A Learning Framework for *n*-Bit Quantized Neural Networks Toward FPGAs

Jun Chen, Liang Liu, Yong Liu, *Member, IEEE*, and Xianfang Zeng

***Abstract*— The quantized neural network (QNN) is an efficient approach for network compression and can be widely used in the implementation of field-programmable gate arrays (FPGAs). This article proposes a novel learning framework for *n*-bit QNNs, whose weights are constrained to the power of two. To solve the gradient vanishing problem, we propose a reconstructed gradient function for QNNs in the back-propagation algorithm that can directly get the real gradient rather than estimating an approximate gradient of the expected loss. We also propose a novel QNN structure named *n*-BQ-NN, which uses shift operation to replace the multiply operation and is more suitable for the inference on FPGAs. Furthermore, we also design a shift vector processing element (SVPE) array to replace all 16-bit multiplications with SHIFT operations in convolution operation on FPGAs. We also carry out comparable experiments to evaluate our framework. The experimental results show that the quantized models of ResNet, DenseNet, and AlexNet through our learning framework can achieve almost the same accuracies with the original full-precision models. Moreover, when using our learning framework to train our *n*-BQ-NN from scratch, it can achieve state-of-the-art results compared with typical low-precision QNNs. Experiments on Xilinx ZCU102 platform show that our *n*-BQ-NN with our SVPE can execute 2.9 times faster than that with the vector processing element (VPE) in inference. As the SHIFT operation in our SVPE array will not consume digital signal processing (DSP) resources on FPGAs, the experiments have shown that the use of SVPE array also reduces average energy consumption to 68.7% of the VPE array with 16 bit.**

***Index Terms*— Deep compression, deep learning, field-programmable gate array (FPGA), quantized neural network (QNN).**

I. Introduction

DEEP convolutional neural networks (CNNs) have substantially become the dominant artificial intelligence (AI) approach for a variety of computer vision tasks such as image classification [1]–[3], face recognition [4], [5], semantic segmentation [6], [7], and object detection [8], [9]. The significant accuracy improvement of CNNs brings with the cost of huge computational complexity, resource, and power consumption as it requires a comprehensive estimation of all the scopes within the feature maps [10], [11]. For example, the AlexNet model is over 200 MB, and the VGG-16 model is over 500 MB [10]. Toward such overwhelming resources and computation pressure, hardware accelerators such as GPUs, field-programmable gate arrays (FPGAs), and ASICs have been applied to accelerate CNNs. Among these accelerators, FPGAs have emerged as one of the popular solutions when considering both the reprogramability and energy efficiency.

Implementing CNN on FPGAs is not an efficient practice due to limited resources and bandwidth. Thus, quantized neural network (QNN) is a good choice for FPGA implementation, which simultaneously gives consideration to computational efficiency, resources, and classification accuracy in inference. In general, QNNs can be achieved in two ways: 1) an estimator is used to estimate the gradient of the expected loss to solve the problem of gradient vanishing so that QNNs can be trained from scratch with the help of this estimator and 2) fine-tuning on a pretrained full-precision model obtains QNNs that bypasses the problem of gradient vanishing. Although the first method estimates a gradient, which makes it possible to train QNNs from scratch, the gradient of expected loss obtained by estimators has a noise source compared to the real gradient that causes a gap in classification accuracy between the QNNs and full-precision CNNs. The second method fine-tunes QNNs on a pretrained full-precision model that solves the problem of classification accuracy better, but a challenging factor is that the structure of QNNs is limited by the original structure of the pretrained CNNs model, and the structure of QNNs cannot be flexibly adjusted. Due to the constraints of computational resources and computational efficiency on FPGAs, it is inevitable to adjust the network structure for the hardware environment. In order to transform different CNNs into QNNs that can run efficiently on FPGAs, it is essential for a general learning framework to solve the above two challenges [12]–[15].

In this article, we propose a novel learning framework for *n*-bit QNNs, whose weights are constrained to the power of two $(\pm 2^{-0}, \pm 2^{-1}, \ldots, 0)$. We introduce a reconstructed gradient function in the back-propagation algorithm that can directly get the real gradient, rather than the estimated gradient given by estimators. Thus, the QNNs trained by our framework will be more accurate. At the same time, QNNs after adjusting the structure can continue to fine-tune with our framework. The learning framework is applied to train our proposed *n*-BQ-NN, which is suitable for efficient implementation on FPGAs. We also evaluate the effectiveness of our approach on state-of-the-art networks such as ResNet [16], DenseNet [17], and AlexNet [1]. The main contributions of this article are summarized as follows.

1) We propose a novel learning framework for *n*-bit QNNs. In this framework, we propose a reconstructed gradient

Manuscript received May 23, 2019; revised October 11, 2019 and February 2, 2020; accepted March 5, 2020. This work was supported in part by the Development Program of Guangdong Province of China under Grant 2019B010120001 and in part by the Key Research and Development Project of Zhejiang Province under Grant 2019C01004. *(Corresponding author: Yong Liu.)*

The authors are with the Institute of Cyber-Systems and Control, Zhejiang University, Hangzhou 310027, China (e-mail: yongliu@iipc.zju.edu.cn).

Color versions of one or more of the figures in this article are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNNLS.2020.2980041

function in the back-propagation algorithm, which can overcome the gradient vanishing problem during training the QNNs and can calculate the accurate gradient compared with the estimators-based approaches. We achieve state-of-the-art results compared with typically low-precision QNNs.

2) We propose a highly efficient QNN structure called n-BQ-NN for FPGAs. Our proposed architecture, which consists entirely of convolutional layers and implements a uniform convolution kernel, can maximize the resource utilization and improve the parallel computational efficiency on FPGAs while preserving the accuracy of QNNs.
3) We propose a novel shift-vector processing element (SVPE) array for FPGAs, which replaces the multiplication with the SHIFT operation when calculating convolution operation on FPGAs. The computational efficiency of our SVPE array can achieve a performance of 2.9 times higher than that of the vector processing element (VPE) array in the case of the same network structure on FPGAs.

The rest of this article is organized as follows. Section II summarizes related prior works on QNNs and FPGAs. Our learning framework is presented in Section III. In Section IV, we demonstrate the effectiveness of our learning framework via comparable experiments. We theoretically analyze and practically test the computational efficiency of our n-BQ-NN using our quantization method in Section V. The conclusion is given in Section VI.

II. Related Work

A. Learning for QNNs

Since the amount of the model capacity is too large, it is necessary to cut down it to perform CNNs on FPGAs, which is consistent with the purpose of deep compression. In general, deep compression can be divided into three categories, i.e., pruning, Huffman coding, and quantization. The pruning method will simplify the deep neural network by cutting off the network connections with small weights on the normal trained network [18]–[20]. The Huffman coding method is an optimal code used for lossless data compression [21], which uses entropy to encode source symbols by variable-length codewords. Han *et al.* [20] show that 20%–30% of the network storage will be saved after Huffman coding the nonuniformly distributed values. When considering perform compressed networks on FPGAs, the network after pruning is an asymmetric structure, which is unsuitable for hardware implementation, and the Huffman coding may only be regarded as a postcompression combined with the other two compression methods, so most of the hardware accelerators will focus on the quantization method.

The quantization-based method normally employ the low-precision weights, varied from 1 to 5 bits, to represent the CNNs [15], [22]–[28]. Some studies train QNNs from scratch by estimating the gradient of expected loss based on straight-through estimator [15], [22], [23], [27]. For example, Courbariaux *et al.* [15] train a classification neural network from scratch with 1-bit weight and activation, which can run seven times faster than the CNNs. Choi *et al.* [23] propose a neural quantization scheme called parameter clipping activation, which uses a parameter to find the optimal quantization scale for arbitrary bit-width activations. Choi *et al.* [22] introduce a novel technique called statistics-aware weight binning, which finds the optimal scaling factor based on statistical characteristics of the distribution of the weights to minimize the quantization error. The QNNs trained by the above quantization methods only accelerate the inference; Zhou *et al.* [27] propose a DoReFa-Net that can accelerate both training and inference by low bit-width weights, activations, and gradients, respectively. However, these estimator-based methods have a noise compared to the real gradient. Thus, these QNNs cannot achieve an ideal classification accuracy, especially on multiclassification data sets such as CIFAR-100.

Some other quantization methods are dedicated to design special strategies to fine-tune QNNs, which will not rely on the backpropagation algorithm and can bypass the problem of gradient vanishing [26], [28], [29]. They can achieve much better accuracy as they are independent of estimators. For example, Park *et al.* [29] propose precision highway that has an end-to-end high-precision information flow for ultralow-precision computation. This linear weight quantization method is based on the assumption that the weight distribution is the Laplace distribution. Recently, Zhou *et al.* [26] propose an incremental network quantization method, which converts pretrained full-precision CNNs model into a low-precision model, where the weights are constrained to the power of two or zero. It has been studied that there will be little loss on the classification accuracies when using 2–5-bit low-precision weight [26], [27]. However, these quantization methods will depend on the pretrained network structure rather than the backpropagation algorithm, which will be difficult to satisfy the network-structure-optimization requirements due to the hardware limitation.

B. CNNs Implemented by FPGAs

Considering the inference, the CNNs have a highly hierarchical structure of multiple feature maps, whose structure exposes a large amount of parallelism that makes CNNs very suitable for FPGA implementation. This structure builds on the accumulation of a huge number of convolutions that will consume a huge number of floating-point resources on FPGAs. In addition, the structure of CNNs often contains many convolutional layers. Thus, the convolution module with different parameters needs to be executed iteratively during the inference. Frequent execution of data caching and parameter loading will be limited by the bandwidth. Therefore, in many studies, their hardware structures of CNNs are designed mainly for the two bottlenecks of floating-point resources and bandwidth [12], [13], [16], [17].

In terms of optimizing for floating-point resources, Lu *et al.* [30] design a fast Winograd algorithm, which can decrease the use of floating-point resources on FPGAs and reduce the complexity of convolution dramatically. Simultaneously, they also give the formula for estimating the computational efficiency, which demonstrates that the fast Winograd

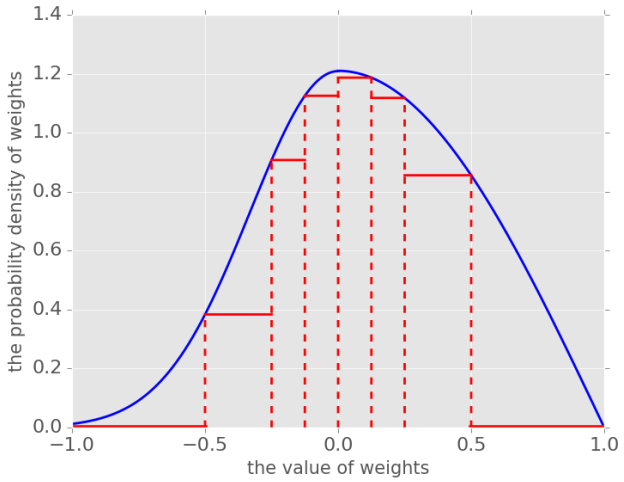

Fig. 1. For any full-precision weight distribution (indicated by the blue curve in the figure) trained by CNN model, nonuniform sampling can be used to approximate the full-precision distribution, which represented by the red curve in the figure.

algorithm is more efficient than the conventional convolutional algorithm due to the use of fewer floating-point resources on FPGAs. Meloni *et al.* [13] present an accelerator configuration for CNNs that reach more than 97% digital signal processing (DSP) resource utilization at a 150-MHz operating frequency with 16-bit precision. They show that the floating-point resource utilization is the highest when executing 3×3 filters on FPGAs.

Other studies have focused on optimizing the data scheduling structure to reduce the impact of the bandwidth. For example, Sankaradas *et al.* [14] implement a VPE array coprocessor, which can accelerate the CNNs by optimizing the cache between distributed off-chip memory banks and on-chip computing elements on FPGAs. Peemen *et al.* [12] show that their scheduler prefers to use only convolutional layers without fully connected layers on FPGAs, which can maximize the efficiency of on-chip memories by reducing the impact of the bandwidth bottleneck.

The crucial issue with the above methods is that they usually only consider the bottleneck at a single level and fail to coordinate these two constraints to improve the computational efficiency of the hardware accelerators. In this article, we reduce the impact of the above two constraints by introducing the QNNs into FPGAs, which provides a new idea to deal with the above two bottlenecks. Since the weights in our QNNs are quantized to the power of two, the quantized weights directly reduce the bandwidth required to load the weights. In addition, the use of the quantized weights can translate the multiplication into shifting in the convolution module, which greatly reduces the use of floating-point resources.

III. n-BQ-NN

A. Fundamental Idea of Our n-BQ-NN

The main idea of our n-BQ-NN is based on Fig. 1, which shows that the information loss led by the quantization method with the power of 2 can be interpreted as the sampling loss caused by nonuniform sampling. In fact, the weights of CNNs with large absolute values will be dominant to the overall classification accuracy of the networks although these weights with large values only account for a small ratio among all the weights [11], [31]. For an arbitrary probability density function of the weights in a neural network, denoted by $\phi(x)$, we can use the blue curve in Fig. 1 to represent $\phi(x)$ that meets $\int_{-1}^{1} \phi(x)dx = 1$. In this way, we can calculate the sampling loss $\Phi(x)$, which can be represented as the area between two distributions (red and blue curves) in Fig. 1. By calculating, the sampling loss $\Phi(x)$ is represented as the following recursive formula, where n is the quantized bit width of the weights:

$$\begin{cases} \Phi(1) = 1-\phi(2^{-1}), & n = 1 \\ \Phi(n) = \Phi(n-1) + 2^{1-n}\phi(2^{1-n}) \\ \qquad\quad - 2^{1-n}\phi(2^{-n}), & n > 1. \end{cases} \tag{1}$$

It can be seen from the above formula that the sampling loss always decreases as the increasing of quantized bit width of the weights, which indicates that the sampling loss is negatively related to the quantized bit width of weights. However, the bit width is limited and needs to reduce as much as possible in QNNs. Thus, finding the best balance between quantized bit width and the sampling loss is the key to balancing the performance, speed, and resources of QNNs. We define the $\mathcal{L}(n) = 2^{1-n}[\phi(2^{1-n}) - \phi(2^{-n})]$ as the variation between two sampling losses, $\Phi(n)$ and $\Phi(n-1)$, from (1). Then, we can prove that $\mathcal{L}(4)$ will approach to zero in our quantization method with the power of 2, which can be ensured by the Theorem 1. Therefore, continuing to increase the quantized bit width of n after 3 is not helpful to decrease the sampling loss.

Theorem 1: $0 < |\mathcal{L}(4)| < 7.8 \times 10^{-3}$.

Proof: We use the Taylor expansion with Peano residuals to represent the probability density function $\phi(x)$

$$\begin{aligned} \phi(2^{-n}) &= \phi(0) - ln2\phi'(0)n2^{-n} + o(n2^{-n}) \\ \phi(2^{1-n}) &= \phi(0) - ln2\phi'(0)n2^{1-n} + o(n2^{1-n}). \end{aligned} \tag{2}$$

Substituting (2) into $\mathcal{L}(n)$, we get

$$\begin{aligned} \mathcal{L}(n) &= 2^{1-n}[\phi(2^{1-n}) - \phi(2^{-n})] \\ &= 2^{1-n}[ln2\phi'(0)n2^{-n} - \cancel{o(n2^{-n})} - ln2\phi'(0)n2^{1-n} \\ &\qquad + \cancel{o(n2^{1-n})}] \\ &\approx -ln2\phi'(0)n2^{1-2n}. \end{aligned} \tag{3}$$

Since $0 < \Phi(n) < 1$, we deduce that $0 < |\mathcal{L}(2)| < 1$ and $0 < ln2\phi'(0) < (1/4)$. In final, we get $0 < |\mathcal{L}(4)| < (1/128)$ by substituting the range of $ln2\phi'(0)$ into $\mathcal{L}(4)$ due to (3). □

From the hardware perspective, the resource consumption of SHIFT operation is much less than multiplication, so our intention is to use the SHIFT operation instead of multiplication. Consider that the shift right operation will make the weights exceed the constraint range of $(-1,1)$; thus, all SHIFT operations are shift left and every quantized weight is chosen from the entries $(\pm 2^{-0}, \pm 2^{-1}, \ldots, \pm 2^{-i}, 0)$,

where $\pm 2^{-i}$ indicates that its multiplication can be calculated by $\ll i$ and 0 indicates that no operations are required. Our n-BQ-NN quantizes the weights to the entries, which are encoded to n-bit and suitable for hardware computation. Under such circumstance, the staircase function staircase(W) can be used to describe our n-bit quantized weights as (4) (typically, n is greater than 1, and staircase(W) is degraded to sign(W) if n is equal to 1), where W are full-precision weights

$$\text{staircase}(W) = \begin{cases} 2^{-i}\,\text{sign}(W), & \Delta_{i+1} \le |W| < \Delta_i \\ 0, & |W| < \Delta_r. \end{cases} \tag{4}$$

Here, i is taken from $r-1$ to 0 in turn, where $r = 2^{n-1}-1$ and sign(W) is the sign function

$$\text{sign}(W) = \begin{cases} +1, & W \ge 0 \\ -1, & W < 0. \end{cases} \tag{5}$$

B. Gradients Computation in n-BQ-NN

In order to facilitate the discussion as follows, we need to define some variables first, where W_{jk}^l represents the weight that connects the kth neuron of the $(l-1)$th layer to the jth neuron of the lth layer, b_j^l represents the bias of the jth neuron of the lth layer, z_j^l represents the input of the jth neuron of the lth layer ($z_j^l = \sum_k W_{jk}^l a_k^{l-1} + b_j^l$), a_j^l represents the output of the jth neuron of the lth layer ($a_j^l = \theta(z_j^l)$), and θ is the activation function.

We have also to add an extra quantized weight so that we can train our n-BQ-NN, where the quantized weight is shown as follows:

$$\hat{W}_{jk}^l = \text{staircase}\left(W_{jk}^l\right). \tag{6}$$

The cost function of mini-batch of m samples in our n-BQ-NN is

$$C = \frac{1}{2m}\sum_x \|y(x) - a^L(x)\|^2 \tag{7}$$

where x is the input sample, y is the actual classification, a^L is the prediction output, and L is the maximum number of layers in the network.

By defining $\mathcal{T}_j^l \equiv (\partial C/\partial z_j^l)$ as the error produced by the jth neuron of the lth layer, we can use the back-propagation algorithm to calculate the gradient and update the parameters according to the following three steps.[1]

1) Calculating the error of the last layer of the network

$$\begin{aligned} \mathcal{T}_j^L &= \frac{\partial C}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L}\cdot\frac{\partial a_j^L}{\partial z_j^L} \\ \boldsymbol{\mathcal{T}}^L &= \frac{\partial C}{\partial \mathbf{a}^L} \odot \frac{\partial \mathbf{a}^L}{\partial \mathbf{z}^L} = \nabla_{\mathbf{a}} C \odot \theta'\left(\mathbf{z}^L\right). \end{aligned} \tag{8}$$

[1] $\odot$ represents the Hadamard product that is used for point-to-point product between matrices or vectors.

2) Calculating the error of each layer of the network from the back to the front

$$\begin{aligned} \mathcal{T}_j^l &= \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}}\cdot\frac{\partial z_k^{l+1}}{\partial a_j^l}\cdot\frac{\partial a_j^l}{\partial z_j^l} \\ &= \sum_k \mathcal{T}_k^{l+1}\cdot\frac{\partial\left(\hat{W}_{kj}^{l+1}a_j^l + b_k^{l+1}\right)}{\partial a_j^l}\cdot\theta'\left(z_j^l\right) \\ &= \sum_k \mathcal{T}_k^{l+1}\cdot\hat{W}_{kj}^{l+1}\cdot\theta'\left(z_j^l\right) \\ \boldsymbol{\mathcal{T}}^l &= \left(\left(\hat{\mathbf{W}}^{l+1}\right)^T\boldsymbol{\mathcal{T}}^{l+1}\right)\odot\theta'\left(\mathbf{z}^l\right). \end{aligned} \tag{9}$$

3) Calculating the gradient of weight and bias, respectively

$$\begin{aligned} g^b &= \frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l}\cdot\frac{\partial z_j^l}{\partial b_j^l} \\ &= \mathcal{T}_j^l\cdot\frac{\partial\left(\hat{W}_{jk}^l a_k^{l-1} + b_j^l\right)}{\partial b_j^l} = \mathcal{T}_j^l \end{aligned} \tag{10}$$

$$\begin{aligned} g^W &= \frac{\partial C}{\partial W_{jk}^l} = \frac{\partial C}{\partial z_j^l}\cdot\frac{\partial z_j^l}{\partial W_{jk}^l} = \mathcal{T}_j^l\cdot\frac{\partial\left(\hat{W}_{jk}^l a_k^{l-1} + b_j^l\right)}{\partial W_{jk}^l} \\ &= \mathcal{T}_j^l\cdot\frac{\partial\left(\hat{W}_{jk}^l a_k^{l-1} + b_j^l\right)}{\partial \hat{W}_{jk}^l}\cdot\frac{\partial \hat{W}_{jk}^l}{\partial W_{jk}^l} \\ &= \mathcal{T}_j^l\cdot a_k^{l-1}\cdot\frac{\partial \hat{W}_{jk}^l}{\partial W_{jk}^l}. \end{aligned} \tag{11}$$

In the above process of deriving the entire back-propagation, except for the gradient of weight of the last step, the other steps are well-defined. Based on (11), the gradient of weight can be calculated as follows:

$$g^W = \mathcal{T}_j^l\cdot a_k^{l-1}\cdot\frac{\partial \hat{W}_{jk}^l}{\partial W_{jk}^l} = 0 \tag{12}$$

where $(\partial \hat{W}_{jk}^l/\partial W_{jk}^l)$ is exactly the staircase$'(W_{jk}^l)$, which is the derivative of staircase(W_{jk}^l). This derivative satisfies the conditions of the Dirac delta function $\delta(x)$. According to the properties of $\delta(x)$, $(\partial \hat{W}_{jk}^l/\partial W_{jk}^l)$ can be calculated as follows:

$$\frac{\partial \hat{W}_{jk}^l}{\partial W_{jk}^l} = \delta\left(W_{jk}^l\right) = 0. \tag{13}$$

Substituting $(\partial \hat{W}_{jk}^l/\partial W_{jk}^l) = 0$ into (12), we discover that model cannot be trained by the back-propagation algorithm due to gradient vanishing.

To resolve the above problem, we reconstruct the quantized weight function as (14) to ensure that the weights can be updated by using the back-propagation algorithm as shown in Fig. 2, the blue full line, where α is an adjustable parameter in the range of (0, 1)

$$\tilde{W}_{jk}^l = (1-\alpha)\hat{W}_{jk}^l + \alpha W_{jk}^l. \tag{14}$$

By substituting (14) into (11), we can recalculate the gradient of weight as follows again with $(\partial \tilde{W}_{jk}^l/\partial W_{jk}^l) =$

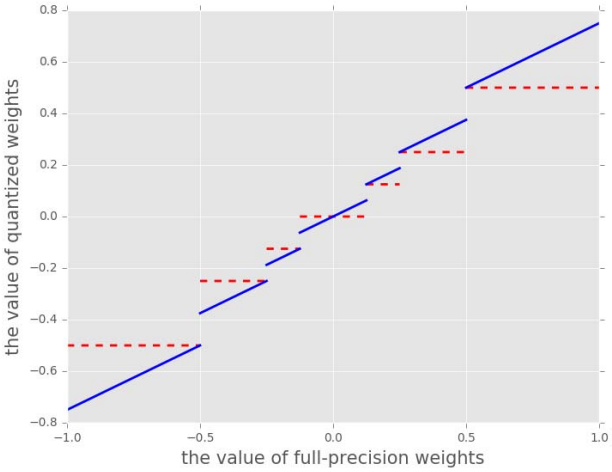

Fig. 2. Red dotted line represents the staircase function, and the blue full line represents our reconstructed function.

$\alpha + (1-\alpha)\delta(W_{jk}^l) = \alpha$:

$$g^W = \mathcal{T}_j^l \cdot a_k^{l-1} \cdot \frac{\partial \tilde{W}_{jk}^l}{\partial W_{jk}^l} = \alpha \mathcal{T}_j^l \cdot a_k^{l-1}. \tag{15}$$

At this point, we have reconstructed the quantized weight function as (14) to solve the gradient vanishing, but the weights cannot be quantized to the entries $(\pm 2^{-0}, \pm 2^{-1}, \ldots, 0)$ directly as (6). However, we can prove that the reconstructed quantized weight function will approximate to the entries after several iterations, which can be ensured by the Theorem 2.

Assumption 1: Since the algorithm needs to be iterated, our problem needs to be discussed within the framework of the series. We define W_{jk}^l as an iteration of x_n, $\tilde{W}_{jk}^l$ is equivalent to x_{n+1}, and the value of a_j is chosen from $\hat{W}_{jk}^l = \text{staircase}(W_{jk}^l) = (\pm 2^{-0}, \pm 2^{-1}, \ldots, \pm 2^{-i}, 0)$.

Theorem 2: In the framework of the series, $\tilde{W}_{jk}^l$ will approach to $\hat{W}_{jk}^l$ when the number of iterations is sufficient, where n is the number of iterations.

Proof: The general terms of series x from 1 to n are written as follows based on (14):

$$\begin{cases} x_2 - \alpha x_1 = (1-\alpha)a_j & (1) \\ \quad\vdots & \quad\vdots \\ x_n - \alpha x_{n-1} = (1-\alpha)a_j & (n-1) \\ x_{n+1} - \alpha x_n = (1-\alpha)a_j & (n). \end{cases} \tag{16}$$

We let $\alpha^{(n-1)} \times (1) + \cdots + \alpha \times (n-1) + (n)$, then, we get the equation as follows:

$$\begin{aligned} x_{n+1} - \alpha^{(n-1)} x_1 &= (1-\alpha)a_j(1 + \alpha + \cdots + \alpha^{(n-1)}) \\ &= a_j(1 - \alpha^{(n-1)}). \end{aligned} \tag{17}$$

As the number of iterations increases, x_{n+1} will approach a_j. With the guarantee of Theorem 2, the above equation can be rewritten as $\tilde{W}_{jk}^l = \text{staircase}(W_{jk}^l)$ (namely, $x_{n+1} = a_j$) when the number of iterations is enough ($n \to \infty$) and α is in the range of $(0, 1)$. In the actual algorithm implementation, it is only necessary to iterate through several steps following the training process, and the networks can be quantized completely as (6). □

The design of α in our reconstructed quantized weight function takes three aspects into consideration. First, the designed function must satisfy the Theorem 2. Second, our reconstructed function indicates that the ratio of $1-\alpha : \alpha$ between quantized weights and full-precision weights can be used to adjust the information ratio of quantized weights and full-precision weights in the training process. Third, on the other hand, α is the slope of our reconstructed function shown as the blue full line in Fig. 2, which can be used to change the gradient descent rate of back-propagation based on (15) during the training.

C. Posterior-Distribution Adjustment

In the initialization of the networks, the initialization modes MSRA and Xavier [32] that will adjust variance based on the number of inputs are prone to converge than the traditional Gaussian distribution initialization mode with fixed variance in DNNs. Inspiring by this fact, we suspect that adjusting the distribution of quantized weights may make it easier for us to train our n-BQ-NN. Here, we consider that full-precision networks are prone to converge than quantized networks; thus, we prefer to keep the distribution of quantized weights consistent with that of full-precision weights. Comparing the probability density function before quantization $\phi(x)$ (its corresponding expectation and variance are $E(x)$ and Var(x), respectively) and the probability density function after quantization staircase(x) [as (4)], we make their expectation and variance equal, respectively, so that their distribution is consistent as follows:

$$\begin{cases} E(x) = \displaystyle\int_{-1}^{1} x \,\text{staircase}(x)dx \\ \text{Var(x)} = \displaystyle\int_{-1}^{1} (x - E(x))^2 \,\text{staircase}(x)dx. \end{cases} \tag{18}$$

The original full-precision probability density function $\phi(x)$ and the value of quantized weight function staircase(x) are fixed, so we can only adjust the value range of staircase(x) to meet (18).

D. Training Algorithm for n-BQ-NN

In the actual training algorithm for n-BQ-NN, the batch normalization (BN [31]) is added in our n-BQ-NN because it is conducive to reduce the overall impact of the weight scale and accelerate the training. Thus, we will derive the back-propagation algorithm for n-BQ-NN with BN and give the training algorithm in this section.

First, we define four variables of BN, where σ represents the variance of all samples of a batch, μ represents the sample mean, and γ, β are the scale variation coefficients. Due to the existence of BN, the bias term can be ignored, so the input of the neuron is re-expressed as $z_j^l = \sum_k W_{jk}^l a_k^{l-1}$, the normalized input of the neuron is $\hat{z}_j = \gamma((z_j - \mu)/\sigma) + \beta$, and the output of the neuron is $a_j = \theta(\hat{z}_j)$. Then, we can calculate the error and the gradient, based on the discussion of Section III-B, according to the following three steps.

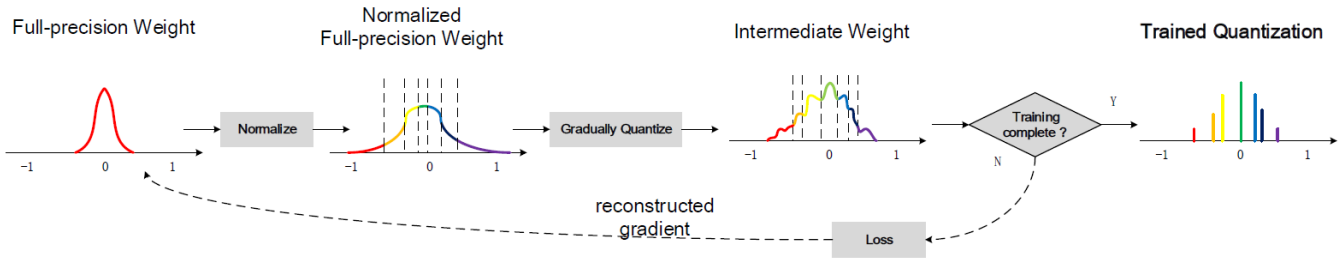

Fig. 3. Overview of our trained quantization procedure.

1) Counting the mean and variance of the sample, and calculating the gradient of them

$$\mu = \frac{1}{m}\sum_{j=1}^{m} z_j$$
$$\sigma^2 = \frac{1}{m}\sum_{j=1}^{m}(z_j-\mu)^2 \tag{19}$$
$$\frac{\partial C}{\partial \sigma^2} = \sum_k \frac{\partial C}{\partial a_k}\frac{\partial a_k}{\partial \hat{z}_k}\frac{\partial \hat{z}_k}{\partial \sigma^2} = -\frac{1}{2}\gamma\sigma^{-3}\sum_k \frac{\partial C}{\partial a_k}\theta'(z_k)(z_k-\mu) \tag{20}$$
$$\frac{\partial C}{\partial \mu} = \sum_k \frac{\partial C}{\partial a_k}\frac{\partial a_k}{\partial \hat{z}_k}\frac{\partial \hat{z}_k}{\partial \mu} + \frac{\partial C}{\partial \sigma^2}\frac{\partial \sigma^2}{\partial \mu} = -\frac{\gamma}{\sigma}\sum_k \frac{\partial C}{\partial a_k}\theta'(z_k) - \frac{2}{m}\frac{\partial C}{\partial \sigma^2}\sum_k (z_k-\mu). \tag{21}$$

2) Calculating the error of the network

$$\mathcal{T}_j = \frac{\partial C}{\partial z_j} = \frac{\partial C}{\partial a_j}\frac{\partial a_j}{\partial \hat{z}_j}\frac{\partial \hat{z}_j}{\partial z_j} + \frac{\partial C}{\partial \sigma^2}\frac{\partial \sigma^2}{\partial z_j} + \frac{\partial C}{\partial \mu}\frac{\partial \mu}{\partial z_j} = \frac{\gamma}{\sigma}\frac{\partial C}{\partial a_j}\theta'(\hat{z}_j) + \frac{2}{m}\sum_k (z_k-\mu)\frac{\partial C}{\partial \sigma^2} + \frac{1}{m}\frac{\partial C}{\partial \mu}. \tag{22}$$

3) Calculating the gradient of weight, γ, and β, respectively

$$g^W = \mathcal{T}_j \cdot a_k \cdot \frac{\partial \tilde{W}_{jk}}{\partial W^l_{jk}} = \alpha \mathcal{T}_j \cdot a_k \tag{23}$$
$$g^\gamma = \sum_k \frac{\partial C}{\partial a_k}\frac{\partial a_k}{\partial \hat{z}_k}\frac{\partial \hat{z}_k}{\partial \gamma} = \sum_k \frac{\partial C}{\partial a_k}\theta'(\hat{z}_k)\frac{z_k-\mu}{\sigma} \tag{24}$$
$$g^\beta = \sum_k \frac{\partial C}{\partial a_k}\frac{\partial a_k}{\partial \hat{z}_k}\frac{\partial \hat{z}_k}{\partial \beta} = \sum_k \frac{\partial C}{\partial a_k}\theta'(\hat{z}_k). \tag{25}$$

With the foundation of the above formulas, we can propose our training algorithm for n-BQ-NN, as indicated in Algorithm 1. This algorithm covers two learning modes: training from scratch and fine-tuning on the pretrained model, where the first mode means that the weights are randomly initialized and the second mode means that the weights are initialized by the pretrained full-precision network model. The overall quantization process is illustrated as Fig. 3. The code for training algorithm is available.[2]

E. Activation Quantization in n-BQ-NN

The above discussion is all about the quantization of weights. To take the integrity of our n-BQ-NN and the necessity of subsequent ablation experiments into consideration, we need to discuss the quantization of activations in this section. Now, let us put our eyes back on Section III-B. In the case of the quantized activations, the output of the jth neuron of the lth layer can be rewritten as follows:

$$\hat{a}^l_j = \text{staircase}\left(z^l_j\right) \tag{26}$$

where staircase() is the activation function.

At this point, we have encountered the same problem; the error of network $\mathcal{T}^l_j$ becomes zero due to the existence of $(\partial \hat{a}^l_j/\partial z^l_j)$, when (26) is substituted into (8) and (9).

Considering the expectation of $(\partial C/\partial z^L_j)$, the error of network has reappeared, which is guaranteed by Theorem 3.

Theorem 3: Let us define $C = C(\hat{a}_j, \epsilon_j)$, where $\hat{a}_j$ follows (26) that is chosen from $(\pm 2^{-0}, \pm 2^{-1}, \ldots, 0)$, then, we get a new expression as follows:

$$\mathbb{E}_{\epsilon_j}\left[\frac{\partial C}{\partial z_j}\right] = \lambda\frac{\partial C}{\partial \hat{a}_j}, \quad \text{if } |z_j| \le 1 \tag{27}$$

where ϵ_j is the noise source that influences z_j, $\mathbb{E}_{\epsilon_j}[\cdot]$ means the expectation over z_j, and λ is a constant.

Proof:

$$\begin{aligned}
&\mathbb{E}_{\epsilon_j}\left[\frac{\partial}{\partial z_j}C\right] \\
&= \frac{\partial}{\partial z_j}\mathbb{E}_{\epsilon_j}[C] \\
&= \frac{\partial}{\partial z_j}\left[\sum_i C(\hat{a}_j = +2^i)P(z_j > \epsilon_j|z_j) + \sum_i C(\hat{a}_j = -2^i)(1 - P(z_j > \epsilon_j|z_j))\right] \\
&= \frac{\partial P(z_j > \epsilon_j|z_j)}{\partial z_j}\left[\sum_i C(\hat{a}_j = +2^i) - \sum_i C(\hat{a}_j = -2^i)\right].
\end{aligned} \tag{28}$$

[2]https://github.com/papcjy/n-BQ-NN

Algorithm 1: Training algorithm for n-BQ-NN with BN. C is the cost function for mini-batch, θ is the activation function, and L is the number of layers. The function staircase$(\cdot)$ specifies how to quantize the weights. BatchNorm() specifies how to batch-normalize the inputs. BackBatchNorm() specifies how to back-propagate through the BN. Update() specifies how to update the parameters when their gradients are known, using either SGD or ADAM

Require: a minibatch of outputs and targets (a^L, y), learning rate η, previous weights W^k, previous BN parameters (γ^k, β^k), and a constant α.

Ensure : the updated weights $(W^k)^*$ and updated BN parameters $((\gamma^k)^*, (\beta^k)^*)$

{1. Computing the parameter gradients:}
{1.1 Forward propagation:}
for $k = 1$ *to* L **do**
  $\tilde{W}^k \leftarrow (1-\alpha)\,\text{staircase}(W^k) + \alpha W^k$
  $z^k \leftarrow a^{k-1}\tilde{W}^k$
  $\hat{z}^k \leftarrow \text{BatchNorm}(z^k, \gamma^k, \beta^k)$
  $a^k \leftarrow \theta(\hat{z}^k)$
end
{1.2 Backward propagation:}
Computing $g^{a^L} = \frac{\partial C}{\partial a^L}$ based on a^L and y.
for $k = L$ *to* 1 **do**
  $(g^{\gamma^k}, g^{\beta^k}) \leftarrow \text{BackBatchNorm}(g^{a^k}, z^k, \gamma^k, \beta^k)$
  $\mathcal{T}^k \leftarrow g^{a^k}\theta'(\hat{z}^k)$
  $g^{a^{k-1}} \leftarrow \mathcal{T}^k\tilde{W}^k$
  $g^{W^k} \leftarrow \alpha(\mathcal{T}^k)^T a^{k-1}$
end
{2. Updating the parameter gradients:}
for $k = 1$ *to* L **do**
  $((\gamma^k)^*, (\beta^k)^*) \leftarrow \text{Update}(\gamma^k, \beta^k, \eta, g^{\gamma^k}, g^{\beta^k})$
  $(W^k)^* \leftarrow \text{Update}(W^k, g^{W^k}, \eta)$
end

For $C(\hat{a}_j = \pm 2^i)$, we can approximate it using the Taylor expansion

$$\begin{aligned} C(\hat{a}_j = +2^i)) &= C(\hat{a}_j = 0) + \frac{\partial C}{\partial \hat{a}_j}\bigg|_{\hat{a}_j=0} 2^i \\ &\quad + \frac{\partial^2 C}{\partial \hat{a}_j^2}\bigg|_{\hat{a}_j=0} 2^{2i} + O\left(\frac{\partial^3 C}{\partial \hat{a}_j^3}\bigg|_{\hat{a}_j=0} 2^{3i}\right) \\ C(\hat{a}_j = -2^i)) &= C(\hat{a}_j = 0) - \frac{\partial C}{\partial \hat{a}_j}\bigg|_{\hat{a}_j=0} 2^i \\ &\quad + \frac{\partial^2 C}{\partial \hat{a}_j^2}\bigg|_{\hat{a}_j=0} 2^{2i} + O\left(\frac{\partial^3 C}{\partial \hat{a}_j^3}\bigg|_{\hat{a}_j=0} 2^{3i}\right). \end{aligned} \tag{29}$$

For $(\partial P(z_j > \epsilon_j|z_j)/\partial z_j)$, we split it into two parts

$$\begin{aligned} \frac{\partial P(z_j > \epsilon_j|z_j)}{\partial z_j} &= \underset{|z_j|>1}{\frac{\partial P(z_j > \epsilon_j|z_j)}{\partial z_j}} + \underset{|z_j|\le 1}{\frac{\partial P(z_j > \epsilon_j|z_j)}{\partial z_j}} \\ &= \frac{\partial \int_{-1}^{1} \frac{1}{2}\, d\epsilon_j}{\partial z_j} + \frac{\partial \int_{-z_j}^{z_j} \frac{1}{2}\, d\epsilon_j}{\partial z_j} = \mathbf{1}_{|z_j|\le 1}. \end{aligned} \tag{30}$$

Combining (29) and (30), (28) can be derived as follows:

$$\mathbb{E}_{\epsilon_j}\left[\frac{\partial C}{\partial z_j}\right] = \mathbf{1}_{|z_j|\le 1}\left(2\sum_i 2^{2i}\frac{\partial C}{\partial \hat{a}_j}\bigg|_{\hat{a}_j=0}\right). \tag{31}$$

Let $2\sum_i 2^{2i} = \lambda$, then

$$\mathbb{E}_{\epsilon_j}\left[\frac{\partial C}{\partial z_j}\right] = \lambda\frac{\partial C}{\partial \hat{a}_j}\mathbf{1}_{|z_j|\le 1}. \tag{32}$$

□

Under the Theorem 3, we can re-express the error of network and quantize the activations in our n-BQ-NN by rewriting (8) and (9) as follows:

$$\mathcal{T}_j^L = \frac{\partial C}{\partial z_j^L} = \lambda\frac{\partial C}{\partial \hat{a}_j^L}\mathbf{1}_{|z_j|\le 1} \tag{33}$$

$$\begin{aligned} \mathcal{T}_j^l &= \frac{\partial C}{\partial z_j^l} = \lambda\frac{\partial C}{\partial \hat{a}_j^l}\mathbf{1}_{|z_j|\le 1} = \lambda\sum_k \frac{\partial C}{\partial z_k^{l+1}}\cdot\frac{\partial z_k^{l+1}}{\partial \hat{a}_j^l} \\ &= \lambda\sum_k \mathcal{T}_k^{l+1}\cdot\frac{\partial\left(\hat{W}_{kj}^{l+1}\hat{a}_j^l + b_k^{l+1}\right)}{\partial \hat{a}_j^l} \\ &= \lambda\sum_k \mathcal{T}_k^{l+1}\cdot\hat{W}_{kj}^{l+1}\mathbf{1}_{|z_j|\le 1}. \end{aligned} \tag{34}$$

IV. Experiment

In our experiments, we use three network structures, such as ResNet, DenseNet, and AlexNet. The network structure of our n-BQ-NN (n can take 1–5) is similar to the architecture of All-CNN [33] that consists solely of convolution layers and Network in Network block [34]. Table I details the parameter settings and our network architecture. In the following experiments, our training algorithm is used to train the model from scratch or fine-tune on the full-precision model in five benchmark data sets, such as MNIST, SVHN, CIFAR-10, CIFAR-100, and ImageNet. We unfold our experiments from four dimensions, respectively, such as classification accuracy compared with low-precision QNNs, quantization errors by our training method, compression ratio in different data sets, and convergence speed compared with BNN.

A. MNIST

The MNIST data set [35] consists of handwritten digit images with 32 × 32 pixels, organized into ten classes (0–9). The training and test sets contain 60 000 and 10 000 images, respectively. We perform this data set without data augmentation [36].

B. CIFAR

The two CIFAR data sets [37] consist of natural color images with 32 × 32 pixels, respectively, 50 000 training and 10 000 test images, and we hold out 5000 training images as a validation set from the training set. CIFAR-10 (C10) consists of images organized into ten classes and CIFAR-100 (C100) into 100 classes. We adopt a standard data augmentation scheme (random corner cropping and random flipping) that is widely used for these two data sets [33], [34], [36], [38]–[41]. We normalize the images using the channel means and standard deviations in preprocessing.

TABLE I

OUTLINE OF THE PROPOSED n-BQ-NN NETWORK ARCHITECTURE. HERE, TAKING THE CIFAR DATA SETS AS AN EXAMPLE, THE INITIAL INPUT SIZE OF THE NETWORK IS $32 \times 32 \times 3$. THE CONV QUANTIZED CONTAINS THREE CALCULATION STEPS, RESPECTIVELY, $\hat{W} = \text{staircase}(W)$, $\text{net} = \text{conv2d}(\hat{W}, x)$, AND $\text{net} = \text{BatchNorm}(\text{net})$, WHERE THE WEIGHTS INVOLVED IN CONVOLUTION CALCULATION ARE QUANTIZED WEIGHTS THAT ARE CHOSEN FROM THE ENTRIES $(\pm 2^{-0}, \pm 2^{-1}, \ldots, 0)$. IN CONVOLUTION CALCULATION, THE MULTIPLICATIONS ARE REPLACED BY SHIFT OPERATIONS DURING THE INFERENCE, BECAUSE THE WEIGHTS ARE POWER OF 2

type	patch size/stride	output size
conv quantized	3×3/1	32×32×128
conv quantized	3×3/1	32×32×128
conv quantized	3×3/1	32×32×128
pool	2×2/2	16×16×128
conv quantized	3×3/1	16×16×256
conv quantized	3×3/1	16×16×256
conv quantized	3×3/1	16×16×256
pool	2×2/2	8×8×256
conv quantized	3×3/1	8×8×512
conv quantized	1×1/1	8×8×1024
conv quantized	1×1/1	8×8×10 (100)
pool	8×8	1×1×10 (100)
softmax	classifier	1×1×10 (100)

C. SVHN

The SVHN data set [42] consists of color images of house numbers collected by Google Street View with 32×32 pixels, organized into ten classes (0–9). There are 73 257 images in the training set, 531 131 images for additional training, and 26 032 images in the test set, respectively. We divide the pixel values by 255.0 so that they are in the [0,1] range as [43]. Moreover, we do not preprocess the data set following common practice without data augmentation [34], [36], [39], [44], [45].

D. Experimental Results

1) n-Bit: As the theoretical analysis in Section III, different quantized bit width brings different sampling loss, and the larger bit width means the less sampling loss. Thus, in this experiment, we evaluate the test error rates of our n-BQ-ResNet that is fine-tuned on full-precision ResNet-110 when n takes different values on CIFAR-10. The experimental results from Table II are consistent with (1). Therefore, the choice of 3 bit is better because $\mathcal{L}(4)$ is close to 0 as (3) when considering both the sampling loss and the conciseness of weight representation. Obviously, this result is also experimentally proved by works in [26]. Thus, our n-BQ-NN is chosen as T-BQ-NN when $n = 3$ in the subsequent experiments.

2) Accuracy and Capacity: As hardware devices require relatively simple architecture and less number of layers, we have selected some networks suited for hardware implementation as our comparative experiment. For example, BNN with binary weights and activations replaces most multiplications by 1-bit XNOR operations. Network in Network utilizes the global average pooling over feature maps in the classification layer, which is less prone to overfitting than the fully connected layers. All-CNN achieves a new architecture that consists solely of convolution layers replacing max-pooling by a convolution layer without loss in accuracy on several benchmarks. Highway Network allows unimpeded information flow across many layers using adaptive gating units to regulate the information flow.

TABLE II

OUR n-BQ-RESNET GENERATES EXTREMELY LOW-PRECISION MODELS WITH VERY SIMILAR ACCURACY COMPARED WITH FULL-PRECISION RESNET-110 MODEL ON CIFAR-10

Model	Bit-width	Test error
ResNet-110 ref	16	6.61%
n-BQ-ResNet	5	7.04%
n-BQ-ResNet	4	7.07%
n-BQ-ResNet	3	7.15%
n-BQ-ResNet	2	8.76%
n-BQ-ResNet	1	10.52%

There is a general manifestation that T-BQ-NN performs better than most other network structures, while these network structures have never been quantized except BNN. In the experiment here, T-BQ-NN is trained by our training algorithm from scratch due to the lack of pretrained model, and this model is trained with a mini-batch size of 50 and a weight decay of 0.0001. Its test error rates of 7.59% on CIFAR-10, 28.9% on CIFAR-100, 2.29% on SVHN, and 0.5% on MNIST are lower than the test error rates achieved by Network in Network, Highway Network, and BNN. Particularly, T-BQ-NN makes up for the classification accuracy of BNN on CIFAR-100 to some extent. The best result for all listed data sets is T-BQ-NN except CIFAR-10 is All-CNN, and all results are shown in Table III.

Our model capacity is even more encouraging: the number of parameters of T-BQ-NN is significantly lower than those of other network structures. Particularly, T-BQ-NN achieves the number of parameters of 1.2M that is even lower than 1.7M of BNN shown in Table III.

3) Extension: One positive effect of our training algorithm is universal. We popularize our training method to the better and deeper architectures, not just limited to CNNs, such as ResNet [16] and DenseNet [17]. In the experiment here, T-BQ-ResNet and T-BQ-DenseNet are 3 bit that are fine-tuned by our training algorithm based on the full-precision model of ResNet and DenseNet.

For T-BQ-ResNet, all the multiplications are converted to SHIFT and ADDER operations using 3-bit weights in all convolutional layers and shortcut connections. We use a momentum of 0.9 and a weight decay of 0.0001 [44], [49], and adopt the weight initialization and BN [31], [32] without dropout [50]. This model is trained with a mini-batch size of 128 and a learning rate of 0.1, divided by 10 at 32k and 38k

TABLE III

ERROR RATES ON CIFAR-10 AND CIFAR-100 DATA SETS WITH STANDARD DATA AUGMENTATION (TRANSLATION AND MIRRORING). ERROR RATES ON MNIST AND SVHN DATA SETS WITHOUT DATA AUGMENTATION. THE OVERALL BEST RESULTS ARE **BOLD**. "*" REPRESENTS THE RESULTS RUN BY OUR IMPLEMENTATION, THE REST OF THE RESULTS REPRESENTS THAT THEY ARE DIRECTLY CITED FROM THIS ARTICLE IN THE FRONT OF THE ROW

			Test error			
Method	**Depth**	**Params**	CIFAR-10	CIFAR-100	SVHN	MNIST
Network in Network [34]	9	1.9M	8.81%	35.68%	2.35%	0.53%
All-CNN [33]	9	1.4M	**7.25%**	33.71%	*3.17%	*0.63%
Highway Network [38]	19	2.3M	7.72%	32.39%	*2.61%	0.67%
BNN [15]	9	1.7M	11.40%	*42.13%	2.80%	0.96%
Round Quantization	9	1.2M	85.88%	98.90%	83.72%	80.55%
T-BQ-NN	9	1.2M	7.59%	**28.90%**	**2.29%**	**0.50%**

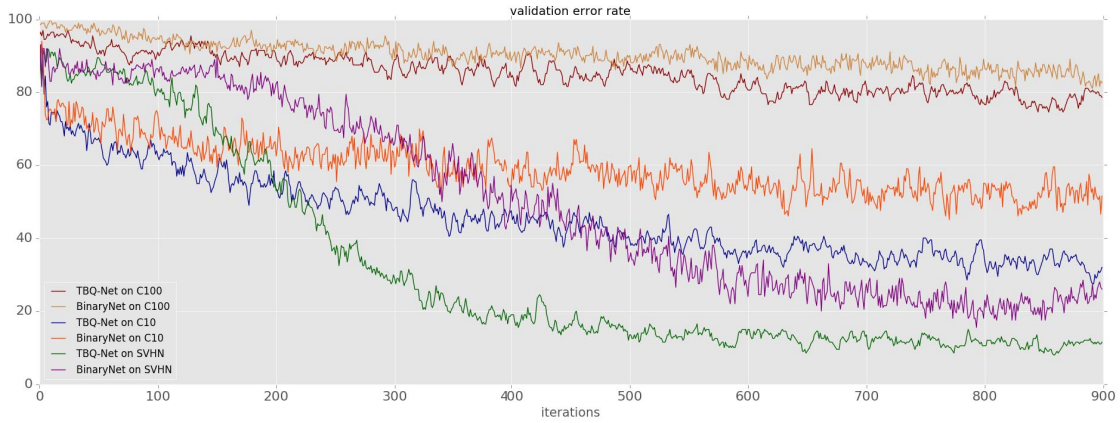

Fig. 4. Comparison of the T-BQ-NN and BNN top-1 error rates on CIFAR10, CIFAR100, and SVHN validation data sets, where the curves, from top to bottom at 900 iterations, represent T-BQ-NN on C100, BNN on C100, T-BQ-NN on C10, BNN on C10, T-BQ-NN on SVHN, and BNN on SVHN, respectively. Note that the results of data sets run by ourselves.

TABLE IV

FINE-TUNING RESNET AND DENSENET BY OUR TRAINING ALGORITHM ON CIFAR10(100) AND SVHN, WHERE THE RESULTS ON C10 AND C100 WITH DATA AUGMENTATION AND THE RESULTS ON SVHN WITHOUT DATA AUGMENTATION

	Network	**Depth**	**Bit-width**	**Params**	**Test error(%)**
CIFAR-10	ResNet	110	16	1.7M	6.61
	T-BQ-ResNet	110	3	0.3M	7.15 (+0.54)
	DenseNet	100	16	0.8M	4.51
	T-BQ-DenseNet	100	3	0.15M	5.31 (+0.80)
CIFAR-100	ResNet	110	16	1.7M	35.87
	T-BQ-ResNet	110	3	0.3M	37.56 (+1.69)
	DenseNet	100	16	0.8M	22.27
	T-BQ-DenseNet	100	3	0.15M	24.10 (+1.83)
SVHN	ResNet	110	16	1.7M	3.13
	T-BQ-ResNet	110	3	0.3M	3.25 (+0.12)
	DenseNet	100	16	0.8M	1.76
	T-BQ-DenseNet	100	3	0.15M	2.10 (+0.34)

iterations, and terminates training at 64k iterations. We achieve the test error rates of 7.15% on C10, 37.56% on C100, and 3.25% on SVHN using T-BQ-ResNet, just rises 0.54% on C10, 1.69% on C100, and 0.12% on SVHN compared with ResNet on the basis of Table IV.

TABLE V

DEEP COMPRESSION METHOD FOR T-BQ-RESNET AND T-BQ-DENSENET. P: PRUNING, Q: QUANTIZATION, AND H: HUFFMAN CODING

Method	**Encoding bit-width**	**Compression ratio**
T-BQ-ResNet on C10 (P+Q)	3	49×
T-BQ-ResNet on C10 (P+Q+H)	2.6	57×
T-BQ-ResNet on C100 (P+Q)	3	25×
T-BQ-ResNet on C100 (P+Q+H)	2.8	27×
T-BQ-ResNet on SVHN (P+Q)	3	24×
T-BQ-ResNet on SVHN (P+Q+H)	2.8	26×
T-BQ-DenseNet on C10 (P+Q)	3	38×
T-BQ-DenseNet on C10 (P+Q+H)	2.5	46×
T-BQ-DenseNet on C100 (P+Q)	3	15×
T-BQ-DenseNet on C100 (P+Q+H)	2.8	16×
T-BQ-DenseNet on SVHN (P+Q)	3	133×
T-BQ-DenseNet on SVHN (P+Q+H)	2.1	190×

For T-BQ-DenseNet, its model consists of Bottleneck layers indicated to BN-ReLU-Conv(1 × 1)-BN-ReLU-Conv(3 × 3) and transition layers indicated to BN-ReLU-Conv(1 × 1)-

TABLE VI

COMPARISON OF CLASSIFICATION ACCURACY ON THE TEST SET FOR IMAGENET WITH DIFFERENT BIT WIDTHS OF WEIGHTS AND ACTIVATIONS. SINGLE-CROP EVALUATION RESULTS TOP-1 AND TOP-5 ACCURACY ARE GIVEN BASED ON ALEXNET. NOTE THE GRAY RESULTS ARE IMPLEMENTED BY OUR n-BQ-NN, WHERE THE TRAINING METHOD OF 1-bit ACTIVATIONS IS INTRODUCED IN SECTION III-E, AND OTHER RESULTS ARE REPORTED BY [46]. WE QUANTIZE THE SAME LAYERS OF ALEXNET TO LOW PRECISION, AS BNN [15], BC [47], TWN [48], AND DOREFA-NET [27] DO

n-bit Weights	n-bit Activations	Inference Operation	AlexNet Top-1 Accuracy	AlexNet Top-5 Accuracy
1	1	XNOR	0.279 (BNN)	0.504 (BNN)
1	1	XNOR	0.348	0.601
1	32 (float)	XNOR ADDER	0.368 (BC)	0.620 (BC)
1	16 (float)	XNOR ADDER	0.486	0.734
2	32 (float)	XNOR ADDER	0.529 (TWN)	0.766 (TWN)
2	16 (float)	XNOR ADDER	0.536	0.777
3	16 (float)	SHIFT ADDER	**0.560**	**0.795**
8 (float)	8 (float)	MAC	0.530 (DoReFa-Net)	0.768 (DoReFa-Net)
32 (float)	32 (float)	MAC	0.566	0.802

averpool(2×2), and both of these layers contain 1×1 convolution. We use a weight decay of 0.0001 and a momentum of 0.9 [51], and adopt the weight initialization and BN without dropout. This model is trained with an initial learning rate of 0.1, divided by 10 at 50% and 75% of the total number of training epochs. We train using a batch size of 64 for 300 and 40 epochs, respectively, on CIFAR and SVHN. Compared between DenseNet and T-BQ-DenseNet, the increasing in error is 0.80% from 4.51% to 5.31% on C10, 1.83% from 22.27% to 24.10% on C100, and 0.34% from 1.76% to 2.10% on SVHN, as shown in Table IV.

We attribute this primarily to reduce the number of parameters approximately five times from 0.8M to 0.15M on T-BQ-DenseNet and from 1.7M to 0.3M on T-BQ-ResNet, as shown in Table IV. Furthermore, a hybrid network compression solution combined with three different techniques, respectively, such as network pruning [19], quantization, and Huffman coding, is tested for T-BQ-ResNet and T-BQ-DenseNet in a scene with the same classification accuracy. Compared with the original full-precision ResNet-110 model, we achieve the compression ratio of 57× on C10, 27× on C100, and 26× on SVHN for T-BQ-ResNet. For T-BQ-DenseNet, the compression ratio is 46× on C10, 16× on C100, and 190× on SVHN, as shown in Table V.

4) Convergence Speed: In this experiment, we train our T-BQ-NN and BNN from scratch on C10, C100, and SVHN. The results in Fig. 4 indicate that T-BQ-NN not only has a better performance on classification accuracy than BNN but also converges much faster. We just only compare our method with BNN because the weights of other network models in Table III are full precision and these models are not quantized except BNN and T-BQ-NN. We use the same conditions, including learning rate, batch size, and iterations, to test the error rates of BNN and T-BQ-NN at first epoch. Compared with BNN, T-BQ-NN reaches the best test error dropping from 500 to 150 epochs on C10, from 1000 to 100 epochs on MNIST, and from 1000 to 180 epochs on C100. As a result, T-BQ-NN can be trained much easier and faster than BNN.

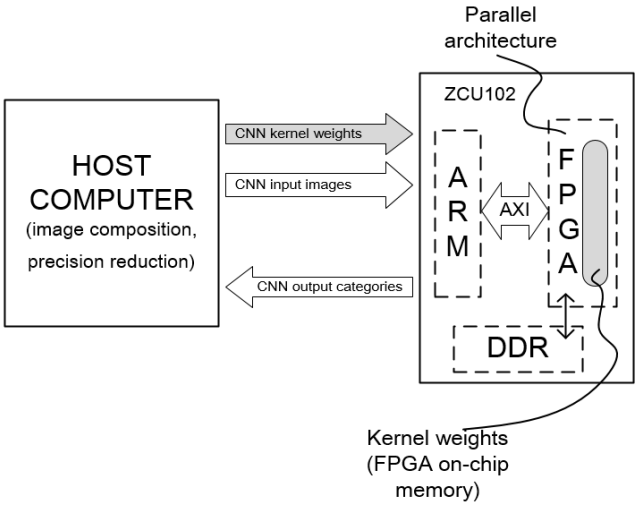

Fig. 5. High-level block diagram of our system.

This result may be due to the fact that straight-through estimator used by BNN contains noise, which causes the unexpected deviation, while our training algorithm is based entirely on back-propagation without the effect of noise and weight representation is more abundant.

E. ImageNet

We further evaluate our n-BQ-NN on ILSVRC2012 [11] image classification data set that consists of 1.2 million high-resolution natural images, where the validation set contains 50k images. These images are organized into 1000 categories of the object for training, which are resized to

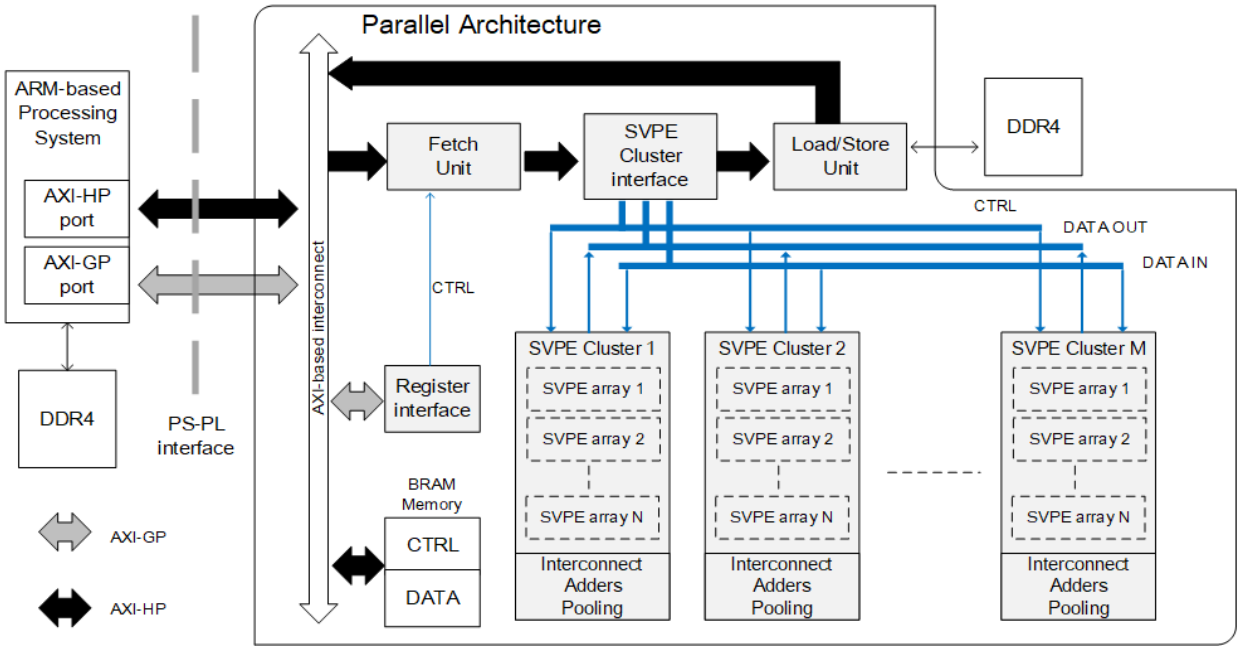

Fig. 6. Overall parallel architecture with cluster of SVPEs.

224 × 224 pixels before fed into the network. In the next experiments, we report our single-crop evaluation results using top-1 and top-5 accuracies.

AlexNet: This CNN architecture is the first structure that shows to be successful on ImageNet classification task, which consists of five convolutional layers and two fully connected layers [1]. We use AlexNet coupled with BN that contains a total of 61M parameters.

In training, images are resized randomly to 256 × 256 pixels, and then, a random crop of 224 × 224 is selected for training. We train our *n*-BQ-NN for 50 epochs with a batch size of 128/16 based on AlexNet/Vgg-16. We use ADAM optimizer with a learning rate of 1e-4. We replace the local contrast renomalization layer with batch normalization layer. At inference, we use the 224 × 224 center crop for forward propagation.

The ablation experiments are listed in Table VI. The baseline AlexNet model scores 56.6% top-1 accuracy and 80.2% top-5 accuracy that is reported in [52]. For ablation studies, we strictly control the consistency of variables, including network structure, bit width, and quantized layers. The only difference is the quantization method. In experiments of "1-1" versus "1-1," "1-16" versus "1-32," and "2-16" versus "2-32," our *n*-BQ-NN achieves 6.9%, 11.8%, and 0.7% accuracy improvements, respectively. For "3-16" versus "32-32," our *n*-BQ-NN only reduces the accuracy by 0.6%.

V. Acceleration on FPGA

We evaluate our *n*-BQ-NN on the FPGA platform: Xilinx ZCU102, which consists of an ultrascale FPGA, quad ARM Cortex-A53 processors, and 500-MB DDR3. To measure the performance of our *n*-BQ-NN running on FPGA, we get the data of run-time, resource utilization, and power by simulating and testing on Vivado-2017 when the operating frequency is 200 MHz. Our *n*-BQ-NN implementation involves a few design parameters, parallelization degree (P_n and P_m), filter size (k), input feature maps width (W), input feature maps height (H), input feature channels (M), and output feature channels (N).

A. Coprocessor Architecture

Fig. 5 shows the block diagram of our system. In the CNN calculation process, the host computer hands off the weights and images to the coprocessor (ZCU102) and collects the predicted classification results. The transmission mode between host computer and ARM CPU can be switched in PCI or UDP. Before the computation, the host computer is responsible for feeding the images and reducing precision. In addition, the ARM CPU needs to complete the calculation of fully connected layers that is not suitable for FPGA parallel acceleration, and the FPGA accelerates the calculation of convolutional layers.

We build the coprocessor with parallel architecture, as shown in Fig. 6. The critical part of the coprocessor is SVPE cluster interface that has M SVPE clusters, where each SVPE cluster consists of N SVPE arrays with a size of $k \times k$. The adders are used to compute partial sums of convolutions while the SVPE arrays compute convolutions. The fetch unit is programmed to fetch images and weights from ARM-based processing system (PS), and the load/store unit is used to load or store intermediate calculation results. The AXI-HP port is used to receive or send the data, and the AXI-GP port is used to receive or send the network structure information and the control signal. A key point to note is 16-bit computational accuracy acts on the data buses to save data bandwidth.

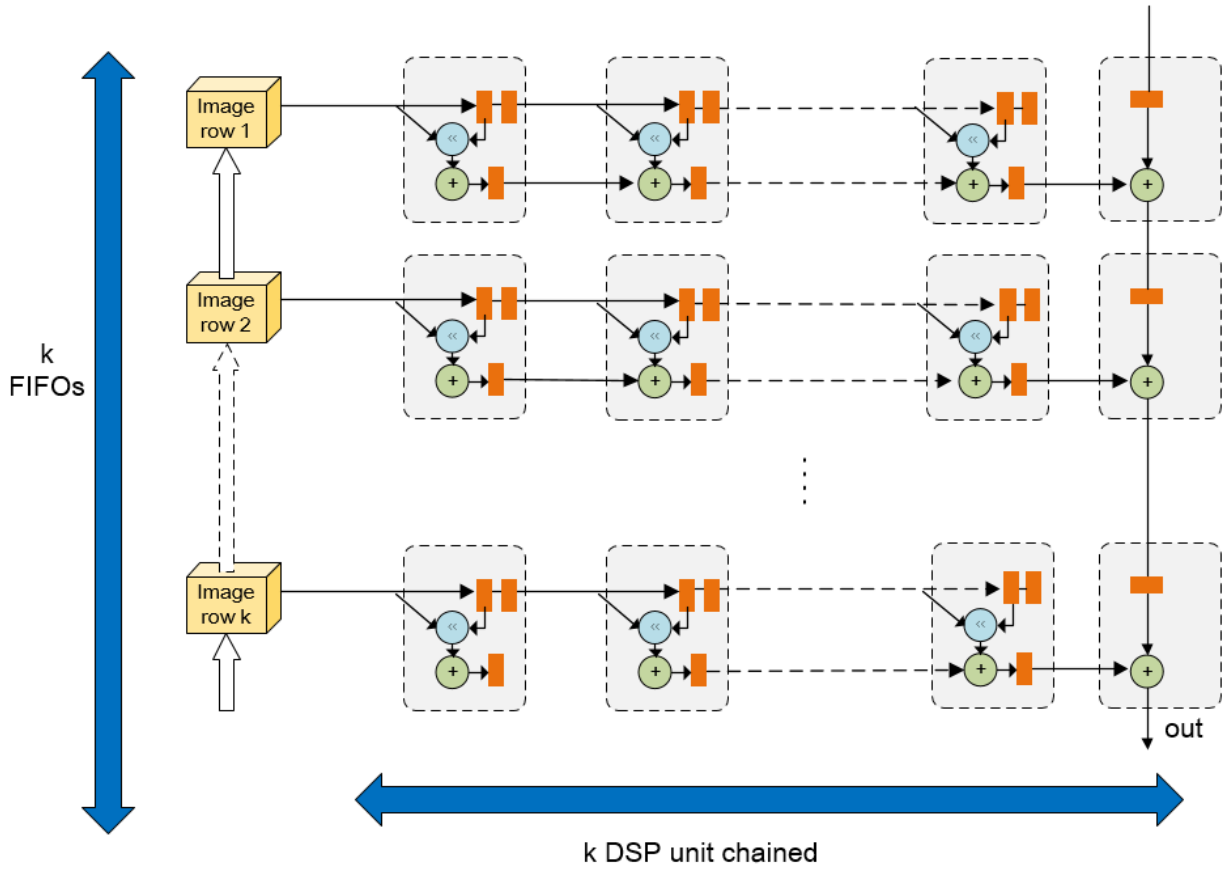

Fig. 7. SVPE array implementing the primitive 2-D convolver unit, where the double orange rectangles represent on-chip memory banks to buffer the weights, the single orange rectangles represent registers, the sky blue roundnesses represent shift operations instead of multipliers in VPE array, and the deep-green roundnesses represent adders.

The basic design ideas continue the architecture of n-BQ-NN, which converts all 16-bit weights as n-bit weights to reduce memory usage and increase the parallelization degree. On FPGAs, due to the shortage of DSPs, this has become a major factor limiting the increase in parallelization degree that directly affects the ability to accelerate calculation for CNNs because the multiplication in the convolution calculation needs to call DSPs. Instead, we implement the multiplication with SHIFT operation that consumes the lookup table (LUT) arrays on FPGAs, while the resource of LUTs is more abundant than that of DSPs [30]. In general, our n-BQ-NN consists of 16-bit activations and n-bit weights.

Our architecture of convolution computation is characterized by several key attributes compared with VPEs [14]. First, we organize the architecture as arrays of SVPEs, where the SVPE array is an array of 2-D convolvers, each of which consists of k^2 connected SHIFT and ADDER units instead of multiply accumulate (MAC) units, as shown in Fig. 7. The weights and feature maps are loaded into each PE alternately by AXI-HP port. Before each calculation, the weights are buffered to the specified areas (the double orange rectangles in Fig. 7), and then, the pipeline calculation starts with the enablement of feature maps. Modeling the SVPE and VPE arrays, we compare their resource consumption, parallelization degree, and power on FPGAs as shown in Table VII, and our SVPE array achieves the average energy consumption of 3.81 W at different n that is less than VPE array of 5.53 W. Second, we reduce banded off-chip data memory and improve the data movement between the SVPE clusters and the off-chip memory by using n-bit weights. Third, all convolvers are homogeneous when k is fixed as our primitive operator. We evaluate the improvement of the computational efficiency of n-BQ-NN in hardware in Section V-B.

B. Computational Efficiency

Since the filter size (3×3) is fixed for our n-BQ-NN, resource utilization will be maximized. Here, we can predict the performance of n-BQ-NN on FPGAs by developing an analytical model. In the following, we rely on it to compare computational efficiency between traditional implementation and n-BQ-NN on FPGAs.

On the hardware, MAC unit, adder, and multiplier will consume DSP. In fact, the number of DSPs only depends on the size of filter and parallelization degree [12], [30] as follows:

$$\text{DSP} = (k^2 + k) \times P_n \times P_m. \tag{35}$$

We must balance the memory bandwidth between the on-chip and off-chip memory and ensure that the speed of

TABLE VII
COMPARISON OF VPE AND SVPE ARRAY RESOURCE CONSUMPTION, PARALLELIZATION DEGREE, AND POWER

Array		**SVPE**					**VPE**
Precision (n)		1	2	3	4	5	16
Power (W)	Signal	1.94	2.31	2.61	2.53	2.13	4.88
	Logic	1.03	1.33	1.55	1.63	1.62	0.25
	DSPs	0.08	0.09	0.09	0.08	0.06	0.40
	Total	3.05	3.73	4.25	4.24	3.81	5.53
Used Resource	LUTs	353	280	307	334	346	41
	FFs	220	226	232	238	244	213
	DSPs	3	3	3	3	3	12
Parallelization degree (P_n, P_m)		(8,32)					(4,16)

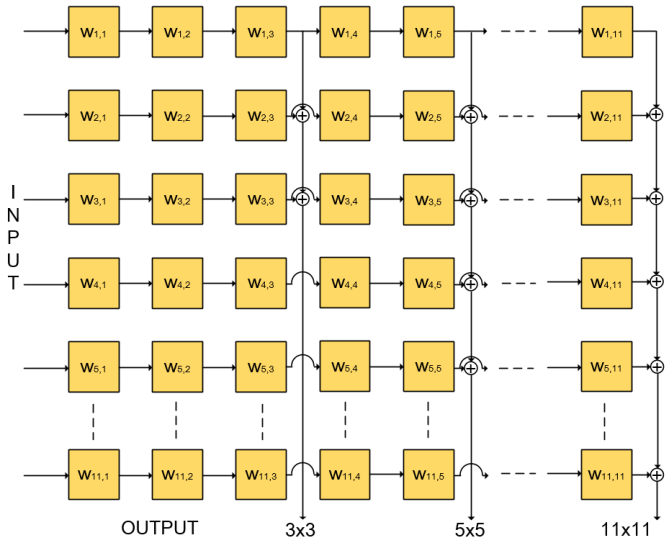

Fig. 8. Universal SVPE array supporting AlexNet with multiple kernel sizes (3, 5, and 11), where the yellow blocks are shift-accumulator units.

transmission is greater than or equal to the speed of computation for utilizing the resource efficiently. The formula of the time to process input data in the line buffer on FPGA is

$$T_{\text{compute}} = \left(H \times W \times \frac{M}{P_m} \times \frac{N}{P_n}\right) \times \frac{1}{\text{Freq}} \tag{36}$$

where Freq is the operating frequency of the FPGA. Together, we have to parallel the speed of transmission between input and output data as follows:

$$T_{\text{transfer}} = \frac{M \times N \times k^2 + k \times W \times M}{\text{Bandwidth}}. \tag{37}$$

We require that $T_{\text{transfer}} \leq T_{\text{compute}}$. Therefore, we can get that the minimum requirement of bandwidth is

$$\text{Bandwidth}_{min} = \frac{P_m \times P_n}{\min(N, M)} \times b_{\text{compute}} \times \text{Freq} \tag{38}$$

where b_{compute} is the bit width of computation, and we evaluate the performance of hardware acceleration choosing 16-bit width. We define the T_{init} as the time to load the first n rows of input image and filter needed into on-chip memory as follows:

$$T_{\text{init}} = \frac{M \times N \times k^2 \times b_{\text{weight}}}{\text{Bandwidth}} + \frac{W \times M \times k}{\text{Bandwidth}} \times b_{\text{compute}} \tag{39}$$

where b_{weight} is the bit width of the weights. The total operations are

$$\text{OPs} = H \times W \times M \times N \times k^2 \times 2. \tag{40}$$

The total processing time of the convolution is

$$T_{\text{total}} = T_{\text{compute}} + T_{\text{init}}. \tag{41}$$

Finally, we can compare the computational efficiency of different models defining the effective performance of convolution as follows:

$$\text{Perf}_{\text{eff}} = \frac{\text{OPs}}{T_{\text{total}}}. \tag{42}$$

We obtain the computational efficiency $\text{Perf}_{\text{eff}}(n)$ corresponding to different bit widths of the weights, where $n = b_{\text{weight}}$ represents the bit width of our n-BQ-NN

$$\text{Perf}_{\text{eff}}(n) = \frac{32\text{Freq}P_m P_n HWNk^2}{16HWN + nMNk^2 + 16WMk}. \tag{43}$$

Now, given a convolutional layer represented as (W, H, M, N), we get the computational efficiency based on design parameters (k, P_m, P_n).

The main reason for restricting the computational efficiency of CNNs on FPGA is a parallelization degree, which is directly related to DSPs when the setting of bandwidth is reasonable. To speed-up the inference of CNNs on FPGAs, we use our SVPE cluster to replace the traditional VPE cluster by converting the multiplications as the SHIFT and ADDER operations. Since we no longer use the multiplication, the amount of DSPs is reduced as follows:

$$\text{DSP} = k \times P_n \times P_m. \tag{44}$$

Since SVPE array consumes much less DSPs than VPE array compared (35) with (44), n-BQ-NN with SVPE array can get a larger amount of parallelization degree than CNNs with VPE array when the consumed DSPs are the same. Based on the maximum DSP number of 2520 as shown in Table VIII and the balanced memory bandwidth, we can design the maximum parallelization degree of ($P_m = 32$, $P_n = 8$) and ($P_m = 16$, $P_n = 4$), respectively, on SVPE and VPE arrays with filters 3×3. Thanks to the SVPE array, the parallelization degree increases by four times to improve the computational efficiency greatly when the total consumed DSPs is 768. Supposing the color image is $32(W) \times 32(H) \times 3(M)$ pixels, filter size k is 3, and DDR bit width N is 128, the computational efficiency of our n-BQ-NN using SVPE array has improved by about 4.1 times compared with the traditional network using VPE array on the basis of (43).

C. Performance on FPGA

We evaluate our SVPE cluster implementation using AlexNet, where our n-BQ-NN contains 16-bit activations and 3-bit weights. Table VIII gives the evaluation results with the comparison of the state-of-the-art FPGA accelerators, where GOP indicates the unit of the number of operations. From the hardware perspective, we prefer to use computational efficiency to describe the performance of the algorithm.

TABLE VIII
PERFORMANCE COMPARISON FOR ALEXNET

	[53]	**Baseline**	[30]	**Our Impl.**
Precision	32bits fixed	16bits fixed	16bits fixed	16bits fixed
Device	VX485T	ZCU102	ZCU102	ZCU102
Freq(MHz)	100	200	200	200
Logic cell(K)	485.7	600	600	600
DSP	2800	2520	2520	2520
BRAM(Kb)	2060×18	1824×18	1824×18	1824×18
conv1(GOP/s)	27.5	227.5	409.6	410.5
conv2(GOP/s)	83.8	535.8	1355.6	1744.3
conv3(GOP/s)	78.8	655.9	1535.7	1680.7
conv4(GOP/s)	77.9	634.4	1361.7	1739.4
conv5(GOP/s)	77.6	559.5	1285.7	1456.1
CNN average (GOP/s)	61.6	332.2	854.6	957.4
Power(W)	18.6	28.7	23.6	19.6
DSP Efficiency (GOP/s/DSPs)	0.022	0.131	0.339	0.381
Logic cell Efficiency (GOP/s/cells/K)	0.127	0.553	1.424	1.596
Energy Efficiency (GOP/s/W)	3.31	11.57	36.2	48.85
DSP Utilization	80%	30%	63%	30%
LUT Utilization	61%	48%	39%	73%
FF Utilization	34%	42%	33%	68%
BRAM Utilization	50%	50%	43%	83%

Because the total operations of computing a network are fixed, we can get the execution time(s) by dividing the total operations (GOP) by the computational efficiency (GOP/s). Similar to the structure of Fig. 7, a universal SVPE array designed by the largest filter size of AlexNet is proposed in Fig. 8. This experiment will use the universal SVPE array that fits the full size after a slight adjustment. Our array designs to be recycled when calculating the convolutions of different layers, and 11×11 filter of AlexNet is only used in the first convolutional layer, so most of the array utilization is extremely low. This also confirms the necessity of designing the network architecture with 3×3 unified filter to improve resource utilization as shown in Table I.

Compared to prior works [30], [53], we improve the average CNN performance to 957.4 GOP/s, where the work [30] is implemented by the Winograd algorithm. The baseline is to implement the same hardware architecture as our implementation. The only difference is that it uses VPE cluster because its weights and activations are both 16 bits. The computational efficiency of our implementation has improved by 2.9 times compared with the baseline, which is slightly less than 4.1 times based on the theoretical calculations of Section V-B. On the other hand, our implementation also improves the energy efficiency to 48.9 GOP/s/W. The better energy efficiency and resource efficiency come from the novel SVPE structure.

VI. CONCLUSION AND FUTURE WORK

In this article, we present a novel learning framework to quantize full-precision CNN models into low-precision QNN models, whose weights are constrained to the power of two. We solve the problem of gradient vanishing by adding a reconstructed gradient function into the back-propagation algorithm. To satisfy the network-structure-optimization requirements for hardware limitation, we propose n-BQ-NN, a novel QNN structure, to replace the multiplication with the SHIFT operation, whose structure is more suitable for the inference on FPGAs. Furthermore, we also design the SVPE array to replace all 16-bit multiplications with SHIFT operations in convolution operation on FPGAs. For proving the validity of our learning framework, we conduct experiments and show that the quantized models of ResNet, DenseNet, and AlexNet through our learning framework can achieve almost the same accuracies with the original full-precision models. Moreover, when using our learning framework to train our n-BQ-NN from scratch, it can achieve nearly state-of-the-art results compared with typically low-precision QNNs. We also evaluate the computational efficiency and energy consumption by implementing our QNNs models on Xilinx ZCU102 platform. In our hardware experiments, our n-BQ-NN with our SVPE can execute 2.9 times faster than with the VPE in inference, and the use of SVPE array also reduces average energy consumption to 68.7% of the VPE array with 16 bits. Our future work should explore how to decrease the accumulated quantization errors further when our learning framework is used on different CNN structures.

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.
[2] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2014, pp. 1–9.
[3] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, pp. 1–14, Sep. 2015.
[4] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, "DeepFace: Closing the gap to human-level performance in face verification," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2014, pp. 1701–1708.
[5] Y. Sun, X. Wang, and X. Tang, "Deep learning face representation from predicting 10,000 classes," in *Proc. IEEE Conf. Comput. Vis. pattern Recognit.*, Jun. 2014, pp. 1891–1898.
[6] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2015, pp. 3431–3440.
[7] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, "Semantic image segmentation with deep convolutional nets and fully connected CRFs," 2014, *arXiv:1412.7062*. [Online]. Available: http://arxiv.org/abs/1412.7062
[8] R. Girshick, "Fast R-CNN," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Dec. 2015, pp. 1440–1448.
[9] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2015, pp. 91–99.
[10] O. Russakovsky *et al.*, "ImageNet large scale visual recognition challenge," *Int. J. Comput. Vis.*, vol. 115, no. 3, pp. 211–252, Dec. 2015.

[11] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2009, pp. 248–255.

[12] M. Peemen *et al.*, "Memory-centric accelerator design for convolutional neural networks," in *Proc. ICCD*, 2013, pp. 13–19.

[13] P. Meloni, G. Deriu, F. Conti, I. Loi, L. Raffo, and L. Benini, "A high-efficiency runtime reconfigurable IP for CNN acceleration on a mid-range all-programmable SoC," in *Proc. Int. Conf. ReConFigurable Comput. FPGAs (ReConFig)*, Dec. 2016, pp. 1–8.

[14] M. Sankaradas *et al.*, "A massively parallel coprocessor for convolutional neural networks," in *Proc. 20th IEEE Int. Conf. Appl.-Specific Syst., Archit. Processors*, Jul. 2009, pp. 53–60.

[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," 2016, *arXiv:1602.02830*. [Online]. Available: https://arxiv.org/abs/1602.02830

[16] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.

[17] G. Huang, Z. Liu, L. V. D. Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, vol. 1, no. 2, p. 3.

[18] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Proc. Adv. Neural Inf. Process. Syst.*, 2015, pp. 1135–1143.

[19] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient ConvNets," 2016, *arXiv:1608.08710*. [Online]. Available: http://arxiv.org/abs/1608.08710

[20] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," 2015, *arXiv:1510.00149*. [Online]. Available: http://arxiv.org/abs/1510.00149

[21] J. Van Leeuwen, "On the construction of Huffman trees," in *Proc. ICALP*, 1976, pp. 382–410.

[22] J. Choi, P. I-Jen Chuang, Z. Wang, S. Venkataramani, V. Srinivasan, and K. Gopalakrishnan, "Bridging the accuracy gap for 2-bit quantized neural networks (QNN)," 2018, *arXiv:1807.06964*. [Online]. Available: http://arxiv.org/abs/1807.06964

[23] J. Choi, Z. Wang, S. Venkataramani, P. I-Jen Chuang, V. Srinivasan, and K. Gopalakrishnan, "PACT: Parameterized clipping activation for quantized neural networks," 2018, *arXiv:1805.06085*. [Online]. Available: http://arxiv.org/abs/1805.06085

[24] Z. Liu, B. Wu, W. Luo, X. Yang, W. Liu, and K.-T. Cheng, "Bi-real net: Enhancing the performance of 1-bit CNNs with improved representational capability and advanced training algorithm," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2018, pp. 722–737.

[25] B. Zhuang, C. Shen, M. Tan, L. Liu, and I. Reid, "Towards effective low-bitwidth convolutional neural networks," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 7920–7928.

[26] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, "Incremental network quantization: Towards lossless CNNs with low-precision weights," 2017, *arXiv:1702.03044*. [Online]. Available: http://arxiv.org/abs/1702.03044

[27] 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," 2016, *arXiv:1606.06160*. [Online]. Available: http://arxiv.org/abs/1606.06160

[28] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," 2016, *arXiv:1612.01064*. [Online]. Available: http://arxiv.org/abs/1612.01064

[29] E. Park, D. Kim, S. Yoo, and P. Vajda, "Precision highway for ultra low-precision quantization," 2018, *arXiv:1812.09818*. [Online]. Available: http://arxiv.org/abs/1812.09818

[30] L. Lu, Y. Liang, Q. Xiao, and S. Yan, "Evaluating fast algorithms for convolutional neural networks on FPGAs," in *Proc. IEEE 25th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr./May 2017, pp. 101–108.

[31] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," 2015, *arXiv:1502.03167*. [Online]. Available: http://arxiv.org/abs/1502.03167

[32] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Dec. 2015, pp. 1026–1034.

[33] J. Tobias Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, "Striving for simplicity: The all convolutional net," 2014, *arXiv:1412.6806*. [Online]. Available: http://arxiv.org/abs/1412.6806

[34] M. Lin, Q. Chen, and S. Yan, "Network in network," 2013, *arXiv:1312.4400*. [Online]. Available: http://arxiv.org/abs/1312.4400

[35] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[36] C.-Y. Lee, S. Xie, P. Gallagher, Z. Zhang, and Z. Tu, "Deeply-supervised nets," in *Proc. Artif. Intell. Statist.*, 2015, pp. 562–570.

[37] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," Dept. Comput. Sci., Univ. Toronto, Toronto, ON, Canada, Tech. Rep, Jan. 2009, vol. 1.

[38] R. K. Srivastava, K. Greff, and J. Schmidhuber, "Training very deep networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2015, pp. 2377–2385.

[39] G. Huang, Y. Sun, Z. Liu, D. Sedra, and K. Q. Weinberger, "Deep networks with stochastic depth," in *Proc. Eur. Conf. Comput. Vis.* Springer, 2016, pp. 646–661.

[40] G. Larsson, M. Maire, and G. Shakhnarovich, "FractalNet: Ultra-deep neural networks without residuals," 2016, *arXiv:1605.07648*. [Online]. Available: http://arxiv.org/abs/1605.07648

[41] A. Romero, N. Ballas, S. Ebrahimi Kahou, A. Chassang, C. Gatta, and Y. Bengio, "FitNets: Hints for thin deep nets," 2014, *arXiv:1412.6550*. [Online]. Available: http://arxiv.org/abs/1412.6550

[42] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading digits in natural images with unsupervised feature learning," in *Proc. NIPS Workshop Deep Learn. Unsupervised Feature Learn.*, 2011, vol. 2011, no. 2, p. 5.

[43] S. Zagoruyko and N. Komodakis, "Wide residual networks," 2016, *arXiv:1605.07146*. [Online]. Available: http://arxiv.org/abs/1605.07146

[44] I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio, "Maxout networks," 2013, *arXiv:1302.4389*. [Online]. Available: http://arxiv.org/abs/1302.4389

[45] P. Sermanet, S. Chintala, and Y. LeCun, "Convolutional neural networks applied to house numbers digit classification," in *Proc. 21st Int. Conf. Pattern Recognit.*, Jun. 2012, pp. 3288–3291.

[46] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proc. IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec. 2017.

[47] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Proc. Adv. Neural Inf. Process. Syst.*, 2015, pp. 3123–3131.

[48] F. Li, B. Zhang, and B. Liu, "Ternary weight networks," 2016, *arXiv:1605.04711*. [Online]. Available: http://arxiv.org/abs/1605.04711

[49] S. Gross and M. Wilber. (2016). Training and investigating residual nets. Facebook AI Research, Menlo Park, CA, USA. [Online]. Available: http://torch.ch/blog/2016/02/04/resnets.html

[50] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, 2014.

[51] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *Proc. Int. Conf. Mach. Learn.*, 2013, pp. 1139–1147.

[52] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-net: ImageNet classification using binary convolutional neural networks," in *Proc. Eur. Conf. Comput. Vis.* Springer, 2016, pp. 525–542.

[53] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (FPGA)*, 2015, pp. 161–170.