# Sub-bit Neural Networks: Learning to Compress and Accelerate Binary Neural Networks

Yikai Wang[1*] Yi Yang[2] Fuchun Sun[1] Anbang Yao[2]
[1]Beijing National Research Center for Information Science and Technology (BNRist),
State Key Lab on Intelligent Technology and Systems,
Department of Computer Science and Technology, Tsinghua University [2]Intel Corporation
{wangyk17@mails., fcsun@}tsinghua.edu.cn, {yi.b.yang, anbang.yao}@intel.com

## Abstract

*In the low-bit quantization field, training Binarized Neural Networks (BNNs) is the extreme solution to ease the deployment of deep models on resource-constrained devices, having the lowest storage cost and significantly cheaper bit-wise operations compared to 32-bit floating-point counterparts. In this paper, we introduce Sub-bit Neural Networks (SNNs), a new type of binary quantization design tailored to compress and accelerate BNNs. SNNs are inspired by an empirical observation, showing that binary kernels learnt at convolutional layers of a BNN model are likely to be distributed over kernel subsets. As a result, unlike existing methods that binarize weights one by one, SNNs are trained with a kernel-aware optimization framework, which exploits binary quantization in the fine-grained convolutional kernel space. Specifically, our method includes a random sampling step generating layer-specific subsets of the kernel space, and a refinement step learning to adjust these subsets of binary kernels via optimization. Experiments on visual recognition benchmarks and the hardware deployment on FPGA validate the great potentials of SNNs. For instance, on ImageNet, SNNs of ResNet-18/ResNet-34 with 0.56-bit weights achieve 3.13/3.33× runtime speed-up and 1.8× compression over conventional BNNs with moderate drops in recognition accuracy. Promising results are also obtained when applying SNNs to binarize both weights and activations. Our code is available at https://github.com/yikaiw/SNN.*

## 1. Introduction

To enable easy deployment of Convolutional Neural Networks (CNNs) on mobile devices, many attempts are devoted to improving network efficiency, *e.g.*, reducing the storage overheads or computational costs. Existing methods on this effort include efficient architectural designs [11, 28], pruning [8, 17], network quantization [14, 19, 35, 37], knowledge distillation [9, 27], etc. Among these works, network quantization converts full-precision weights and activations to low-bit discrete values, which can be particularly hardware-friendly. Binary Neural Networks (BNNs) [1], regarded as the extreme case of quantization, resort to representing networks with 1-bit values including only $\pm 1$. In addition, when weights and activations are both binarized, addition and multiplication operations can be replaced by cheap bit-wise operations. Under this circumstance, BNNs achieve remarkable compression and acceleration performance. Given the great potentials of BNNs, the community has made numerous efforts to narrow the accuracy gap between full-precision models and BNNs, *e.g.*, introducing scaling factors to lessen the quantization error [26], or retaining the kernel information by normalizations [25]. De-

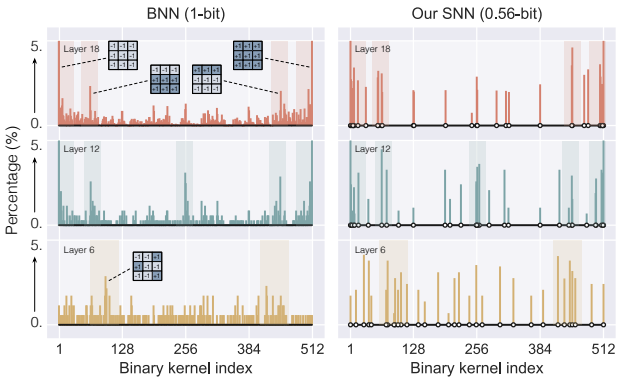


Figure 1. Frequencies of different binary kernels in each layer, collected on well-trained ResNet-20 models. **Left:** a standard BNN model, with 1-bit per weight, which converts each floating-point kernel to one of the 512 binary kernels. **Right:** the proposed SNN model, with 0.56-bit per weight. Instead of using 512 binary kernels, 0.56-bit SNN only adopts 32 binary kernels for each layer, achieving larger compression and acceleration ratios than BNN.

*This research was done when Yikai Wang was an intern at Intel Labs China, supervised by Anbang Yao who is responsible for correspondence.

spite the advances in accuracy, there remains a less-explored direction of further compressing and accelerating BNNs.

A common understanding in the network quantization field is that BNNs are the extreme case enjoying the best compression and acceleration performance, via converting 32-bit floating-point weights/activations into 1-bit ones. However, in this work we challenge this common wisdom. The main goal of this paper is to train a neural network with lower than 1-bit per weight, which is thus even more compressed than a conventional BNN model. We are motivated by an inspiring observation. Regarding a $3\times3$ full-precision convolutional kernel, its binarized counterpart belongs to the set which has $|\{\pm1\}^{3\times3}| = 512$ different binary kernels. In Figure 1 (Left), we visualize the distribution of binary kernels in a binarized ResNet-20 [7] on CIFAR10 [15]. The distribution indicates that binary kernels in a well-trained BNN model tend to be clustered to a subset at each layer, especially in deep layers where there are more filters. Such clustering distribution motivates us that potentially a more compressed model can be attained by learning to identify these important subsets which contain most binary kernels, which is illustrated in Figure 1 (Right). For a $3 \times 3$ convolutional layer with $c_{out}$ output channels, if we choose a subset which consists of $2^\tau$ binary kernels (*e.g.*, $\tau = 5$), we will approximately obtain a compression ratio of $\frac{\tau}{9}$ and an acceleration ratio of $\frac{2^\tau}{c_{out}}$ (see Sec. 4.4 and Sec. 4.5).

To determine the optimal subsets, we propose a method with two progressive steps. In the first step, we randomly sample layer-specific subsets of binary kernels. To alleviate the potential impact of randomness, in the second step, we refine the subsets by optimization. Models learnt with our full method are named Sub-bit Neural Networks (SNNs). Figure 1 (Right) illustrates the learnt subsets after training an SNN model. With subsets refinement, SNNs tend to learn similar cluster regions as BNNs. *To the best of our knowledge, this is the first method that simultaneously compresses and accelerates BNNs in a quantization pipeline.*

Detailed experiments on image classification verify that our SNNs achieve impressively large compression and acceleration ratios over BNNs while maintaining high accuracies. Besides, the hardware deployment proves that our SNNs using ResNets as test examples bring more than $3\times$ speed up over their BNN counterparts on ImageNet [4].

On the one hand, SNNs introduce a new perspective for the design space and the optimization of binary neural network quantization. Particularly, SNNs uncover and leverage the relationships of clustered binary kernel distributions in different layers of BNN models. On the other hand, SNNs open a new technical direction for compressing and accelerating BNNs while maintaining their hardware-friendly properties, creating potentially valuable opportunities for specialized hardware designs conditioned on the advantages of SNNs.

## 2. Related Work

**Binary neural networks.** Network quantization converts full-precision weights into low-bit counterparts. And in this pipeline, Binary Neural Networks (BNNs) [1] are usually treated to be extremely compressed structures. In BNNs, each weight is either $-1$ or $+1$, occupying only 1-bit which directly leads to a $32\times$ compression ratio compared with a 32-bit floating-point weight. When activations are binarized together, convolutional operations are equivalent to bit-wise operations. Although BNNs are highly efficient and hardware-friendly, one of the main drawbacks is the performance drop. Subsequent works seek various approaches to alleviate this issue. XNOR-Net [26] reduces the quantization error with channel-wise scaling factors. ABC-Net [22] adopts multiple binary bases to approximate the weights and activations to improve the performance. Bi-Real [23] recommends using short residual connections to reduce the information loss. IR-Net [25] maximizes the information entropy of binarized parameters apart from minimizing the quantization error, and benefits from the improved information preservation. RBNN [20] learns rotation matrices to reduce the angular bias of the quantization error. Although these works attempt to reduce the performance drop of BNNs, few of them consider the further compression/acceleration for a BNN model. FleXOR [16] converts every flattened binary segment to a shorter segment by encrypting. Although storing encrypted codes can reduce the model size, the complete BNN model needs to be reconstructed before inference and thus FleXOR cannot provide additional benefits in improving the inference speed.

**Other efficient designs.** There are other solutions that aim to compress or accelerate a neural network. Structural network pruning methods explicitly prune out filters [8, 17]. Knowledge distillation methods [9, 27] can guide the training of a student model with learnt knowledge, such as predictions and features, from a higher-capacity teacher. Some works design lightweight CNN backbones, such as MobileNets [11, 28] and ShuffleNets [34]. Another pipeline for network efficiency is to adjust the network depths [3], or widths [31, 32], or resolutions [30]. In our work, however, we aim to compress and accelerate a BNN model based on a new binary quantization optimization, and other efficient designs are potentially orthogonal to our method.

## 3. Preliminaries

For a CNN model, suppose the weights and activations of its $i$-th layer are denoted by $\mathbf{w}^i \in \mathbb{R}^{n^i}$ and $\boldsymbol{a}^i \in \mathbb{R}^{m^i}$, where $n^i = c^i_{out} \cdot c^i_{in} \cdot w^i \cdot h^i$ and $m^i = c^i_{out} \cdot W^i \cdot H^i$. Here, $c^i_{out}$ and $c^i_{in}$ are the numbers of output and input channels; $(w^i, h^i)$ and $(W^i, H^i)$ represent the width and height of the weights and activations, respectively. Given the standard convolutional operator $\circledast$, the computation of the $i$-th layer

is $\mathbf{a}^i = \mathbf{w}^i \circledast \mathbf{a}^{i-1}$, with the bias term omitted for simplicity.

To save the storage and computational costs, BNNs represent weights and/or activations with 1-bit values ($\pm 1$). We denote $\bar{\mathbf{w}}^i \in \{\pm 1\}^{n^i}$ and $\bar{\mathbf{a}}^i \in \{\pm 1\}^{m^i}$ as the binary vectors of weights and activations, respectively. The convolution is then reformulated as $\mathbf{a}^i = \lambda^i \cdot (\bar{\mathbf{w}}^i \circ \bar{\mathbf{a}}^{i-1})$, where $\lambda^i$ represents channel-wise scaling factors to lessen the quantization error; $\circ$ represents the bitwise operations including XNOR and Bitcount, which can largely improve the computational efficiency compared with the standard convolutional operations.

Among popular methods, the forward pass of weight binarization is simply realized by $\bar{\mathbf{w}}^i = \text{sign}(\mathbf{w}^i)$, where the element-wise function $\text{sign}(\cdot)$ converts each weight value to $-1$ or $+1$ according to its sign. As the gradient vanishes almost anywhere, the technique named Straight-Through Estimator (STE) [1] is widely used to propagate the gradient.

## 4. Sub-bit Neural Networks

We first introduce our observation which motivates us to propose a two-step approach, *i.e.*, random subsets sampling and subsets refinement. In Sec. 4.4 and Sec. 4.5, we discuss that our approach reduces the model size and increases the inference speed of conventional BNNs. In Sec. 4.6, we provide a hardware design for the practical deployment.

### 4.1. Formulation and Observation

It can be easily proved that binarizing weights with the $\text{sign}(\cdot)$ function is equivalent to clustering weights to the nearest binary vectors. In other words, there is,

$$\bar{\mathbf{w}}^i = \text{sign}(\mathbf{w}^i) = \underset{\mathbf{b} \in \{\pm 1\}^{n^i}}{\arg\min} \|\mathbf{b} - \mathbf{w}^i\|_2^2. \tag{1}$$

In addition, considering the independence of channels, $\bar{\mathbf{w}}^i$ can be further derived as the channel-wise concatenation of each binary kernel $\bar{\mathbf{w}}_c^i = \arg\min_{\mathbf{k} \in \mathbb{K}} \|\mathbf{k} - \mathbf{w}_c^i\|_2^2$, where $\mathbb{K} = \{\pm 1\}^{w^i \cdot h^i}$ and $c = 1, 2, \cdots, c_{out}^i \cdot c_{in}^i$.

Given the definition of $\mathbb{K}$, there are $|\mathbb{K}| = 2^{w^i \cdot h^i}$ elements in total, where each element is a binary kernel. We mainly focus on $3 \times 3$ convolutional kernels as they usually occupy major parameters and computations in modern neural networks, hence here $|\mathbb{K}| = 2^9 = 512$. In BNNs, every single weight occupies 1-bit, and thus we need 9-bit to represent a binary kernel. We assign indices (from 1 to 512) for these binary kernels, and an example of the assigning process is depicted in Figure 2. Specifically, we represent the flattened binary kernel with a binary sequence and then convert it to a decimal number. For example, the binary kernel with all $-1$ values is indexed as 1, and the one with all $+1$ values is indexed as 512.

Here a question arises: how does the training process of a BNN model tend to distribute the 512 binary kernels? To figure out this, Figure 3 illustrates the distribution of different binary kernels in a standard BNN model after training, and the total percentage of the top-k most frequent binary kernels in the corresponding layers. Surprisingly, the training process binarizes floating-point kernels to certain clusters/subsets of binary kernels, and given the regions highlighted with colors, the subsets are noticeably different in different layers. The top-k total percentage indicates that, *e.g.*, by only extracting the most frequent 64 (from the total 512) binary kernels, the total percentage can surpass 70% in these layers. Besides, in the deep layer with more filters, the clustering property tends to be more obvious. For example, in the 25-th layer, 96.3% floating-point kernels are binarized to 64 binary kernels, and about 60% floating-point kernels are binarized to only 8 binary kernels.

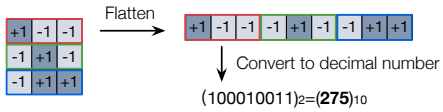


Figure 2. Illustration of assigning an index to a binary kernel. The binary kernel is flattened to a 1-dimension vector and then is represented with a binary sequence. The assigned index is the corresponding decimal number converted from the binary sequence.

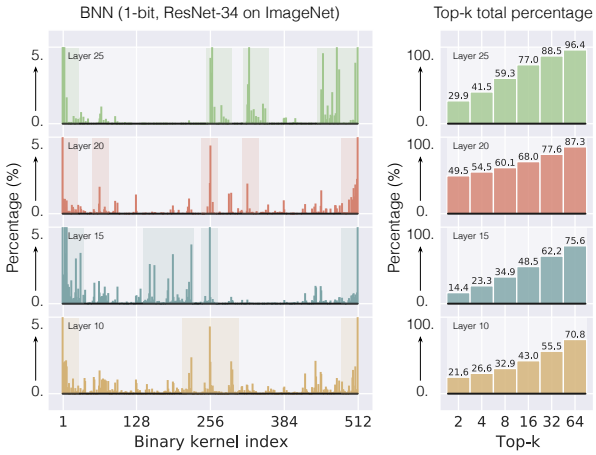


Figure 3. **Left:** Distributions of binary kernels for a standard BNN model which is well-trained on ImageNet using ResNet-34. We do NOT apply any non-linear transformation on the distributions, and only clip the values which are larger than 5%. Regions with dense clusters are highlighted with colors. **Right:** Total percentage of top-k most frequent binary kernels.

These findings suggest that a BNN model can be potentially compressed by sampling *layer-specific* subsets of binary kernels and still maintains high performance. Therefore, we design the following two-step method.

### 4.2. Random Kernel Subsets Sampling

Here we propose a simple yet effective method to generate subsets with binary kernels, which can further reduce the

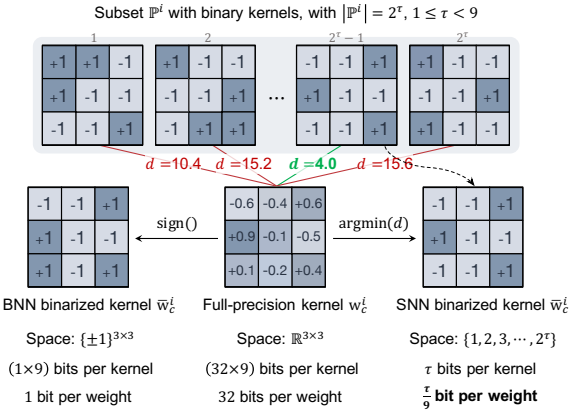


Figure 4. Binarization comparison of a standard BNN model and the proposed method (SNN). Here, $d$ denotes $\|\mathbf{k} - \mathbf{w}_c^i\|_2^2$, as defined in Eq. (2), where $\mathbf{k} \in \mathbb{P}^i$. In our SNN, each kernel corresponds to an index in $\{1, 2, 3, \cdots, 2^\tau\}$, which has the same size with $\{\pm1\}^\tau$, hence each kernel occupies $\tau$ bits. In average, each weight of SNN only needs $\frac{\tau}{9}$ bit.

model size and improve inference speed over a BNN model. In contrast with existing binarization methods which convert weights to the full binary set $\mathbb{K}$ (defined in Sec. 4.1) without considering kernel structures, we cluster weights in different layers to the different subsets of $\mathbb{K}$, and thus we are able to train a model with lower than 1 bit per weight. Suppose we want to use $\tau$-bit to represent each kernel (where $1 \le \tau < 9$), the size of each subset should be $2^\tau$. We set $\tau$ as an integer in our method. Note that we adopt layer-specific subsets instead of sharing the subset throughout the network, in order to maintain the diversity of binary kernels.

Formally, given the $i$-th layer of a CNN model, we *randomly* sample a subset $\mathbb{P}^i \subset \mathbb{K}$, which satisfies $|\mathbb{P}^i| = 2^\tau$. We binarize every $\mathbf{w}_c^i$ by clustering it to the nearest binary kernel in $\mathbb{P}^i$, and use the standard STE [1] method to update $\mathbf{w}_c^i$. Under this circumstance, denoting $\mathcal{L}$ as the loss function, the forward and backward passes are computed as,

$$\text{Forward}: \bar{\mathbf{w}}_c^i = \arg\min_{\mathbf{k}\in\mathbb{P}^i} \|\mathbf{k} - \mathbf{w}_c^i\|_2^2, \tag{2}$$

$$\text{Backward}: \frac{\partial \mathcal{L}}{\partial \mathbf{w}_c^i} \approx \begin{cases} \frac{\partial \mathcal{L}}{\partial \bar{\mathbf{w}}_c^i}, & \text{if } \mathbf{w}_c^i \in (-1, 1)\,, \\ 0, & \text{otherwise.} \end{cases} \tag{3}$$

Figure 4 illustrates why our method can achieve larger compression ratios than a standard BNN model considering the most widely used $3 \times 3$ convolutional kernel. Details of the compression are further provided in Sec. 4.4.

This random sampling approach is dubbed the Vanilla version of Sub-bit Neural Networks (Vanilla-SNNs). For this version, the sampling is performed during initialization, and the sampled subsets are fixed throughout the training. Experimental results verify that this method can already demonstrate a good balance between performance and efficiency. In addition, sampling layer-specific subsets would be less vulnerable to randomness, compared with sharing a global random subset, as will be discussed in Sec. 5.2.

### 4.3. Kernel Subsets Refinement by Optimization

Since random sampling cannot guarantee the optimal binary kernels, there is still some room for improvement. To this end, we refine the sampled subsets by optimization.

We represent the subset $\mathbb{P}^i$ with the tensor format $\mathbf{p}^i \in \mathbb{R}^{w^i \cdot h^i \cdot |\mathbb{P}^i|}$, which is the concatenation of the kernels. Each single weight in $\mathbf{p}^i$ is initialized to $-1.0$ or $+1.0$ according to $\mathbb{P}^i$, and will be updated during training. Since $\mathbf{p}^i$ is no longer limited to $\{\pm1\}$ during optimization, we apply an element-wise function $\text{sign}(\cdot)$ on it. Note that the absolute magnitude of each weight of $\mathbf{p}^i$ does not affect the forward calculation unless its sign changes. When the input of the $\text{sign}(\cdot)$ function oscillates around 0, the output value rapidly changes between $-1$ and $+1$, which may lead to an unstable training. To tactile this issue, we further introduce a binary tensor $\mathbf{m}^i \in \{\pm1\}^{w^i \cdot h^i \cdot |\mathbb{P}^i|}$ to "memorize" the sign of each weight in $\mathbf{p}^i$. Before training, $\mathbf{m}^i$ is initialized to $\text{sign}(\mathbf{p}^i)$, and during training, $\mathbf{m}^i$ is updated as below,

$$\mathbf{m}^i = \mathbf{m}^i \odot \mathbb{I}_{|\mathbf{p}^i| \le \theta} + \text{sign}(\mathbf{p}^i) \odot \mathbb{I}_{|\mathbf{p}^i| > \theta}, \tag{4}$$

where $\theta$ is the threshold which is a positive and small hyper-parameter for alleviating the rapid sign change and improving the learning stability; $\mathbb{I}$ is the indicator function which obtains a mask tensor belonging to $\{0, 1\}^{w^i \cdot h^i \cdot |\mathbb{P}^i|}$; $\odot$ represents the element-wise multiplication. To summarize, $\mathbf{m}^i$ follows the sign of $\mathbf{p}^i$ only if the magnitude of $\mathbf{p}^i$ is larger than $\theta$; otherwise, $\mathbf{m}^i$ remains unchanged. $\theta$ is set to $10^{-3}$ in our experiments.

We follow the same expression with Eq. (3) to update $\mathbf{w}_c^i$. Similarly, applying the $\text{sign}(\cdot)$ function on $\mathbf{p}^i$ in Eq. (4) prevents the gradient propagation, and thus we again adopt the STE technique to update $\mathbf{p}^i$ based on the gradient *w.r.t.* $\mathbf{m}^i$. The forward pass of binarizing $\mathbf{w}_c^i$ and the backward pass for updating both $\mathbf{w}_c^i$ and $\mathbf{p}^i$ are computed as,

$$\text{Forward}: \bar{\mathbf{w}}_c^i = \arg\min_{\mathbf{m}_j^i} \|\mathbf{m}_j^i - \mathbf{w}_c^i\|_2^2, \tag{5}$$

$$\text{Backward}: \text{Eq. (3)}, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{p}^i} \approx \frac{\partial \mathcal{L}}{\partial \mathbf{m}^i}, \tag{6}$$

where $j = 1, 2, \cdots, |\mathbb{P}^i|$; $\mathbf{m}_j^i \in \{\pm1\}^{w^i \cdot h^i}$ is the $j$-th binary kernel of $\mathbf{m}^i$. Since $\mathbf{m}^i$ directly affects $\bar{\mathbf{w}}_c^i$ in Eq. (5), the gradient *w.r.t.* $\mathbf{m}^i$ could be automatically obtained by accumulating the gradient *w.r.t.* $\bar{\mathbf{w}}_c^i$, with $c$ ranging from 1 to $c_{out}^i \cdot c_{in}^i$. We empirically find that using the same learning rate for updating $\mathbf{w}$ and $\mathbf{p}$ already yields good results.

With the help of kernel subsets refinement, new binary kernels can be obtained which may be potentially more informative. Given that kernels $\mathbf{p}_1^i, \mathbf{p}_2^i, \cdots, \mathbf{p}_{|\mathbb{P}^i|}^i$ are independently optimized during training, there may appear two

**Algorithm 1** Training: forward and backward processes of Sub-bit Neural Networks (SNNs).

**Require**: input data; full-precision weights $\mathbf{w}$; threshold $\theta$; learning rate $\eta$.
**for** layer $i = 1 \rightarrow L$ **do**
  Randomly sample a layer-specific subset $\mathbb{P}^i \subset \mathbb{K}$ and there is $|\mathbb{P}^i| = 2^\tau$; Represent $\mathbb{P}^i$ as $\mathbf{p}^i \in \mathbb{R}^{w^i \cdot h^i \cdot |\mathbb{P}^i|}$.
  Initialize $\mathbf{m}^i = \text{sign}(\mathbf{p}^i)$.
**for** step $t = 1 \rightarrow T$ **do**
  **Forward propagation:**
    **for** layer $i = 1 \rightarrow L$ **do**
      Compute $\mathbf{m}^i = \mathbf{m}^i \odot \mathbb{I}_{|\mathbf{p}^i| \leq \theta} + \text{sign}(\mathbf{p}^i) \odot \mathbb{I}_{|\mathbf{p}^i| > \theta}$.
      **for** channel $c = 1 \rightarrow c^i_{out} \cdot c^i_{in}$ **do**
        Compute $\bar{\mathbf{w}}^i_c = \arg\min_{\mathbf{m}^i_j} \|\mathbf{m}^i_j - \mathbf{w}^i_c\|^2_2$.
        Compute $\mathbf{a}^i_c = \lambda^i_c \cdot (\bar{\mathbf{w}}^i_c \circ \text{sign}(\mathbf{a}^{i-1}_c))$ in Sec. 3.
  **Back propagation:**
    **for** layer $i = L \rightarrow 1$ **do**
      **for** channel $c = 1 \rightarrow c^i_{out} \cdot c^i_{in}$ **do**
        Compute $\frac{\partial \mathcal{L}}{\partial \mathbf{w}^i_c}$ via Eq. (3).
      Compute $\frac{\partial \mathcal{L}}{\partial \mathbf{p}^i} \approx \frac{\partial \mathcal{L}}{\partial \mathbf{m}^i}$.
  **Parameters Update:**
    Update $\mathbf{w} = \mathbf{w} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{w}}$, $\mathbf{p} = \mathbf{p} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{p}}$.
  Check repetitive binary kernels in $\mathbf{m}^i$ and substitute these corresponding kernels in $\mathbf{p}^i$ with random new kernels.

repetitive binary kernels in $\mathbf{m}^i$ after one certain update, *i.e.*, $\exists\, j_1 \neq j_2$ s.t. $\mathbf{m}^i_{j_1} = \mathbf{m}^i_{j_2}$. This issue reduces the number of selectable binary kernels in binarization. In other words, the actual bit-width decreases and may impact expected accuracy. To address the problem, we perform a glance of $\mathbf{m}^i$ in every training iteration to check if there are repetitive binary kernels. If repetitive binary kernels are detected in $\mathbf{m}^i$, we remove the corresponding kernels in $\mathbf{p}^i$ and randomly sample new ones (the same number with the removed ones) from $\mathbb{K}$ to complement $\mathbf{p}^i$. By this approach, the aimed bit-width can be reached during training. These new kernels would together be updated in the later training process.

The learnable approach is dubbed the full version of Sub-bit Neural Networks (SNNs). Forward and backward processes of training SNNs are summarized in Algorithm 1.

### 4.4. Compression

As preliminarily showed in Figure 4, in standard BNNs each weight is represented by 1 bit, and thus each $3 \times 3$ binary kernel occupies 9 bits. To compress BNNs, we adopt an index for each binary kernel, and such index points to one of the $2^\tau$ elements in the subset. Thus the index belongs to $\{1, 2, 3, \cdots, 2^\tau\}$, or equivalently can be represented by a $\tau$-bit $\pm 1$ vector. Given $1 \leq \tau < 9$, every single weight occupies $\frac{\tau}{9}$ bit in average. Here, we further provide Figure 5 for a better understanding of the process to store the weights and the subset *in a layer*. According to Figure 5, in SNN, denoting the input and output channel numbers of

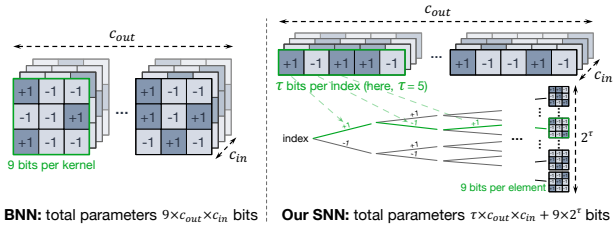


Figure 5. Explanation of storing weights and the subset in a layer.

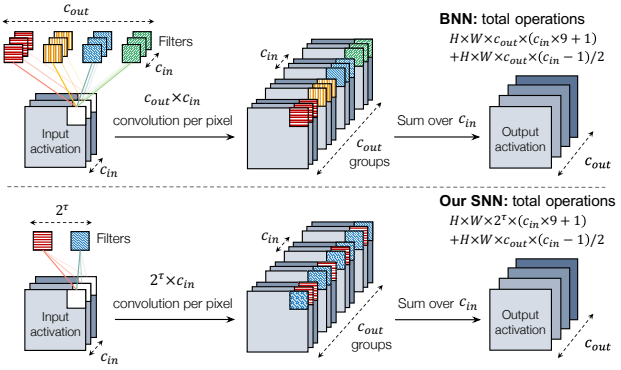


Figure 6. Comparison of convolution procedures in a standard BNN and our SNN. Instead of computing activations for all the $c_{out}$ filters, our method shares the computations, which leads to a larger acceleration as $2^\tau \ll c_{out}$ in most popular backbones such as ResNets. The superscript $i$ is omitted here for simplicity.

the layer as $c_{in}$ and $c_{out}$ respectively, weights of this layer occupy $\tau \times c_{in} \times c_{out}$ bits in total. Unlike BNN, we need to additionally store a subset $\mathbb{P}^i$ with $2^\tau$ elements (*i.e.*, $3 \times 3$ binary kernels) in SNN, and such subset occupies $9 \times 2^\tau$ bits which can be ignored as it is shared per layer. For example, in a 0.56-bit ($\tau = 5$) ResNet-18 or ResNet-34, there are 32 binary kernels in $\mathbb{P}^i$, which only occupies $0.01\% \sim 0.7\%$ parameters depending on the channel numbers of the layer. As weights of a layer occupy $9 \times c_{in} \times c_{out}$ bits in a standard BNN, our SNN achieves a compression ratio of $\frac{\tau}{9}$.

### 4.5. Acceleration

Apart from compressing the model size, SNNs achieve higher acceleration ratios than standard BNNs. As the number of different binary kernels is limited to $2^\tau$, convolutional operations can be largely shared. Figure 6 depicts the convolution procedures of a standard BNN and our method. The basic difference lies in the convolutional operations, where BNN needs $H \times W \times c^i_{out} \times (c^i_{in} \times 3 \times 3 + 1)$ bit-wise operations for a $3 \times 3$ convolutional layer, yet SNN reduces this number with a ratio $\frac{2^\tau}{c^i_{out}}$. Both BNN and SNN then perform channel-wise addition along $c^i_{in}$ for $c^i_{out}$ times, and this step occupies much lower computational burdens compared with the previous step (convolution). Note that SNN needs additional indexing operations to reuse the calculated activations, and such indexing costs can be largely saved by designing the data flow as will be introduced in Sec. 4.6.

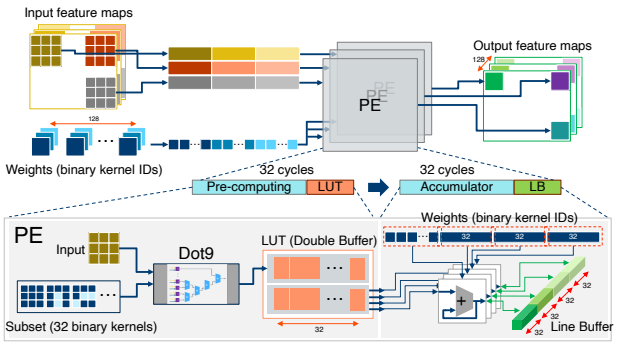


Figure 7. Hardware design for the deployment with a 0.56-bit SNN for example, where the subset of each layer contains 32 binary kernels. The pre-computing unit needs 32 cycles to traverse all binary kernels. The accumulator unit has 4 parallel accumulators, which accumulate 4 groups of data in one cycle and write the results back to the corresponding positions of LB simultaneously. To match the number of cycles of the pipeline stage, the width of LB could be set to 128, so that it also needs 32 cycles to complete the process.

### 4.6. Practical Deployment

We have discussed that our SNNs are theoretically more efficient than standard BNNs. To verify the runtime acceleration in practical deployment, we present a hardware accelerator architecture for SNNs in Figure 7. Specially, we divide the Processing Engine (PE) into two pipeline units. First, the pre-computing unit fetches binary kernels serially by kernel IDs and convolutes them with input activations. The result is immediately stored in the Lookup Table (LUT) within the current cycle. Second, the accumulator unit reads kernel IDs and fetches the data from LUT with kernel IDs as addresses. The data from LUT will be accumulated with the data stored in a Line Buffer (LB), and then will be written back to LB. Once all input activations along input channels are processed, data in LB will be stored into a dedicated output buffer and cleared for the next round. Such architecture design is particularly FPGA friendly: as FPGA itself is a LUT-based structure, the cost of implementing the distributed lookup tables is very small. Both units support parallel computing in different PEs. Two units follow a data pipeline and are parallelly computed within the same cycles.

Theoretically, each binary kernel in the subset $\mathbb{P}^i$ is pre-computed per channel and per pixel of the $c_{in} \times H^i \times W^i$ input activation, and thus there are $2^\tau \times c_{in} \times H^i \times W^i$ pre-computed results need storing in LUT. However, by well designing the computation flow, we can largely reduce the LUT size and thus decrease the lookup time costs. As depicted in Figure 7, the $c_{in} \times H^i \times W^i$ input activation is split into $1 \times 3 \times 3$ activation slices which are overlapped (given the convolutional stride 1) and are computed in parallel. At each time, all pre-computed results in LUT only correspond to one same activation slice. Therefore the practical LUT size is no longer $2^\tau \times c_{in} \times H^i \times W^i$ but is reduced to only $2^\tau$. Such a small LUT size leads to very low latency for the lookup process, *e.g.*, less than 0.5ns for our 0.56-bit model which can be easily implemented in the current clock cycle. Thus the lookup process is well integrated into the deployment and may not affect the number of cycles.

## 5. Experiments

We conduct comprehensive experiments to evaluate the effectiveness and efficiency of our method. All our experiments are implemented with the PyTorch [24] library.

### 5.1. Basic Results

We choose CIFAR10 [15] and ImageNet [4] datasets for experiments. Following IR-Net [25], the Bi-Real [23] technique is adopted when activations are also binarized. Following the common paradigm [23, 25, 26], we keep the first and last layers to be full-precision and binarize the others.

**CIFAR10.** Following popular settings in state-of-the-art BNNs, we adopt ResNet-20 [7], ResNet-18, and VGG-small [29] to evaluate our method on CIFAR10. Results including our Vanilla-SNNs, SNNs, and other existing methods are provided in Table 1. For our method, we use three typical bit-widths 0.67-bit, 0.56-bit, and 0.44-bit, which are obtained by setting $\tau$ to 6, 5, and 4 respectively. Bit-wise parameters (Params) and bit-wise operations (Bit-OPs) are compared to highlight the superiority of our method[1]. Our method presents increasing compression and acceleration rates as the bit-width decreases, with slight and stable performance variations. The acceleration rate is highly correlated to the number of channels in the architecture. For example, by compressing BNN to 0.56-bit in VGG-small, SNN has almost no accuracy drop yet reduces the model size and Bit-OPs by $1.80\times$ and $5.34\times$ respectively.

**ImageNet.** We also conduct experiments on ImageNet, one of the most challenging benchmarks for visual recognition. In Table 2, we report results based on ResNet-18 and ResNet-34 with bit-width settings 0.67-bit, 0.56-bit, and 0.44-bit to evaluate our method. Compared with state-of-the-art BNN methods, in general, SNNs with 0.67-bit, 0.56-bit, and 0.44-bit have around $1.5\%$, $3.0\%$, $4.5\%$ accuracy drops respectively, yet saving the parameters by $1.50\times$, $1.80\times$, $2.25\times$, and saving Bit-OPs by $1.90\sim2.08\times$, $3.35 \sim 3.65\times$, $5.65 \sim 6.07\times$. By reducing the bit-width from 1-bit to 0.44-bit, the number of selectable binary kernels for each layer is significantly reduced from 512 to 16. Considering this, the accuracy drop is moderate. As introduced in Sec. 2, FleXOR [16] reduces parameters by encryption and relies on decryption before inference, leading no further benefits to the running speed. Table 2 shows that our SNNs outperform FleXORs in two aspects: 1) SNNs

[1]For comparison, in Table 1 and Table 2, parameters and bit-wise operations for all methods are collected excluding the first and last layers as they are not binarized following the common paradigm.

Table 1. Image classification results on CIFAR10 based on the single-crop testing with $32 \times 32$ crop size. We follow the training details and techniques in IR-Net [25], which could be the reference for our 1-bit baselines. Each result of our method is the average of three runs. Numbers highlighted in green are reduction ratios over BNN counterparts. * indicates our implemented results.

| Method | Bit-width (W/A) | #Params (Mbit) | Bit-OPs (G) | Top-1 Acc. (%) |
|---|---|---|---|---|
| ResNet-18 | | | | |
| Full precision | 32/32 | 351.54 | 35.03 | 93.0 |
| RAD [5] | 1/1 | 10.99 | 0.547 | 90.5 |
| IR-Net [25] | 1/1 | 10.99 | 0.547 | 91.5 |
| Vanilla-SNN \| SNN | 0.67/1 | 7.324 (1.5×) | 0.289 (1.9×) | 89.7 \| 91.0 |
| Vanilla-SNN \| SNN | 0.56/1 | 6.103 (1.8×) | 0.164 (3.3×) | 89.3 \| 90.6 |
| Vanilla-SNN \| SNN | 0.44/1 | 4.882 (2.3×) | 0.097 (5.6×) | 88.3 \| 90.1 |
| IR-Net* [25] | 1/32 | 10.99 | 17.52 | 92.9 |
| Vanilla-SNN \| SNN | 0.67/32 | 7.324 (1.5×) | 9.236 (1.9×) | 92.4 \| 92.7 |
| Vanilla-SNN \| SNN | 0.56/32 | 6.103 (1.8×) | 5.239 (3.3×) | 92.0 \| 92.3 |
| Vanilla-SNN \| SNN | 0.44/32 | 4.882 (2.3×) | 3.106 (5.6×) | 91.6 \| 91.9 |
| ResNet-20 | | | | |
| Full precision | 32/32 | 8.54 | 2.567 | 91.7 |
| DoReFa [36] | 1/1 | 0.267 | 0.040 | 79.3 |
| IR-Net [25] | 1/1 | 0.267 | 0.040 | 86.5 |
| Vanilla-SNN \| SNN | 0.67/1 | 0.178 (1.5×) | 0.040 | 83.9 \| 85.1 |
| Vanilla-SNN \| SNN | 0.56/1 | 0.148 (1.8×) | 0.034 (1.2×) | 82.7 \| 84.0 |
| Vanilla-SNN \| SNN | 0.44/1 | 0.119 (2.3×) | 0.025 (1.6×) | 82.0 \| 82.5 |
| DoReFa [36] | 1/32 | 0.267 | 1.283 | 90.0 |
| LQ-Net [33] | 1/32 | 0.267 | 1.283 | 90.1 |
| IR-Net [25] | 1/32 | 0.267 | 1.283 | 90.8 |
| Vanilla-SNN \| SNN | 0.67/32 | 0.178 (1.5×) | 1.283 | 88.7 \| 90.0 |
| Vanilla-SNN \| SNN | 0.56/32 | 0.148 (1.8×) | 1.099 (1.2×) | 87.8 \| 88.9 |
| Vanilla-SNN \| SNN | 0.44/32 | 0.119 (2.3×) | 0.822 (1.6×) | 87.1 \| 87.6 |
| VGG-small | | | | |
| Full precision | 32/32 | 146.24 | 38.66 | 92.5 |
| LAB [10] | 1/1 | 4.571 | 0.603 | 87.7 |
| XNOR [26] | 1/1 | 4.571 | 0.603 | 89.8 |
| BNN [13] | 1/1 | 4.571 | 0.603 | 89.9 |
| RAD [5] | 1/1 | 4.571 | 0.603 | 90.0 |
| IR-Net* [25] | 1/1 | 4.571 | 0.603 | 91.3 |
| Vanilla-SNN \| SNN | 0.67/1 | 3.047 (1.5×) | 0.194 (3.1×) | 90.3 \| 91.0 |
| Vanilla-SNN \| SNN | 0.56/1 | 2.540 (1.8×) | 0.113 (5.3×) | 89.8 \| 90.6 |
| Vanilla-SNN \| SNN | 0.44/1 | 2.032 (2.3×) | 0.074 (8.1×) | 89.2 \| 90.0 |
| IR-Net* [25] | 1/32 | 4.571 | 19.30 | 92.5 |
| Vanilla-SNN \| SNN | 0.67/32 | 3.047 (1.5×) | 6.208 (3.1×) | 92.0 \| 92.4 |
| Vanilla-SNN \| SNN | 0.56/32 | 2.540 (1.8×) | 3.616 (5.3×) | 91.7 \| 92.1 |
| Vanilla-SNN \| SNN | 0.44/32 | 2.032 (2.3×) | 2.368 (8.1×) | 91.3 \| 91.9 |

achieve better accuracies with lower bit-width settings, *e.g.*, SNNs with $0.56/32$-bit achieving $63.4\%$ vs. FleXORs with $0.6/32$-bit achieving $62.0\%$; 2) SNNs largely reduce inference Bit-OPs. In addition, SNNs obtain better performance than the corresponding Vanilla-SNNs, *e.g.*, from $62.8\%$ to $63.4\%$ for $0.56/32$-bit ResNet-18, which verifies the advantage of subsets refinement proposed in Sec. 4.3.

## 5.2. Ablation Studies

**Layer-specific or layer-shared kernel subsets.** The observation in Sec. 4.1 suggests that binary kernels tend to cluster to different subsets in different layers, and thus the random sampling process in Sec. 4 privatizes subsets for different layers. Using layer-specific subsets also alleviates the instability brought by randomness. Comparison results in Table 3 indicate that: 1) Settings with layer-shared subsets present unstable results especially at a low bit-width such as $0.11$ or $0.33$, while using layer-specific subsets is more effective and stable. 2) Kernel subsets refinement, proposed in Sec. 4.3, can improve the results for both the layer-specific case as well as the layer-shared case.

Table 2. Image classification results on ImageNet based on the single-crop testing with $224 \times 224$ crop size. We follow the training details and techniques in IR-Net [25], which could be the reference for our 1-bit baselines. Numbers highlighted in green are reduction ratios over BNN counterparts.

| Method | Bit-width (W/A) | #Params (Mbit) | Bit-OPs (G) | Top-1 Acc. (%) |
|---|---|---|---|---|
| ResNet-18 | | | | |
| Full precision | 32/32 | 351.54 | 107.28 | 69.6 |
| XNOR [26] | 1/1 | 10.99 | 1.677 | 51.2 |
| BNN+ [13] | 1/1 | 10.99 | 1.677 | 53.0 |
| Bi-Real [23] | 1/1 | 10.99 | 1.677 | 56.4 |
| XNOR++ [2] | 1/1 | 10.99 | 1.677 | 57.1 |
| IR-Net [25] | 1/1 | 10.99 | 1.677 | 58.1 |
| Vanilla-SNN \| SNN | 0.67/1 | 7.324 (1.5×) | 0.883 (1.9×) | 55.7 \| 56.3 |
| Vanilla-SNN \| SNN | 0.56/1 | 6.103 (1.8×) | 0.501 (3.3×) | 54.6 \| 55.1 |
| Vanilla-SNN \| SNN | 0.44/1 | 4.882 (2.3×) | 0.297 (5.6×) | 52.5 \| 53.0 |
| BWN [26] | 1/32 | 10.99 | 53.64 | 60.8 |
| HWGQ [18] | 1/32 | 10.99 | 53.64 | 61.3 |
| BWHN [12] | 1/32 | 10.99 | 53.64 | 64.3 |
| IR-Net [25] | 1/32 | 10.99 | 53.64 | 66.5 |
| FleXOR [16] | 0.80/32 | 8.788 (1.3×) | 53.64 | 63.8 |
| FleXOR [16] | 0.60/32 | 6.591 (1.7×) | 53.64 | 62.0 |
| Vanilla-SNN \| SNN | 0.67/32 | 7.324 (1.5×) | 28.26 (1.9×) | 63.7 \| 64.7 |
| Vanilla-SNN \| SNN | 0.56/32 | 6.103 (1.8×) | 16.03 (3.3×) | 62.8 \| 63.4 |
| Vanilla-SNN \| SNN | 0.44/32 | 4.882 (2.3×) | 9.504 (5.6×) | 60.1 \| 60.9 |
| ResNet-34 | | | | |
| Full precision | 32/32 | 674.88 | 225.66 | 73.3 |
| Bi-Real [23] | 1/1 | 21.09 | 3.526 | 62.2 |
| IR-Net [25] | 1/1 | 21.09 | 3.526 | 62.9 |
| Vanilla-SNN \| SNN | 0.67/1 | 14.06 (1.5×) | 1.696 (2.1×) | 60.6 \| 61.4 |
| Vanilla-SNN \| SNN | 0.56/1 | 11.71 (1.8×) | 0.965 (3.7×) | 59.5 \| 60.2 |
| Vanilla-SNN \| SNN | 0.44/1 | 9.372 (2.3×) | 0.581 (6.1×) | 58.1 \| 58.6 |
| IR-Net [25] | 1/32 | 21.09 | 112.83 | 70.4 |
| Vanilla-SNN \| SNN | 0.67/32 | 14.06 (1.5×) | 54.27 (2.1×) | 67.5 \| 68.0 |
| Vanilla-SNN \| SNN | 0.56/32 | 11.71 (1.8×) | 30.88 (3.7×) | 66.3 \| 66.9 |
| Vanilla-SNN \| SNN | 0.44/32 | 9.372 (2.3×) | 18.59 (6.1×) | 64.5 \| 65.1 |

**Learning procedure of refinement.** Although experiments have verified the validity of subsets refinement, we would like to figure out why it works. Figure 8 illustrates the changes of subsets during training (with subsets refinement). We observe that binary kernels seem to be irregular at the first epoch since they are initially randomly sampled. With the help of subsets refinement, distributions of binary kernels tend to be regular and symmetric in later epochs.

**Analysis of hyper-parameter $\theta$.** In Eq. (4), we introduce $\theta$ as a threshold to alleviate the rapid sign change when $\mathbf{p}^i$ is around 0 during training. To verify whether $\theta$ works, in Figure 9, we provide the accuracy curves with different $\theta$ settings. When $\theta = 0$, the validation curve oscillates during $100 \sim 200$ epochs, and the final prediction is impacted by the instability. We observe that setting $\theta$ to $10^{-3}$ or $10^{-2}$ leads to similar results, and $\theta = 10^{-3}$ is slightly better.

Our supplementary materials include training details, extension to $1 \times 1$ convolutions, evaluation on object detection with PASCAL VOC [6] and MS-COCO [21] datasets, etc.

Table 3. Comparison of using layer-specific subsets and layer-shared subsets. Experiments are performed on CIFAR10 with ResNet-20 and each setting is performed three times with different random seeds. Numbers highlighted in green are gains of the layer-specific settings over their layer-shared counterparts.

| Method | Subsets strategy | Bit-width (W/A) | | | |
|---|---|---|---|---|---|
| | | 0.11/32 | 0.33/32 | 0.56/32 | 0.78/32 |
| Vanilla-SNN | Layer-shared | 70.2±6.5 | 79.4±3.7 | 82.6±2.2 | 88.2±1.0 |
| SNN | Layer-shared | 73.5±3.3 | 82.5±2.6 | 86.7±1.3 | 90.0±0.3 |
| Vanilla-SNN | Layer-specific | 80.1±2.3 (+9.9) | 85.5±2.1 (+6.1) | 87.8±1.6 (+5.2) | 89.9±0.6 (+1.7) |
| SNN | Layer-specific | 81.6±1.7 (+8.1) | 86.8±1.4 (+4.3) | 88.9±0.4 (+2.5) | 90.2±0.3 (+0.2) |

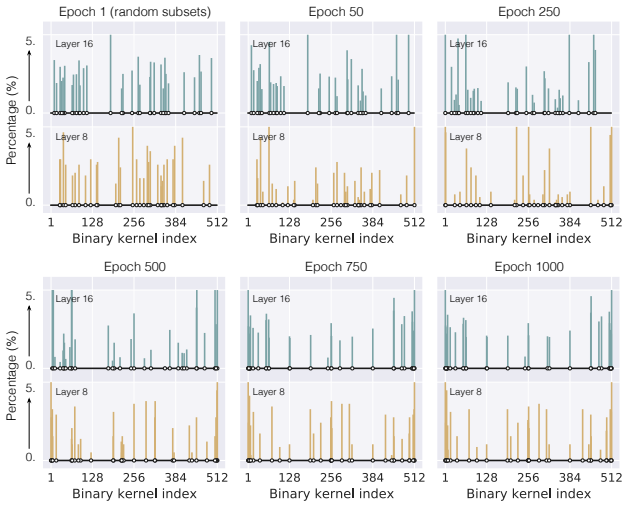


Figure 8. Visualization of how binary kernel subsets change during the training process of a 0.56-bit SNN. The experiment is performed on CIFAR10 with ResNet-20.

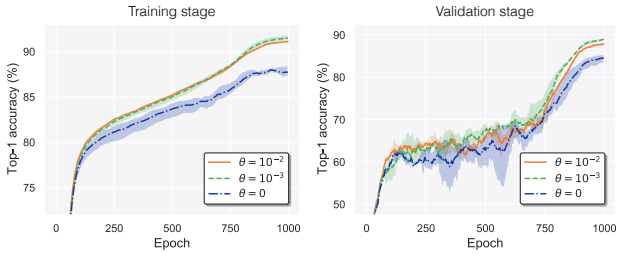


Figure 9. Top-1 accuracies vs. epochs with different $\theta$ settings in 0.56-bit SNNs, regarding the training stage (Left) and the validation stage (Right). Experiments are performed on CIFAR10 with ResNet-20, and each setting is performed three times with different random seeds.

### 5.3. Deployment Results

We use Intel® CoFluent™ Technology[2] to build a transaction level model and evaluate the performance of standard BNNs and our hardware design for SNNs. Intel CoFluent is a system modeling and simulation solution for performance predicting and architecture optimization during early design stages. Results of ResNet-18 and ResNet-34 are provided in Table 4, and per layer comparison is illustrated in Figure 10. We observe that our SNNs constantly achieve faster speeds on all layers compared with their BNN counterparts. For either ResNet-18 or ResNet-34, our 0.56-bit SNNs are **over three times faster** than the corresponding BNNs. In summary, compared with BNNs, when given the same computational resources, SNNs achieve much faster runtime speeds; when given the same throughput, SNNs greatly reduce the number of Arithmetic and Logic Units (ALUs) and thus reduce the chip area and power consumption.

[2] https://www.intel.com/content/www/us/en/cofluent/overview.html

Table 4. Practical speed tests of 0.56-bit SNNs and 1-bit BNNs deployments. The running time is evaluated with the $224 \times 224$ image input and based on a hardware configuration of 64PEs@1GHz. Activations are not binarized in the models.

| Backbone | Running time (ms) | | Speed up |
|---|---|---|---|
| | 1-bit BNN | 0.56-bit SNN | |
| ResNet-18 | 3.626 | **1.159** | **3.13×** |
| ResNet-34 | 7.753 | **2.329** | **3.33×** |

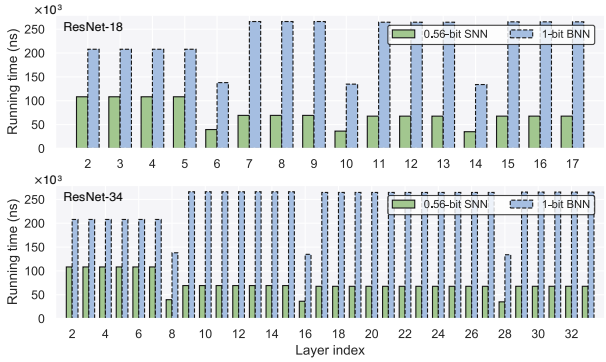


Figure 10. Comparison of the running time per layer for 0.56-bit SNNs and 1-bit BNNs deployments.

## 6. Conclusion

We propose Sub-bit Neural Networks (SNNs) which further compress and accelerate BNNs. The motivation is inspired by the observation that binary kernels in BNNs are largely clustered, especially in large networks or deep layers. Our method consists of two steps, randomly sampling layer-specific subsets of binary kernels, and subsets refinement with optimization. Experimental results and practical deployments verify that SNNs can be remarkably efficient compared with BNNs while maintaining high accuracies.


## Acknowledgement

This work is funded by Major Project of the New Generation of Artificial Intelligence (No. 2018AAA0102900) and the Sino-German Collaborative Research Project Crossmodal Learning (NSFC 62061136001/DFG TRR169). We thank Feng Chen and Zhaole Sun for the insightful discussions.

## References


[1] Yoshua Bengio, Nicholas Léonard, and Aaron C. Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. In *arXiv preprint arXiv:1308.3432*, 2013. 1, 2, 3, 4

[2] Adrian Bulat and Georgios Tzimiropoulos. Xnor-net++: Improved binary neural networks. In *BMVC*, 2019. 7

[3] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once-for-all: Train one network and specialize it for efficient deployment. In *ICLR*, 2020. 2

[4] Jia Deng, Wei Dong, Richard Socher, Li Jia Li, Kai Li, and Fei Fei Li. Imagenet: a large-scale hierarchical image database. In *CVPR*, 2009. 2, 6

[5] Ruizhou Ding, Ting-Wu Chin, Zeye Liu, and Diana Marculescu. Regularizing activation distribution for training binarized deep networks. In *CVPR*, 2019. 7

[6] Mark Everingham, Luc Van Gool, Christopher K. I. Williams, John M. Winn, and Andrew Zisserman. The pascal visual object classes (VOC) challenge. *IJCV*, 2010. 7

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016. 2, 6

[8] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *ICCV*, 2017. 1, 2

[9] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. In *arXiv preprint arXiv:1503.02531*, 2015. 1, 2

[10] Lu Hou, Quanming Yao, and James T. Kwok. Loss-aware binarization of deep networks. In *ICLR*, 2017. 7

[11] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017. 1, 2

[12] Qinghao Hu, Peisong Wang, and Jian ChengT. From hashing to cnns: Training binary weight networks via hashing. In *AAAI*, 2018. 7

[13] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *NeurIPS*, 2016. 7

[14] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. In *J. Mach. Learn. Res.*, 2017. 1

[15] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. In *Tech Report*, 2009. 2, 6

[16] Dongsoo Lee, Se Jung Kwon, Byeongwook Kim, Yongkweon Jeon, Baeseong Park, and Jeongin Yun. Flexor: Trainable fractional quantization. In *NeurIPS*, 2020. 2, 6, 7

[17] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. In *ICLR*, 2017. 1, 2

[18] Zefan Li, Bingbing Ni, Wenjun Zhang, Xiaokang Yang, and Wen Gao. Performance guaranteed network acceleration via high-order residual quantization. In *ICCV*, 2017. 7

[19] Darryl Dexu Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. In *ICML*, 2016. 1

[20] Mingbao Lin, Rongrong Ji, Zihan Xu, Baochang Zhang, Yan Wang, Yongjian Wu, Feiyue Huang, and Chia-Wen Lin. Rotated binary neural network. In *NeurIPS*, 2020. 2

[21] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: common objects in context. In *ECCV*, 2014. 7

[22] Xiaofan Lin, Cong Zhao, and Wei Pan. Towards accurate binary convolutional neural network. In *NeurIPS*, 2017. 2

[23] Zechun Liu, Wenhan Luo, Baoyuan Wu, Xin Yang, Wei Liu, and Kwang-Ting Cheng. Bi-real net: Binarizing deep network towards real-network performance. In *IJCV*, 2020. 2, 6, 7

[24] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, 2019. 6

[25] Haotong Qin, Ruihao Gong, Xianglong Liu, Mingzhu Shen, Ziran Wei, Fengwei Yu, and Jingkuan Song. Forward and backward information retention for accurate binary neural networks. In *CVPR*, 2020. 1, 2, 6, 7

[26] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *ECCV*, 2016. 1, 2, 6, 7

[27] Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. Fitnets: Hints for thin deep nets. In *ICLR*, 2015. 1, 2

[28] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR*, 2018. 1, 2

[29] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015. 6

[30] Yikai Wang, Fuchun Sun, Duo Li, and Anbang Yao. Resolution switchable networks for runtime efficient image recognition. In *ECCV*, 2020. 2

[31] Jiahui Yu and Thomas S. Huang. Universally slimmable networks and improved training techniques. In *ICCV*, 2019. 2

[32] Jiahui Yu, Linjie Yang, Ning Xu, Jianchao Yang, and Thomas S. Huang. Slimmable neural networks. In *ICLR*, 2019. 2

[33] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. In *ECCV*, 2018. 7

[34] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *CVPR*, 2018. 2

[35] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. Incremental network quantization: Towards lossless cnns with low-precision weights. In *ICLR*, 2017. 1

[36] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. In *arXiv preprint arXiv:1606.06160*, 2016. 7

[37] Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. Trained ternary quantization. In *ICLR*, 2017. 1