factors. Consider a convolution layer with $K \times K$ filters and input image $I_{H\times H\times F}$ to be binarized using online scaling factors. Recall the SWU module that is responsible for generating input vectors for the MVTU. The SWU is implemented using streaming buffers, meaning

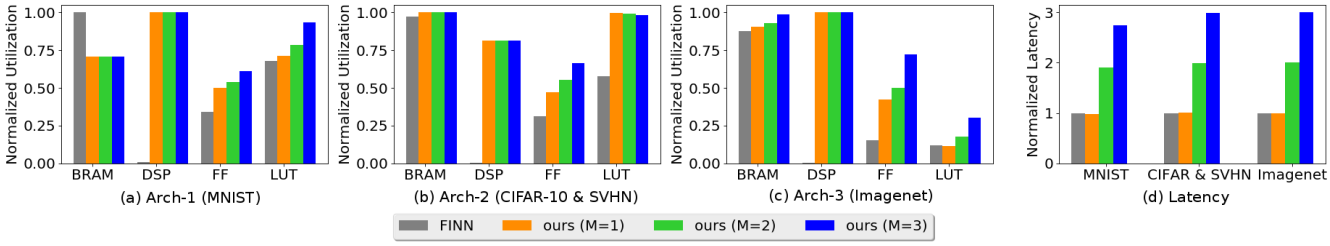


Figure 10: (a, b, c) Normalized hardware utilization and (d) latency of ReBNet compared to FINN. The utilizations are normalized by maximum resources available on each platform. The latencies are normalized by the latency of FINN. In ReBNet, the hardware overhead of reconfigurability is minimal and latency grows linearly with M. Note that the accuracy of FINN in all benchmarks is lower than ReBNet with $M = 1$ and as we increase M, the accuracy is improved (see Table III). FINN, however, incurs excessive area and latency costs to increase the accuracy and thus is not scalable (see Figure 12).

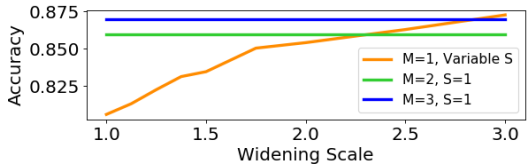


Figure 11: Effect of network widening on the accuracy of the CIFAR-10 benchmark. The widened CNNs are trained for 200 epochs and the best accuracy is accounted. Although wider single-level networks are capable of achieving the same accuracy as multi-level networks, the hardware cost and latency of wider networks are significantly larger than those of multi-level CNNs (see Figure 12).

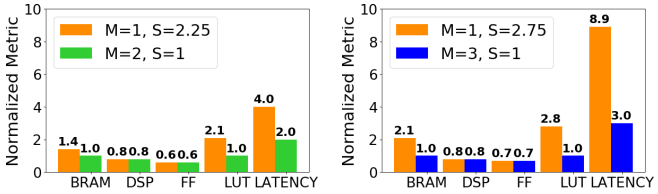


Figure 12: Hardware cost and latency comparison of residual binarized CNN and widened CNN with the same accuracy. (a) *Arch-2* widened by a scale of 2.25 compared to *Arch-2* with 2-level binarization. (b) *Arch-2* widened by a scale of 2.75 compared to *Arch-2* with 3-level binarization. Network widening explodes the resource utlization and latency of the accelerator while ReBNet offers a scalable solution by providing a tradeoff between accuracy and latency.

that it does not store the whole input image at once. Instead, it only buffers one row of sliding windows at a time. If the features are binary, then a buffer of size KHF suffices in the SWU. In the case of XNOR-net, assuming that the fixed-point values are represented with T bits, the binarization operation requires two additional buffers of size $KHFT$ (for fixed-point features) and KHT (for scaling factors). The ratio of the memory footprint with and without online scaling factor computation is as follows:

$$\frac{Mem_{XNOR-net}}{Mem_{ours}} = \frac{KHF+KHFT+KHT}{KHF} = 1+T+\frac{T}{F}\ , \qquad (9)$$

Table IV: Memory utilization of the baseline binary CNN accelerator and estimated overhead of online scaling factor computation suggested by XNOR-net [16]. The bitwidth of fixed-point features is $T = 24$ in our designs.

	Arch2	Arch3
FF Utilization without XNOR-net(%)	47.02	42.49
FF Utilization with XNOR-net (%)	373.83	62.69
BRAM Utilization without XNOR-net (%)	99.64	90.42
BRAM Utilization with XNOR-net (%)	236.78	283.47

where T is the fixed-point representation bitwidth and F is the number of input channels. For a single SWU, if the flip-flop and/or BRAM utilization is $P\%$, the overhead would be $(T+\frac{T}{F})P$. Table IV presents the estimated memory overhead for two of our architectures that include convolution layers. The overall overhead is obtained by summing the overheads corresponding to all SWUs: $\sum_{i=2}^{L}(T+\frac{T}{F_i})P_i$, where L is the total number of convolution layers. Note that the first layer is not considered in this summation because its input is not binarized and does not need scaling factor computation.

V. Related Work

Training CNNs with binary weights and/or activations has been the subject of very recent works [24], [16], [15], [17]. The authors of Binaryconnect [24] suggest a probabilistic methodology that leverages the full-precision weights to generate binary representatives during forward pass while in the back-propagation the full-precision weights are updated. The authors of [15] introduced binarization of both weights and activation of CNNs. The authors also suggest replacing the costly dot products by *XnorPopcount* operations. XNOR-net [16] proposes to use scale factors during training, which results in an improved accuracy. The authors of XNOR-net do not provide a hardware accelerator for their binarized CNN. Although XNOR-net achieves higher accuracy compared to the available literature, it sacrifices the simplicity of the hardware accelerator for binary CNNs due to two reasons: (i) It utilizes multiple scaling factors for each parameter set (e.g. one scaling factor for each column of a weight matrix), which would increase the memory footprint and logic utilization. (ii) The online computation of the scaling factors for the activations requires a significant number of full-precision operations.

The aforementioned works propose optimization solutions that enable the use of binarized values in CNNs which, in turn, enable the design of simple and efficient hardware accelerators. The downside of these works is that, aside from changing the architecture of the CNN [17], they do not offer any other reconfigurability in their designs. Providing an easily reconfigurable architecture is the key to adapting the accelerator to the application requirements.

In a separate research track, the reconfigurability of CNN accelerators has been investigated. Using adaptive low bitwidth representations for compressing the parameters and/or simplifying the pertinent arithmetic operations is investigated in [25], [26], [27]. The proposed solutions, however, do not enjoy the same simplified *XnorPopcount* operations as in binarized CNNs. Considering the aforementioned works, the reconfigurability and scalability of binary CNN accelerators require further investigation. To the best of our knowledge, ReBNet is the first scalable binary CNN solution that embeds reconfigurability and, at the same time, enjoys the benefits of binarized CNNs.

VI. Conclusion

This paper introduces ReBNet, a novel reconfigurable binarization scheme which aims to improve the convergence rate and the final accuracy of binary CNNs. Many existing works have tried to compensate for the accuracy loss of binary CNNs but did not consider hardware implications of their proposals. As a result, they suffer from a degraded performance and resource efficiency. We argue that a practical and scalable effort towards enhancing the accuracy of binary CNNs should not add significant fixed-point operations and/or memory overhead to the computation flow of binary CNNs. In this paper, we evaluated the hardware cost of two of the state-of-the-art methods (XNOR-net and wide-networks) for enhancing the accuracy of binary CNNs and showed that their implementation overhead is considerable. Unlike prior methods, ReBNet does not sacrifice the simplicity of the hardware architecture to achieve a higher accuracy. Our work is accompanied by an API that facilitates training and design of residual binary networks. The API is open-source to foster the research in reconfigurable machine-learning on FPGA platforms.

References

[1] D. Li, X. Wang, and D. Kong, "Deeprebirth: Accelerating deep neural network execution on mobile devices," *arXiv preprint arXiv:1708.04728*, 2017.

[2] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," *arXiv preprint arXiv:1707.01083*, 2017.

[3] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[4] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 161–170, ACM, 2015.

[5] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating deep convolutional neural networks using specialized hardware," *Microsoft Research Whitepaper*, vol. 2, no. 11, 2015.

[6] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao, "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 16–25, ACM, 2016.

[7] Y.-D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin, "Compression of deep convolutional neural networks for fast and low power mobile applications," *arXiv preprint arXiv:1511.06530*, 2015.

[8] X. Zhang, J. Zou, X. Ming, K. He, and J. Sun, "Efficient and accurate approximations of nonlinear convolutional networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1984–1992, 2015.

[9] M. Nazemi, A. E. Eshratifar, and M. Pedram, "A hardware-friendly algorithm for scalable training and deployment of dimensionality reduction models on FPGA," in *Proceedings of the 19th IEEE International Symposium on Quality Electronic Design*, 2018.

[10] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *arXiv preprint arXiv:1609.07061*, 2016.

[11] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[12] M. Samragh, M. Ghasemzadeh, and F. Koushanfar, "Customizing neural networks for efficient fpga implementation," in *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on*, pp. 85–92, IEEE, 2017.

[13] B. Liu, M. Wang, H. Foroosh, M. Tappen, and M. Pensky, "Sparse convolutional neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 806–814, 2015.

[14] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Advances in Neural Information Processing Systems*, pp. 2074–2082, 2016.

[15] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1," *arXiv preprint arXiv:1602.02830*, 2016.

[16] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European Conference on Computer Vision*, pp. 525–542, Springer, 2016.

[17] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 65–74, ACM, 2017.

[18] W. Tang, G. Hua, and L. Wang, "How to train a compact binary neural network with high accuracy?," in *AAAI*, pp. 2625–2631, 2017.

[19] H. Alemdar, V. Leroy, A. Prost-Boucle, and F. Pétrot, "Ternary neural networks for resource-efficient ai applications," in *Neural Networks (IJCNN), 2017 International Joint Conference on*, pp. 2547–2554, IEEE, 2017.

[20] N. Mellempudi, A. Kundu, D. Mudigere, D. Das, B. Kaul, and P. Dubey, "Ternary neural networks with fine-grained quantization," *arXiv preprint arXiv:1705.01462*, 2017.

[21] F. Chollet *et al.*, "Keras." https://github.com/fchollet/keras, 2015.

[22] Xilinx, "Vivado." https://www.xilinx.com/products/design-tools/vivado.html, 2017.

[23] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[24] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in Neural Information Processing Systems*, pp. 3123–3131, 2015.

[25] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *arXiv preprint arXiv:1606.06160*, 2016.

[26] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 243–254, IEEE Press, 2016.

[27] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng, "Quantized convolutional neural networks for mobile devices," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4820–4828, 2016.