FxpNet: Training deep convolutional neural network in fixed-point representation

Xi Chen*
Department of Computer Science and Technology
Tsinghua University
100084, Beijing, China
aaron.xichen@gmail.com

Xiaolin Hu
Department of Computer Science and Technology
Tsinghua University
100084, Beijing, China
xlhu@tsinghua.edu.cn

Ningyi Xu, Hucheng Zhou
Microsoft Research Asia
100080, Beijing, China
{ningyixu, huzho}@microsoft.com

***Abstract*—We introduce FxpNet, a framework to train deep convolutional neural networks with low bit-width arithmetics in both forward pass and backward pass. During training procedure, FxpNet further reduces the bit-width of stored parameters (also known as primal parameters) by adaptively updating their fixed-point formats. These primal parameters are ususally represented in full resolution of floating-point values in previous binarzied and quantized neural networks. In FxpNet, during forward pass fixed-point primal weights and activations will first be binarized before computation, while in backward pass all gradients are represented as low resolution fixed-point values and then accumulated to corresponding fixed-point primal parameters. To have highly efficient implementations in FPGAs, ASICs and other dedicated devices, FxpNet introduces Integer Batch Normalization (IBN) and Fixed-point ADAM (FxpADAM) methods to further reduce the required floating-point operations, which will save considerable power and chip area. The evaluation on CIFAR-10 dataset indicates the effectiveness that FxpNet with 12-bit primal parameters and 12-bit gradients achieves comparable prediction accuracy with state-of-art binarized and quantized neural networks. The corresponding training code implemented in TensorFlow is available online.**

I. Introduction

In recent years, advances in deep Convolutional Neural Networks (CNNs) have changed the landscape of many nontrival Artificial Intelligence tasks, such as computer vision [1], [2], natural language processing [3], [4] and speech recognition [5], [6]. CNNs usually contain large amount of learnable parameters and most of them took days or weeks to train on one or more fast and power-hungry GPUs [7]–[10]. There are substantial research efforts [9]–[11] on speeding up the offline training and online serving, especially in power-efficienct devices such FPGAs and dedicated ASICs [12]–[17]. Among all these methods, model quantization [9]–[11], [18], [19] has been considered as the most promising approach, which has not only significantly accelerated the speed and increased the power-efficiency, but also achieved comparable accuracy. Model quantization tries to quantize the model parameters (as well as activations and gradients) to low bit-width values, while model binarization [20]–[25] further pushes the limit of quantization by extremely quantizing the parameter as one single bit (-1 and 1). As a consequence, during inference, memory footprint and memory accesses can be drastically reduced, and most arithmetic operations can be implemented with bit-wise operations, i.e., binary convolution kernel [22].

However, all these state-of-the-art methods neglect the quantization of `primal` parameters, which is defined as those high resolution weights and bias stored to be updated across different training mini-batches. These `primal` parameters will be quantized or binarized every time before forward pass, and the associated gradient accumulation are still performed in floating-point domain. Thus, the FPGA and ASIC still need to implement expensive floating-point multiplication-accumulation to handel parameter updates, and even the much more expensive nonlinear quantization arithmetics.

In this paper, we present FxpNet that pushes the limit of quantization and binarization by representing the `primal` parameters in fixed-point format, which has never been considered before in binarized networks, to the best of our knowledge. This brings advantages in two folds: 1). Decreasing the bitwidth of `primal` parameters will dramatically reduce total memory footprint. For example, 8-bit fixed-point representation can reduce the total memory footprint by $4\times$ compared with the floating-point counterpart. This makes it possible to place the parameters in on-chip memory rather than off-chip memory on dedicated hardware devices like FPGAs and ASICs, which means 100x energy efficiency of memory accessing with 45nm CMOS technology [10]. 2). Low bit-width of fixed-point arithmetic operations in FPGAs and ASICs are much more faster and energy efficient than floating-point ones [16]. Besides, fixed-point operators generally will dramatically reduce logic usage and power consumption, combined with higher clock frequecy, shorter pipelines, and increased throughput capabilities. Combining these advantages with binarized inference, it is possible to make highly efficient computing devices that are flexbible enough to train itself in a new and continuing changing environment with limited power budget.

Specifically, this paper makes the following contributions:

*This work was done when Xi Chen was an intern at MSRA, supervised by Ningyi Xu

- We introduce FxpNet to train a binarized neural network, whose `primal` parameters and gradients are represented as adaptive fixed-point values. In forward pass, both fixed-point `primal` weights and activiations are binarized before computation. In backward pass gradients are also quantized to low resolution fixed-point values and then accumulated to the corresponding fixed-point `primal` parameters. In this way, bit convolution kernels [22] can be used in FxpNet to accelerate both training and inference.
- To have highly efficient implementation in FPGAs and dedicated ASICs, FxpNet further adopts linear rather than non-linear quantization approach to save the quantization cost, and utilizes Integer Batch Normalization (IBN) and Fixed-point ADAM (FxpADAM) methods to further reduce the required floating-point operations.
- The evaluation on CIFAR-10 dataset indicates the effectiveness that FxpNet with 12-bit `primal` parameters and 12-bit gradients achieves the similar accuracy as the state-of-art BNN [22] with the float (32-bit) counterparts.
- The corresponding training code implemented in TensorFlow [26] is available online [1].

II. FxpNet

In this section we first introduces vanilla BNN, which is derived from popular BNNs [22], [25]. The intention is to design a concise BNN which serves as a starting point and baseline of FxpNet by taking both computation efficiency and prediction accuracy into consideration. We then describe the fixed-point representation as well as critical techniques in FxpNet, and give a detailed description of the training algorithm.

A. Vanilla BNN

As indicated by [20], [22], [23], [25], binarizing weight and activation can significantly speedup the performance by using the bit convolution kernels. There are two binarization approaches, deterministic and stochastic, used to transform floating-point value into one single bit. Stochastic binarization could get slightly better performance [25] at the cost of more complex implementation since it requires the hardware to generate random bits when quantizing. Thus, we propose to use only the deterministic binarization method (a simple sign function):

$$w^b = \mathrm{sign}(w) = \begin{cases} +1 & w \geq 0, \\ -1 & otherwise. \end{cases} \tag{1}$$

Binarization dramatically reduces computation and memory consumption in forward pass, nevertheless, the derivative of the sign function is 0 almost everywhere, makes the gradients of the cost c can't be propagated in backward pass. To address this problem, we adopt the "straight-through estimator" (STE) method [5], [27], and use the same STE formulation in [22]:

$$\begin{aligned} \textbf{Forward}: & \ r_o = \mathrm{sign}(r_i) \\ \textbf{Backward}: & \ \frac{\partial c}{\partial r_i} = \frac{\partial c}{\partial r_o} \mathbf{1}_{|r_i| \leq 1} \end{aligned} \tag{2}$$

[1]https://github.com/aaron-xichen/FxpNet

Above STE preserves the gradient information and cancels the gradient when r_i is too large. No-cancelling will cause significant performance drop. As also pointed out in QNN [25], STE can also be seen as applying the well-known *hard tanh* activation function to r_i, defined as

$$HT(r_i) = \begin{cases} +1 & r_i > 1, \\ r_i & r_i \in [-1, 1], \\ -1 & r_i < -1. \end{cases} \tag{3}$$

Correspondingly, the derivative of *hard tanh* is defined as

$$\frac{\partial HT(r_i)}{\partial r_i} = \begin{cases} 0 & r_i > 1, \\ 1 & r_i \in [-1, 1], \\ 0 & r_i < -1. \end{cases} \tag{4}$$

which is exactly the STE defined in Equation 2. With Equation 2 and 4, BNN binarizes both activations and weights during the forward pass, while still reserves real-valued gradients of weights to guarantee Stochastic Gradient Descent (SGD) to work well. BNN further proposes shift-based Batch Normalization (BN) and shift-based ADAM learning rule to accelerate training and reduce the impact of weights' scale. The shift operations are intended to replace the expensive multiplications, while here we argue that required log operations are also expensive. So we reserve the floating point version of BN and ADAM as the baseline. A detailed description of our baseline BNN can be found in section III.

B. Fixed-Point Format

FxpNet maintains primal weights and gradients as fixed-point numbers. Differ from floating point format which has a sign, an exponent and a mantissa, fixed-point format consists of a l-bit signed integer mantissa and a global scaling factor (e.g. $2^{-\mathrm{n}}$) shared among all fixed-point values, i.e.,

$$\mathbf{v} = \begin{pmatrix} v_1 \\ \vdots \\ v_K \end{pmatrix} = \begin{pmatrix} m_1 \\ \vdots \\ m_K \end{pmatrix} \cdot 2^{-\mathrm{n}} \tag{5}$$

where $m_1, \cdots, m_K, \mathrm{n} \in \mathbb{Z}$. Here the integer n actually indicates the radix point position of the l-bit fixed-point number. We use the following conversion formulation to quantize a floating-point value x into a l-bit fixed-point value with the scaling factor $2^{-\mathrm{n}}$:

$$\mathrm{FXP}(x, l, n) = \mathrm{Clip}(\lfloor \frac{x}{2^{-n}} + 0.5 \rfloor \cdot 2^{-n}, \mathrm{MIN}, \mathrm{MAX}) \tag{6}$$

where

$$\begin{cases} \mathrm{MIN} = -2^{l-1} * 2^{-n} \\ \mathrm{MAX} = (2^{l-1} - 1) * 2^{-n} \end{cases} \tag{7}$$

Note that 1). Equation 6 defines both the saturating behavior and rounding to nearest behavior (ties are rounded to positive infinity); 2). The MIN and MAX values are defined according to the range of l-bit integer in two's complement, which fully utilizes all ordinality 2^l and makes addition/substraction circuits simple.

C. Quantization with Adaptive Fixed-point Scheme

It is well known that the magnitudes of weight, activation and gradient will fluctuate during the whole training procedure. We demonstrate this characteristics with our result in Figure 1,

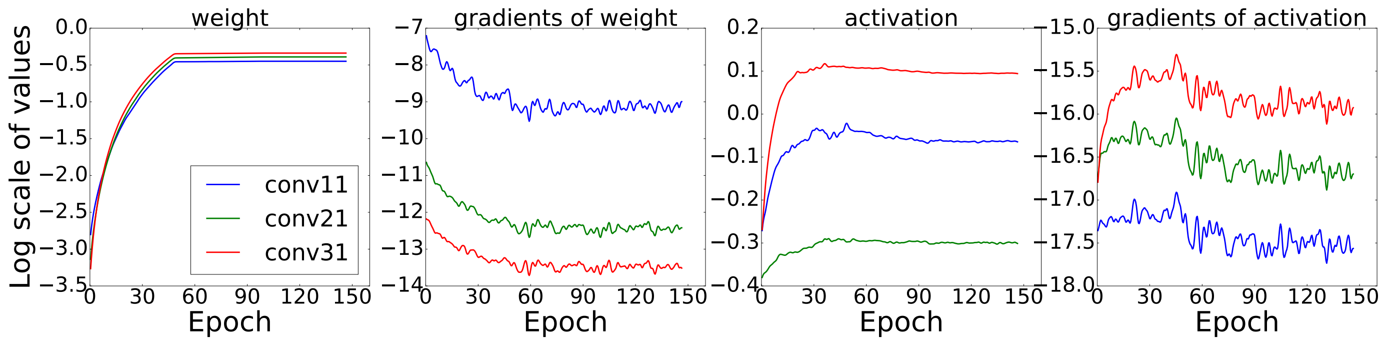


Fig. 1. Mean(Abs($\cdot$)) of weights, activations and corresponding gradients in $\log 2$ scale

where $conv11$, $conv12$ and $conv31$ are three different layers of a 5-layer floating point CNN on CIFAR-10 dataset (see Session III). It can be observed that gradients for both activation and weight diminish slowly, and weights and activations boost rapidly at the beginning and converge with tiny changes.

Clearly, in order to capture the change of parameters and gradients and fit their actual value distribution, it is necessary to assign different bit-widths and scaling factors to different groups of variables in different layers. FxpNet utilizes the dynamic fixed-point transformation [28] with adaptive update of scaling factor [18]. Algorithm 1 depicts the details. In particular, differs from $\mathrm{Clip}(\cdot)$ in Equation 6, $\mathrm{OverflowRate}(\cdot)$ from Alrogithm 1 will count the number of values that are not in $[\min, \max]$. Section II-E explores the effectiveness regarding hyper-parameters such as the initial bitwidth, scaling factor, and the overflow threshold.

Algorithm 1 Adaptive update of scaling factor.

```
Require: matrix $\mathbf{M}$, scaling factor $2^{\mathrm{sf}_t}$, bitwidth bw, overflow
    threshold thr
Ensure: $2^{\mathrm{sf}_{t+1}}$
1: if $\mathrm{OverflowRate}(\mathbf{M}, \mathrm{bw}, \mathrm{sf}_t) \geq \mathrm{thr}$ then
2:     $\mathrm{sf}_{t+1} \leftarrow \mathrm{sf}_t - 1$
3: else if $\mathrm{OverflowRate}(\mathbf{M}, \mathrm{bw}, \mathrm{sf}_t + 1) < \mathrm{thr}$ then
4:     $\mathrm{sf}_{t+1} \leftarrow \mathrm{sf}_t + 1$
5: else
6:     $\mathrm{sf}_{t+1} \leftarrow \mathrm{sf}_t$
7: end if
```

Compared with binarized weights and activations, gradients usually need higher precison [18], [24]. Hubara et al. [25] (QNN) points out that linear quantization approach does not converge well and they instead use a logarithmic method. DoReFa-Net [24] utilizes well-designed and complicated non-linear function to quantize the gradient, which contains costly division operations. All these non-linear operations will inevitably increase the computation overhead, especially on FPGAs. Table I compares the gradient quantization approaches used in existing works, where the area cost of quantization operations is estimated if they are implemented in FPGA.

We choose to adopt linear quantization approach to trade for simplicity. While simply propogate gradients with the same scaling factor (statically) will introduce too strong regularization and impede convergence (see III-B for details), FxpNet revise the potential accuracy loss by the adaptive scaling factors among layers, as detailed in the following subsections.

TABLE I
QUANTIZATION OPERATIONS OF DIFFERENT METHODS

Method	Quantization operation	LEs[1]	DSPs[1]
DoReFa-Net [24]	division	907	4
QNN [25]	log	1504	4
Miyashita et al. [19]	log	1504	4
FxpNet	linear($l = 16$)	35	0

[1] Logical Elements (LEs) and Digital Signal Processors (DSPs)

D. Forward Pass

With all the techniques aforementioned, we begin to introduce the framework to train FxpNet. Firstly we give the details of forward pass.

Figure 2 depicts the fixed-point building block (FPB) of CNNs.

- All grey squares represent fixed-point variables, while all black squares are binarized variables.
- W_k^{fxp} and b_k^{fxp} represent the primal weight and bias of layer k, respectively. In FxpNet, they are represented as fixed-point format rather than floating-point format as in previous works. The primal variables (weights and bias) will be iteratively updated during the whole training procedure.
- In forward pass, primal weights are first binarized (Blue layer), which converts the input to $+1$ or -1 according to sign function (Equation 1). X_k^b represents either input images (for the first block) or output binarized activation of previous layer. In the former case, X_k^b can be regarded as an 8-bit integer vector (0-255), while in the latter case X_k^b is a binary ($+1$ or -1) vector. In both case Conv operation only contains integer multiplication and accumulation and can be computed by bit convolution kernels [22], [25]. Note that the first layer is processed as the following (in the same way as BNN and QNN):

$$s = x \cdot w^b;$$
$$s = \sum_{n=1}^{8} 2^{n-1}(x^n \cdot w^b). \tag{8}$$

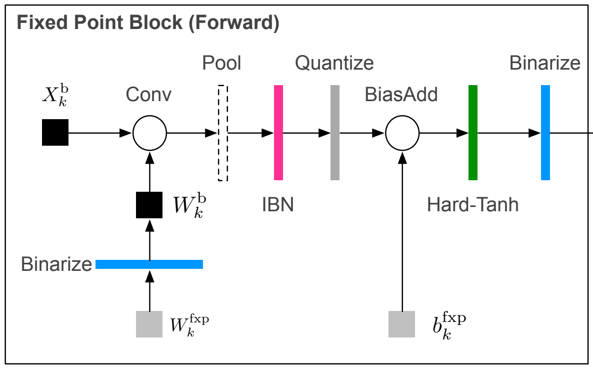


Fig. 2. Forward pass of fixed-point block (FPB).

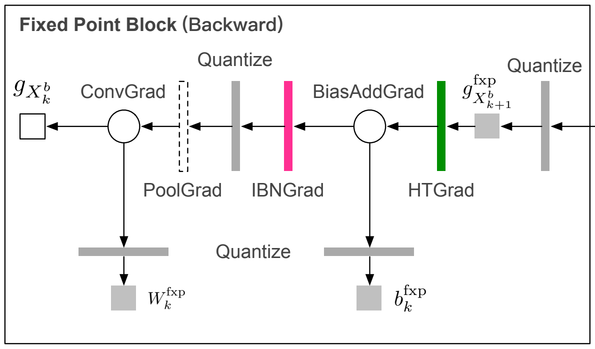


Fig. 3. Backward pass of fixed-point block in FxpNet.

- Pink layer stands for integer batch normalization (IBN) layer, which normalizes input tensor with mean and variance within a mini-batch. Differ from standard BN and shift-based BN [25] which are performed in floating-point domain, all intermediate results involved in IBN are either 32-bit integers or low resolution fixed-point values. (see Section III-A for more discussion). Algorithm 2 describes the detail of IBN.
- Gray layer refers to fixed-point quantization which coverts the output of IBN to fixed-point values (see Equation 6)
- Green layer is the non-linear activation function (Hard-Tanh shown in Equation 3).
- Besides, we disable the γ and substitute the β in standard BN with an extra bias b in BiasAdd block.

Algorithm 2 Integer Batch Normalization. Round$(\cdot)$ refers to rounding to the nearest 32-bit integer operation. FXP$(\cdot)$ denotes Equation 6 and we omit l and n for simplicity.

```
Require: Fixed-point values of x over a mini-batch $\mathbb{X}_{\text{in}} = \{x_1, \cdots, x_N\}$
Ensure: Normalized output $\mathbb{X}_{\text{out}} = \{y_1^{\text{fxp}}, \cdots, y_N^{\text{fxp}}\}$
1: $\text{sum1} \leftarrow \sum_{i=1}^{N} x_i$
2: $\text{sum2} \leftarrow \sum_{i=1}^{N} x_i^2$
3: $\text{mean} \leftarrow \text{Round}(\text{sum1}/N)$
4: $\text{var} \leftarrow \text{Round}(\text{sum2}/N) - \text{mean}^2$
5: $y_i \leftarrow (x_i - \text{mean})/\text{Round}(\sqrt{\text{var}})$
6: $y_i^{\text{fxp}} \leftarrow \text{FXP}(y_i)$
```

For the purpose of reducing computation complexity, pooling layers after FPB (if exist) will be absorbed into FPB and inserted between Conv layer and IBN layer as shown in Figure 2. Note that the last FPB does not contain Hard-Tanh and Binarize layer, i.e., the SoftmaxCrossEntropyLoss layer is computed in floating-point domain.

E. Backward Pass

In backward pass, float gradients comes from the last SoftmaxCrossEntropyLoss layer will be first converted to fixed-point values (denoted as $g^{\text{fxp}}_{X^b_{k+1}}$). The specific backward process is shown in Figure 3 from right to left.

- All white squares represent non-fixed-point values
- Green layer (HTGrad) computes the gradient of Hard-Tanh according to Equation 4.
- Gradients flow into IBN layer are with fixed-point values. Although we can compute IBN in forward pass with fixed-point format, unfortunately, non-negligible accuracy degradation occurs if we restrict the backward computation of IBN (IBNGrad) in fixed-point representation. Consequently, we choose to relax it back to floating-point domain. And we add a quantization layer right after IBNGrad in order to transform float gradients back to fixed-point format. This quantization layer will be associated with an unique scaling factor that will be adaptively updated according to Algorithm 1.
- ConvGrad layer further propagates the gradients with respect to X_k^b of last layer and W_k^b. Consider X_k^b is either 8-bits integer vector (the first layer) or binary vector, and W_k^b is binary vector, thus ConvGrad operation only contains fixed-point multiplication and accumulation.
- Last but not least, we utilize fixed-point ADAM optimization method (FxpADAM) to apply the fixed-point gradient to fixed-point primal parameters rather than standard ADAM [29]. (See Algorithm 3 for details)

Algorithm 3 Fixed-point ADAM learning rule. g_t^2 denotes element-wise square $g_t \circ g_t$. For simplicity we fix 1-β_1^t and 1-β_2^t to 1-β_1 and 1-β_2 respectively. FXP$(\cdot)$ denotes Equation 6. Default settings are $1-\beta_1 = 2^{-4}$, $1-\beta_2 = 2^{-8}$ and $\epsilon = 2^{-20}$

```
Require: Previous primal parameters $\theta_{t-1}$ with fixed-point format $l_1.n_1$, their gradients $g_t$ with fixed-point format $l_2.n_2$, learning rate $\eta_t$
Ensure: Updated fixed-point primal parameters $\theta_t$
1: $m_t^{\text{fxp}} \leftarrow \text{FXP}(\beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t, l_2, n_2)$
2: $v_t^{\text{fxp}} \leftarrow \text{FXP}(\beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot g_t^2, 2l_2, 2n_2)$
3: $u_t^{\text{fxp}} \leftarrow \text{FXP}(\sqrt{v_t^{\text{fxp}} + \epsilon}, l_2, n_2)$
4: $\theta_t \leftarrow \text{FXP}(\theta_{t-1} - \eta_t \cdot \sqrt{1-\beta_2}/(1-\beta_1) \cdot m_t^{\text{fxp}}/u_t^{\text{fxp}}, l_1, n_1)$
```

Algorithm 4 Training a FxpNet. C is the loss function within minibatch and L is the number of layers. $(\circ)$ denotes element-wise mulitiplications and $\mathrm{LRDecay}(\cdot)$ stands for how to decay the learning rate. $\mathrm{FxpADAM}(\cdot)$ means updating parameters with Fixed-point ADAM optimitzation methods. $\mathrm{ClipGrad}(\cdot)$ refers to clamp the absolute values of gradients larger than 1. $\mathrm{Binarize}(\cdot)$ denotes Equation 1 and $\mathrm{HardTanh}(\cdot)$ denotes Equation 3. $\mathrm{FXP}(\cdot)$ denotes Equation 6 and we omit l and n for simplicity.

```
Require: a minibatch of inputs $x_0^b$ and targets $y^*$, previous
    weights $W_t^{\mathrm{fxp}}$, $b_t^{\mathrm{fxp}}$ and learning rate $\eta_t$ in iteration $t$,
Ensure: updated $W_{t+1}^{\mathrm{fxp}}$, $b_{t+1}^{\mathrm{fxp}}$ and $\eta_{t+1}$
    {1.Computing the parameter gradients:}
    {1.1. Forward propagation:}
 1: for $k \leftarrow 0, L-1$ do
 2:     $W_{k,t}^b \leftarrow \mathrm{Binarize}(W_{k,t}^{\mathrm{fxp}})$
 3:     $s_k \leftarrow \mathrm{BiasAdd}(\mathrm{FXP}(\mathrm{IBN}(\mathrm{Conv}(x_k^b, W_{k,t}^b))), b_{k,t}^{\mathrm{fxp}})$
 4:     if $k < L-1$ then
 5:         $x_k \leftarrow \mathrm{HardTanh}(s_k)$
 6:         $x_{k+1}^b \leftarrow \mathrm{Binarize}(x_k)$
 7:     else
 8:         $x_{k+1} \leftarrow s_k$
 9:     end if
10: end for
    {1.2. Backward propagation:}
    Compute $g_{x_L} \leftarrow \frac{\partial C}{\partial x_L}$ given $x_L$ and $y^*$
11: for $k \leftarrow L-1, 0$ do
12:     $g_{x_{k+1}^b}^{\mathrm{fxp}} \leftarrow \mathrm{FXP}(g_{x_{k+1}^b})$
13:     if $k < L-1$ then
14:         $g_{s_k}^{\mathrm{fxp}} \leftarrow g_{x_{k+1}^b}^{\mathrm{fxp}} \circ \mathbf{1}_{|x_{k+1}| \le 1}$
15:     else
16:         $g_{s_k}^{\mathrm{fxp}} \leftarrow g_{x_{k+1}^b}^{\mathrm{fxp}}$
17:     end if
18:     $g_{x_k^b} \leftarrow \mathrm{BackwardInput}(g_{s_k}^{\mathrm{fxp}}, W_{k,t}^b)$
19:     $g_{W_{k,t}^{\mathrm{fxp}}} \leftarrow \mathrm{BackwardWeight}(g_{s_k}^{\mathrm{fxp}}, x_k^b)$
20:     $g_{b_{k,t}^{\mathrm{fxp}}} \leftarrow \mathrm{BackwardBias}(g_{s_k}^{\mathrm{fxp}})$
21: end for
    {2. Accumulating the parameters gradients}
22: for $k \leftarrow 0, L-1$ do
23:     $W_{k,t+1}^{\mathrm{fxp}} \leftarrow \mathrm{Clip}(\mathrm{FxpADAM}(W_{k,t}^{\mathrm{fxp}}, g_{W_{k,t}^{\mathrm{fxp}}}, \eta_t))$
24:     $b_{k,t+1}^{\mathrm{fxp}} \leftarrow \mathrm{FxpADAM}(b_{k,t}^{\mathrm{fxp}}, g_{b_{k,t}^{\mathrm{fxp}}}, \eta_t)$
25:     $\eta_{t+1} \leftarrow \mathrm{LRDecay}(\eta_t, t)$
26: end for
```

F. Overall Algorithm

Combining all ingredients, we describe the whole training algorithm of FxpNet in Algorithm 4.

III. Experiments

In this section, we will evaluate the important factors of FxpNet that affect the final prediction accuracy, including the batch normalization scheme, bit-width of primal parameters and bit-width of gradients. We independently investigate the effect of each component by applying them one by one on vanilla BNN respectively. Finally, we combine all these components to obtain FxpNet and explore the extremity of bit-width while maintaining prediction accuracy.

TABLE II
Model complexity

Model	#Parameter	#MACs
Model-S	0.58M	39.82M
Model-M	2.32M	156.60M
Model-L	9.29M	623.74M
BNN [22], QNN [25]	14.02M	616.97M
Miyashita et al. [19]	14.02M	616.97M

Evaluation setup. We use CIFAR-10 [30] for evaluation that is an image classification benchmark with 60K 32×32 RGB tiny images. It consists of 10 classes object, including airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. Each class has 5K training images and 1K test images. To evaluate the model fitting capability and training efficiency, we have designed three networks with different size by stacking multiple FPB, including Model-S (Small), Model-M (Medium) and Model-L (Large). The overall network structure is illustrated in Figure 4. All convolution kernels are 3×3, and the number of output channels in first convolution layer is 32, 64 and 128 for Model-S, Model-M and Model-L, respectively. Table II lists the number of parameters and the number of multiply-accumulate operations (MACs) in these three models. "$\times 2$ $(4$ or $8)$" in in C21 in Figure 4 means the number of output channel in C21 is two times (4X or 8X) as much as the number in C11 and C12. It is noteworthy that the loss layer is computed in floating-point domain in both forward pass (Figure 4a) and backward pass (Figure 4b).

In all experiments we train 37,500 iterations with mini-batch size of 200, i.e., given with 50K training images, there are totally 150 epochs and each epoch has 250 iterations. We use either FxpADAM or standard ADAM optimization method [29] with initial learning of 2^{-6} and we decrease the learning with a factor of 2^{-4} every 50 epochs. Dropout [31] has not been used.

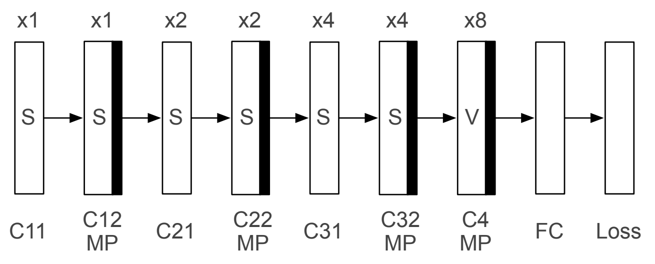
A. Effects of Batch Normalization Scheme

We evaluated the effects of different batch normalization scheme on prediction accuracy, including standard floating-point BN and different bit-width of IBN output. Here we keep primal parameters and all gradients in floating-point format, and we use standard ADAM algorithm to optimize the network. Note that we perform Algorithm 1 upon the bit-width of IBN output every 1,125 iterations (3% of total iterations), and the threshold of Algorithm 1 is set to be 0.01%.

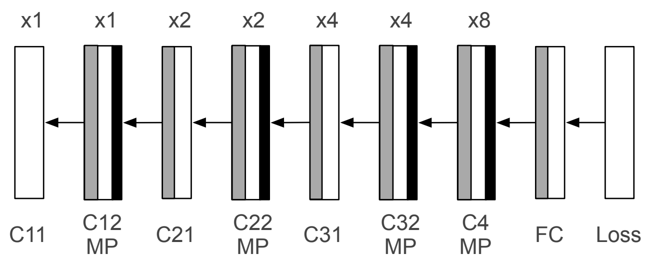
As shown in Figure 5, it is interesting to find that the network is robust to the loss of resolution within IBN output, as low as 6-bit.

B. Effects of Primal Parameters Bit-width

To evaluate the effects resulted from bit-width of primal parameters (weights and bias), we conduct experiments with



(a) Forward pass of whole network



(b) Backward pass of whole network

Fig. 4. Forward and backward pass of whole network (C-Convolution, MP-Max Pooling, FC-Fully Connected, S-Same Padding, V-Valid Padding)

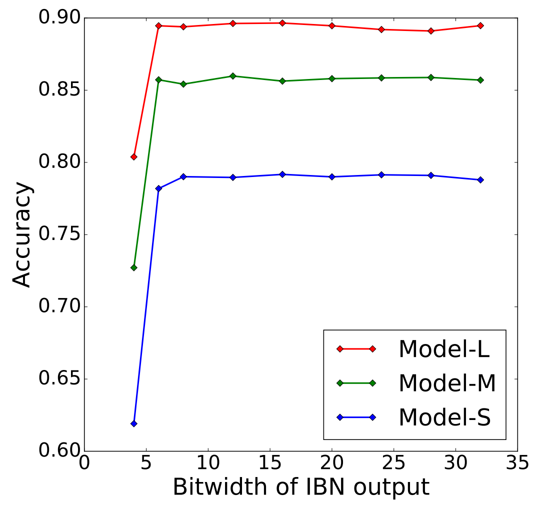


Fig. 5. Accuracy of different BN schemes. Standard BN corresponds to rightmost point (32-bit) and the remaining results denote different bit-width of IBN output (4, 6, 8, 12, 16, 20, 24, 28, 32-bits from left to right).

floating-point gradients. As show in Figure 6, 8-bits primal parameters is sufficient for maintaining performance, and bit-width lower than 8-bits will bring significant accuracy loss. We can also conclude that the adaptive update of scaling factor (Algorithm 1) can fit the value drift and keep values in normal range. On the contrary, static scaling factor imposes too strong regularization on model parameters and fails to converge when bit-width lower than 8-bit. Hyper-parameter setting is identical with Session III-A.

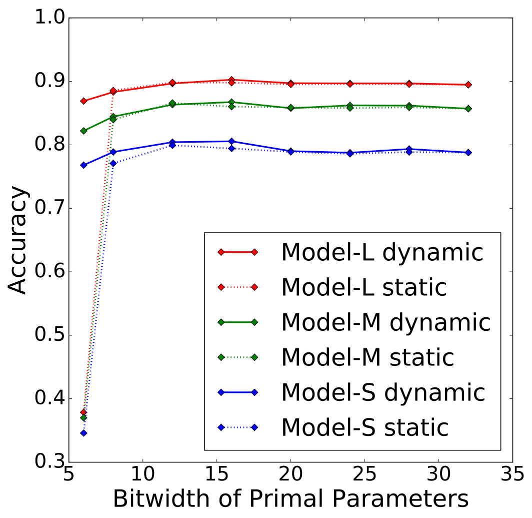


Fig. 6. Accuracy of different bit-width of primal parameters. Gradients are with floating-point values. X-axis denotes different bit-width of primal parameters (6, 8, 12, 16, 20, 24, 28, 32-bits from left to right).

C. Effects of Gradients Bit-width

It has been shown from Figure 1 that gradients are more unstable than primal parameters, which indicates that we should update the scaling factors for gradients more frequently. In our experiments, we update it every 375 iterations (1% of total iterations) and use FxpADAM method. Primal parameters are with floating-point values. The experiment result is shown in Figure 7.

We can see that when we reduce the bit-width of gradients, the prediction accuracy also degrades slowly. And it then suffers a cliff-off drop when the bit-width is lower than 12 bits for Model-S, Model-M and Model-L. We have observed similar phenomenons in Session III-A and Session III-B. All these results indicate that the whole training procedure is likely to be suddenly stucked when the bit-width of any component among IBN output, primal parameters and corresponding gradients is less than a threshold.

D. Combining All Components

We further investigate the effects by combining quantization of IBN output, primal parameters and corresponding gradients together. Table III summarizes these results and compares them with the state-of-art works. It is shown that FxpNet has managed to push the boundary of model quantization and binarization with 12-bit primal weights and 12-bits gradient, achieving comparable result with the state-of-art works on CIFAR-10 dataset. Compared with BNN [22], FxpNet consume one quarter of storage capacity and much less training complexity [2] to achieve similar prediction accuracy on CIFAR-10 dataset. Besides, FxpNet proves that linear quanitzation on gradients works well on networks with binarized weights and activations.

[2]It is hard to give a quantitative analysis on training complexity.

TABLE III
PERFORMANCE OF DIFFERENT METHODS

Method	Primal Weight [1]	Running Weight [1]	Activation [1]	Gradient [1]	#Paramter	Inference Relative Complexity [2]	Training Relative Storage [3]	CIFAR-10 Err. Rate
Courbariaux et al. [18]	12	10	10	10	2.88M	100	0.3	14.82%
Miyashita et al. [19]	32	5	4	5	14.02M	20	4.0	6.21%
BCN (det.) [20]	32	1	32	32	14.02M	32	4.0	9.90%
BNN, QNN (Theano) [22], [25]	32	1	1	32	14.02M	1	4.0	11.40%
FxpNet[4] (Model-L)	24	1	1	24	9.29M	1	2.0	10.30%
	16	1	1	16	9.29M	1	1.3	10.51%
	12	1	1	12	9.29M	1	1.0	11.48%

[1] Bit-width of these primal weight, running weight, activation and gradient
[2] Inference Relative Complexity = Running Weight × Activation
[3] Training Relative Storage = #Parameter × Primal Weight
[4] Bit-width of IBN output is 12-bit

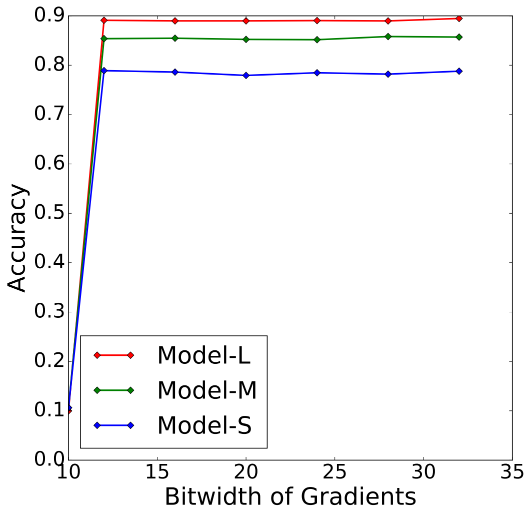


Fig. 7. Accuracy of different gradient bit-widths. Parameters are with floating-point values. X-axis denotes the bit-width (8, 10, 12, 16, 20, 24, 28, 32-bits from left to right).

IV. RELATED WORK

Model quantization was first fully investigated in CNNs during training stage in [18] that 10-bit multiplications was sufficient to train Maxout networks. Miyashita et al. [19] proposed logarithmic representation to quantize weights, activations and gradient, finding that logarithmic quantization is better than linear quantization method and 5-bits logarithmic representation reserves comparable accuracy over CIFAR-10 dataset. Vanhoucke et al. [8] stored activations and weights with 8-bit values, but they only applied to the trained neural networks without the model retraining. Courbariaux et.al. [20] first introduces model binarization and proposed BinaryConnect algorithm (BCN) to achieve nearly state-of-art performance on PI-MNIST, CIFAR-10 and SVHN datasets. Lin et al. [21] continued the work and converted the multiplications to bit-shift during back propagation. BNN [22] extends BCN and successfully binarized both weights and activations without accuracy degradation in aforementioned datasets. Almost in the same time, Rastegari et al. [23] proposed XNOR-Net and evaluated it on more challenging ImageNet dataset. Later on Zhou et al. [24] proposed DoReFa-Net and further quantized the gradients and obtained about 46.1% top-1 accuracy on ImageNet validation set with 1-bit weights, 2-bit activations and 6-bit gradients. Until recently, Hubara et al. [25] proposed QNN, which surpasses DoReFa-Net and becomes the state-of-art approach. Table IV lists the comparison among these methods. All gradients and primal parameters are floating point values in BNN [22]. QNN [25] and DoReFa-Net [24] quantized the gradients but still kept the floating-point values for primal parameters. As a comparison, both of them are quantized into fixed-point representation in FxpNet. It is noteworthy that FxpNet still keeps the binarized activations that are actually with 2-bits quantized format in QNN and DoReFa-Net.

TABLE IV
COMPARISON AMONG DIFFERENT MODEL QUANTIZATION AND BINARIZATION APPROACHES.

Method	Primal Weight [1]	Running Weight [1]	Activation [1]	Gradient [1]
Courbariaux et al. [18]	12	10	10	10
Miyashita et al. [19]	32	5	4	5
BCN [20]	32	1	32	32
BNN [22]	32	1	1	32
XNOR-Net [23]	32	1	1	32
DoReFa-Net [24]	32	1	2	6
QNN [25]	32	1	2	6

[1] Bit-width of primal weight, running weight, activation and gradient

Related works also exist for accelerating the model training and serving. Gong et al. [9] systematically analyzed different vector quantization approaches and found that k-mean clustering among weights made appropriate tradeoff between compression efficiency and prediction accuracy. Zhang et al. [11] took nonlinear units into consideration and minimized the reconstruction error with low-rank constraints. Han et al. [10] proposed a pipeline to compress trained models, including pruning small weights, quantizing the remaining ones and applying Huffman coding.

V. Conclusion and Future Work

We have introduced FxpNet, a framework to train DCNNs with low bit-width arithmetics in both forward pass and backward pass. In forward pass, FxpNet takes advantage of fixed-point format to represent primal parameters and binarizes both weights and activations. While in backward pass all gradients are represented in fixed-point format. FxpNet manages to outperform the state-of-art works, and pushes the limit of model quantization and binarization. In addition, FxpNet introduces IBN and FxpADAM methods to further reduce the required floating-point operations. More importantly, FxpNet goes a step further to provide a pure fixed-point neural network almost without expensive floating-point operations, which is of vital importance for FPGAs and dedicated ASICs implementation.

There are still opportunities on further improvements. One direction is to allocate more bitwidth to activations as QNN and DoReFa-Net, in this way, the bit-width of primal parameters and gradients would be further reduced. Another direction is to remove the remaining floating-point operations: 1). The SoftmaxCrossEntropyLoss layer. 2). The gradients within IBN. Combining them together, we believe that FxpNet could achieve good results on more challenging ImageNet dataset.

Acknowledgment

We would like to express our appreciation to Hao Liang, Wenqiang Wang and Fangzhou Liao for their continuous support and valuable feedback.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[2] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 1–9.

[3] J. Devlin, R. Zbib, Z. Huang, T. Lamar, R. M. Schwartz, and J. Makhoul, "Fast and robust neural network joint models for statistical machine translation." in *ACL (1)*. Citeseer, 2014, pp. 1370–1380.

[4] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[5] G. Hinton, N. Srivastava, and K. Swersky, "Neural networks for machine learning," *Coursera, video lectures*, vol. 264, 2012.

[6] T. N. Sainath, A.-r. Mohamed, B. Kingsbury, and B. Ramabhadran, "Deep convolutional neural networks for lvcsr," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 2013, pp. 8614–8618.

[7] T. W. D. J. W. A. Y. N. Adam Coates, Brody Huval and B. Catanzaro, "Deep learning with cots hpc systems," in *ICML*, 2013.

[8] V. Vanhoucke, A. Senior, and M. Z. Mao, "Improving the speed of neural networks on cpus," 2011.

[9] Y. Gong, L. Liu, M. Yang, and L. Bourdev, "Compressing deep convolutional networks using vector quantization," *arXiv preprint arXiv:1412.6115*, 2014.

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

[11] X. Zhang, J. Zou, X. Ming, K. He, and J. Sun, "Efficient and accurate approximations of nonlinear convolutional networks," in *CVPR*, 2015, pp. 1984–1992.

[12] S. K. Kim, L. C. McAfee, P. L. McMahon, and K. Olukotun, "A highly scalable restricted boltzmann machine fpga implementation," in *2009 International Conference on Field Programmable Logic and Applications*. IEEE, 2009, pp. 367–372.

[13] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "Neuflow: A runtime reconfigurable dataflow processor for vision," in *CVPR 2011 Workshops*. IEEE, 2011, pp. 109–116.

[14] P.-H. Pham, D. Jelaca, C. Farabet, B. Martini, Y. LeCun, and E. Culurciello, "Neuflow: Dataflow vision processing system-on-a-chip," in *2012 IEEE 55th International Midwest Symposium on Circuits and Systems (MWSCAS)*. IEEE, 2012, pp. 1044–1047.

[15] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun *et al.*, "DaDianNao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014, pp. 609–622.

[16] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," *arXiv preprint arXiv:1602.01528*, 2016.

[17] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2016, pp. 26–35.

[18] M. Courbariaux, J.-P. David, and Y. Bengio, "Training deep neural networks with low precision multiplications," *arXiv preprint arXiv:1412.7024*, 2014.

[19] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional neural networks using logarithmic data representation," *arXiv preprint arXiv:1603.01025*, 2016.

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

[21] Z. Lin, M. Courbariaux, R. Memisevic, and Y. Bengio, "Neural networks with few multiplications," *arXiv preprint arXiv:1510.03009*, 2015.

[22] M. Courbariaux and Y. Bengio, "Binarynet: Training deep neural networks with weights and activations constrained to+ 1 or-1," *arXiv preprint arXiv:1602.02830v3*, 2016.

[23] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," *arXiv preprint arXiv:1603.05279*, 2016.

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

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

[26] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[27] Y. Bengio, N. Léonard, and A. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," *arXiv preprint arXiv:1308.3432*, 2013.

[28] J. L. Holt and T. E. Baker, "Back propagation simulations using limited precision calculations," in *Neural Networks, 1991., IJCNN-91-Seattle International Joint Conference on*, vol. 2. IEEE, 1991, pp. 121–126.

[29] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[30] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," 2009.

[31] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting." *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.