Vol.2019-ARC-235 No.36
Vol.2019-SLDM-187 No.36
Vol.2019-EMB-50 No.36
2019/3/18

High-Accuracy and Cost-Effective Neural Networks for Embedded Systems

Jiajun Guo[1,a)] Amr Ashmawy[1,b)] Thiem Van Chu[1,c)] Kiyofumi Tanaka[1,d)]

Abstract: Binarized Neural Network (BNN) is a promising technique for embedded inference hardware due to the small hardware cost, but the inference accuracy is degraded compared to a full-precision CNN. In this study, we show that the inference accuracy can be improved by using an ensemble of a few BNNs. In addition, we report our implementation on an FPGA.

1. Introduction

In the last few years, Convolutional Neural Networks (CNNs) have become the state-of-the-art technology in a wide range of fields such as computer vision, speech recognition, and natural language processing. However, CNNs require a considerable amount of computation and power since they use large training sets and models which lead to millions of floating-point parameters and billions of operations. This makes it difficult to deploy CNNs on embedded systems which have very limited power envelopes.

High-end Graphics Processing Units (GPUs) with great computational speed are strong candidates for implementing CNNs. However, they consume a lot of energy, up to 200 Watts. Mobile GPUs designed for embedded systems and mobile devices have a power consumption of a few Watts, at the expense of a severely degraded computational speed.

Recent studies have shown that Field-Programmable Gate Arrays (FPGAs) can provide competitive performance to high-end GPUs in many applications while having similar power envelopes to mobile GPUs, making them an ideal choice for embedded systems. However, conventional CNNs require a large number of floating-point operations along with a large memory for storing floating-point weights and inflight processed data, which FPGAs generally do not handle well. Fortunately, recent trends have made FPGAs more attractive. Courbariaux *et al.* [1], [2] have shown that CNNs with binary weights and activations (Binarized Neural Networks – BNNs) can deliver a comparable degree of accuracy compared with full-precision CNNs. In BNNs, weights and activations have only two possible values (e.g., -1 or +1), so floating-point operations are reduced to binary operations which can be efficiently implemented on FPGAs. Moreover, binary weights and inflight processed data can fit into the FPGA on-chip memory with very low access latency and high bandwidth, which can lead to dramatic performance improvements.

While BNNs can be implemented very efficiently on FPGAs, they may fail to deliver an acceptable level of accuracy in certain cases due to the low precision of weights and activations. As an attempt to improve the prediction/inference accuracy, ensemble methods which train several CNNs and combine them [3] in the inference phase have been proposed. It has been revealed that these methods are effective for full-precision CNNs. They, however, have not been fully investigated for BNNs.

The objective of this work is to raise the prediction accuracy of BNNs with ensemble methods. In addition, we show an implementation approach to achieve high prediction accuracy and cost-effective BNNs for FPGAs.

2. Related Work

In this section, we briefly review CNNs, BNNs, and ensemble methods.

2.1 Convolutional Neural Networks (CNNs)

Since AlexNet [4] competed in the ImageNet Large Scale Visual Recognition Challenge [5] in 2012, CNNs have become the state-of-the-art technology in a wide range of tasks, especially those in computer vision. CNNs are feed-forward neural networks which use convolution operations in place of general matrix multiplications. **Figure 1** shows a typical CNN which consists of convolution layers, pooling layers, and fully connected layers.

2.1.1 Convolution Layer

The layers which apply convolution operations are called convolution layers (**ConvLayers**). A convolution operation has two inputs: input features and kernels. In mathematics, it can be defined as:

[1] Japan Advanced Institute of Science and Technology
JAIST, Asahidai 1-1, Nomi, Ishikawa 923–1292, Japan
[a)] jiajun-guo@jaist.ac.jp
[b)] a.ashmawy@jaist.ac.jp
[c)] thiem@jaist.ac.jp
[d)] kiyofumi@jaist.ac.jp

Vol.2019-ARC-235 No.36
Vol.2019-SLDM-187 No.36
Vol.2019-EMB-50 No.36
2019/3/18

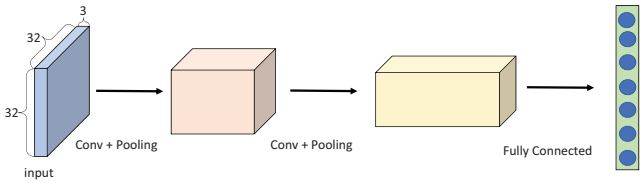

Fig. 1 A typical CNN.

$$h(x) = (f * g)(x) \tag{1}$$

where f represents the input features, g represents the kernels (also called filters), the asterisk denotes the convolution operator, and h represents the output features.

In image recognition, inputs are, in most cases, high-dimensional matrices. When inputs are two-dimensional (2D) images, 2D kernels are used.

Figure 2 shows an example of convolution operations in CNNs. The input feature here is a 4×4 matrix and the kernel is a 3×3 matrix. We slide the kernel across the width and height of the input feature and compute the dot products to obtain a 2×2 output feature. The number of colums or rows by which the kernel is slided in every step is called *stride*. In Fig. 2, *stride* is equal to 1 since the kernel is slided one column/row at a time.

The example in Fig. 2 considers only one input feature. In general, an output feature may be produced from multiple input features. To produce an output feature from M input features, a set of M kernels are required for each input feature. To produce N output features, we need N sets of M kernels.

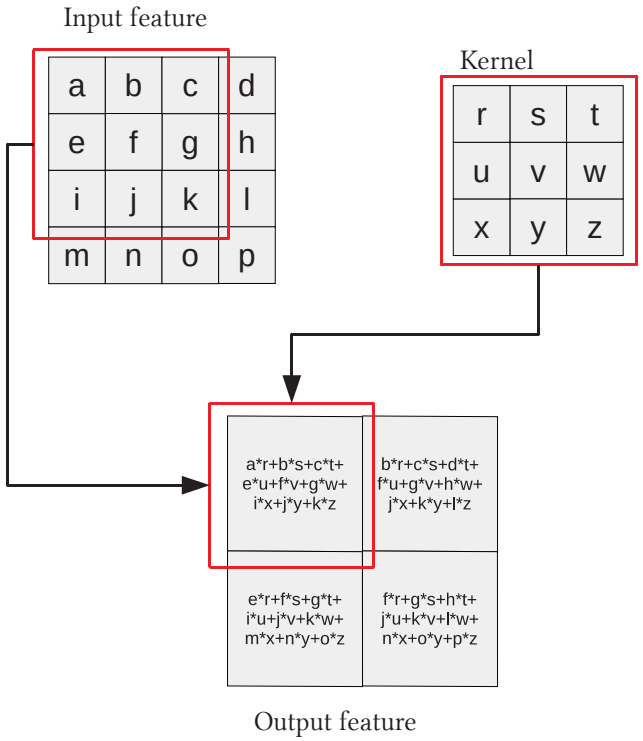

Fig. 2 An example of convolution operations. Here, the input feature is a 4×4 matrix and the kernel is a 3×3 matrix. The kernel is slided across the input feature one column/row at a time (*stride* = 1).

2.1.2 Padding

In CNNs, padding is a basic operation for preserving as much information about the input features as possible in the output side. If padding is not used, there are two problems. First, every time a convolution operation is done, the size of the input to the next layer is reduced. For example, in Fig. 2, the size of the input feature is 4×4 while the size of the output feature is just 2×2. This way, after only a few convolutions, the input features to the next layer become very small. The second problem can be observed in Fig. 2, element **a** at the upper left corner of the input feature is used only once in producing the output feature while some other elements like **f** are used four times. Therfore, information of element **a** may not be well reflected in the output feature.

The above problems are solved by padding zeros to the edges of the input feature like in **Fig. 3**. We can see now that the size of the output feature is 4×4 which is the same as size of the input feature. The elements at the corners of the input feature are now used four times in producing the output feature.

2.1.3 Pooling Layer

A pooling layer downsamples the dimensions of the input to reduce the number of parameters and computations in the following layers. One of the common pooling functions is max pooling among others as average pooling and L2-norm pooling. The max pooling function returns the maximum value within a rectangular region. **Figure 4** shows max pooling with a 2×2 filter and a stride of 2. Each max pooling operation in this case takes the maximum value over 4

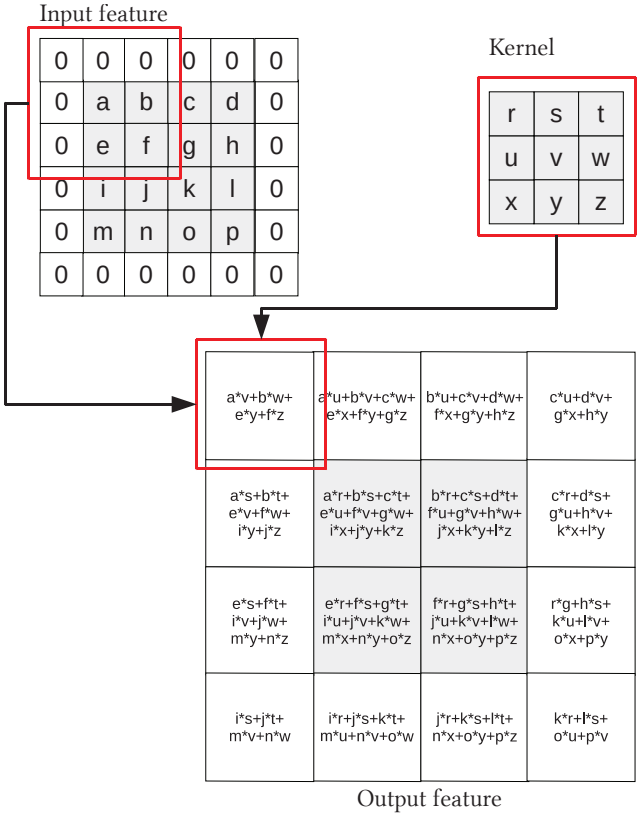

Fig. 3 An example of padding.

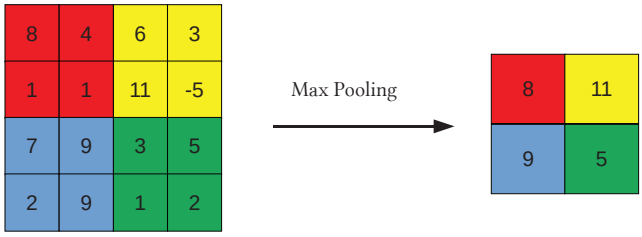

Fig. 4 Max pooling with a 2×2 filter and a stride of 2.

numbers (in a 2×2 region).

2.1.4 Batch Normalization

Batch Normalization (BN) [6] is widely used in neural networks. BN reshapes the distribution of inputs of each layer to a mean of 0 and variance of 1 improving the stability of neural networks. In addition, BN makes a larger learning rate possible, which speeds training a neural network.

Algorithm 1 (from [6]) shows the process of batch normalization. $\mathcal{B}$ is a mini-batch that has m elements. ϵ is a constant added to the variance. γ and β are parameters to be learned. $\mu_{\mathcal{B}}$ and $\sigma^2_{\mathcal{B}}$ are mean and variance of $\mathcal{B}$. The mean and the variance obtained from the training phase are used in the inference phase, so the neural networks only have to execute steps 3 and 4.

2.2 Binarized Neural Networks (BNNs)

Binarized Neural Networks (BNNs) [1], [2] are CNNs whose weights and activations are +1 or -1 as shown in **Fig. 5**. In BNNs, bitwise operations (e.g., XNOR) can be used instead of multiplications. Implementations of dedicated hardware greatly benefit from the fact that BNNs require much smaller memory and lower computational requirements. In the literature [2], BNNs are 7× faster than full-precision CNNs on GPUs.

2.2.1 Binarization Function

Real values are transformed into binary values (-1 or +1) by using one of the following two functions [2]. The first one is a deterministic function:

$$x^b = Sign(x^r) = \begin{cases} +1 & if\ x^r \geq 0 \\ -1 & otherwise, \end{cases} \qquad (2)$$

Algorithm 1 Batch Normalization.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\cdots m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = BN_{\gamma,\beta}(x_i)\}$

1: $\mu_{\mathcal{B}} \longleftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$ //mini-batch mean
2: $\sigma^2_{\mathcal{B}} \longleftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$ //mini-batch variance
3: $\hat{x_i} \longleftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}}$ //normalize
4: $y_i \longleftarrow \gamma\hat{x_i} + \beta \equiv BN_{\gamma,\beta}(x_i)$ //scale and shift

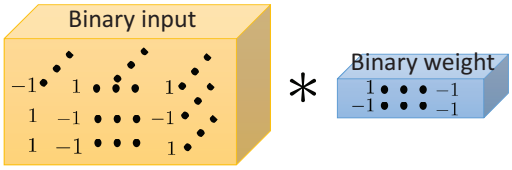

Fig. 5 Binary convolution.

where x^r is a real value and x^b is its binarized value. The second one is a stochastic function:

$$x^b = \begin{cases} +1 & with\ probability\ p = \sigma(x^r) \\ -1 & with\ probability\ 1 - p, \end{cases} \qquad (3)$$

where σ is the hard sigmoid function [2].

The stochastic function is harder to implement in hardware than the deterministic function since the former needs to generate some random bits. On the other hand, `Sign` function has only to extract the sign bit from the real value. Our hardware implementation uses the `Sign` function for binarization.

2.2.2 The First and Last Layers

In BNNs, only binarized values are used in convolution operations, except for the first layer. Inputs to this layer are RGB (Red, Green and Blue) images which cannot be binarized. This, however, is not serious since there are only three input features (channels), R, G, and B. The inputs to the later hidden layers have much more channels (up to 512), and therefore, these layers have higher computational requirements than the first layer.

The last layer outputs a score vector for each target class/label, while the output of other layers is binarized data.

2.3 Ensemble Methods

Ensemble methods train some baseline models then combine them to make predictions using some rules in the inference phase (**Fig. 6**). In [3], Ju *et al.* applied several ensemble methods including Unweighted Average, Majority Voting, Bayes Optimal Classifier, and Super Learner [7]) to some popular CNNs such as Network in Network (NiN) [8], GoogLeNet [9], VGG Net [10], and Residual Network [11]. It was shown that these ensemble methods improved the overall prediction accuracy of the neural networks.

While the binarization in BNNs degrades the prediction accuracy, it is expected that ensemble methods compensate for the losses. In [3], although Super Learner works best, it is too complex and not suitable for hardware implementations. Therefore, we select the second-best one, Unweighted Average, 1% to 2% lower accuracy. The computation of Unweighted Average is simple enough for hardware implementation.

Unweighted Average just takes an average of the scores or probabilities output of all the neural network models. In general, probabilities are obtained after a softmax function is applied. The softmax function is formulated as:

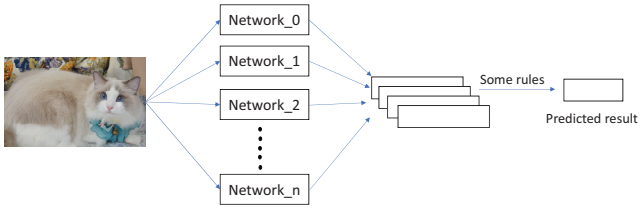

Fig. 6 Ensemble methods.

$$p_{ij} = \texttt{softmax}(\vec{s}_i)[j] = \frac{\exp(\vec{s}_i[j])}{\sum_{k=1}^{K} \exp(\vec{s}_i[k])}, \quad (4)$$

where $\vec{s}_i$ is the score vector output from the last layer of the i-th neural network, K is the number of classes/labels, and p_{ij} is the probability of neural network i predicting class/label j.

The average of output scores called "Before Softmax". On the other hand, "After Softmax" is the average of probabilities. Before Softmax computes the average score vector as follows:

$$\vec{s}_{\texttt{avg}} = \frac{\sum_{i=1}^{N} \vec{s}_i}{N}, \quad (5)$$

where N is the number of neural networks used and $\vec{s}_i$ is the score vector output from the last layer of the i-th neural networks. On the other hand, After Softmax computes the average probability of predicting class/label j as follows:

$$p_{\texttt{avg}}[j] = \frac{\sum_{i=1}^{N} p_{ij}}{N}, \quad (6)$$

where p_{ij} is the probability of neural network i predicting class/label j calculated by formula (4). In our evaluation, the performances of these two Unweighted Average methods were almost the same.

3. Software Implementation and Evaluation

In this section, we show our implementation on software and evaluate the prediction accuracy.

3.1 Experimental Environment

3.1.1 Dataset

We use the CIFAR-10 dataset [12]. It is an RGB image dataset that has been widely used for testing image classification.

CIFAR-10 contains 60,000 32×32 RGB images that are divided into 10 classes, each has 6,000 images. The whole dataset is split into two parts: 50,000 images for training and 10,000 images for testing (inference).

To increase the size of training dataset, we randomly flipped and changed the brightness and contrast of the training images. We did not preprocess the test images and just used the original ones.

3.1.2 Deep Learning Software Library

There are several popular deep learning environments such as Caffe2, PyTorch, and TensorFlow. TensorFlow is the most popular one [13]. Hence, we selected TensorFlow as our software environment.

3.2 Network Architecture

This section shows the architecture of our BNNs and a full-precision CNN that we created for comparison with BNNs. These neural networks use the same basic model architecture. The only difference is the activation function. In BNNs, the activation function is the binarization function that we described in Section 2.2.1. On the other hand, the Rectified Linear Unit (ReLU) function [14] is applied to full-precision CNN. We choose this activation function because it does not saturate (in positive region) and is computationally efficient. ReLU is formulated as follows:

$$\text{ReLU} = \max(0, x). \quad (7)$$

Figure 7 and **Table 1** show the basic architecture and parameter settings.

3.3 Evaluation

We created four BNNs and combined them using the Unweighted Average method. We also created a full-precision CNN having the same structure as the BNNs.

We selected the mini-batch gradient descent [15] to accelerate the training, with size 128. We also applied the learning rate decay to make the training faster. The following formula is used to update the learning rate α:

$$\alpha_n = \alpha_{n-1} \times \texttt{decay\_rate}^n, \quad (8)$$

where `decay_rate` is equal to 0.9 and n is the number of epochs. One epoch indicates one pass through the whole training set. α_n is the learning rate of the n-th epoch.

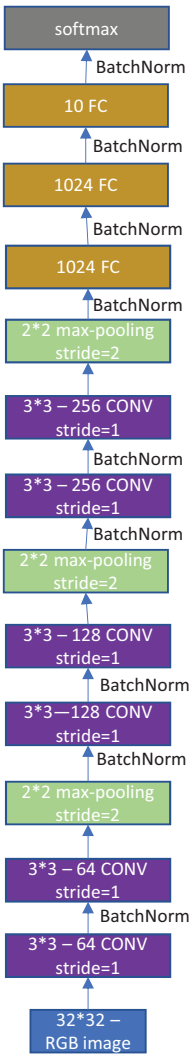

Fig. 7 Network architecture.

Table 1 Architecture of our constructed BNNs and CNN.

Layer Type	Kernel Size	Output Shape	#Parameters
InputLayer		(32, 32, 3)	0
ConvLayer_1	3×3	(32, 32, 64)	1728
BatchNorm_1		(32, 32, 64)	64
Activation_1		(32, 32, 64)	0
ConvLayer_2	3×3	(32, 32, 64)	36,846
MaxPooling_1	2×2	(32, 32, 64)	0
BatchNorm_2		(16, 16, 64)	64
Activation_2		(16, 16, 64)	0
ConvLayer_3	3×3	(16, 16, 128)	73,728
BatchNorm_3		(16, 16, 128)	128
Activation_3		(16, 16, 128)	0
ConvLayer_4	3×3	(16, 16, 128)	147,456
MaxPooling_2	2×2	(8, 8, 128)	0
BatchNorm_4		(8, 8, 128)	128
Activation_4		(8, 8, 128)	0
ConvLayer_5	3×3	(8, 8, 256)	294,912
BatchNorm_5		(8, 8, 256)	256
Activation_5		(8, 8, 256)	0
ConvLayer_6	3×3	(8, 8, 256)	589,824
MaxPooling_3	2×2	(4, 4, 256)	0
BatchNorm_6		(4, 4, 256)	256
Activation_6		(4, 4, 256)	0
FullyConnected_1		(1024)	4,194,304
BatchNorm_7		(1024)	1,024
Activation_7		(1024)	0
FullyConnected_2		(1024)	1,048,576
BatchNorm_8		(1024)	1,024
Activation_8		(1024)	0
FullyConnected_3		(10)	10,240
BatchNorm_9		(10)	10
Total trainable parameters: 6,400,568			

We also need an optimizer algorithm that is in charge of updating the parameters. There are several candidates such as Momentum [16], RMSprop [17], and Adam [18]. We selected Adam Optimizer which was found to be the best by testing.

We trained all the neural networks for 100 epochs and tested them with the CIFAR-10 test dataset. The accuracy of each BNN in the 100th epoch is shown in **Table 2**. The best one is Network_4 has 0.814 accuracy. The accuracy of the full-precision CNN is 0.868 as shown in **Table 3**. It is obviously higher than all the BNNs.

We used the Unweighted Average method to combine the baseline BNNs. The test results are shown in **Table 4**. The accuracy of Before Softmax and that of After Softmax are close. The accuracy is improved compared to that without ensemble and is quite close to that of full-precision CNN.

Table 2 Accuracy of each baseline BNN.

	Accuracy
Network_1	0.810
Network_2	0.813
Network_3	0.802
Network_4	0.814

Table 3 Accuracy of the full-precision CNN.

	Accuracy
full-precision	0.868

Table 4 Accuracy of ensemble of BNNs.

	Accuracy
before softmax	0.861
after softmax	0.860

4. Hardware Implementation and Evaluation

We designed the BNN described in Section 3 with VHDL RTL code and simulated it using Synosys VCS. Then, Xilinx Vivado 2018.3 is used for synthesizing and implementing the design. The target FPGA device is Virtex-7 VX485T. This section describes the details of the implementation and reports the utilization of FPGA resources as well as the execution time for processing a 32×32 RGB image.

4.1 Hardware Implementation

The BNN hardware that we designed is only for the inference phase. Its organization is shown in **Fig. 8**. Layer1, Layer3 and Layer5 include ConvLayer, BatchNorm, and binarization. Layer2, Layer4 and Layer6 include ConvLayer, max-pooling, BatchNorm, and binarization. FC1 and FC2 include fully-connected layer, BatchNorm and binarization. FC3 includes fully-connected layer only.

Between each two layers, there is an input buffer which stores the output of the previous layer. When the processing of a layer is finished, the following layer starts. The first input buffer stores the values read from the input images.

The *weights memory* stores the weights required by the ConvLayers and fully-connected layers. The *BN memory* stores the parameters required by the batch normalization processing.

4.1.1 Convolution

If the number of input features of a convolution layer is M, that convolution layer is said to have M input channels. In this case, to calculate an output feature, M kernels are required. In our design, the size of each kernel is fixed to 3×3 for every convolution layer.

In each clock cycle, as shown in **Fig. 9**, we perform the channel-wise dot-products of the kernels and a portion of the input features and then sum up all of the values to generate a value in the output feature. We call this operation *single step convolution*.

In BNNs, input features and kernels are composed of binary values. In theory, the binary values are +1 or -1. However, in hardware, one bit can have only two values 0 or 1. In our BNN hardware, +1 and -1 are mapped to 1 and 0,

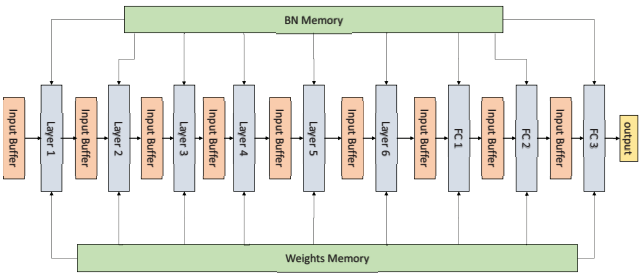

Fig. 8 The organization of our BNN hardware.

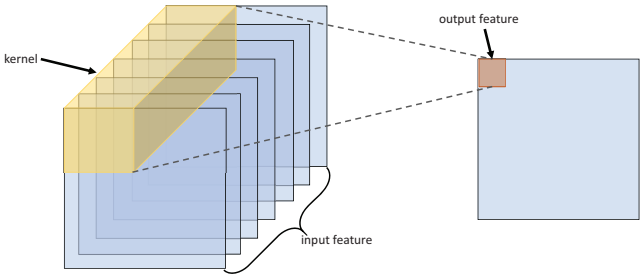

Fig. 9 Single step convolution.

respectively. To adapt to the change of binary values, we calculate the dot-product of two bit vectors as below [19].

$$\mathbf{a} \cdot \mathbf{b} = \mathbf{C} - 2 \times \texttt{bitcount}(\mathbf{a}\ \texttt{XOR}\ \mathbf{b}) \qquad (9)$$

where function `bitcount` takes a bit vector as an input and outputs the number of ones in that bit vector. **C** is the bit-width of **a** and **b**. For example, if **a** and **b** are 9-bit vectors, then **C** is equal to 9.

Figure 10 shows the computation of a single step convolution operation in our hardware design. Here, x_i and w_i ($i = 1, 2, \cdots$, M where M is the number of input channels) are respectively a 9-bit portion of the input feature in the i-th input channel and the 9-bit kernel for this channel. x_i and w_i are 9-bit vectors since the kernel size is 3×3. The operation `OP` is the dot-product defined in Equation 9. The final output is a single integer number.

An output feature is obtained by repeating the single step convolution operation on different portions of the input features. In Fig. 9, the kernel is slided from the top left to the bottom right of the input features and the single step convolution operation is performed at each position.

We perform padding for the input features as described in Section 2.1.2. Note that the padded numbers are *real* zeros while the zeros within the input features indicate -1. Because of this, Equation (9) must be modified for the cases involving padded data. We have nine patterns as shown in **Fig. 11**. In this figure, the yellow cells contain padded zeros. Pattern P4 does not involve any padded data and thus Equation (9) can be applied without any modifications (**C** is equal to 9 and the `XOR` operation is performed normally).

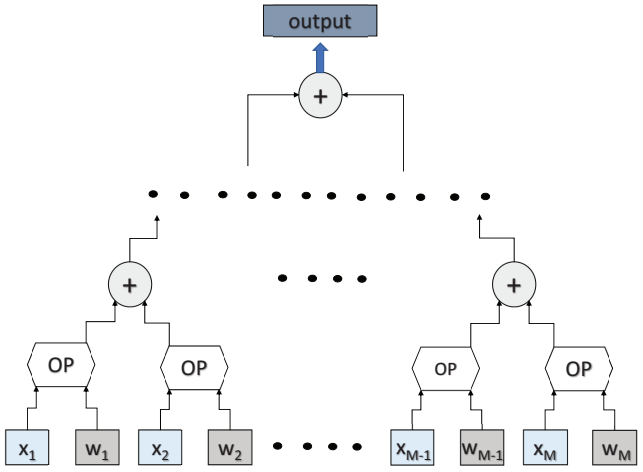

Fig. 10 Computation in the single step convolution operation.

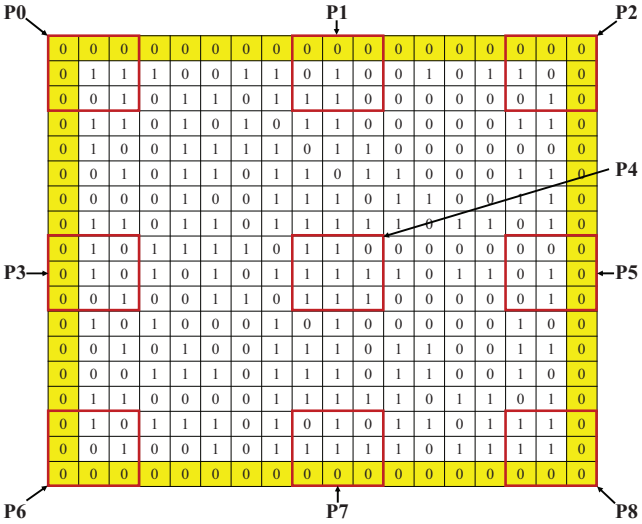

Fig. 11 Nine patterns when performing the single step convolution operation.

However, this is not the case for the other patterns. The `XOR` operation is modified so that the result bits at the padded positions are zeros regardless of the values in the kernel. In patterns P0, P2, P6, and P8, **C** is set to 4 since there are 5 padded cells. Similarly, in patterns P1, P3, P5, and P7, **C** is set to 6 since there are 3 padded cells.

4.1.2 Max Pooling

The input data of a pooling layer are integer numbers generated by the previous ConvLayer. In our design, pooling layers are max-pooling which extracts the maximum values from four values generated one at a time from the single step convolution. A pooling layer outputs a value to the next layer every four clock cycles since it takes one clock cycle for the previous ConvLayer to output each value.

4.1.3 Batch Normalization and Activation

Batch normalization (BatchNorm) is applied to the outputs of pooling layers (or convolution layers if they are not followed by a pooling layer) and fully connected layers. The inputs of BatchNorm are integer numbers. The outputs of BatchNorm are passed through an activation function. We use the `Sign` activation function which does not care about the absolute value of the input but only whether it is a positive or negative number. Based on this observation, we approximate BatchNorm as below.

We combine Step 3 and Step 4 in the Algorithm 1 as:

$$y_i \longleftarrow \gamma \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \beta \qquad (10)$$

where γ is equal to 1 in our design. $\mu_{\mathcal{B}}$, $\sigma_{\mathcal{B}}^2$, ϵ, and β are constants obtained from training. Therefore, we transform Equation 10 as:

$$y_i \longleftarrow \frac{x_i + (\beta\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon} - \mu_{\mathcal{B}})}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}. \qquad (11)$$

Since $\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}$ is positive, we only need to determine the sign of $x_i + (\beta\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon} - \mu_{\mathcal{B}})$. We compute $\beta\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon} - \mu_{\mathcal{B}}$

by software in advance. The results are then rounded to integers using `floor` function (also by software). BatchNorm is now transformed as:

$$c_{\mathcal{B}} = \texttt{floor}(\beta\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon} - \mu_{\mathcal{B}}), \tag{12}$$

$$y_i = x_i + c_{\mathcal{B}}. \tag{13}$$

We use the two's complement binary expression to denote $c_{\mathcal{B}}$ and store it in FPGA on-chip memory (*BN Memory* in Fig. 8).

As mentioned before, the output of BatchNorm is passed through the `Sign` activation function. The output bit value, a_i^b, is computed as:

$$a_i^b = \texttt{Sign}(y_i). \tag{14}$$

`Sign` just takes the highest bit of the input y_i and reverses it since y_i is denoted by the two's complement binary expression.

4.1.4 Fully Connected Layers

Figure 12 shows the structure of the fully-connected layers. Weights are called *neurons* in these layers. Each neuron is a vector whose width is the same as the input. They are stored in the *Weights Memory* (Fig. 8).

In Fig. 12, the operation `OP` is the dot-product defined in Equation 9. The output of this operation is an integer number. This number is passed through BatchNorm and then binarization (except for the last fully connected layer) to produce one output bit (forming one integer number from the last fully connected layer).

4.2 Evaluation

We performed RTL simulations of our BNN hardware by using Synopys VCS. CIFAR-10 images were used. We confirmed the correctness of our hardware by comparing its output results with those reported by TensorFlow.

We synthesized and implemented the VHDL design of a single BNN with Xilinx Vivado 2018.3. The target FPGA device is Virtex-7 VX485T (xc7vx485tffg1761-2).

Table 5 shows the utilization of resources. The BNN requires 57.68% LUTs, 28.04% FFs, and 23.45% BRAM of the FPGA. Focusing on the usage of LUTs and FFs, we can see that it is not possible to implement an ensemble of four BNNs using the current FPGA. For this purpose, a larger FPGA is required.

On the other hand, BRAMs' capacity is enough to store

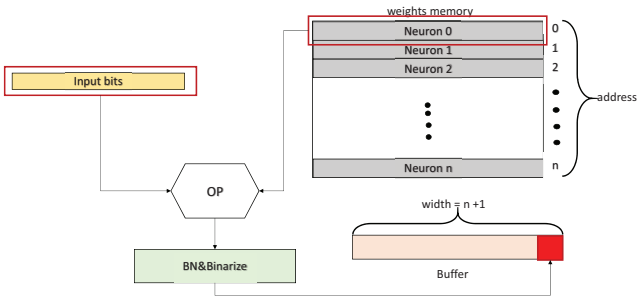

Fig. 12 Structure of the fully connected layers.

Table 5 Utilization of hardware resources for a single BNN.

Resource	Utilization	Available	Utilization %
LUTs	175,111	303,600	57.68
FFs	170,251	607,200	28.04
BRAMs	241.50	1030	23.45

all the parameters for four BNNs. In our design, the four BNNs have the same structure. Therefore, it is possible to emulate the ensemble of four BNNs by operating a BNN four times with different memory addresses, so that different parameters are used in each time. A single BNN spends 231,452 clock cycles to process an image. Executing 4 times needs 925,808 clock cycles. Considering the current implementation in which the BNN runs at 10Mhz frequency, it can process about 10.8 CIFAR-10 images per second.

5. Conclusion

BNNs provide an approach to implement a neural network on dedicated hardware easily and efficiently. But the prediction accuracy is lower than full-precision CNNs.

In this work, we proposed an approach to design a high-accuracy and cost-efficient neural networks through ensemble methods. We trained and tested four BNNs and a full-precision CNN which has the same structure for the CIFAR-10 dataset. We combined the four BNNs with unweighted average methods and compared the inference accuracy with the full-precision CNN. It was shown that the accuracy is obviously improved compared to without the ensemble methods and quite close to the full-precision CNN.

In addition, we designed a BNN in VHDL and implemented it on an FPGA. Form the implementation reports, it is found that our current BNN design cannot be extended to an ensemble of four BNNs and implemented on an FPGA of moderate capacity. An alternative method is to emulate an ensemble of four BNNs by time-multiplexing on a single BNN.

In the future, we will optimize the resource requirements and performance of our hardware design and try to implement an ensemble of a few BNNs on an FPGA.

References

[1] M. Courbariaux, Y. Bengio, J-P. David, "BinaryConnect: Training Deep Neural Networks with Binary Weights during Propagations," Advances in Neural Information Processing Systems 28, Curran Associates, Inc, pp.3123–3131, 2015.
[2] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, Y. Bengio, "Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1," arXiv e-prints, 2016. (https://arxiv.org/abs/1602.02830)
[3] C. Ju, A. Bibaut, M. J. van der Laan, "The Relative Performance of Ensemble Methods with Deep Convolutional Neural Networks for Image Classification," Journal of Applied Statistics, Vol.45, No.15, pp.2800–2818, 2018.
[4] A. Krizhevsky, I. Sutskever, G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," Proc. of Intl. Conf. on Neural Information Processing Systems (NIPS), Vol.1, pp.1097–1105, 2012.
[5] `http://www.image-net.org/challenges/LSVRC/`
[6] S. Ioffe, C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," Proc. of Intl. Conf. on Machine Learning, Vol.37, pp.448–456, 2015.
[7] M. J. van der Laan, E. C. Polley, A. E. Hubbard, "Super

Learner," Statistical Applications in Genetics and Molecular Biology, Vol 6, No, 1, Article 25, 2007.
[8] M. Lin, Q. Chen, S. Yan, "Network in Network," CoRR, vol. abs/1312.4400, 2013. (http://arxiv.org/abs/1312.4400)
[9] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, "Going Deeper with Convolutions," Proc. of IEEE Conf. on Computer Vision and Pattern Recognition (CVPR), pp.1–9, 2015.
[10] K. Simonyan, A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," Proc. of Intl. Conf. on Learning Representations (ICLR), pp.1–14, 2015.
[11] K. He, X. Zhang, S. Ren, J. Sun, "Deep Residual Learning for Image Recognition," Proc. of IEEE Conf. on Computer Vision and Patter Recognition (CVPR), pp.770–778, 2016.
[12] https://www.cs.toronto.edu/~kriz/cifar.html
[13] M. Abadi, et al., "TensorFlow: A System for Large-Scale Machine Learning," Proc. of USENIX conf. on Operating Systems Design and Implementation, pp.265–283, 2016.
[14] X. Glorot, A. Bordes, and Y. Bengio, "Deep Sparse Rectifier Neural Networks," Proc. of Intl. Conf. on Artificial Intelligence and Statistics, pp.315–323, 2011.
[15] G Hinton, N Srivastava, K Swersky, "Neural Networks for Machine Learning," Coursera, video lectures 264, 2012.
[16] B. T. Polyak, "Some methods of speeding up the convergence of iteration methods," USSR Computational Mathematics and Mathematical Physics, Vol.4, NO.5, pp.1–17, 1964.
[17] T. Tieleman, G. Hinton, "Lecture 6.5-rmsprop: Divide the Gradient by a Running Average of Its Recent Magnitude," COURSERA: Neural Networks for Machine Learning, Vol.4, No.2, pp.26–31, 2012.
[18] D. P. Kingma, J. Ba, "Adam: A Method for Stochastic Optimization," Proc. of Intl. Conf. on Learning Representations, 2015.
[19] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, Y. Zou, "Dorefa-net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients," arXiv preprint arXiv:1606.06160, 2016.