CONTRIBUTED PAPER

Model Compression and Hardware Acceleration for Neural Networks: A Comprehensive Survey

This article surveys recent advances toward the goal of efficient compression and execution of deep neural networks, without significantly compromising accuracy.

By Lei Deng, *Member IEEE*, Guoqi Li, *Member IEEE*, Song Han, Luping Shi, and Yuan Xie, *Fellow IEEE*

ABSTRACT | Domain-specific hardware is becoming a promising topic in the backdrop of improvement slow down for general-purpose processors due to the foreseeable end of Moore's Law. Machine learning, especially deep neural networks (DNNs), has become the most dazzling domain witnessing successful applications in a wide spectrum of artificial intelligence (AI) tasks. The incomparable accuracy of DNNs is achieved by paying the cost of hungry memory consumption and high computational complexity, which greatly impedes their deployment in embedded systems. Therefore, the DNN compression concept was naturally proposed and widely used for memory saving and compute acceleration. In the past few years, a tremendous number of compression techniques have sprung up to pursue a satisfactory tradeoff between processing efficiency and application accuracy. Recently, this wave has spread to the design of neural network accelerators for gaining extremely high performance. However, the amount of related works is incredibly huge and the reported approaches are quite divergent. This research chaos motivates us to provide a comprehensive survey on the recent advances toward the goal of efficient compression and execution of DNNs without significantly compromising accuracy, involving both the high-level algorithms and their applications in hardware design. In this article, we review the mainstream compression approaches such as compact model, tensor decomposition, data quantization, and network sparsification. We explain their compression principles, evaluation metrics, sensitivity analysis, and joint-way use. Then, we answer the question of how to leverage these methods in the design of neural network accelerators and present the state-of-the-art hardware architectures. In the end, we discuss several existing issues such as fair comparison, testing workloads, automatic compression, influence on security, and framework/hardware-level support, and give promising topics in this field and the possible challenges as well. This article attempts to enable readers to quickly build up a big picture of neural network compression and acceleration, clearly evaluate various methods, and confidently get started in the right way.

KEYWORDS | Compact neural network; data quantization; neural network acceleration; neural network compression; sparse neural network; tensor decomposition.

Manuscript received September 28, 2019; revised January 21, 2020; accepted February 20, 2020. Date of publication March 20, 2020; date of current version April 8, 2020. This work was supported in part by the National Science Foundation under Grant 1725447; in part by the Beijing Academy of Artificial Intelligence (BAAI), Tsinghua University Initiative Scientific Research Program; and in part by the grant from the Institute for Guo Qiang, Tsinghua University. *(Corresponding author: Guoqi Li.)*

Lei Deng is with the Department of Precision Instrument, Center for Brain Inspired Computing Research, Tsinghua University, Beijing 100084, China, and also with the Department of Electrical and Computer Engineering, University of California at Santa Barbara, Santa Barbara, CA 93106 USA (e-mail: leideng@ucsb.edu).

Guoqi Li and **Luping Shi** are with the Department of Precision Instrument, Center for Brain Inspired Computing Research, Tsinghua University, Beijing 100084, China, and also with the Beijing Innovation Center for Future Chip, Tsinghua University, Beijing 100084, China (e-mail: liguoqi@mail.tsinghua.edu.cn; lpshi@mail.tsinghua.edu.cn).

Song Han is with the Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139 USA (e-mail: songhan@mit.edu).

Yuan Xie is with the Department of Electrical and Computer Engineering, University of California at Santa Barbara, Santa Barbara, CA 93106 USA (e-mail: yuanxie@ucsb.edu).

Digital Object Identifier 10.1109/JPROC.2020.2976475

I. Introduction, Motivation, and Overview

Deep structure endows deep neural networks (DNNs) [1] the ability to learn high-level features from big data.

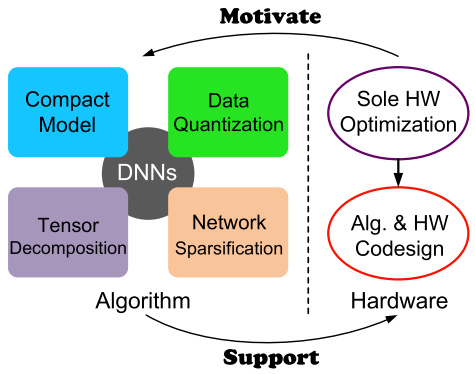


Fig. 1. *Overview of DNN compression and acceleration. "HW" denotes hardware, and "Alg." denotes algorithm.*

This powerful representation distinguishes itself from the traditional ways using hand-crafted features and enables impressive breakthroughs in a myriad of artificial intelligence (AI) applications, including image and speech recognition, natural language processing, object detection, autonomous driving, medical diagnosis, and game playing. Nowadays, DNNs have become the mainstay of AI tasks in both academia and industry.

Besides the powerful model innovations and available data sources, the explosion of DNNs also owes to the performance improvement of contemporary processors, especially graphics processing units (GPUs). The underlying reason for the GPU wave is that the conventional central processing units (CPUs) cannot satisfy the dramatically increasing requirements for memory bandwidth and computational complexity caused by the ever increasing model size of DNNs. For instance, the number of layers in DNNs can reach more than 10 000 [2], with millions to billions or even more parameters and intermediate states. Due to the huge cost of running DNNs, GPUs with high processing parallelism and memory bandwidth have been the major platform for AI in the cloud. However, efficient AI everywhere, not only in the heavy machine room, becomes an aim that humans chase. Therefore, there are growing interests in deploying DNNs on edge devices (e.g., smart sensor, wearable device, mobile phone, robot, drone, etc.) that have a stringent budget on the resource and energy, and expect real-time processing. As a result, reducing the cost of memory and compute in DNNs, that is, neural network compression becomes an urgent and promising topic.

The interaction between algorithm and hardware is a ubiquitous trend in the neural network compression domain, which can be seen from Fig. 1. On the algorithm side, various techniques have been proposed to compress DNNs in recent years, which can be classified into four categories: 1) compact model [3]; 2) tensor decomposition [4]; 3) data quantization [5]; and 4) network sparsification [6]. Specifically, the compact model aims at designing smaller base models that can still achieve acceptable application accuracy. The decomposition method decomposes the bloated parameters into a series of smaller matrices or tensors to shrink the memory volume and the operation number. Data quantization or network sparsification reduces the number of data bits or connections/neurons to compress the original model, respectively. The joint-way compression across multiple techniques also emerges to pursue extreme compression [7]. Interestingly, these approaches produce different tradeoffs in terms of application scenario, compression ratio, model accuracy, and hardware usability.

On the hardware side, the efficient processing of DNNs on specialized devices attracts continuous attention and stimulates many start-up companies. These devices are termed neural network accelerators. Conventional DNN accelerators solely optimize the hardware architecture to improve the compute parallelism and reduce memory accesses [8], [9]. Things have changed recently due to two reasons: 1) the sole hardware optimization is reaching the performance upper bound and 2) the emerging compression algorithms show great potential in reducing the hardware cost, which looks quite promising. A variety of compression techniques have been considered in the design of DNN accelerators at the cost of certain accuracy loss [10]–[13]. The high-level algorithm optimization provides guidance for the design of more efficient hardware and the low-level hardware design provides feedback for the design of more effective algorithms, which is usually called algorithm-hardware codesign. This codesign is ubiquitous in the design of modern DNN accelerators for the performance boost.

However, the number of related works is incredibly huge along with an ultrafast publishing speed, and the reported approaches are quite divergent. This greatly prevents beginners from obtaining a big picture and getting started in the right way; therefore, a survey is highly desired for correct recognition, analysis, and comparison. This article aims to provide such a comprehensive review of DNN compression and acceleration. Our scope involves most compression approaches and their hardware implementations. More importantly, we present vertical comparisons in each category of compression approaches, involving principle difference, compression ratio, accuracy degradation, and sensitivity analysis, as well as the promising joint-way compression. We also illustrate advanced hardware solutions that exploit these compression techniques and demonstrate the consequent benefits they have achieved. The future trends, possible opportunities, and challenging issues in this field are deeply discussed in the end. Compared with recent surveys on accelerators without compression [14], a certain category of compression approaches [15], or a rapid overview of multiple categories on only the algorithm side [16], [17], we present a more systematic summary on most existing methodologies, show more detailed descriptions and comparisons for the individual or joint use of these techniques, and talk about both algorithm and hardware synergistically.

The organization of this article is shown in Table 1 and is summarized as follows. Section II briefly gives preliminary knowledge of neural networks. Section III details the overall taxonomy and various compression

Table 1 Content Guidance of This Article

Preliminaries	Brief Preliminaries of Neural Networks		Section II
Algorithm	Compact Model	Compact CNNs	Section III-B1
		Compact RNNs	Section III-B2
		Neural Architecture Search	Section III-B3
	Tensor Decomposition	Low Rank Matrix Decomposition	Section III-C1
		Tensorized Decomposition	Section III-C2
	Data Quantization	Quantization Object and Philosophy	Section III-D1/III-D2
		Quantization on CNNs and RNNs	Section III-D3
	Network Sparsification	Sparsification Object and Philosophy	Section III-E1/III-E2
		Sparsifiation of Weights and Neurons	Section III-E3
	Joint-way Compression	Methods, Results, Discussions	Section III-F
Hardware	Why Domain-Specific?	From General to Specialized	Section IV-A
	Sole Hardware Optimization	Normal or Processing-in-Memory	Section IV-B
	Algorithm and Hardware Co-design	Transformation into Compact Model Tensorized Processing Engine Quantization Architecture Sparse Architecture	Section IV-C
	Joint-way Compression in Accelerator		Section IV-D
	Compression on Emerging Memory Devices		Section IV-E
	Performance Summary		Section IV-F
Discussions	Possible Opportunities and Challenges		Section V

approaches such as compact model, tensor decomposition, data quantization, network sparsification, individually, and jointly. We will systematically provide an analysis of the algorithm principle and flow, the comprehensive comparison in terms of methodology, compression ratio, model accuracy, and hardware usability, as well as the insights on sensitivity. Section IV goes through the major hardware platforms with different architectures for processing DNNs, including the conventional accelerators with sole hardware optimization and the modern ones with algorithm-hardware codesign. We will present the performance improvement benefited from the incorporation of compression techniques in the hardware design. Section V summarizes this article, discusses the potential tradeoff metrics that should be considered for a fair comparison, and foresees possible opportunities and challenges.

II. BRIEF PRELIMINARIES OF NEURAL NETWORKS

In this section, we briefly describe the background of neural networks and introduce several typical models. The neuron is the basic unit of a neural network, which receives signals from the connected preneurons, conducts a nonlinear transformation, and then produces an output signal that multicasts to postneurons. In the rest of this article, we term the connection as a synapse, the connection efficacy as weight ($\boldsymbol{W}$), the neuronal signal as activation ($\boldsymbol{X}$), and the nonlinear transformation as activation function ($\varphi(\cdot)$). Then, the neural operation follows:

$$y_i = \varphi\left(b_i + \sum_j x_j w_{ij}\right) \tag{1}$$

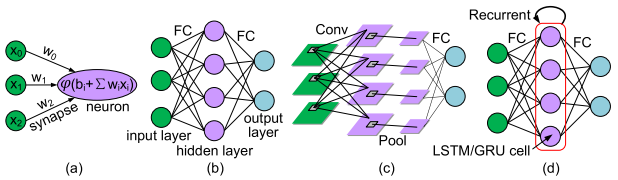


Fig. 2. *Neural network structures. (a) Single neuron. (b) MLP. (c) CNN. (d) RNN.*

where b_i is a bias, as illustrated in Fig. 2(a). There are three typical neural network models distinguished by different network structures: 1) multilayer perceptron (MLP); 2) convolutional neural network (CNN); and 3) recurrent neural network (RNN). All the three are individually shown in Fig. 2(b)–(d). The "deep" in DNNs indicates that the network usually has many hidden layers, while we show only one hidden layer in Fig. 2 for clarity. We provide the model details as follows.

MLP only includes feedforward fully connected (FC) layers as presented in Fig. 2(b). The basic calculation of each neuron is the same as given in (1). The activation function $\varphi(\cdot)$ mainly adopts $\text{ReLU}(x) = \max(x, 0)$ in modern networks.

CNN targets the processing of 2-D features instead of the 1-D ones in MLP, the structure of which is depicted in Fig. 2(c). It usually contains three types of layers: convolutional (Conv) layer, pooling (Pool) layer, and FC layer. In Conv and Pool layers, there are multiple channels (i.e., feature maps, FMs) to extract different local features of the input data. Each channel of a layer convolves and aggregates the features from multiple channels of the previous layer, thus gradually forming a higher abstraction layer by layer to help the final decision of the network.

The operation of the Conv layer is governed by $\mathbf{FM}_n^{\text{out}} = \varphi(b_n + \sum_m \mathbf{FM}_m^{\text{in}} \circledast \mathbf{W}(m,n))$, where $\mathbf{FM}_n^{\text{out}}$ is the nth output FM, that is, a 2-D matrix of activations, b_n is a bias item shared by all neurons in $\mathbf{FM}_n^{\text{out}}$, $\mathbf{FM}_m^{\text{in}}$ is the mth input FM, $\mathbf{W}(m,n)$ is a 2-D weight kernel (e.g., size of 3×3) connecting $\mathbf{FM}_m^{\text{in}}$ and $\mathbf{FM}_n^{\text{out}}$, $\circledast$ denotes the 2-D Conv operation, and $\varphi(\cdot)$ is the same as that in MLP. Next, the Pool layer is used to downsample FMs, that is, $\mathbf{FM}_n^{\text{out}} = \text{Pool}(\mathbf{FM}_n^{\text{in}})$, where $\text{Pool}(\cdot)$ is a pooling operation that generates each output activation based only on the local receptive field (RF) in the corresponding input FM. Max or average pooling produces the maximum or average value of each RF, respectively. Finally, FC layers follow Conv and Pool layers to work as a classifier with the previously extracted features, the computation of which is the same as that of MLP. Note that in this article we term $\mathbf{W}(m,:)$ as a weight channel and $\mathbf{W}(:,n)$ as a weight filter in a Conv layer.

RNN is designed with intralayer recurrent connections [see Fig. 2(d)], which is different from the sole feed-forward structure in MLP and CNN. The motivation is that although MLP and CNN are powerful for feature extraction, it is quite difficult for them to learn sequence. The recurrence in RNN incurs temporal dynamics, which is helpful for speech recognition and language processing with context information. Among them, the long short term memory (LSTM) [18] and gated recurrent unit (GRU) [19] are the most widely used due to the improved memorization over the vanilla RNN model. The dynamics of each LSTM cell can be described as

$$\begin{cases} \boldsymbol{f}(t) = \sigma(\boldsymbol{b}_f + \boldsymbol{W}_{fx}\boldsymbol{X}(t) + \boldsymbol{W}_{fh}\boldsymbol{h}(t-1)) \\ \boldsymbol{i}(t) = \sigma(\boldsymbol{b}_i + \boldsymbol{W}_{ix}\boldsymbol{X}(t) + \boldsymbol{W}_{ih}\boldsymbol{h}(t-1)) \\ \boldsymbol{o}(t) = \sigma(\boldsymbol{b}_o + \boldsymbol{W}_{ox}\boldsymbol{X}(t) + \boldsymbol{W}_{oh}\boldsymbol{h}(t-1)) \\ \boldsymbol{g}(t) = \theta(\boldsymbol{b}_g + \boldsymbol{W}_{gx}\boldsymbol{X}(t) + \boldsymbol{W}_{gh}\boldsymbol{h}(t-1)) \\ \boldsymbol{c}(t) = \boldsymbol{c}(t-1) \odot \boldsymbol{f}(t) + \boldsymbol{g}(t) \odot \boldsymbol{i}(t) \\ \boldsymbol{h}(t) = \theta(\boldsymbol{c}(t)) \odot \boldsymbol{o}(t) \end{cases} \tag{2}$$

where $\boldsymbol{f}$, $\boldsymbol{i}$, and $\boldsymbol{o}$ are the states of forget, input, and output gates, respectively, and $\boldsymbol{g}$ is the input activation. Each gate has its own weight matrices and bias vector. $\boldsymbol{c}$ and $\boldsymbol{h}$ are cellular and hidden states of the hidden layer, respectively. $\sigma(\cdot)$ and $\theta(\cdot)$ are sigmoid function [$\text{sigmoid}(x) = 1/(1+e^{-x})$] and $\tanh$ function [$\tanh(x) = (e^x - e^{-x})/(e^x + e^{-x})$], respectively. $\odot$ denotes the Hadamard product. GRU is similar to LSTM but the gate structure is simplified. We provide its dynamics as follows:

$$\begin{cases} \boldsymbol{z}(t) = \sigma(\boldsymbol{b}_z + \boldsymbol{W}_{zx}\boldsymbol{X}(t) + \boldsymbol{W}_{zh}\boldsymbol{h}(t-1)) \\ \boldsymbol{r}(t) = \sigma(\boldsymbol{b}_r + \boldsymbol{W}_{rx}\boldsymbol{X}(t) + \boldsymbol{W}_{rh}\boldsymbol{h}(t-1)) \\ \boldsymbol{g}(t) = \theta(\boldsymbol{b}_g + \boldsymbol{W}_{gx}\boldsymbol{X}(t) + \boldsymbol{W}_{gh}(\boldsymbol{r}(t) \odot \boldsymbol{h}(t-1))) \\ \boldsymbol{h}(t) = (1 - \boldsymbol{z}(t)) \odot \boldsymbol{h}(t-1) + \boldsymbol{z}(t) \odot \boldsymbol{g}(t) \end{cases} \tag{3}$$

where $\boldsymbol{z}$ and $\boldsymbol{r}$ denote the states of the update and reset gate, respectively. We can see that GRU reduces the gate number and removes the cellular state for simplicity.

Model training is to adjust the parameters (e.g., weights and biases) according to a certain learning rule, to approach a pregiven output object. A naive optimization object can be

$$\min_{\boldsymbol{W},\,\boldsymbol{b}} L = \frac{1}{2}\|f(\boldsymbol{X},\,\boldsymbol{W},\,\boldsymbol{b}) - \boldsymbol{Y}^{\text{label}}\|_2^2 \tag{4}$$

where $f(\cdot)$ denotes the whole network model, $\boldsymbol{X}$ is the set of input samples, and $\boldsymbol{Y}^{\text{label}}$ is the set of target labels. The goal is to adjust $\boldsymbol{W}$ and $\boldsymbol{b}$ to minimize the distance between the actual network output and the expected one (i.e., label). The objective function is also termed the loss function, which can be usually solved by the widely used stochastic gradient descent (SGD).

Error backpropagation (BP) is the explicit expression of SGD, which is convenient for programming deep networks. The training is split into two passes: the forward pass and the backward pass. In the forward pass, the neuronal activations propagate layer by layer according to (1), and the activations are stashed for the parameter update latter. In the backward pass, the error signals (the distance between the actual and expected outputs) propagate reversely from the last layer to the first layer. The error propagation is governed by

$$\frac{\partial L}{\partial x_i} = \sum_j \frac{\partial L}{\partial y_j}\varphi_j' w_{ji} \tag{5}$$

where φ' is the derivative of the activation function. The BP computation is similar to the activation propagation of the forward pass but with a reverse direction, and the activation function changes to its derivative. Then, the parameter gradients are produced by the error signals and the stashed activations

$$\bigtriangledown w_{ji} = \frac{\partial L}{\partial y_j}\varphi_j' x_i, \quad \bigtriangledown b_j = \frac{\partial L}{\partial y_j}\varphi_j'. \tag{6}$$

Finally, the parameters are updated as follows:

$$\boldsymbol{W} \Leftarrow \boldsymbol{W} - \eta \bigtriangledown \boldsymbol{W}, \quad \boldsymbol{b} \Leftarrow \boldsymbol{b} - \eta \bigtriangledown \boldsymbol{b} \tag{7}$$

where η is a learning rate. Here, we present only the simplest objective function and learning rule. Actually, there are many complicated variants for better convergence, which are out of the scope of this article.

III. NEURAL NETWORK COMPRESSION: ALGORITHMS

In this section, we first introduce the overall taxonomy of this article for various neural network compression approaches, and then we detail specific approaches one by one.

A. Taxonomy of Compression Approaches

Fig. 3 shows our taxonomy to categorize such divergent approaches on neural network compression. Note that here

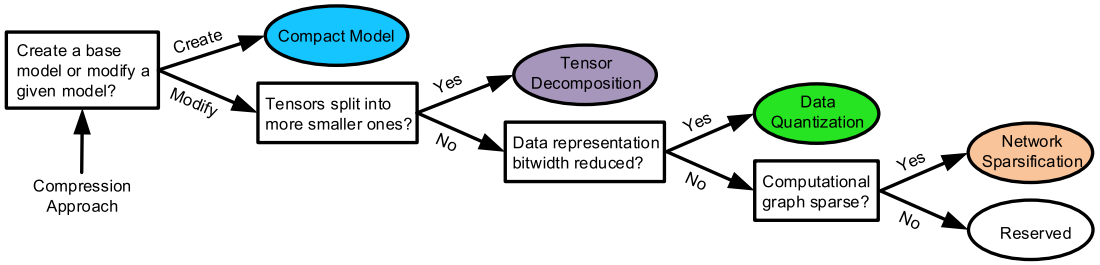


Fig. 3. ***Taxonomy of neural network compression approaches.***

the joint-way compression with multiple approaches is not included in this figure for clarity. For any compression approach, if it aims at creating a base model itself from a pool of candidates, it is recognized as the compact model approach. By contrast, the other three approaches modify a given base model to shrink the memory and compute costs. Specifically, if the original tensors in a neural network are split into more smaller tensors, we call the approach tensor decomposition. Otherwise, we have to do further recognition: if the data representation bitwidth is reduced, it should be the data quantization approach; while if the computational graph is sparsified with fewer effectual weights and/or neurons, it belongs to the network sparsification approach.

Besides the four kinds of compression approaches, there are three kinds of networks mentioned in Section II: MLP, CNN, and RNN. Whereas, we do not make a taxonomy based on network types, which is due to the following reasons. First, the approaches for different network types have large overlaps. An FC layer can be regarded as a Conv layer with 1×1 FM size; therefore, many compression approaches for Conv layers can also be applied in MLP/RNN with FC layers. For example, in tensor decomposition, data quantization, and network sparsification, we think most of the approaches can be shared between different networks; while in compact model, although the spatial correlation for CNN and the unit level design for RNN are incompatible, the channel correlation for CNN and the network level design for RNN are similar to some extent. Second, the reason that the studies on RNN and MLP are much fewer than those on CNN is mainly due to the popularity of CNN in current AI tasks and the more standardized benchmarks for comparison in the CNN family, rather than the approach incompatibility; besides, the FC layers of CNN can include the MLP structure, thus only very few works investigate the compression of MLP. At last, if we separate different network types into different subsections, we have to repeat the four kinds of compression approaches in each of them due to the shared approaches in many cases, which might cause a mess. With all these concerns, we use the different compression approaches as a higher-level taxonomy and inserting the works on different networks into each approach.

B. Compact Model

1) Compact CNNs—Spatial and Channel Correlation: Ever since the significant breakthrough of AlexNet [20] in image classification tasks, CNNs have experienced a lot of architectures and models that achieve better and better accuracy. Besides, their applications also spread to other computer vision tasks such as object detection and semantic segmentation, and even sequential tasks with 1-D Conv and video stream tasks with 3-D Conv. However, the performance improvement is mainly driven by deeper and wider networks with increased parameters and operations (e.g., multiply-and-accumulate, MAC), which usually slow down their execution especially on mobile devices. This motivates the design of compact models with reduced overhead while maintaining accuracy as much as possible.

Although compact networks can be obtained by various techniques for each specific task, the typical and general trends are deeper structure with expanded FMs, more complex branch topology, and more flexible Conv arithmetic. In this section, we summarize the representative methods for 2-D Conv from two aspects as depicted in Fig. 4: 1) the spatial correlation within a Conv layer, where FMs are convolved with multiple weight kernels and 2) the intralayer and interlayer channel correlations, where FMs are aggregated by different topology. Specifically, each output FM corresponds to multiple weight kernels, where each kernel can extract one type of local features from an input FM through the Conv operation. Then, by aggregating the extracted features from multiple input FMs, the output FM produces a higher feature abstraction of the input data. In a similar way, multiple output FMs produce different features by virtue of different weight kernels. The feature propagation gradually improves the abstraction level from local to global, which finally enables the network decision such as image recognition.

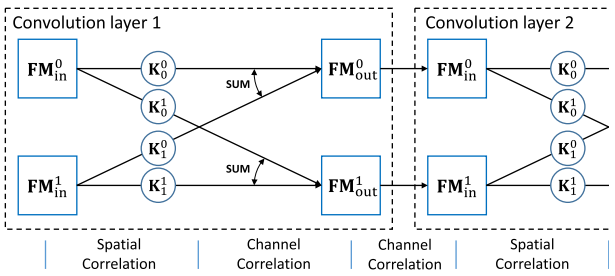


Fig. 4. ***Example of standard Conv. K_i^j denotes the Conv kernel between the ith input FM and the jth output FM.***

Table 2 Comparison of Conv Kernels With Identical RF

Type	C5x5	C3x3	C2x2	Asymmetric	Shift
Factorization	C5x5	C3x3 C3x3	C2x2 C2x2 C2x2 C2x2	C1x5 C5x1	S5x5 C1x1
#Parameters	25	18	16	10	1
Normalized	1.39	1	0.89	0.56	0.06

Note Abbreviations: C-Conv, S-FM shifting. Here #parameters is proportional to #MACs.

a) Spatial correlation to enlarge RF: Compared to MLP, the key feature of CNNs is the local connections with shared weights, which results in high spatial correlation within the RF and the invariance for spatial transformation. Thus, designing effective RFs [21] is crucial to the representation capability of Conv layers, which is jointly determined by the weight kernel size and weight kernel pattern.

Even though RFs can be simply enlarged by using a large kernel size, it will result in a quadratic growth of parameters and operations. A stack of two 3×3 Conv (C3×3) layers approximately has an effective RF of C5×5, but with 28% fewer parameters and operations (Table 2). This can be treated as imposing a regularization on the original 5×5 kernels, and decompose them to a subset of two 3×3 kernels with nonlinearity injected in between. The principle of using the 3×3 kernel is widely adopted in modern CNNs. An intuitive question comes out: can we aggressively use smaller kernel size, for example, 3×1 (asymmetric kernel) or 2×2 (even kernel)?

For the asymmetric case [22], an $n \times n$ Conv can be factorized into an $n \times 1$ Conv followed by a $1 \times n$ Conv, thus the parameters and operations decrease dramatically from $O(n^2)$ to $O(2n)$. Experiments show that the asymmetric Conv gives good results on FMs with size 12–20 [22]. Whereas, as for the case of 2×2 kernel, factorizing a C3×3 into two C2×2s produces only 11% saving of overhead. Besides, the even-size kernel makes the zero-padding necessarily asymmetric and leads to a half-pixel shifting in the resulting FMs. This location offset accumulates when stacking multiple even-sized Conv layers and eventually squeezes the spatial information to a certain corner of the original input FMs. In ShiftNet [23], the authors therefore directly leverage the shift effect and enlarge RFs by replacing the 3×3 kernel with an isotropic FM shifting followed by a 1×1 pointwise Conv. Then the spatial correlation is achieved without parameters and MACs.

Another way to enlarge RFs is to use atrous Conv [24], also known as dilated Conv [25]. As shown in Fig. 5, atrous Conv uses irregular kernels with holes, dilations, or disperse organization. Dilated Conv supports the exponential expansion of RFs without loss of resolution or coverage. The dilation algorithm can apply the same kernel at different ranges using different dilation factors, thus aggregating multiscale contextual information without compromising resolution. Dilated Conv effectively enlarges RFs without increasing the number of parameters and MACs, which is specifically suitable for dense prediction tasks such as semantic segmentation, voice generation, and machine translation. Deformable Conv [26] is a generalization of atrous Conv, which augments the spatial sampling locations of kernels via additional 2-D offsets and learning the offsets directly from the target tasks. Thus, the deformable kernels are more task-specific and greatly enhance the capability of modeling geometric transformation.

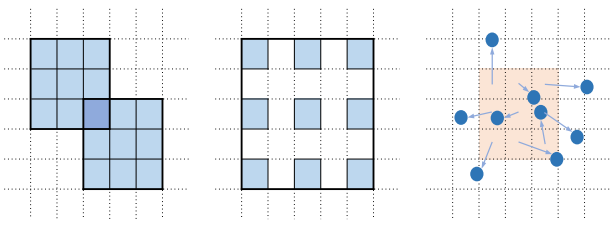


Fig. 5. *(a) Sampling locations in standard Conv (stacked 3x3 conv). (b) and (c) Sampling locations in atrous Conv (dilated 3 × 3 conv and deformable 3 × 3 conv, resp.).*

b) Channel correlation—The topology art: Other than enlarging RFs, most research studies on compact CNNs focus on the network structure, that is, the topology of channel aggregation within a Conv layer or between Conv layers.

In VGG-Nets [27], the interlayer topology is rather simple, where multiple Conv layers are stacked head-to-tail in sequence. However, as the network deepens, the accuracy saturates and degrades rapidly in practice, which is termed the degradation problem [28]. In this situation, it seems that the SGD optimizer meets difficulty in finding a better solution. To address this issue, Inception-V1 [29] adds two extra auxiliary classifiers connected to the intermediate layers to help the propagation of gradients back through all layers.

A more effective and elegant way is the shortcut connection, where the input of one Conv layer and the output after several stacked Conv layers are connected by gating or identity functions [28], [30]. Specifically, an example of the identity connection used in ResNet [30] [see Fig. 6(a)] can be formulated as

$$\mathbf{x}_{\ell+2} = \text{Conv}_{\ell+1}(\text{Conv}_{\ell}(\mathbf{x}_{\ell})) + \mathbf{x}_{\ell} \tag{8}$$

where $\mathbf{x}_{\ell}$ is the input of the lth layer, $\text{Conv}_{\ell}(\cdot)$ denotes the Conv operation and the activation function within that layer. ResNet achieves accuracy gains via the alleviated gradient vanishing and thus extremely increased depth (over 100 layers). An interpretation of ResNet is that the ultra-deep network is decomposed by identity connections and is formed as an ensemble of relatively shallow networks

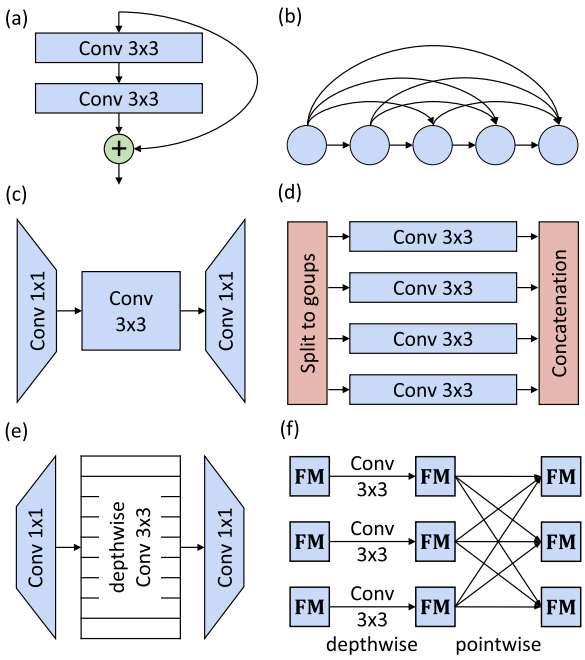


Fig. 6. ***Typical interlayer and intralayer channel correlations. (a) Residual connection [30]. (b) Densely connected block [31]. (c) Bottleneck architecture [22]. (d) Group Conv [33]. (e) Reversed bottleneck architecture with depthwise Conv in between [3]. (f) Depthwise-separable Conv [34].***

that are much easier to train. Similar to but different from ResNet, DenseNet [31] constructs shortcut connections by densely connected blocks

$$\mathbf{x}_{\ell+1} = \text{Conv}_\ell([\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_\ell]) \tag{9}$$

where $[\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_\ell]$ refers to the concatenation of FMs produced by all the preceding layers [see Fig. 6(b)]. DenseNet exploits the network potential through feature reuse, yielding more condensed models that are highly parameter efficient.

The above networks are composed of relatively simple layers and modules in sequence. Further increase of the depth or width of the network usually results in a diminishing performance gain (e.g., a 1202-layer ResNet performs worse than that with 110 layers [32]), meanwhile the cost is a quadratic growth in both parameters and operations. The topology of channels within a Conv layer (intralayer) aims to reduce the overhead of a standard Conv. For each output channel, the Conv can be divided into two steps (see Fig. 4): 1) for each input channel, getting the Conv result by convolving with a weight kernel (spatial correlation) and 2) linearly aggregating all Conv results from different input channels (channel correlation).

The first idea is to explicitly leverage the linear channel correlation, or the pointwise C1×1 utilized in network in network (NIN) [35]. Even though C1×1 is essentially a linear projection across all input FMs without any spatial correlation, it can reduce the number of channels without serious adverse effects. This principle is latter formed as the bottleneck architecture [22], [31] [see Fig. 6(c)]: instead of using a single standard C3×3 with n fan-in FMs and n fan-out FMs, the bottleneck architecture first projects n FMs to n/k FMs using C1×1, then does a standard C3×3 with k-time lower dimensions, and finally projects the n/k FMs back to n FMs through another C1×1. The bottleneck architecture also has a reversed version [3] [see Fig. 6(e)].

Another method to alleviate the Conv overhead is group Conv [33]. The implementation of group Conv is simple, that is, splitting n fan-in FMs and n fan-out FMs to k individual groups, where each group has n/k fan-in FMs and n/k fan-out FMs [see Fig. 6(d)]. The calculation follows:

$$\mathbf{x}_{\ell+1} = \left[\text{Conv}_\ell^1(\mathbf{x}_\ell^1), \text{Conv}_\ell^2(\mathbf{x}_\ell^2), \ldots, \text{Conv}_\ell^k(\mathbf{x}_\ell^k)\right] \tag{10}$$

where k denotes the group index. Group Conv is especially suitable for the distributed training over multiple GPUs with inadequate memory for each [20]. ResNeXt [33] is a group Conv version of ResNet that improves accuracy. Condensenet [36] introduces a learned group Conv based on the original DenseNet. During training, the learned group Conv removes the C1×1 connections and rearranges them during inference so that the resulting model can be implemented as a standard one-to-one group Conv. A side effect of stacking multiple group Convs is that the output from a certain channel is derived only from a certain group of input channels. This property strongly regularizes the channel correlation and weakens the network expressive power. ShuffleNet [37] overcomes it via the channel shuffle operation, which rearranges FMs and links channels among different groups. Gao *et al.* [38] proposed a channel-wise Conv to fuse information from different channel groups. Group Conv is further studied in [39], where each Conv block is composed of multiple interleaved group convolutions (IGCs) and each of them has different permutation strategy and kernel size.

An extreme case of group Conv is depthwise Conv in which the number of groups equals the number of channels. In practice, depthwise Conv is usually followed by a pointwise Conv, coupled as the depthwise-separable Conv [34] [see Fig. 6(f)]. The fundamental hypothesis behind depthwise-separable Conv is that spatial and channel correlations can be sufficiently decoupled and

Table 3 Comparison of Convolutions With 128 Fan-In FMs and 128 Fan-Out FMs

Type	Standard	Group	Bottleneck	Depthwise	Shift
Factorization	128C3x3	32C3x3 32C3x3 32C3x3 32C3x3	32C1x1 32C3x3 128C1x1	128D3x3 128C1x1	128S5x5 128C1x1
#Parameters	147456	36864	17408	17536	16384
Normalized	100	25	11.8	11.9	11.1
Percentage of C1x1 (%)	0	0	47	93	100

Note Abbreviations: C-standard Conv, D-depthwise convolution, S-FM shifting. Here #parameters is proportional to #MACs.

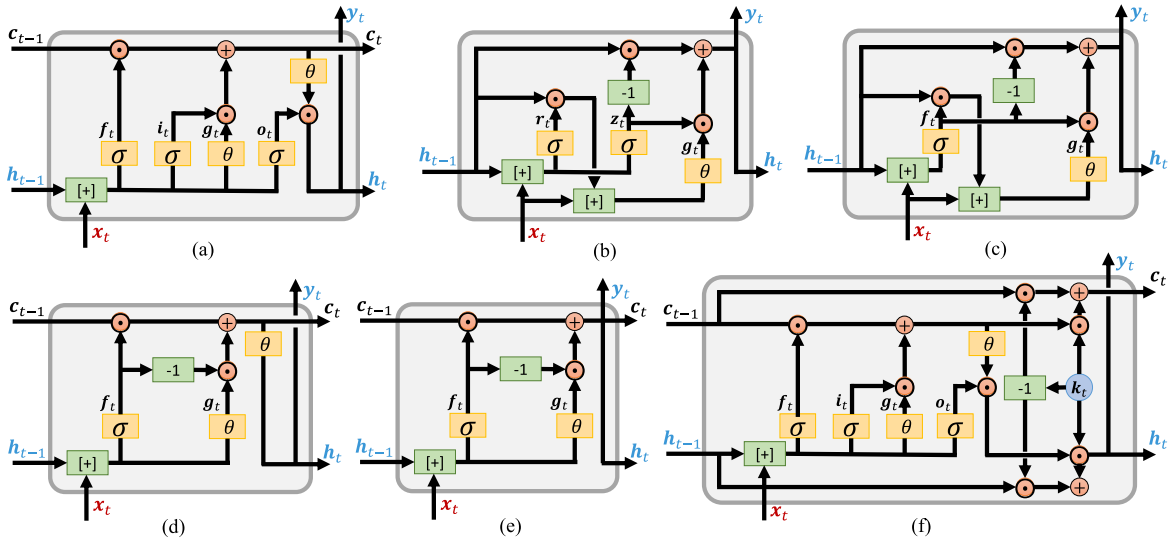


Fig. 7. ***Examples of standard and compact RNN units. (a) LSTM [18]. (b) GRU [19]. (c) MGU [40]. (d) S-LSTM [41]. (e) JANET [42]. (f) Phased LSTM [43].***

separately realized. Compared to the standard Conv, MobileNet [44] shows that using depthwise-separable Conv only reduces accuracy by 1% on ImageNet but saves lots of parameters and MACs. Depthwise-separable Conv has become a popular component for compact CNNs [3], [37], [45]. In Table 3, modern compact models tend to use more C1×1s that can occupy the vast majority of parameters and operations, for example, about 95% in MobileNet. Therefore, how to sparsify or reduce the overhead of C1×1 is of great interest for future work.

2) Compact RNNs—Unit and Network Level: Proverbially, the internal structure of RNN units is complex, which can be seen from the LSTM and GRU units defined in (2) and (3); besides, the information in an RNN model propagates along both spatial and temporal dimensions, which is also complex. Therefore, in this part, we discuss recent advances in RNNs that aim to improve the execution efficiency by designing compact RNNs in the level of unit or network.

a) Unit level: LSTM is designed to improve the long-term dependence over vanilla RNNs by alleviating the issue of gradient explosion or vanishing [18], [46]. Although LSTM performs reasonably well in various tasks, the complex design of the gates brings considerable parameters and operations. It is also not clear whether all the components in LSTM are indispensable. Studies on the gate search and evolution of LSTM [47] indicate that the forget gate is the most critical component, the input gate is relatively important, while the output gate is not.

With the above concerns, GRU [19] is proposed and widely used as a simplified version of LSTM. It removes the peephole connections and couples the input and forget gates into an update gate. Furthermore, S-LSTM [41] and JANET [42] contain only one forget gate. Minimal gated unit (MGU) [40] couples the reset and update gates of GRU. These architectures attempt to simplify the gates and have significantly fewer parameters than the regular LSTM. Experiments show that these compact units significantly reduce the model complexity without degrading performance. On the contrary to the gate simplification, phased LSTM [43] extends the LSTM unit by adding an extra time gate, which is controlled by a parameterized oscillation and produces sparse updates of the memory cell, thus achieving faster convergence and better performance than the regular LSTM. Fig. 7 illustrates the detailed topology of these compact RNN units.

There are also works beyond the optimization of gates. Independently RNN (IndRNN) [48] is proposed to prevent the gradient explosion and vanishing problems without using additional gates. In IndRNN, neurons in the same layer are independent of each other, connected as

$$\mathbf{h}_t = \text{sigmoid}(\mathbf{W}\mathbf{x}_t + \mathbf{u} \odot \mathbf{h}_{t-1} + \mathbf{b}) \tag{11}$$

where $\boldsymbol{u}$ is a vector-format parameter. IndRNN can be seen as a nongate version of GRU. Quasi-RNN (QRNN) [49] replaces the previous hidden state $\mathbf{h}_{t-1}$ with the previous input $\mathbf{x}_{t-1}$ in the gate calculation, taking the forget gate as an example

$$\mathbf{f}_t = \sigma(\mathbf{W}_f[\mathbf{x}_t, \mathbf{x}_{t-1}] + \mathbf{b}_f). \tag{12}$$

In this way, the heaviest calculation can be performed in parallel along both the temporal and mini-batch dimensions. Inspired by QRNN, simple recurrent unit (SRU) [50] replaces $\mathbf{W}\mathbf{h}_{t-1}$ with a lightweight Hadamard product of a vector-format parameter and the previous cellular state, that is, $\mathbf{V} \odot \mathbf{c}_{t-1}$, then the computation of $\mathbf{c}_t$ becomes independent and parallelizable. Unitary RNN (uRNN) [51] explores the use of orthogonal and unitary weight matrices in memory-restricted conditions.

b) Network level: Another path to explore compact RNNs is the simplification of stacked layers, which in many ways follows the similar inspirations as the channel correlation for CNNs and extends to both spatial and temporal dimensions. Sak *et al.* [52] introduced a linear recurrent projection layer to reduce the dimension of hidden states, that is, $\boldsymbol{r}_t = \boldsymbol{W}_r \boldsymbol{h}_t$ where the length of $\boldsymbol{r}_t$ is smaller than that of $\boldsymbol{h}_t$, and feed the shorter $\boldsymbol{r}_{t-1}$ back to replace the longer $\boldsymbol{h}_{t-1}$. Specifically, the gate calculation becomes

$$\textbf{gate}_t = \sigma(\boldsymbol{W}_x \boldsymbol{x}_t + \boldsymbol{W}_h \boldsymbol{r}_{t-1} + \boldsymbol{b}) \tag{13}$$

where **gate** represents any specific gate. Thus, the new linear recurrent LSTM block can achieve more efficient use of the model parameters. Factorized LSTM [53] factorizes each weight matrix of an LSTM into two smaller matrices and feeds them with two subvectors. The outputs from individual groups are then concatenated as the final output. The group-LSTM can achieve state-of-the-art perplexity while using significantly fewer parameters and less convergence time. Taking the following equation as an example:

$$\begin{pmatrix} \boldsymbol{i}_t \\ \boldsymbol{f}_t \\ \boldsymbol{o}_t \\ \boldsymbol{g}_t \end{pmatrix} = \left(\begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \theta \end{pmatrix} \boldsymbol{T}^1 \begin{pmatrix} \boldsymbol{x}_t^1 \\ \boldsymbol{h}_{t-1}^1 \end{pmatrix}, \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \theta \end{pmatrix} \boldsymbol{T}^2 \begin{pmatrix} \boldsymbol{x}_t^2 \\ \boldsymbol{h}_{t-1}^2 \end{pmatrix} \right) \tag{14}$$

the factorized matrices $\boldsymbol{T}^1$ and $\boldsymbol{T}^2$ have in total $2\times$ fewer parameters compared to the original large matrix. Inspired by ResNet [30], Wu *et al.* [54] added residual connections into their eight-layer LSTM for machine translation. According to their experiments, a simply stacked LSTM network can work well with only up to four layers. A similar but different approach is the skip-connected RNN [55], where skip connections are explicitly added into the RNN graph for improving the performance by allowing direct information transmission between nonconsecutive timesteps. In grid LSTM [56], the LSTM cells are arranged in a 3-D grid: the temporal dimension of the sequence, the vertical dimension of the network, and the depth. Grid LSTM stacks layers in a multidimensional state projection with flexible weight sharing, that is, the weights are shared along any single dimension or even all dimensions. The whole network can be applied to process high-dimensional data such as images. Tensorized RNN and LSTM [57] represent the hidden states as tensors and update them via a cross-layer convolution. By increasing the tensor size and sharing parameters across different locations in the tensor, the network can be efficiently widened without additional parameters. It should be emphasized here that they tensorize the intermediate state data rather than weights to simplify the computation, and moreover the parameter sharing rather than decomposition (to be introduced in Section III-C) is utilized so that the topology between layers is changed. In detail, taking the RNN case as an example, the input activation vector $(\boldsymbol{W}_x \boldsymbol{x}_t + \boldsymbol{b}_x) \in \mathbb{R}^M$ and the P (number of stacked layers) hidden state vectors $\boldsymbol{h}_t^p \in \mathbb{R}^M$ can be concatenated to become $\boldsymbol{H}_t^{\text{cat}} \in \mathbb{R}^{(P+1)\times M}$. Then the corresponding recurrent weights can be defined in a shared format as $\boldsymbol{W}_h \in \mathbb{R}^{K\times M\times M}$ where K is termed kernel size. Finally, the hidden states can be updated by a cross-layer convolution like

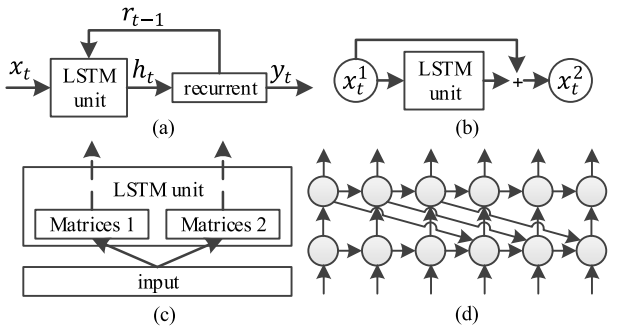


Fig. 8. ***Examples of compact RNNs in network level. (a) Linear recurrent LSTM [52]. (b) Residual LSTM [54]. (c) Factorized LSTM [53]. (d) Skip-connected RNN [55].***

$$\boldsymbol{H}_t = \theta\left(\boldsymbol{H}_{t-1}^{\text{cat}} \circledast \{\boldsymbol{W}_h, \boldsymbol{b}_h\}\right) \tag{15}$$

where $\boldsymbol{b}_h \in \mathbb{R}^M$ denotes the bias vector.

Generally, compact RNNs in the network level pay more attention to the topology between LSTM units and the whole network architecture, rather than the boundary of a single unit. Some of the mentioned examples are illustrated in Fig. 8. In essence, the compact design in the unit level and the network level has no conflicts with each other, that is to say, one may design a compact RNN model the units of which are concise and the network architecture is also tiny. However, practical evidence is still lacking. We suppose that there exist coupling effects between units and layers. Therefore, if a compact RNN unit is already compact at the unit level, there might be poor room for us to compress the network-level redundancy. Anyway, we expect that future studies can make a final conclusion to confirm this point or develop new compact RNNs by fully exploiting the techniques at different levels.

3) Neural Architecture Search: Although the above network engineering has achieved great success, they crucially rely on human expert knowledge, hand-crafted designs, and tons of efforts for training them. In recent years, there is a growing demand and trend for applying machine-learning techniques, known as neural architecture search (NAS) [45], [58], to automatically optimize the neural network architectures. Here the term "architecture" denotes the network structure rather than the latter hardware architecture in Section IV. Note that the automated NAS techniques can also be employed to design RNN [45], [59], [60] and transformer [61] models. Whereas, they are still too few compared with

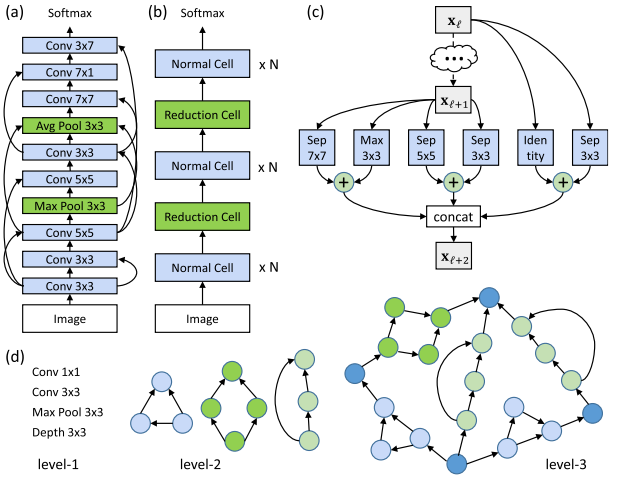


Fig. 9. ***Illustration of NAS. (a) Full-network search [45]. (b) Cell-based search [58]. (c) Example of learned cell structures [66]. (d) Three-level hierarchical search [67].***

CNNs, which is the reason that we mainly review the NAS research studies on CNNs in this part.

In the conventional NAS formulation [45], designing neural network architectures is modeled as a sequence generation problem, where an autoregressive RNN controller is introduced to generate neural network architectures. This RNN controller is trained by repeatedly sampling neural network architectures, evaluating the sampled architectures, and updating the controller based on the feedback. To find a good neural network architecture in the vast search space, this process typically needs to train and evaluate tens of thousands of neural networks [45] on the target task, leading to a prohibitive computational cost (e.g., 10^4 GPU hours). To address this challenge, many techniques are proposed to improve different components of NAS, including search space, search algorithm, and quality (typically accuracy on the validation set) evaluation of sampled architectures.

a) Search space: All the NAS methods need a predefined search space that contains basic network elements and how they connect with each other. The typical basic elements consist of: 1) Convs [58], [62]: standard Conv (e.g., C1×1, C3×3, C5×5), asymmetric Conv (e.g., C1×3 and C3×1, C1×7 and C7×1), depthwise-separable Conv (e.g., C3×3, C5×5), dilated Conv (e.g., 3×3); 2) pool: average pooling (e.g., 3×3), max pooling (e.g., 3×3); and 3) activation function [63]. Then these basic elements are stacked sequentially [64] with identity connections [45]. The full-network search space grows exponentially as the network deepens [see Fig. 9(a)]. For example, when the depth reaches 20, this search space contains more than 10^{36} different neural network architectures in [45].

Instead of directly searching in such an exponentially large space, restricting the search space is a very effective approach for improving the search speed. Specifically, it is efficient to search for basic building cells [see Fig. 9(b)] that can be stacked sequentially to construct neural networks [58], [65], rather than to directly find the neural network architecture as a whole. As such, the architecture complexity is independent of the network depth, and the learned cells are transferable across different data sets. It enables NAS to search on a small proxy data set (e.g., CIFAR-10), and then transfer to another large-scale data set (e.g., ImageNet) by adapting the number of cells. Within the cell, the complexity is further reduced by increasing the number of blocks in a progressive manner [66] [see Fig. 9(c)] or supporting hierarchical topology [67] [see Fig. 9(d)].

b) Search algorithm: NAS methods usually have two stages at each search iteration: 1) the generator produces an architecture, and then 2) the evaluator trains the network and evaluates the quality. As getting the quality of a sampled neural network architecture involves training a neural network, which is very expensive, search algorithms that affect the sampling efficiency play an important role in improving the search speed of NAS. Most of the search algorithms fall into five categories: random search, reinforcement learning (RL), evolutionary algorithms (EAs), Bayesian optimization, and gradient-based methods. Among them, RL, EA, and gradient-based methods often provide the most competitive results.

RL-based methods model the architecture generation process as a Markov decision process, treat the validation accuracy of the sampled architecture as the reward, and update the architecture generation model using RL algorithms, including Q-learning [64], [65], policy-based REINFORCE [45], and proximal policy optimization (PPO) [58]. Instead of training an architecture generation model, evolutionary methods [62], [67] maintain a population of neural network architectures. This population is updated through mutation and recombination. While both RL-based and evolutionary methods optimize the architectures in a discrete space, DARTS [60] proposes continuous relaxation of the architecture representation

$$y = \sum_i \alpha_i o_i(x), \quad \alpha_i \geq 0 \ \& \ \sum_i \alpha_i = 1 \tag{16}$$

where $\{\alpha_i\}$ denotes architecture parameters, $\{o_i\}$ denotes candidate operations, x is the input, and y is the output. Such continuous relaxation allows the optimization of neural network architectures in a continuous space using gradient descent, which greatly improves the search efficiency. Besides the above techniques, the search efficiency of NAS can also be improved by exploring the architecture space with network transformation operations, starting from an existing network and reusing the weights [68]–[70].

c) Quality evaluation: To guide the search process, NAS methods need to get the quality of sampled neural network architectures. A trivial solution is to train sampled

Table 4 Comparison of NAS Methods

Reference	Algorithm	GPU days	CIFAR10		Imagenet	
			(%)	(M)	(%)	(M)
MetaQNN [64]	Q-learning	100 (-)	6.92	-	-	-
NAS [45]	REINFORCE	22400[1] (K40)	3.65	37.4	-	-
EAS [68]	REINFORCE	10 (GTX1080)	3.44	10.7	-	-
BlockQNN [65]	Q-learning	96 (TitanX)	3.54	39.8	24.3	-
Hierarchical [67]	EA	300 (P100)	3.75	15.7	20.3	64
NASNet [58]	PPO	2000 (P100)	3.41	3.3	26	5.3
PNASNet [66]	SMBO	250 (P100)	3.41	3.2	25.8	5.1
AmoebaNet [62]	EA	3150 (K40)	3.40	2.6	17.2	86.7
TreeCell [69]	REINFORCE	8.3 (-)	3.64	3.2	25.5	-
DARTS [60]	Gradient	4 (GTX1080Ti)	2.76[2]	3.3	26.7	4.7
	Gradient	- (-)	2.08[2]	5.7	-	-
ProxylessNAS [74]	Gradient	8.3 (-)	-	-	25.8[3]	-
	REINFORCE	8.3 (-)	-	-	25.4[3]	-
OFA [72]	Predictor+EA	50	-	-	20	-

Note Abbreviations: %-top-1 error rate, M-#parameters in million.
[1]Reported in [58].
[2]With cut-out data argumentation.
[3]Latency optimization on mobile phone.

neural network architectures on the training data and measure their accuracy on the validation set. However, it will result in excessive computational cost [45], [58], [62]. This motivates many techniques that aim at speeding up the quality evaluation step.

One possible way of speeding up the evaluation step is to build an accuracy predictor [66], [72]. It takes the architecture of a neural network as the input, and output its validation accuracy. This accuracy predictor is trained on the data set that is collected by training a certain number of neural network architectures in the search space. With a trained accuracy predictor, we can directly use the predicted accuracy to guide the search process, which does not need any extra training cost. Alternatively, the evaluation step can also be accelerated using Hypernetwork [73], which can directly generate weights of a neural network architecture without training it. As such, only a single Hypernetwork needs to be trained, which greatly saves the search cost. Similarly, one-shot NAS methods [59], [60], [74] focus on training a single super-net, from which small subnetworks directly inherit weights without additional training cost. Recently, Cai *et al.* [72] introduced once-for-all (OFA) networks, showing that it is possible to train a single network that supports all neural network architectures in the search space while maintaining the same level of accuracy as the independent training.

Table 4 shows typical NAS advances in recent years. By restricting the search space and generating architectures progressively, NAS architectures can surpass the best human designs in both accuracy and execution efficiency [reflected by the number of parameters/MACs, (see Fig. 10)]. The searching time is also decreased significantly from 22 400 to four GPU days, making it a universal and powerful tool for researchers with restricted resources. However, the autodesigned NAS architectures mostly have complex topology and inscrutable combinations of fragmentations, which sometimes aggravate the efforts for the implementation of embedded devices. Usually, different hardware platforms have different properties (e.g., parallelism, buffer size, etc.). Using the same neural network model for all hardware platforms will result in suboptimal execution performance. Recent hardware-aware NAS approaches start to tackle this challenge by incorporating hardware feedback into the search process, such as ProxylessNAS [74]. These approaches allow designing specialized models that better fit the hardware, but require to repeat the architecture search and model training process. The design cost grows linearly as the number of target platforms increases, making them unable to handle the vast amount of devices in reality (e.g., billions of Internet-of-Things devices). This problem is addressed in [72] by introducing techniques to train a single network (i.e., OFA network), which can be used under diverse architectural configurations, thereby amortizing the total training cost. Future improvements may rely on the understanding and exploitation of principles and theories behind these models, and then derive and leverage more human priors to guide the exploration.

C. Tensor Decomposition

Tensor (including matrix) operation is the basic computation in neural networks. Therefore, the compression of tensors is a promising way to shrink and accelerate neural network models. In this section, we introduce a series of research studies on tensor decomposition and practices for neural network compression, from simple to complex. In the end, we provide a summary of the characteristics of different decomposition methods and discuss future trends.

1) Low-Rank Matrix Decomposition:

a) Full-rank decomposition: For any given matrix $\boldsymbol{A} \in \mathbb{R}^{m\times n}$ with rank $r \leqslant \min\{m,n\}$, the full-rank decomposition of $\boldsymbol{A}$ can be represented as $\boldsymbol{A} = \boldsymbol{W}\boldsymbol{H}$, where $\boldsymbol{W} \in \mathbb{R}^{m\times r}$ and $\boldsymbol{H} \in \mathbb{R}^{r\times n}$. The spatial complexity can be significantly reduced from $\mathcal{O}(mn)$ to $\mathcal{O}(r(m+n))$ if r is much smaller than m or n. But this benefit cannot always hold, especially when m and n are close and the original matrix is a row or column full rank. One can give a positive integer $k < r$, and then an optimization problem to find $\boldsymbol{W} \in \mathbb{R}^{m\times k}$ and $\boldsymbol{H} \in \mathbb{R}^{k\times n}$ could be formulated as

$$\min_{\boldsymbol{W},\,\boldsymbol{H}} \frac{1}{2}\|\boldsymbol{A} - \boldsymbol{W}\boldsymbol{H}\|_F^2 \tag{17}$$

where F means the Frobenius norm. Nevertheless, such solution incurs approximation error especially when k is small.

b) Singular value decomposition (SVD): SVD [75] may be the best low-rank approximation of a matrix because its most information can be described by the singular values. For any given matrix $\boldsymbol{A} \in \mathbb{R}^{m\times n}$, one can find

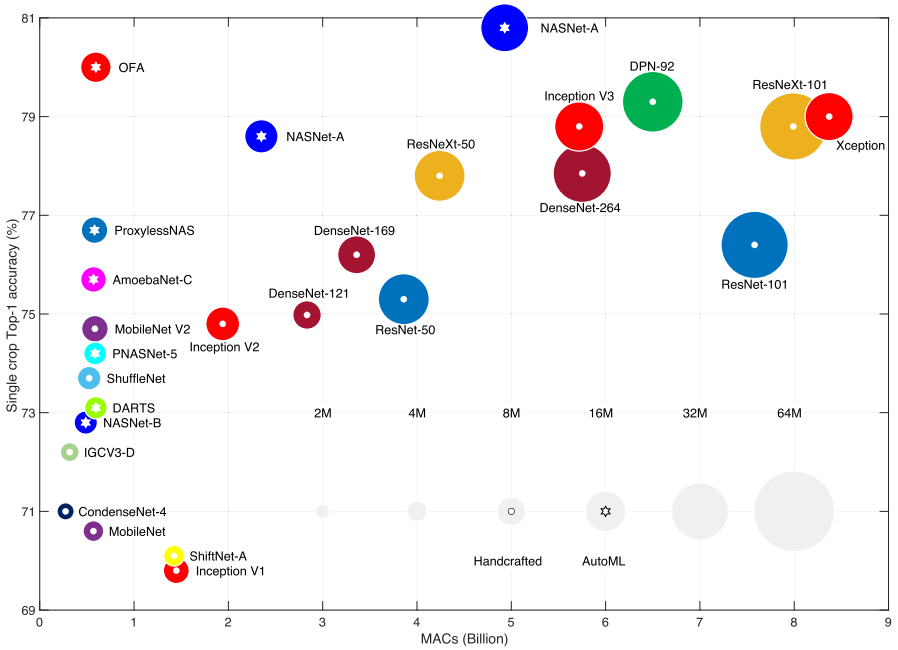


Fig. 10. ***Accuracy results of various CNN models on ImageNet with the number of parameters and MACs. Inspired by Canziani et al. [71].***

matrices satisfying

$$\boldsymbol{A} = \boldsymbol{U}'\boldsymbol{S}'\boldsymbol{V}'^{\mathrm{T}} \tag{18}$$

where $\boldsymbol{U}' \in \mathbb{R}^{m\times r}$ and $\boldsymbol{V}'^{\mathrm{T}} \in \mathbb{R}^{r\times n}$ are orthogonal matrices and $\boldsymbol{S}' \in \mathbb{R}^{r\times r}$ is a diagonal matrix with only singular values of $\boldsymbol{A}$ on the diagonal. The spatial complexity can be reduced from $\mathcal{O}(mn)$ to $\mathcal{O}(r(m+n+1))$. Similarly, one can replace r with a smaller k, which is referred as truncated SVD.

Full-rank decomposition is an effective method for FC layer-based networks, especially between two layers with largely different amounts of neurons [76] or a sparse weight matrix with intrinsic low rank [77]. Since the easy-to-use characteristic, it has been extended to CNNs although the Conv weight kernel becomes a 4-D tensor rather than a 2-D matrix [78]. Whereas, this naive approach cannot achieve satisfactory accuracy in practice as pointed out in [79]. Compared to the full-rank decomposition, SVD can obtain less error benefited from the intrinsic powerful representation capability of singular values. Li and Park [80] and Xue *et al.* [81] utilized a simplified SVD to replace the original weight matrix, both of which have reduced the spatial complexity. Huynh and Won [82] proposed a new training method to achieve acceleration, which is compatible with the SVD format for networks with a single hidden layer. Masana *et al.* [83] used SVD to decompose the product of the input and the weight matrix. Furthermore, SVD can also be used to compress the input data to simplify and accelerate the training circuitously [84]–[86]. The model accuracy sometimes even increases after removing the data redundancy [86] because the singular values and orthogonal bases may highlight potential inner relationships within the data [87].

Beyond using SVD to directly replace the weight matrix, more flexible reconstruction of neural networks can be achieved by leveraging the SVD characteristics. For example, Zhang *et al.* [88], [89] proposed a new structure based on SVD that splits one Conv weight kernel into two subkernels which is widely known as filter groups. Shim *et al.* [90] utilized SVD to compress the last softmax layer for large vocabulary neural networks. In the scenario of distributed training, Yu *et al.* [91] utilized principal component analysis (PCA) to linearly project the weight gradients into a low-dimensional space that enables fast decentralized gradient aggregation (e.g., ring all-reduce) in the compressed domain.

c) Other matrix decomposition: Except for the full-rank decomposition and SVD, there are also some other matrix decomposition methods such as QR and CUR [92]. For QR, it usually performs faster than SVD but with more accuracy loss [85]. Additionally, using complex number shows potential to improve the expression of neural networks in recent years [93], so the explanation of a complex neuron with multiple inputs and a single output based on the QR decomposition [94] deserves more attention. For CUR, $\boldsymbol{C}$ and $\boldsymbol{R}$ matrices are sparse rather than the orthogonal matrices of $\boldsymbol{U}'$ and $\boldsymbol{V}'^{\mathrm{T}}$ in SVD. Therefore, it can often achieve a better compression ratio. A number of research studies utilize the CUR decomposition [95] for machine-learning applications, especially a variant similar to CUR called Nyström method [96]. However, the accuracy loss still needs further improvement.

In a nutshell, SVD is the best matrix decomposition method to compress neural networks with overall high

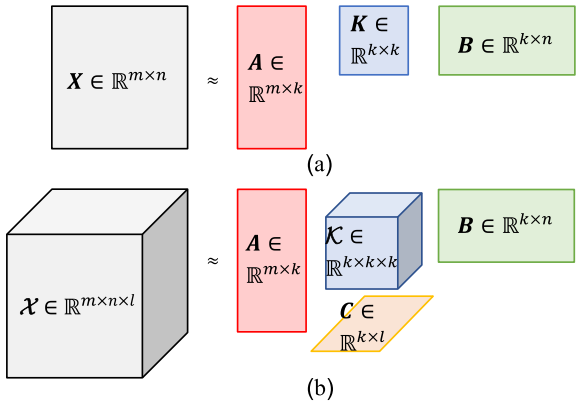


Fig. 11. ***Analogy between low-rank matrix decomposition and tensor decomposition. (a) Low-rank matrix decomposition with $k \times k$ kernel. (b) Low-rank tensor decomposition with $k \times k \times k$ kernel.***

quality. Although the full-rank decomposition suffers from worse accuracy than SVD, this naive approach achieves faster execution speed so that it is also useful for deep learning acceleration. There are still insufficient practices in DNNs to evidence the effectiveness of the QR or CUR decomposition.

2) Tensorized Decomposition: Although the low-rank matrix decomposition can optimize both the spatial and computational complexity of neural networks, however, the plane view of a matrix limits the potential for extreme compression. In tensorial language, a 2-D matrix is just a second-order tensor. If the number of orders or the value of modes within each order of a tensor is large, more flexible algorithms can then be considered to achieve an extremely high compression ratio. In this part, we discuss the continuation of decomposition from matrix to tensor and give their applications in neural networks. From now on, in order to distinguish tensors from matrices, the matrix symbol (e.g., $\boldsymbol{A}$) will be modified to (e.g., $\mathcal{A}$) for tensor notation.

a) Tucker and classical prolongation (CP): Recalling the aforementioned truncated SVD [see Fig. 11(a)], it has a $k \times k$ kernel matrix. The truncated rank k and the consequent approximation error are all determined by the kernel matrix. This format can also be extended to tensor. Here, we first introduce an operation between a tensor and a matrix called mode-i contracted product [97]. For a dth-order tensor $\mathcal{A} \in \mathbb{R}^{n_1\times n_2\times\ldots\times n_d}$ and a matrix $\boldsymbol{B} \in \mathbb{R}^{n_i\times m}$ $(i \in \{1, 2, \ldots, d\})$, the product between $\mathcal{A}$ and $\boldsymbol{B}$ is $\mathcal{R} = \mathcal{A} \times_i \boldsymbol{B}$, where $\mathcal{R}$ is also a dth-order tensor of $\mathbb{R}^{n_1\times\ldots\times n_{i-1}\times m\times n_{i+1}\times\ldots\times n_d}$. Given this preliminary, a dth-order tensor $\mathcal{X} \in \mathbb{R}^{n_1\times n_2\times\ldots n_d}$ can be decomposed into one kernel tensor $\mathcal{K} \in \mathbb{R}^{r_1\times r_2\times\ldots\times r_d}$ and d factor matrices $\boldsymbol{F}^{(i)} \in \mathbb{R}^{r_i\times n_i}$ $(i \in \{1, 2, \ldots, d\})$, where r_i is called the i-rank that is actually the rank of the matrix produced by reshaping the tensor into $\boldsymbol{X}^{(i)} \in \mathbb{R}^{n_i\times n_1\ldots n_{i-1}n_{i+1}\ldots n_d}$. The kernel tensor has the same number of orders as the original tensor, that is, d. By utilizing the mode-i contracted product continuously, $\mathcal{X}$ can be rewritten as

$$\mathcal{X} \approx \mathcal{K} \times_1 \boldsymbol{F}^{(1)} \times_2 \boldsymbol{F}^{(2)} \times_3 \ldots \times_d \boldsymbol{F}^{(d)}. \tag{19}$$

Fig. 11(b) illustrates the case of $d = 3$. The decomposition format in (19) is called Tucker decomposition [98]. When every r_i equals a positive integer r_C and the kernel tensor $\mathcal{K}$ presents like a superdiagonal tensor, which means all elements in $\mathcal{K}$ are 0 except $\mathcal{K}(x_1, x_2, \ldots, x_d)$ with $x_1 = x_2 = \ldots = x_d$, then (19) becomes CP decomposition [98]. It seems that CP decomposition should achieve better compression performance via the superdiagonal kernel tensor; however, the CP rank r_C may not be bounded by r_i in reality [99]. This situation usually results in a large r_C value. Hence, CP is typically used for interpreting data components while Tucker is often used for data compression [99].

Many researchers have applied Tucker, CP, or some other modified decomposition formats to neural networks in recent years. Kim *et al.* [100] used Tucker-2 decomposition to compress Conv weight kernels and they further deploy the compressed CNNs in cellphone. Instead of directly replacing weights with decomposed tensors, Chien and Bao [101] considered the entire decomposition process of (19) as a connection, similar to [102] and [103]. Specifically, the input tensor $\mathcal{X}$ is treated as the input layer, the kernel tensor $\mathcal{K}$ is treated as the output layer, and the weights are replaced with sequential mode-i contracted products among pseudo-inverse factor matrices of $\boldsymbol{F}^{(i)}$, then the model accuracy would be improved. Janzamin *et al.* [104] used CP decomposition instead of the traditional BP to solve the nonconvex problem of training neural networks with one hidden layer. Lebedev *et al.* [105] and Astrid and Lee [106] leveraged CP decomposition to compress Conv weight kernels into several subkernels to reduce the number of parameters and speed up the running. Chen *et al.* [107] introduced a collective residual unit (CRU) based on block term decomposition (BTD), which is a combination of Tucker and CP, to enhance the utilization of parameters in residual CNNs. Tran *et al.* [108] proposed a new tensor operation based on the mode-i contracted product to replace the normal Conv to reduce both the spatial and computational complexity. Schütt *et al.* [109] used tensorized neural networks with low rank to learn molecular energies. Zhou *et al.* [110] conversely used neural networks to learn an appropriate CP rank. Recently, Oymak and Soltanolkotabi [111] rewrote the nonoverlapping Conv kernel and its input into the rank-1 CP format to improve the training efficiency. Ye *et al.* [112] applied BTD to achieve the highest accuracy on RNNs over the UCF11 data set compared with other methods.

b) Higher order SVD (HOSVD): In the earlier texts, we have concluded that SVD might have the best performance in matrix decomposition. Similarly, extending to the high order space, HOSVD becomes a suitable candidate for PCA. It is also called multilinear SVD (MLSVD) [113], because it is a special form of Tucker decomposition that all the factor matrices $\boldsymbol{F}^{(i)}$ in (19) are orthogonal and thus the kernel tensor $\mathcal{K}$ is called all-orthogonal. For the tensors that have extremely large scale, calculating the factor matrices $\boldsymbol{F}^{(i)}$ from $\boldsymbol{X}^{(i)}$ under HOSVD will be very

expensive in both space and time. This situation makes it difficult to control the tradeoff between the approximation error and the truncation degree. Thus, the applications in neural networks are rare, only for pilot [114] or small-scale [115] ones.

c) Tensorizing and curse of dimensionality: Since high-order tensors can usually help achieve better decomposition results, an intuitive idea comes out to convert the matrix to a high-order tensor before conducting decomposition. Consider a large matrix $\boldsymbol{W} \in \mathbb{R}^{M\times N}$ and each number of its modes can be factorized into d integers: $M = m_1 \times m_2 \times \ldots \times m_d$ and $N = n_1 \times n_2 \times \ldots \times n_d$. Then a dth-order tensor $\boldsymbol{\mathcal{W}} \in \mathbb{R}^{m_1 n_1 \times m_2 n_2 \times \ldots \times m_d n_d}$ can be constructed by two bijections from the original matrix mode M and N to d tensor modes m_i and n_i ($i \in \{1, 2, \ldots, d\}$), respectively. This process is called tensorizing [4]. We denote the maximum value of modes in tensor $\boldsymbol{\mathcal{W}}$ as mn and the maximum value of i-rank as r, usually $r \ll mn$. Then the spatial complexity of the original matrix $\boldsymbol{W}$ could be reduced from $\mathcal{O}((mn)^d)$ to $\mathcal{O}(dmnr + r^d)$ if we use the common Tucker decomposition. However, the kernel tensor still gives an exponential contribution (i.e., r^d) to the spatial complexity. Thus, the curse of dimensionality as d grows cannot be solved completely by either normal Tucker or HOSVD. Fortunately, a tensor network [97], [99], [116] that represents a tensor with a link of matrices or low-order tensors with contracted products, is promising to avoid the curse of dimensionality by eliminating the high-order kernel tensor. To this end, hierarchical tucker (HT) and tensor train (TT) are proposed as two well-known formats for tensor networks.

d) HT and dimension tree: In order to explain the HT format, we first introduce the concept of t-modes matricization [117] that is different from the mode-i matricization [98] mentioned earlier. Consider a dth-order tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{n_1\times n_2\times\ldots\times n_d}$ with the set of indexes of modes $u = \{1, 2, \ldots, d\}$ which can be split into two subsets $t = \{t_1, t_2, \ldots, t_k\}$ and $s = \{s_1, s_2, \ldots, s_{d-k}\}$ ($u = t \cup s$). If the set t can be further split into $t = t_l \cup t_r$, we can conclude "$\text{span}(\boldsymbol{A}^{(t)}) \subset \text{span}(\boldsymbol{A}^{(t_l)} \otimes \boldsymbol{A}^{(t_r)})$" where $\boldsymbol{A}^{(t)} \in \mathbb{R}^{n_{t_1}\ldots n_{t_k}\times n_{s_1} n_{s_2}\ldots n_{s_{d-k}}}$ and $\otimes$ means the Kronecker product. Similarly, the size of $\boldsymbol{A}^{(t_l)}$ is determined by the serial products of elements in t_l and $u \setminus t_l$, respectively, and so does $\boldsymbol{A}^{(t_r)}$. Such a format like $\boldsymbol{A}^{(t)}$ is called t-modes matricization of tensor $\boldsymbol{\mathcal{A}}$. Furthermore, given $\boldsymbol{U}_t$, $\boldsymbol{U}_{t_l}$, and $\boldsymbol{U}_{t_r}$ as bases of the column spaces of $\boldsymbol{A}^{(t)}$, $\boldsymbol{A}^{(t_l)}$, and $\boldsymbol{A}^{(t_r)}$, respectively, we have

$$\boldsymbol{U}_t = (\boldsymbol{U}_{t_l} \otimes \boldsymbol{U}_{t_r})\boldsymbol{B}_t \tag{20}$$

where $\boldsymbol{U}_t \in \mathbb{R}^{n_{t_1} n_{t_2}\ldots n_{t_k}\times r_t}$, and $\boldsymbol{B}_t \in \mathbb{R}^{r_{t_l} r_{t_r}\times r_t}$ is called transfer matrix. Note that r_t, r_{t_l}, and r_{t_r} are the ranks of $\boldsymbol{A}^{(t)}$, $\boldsymbol{A}^{(t_l)}$, and $\boldsymbol{A}^{(t_r)}$, respectively.

Particularly, in the case of $t = u$, $\boldsymbol{A}^{(t)} \in \mathbb{R}^{n_1 n_2\ldots n_d\times 1}$ will be stretched as a vector and we have $\boldsymbol{U}_t = \boldsymbol{A}^{(t)}$ because $r_t = 1$. Suppose we begin to utilize (20) with $t = u$ to reform $\boldsymbol{\mathcal{A}}$, then iteratively decompose $\boldsymbol{U}_{t_l}$ and $\boldsymbol{U}_{t_r}$ until every subsequent base matrix $\boldsymbol{U}_x$ has only one mode value. Such a final format that equals the vectorization of the original tensor $\boldsymbol{\mathcal{A}}$, is called HT format. We should note that the HT format of a given tensor is not unique because there are multiple choices of $t = t_l \cup t_r$. Fig. 12(a) presents a fourth-order example of the dimension tree during decomposing $\boldsymbol{\mathcal{X}}$ into its HT format. It can be observed that the set of modes t is split into only two subsets every time using (20), thus a specific HT format corresponds to a binary dimension tree.

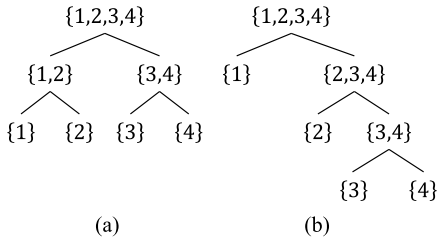


Fig. 12. ***Different dimension trees of tensor $\mathcal{X} \in \mathbb{R}^{n_1\times n_2\times n_3\times n_4}$. (a) Canonical dimension tree for the HT format. (b) Degenerate dimension tree for the TT format.***

Although the HT format has flexible organization forms (because the count of different dimension trees with a fixed number of nodes is distributed as Catalan numbers [125]), it suffers from an intractable obstacle to obtain an optimal specific form. As a result, just a few practices of HT can be found [126]–[128]. Among them, only Cohen and Shashua [128] and Cohen *et al.* [129] showed neural network applications, but they are yet to answer how to obtain an optimal HT form.

e) Tensor train (TT): According to [130], the TT format of a dth-order tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{n_1\times n_2\times\ldots\times n_d}$ can be represented as $\mathcal{A}(j_1, j_2, \ldots, j_d) = \boldsymbol{G}_1[j_1]\boldsymbol{G}_2[j_2]\ldots\boldsymbol{G}_d[j_d]$, where $\boldsymbol{G}_k[j_k]$ is a factor matrix with a size of $r_{k-1} \times r_k$ ($r_0 = r_d = 1$) and $j_k \in \{1, 2, \ldots, n_k\}$ ($k \in \{1, 2, \ldots, d\}$). The set of r_k is collectively called TT ranks. Moreover, all $\boldsymbol{G}_k[j_k]$ belonging to the same mode can be stacked into a third-order core tensor $\boldsymbol{\mathcal{G}}_k \in \mathbb{R}^{n_k\times r_{k-1}\times r_k}$. Thus, the TT format of a tensor $\boldsymbol{\mathcal{A}}$ can be rewritten as

$$\boldsymbol{\mathcal{A}} = \boldsymbol{\mathcal{G}}_1 \times^1 \boldsymbol{\mathcal{G}}_2 \times^1 \ldots \times^1 \boldsymbol{\mathcal{G}}_d \tag{21}$$

where $\times^1$ is similar to mode-i contracted product and is called mode-$(N, 1)$ contracted product here [97]. $\boldsymbol{\mathcal{X}} \times^1 \boldsymbol{\mathcal{Y}}$ means one and only one pair of equal modes in any Nth-order tensor $\boldsymbol{\mathcal{X}}$ and Mth-order tensor $\boldsymbol{\mathcal{Y}}$ will be contracted to produce a new $(N + M - 2)$th-order tensor. Apparently, according to the equation above, the spatial complexity of the TT format is $\mathcal{O}(dnr^2)$ where n is the maximum value of modes, and r is the maximum value of TT ranks. Fig. 13(a) shows the structure of TT in a tensor network graph, where each node represents a core tensor and each edge is a mode of its connected tensors.

f) HT versus TT: It is widely accepted that TT is a special form of HT [131]–[134]. In particular, for a dth-order tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{n_1\times n_2\times\ldots\times n_d}$ with the set of indexes of

Table 5 TT Decomposition for CNNs

Reference	Format	Compressed Parts	Dataset	Compression Ratio	Accuracy Loss
A. Novikov et al. (2015) [4]	TT	FC	CIFAR10	11.9×	1.26%
Q. Zhao et al. (2019) [118]	TC	FC	CIFAR10	444× / 1300×	0.13% / 2.18%
H. Huang et al. (2018) [119]	TT	FC	MINST	14.85×	1.5%
J. Su et al. (2018) [120]	TT	FC	MINST	500×	2%
T. Garipov et al. (2016) [121]	TT	Conv & FC	CIFAR10	82.87×	1.1%

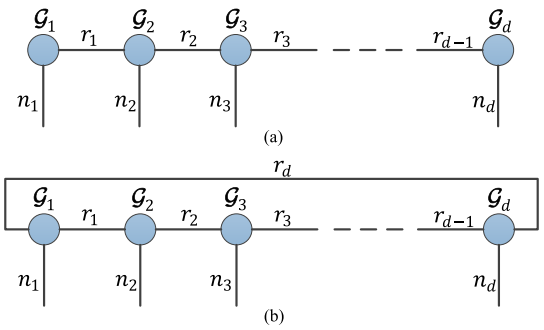


Fig. 13. *Tensor network graph for (a) TT or (b) TC format of a dth-order tensor* $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times \ldots \times n_d}$.

modes $u = \{1, 2, \ldots, d\}$, if we split out only the first mode every time until the dimension tree is constructed, the generated HT format is equivalent to the TT format, as shown in Fig. 12(b). The ranks of TT cannot be flexibly selected since using the 1-mode matricization produces a fixed form. However, the TT format is much easier to program so that it is more practical to be applied in neural network compression at this stage. By tensorizing the weight matrices first, the TT format with small core tensors can be easily achieved, which is known as quantized TT (QTT) [135] with lower ranks and higher compression ratios.

g) TT variant: Tensor chain (TC): The TC format [136] (also called tensor ring [118]) is a variant of TT, which has a comparable approximation capability. The only difference in TC is $r_0 = r_d \neq 1$. Thus, compared to (21), there are two pairs of equal modes being contracted in the end like

$$\boldsymbol{\mathcal{A}} = (\boldsymbol{\mathcal{G}}_1 \times^1 \boldsymbol{\mathcal{G}}_2 \times^1 \ldots \times^1 \boldsymbol{\mathcal{G}}_{d-1}) \times^2 \boldsymbol{\mathcal{G}}_d \tag{22}$$

where $(\boldsymbol{\mathcal{G}}_1 \times^1 \boldsymbol{\mathcal{G}}_2 \times^1 \ldots \times^1 \boldsymbol{\mathcal{G}}_{d-1}) \in \mathbb{R}^{n_1 \times n_2 \times \ldots \times n_{d-1} \times r_0 \times r_{d-1}}$ and $\boldsymbol{\mathcal{G}}_d \in \mathbb{R}^{n_d \times r_{d-1} \times r_d}$. Fig. 13(b) shows the tensor network graph of TC. Although TC seems to have a greater potential to obtain better quality than TT [118], [137], more research studies are needed to evidence this point in neural networks.

h) TT in neural networks: In tensor network decomposition, the TT format is widely applied in neural networks. In CNNs, FC layers often occupy the most memory cost, whereas Conv layers contribute most to the final accuracy. Therefore, most research studies pay attention to compress FC layers, whereas the Conv layers remain unchanged [4], [118]–[120]. Table 5 enumerates recent TT compression on CNNs. Note that only Garipov *et al.* [121] extended tensorizing to Conv. Besides, the compression ratios here are just listed to evidence the compression capability of TT rather than for lateral comparison since the practical compression ratio relies heavily on the model scale. Compared to CNNs, the works on RNNs [122]–[124], [138] are fewer perhaps due to the complicated gate structure of LSTM/GRU and the less popularity and standardization of RNNs in current AI tasks when compared to CNNs. Table 6 enumerates recent works. From (2), there are two kinds of weight matrices ($\boldsymbol{W}_x$ and $\boldsymbol{W}_h$) that correspond to the multiplication with inputs ($\boldsymbol{X}$) and hidden states ($\boldsymbol{h}$), respectively. Note that in [124], only $\boldsymbol{W}_x$ is compressed and an extreme compression ratio is demonstrated along with miraculous accuracy improvement. It seems that the performance of TT decomposition on RNNs is better than that on CNNs. This difference might rely on the structure of the Conv kernel which is already a compact fourth-order tensor rather than a large matrix. Hence, the compression degree is restricted even if using the tensorizing approaches [121], and the extracted features may be disorganized.

Table 6 TT Decomposition for RNNs

Reference	RNN Model	Compressed Parts	Dataset	Compression Ratio	Accuracy Loss
A. Tjandra et al. (2017) [122]	GRU	$\boldsymbol{W}_x$ & $\boldsymbol{W}_h$	Sequential MNIST	43.52× / 69.80×	0.3% / 0.3%
A. Tjandra et al. (2018) [123]	GRU	$\boldsymbol{W}_x$ & $\boldsymbol{W}_h$	Nottingham	139.82× / 439.43×	0.28% / 1.61%
Y. Yang et al. (2017) [124]	LSTM	$\boldsymbol{W}_x$	UCF11 / Hollywood2	17554.3× / 23158.8×	-30.4% / -43.8%
	GRU			13687.1× / 18313×	-32.5% / -28.8%

Table 7 Comparison of the Spatial Complexity of Tensor Decomposition Methods

Methods	Spatial Complexity	Remark
–	$\mathcal{O}(MN)$	Original weight matrix
Full Rank	$\mathcal{O}(r(M+N))$ $\mathcal{O}(k(M+N))$	r: rank of matrix k: truncated rank
SVD	$\mathcal{O}(r(M+N+1))$ $\mathcal{O}(k(M+N+1))$	–
QR	$\mathcal{O}(k(M+N))$	–
CUR	$\mathcal{O}(r(M+N+r))$ $\mathcal{O}(k(M+N+k))$	–
CP	$\mathcal{O}(r_C(dmn+1))$	r_C: CP rank
Tucker HOSVD	$\mathcal{O}(dmnr_T + r_T^d)$	r_T: maximum i-rank
HT	$\mathcal{O}(dmnr_{HT} + (d-1)r_{HT}^3)$	r_{HT}: maximum HT rank
TT TC	$\mathcal{O}(dmnr_{TT}^2)$	r_{TT}: maximum TT rank

3) Discussion: Fig. A1 in Appendix A illustrates the vein with concise property descriptions for each specific decomposition method and Table 7 further lists their spatial complexity, on the basis of which we give several conclusions and discussions as follows.

1) Tensor decomposition is also a special case of topology organization like that used in designing compact models (Section III-B). Whereas, they have distinct goals as mentioned in Section III-A. The design of compact models finds the expected base model itself from a pool of candidates with distinct topologies, whereas tensor decomposition modifies a given base neural network model.
2) For Tucker and CP, the combination of them especially BTD [107], [112] presents promising potential in neural network compression.
3) Tensor networks including HT, TT, and TC are the most effective decomposition formats according to their capability to break the curse of dimensionality and the recent practices. TC can achieve better accuracy than TT under the same compression ratio, thus TC deserves more attention in the future. For HT, there are only few results due to its complex nonuniqueness. It is still interesting to answer whether HT is effective for neural network compression, because HT has more theoretical potential to attain lower spatial complexity than others.
4) Current decomposition methods are more suitable for FC layers rather than Conv layers, since FC layers usually have more redundancy and the mode values of the Conv weight tensor are usually imbalanced. The effectiveness of tensor decomposition on Conv layers needs further evidence.
5) Most of existing tensor decomposition works only conduct experiments for relatively small-scale tasks, which are not enough to prove their true effectiveness. More practices on large-scale tasks (e.g., on ImageNet [139]) are encouraged.
6) In fact, tensorizing is still a matrix-oriented method, even if in the case of converting the fourth-order Conv kernel to a matrix first before decomposition [121]. As the dimension of data increases continuously, Conv kernels might have higher orders such as in 3DCNN [140]. Therefore, more general decomposition methods without tensorizing should be developed to compress neural networks but guarantee accuracy.

D. Data Quantization

Quantization attempts to reduce the bitwidth of the data flowing through a neural network model, thus it is possible to shrink the model size for memory saving and simplify the operations for compute acceleration.

1) Quantization Data Object: Fig. 14 illustrates all the data objects that can be quantized in a neural network, involving weight (W), activation (A), error (E), gradient (G), and weight update (U). Here "E" dedicatedly denotes the activation gradient in (5) during BP, while "G" is specialized for the parameter gradient in (6). The weight update means the gradient accumulation as shown in (7). Because the bias just participates in a small amount of operations, very few works discuss it as we do. Moreover, the accumulation in the MAC operation often remains in full precision because only few works touch this point.

Usually, there are two approach categories distinguished by quantization data objects. The dominant one saves a high-precision (e.g., FP32: 32-bit floating point) master weight, as shown in Fig. 14(a). In both the forward and backward passes, the high-precision master weight is quantized to K_W-bit fixed point data before neural computation. The activation and error are quantized to K_A and K_E bits in the forward and backward passes, respectively. Then we can obtain the weight gradient based on the quantized activation and error, quantize it to K_G bits, and accumulate it onto the master weight. On the

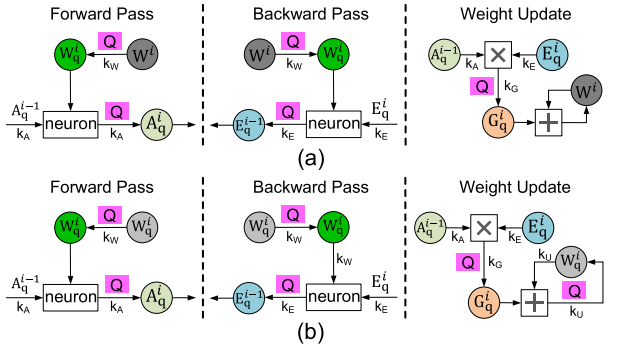


Fig. 14. *Quantization data objects. (a) With high-precision master weight. (b) Without high-precision master weight. Subscript "q" denotes quantized data and "Q" box denotes quantization.*

Table 8 Representative Quantization Works for CNNs

Reference	Object	Problem Formulation	Level Distribution	Level Projection	Batch Normalization
BinaryConnect (2015) [150]	W	Heuristic	Uniform	Deterministic/Stochastic	–
QBP1 (2015) [156]	W/A	Heuristic	Uniform (W)/Non-uniform (A)	Stochastic (W)/Deterministic (A)	–
BNN (2016/2017) [5, 152]	W/A	Heuristic	Uniform	Deterministic/Stochastic	Vanilla/Shift BN
XNOR-Net (2016) [157]	W/A	Optimization	Uniform	Deterministic	–
BinaryNet (2017) [158]	W/A	Heuristic	Uniform (W)/Non-uniform (A)	Deterministic	–
TrueNorth (2016) [159]	W/A	Heuristic	Uniform	Deterministic	–
HWGQ (2017) [160]	W/A	Heuristic	Uniform/Non-uniform	Deterministic	–
TWN (2016) [161]	W	Optimization	Uniform	Deterministic	–
TTQ (2016) [162]	W	Optimization	Non-uniform	Deterministic	–
VNQ (2018) [163]	W	Optimization	Uniform	Deterministic	–
BinaryRelax (2018) [164]	W	Optimization	Uniform	Deterministic	–
ADMM (2018) [165]	W	Optimization	Non-uniform	Deterministic	–
TBN (2018) [166]	W/A	Optimization	Uniform	Deterministic	–
ABC-Net (2017) [142]	W/A	Optimization	Non-uniform	Deterministic	–
LQ-Nets (2018) [143]	W/A	Optimization	Non-uniform	Deterministic	–
QIL (2019) [146]	W/A	Heuristic	Uniform/Non-uniform	Deterministic	–
TernGrad (2017) [167]	G	Heuristic	Uniform	Stochastic	–
TQN (2017) [168]	W/U	Heuristic	Uniform	Deterministic/Stochastic	–
GXNOR-Net (2018) [151]	W/A/U	Heuristic	Uniform	Stochastic (W)/Deterministic (A)	–
DoReFa-Net (2016) [147]	W/A/E	Heuristic	Uniform	Deterministic	–
Balanced DoReFa-Net (2017) [169]	W/A	Heuristic	Uniform	Deterministic	–
WRPN (2017) [170]	W/A	Heuristic	Uniform	Deterministic	–
Sketching (2017) [171]	W	Optimization	Non-uniform	Deterministic	–
INQ (2017) [172]	W	Heuristic	Non-uniform	Deterministic	–
Regularization (2018) [173]	W/A	Heuristic	Uniform	Deterministic	–
Group-Net (2018) [174]	W/A	Heuristic	Uniform	Deterministic	–
UNIQ (2018) [175]	W/A	Heuristic	Non-uniform	Deterministic	–
FAQ (2018) [149]	W/A	Heuristic	Uniform	Deterministic	–
HAQ (2019) [176]	W/A	Heuristic & RL	Uniform	Deterministic	–
MP (2017) [177]	W/A/E	Heuristic	Uniform	Deterministic	–
Flexpoint (2017) [178]	W/A/G	Heuristic	Uniform	Deterministic	–
MP-INT (2018) [179]	W/A/E	Heuristic	Uniform	Deterministic	–
QBP2 (2018) [180]	W/A/E	Heuristic	Uniform	Deterministic	Quantized range BN
DST (2018) [181]	W/U	Heuristic	Uniform	Stochastic	–
WAGE (2018) [182]	W/A/E/G/U	Heuristic	Uniform	Deterministic (W/A/E/U)/Stochastic (G)	Linear scaling
FX Training (2018) [183]	W/A/E/G/U	Heuristic	Uniform	Deterministic	–
8b Training (2018) [184]	W/A/E/G/U	Heuristic	Uniform	Deterministic/Stochastic	–

Table 9 Representative Quantization Works for RNNs

Reference	Model	Object	Problem Formulation	Level Distribution	Level Projection
TernaryConnect (2016) [188]	Vanilla RNN/LSTM/GRU	W	Heuristic	Uniform/Non-uniform	Deterministic/Stochastic
BNN (2017) [152]	Vanilla RNN/LSTM	W/A	Heuristic	Uniform	Deterministic
Balanced DoReFa-Net (2016/2017) [153, 169]	LSTM/GRU	W/A	Heuristic	Uniform	Deterministic
Neuron Increase (2017) [190]	LSTM	W/A	Heuristic	Uniform	Deterministic
MP (2017) [177]	LSTM/GRU	W/A/E	Heuristic	Uniform	Deterministic
HitNet (2018) [191]	LSTM/GRU	W/A	Heuristic	Uniform	Deterministic/Probabilistic
Alternating (2018) [144]	LSTM/GRU	W/A	Optimization	Non-uniform	Deterministic

other side, a few works aggressively quantize the master weight as well as in Fig. 14(b). The difference is that the accumulated weight is also quantized. We term it as update quantization and denote the bitwidth of update as K_U. In summary, we can classify the quantization objects into three levels: 1) parameter (W); 2) propagation data (A, E); and 3) gradient and update (G, U).

2) Quantization Philosophy: Different methods have been proposed to quantize the above data objects, and we list the typical ones in Tables 8 and 9 for CNNs and RNNs, respectively. From the perspectives of quantization data object (W/A/E/G/U), problem formulation (heuristic/optimization problem), distribution of discrete levels (uniform/nonuniform), and the level projection

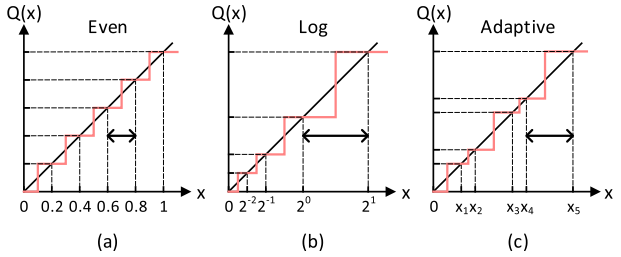


Fig. 15. ***Distribution of discrete levels. (a) Uniform distribution with even step length. Nonuniform distribution with (b) logarithmic or (c) adaptive step length.***

(deterministic/stochastic), these methods can be coarsely classified. Recently, the expensive batch normalization (BN) begins to attract attention, so we also consider it in our collection. We first briefly introduce these classification criteria, and then go through prior work in detail.

a) Problem formulation: Heuristic methods often intuitively project data to the nearest discrete level. This early basic projection form is as follows:

$$Q(x) = \Delta \cdot \text{round}\left(\frac{x}{\Delta}\right) \tag{23}$$

where x is the original high-precision value in a continuous space, $Q(x)$ is the quantized data in a discrete space, round$(\cdot)$ is the rounding function, and Δ is the quantization step length if the discrete levels obey a uniform distribution. If we suppose $x \in [0,\ 1]$ and a k-bit quantization (Q^k), we usually have $\Delta = (1/2^k - 1)$ to produce 2^k discrete levels, or $\Delta = (\max(x)/2^k - 1)$ to fully utilize the dynamic range. The above quantization technique looks like the function of analog-to-digital converters (ADCs) in signal processing systems [141]. Besides the basic quantization form in (23), the quantizer itself can also be learned with adaptive dynamic range and quantization levels for different layers [142]–[146].

Another category formulates the quantization as an optimization problem and tries to analytically or approximately solve it. The classic model can be generally governed by

$$\min_{Q} \|\boldsymbol{X} - Q(\boldsymbol{X})\|_2^2, \quad \text{s.t. } Q_i \in X_Q \quad \text{for all } i \tag{24}$$

where X_Q is a set of discrete levels for data quantization and Q_i is the simplification of $Q(\boldsymbol{X})_i$. Different quantization methods usually design different X_Q and different solvers. In contrast to the element-wise operation in heuristic quantization, the optimization solver here usually considers from a global angle to minimize the overall distance between the original and quantized data.

b) Distribution of discrete levels: In quantization, the data are constrained within a set of discrete levels which can have different distributions. Fig. 15 presents several typical distribution forms. Uniform distribution [147], with even step length between adjacent levels, is the most widely used one. It is easy to implement, while it cannot match the realistic data distribution well. Note that the step length and dynamic range can be fixedly predefined or dynamically determined by data. Nonuniform distribution usually has variable step length that endows the quantization more selectivity for important data regions or a wider dynamic range. The logarithmic distribution [148] is a typical nonuniform quantization with exponential variance on step length. After training, it is able to convert the costly multiplication operation to the simple addition one but the consequent accumulation operation becomes nonstraightforward. In contrast to the regular pattern of the logarithmic one, nonuniform distribution can also have other forms with stronger adaptivity. This often happens when we formulate the quantization as an optimization problem to find a discrete space to approximate the original high-precision one or in the cases that use the mentioned learnable quantizer. All the levels are determined on the fly according to the distribution of real data and the learning target.

c) Level projection: One critical step in quantization is how to project the original high-precision data to a discrete space. There exist two ways for level projection: the deterministic way and the stochastic way. The former projects the high-precision data to the nearest discrete level [147], [149]. In contrast, the latter projection has a possibility to approach one of the two nearby adjacent levels, and the transition probability is determined by the distance from the original data to the nearby discrete levels [150], [151]. In the statistical sense, the stochastic projection is able to achieve better results due to the unbiased estimation.

d) Gradient of quantization function: In fact, the quantization function is nondifferentiable with respect to its input, the gradient of which is presented as delta-spikes at the transition points and slabs over other regions. This is not friendly to the SGD learning framework. Usually, the straight-through estimator (STE) is used to circumvent this problem [147], [152], [153]. It simply bypasses the quantization function during the gradient BP. For example, recalling (23), we approximately have $(\partial L/\partial x) \approx (\partial L/\partial Q)$, that is, $(\partial Q/\partial X) \approx 1$. STE is the mainstream scheme, but not the unique one. Using pulse train to approximate the derivative of discontinuous points [151], designing proper STE formats [154], or gradually rather than immediately closing the gradient BP path to the high-precision data [155] also works.

3) Quantization on CNNs and RNNs:

a) CNN quantization: Table 8 lists existing quantization works for CNNs. All the works quantize W, most works quantize A, and some works begin to quantize E, G, U, or BN. These works involve the aforementioned problem formulation and level distribution/projection. We will detail typical methodologies from these tables to help readers quickly build up a big picture. Note that there

also exist methods with data clustering rather than directly quantizing them to lower bitwidth [185], [186]. Due to the different principles, we do not include them into our quantization scope.

The extensive rise of quantization for neural networks owes to the early binarization [5], [150], [156], [157], [159] and ternarization [161], [162], [187], [188]. Both the heuristic and optimization problem formulations have been proposed at that time. The heuristic binarization [5], [150], [156], [159] can be described as

$$\begin{cases} \text{det.}: \begin{cases} Q(x)=1, & x \geq 0 \\ Q(x)=-1, & \text{otherwise} \end{cases} \\ \text{sto.}: \begin{cases} P(Q(x)=1)=\text{clip}\left(\dfrac{1+x}{2},0,1\right) \\ P(Q(x)=-1)=1-P(Q(x)=1) \end{cases} \end{cases} \tag{25}$$

where $P(\cdot)$ is the transition probability and $clip(\cdot)$ is used to clip the probability range. The binarization can be either deterministic (det.) or stochastic (sto.). Sometimes, the binary levels of $\{-1,\ 1\}$ can be replaced with $\{0,\ 1\}$ [159]. Similarly, the heuristic ternarization [159], [188] can be summarized as

$$\begin{cases} \text{det.}: \begin{cases} Q(x)=1, & x>0.5 \\ Q(x)=-1, & x<-0.5 \\ Q(x)=0, & \text{otherwise} \end{cases} \\ \text{sto.}: \begin{cases} \begin{cases} P(Q'(x)=1)=\text{clip}(|x|,0,1) \\ P(Q'(x)=0)=1-P(Q'(x)=1) \end{cases} \\ Q(x)=\text{sign}(x)\cdot Q'(x) \end{cases} \end{cases} \tag{26}$$

where $\text{sign}(x)=1, x\geq 0; \text{sign}(x)=-1, \text{otherwise}$. The deterministic ternarization also has variants. For example, hysteresis parameters can be introduced to control the level projection by shifting the transition region [151], [159]; pow2-ternarization is also used to ternarize W by introducing the power-of-two quantization for the fractional part [188]. Inspired by these works, Wen *et al.* [167] used the stochastic ternarization for gradient quantization. They additionally introduced a scaling factor $\alpha = \max(|x_i|)$ to tune the data range, in other words, they have $Q(x)\in\{-\alpha,\ 0, \alpha\}$. The extreme gradient compression effectively reduces the volume of communication data and thus accelerating the training in a distributed system. Cai *et al.* [160] maintained the binarization of W while extending the A quantization to multiple bits in the positive region and approximated the quantization function via modified ReLU to help the gradient-based BP. Li *et al.* [181] and Deng *et al.* [151] extrapolated the quantization of both W and A to multiple bits and always constrain W in a discrete space to remove the high-precision master weight.

Different from the heuristic way, the quantization can also be formulated as an optimization problem similar to (24) [157]. For binarization, the problem becomes

$$\min_{\alpha\in R^+,\ Q^B} \|\boldsymbol{X}-\alpha Q^B(\boldsymbol{X})\|_2^2, \quad \text{s.t. } Q_i^B\in\{-1,\ 1\} \tag{27}$$

where α is a scaling factor. The analytic solution for this problem is $Q^B(\boldsymbol{X})=\text{sign}(\boldsymbol{X})$ and $\alpha=\|\boldsymbol{X}\|_1/N$, in which N is the number of elements in $\boldsymbol{X}$. In the case of simultaneously binarizing W and A of neural networks, they approximately yield the same solution form for both of them.

Similar to (27), Li *et al.* [161] extended it to weight ternarization by formulating it as

$$\min_{\alpha,\ Q^T(\boldsymbol{W})} \|\boldsymbol{W}-\alpha Q^T(\boldsymbol{W})\|_2^2, \quad \text{s.t. } Q_i^T\in\{-1,\ 0,\ 1\}. \tag{28}$$

They find an approximated solution featured by a threshold w_{th} for every layer so that the level transition points are $\pm w_{\text{th}}$. They estimate w_{th} and α from the L_1-norm of all weights and the weights the absolute value of which is larger than w_{th}, respectively. Zhu *et al.* [162] further extended it to an asymmetric case that the scaling factors for -1 and 1 are different. Also, w_{th} is empirically determined there, but the biggest difference from previous work is that the scaling factors are learned according to current weights and weight gradients. The ternarization is conducted for each filter in [161], whereas for each layer in [162]. Other methods can also be used for extreme quantization. For example, Wan *et al.* [166] combined (27) for weight binarization with scaling factors and (28) for activation ternarization without scaling factors; Achterhold *et al.* [163] leveraged Bayesian compression to ternarize weight parameters.

Extremely low-bit quantization, such as binarization and ternarization, cannot always produce acceptable accuracy, especially for large models demanding powerful expressive power or RNNs with more complicated dynamics. For instance, the original XNOR-Net [157] with binary W and A suffers from 12.4% accuracy loss on AlexNet over ImageNet. Ott *et al.* [188] even pointed out that binarization cannot preserve the functionality of RNNs, even if only quantizing W. Although some results [153] demonstrate binary quantization of RNNs, the accuracy drop is quite huge. In this situation, multibit quantization is widely studied, which will be detailed in the following parts.

In fact, early works on low-bit quantization also make efforts to increase the bitwidth for accuracy improvement. For example, Lin *et al.* [156] quantized W and A to 4 bits and thus the top-1 accuracy on GooleNet increases by 19.4% compared to that with binary W and A. The iconic multibit quantization starts from DoReFa-Net [147] that uses the heuristic linear quantization as (23). Note that in their definition, x is restricted within $[0,\ 1]$, so they leverage various range transformation to satisfy this condition.

Taking the W quantization as an example, it obeys

$$Q^k(w) = 2Q^k\left(\frac{\tanh(w)}{2\max(|\tanh(\boldsymbol{W})|)} + \frac{1}{2}\right) - 1 \tag{29}$$

where Q^k is the k-bit quantization function in (23). Before Q^k, $\boldsymbol{W}$ is nonlinearly transformed to be within $[0,\ 1]$; whereas after quantization, an affine transformation brings the value back to $[-1,\ 1]$ and constrains it there. It is worth noting that although the quantization idea is similar, that is, transforming to a friendly range before Q^k and then inverting the transformation, there still exist many variants. For instance, the quantization of E can use a different transformation [147]

$$Q^k(e) = 2\max(|E|)\left[Q^k\left(\frac{e}{2\max(|\boldsymbol{E}|)} + \frac{1}{2} + \sigma_k\right) - \frac{1}{2}\right] \tag{30}$$

where σ_k is a random noise. Based on the naive DoReFa-Net, the same group further proposes balanced quantization to match the real data distribution better by introducing mean($|\boldsymbol{X}|$) for normalization [153] and histogram equalization [169] before Q^k.

The early binarization and ternarization have also been extended to multibit cases. The greedy approximation for extending the single-bit binarization [157] to multiple bits is proposed [158], [171]. The original binarization problem in (27) now transfers to

$$\min_{\{\alpha_j,\ Q^{Bj}\}_{j=1}^k} \left\|\boldsymbol{X} - \sum_{j=1}^{k}\alpha_j Q^{Bj}\right\|_2^2, \quad \text{s.t. } Q_i^{Bj} \in \{-1,\ 1\} \tag{31}$$

where k is the quantization bitwidth. This problem can be solved by sequentially minimizing the residual, that is

$$\min_{\alpha_j,\ Q^{Bj}} \|\boldsymbol{X}_j - \alpha_j Q^{Bj}\|_2^2, \quad \text{s.t. } \boldsymbol{X}_j = \boldsymbol{X} - \sum_{j'=1}^{j-1}\alpha_{j'}Q^{Bj'}. \tag{32}$$

The above suboptimization problem at each iteration is the same as (27), which can be solved similarly. To improve accuracy, the final scaling factors α and binary vectors Q^B can be further refined [144], [171]. Other ways to solve (31) also exist [142], [143]. Similar quantization strategy but at different grain has also been reported, using multiple low-precision layers/layer blocks/layer groups to approximate the original high-precision ones, termed network expansion [174]. When quantizing weights, besides minimizing the weight approximation error, the problem can be formulated as to minimize the response error of each layer [189].

Different from the above optimization problems that attempt to find optimal low-precision values, there also exist works adapting the original high-precision data to converge to the quantized levels during training. For example, the following regularization can be introduced into the loss function [173]:

$$\min_{\boldsymbol{W},\lambda} \lambda\|\boldsymbol{X} - Q^k\|_2^2 \tag{33}$$

where λ is a trainable penalty parameter, and $\boldsymbol{X}$ can be weight or activation. Moreover, the quantization step length can also be trainable toward a power-of-two number by similarly optimizing $\min_{\Delta,\lambda_\Delta} \lambda_\Delta(\Delta - \text{round}_{\text{pow2}}(\Delta))^2$, which would make the quantized model more hardware-friendly.

Leng *et al.* [165] introduced the optimization framework of alternating direction method of multipliers (ADMM) into neural networks for weight quantization. The original problem can be formulated as $\min_{\boldsymbol{W}\in W_Q} L = f(\boldsymbol{W})$ where L is the normal loss function as in (4), W_Q is a set of discrete levels such as $W_Q = \alpha\cdot\{0,\ \pm 2^0,\ \pm 2^1,\ \pm 2^2, ...\}$, and each layer can have an independent scaling factor α. Note that here we omit the bias for simplicity. An indicator function $g(\boldsymbol{W})$ can be introduced into the problem: $g(\boldsymbol{W})$ is zero when all weights lie in the quantized space; otherwise, it is positive infinity. Then, the original problem is equivalent to

$$\min_{\boldsymbol{W},\ \boldsymbol{Z}} L = f(\boldsymbol{W}) + g(\boldsymbol{Z}), \quad \text{s.t. } \boldsymbol{W} = \boldsymbol{Z}. \tag{34}$$

Now the quantization problem is the same as the standard ADMM problem and can be similarly solved using the greedy minimization. Specifically, the augmented Lagrangian of (34) is

$$L_\rho(\boldsymbol{W},\boldsymbol{Z},\boldsymbol{Y}') = f(\boldsymbol{W}) + g(\boldsymbol{Z}) + \frac{\rho}{2}\|\boldsymbol{W} - \boldsymbol{Z} + \boldsymbol{Y}'\|_2^2 - \frac{\rho}{2}\|\boldsymbol{Y}'\|_2^2 \tag{35}$$

where ρ is a hyperparameter and $\boldsymbol{Y}' = \boldsymbol{Y}/\rho$. In this way, the greedy minimization iteratively performs the update as

$$\begin{cases} \boldsymbol{W}^{n+1} = \arg\min_{\boldsymbol{W}} L_\rho(\boldsymbol{W},\ \boldsymbol{Z}^n,\ \boldsymbol{Y}'^n) \\ \boldsymbol{Z}^{n+1} = \arg\min_{\boldsymbol{Z}} L_\rho(\boldsymbol{W}^{n+1},\ \boldsymbol{Z},\ \boldsymbol{Y}'^n) \\ \boldsymbol{Y}'^{n+1} = \boldsymbol{Y}'^n + \boldsymbol{W}^{n+1} - \boldsymbol{Z}^{n+1}. \end{cases} \tag{36}$$

Actually, the first subproblem is differentiable and can be solved by normal SGD with an L_2-norm regularizer. The second subproblem is equivalent to $\arg\min_{\boldsymbol{Z}\in W_Q} \frac{\rho}{2}\|\boldsymbol{W}^{n+1} - \boldsymbol{Z} + \boldsymbol{Y}'^n\|_2^2$, which is similar to the quantization problem in (24). Leng *et al.* [165] solved the second subproblem by iteratively fixing the scaling factor and the quantized vector to convert the bivariate optimization to two univariate optimizations. Each layer allows independent quantization parameters. Generally,

in the ADMM training, the auxiliary variable $\boldsymbol{Z}$ is strictly quantized to discrete levels at each training iteration and the weight $\boldsymbol{W}$ gradually approaches the discrete $\boldsymbol{Z}$ space.

Yin *et al.* [164] relaxed the hard quantization constraint of weights during training. The loss function slightly changes to

$$\min_{\boldsymbol{W}} f(\boldsymbol{W}) + \lambda \cdot \text{dist}(\boldsymbol{W}, W_Q) \tag{37}$$

where $\boldsymbol{W}$ is relaxed to nearly discrete levels at each training iteration. In the last training epoch, an enforced quantization is required to produce the final quantized weights. Bai *et al.* [192] adopted an analogous idea. They added a quantization-inducing regularizer into the loss function at each iteration, which is termed a proximal operator $\text{prox}_{\lambda R}$ like

$$\min_{\boldsymbol{W}^{n+1}} \frac{1}{2}\|\boldsymbol{W}^{n+1} - (\boldsymbol{W}^n - \eta \bigtriangledown \boldsymbol{W}^n)\|_2^2 + \lambda R(\boldsymbol{W}^{n+1}) \tag{38}$$

where $R(\boldsymbol{W}^{n+1})$ is zero when $\boldsymbol{W}^{n+1}$ lies in the quantized space; otherwise, it is positive. This proximal operator attracts the weights toward the discrete space especially at the condition of a large λ.

Besides the homogeneous quantization, Zhou *et al.* [172] proposed an incremental quantization strategy in an iterative manner. At each iteration, the unquantized weights are partitioned into two disjoint groups, then one group is quantized into powers of two or zero while the other remained in high precision and retrained to compensate the accuracy loss. The portion of quantized weights gradually increases until it covers all weights. The partition can be either random or magnitude-aware (quantizing smaller weights). Similar incremental quantization by partitioning the whole network into multiple layer groups and gradually quantizing front groups as training evolves is reported in [175], wherein extra noise is injected into the weights in the group right after the quantized groups to emulate the quantization error. Different from the fixed discrete space, as aforementioned, the quantizer itself can also be jointly trained together with the model parameters. For example, the dynamic range and quantization levels can be parameterized in different ways and trained using iterative optimization [143], [144] or gradient descent [142], [145], [146]. In addition, RL can be applied in data quantization to explore flexible bitwidth configuration across layers for accuracy improvement under certain hardware resource constraints [176].

Recently, some works begin to pay attention to the quantization of more objects such as weight update and BN. Wu *et al.* [182] demonstrated an aggressive quantization of most data (W/A/E/G/U). This combines the deterministic projection [see Fig. 16(a)] for W/A/E/U and stochastic projection [see Fig. 16(b)] for G. To realize the full quantization, it introduces layer-wise linear scaling operations

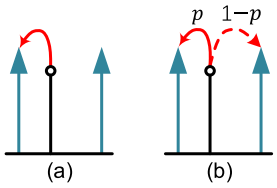


Fig. 16. *Level projection. (a) Deterministic—transition to the nearest level. (b) Stochastic—transition to nearby adjacent levels controlled by probability.*

to replace BN. A recent work [183] proposes a systematic methodology for precision assignment across all layers to achieve nearly optimal full quantization (W/A/E/G/U). Banner *et al.* [180] introduced a new BN operation, termed range BN, to greatly reduce the numerical instability and arithmetic overflow caused by the popular standard deviation-based BN. Specifically, the activation is simply normalized just by its dynamic range (along with a scaling factor determined by the batch size) rather than the complex standard deviation. In this way, the BN layer can also be quantized with acceptable accuracy loss (almost lossless on ResNet18 and ResNet50 over ImageNet at 8-bit quantization of W, A, E, and BN). Note that here G and U are not quantized to maintain the ability to accumulate small gradients, and a copy of error with higher precision (16 bits) is stored for calculating the weight gradient in the consequent backward pass. Aiming at the training quantization, some works push the multiplications to 16-bit floating-point numbers [177] or 16-bit fixed-point numbers [179]. However, the accumulation operation and weight update are still in 32-bit high precision. Recently, Wang *et al.* [184] further pushed the multiplications, accumulations, and updated in training to 8, 16, and 16-bit floating-point numbers, respectively.

b) RNN quantization: Table 9 lists most existing quantization works for RNNs. In contrast to CNNs, an RNN is a dynamic system that reuses the weight and accumulates the activation error in the temporal dimension. Therefore, compared to CNNs, it is more difficult to quantize RNNs with minimized accuracy loss. Furthermore, BP in RNNs has two dimensions: spatial (layer-by-layer) and temporal (timestep-by-timestep), which makes it harder to clarify the training dataflow and quantization sensitivity. This leads to the lack of quantization in the backward pass. Usually, only W and A are involved (few quantize E). Since the cellular state in LSTM is unbounded and independent of the costly matrix multiplication, the A quantization just includes the hidden state.

In fact, many quantization methods can be shared by both CNNs and RNNs [5], [169]. Although the methods are similar, the difficulty in quantizing RNNs and the fewer standardized benchmarks for comparison makes it lagging behind the impressive results from CNNs. Most benchmarking models regarding RNN quantization are only tested on relatively small data sets. For example,

Table 10 Sensitivity Analysis of Quantization Object for CNNs and RNNs

Reference	Sensitivity	Configuration and Accuracy
TernGrad (2017) [167]/ADMM (2018) [165]/TTQ (2016) [162]	CNN: G≤W	ImageNet-AlexNet, **G(ternary)**: top1-↑0.28% [167]; **W(ternary)**: top1-↓1.8% [165]; **W(ternary)**: top1-↑0.3% [162]
WRPN (2017) [170]/HWGQ (2017) [160]	CNN: W<A	ImageNet-AlexNet, **W(2b)**: top1-↑0.3%, **A(2b)**: top1-↓4.5% [170]; **W(binary)**: top1-↓0.3%, **A(ternary)**: top1-↓6.2% [160] ImageNet, ResNet18, **W(binary)**: top1-↓5%, **A(ternary)**: top1-↓28.8% [160]; VGG-Variant, **W(binary)**: top1-↓3.1%, **A(ternary)**: top1-↓20.3% [160]
DoReFa-Net (2016) [147]	CNN: A<E	**W(2b)** on SVHN, **A(2b)/E(4b)**: ↑0%; **A(4b)/E(2b)**: ↓16%
WAGE (2018) [182]	CNN: E<U CNN: BN matters	**W(ternary)/A(8b)/E(8b)** on CIFAR10-VGG8, **G(8b)/U(8b)**: ↓1.07%; **G(4b)/U(4b)**: ↓22.51% **W(ternary)/A(8b)** on ImageNet-AlexNet, **E(8b)/G(12b)/U(12b)**: top5-↓7.59%; **E(12b)/G(8b)/U(8b)**: top5-↓8.77% **W(ternary)/A(8b)**, **BN**: top5-↓1.38%; **Linear Scaling**: top5-↓4.85%
Neuron Increase (2017) [190]	RNN: A<W	PTB-LSTM300×1, **A(4b)**: ↑0.5% PPW; **W(4b)**: ↑5.6 PPW; **A(2b)**: ↑2.7% PPW; **W(2b)**: ↑32.4 PPW PTB-LSTM450/1000×1, **W(4b)/A(2b)**: 111.7/113.1 PPW; **W(2b)/A(4b)**: 130.6/128.4 PPW

Xu *et al.* [144] extended the multibit binarization to RNN quantization inspired by the previous work [171] and achieve advanced accuracy. Wang *et al.* [191] proposed a hybrid quantization method that combined the thresholded quantization for weights [161] and the probabilistic quantization [188] for hidden states. A sloping factor can be introduced into the activation function (e.g., sigmoid and tanh) to properly increase the curve steepness to reduce the error caused by ternarization [191].

4) Results: Summary and Discussion: Tables A1 and A2 in Appendix B present the accuracy results of low-bit (≤2b) and multibit (>2b) CNN quantization, respectively. Table A3 in Appendix B depicts the results of RNN quantization. Note that it is difficult to say which work is absolutely the best since the methods are divergent with the different quantization data object and bitwidth, as well as different model configuration and initialization.

In a nutshell, the heuristic methods are able to achieve sufficiently good accuracy on small or medium data sets such as MNIST, SVHN, and CIFAR10/CIFAR100 [150], [151], [168], [182]. Whereas, on larger data sets such as ImageNet, the optimization methods often perform better [162], [165]. The works on RNNs also present similar conclusions [144], [153], [169]. For CNN inference (W & A), ≥8-bit quantization is able to achieve lossless or even better accuracy, whereas ≤4-bit quantization will produce obvious accuracy degradation, especially on very deep networks [146], [149]. For RNN quantization, since it is more challenging, we only see a few works with extremely low bits [191]. Although more bits [144], [147], [169], [182] and wider layers [170], [174] can help to improve accuracy, they will increase parameters and operations as well.

Next, we provide some discussions on the sensitivity analysis as follows.

1) *Influence of Quantization Data Object:* The quantization of W, A, or G seems easier, while the quantization of E, U, or BN will significantly worsen the model. To clearly see the importance of quantization data objects, we carefully select reported data from existing works to form Table 10 for sensitivity analysis.
   Particularly in CNNs, G has similar or even smaller influence compared to W. Thanks to SGD's tolerance of noise, the extreme quantization of G is feasible for reducing the communication bandwidth in distributed training systems [167]. A often demonstrates a higher sensitivity than W because it has a wider dynamic range under the ReLU activation function. E in the backward pass is more sensitive than A in the forward pass. Banner *et al.* [180] proved that when the quantization object obeys a Gaussian distribution, the quantization loss can be bounded. They also show that E is more fragile due to the violation of the Gaussian distribution. Furthermore, U is the most sensitive data object for quantization since it directly determines the model quality by influencing the parameter update. If it is quantized, the very small parameter change (< the step length of discrete levels) at a certain iteration may not be powerful enough to update the model, which slows down the convergence and even makes it unable to converge. Li *et al.* [181] coarsely proved the convergence when quantizing W and U, and Li *et al.* [168] theoretically interpreted the U quantization as a Markov chain and conclude that there is no exploration benefit even if the learning rate is decreased. Wu *et al.* [182] also confirmed the significant influence of quantizing U through empirical experiments. This is the reason that most quantization methods remain the master weight and its update in high precision [177]–[179], [193]. Recently, the quantization of BN also attracts attention. Wu *et al.* [182] showed that the removal of BN will cause obvious accuracy loss, but the remaining of high-precision BN suffers from large training cost [194]. Therefore, reducing BN's nonlinearity and quantizing BN operations also matter [180], [194], [195]. In RNNs, as mentioned earlier, current quantizations only involve W and A. Since the hidden state is bounded by sigmoid and tan$h$, the quantization of A suffers from less accuracy loss than quantizing W. However, the quantization of the unbounded cellular state (in LSTM), E, G, U, or BN has yet to be fully touched.
2) *Influence of Network Structure:* In DNN quantization, there exist quite big result differences if using variable testing networks, especially on large data sets (e.g., ImageNet). Leng *et al.* [165] found that the accuracy gap resulting from different optimization methods is small on AlexNet and VGG models, which have more redundancy. Whereas, the quantization of compact models like ResNet and GoogleNet seems much

more challenging and presents a larger accuracy gap across different optimization methods. Besides the compactness, another reason that makes the quantization (on ResNet or GoogleNet) fragile is that the quantization of small C1×1 kernels probably breaks the short paths, which is the key structural feature of these networks. Hence, the remaining small kernels of compact networks in high precision, at least 8-bit fixed point, can be helpful for holding accuracy.

3) *Influence of the First/Last Layer:* The first and last layers are directly interfacing the network input and output, respectively. So, they usually play a more sensitive role in the network accuracy than other hidden layers. First, it is unreasonable to quantize the input data and the network output when considering the model accuracy. Although Tang *et al.* [158] successfully quantized the output of the last layer by introducing an additional scaling layer before the softmax classification, it is still not a mainstream solution. Second, regarding the weights in these two layers, most works quantize them but several [147], [157], [166], [173] still keep them intact to stay away from potential accuracy degradation. Another reason is that the memory and compute costs of these two layers are much lower than those of other layers, so it is acceptable even if they remained in high precision.
4) *Influence of Hyperparameters:* First, previous work found that a pretrained model can reduce the solution distance thus providing better accuracy after quantization [149]. Second, a larger batch size helps average the quantization noise thus giving better results [149]. Third, a quantized network has a smaller state space than its high-precision counterpart, which often causes fluctuated convergence if the learning rate is largely due to the frequent switching between remote levels. Therefore, a small learning rate (e.g., $<10^{-3}$) is widely adopted in quantized training, especially for extreme quantization such as binarization and ternarization [147], [158], [162].

E. Network Sparsification

Different from reducing the bitwidth of operands in quantization, network sparsification attempts to reduce the number of operands. Although it cannot simplify the arithmetic itself as quantization does, it is able to reduce the number of memory accesses and computational operations thus obtaining acceleration. In the single-way compression, the sparsification methods can usually achieve higher accuracy than the quantization ones, but the additional index overhead for addressing the nonzero elements and the irregular access/execution pattern becomes the major drawbacks. Furthermore, most works require retraining after network pruning to recover the model accuracy.

1) Sparsification Data Object: Fig. 17 presents two typical sparsification data objects: weight pruning and neuron pruning. The former reduces the number of edges, whereas the latter reduces the number of nodes. On the memory side, since the weights and activations occupy the most memory cost in inference and training, respectively, the pruning of them can accordingly save more memory in the corresponding scenario. On the compute side, since each neuron is usually connected to many weights, neuron pruning often obtains more operation reduction. Another big difference is that the weights are fixed during each iteration (in training) or the whole lifetime (in inference), while the neuron activations are variable as the input sample changes. Therefore, the normal dynamical neuron pruning requires a variable index, and the weight storage cannot be saved since they have never been pruned, just dynamically skipped. By contrast, the extreme static neuron pruning requires incorporating all the sparse patterns across different input samples, thus usually suffering from more accuracy loss.

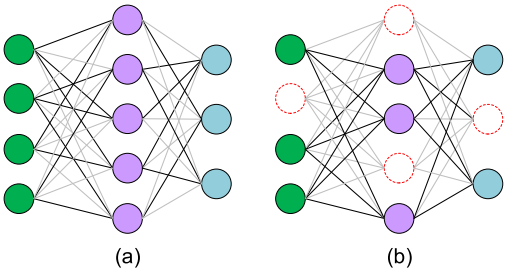


Fig. 17. ***Sparsification data objects. (a) Weight pruning. (b) Neuron pruning.***

If we further consider the sparsification in the forward and backward passes, we can classify the sparsification data objects into similar but fewer categories as those in data quantization: weight (W), activation (A), error (E), and gradient (G). W and G correspond to the weight, while A and E indicate the neuron. Here, we do not emphasize the weight update (U) because the sparse pattern of U is coupled with other data objects such as W, A, and E. In general, the inference sparsification only involves W and A, while the training sparsification additionally involves E and G.

2) Sparsification Philosophy:

a) Sparsification grain and structure: The sparsification data object and the pruning ratio mainly determine the memory saving degree, whereas the computation acceleration has a lot to do with the sparse pattern, that is, sparsification structure. The basic operation of neural networks can be abstracted as matrix multiplication, thus the distribution of zeros in the matrix reflects the sparsification structure. Note that the Conv computation can be commonly converted to the modality of general matrix multiplication (GEMM) by lowering the feature and weight tensors to matrices [196]. Fig. 18 depicts different sparsification structures, including element-wise, vector-wise, and block-wise. In some cases, the whole layer can be aggressively removed, that is, layer-wise.

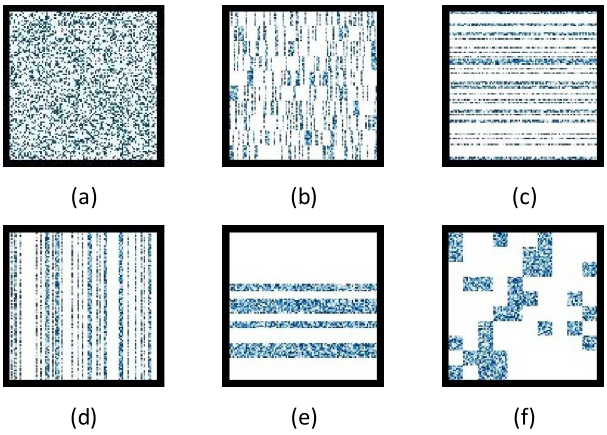


Fig. 18. *Sparsification structure. (a) Element-wise. (b)–(d) Vector-wise. (e) and (f) Block-wise.*

Here, we just take the weights, for example, in fact, it can be extended to neurons. Different structures can be produced by different pruning grains. For instance, in the weight sparsification, the kernel, fiber, or filter pruning produces vector-wise sparsity, while the channel or kernel group pruning produces block-wise sparsity. In the neuron sparsification, the FM pruning generates block-wise sparsity. Usually, as the sparse pattern becomes more structured, better acceleration can be gained since the index overhead is reduced and the resulting access/execution pattern is more regular. Unfortunately, structured sparsity often meets more difficulties in maintaining the model accuracy.

b) Problem formulation: A naive way for network sparsification is based on the heuristic search of unimportant elements. Here the "importance" can be simply determined by the magnitude of the absolute value, which is quite easy to obtain. The magnitude threshold for sparsification reflects a tradeoff between the sparsity and accuracy. The magnitude-aware pruning can also consider structured sparsity by grouping individual elements and comparing the group-aware magnitude (e.g., L_1-norm) for group pruning. The drawback of this naive pruning way is the lack of effective accuracy guarantee, which usually incurs large accuracy degradation, especially in the cases of coarse-grain sparsification or high sparsity. To this end, recent works prefer to formulate the network sparsification as an optimization problem to find the pruning locations. In this way, the possibility to achieve an optimal solution is increased, resulting in higher accuracy under the same sparsification structure and sparsity.

3) Sparsification of Weights and Neurons: Tables 11 and 12 list some typical works on network sparsification for CNNs and RNNs, respectively. Different from the quantization categories determined by the data object and bitwidth, network sparsification is further coupled with the sparsification grain/structure. For clarity, we mainly follow the categories of weight pruning and neuron pruning, whereas the different passes (forward/backward), grains/structures, and methods are embedded into both of them.

a) Weight pruning: Inspired by the pioneering works on weight pruning by LeCun *et al.* [226] and Hassibi and Stork [227], deep compression [6], [185] evidences that DNNs can still work well even if plenty of edges are pruned. Deep compression uses a straightforward method of ablating the weights with small absolute magnitude. Based on a pretrained model, an iterative ablation and fine-tuning gradually sparsify the network. The dead neurons, the connected weights of which are all removed after weight pruning, can also be safely ablated. With a similar pruning method, researchers prove that the pruned large sparse models including CNNs and RNNs consistently show higher accuracy than their small dense counterparts [198], [228] and the sparse ones trained from scratch [229] under the condition of identical memory footprint. The pruned weights can be either frozen throughout the consequent training [6], [185], [198] or recoverable when their gradients significantly increase [228]. Given a small seed network, NeST [217] combines the gradient-aware growing and magnitude-aware pruning, which can be used on both weights and neurons. Recently, NeST has been extended to LSTM by introducing the DNN structure into the gates and then using similar growing and pruning techniques to produce sparse weights, which is able to reduce the layers required to stack [230]. The spatial weights can also be converted to the weight coefficients in the frequency domain and then dynamically pruning

Table 11 Representative Sparsification Works for CNNs

Reference	Object	Problem Formulation	Sparsification Grain
Deep Compression (2015) [185]	W	Heuristic	Element
Sparsely-Connected (2017) [197]	W	Heuristic	Element
Prune or Not (2017) [198]	W	Heuristic	Element
FDNP (2018) [199]	W	Heuristic	Element
Filter Pruning (2016) [200]	W	Heuristic	Filter
SSL (2016) [201]	W	Optimization	Fiber/Channel/Filter/Layer
meProp (2017) [202]	E	Heuristic	Element
DGC (2017) [203]	G	Heuristic	Element
Taylor Expansion (2016) [204]	A	Optimization	FM
ThiNet (2017) [205]	A	Optimization	FM
Channel Pruning (2017) [206]	A	Optimization	FM
Slimming (2017) [207]	A	Optimization	FM
NISP (2018) [208]	A	Optimization	FM
ISTA Pruning (2018) [209]	A	Optimization	FM
AutoPrunner (2018) [210]	A	Optimization	FM
2PFPCE (2018) [211]	A	Optimization	FM
SFP (2018) [212]	W	Heuristic	Filter
Layer-compensated (2018) [213]	W	Heuristic & Evolution	Filter
AMC (2018) [214]	W	Heuristic & RL	Element/Channel
NestedNet (2018) [215]	W	Heuristic	Element/Kernel/Layer
Hybrid Pruning (2018) [216]	W	Heuristic	Element/Filter
NeST (2019) [217]	W/A	Heuristic	W(Element)/A(FM)
Channel Pruning (2018) [218]	W	Optimization	Kernel
Crossbar-aware Pruning (2018) [219]	W	Optimization	Block
TETRIS (2018) [220]	W	Optimization	Block
Joint Sparsity (2018) [221]	W	Optimization	Kernel
Synaptic Strength (2018) [222]	W	Optimization	Kernel
ADMM (2018) [7, 223–225]	W	Optimization	Any Structure

Table 12 Representative Sparsification Works for RNNs

Reference	Object	Problem Formulation	Sparsification Grain
Soft Pruning (2017) [228]	W	Heuristic	Element
Prune or Not (2017) [198]	W	Heuristic	Element
H-LSTM (2018) [230]	W	Heuristic	Element
ISS (2017) [232]	W	Optimization	Vector
meProp (2017) [202]	E	Heuristic	Element
DGC (2017) [203]	G	Heuristic	Element
Sparse Gate Gradients (2018) [233]	E[1]	Heuristic	Block

[1]Here the sparsification is conducted on the gate gradients rather than the state gradients.

the unimportant coefficients can help achieve a higher compression ratio [199]. Although most pruning methods are deterministic (according to weights and thresholds), probabilistic pruning also exists [197]. Different from previous work considering only the compression ratio and the accuracy loss, the energy reduction is also incorporated to determine the specific pruning strategy [231]. Besides the weight itself, the weight gradient can also be pruned using a similar magnitude-aware pruning [203].

The element-wise pruning causes extra index overhead and it is difficult to gain practical acceleration if without powerful support for the sparse computation [201]. Therefore, modern sparsification works begin to consider more coarse-grain structures. Ji *et al.* [220] produced block-wise weight sparsity via iteratively reordering the element-wise sparse weights. Yu *et al.* [234] grouped weights and pruned the groups with small values, which sufficiently utilizes the data length of single-instruction–multiple-data (SIMD) units. Anwar *et al.* [235] forced an identical nonzero map on all outgoing weight kernels of each FM and determine the maps using the evolutionary particle filtering approach. Li *et al.* [200] further demonstrated filter pruning in CNNs by simply leveraging the mentioned group-wise magnitude comparison. After the pruning of one weight filter in the current layer, more benefits we can get are that both the post-FM connected to the pruned weight filter and the corresponding weight channel of the next layer can be safely removed. Different from the hard pruning, the pruned filters can also be invoked during gradient descent, termed soft filter pruning [212]. EAs have been introduced to compensate the loss error caused by the naive greedy pruning that in each iteration the top-k unimportant filters are heuristically pruned [213]. Also based on the heuristic method, RL allows the automation of conventional hand-crafted pruning under a given resource or accuracy constraint [214]. The element-wise irregular pruning can be stacked on the filter-wise structured pruning to boost the compression ratio [216]. Multiple sparse networks with different sparsity level (lower level has higher sparsity) can be simultaneously trained from scratch using a merged object function, wherein the weights in the lower level are shared by the high-level networks [215]. This article demonstrates element/kernel/layer-wise weight sparsity for adaptive deep compression, knowledge distillation, and hierarchical classification.

Besides the above heuristic methods, optimization methods are more preferred by researchers to improve the model accuracy and increase the compression ratio, especially in Conv layers. Structured sparsity learning (SSL) is proposed, which groups weights into structured shapes and adds a least absolute shrinkage and selection operator (LASSO) regularization onto each group [201]. Now the loss function is modified as

$$\min_{\boldsymbol{W}} L = L_0(\boldsymbol{W}) + \lambda \sum_{g=1}^{G} \|\boldsymbol{W}^{(g)}\|_2 \tag{39}$$

where G is the number of structured weight groups, λ is a penalty parameter that affects the sparsity, and $L_0(\boldsymbol{W})$ is the normal objective function in (4). Here, we omit the bias item for clarity. After SSL training, it is easy to zero out the groups with small weights. The grouping scheme determines the resulting sparsification structure which can be vector-wise, channel/filter-wise, or even layer-wise. This method has been extended to LSTM with further concerns on the dimension constraint caused by feedback connections when grouping weight elements [232]. Many similar methods can be found in this domain. For example, Liu *et al.* [236] sparsified the parameters in a similar way based on tensor-decomposed models; Choi *et al.* [221] added a similar regularization onto each weight kernel in both spatial and Winograd domains; Lin *et al.* [222] introduced the kernel-level scaling factors (termed synaptic strength), which equals the products of the BN scaling factor from each input FM and the Frobenius norm of its connected weight kernels, and add a similar regularization onto each scaling factor for weight kernel pruning.

Recently, the ADMM method mentioned in Section III-D has also been used for high-accuracy sparsification [7], [223]–[225]. The optimization problem here is very similar to that in quantization, but the quantization constraint is changed to a sparsification one. Via different grouping schemes in the constraint condition, various sparsification structures can be produced [224]. Furthermore, the gradual sparsification with moderate pruning ratios can be leveraged to improve accuracy under extremely high compression ratios [225]. For weight kernel pruning, an optimization problem similar to the following (41) has been formulated, which can be solved by the genetic algorithm [218] or L_0-norm constrained gradient descent [219].

b) Neuron pruning: Neurons are also prunable by using the simple magnitude-aware rule (e.g., for A [237] or for E [202], [233]), according to the neuronal selectivity [238] or even based on random [239]/probabilistic [240] selection. Both element-wise (see [202] and [239]) and structured sparsity (see [233]) have been reported. However, these heuristic methods are difficult to maintain the model accuracy on large data sets. Formulating and

solving an optimization problem seem more promising. Luo *et al.* [205] did so in the FM pruning. The calculation of each output neuron in a Conv layer can be described as

$$y = \sum_{c=1}^{C} \sum_{k_1=1}^{K} \sum_{k_2=1}^{K} w_{ck_1k_2} x_{ck_1k_2} + b \tag{40}$$

where $\boldsymbol{W} \in \mathbb{R}^{C\times K\times K}$ is one weight filter and $\boldsymbol{X} \in \mathbb{R}^{C\times K\times K}$ is a corresponding sliding window across all input channels. If $\sum_{k_1=1}^{K}\sum_{k_2=1}^{K} w_{ck_1k_2}x_{ck_1k_2}$ and $y-b$ are denoted as $\hat{x}_c$ and $\hat{y}$, respectively, we have $\hat{y} = \sum_{c=1}^{C} \hat{x}_c$. After sampling a set of m output neurons from all output FMs over the entire training data set, that is, $\{(\hat{\boldsymbol{x}}_i, \hat{y}_i)\}$ where $i = 1, 2, \ldots, m$ and $\hat{\boldsymbol{x}}_i = \{\hat{x}_{ij}, j = 1, 2, \ldots, C\}$, to solve the optimization problem, the FM pruning issue is equivalent to

$$\min_{S} \sum_{i=1}^{m} \left(\hat{y}_i - \sum_{j\in S} \hat{x}_{ij} \right)^2, \quad S \subset \{1, 2, \ldots, C\} \tag{41}$$

where S is a subset of the C input channels. However, only a heuristic greedy algorithm is provided in [205] to solve this optimization problem. After pruning one input FM, both the corresponding weight filter in the current layer and the weight channel in the next layer can be removed. A linear regression is further used to modify the weights for better fitting the relationship between the pruned inputs and outputs, which provides a better initialization for the final fine-tuning. He *et al.* [206] solved a similar problem through a two-step algorithm that iteratively fixes weights to solve the sparsification mask via L_1-norm LASSO regression and fixes the sparsification mask to update weights via least square reconstruction. For extending to multibranch structures (e.g., ResNet), the output dimension for each block should remain unchanged [205], [206].

Liu *et al.* [207] added an L_1-norm regularization onto the scaling factors of BN layers. As well known, the BN transformation is governed by $\hat{y} = (y_{\text{in}} - \mu_B)/(\sqrt{{\sigma_B}^2 + \epsilon}), y_{\text{out}} = \gamma\hat{y} + \beta$, where μ_B and σ_B are the mean and standard deviation values of the input activations in one mini-batch, and γ and β are trainable affine transformation parameters (scale and shift). This implies that the output FM can be safely removed if its γ is quite small (resulting small y_{out}). During training, they introduce the γ regularization as

$$\min_{\boldsymbol{W},\,\boldsymbol{\gamma}} L = L_0(\boldsymbol{W}) + \lambda\|\boldsymbol{\gamma}\|_1 \tag{42}$$

where λ is a similar penalty parameter like that in (39). Subgradient descent is adopted to solve the nonsmooth L_1-norm regularization. After training, the output FMs with small γ can be deleted, as well as the connected weights. Ye *et al.* [209] leveraged a different regularization method termed ISTA to constrain γ and further modify the bias/moving average to compensate for the error caused by removing FMs. Yu *et al.* [234] proposed a neuron pruning method by introducing a binary mask sampled from a trainable scaling factor for each FM. Luo and Wu [210] added an extra layer named AutoPrunner after each activation tensor to generate a similar scaling effect. The AutoPrunner layer includes a batch pooling and a trainable FC layer to transform the activation tensor in $\mathbb{R}^{N\times C\times H\times W}$ to a scaling vector $\boldsymbol{v} \in \mathbb{R}^C$. Then a regularization is also used to control $\boldsymbol{v}$ for reaching a target r (percentage of preserved FMs), as follows:

$$\min_{\boldsymbol{W}} L = L_0(\boldsymbol{W}) + \lambda \left\| \frac{\|\boldsymbol{v}\|_1}{C} - r \right\|_2^2. \tag{43}$$

Actually, the added FC layer produces $\boldsymbol{v}$ using a modified sigmoid function, that is, $\boldsymbol{v} = \sigma(\alpha\boldsymbol{x})$. By gradually increasing α, the scaling vector $\boldsymbol{v}$ tends to be a binary mask due to the increased steepness of the sigmoid curve. In this way, the pruning and fine-tuning can be incorporated into an end-to-end training framework.

Taylor expansion has also been leveraged to prune neurons [204]. As well known, the Taylor expansion of $f(x)$ at point $x = a$ is $f(x) = \sum_{p=0}^{P}((f^{(p)}(a))/p!)(x-a)^p + R_p(x)$, where $R_p(x)$ is the pth order remainder. Now the following approximation of loss change when pruning neuron y_i can be described as

$$L(y_i) \approx L(y_i = 0) + \frac{\partial L}{\partial y_i} y_i \tag{44}$$

where $L(y_i)$ or $L(y_i = 0)$ denotes the loss with neuron y_i remained or pruned, respectively. Thus, the influence of neuron pruning on the loss can be simply estimated by $|(\partial L/\partial y_i)y_i|$, which reflects the neuron importance. For FM pruning, the estimation of loss change becomes

$$\Theta(y_c^{(l)}) = \left| \frac{1}{N} \sum_{n} \frac{\partial L}{\partial y_{c,n}^{(l)}} y_{c,n}^{(l)} \right| \tag{45}$$

where $y_c^{(l)}$ is the cth FM in the lth layer, and N is the number of neurons within the FM. $\Theta(y_c^{(l)})$ should be the average value across all samples in the data set. By contrast, from a statistical perspective, Min *et al.* [211] estimated the FM's importance according to its conditional entropy to minimize the global loss entropy error. Unlike minimizing the final loss error, Yu *et al.* [208] attempted to minimize the pruning error bound of the last response layer before classification by backpropagating the neuron importance scores. Based on these importance estimations at each layer, the FMs with less importance can be safely pruned.

The above neuron sparsification requires the complete calculation of output activations before pruning, which cannot simplify the computation itself of the current layer. To this end, the before-calculation neuron pruning is proposed. The preselection of unimportant neurons can

be finished by hashing search [241], negative-activation prediction [242], low-precision estimation [243], [244], prechecking neighboring pixels [245], or dimension-reduction estimation [246].

4) Results: Summary and Discussion: Tables A4 and A5 in Appendix C summarize the results of typical sparsification works on CNNs and RNNs, respectively. Some works show only sparsity and speedup for each layer, which are not included in these tables due to the difficulty in assessing the overall performance. Different from the quantization that can be well described by the data object and bitwidth, network pruning usually has a more flexible configuration including the data object, sparsity, and sparsification structure. This impedes the fair comparison across approaches.

Generally, the tradeoff between the sparsification ratio/structure and accuracy is the key to be optimized. Several empirical conclusions are summarized as follows.

1) Early works focus on increasing sparsity while they are restricted in the element-wise structure, by contrast, modern works pay more attention to structured sparsity for practical acceleration.
2) Compared to early heuristic methods, the automatic pruning and optimization methods emerge to improve the model accuracy.
3) It is more challenging to prune weights of Conv layers than FC layers due to the much less redundancy, therefore, the available sparsity on networks without large-size FC layers (e.g., ResNet, GoogleNet, and DenseNet) is often lower than that of others (e.g., AlexNet and VGG). Furthermore, the C1 $\times$ 1 weight kernel usually has much less redundancy than the ones with the larger size.
4) Neuron pruning usually suffers from more accuracy loss than weight pruning, while the gradient presents sufficient tolerance to pruning.
5) Different from the static weights (during each training iteration or the whole inference), the neuron activations are variable as the sliding window shifts (in the Conv layer) or as the input sample changes. Therefore, the generation of the neuron sparsification mask should comprehensively consider all sampled points from windows and inputs to solve the optimization problem if a lifetime pruning index is expected.
6) Due to the lack of natural zeros in RNNs (not like those in CNNs produced by ReLU function), few works report the activation (A) sparsification.

Interestingly, a recent work [247] gives a surprising observation that contradicts common beliefs: training a pruned network with sparse structure from scratch mostly performs better than the fine-tuned models. This seems opposed to the conclusion in [229], but the advocation of rethinking the pruning value is an interesting topic. It merits further investigations to answer which one is indeed critical after network pruning: the remained connection structure or the remained weights.

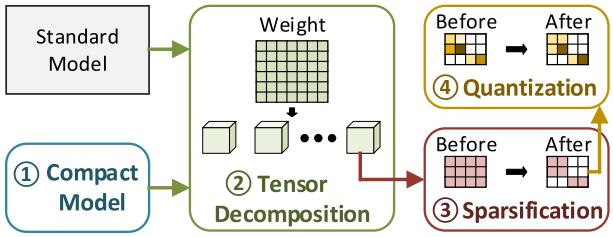


Fig. 19. *Joint-way compression.*

F. Joint-Way Compression

In this section, we introduce compression using more than one approach mentioned in previous subsections, termed joint-way compression here. Fig. 19 illustrates the mainstream compression flow. Given a standard model or a compact model made by the methods in Section III-B, one can decompose the large tensors of the network into more small tensors using tensor decomposition (see Section III-C), and then the resulting small tensors can be further quantized and sparsified (see Sections III-D and III-E). In this way, it is possible to achieve a much higher compression ratio compared with the single-way compression.

Table 13 lists some typical works in the literature that use the joint-way compression. Next, we introduce more details about them as follows:

1) Compact Model and Others: Recall that the compact model approach designs the base network itself while others compress a given base model. In this sense, the design of compact models is orthogonal to other compression approaches. A usual case is using compact models such as MobileNet as the base model to do additional compression such as quantization [175] or sparsification [198], [214]. However, compact models are already size-efficient than normal models, which usually impedes further model compression. For example, the weight pruning on MobileNet can achieve only 50%–60% sparsity without paying significant accuracy loss [198], [214].

Polino *et al.* [145] combined model distillation and data quantization. Model distillation trains a compact student model to simulate the output behaviors of a large teacher

Table 13 Existing Works on Joint-Way Compression

Reference	Compact NN	Decomposition	Quantization	Sparsification
Deep Compression (2015) [185]			✓	✓
SCNN (2015) [236]		✓		✓
Force Regularization (2017) [139]		✓		✓
INQ (2017) [172]			✓	✓
Quantized Distillation (2018) [145]	✓		✓	
VNQ (2018) [163]			✓	✓
Joint Sparsity (2018) [221]			✓	✓
Regularization (2018) [173]			✓	✓
PQASGD (2018) [248]			✓	✓
QIL (2019) [146]			✓	✓
ADMM (2018/2019) [7, 225]			✓	✓

model by minimizing the distillation loss. Different from the sole distillation, the weights are further quantized into finite levels with a trainable level distribution. Taking the standard ResNet34 with 87.2-MB parameters as the teacher model, they can achieve a 4-bit quantized student model with only 22-MB parameters. Interestingly, the 73.1% top-1 accuracy of the quantized compact model is even better than the 69.75% top-1 accuracy of the standard ResNet18 with 46.8-MB parameters.

2) Tensor Decomposition and Network Sparsification: The resulting small tensors produced by tensor decomposition still have room for further compression, which has been evidenced in [139] and [236]. In [236], L_1 and L_2 regularization constraints are added onto the small weight tensors from the previous tensor decomposition to force element-wise and vector-wise sparsity. With a customized sparse matrix multiplication algorithm, they demonstrate 2.5-6.9× execution speedup for Conv layers. In [139], the SSL weight pruning proposed in [201] is successfully conducted on the predecomposed model. Although the sparsity achieved by the postpruning is usually lower than that from a direct pruning of the undecomposed model, the final overall compression ratio of the joint compression can be higher.

3) Data Quantization and Network Sparsification: In recent works, the combination of data quantization and network sparsification is more popular than other combinations, which might be due to the intuitive methodology and the wider deployment in the design of hardware (see Section IV). Han *et al.* [185] used k-means to cluster weights after pruning and train the codebook. On AlexNet, data quantization can help them achieve an extra >3× compression ratio based on the network sparsification. Zhou *et al.* [172] combined the proposed incremental weight quantization and prior dynamic surgery based pruning in [249], which increases the 6.4× compression ratio of the sole quantization on AlexNet to 53× without compromising accuracy. Achterhold *et al.* [163] leveraged the quantization prior to force the weights of low variance only at the locations of quantization levels, and other weights with high variance are removable. Jung *et al.* [146] parameterized and trained the boundaries of the quantization range and prune the values smaller than the lower bound. Choi *et al.* [173], [221] used two steps to compress the models: 1) they first add magnitude-aware pruning regularization $\|\boldsymbol{W} \odot \mathbf{1}_{\{w_i \le w_{\text{th}}\}}\|_2$ to yield more removal of small weights; and 2) then, they conduct direct weight quantization [221] or add additional quantization-aware regularization $dist(\boldsymbol{W}, W_Q)$ to produce more quantizable weights [173] before the final hard quantization. With the help of joint-way compression, they push the 100× compression ratio of the sole sparsification on LeNet5 to 401× or the 5× ratio on ResNet18 to 24.2× without extra accuracy loss. The improvement of the compression ratio is boosted by a recent work [7] using the ADMM-based quantization on the top of the ADMM-based sparsification. In addition, for distributed learning, sparsifying, and quantizing the gradients can also be combined to reduce the communication cost without impacting the convergence rate [248].

Table 14 Advanced Results of Joint-Way Compression

Dataset	MNIST	CIFAR10	ImageNet	ImageNet	ImageNet
Network	LeNet5	DenseNet	AlexNet	ResNet18	ResNet50
Reference	ADMM [7, 225]	INQ [172]	ADMM [7]	Joint Sparsity [221]	ADMM [7]
Comp. Ratio	1910×	34.8×	231×	24.2×	38×
Top-1 Accuracy	99%	91.17%	57%	67.4%	↓0.0%

The joint-way compression can help achieve an extremely high compression ratio compared with the single-way compression mentioned earlier. Table 14 lists some advanced results on CNNs. Specifically, the compression ratio can reach 1910×, 34.8×, 231×, 24.2×, and 38× on LeNet5 (MNIST), DenseNet (CIFAR10), AlexNet (ImageNet), ResNet18 (ImageNet), and ResNet50 (ImageNet), respectively, with insignificant accuracy degradation.

An interesting question is whether the joint-way compression can really outperform the conventional single-way compression. To answer it, we present a direct comparison between the single-way compression and the joint-way compression in Table 15. For fairness, we restrict each comparison within the same reference (see each row in Table 15) and do not compare results from different works with distinct compression methods. Apparently, the joint-way compression can greatly boost the compression ratio while maintaining accuracy. Not that on more compact models such as ResNet with less redundancy, the improvement would be narrowed to some extent.

4) Discussion: At last, from the literature on joint-way compression and the collected results, we provide the following observations and conclusions.

1) Currently, most of the works focus on CNNs while only a few involve RNNs such as in [145], which might be due to the lack of standardized benchmarks for comparison in the RNN domain, rather than the methodology reason.
2) Data quantization can help increase the sparsity since more nonzero data would be forced to zeros during quantization, as shown in Fig. 20. Therefore, in the joint-way compression with the pruning-quantization order, the final sparsity is higher than the pruning ratio and grows as the bitwidth reduces.

Table 15 Comparison Between Single-Way Compression and Joint-Way Compression

Approach		Quantization	Sparsification	Joint
Regularization [173]	LeNet5 (MNIST)	–	100× (top1-99%)	401×[1,2] (top1-99%)
Deep Compression [185]	AlexNet (ImageNet)	–	9× (top1-57.22%)	27×[1] (top1-57.22%)
INQ [172]	AlexNet (ImageNet)	6.4× (top1-57.39%)	–	53× (top1-57.32%)
Joint Sparsity [221]	ResNet18 (ImageNet)	–	5× (top1-67.8%)	24.2× (top1-67.4%)
ADMM [7]	LeNet5 (MNIST)	–	167× (99%)	1910× (99%)
	AlexNet (ImageNet)	–	30× (top1-56.8%)	231× (top1-57%)
	ResNet50 (ImageNet)	–	17.4× (top1-↓0.8%)	38× (top1-↓0.0%)

[1] The index overhead is included.
[2] The Huffman coding is counted.

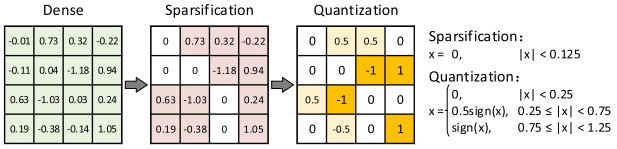


Fig. 20. ***Increasing sparsity through quantization.***

3) In existing works using the joint-way compression, most of them adopt a naive solution that applies two approaches sequentially. We encourage more research to invent integrated solutions to reduce the compression time. For example, the quantization can incorporate the pruning step like in [146] or it is possible to merge the quantization and sparsification constraints together when solving the $\boldsymbol{Z}$ variable in the ADMM compression [7].
4) Although the current joint-way compression has shown great potential for extreme compression, we expect more works that use different combinations or combine more approaches rather than only two to further improve the overall performance.
5) In the joint-way compression, one difficulty is the large configuration space of hyperparameters, which may hinder the search of the optimal result. Self-learning of hyper-parameters or the RL method might help but need future evidence.

IV. NEURAL NETWORK ACCELERATION: HARDWARE

In this section, we introduce the hardware implementation of neural networks, from general-purpose processors to vanilla accelerators with sole hardware optimization and modern accelerators with algorithm-hardware codesign. Before presenting the details, we first explain the computation pattern of neural networks because it is the basis of the latter hardware design. Fig. 21 illustrates two typical workloads in running neural networks, that is, the Conv layer and the FC layer. The former features Conv operations while the latter features matrix–vector multiplications (MVMs). In the Conv operation, there is huge data reusability including both activations and weights; by contrast, the data cannot be reused in the MVM operation if without the batching technique. In fact, the computation of one sliding window in the Conv operation is equivalent to an MVM operation.

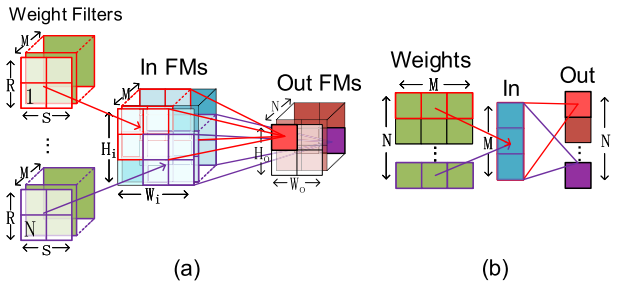


Fig. 21. ***Computation pattern. (a) Conv layer. (b) FC layer. Adapted from [14].***

A. Why Domain-Specific Accelerator for Neural Networks?

Besides innovative algorithms, increasing data resources, and easy-to-use programming tools, the rapid development of neural networks also heavily relies on the computing capability of the hardware. The general-purpose processors such as GPUs act as the mainstay in the deep learning era, especially on the cloud side. GPUs keep an ongoing pursuit of high throughput, whereas they pay the cost of huge resource overhead and energy consumption.

For the edge applications, the budget on resource and energy is usually very limited. How to minimize the latency, energy, and area has become an inevitable design concern. Although the use of on-GPU compression such as [201], [233], [250], and [251] can improve the performance, there is still a big gap far from our expectation due to the redundant design of general-purpose processors for flexible programmability and general applicability. This motivates the study of specialized accelerators tailored for neural networks. By sacrificing the flexibility to some extent, these accelerators focus on the specific pattern of neural networks to achieve satisfactory performance through the optimization of processing architecture, memory hierarchy, and dataflow mapping. Note that most neural network accelerators are intended for the inference phase and the CNN models due to their wide applications on the edge side. Although we can find a few ones for the training phase [252]–[254], RNN models [255] or both CNNs and RNNs [256]–[258], they are still not the mainstream. Therefore, the default neural network accelerators in this article indicate the CNN inference scenario unless otherwise specified. Due to the limited space, we just review the recent accelerators that can support large-scale neural networks and ignore the early ones [259], [260].

B. Sole Hardware Optimization

1) Parallel Compute Units and Orchestrated Memory Hierarchy: Usually, neural network accelerators make efforts in two aspects: enhancing the compute parallelism and optimizing the memory hierarchy. For example, DaDianNao [8] distributed weight memory into multiple tiles for the better locality. The MAC operations are performed in parallel by these tiles and the intermediate activations of different tiles are exchanged through a central memory. By contrast, in other neural network accelerators (e.g., Eyeriss [261], TPU [9], and Thinker [257]), the architecture often has an array of processing elements (PEs) with a small local buffer for each and a global buffer to hide the off-chip DRAM access latency, as shown in Fig. 22. Double buffering

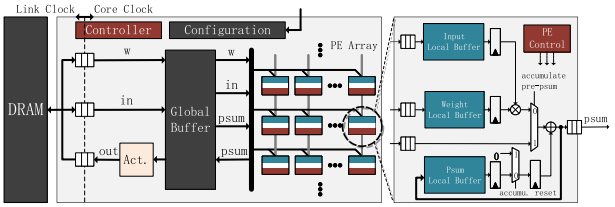


Fig. 22. *Typical digital architecture of neural network accelerators. Adapted from [261].*

technique can be used in the global buffer to prefetch data before layer computation [257].

The PE array features a dataflow processing fashion with an orchestrated network-on-chip (NoC) enabling direct message passing between PEs. Three types of data, input activation, weight, and partial sum (psum) of output activation, flow through the PE array when performing a Conv or MVM operation, which increases the data reuse thus decreasing the requirement for memory bandwidth. Furthermore, the dataflow pattern can be variable in a different design. We use Fig. 23 to briefly explain it. As depicted in Fig. 23(a), the output psum is stationary in each PE, and the input activations and weights propagate across PEs along the row and column directions, respectively. In this way, the inputs and weights can be reused by multiple PEs, which can reduce the memory access. Besides the output stationary dataflow [257], we can also see architectures with input stationary dataflow [262] or weight stationary dataflow [9], as shown in Fig. 23(b) and (c). Eyeriss [261] uses another dataflow called row stationary dataflow that is illustrated in Fig. 23(d). Specifically, each PE performs the Conv operation between one weight row and one input row, and the PEs in the same column generates one output row. The weights and psums propagate across

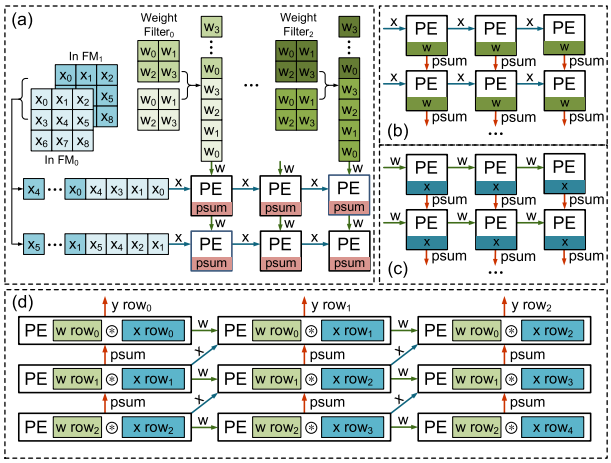


Fig. 23. *Various dataflow designs. (a) Output stationary. (b) Weight stationary. (c) Input stationary. (d) Row stationary. Adapted from [14], [257], and [263].*

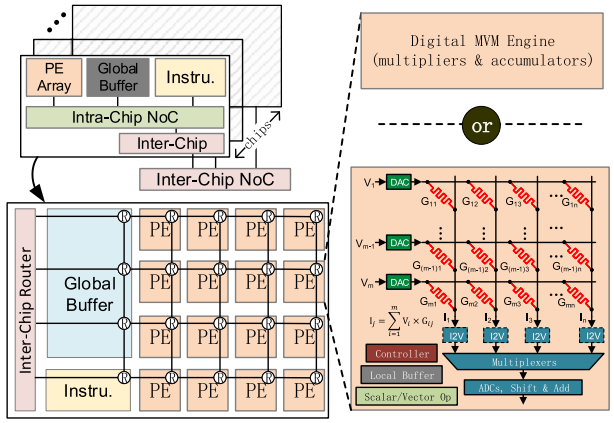


Fig. 24. *Scalable neural network accelerators using digital or analog–digital mixed technology. Adapted from [258] and [266].*

PEs along the row and column directions, respectively, whereas the input activations propagate along the diagonal direction that is different from other dataflow solutions mentioned above.

Combining the weight distribution [8] with the inter-PE data passing [9], [257], Tianjic [264], [265] and Simba [266], [267] adopt a scalable many-core/many-chip architecture where all cores (i.e., PEs) work in a decentralized manner without the global off-chip memory. Compared with the above accelerators, this emerging architecture is more spatial, as presented in Fig. 24. The weights are preloaded into each PE and remain stationary during the entire inference, and the activations propagate across intrachip and interchip cores.

2) Processing-in-Memory (PIM) Architecture: In conventional digital neural network accelerators, an MVM operation is split into many MAC operations to perform cycle by cycle. To improve the efficiency of performing MVMs, the PIM architecture based on emerging nonvolatile memory (eNVM) technologies has been widely studied. Taking memristor (e.g., RRAM [268], PCRAM [252]) as an example, the MVM can be performed in the analog domain. Each column of the crossbar obeys $I_j = \sum_i V_i G_{ij}$, where V_i is the input voltage of the ith row, I_j is the output current of the jth column, and G_{ij} is the memristor device conductance at the (i,j)th crosspoint. The weights are prestored as $\boldsymbol{G}$, and the input and output activations correspond to $\boldsymbol{V}$ and $\boldsymbol{I}$, respectively. The entire MVM can be processed in the analog domain with only one cycle, which is ultrafast. Nevertheless, ADCs (ADCs and DACs) are usually needed, which causes extra overhead. For the current to voltage (I2V) converters, they can be implemented either explicitly [258], [268] or implicitly [269] in different designs. The complete architecture is similar to the spatial architectures in [264]–[267] (see Fig. 24), where each MVM engine is a weight-stationary PE and a communication infrastructure helps the activation passing between PEs. Besides the MVM operation, some other

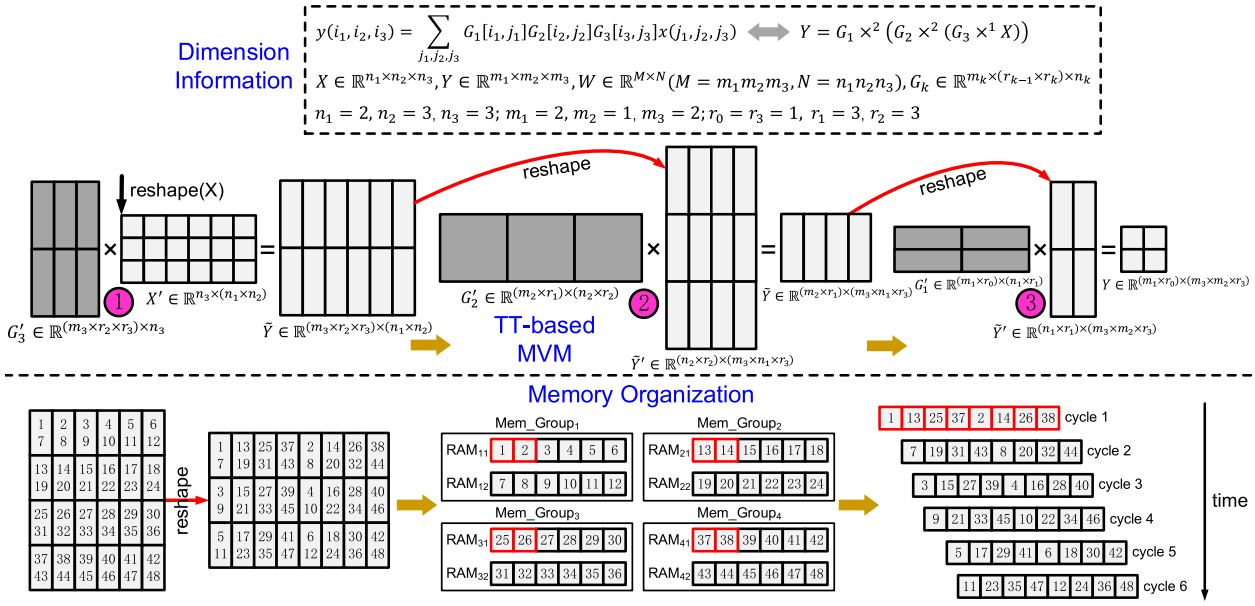


Fig. 25. *TT-based MVM and an example of its memory organization. Adapted from [11].*

operations such as scalar and vector operations should be additionally supported by PEs since they are necessary for some neural network models such as RNNs but cannot be handled by the memristor array efficiently [258]. Besides eNVM devices, traditional memories such as SRAM [270], DRAM [271], and Flash [272] can also be modified to support the PIM-fashion processing of neural networks. However, only small-scale prototype chips have been taped out due to the difficulty in fabrication, therefore they are not widely used in industry although they are very hot in academia.

C. Algorithm and Hardware Codesign

The above optimizations from the sole hardware side gradually meet the performance wall when the parallelism and data reuse are exhausted. Therefore, researchers start to seek help from both algorithm and hardware sides, termed algorithm-hardware codesign. Since then, various compression techniques for neural networks mentioned earlier in this article are widely exploited in the design of modern accelerators.

1) Transformation Into Compact Model: Compact models are able to shrink the model size, and they do not change the dense execution pattern and data precision as usual neural networks. Although they can directly run on most platforms, including both general-purpose processors and specialized accelerators, their compact connections/operations and variable data paths might degrade the hardware utilization. Eyeriss V2 [10] addresses this issue by codesigning flexible tiling/mapping schemes and an efficient NoC infrastructure with both high bandwidth and high data reuse. In addition, some compressed networks produced by other approaches can also be transformed into compact models. For example, the quantized networks can be viewed as dense models with shorter bit-width [179]; structurally pruned networks (e.g., vector-wise in RNNs and FM-wise in CNNs) [204], [205], [232], [234] or decomposed networks [139] can be reassembled into smaller dense models to speed up the running.

2) Tensorized Processing Engine: As mentioned above, most decomposed networks can be described as dense matrix operations with a smaller size, which are still suitable for running on general-purpose processors [139]. However, it is worth noting that for the tensor decomposition cases [4], [121], some abnormal operations are induced (e.g., dimension reshape of tensor cores and intermediate results), needing additional support if higher efficiency is expected. Recently, TIE [11] accelerates the TT decomposed neural networks, where the reshape is implemented by partitioning the working SRAM into multiple groups with a well-designed data selection mechanism.

Fig. 25 illustrates a detailed example of TT-decomposed MVM to help understanding. The original dimension information is $\boldsymbol{W} \in \mathbb{R}^{4\times 18}$, $\boldsymbol{X} \in \mathbb{R}^{18\times 1}$, and $\boldsymbol{Y} \in \mathbb{R}^{4\times 1}$. Using the TT decomposition, they are tensorized with dimensions of $\boldsymbol{X} \in \mathbb{R}^{2\times 3\times 3}$, and $\boldsymbol{Y} \in \mathbb{R}^{2\times 1\times 2}$, $\boldsymbol{G}_1 \in \mathbb{R}^{2\times(1\times 3)\times 2}$, $\boldsymbol{G}_2 \in \mathbb{R}^{1\times(3\times 3)\times 3}$, $\boldsymbol{G}_3 \in \mathbb{R}^{2\times(3\times 1)\times 3}$, where $\boldsymbol{W} = \text{reshape}(\boldsymbol{G}_1 \times^1 \boldsymbol{G}_2 \times^1 \boldsymbol{G}_3)$ [here the symbol $\boldsymbol{G}$ is equivalent to $\mathcal{G}$ in (21)]. Then the MVM is transformed into tensor contractions of $\boldsymbol{Y} = \boldsymbol{G}_1 \times^2 (\boldsymbol{G}_2 \times^2 (\boldsymbol{G}_3 \times^1 \boldsymbol{X}))$, from the right side to the left side corresponding to the step ①–③ in Fig. 25. The tensor contractions are essentially implemented by first matricizing the tensors into matrices and performing matrix–matrix multiplication

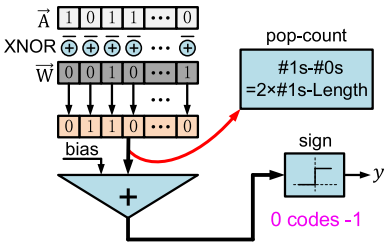


Fig. 26. *Logic operations in binary neural networks. Adapted from [273].*

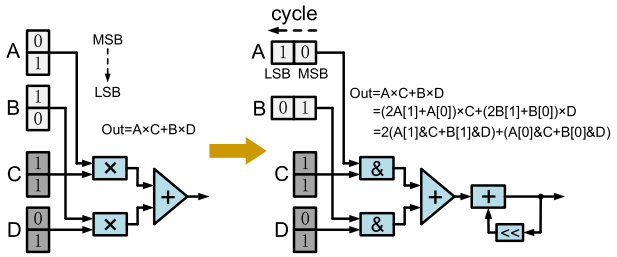


Fig. 27. *Bit-serial MAC processing. Adapted from [278].*

in a PE array. After each contraction step, the intermediate output $\widetilde{Y}$ should be reshaped to $\widetilde{Y}'$ for the computation of the next step. To realize the reshape efficiently, TIE provides an SRAM partition scheme. The memory consists of multiple groups and multiple chunks in each group. At each cycle, it reads a fraction of required data from different memory chunks and then assembles them into a new row that is expected in the reshaped format. For each chunk, the access addresses are continuous with high efficiency. In this way, the reshape operation is completed virtually, which does not consume any extra overhead.

3) Quantized Architecture: With negligible accuracy loss, the required data precision is usually $\geq$8-bit fixed point for CNN inference [8], [9]. Whereas, more aggressive low-bit quantization can be seen in the research community for the pursuit of ultrahigh execution performance with a certain degree of accuracy loss, which is the focus here. There are two categories of quantized neural network architecture: for fixed bitwidth or for variable bitwidth.

For the cases with fixed bitwidth, we can easily replace the high-precision multipliers and adders with identically lower-bit ones, and the overall architecture and dataflow remain the same as conventional ones mentioned in Section IV-B. Here, we emphasize the extreme quantization with only binary/ternary data precision. If only the weights are binarized/ternarized, the costly MAC operations can be reduced to simpler accumulations [274]. If both the weights and activations are binarized/ternarized, the MAC operations can be implemented by quite simple XNOR and pop-count logic operations [273], [275]–[277], which is illustrated in Fig. 26. Compared to binarization, ternarized neural networks can further exploit the sparsity due to the extra zero values.

For the cases with variable bitwidth, the motivation is that the required bitwidth probably varies across layers and models rather than always keeping uniform [176]. To this end, it is better to provide flexible architectural support for variable bitwidth. Two ways are utilized in this context: bit-serial [12], [278] and bit-decomposed [279], which are depicted in Figs. 27 and 28, respectively. Different from the normal bit-parallel processing, the bit-serial processing serially feeds one of the two operands bit by bit at each cycle. In this way, the MAC operations can be transformed into AND logic operations and shifted accumulations. Because the single MAC operation now needs more cycles, the bit-serial processing usually uses more PEs to enhance the parallelism. Owing to the mutiplierless architecture, this causes only a little area overhead but achieving continuous bitwidth support. The performance gain linearly increases with the bitwidth reduction. The bit-decomposed scheme fuses multiple low-precision MACs with shifted accumulations to form a higher precision MAC. The fusion group size is configurable to support flexible bitwidth. Compared to the bit-serial architecture, the bit-decomposed one can quantize both operands while paying the cost of discontinuity in the bitwidth distribution.

In addition, the tradeoff between the model accuracy and the execution performance should be carefully considered. Su *et al.* [280] profiled this tradeoff in inference via building a throughput estimation model. They conclude that on small-size data sets (e.g., MNIST or CIFAR10) INT2 or INT4 can reach a good tradeoff, while on large-size data sets (e.g., ImageNet) INT4 is the best choice. In HAQ [176], an RL-based AutoML approach is

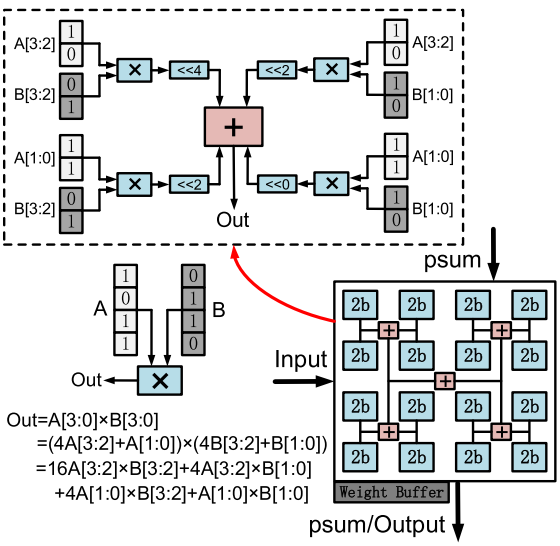


Fig. 28. *Bit fusion architecture based on bit-decomposed MAC processing. Adapted from [279].*

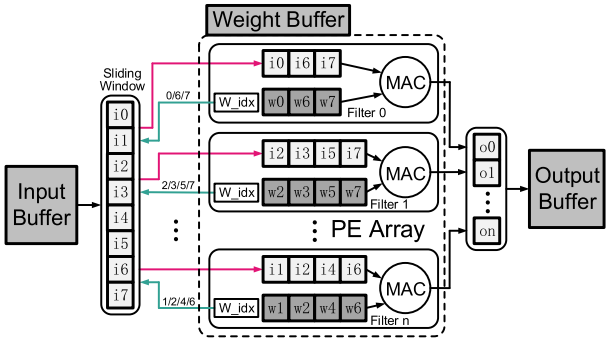


Fig. 29. ***Example of accelerators with weight sparsity. Adapted from [281].***

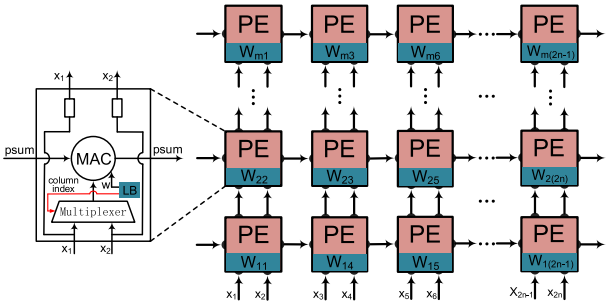


Fig. 30. ***Systolic array exploiting vector-wise weight sparsity via column combining. Adapted from [263].***

utilized to quantize neural networks considering both the accuracy loss and the latency/energy cost fed back from a simulated neural network accelerator.

4) Sparse Architecture: In the family of sparse neural network accelerators, the exploitable sparse patterns include weight sparsity, input sparsity, output sparsity, and the combination of multiple patterns, which will be discussed one by one.

Early sparse accelerators leverage weight sparsity by naively skipping the MACs with zero weights. Fig. 29 gives an implementation example adapted from Cambricon-X [281]. The weights are stored in a sparse format with addressing indexes. Each PE accesses the required activations according to the weight index and then performs the residual MACs. In this way, the number of operations can be reduced to a great extent, thus lowering the latency and energy. Nevertheless, the distribution of nonzero weights is usually very irregular, which causes large indexing overhead, PE imbalance, and memory access inefficiency.

To this end, structured sparse patterns mentioned in Fig. 18 have been further exploited. For example, the block-wise weight sparsity is leveraged to optimize the running performance on general-purpose processors [13], [220], [234], and a more aggressive diagonal weight matrix is utilized in the accelerator design [282]. In these architectures, the required indexes can be greatly decreased, the number of MACs in each PE becomes more balanced, and the memory organization/access can be more efficient. Fig. 30 shows an example of exploiting vector-wise sparsity in a systolic array [263]. The nonzero weights on multiple columns (e.g., two adjacent columns in this example) are combined together by only remaining the absolutely largest element on each row (the rest are pruned). Actually, a greedy column combining and iterative training is used to guarantee accuracy. Each PE buffers the nonzero weights with a column index, and a multiplexer is additionally integrated to select the correct input according to the weight column index.

Besides the weight sparsity exploitation, many accelerators also utilize the neuron sparsity. The ReLU activation function naturally produces abundant zeros which can help speed up the execution and reduce the energy consumption. On the input side, the simple run-length activation compression, zero input detection, and PE data gating to save bandwidth and energy are demonstrated in [261]. The zero input detection can be aggressively extended to small inputs [284]. Furthermore, it is possible to remain input activations in the compressed format and the activation indexes are used to access only the required weights for each PE [283], [285], as shown in Fig. 31. To address the irregularity issue, structured activation sparsity such as FM-wise pruning has also been adopted [234].

On the output side, some recent architectures can predict unimportant neurons (e.g., with negative accumulations that will be zero out by ReLU or with small outputs that will be ignored by max pooling). Fig. 32 provides two examples that exploit output sparsity. Fig. 32(a) monitors the sign of the accumulated activation in each PE and terminates its computation once a negative value is detected [242]. The accumulated activation will have no chance to be positive anymore once it becomes negative because the input activations after the previous ReLU function are non-negative and the weights are reordered with positive values first. Beyond the sign detection, the PE can also

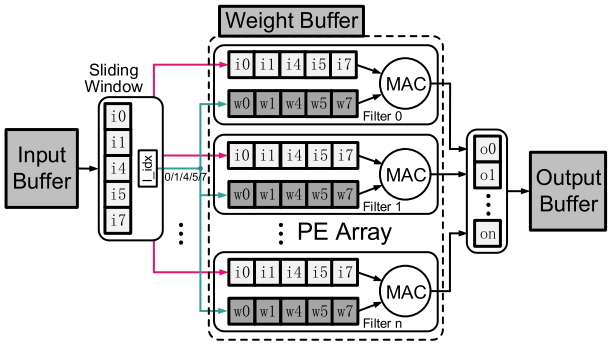


Fig. 31. ***Example of accelerators with input sparsity. Adapted from [283].***

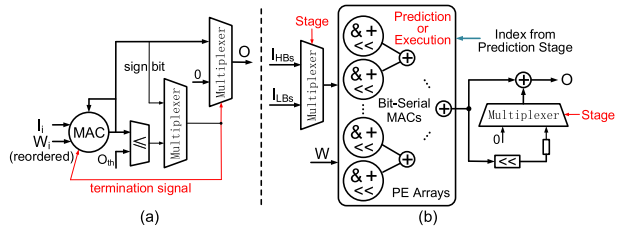


Fig. 32. *Examples of accelerators with output sparsity through (a) in-place speculation or (b) decoupled speculation and execution. Adapted from [242] and [244].*

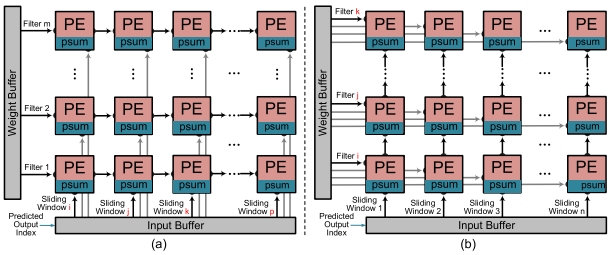


Fig. 34. *Workload mapping with output sparsity using 1-D data reuse. (a) Weight reuse. (b) Input reuse. Adapted from [244].*

terminate the computation once the accumulated activation is smaller than a predefined threshold, at the cost of a slight accuracy loss. Different from the in-place speculation, Fig. 32(b) performs the prediction separately and then executes the sparse computation with the predicted output indexes [244]. The speculation of unimportant output neurons is done by using only the high-order bits (HBs), and the residual processing of low-order bits (LBs) goes ahead only for the important output neurons. The final outputs are the addition of results from shifted HBs and LBs. The bit-serial MAC is adopted to support a flexible bitwidth assignment between prediction and execution. For the in-place speculation, there is serious PE idleness [242] as depicted in Fig. 33; while for the temporally decoupled speculation and execution, the PE array only processes the computation of important neurons according to the predicted output indexes [244], as illustrated in Fig. 34. By compromising the 2-D reuse of both weights and inputs, Fig. 34(a) only reuses the weight filter across each PE row and selects the corresponding input sliding window in each PE; Fig. 34(b) only reuses the input sliding window across each PE column and selects the corresponding weight filter in each PE. Other schemes to exploit the output sparsity also exist such as utilizing the distribution pattern of nonzero outputs [245].

To complement the above single-way sparsity, sparse architectures exploiting both the weight and neuron sparsity emerge recently. In this context, the weight and activation indexes should be intersected to jointly determine the data accesses and MAC operations [13], [262], [286], as shown in Fig. 35.

D. Joint-Way Compression in Accelerator

Besides the accelerators with the single-way compression approaches, the joint-way compression is also adopted to maximize the hardware performance. Quantization and sparsification are the most widely combined ones to simultaneously reduce the data precision and the number of operations for higher throughput and lower energy [13], [255], [284], [286]–[289]. As mentioned in Section III-F, the quantization is able to enlarge the sparsity due to the increased probability of being zero, which brings more performance improvement [285]. Although other combinations have yet to be fully touched in the neural network hardware community, we believe the joint-way compression has great potential in improving the performance of accelerators.

E. Compression on Emerging Memory Devices

Except for few works such as accelerating tensorized neural networks on the eNVM-based crossbar architecture [119], most works that study neural network compression on emerging memory devices discuss about data quantization and network sparsification.

Different from the previous digital accelerators using data quantization to improve the running performance, eNVM-based crossbar architectures naturally suffer from limited data precision. This precision limitation lies in weights (i.e., memory cell defects), input activations (i.e., DAC resolution), and (partial) output activations (i.e., ADC resolution) [290]. Considering the crossbar mapping and the low-bit limitation during the training of models can

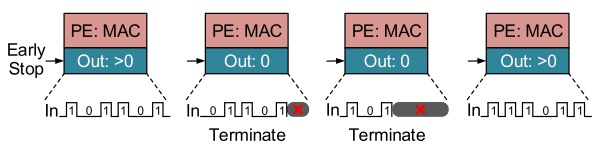


Fig. 33. *PE idleness problem in the sparse architecture. Here "1" represents nonzero values for simplicity.*

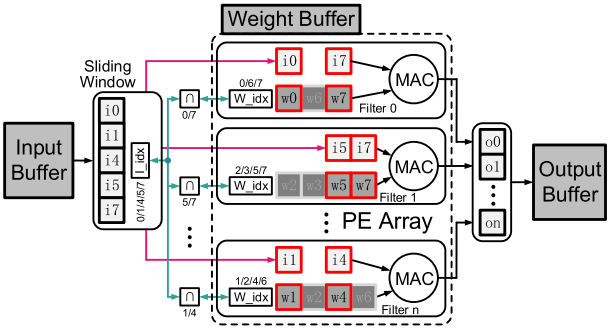


Fig. 35. *Example of accelerators with both input and weight sparsity. Adapted from [13].*

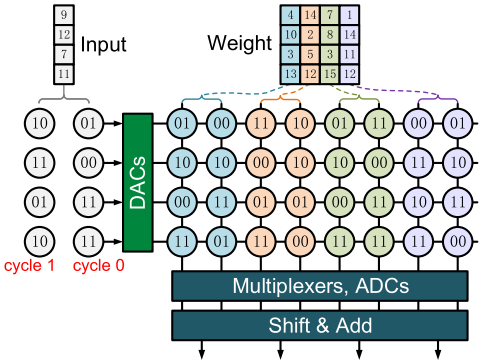


Fig. 36. *Bit decomposition on eNVM-based crossbar architectures. Adapted from [296].*

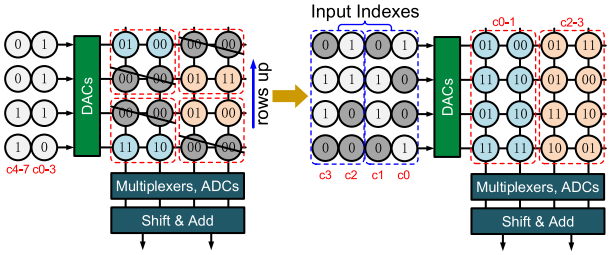


Fig. 37. *Sparsity exploitation on eNVM-based crossbar architectures. "c" denotes cycle. Adapted from [296].*

help recover the application accuracy [290], [291]. To support a flexible bitwidth, the bit decomposition technique is usually adopted, as presented in Fig. 36. Specifically, multiple memory cells can be grouped to represent a single weight value, where the outputs from different columns are added with shift; the bit-serial processing can be used to support a variable input bitwidth and reduce the required DAC resolution. In addition, binary neural networks are beneficial for the deployment on emerging memory architectures with only two stable levels or supporting only simple logic operations [292]–[295].

Although the quantization exploitation is easy, it is challenging to exploit sparsity due to the tight crossbar processing. Irregular zeros in the weight matrix cannot benefit the execution speedup and energy saving because the MVM operation still runs on a coupled crossbar. SNrram [297] extracts small dense submatrices from a large sparse matrix and then map them onto crossbars. The activation sparsity is transferred into the weight sparsity and only applies to deconvolution. Lin *et al.* [298] partitioned the weight matrix into blocks and further reorder the columns in each block to cluster zero columns onto the same crossbar for coarse-grain pruning. Liang *et al.* [219] directly solved a constrained optimization problem rather than the simple k-means clustering in [298] to realize the crossbar-grain pruning. ReCom [299] employs the SSL pruning [201] to produce vector-wise weight sparsity for friendly crossbar execution. Yang *et al.* [296] argued that the entire crossbar cannot be activated at once if considering the computation accuracy on practical eNVM devices. Therefore, they only activate a matrix block named operation unit (OU, marked by the red dotted boxes in Fig. 37) every time. In this way, the fine-grain zero rows in OUs can be removed and new rows can be accommodated. Extra input indexes are required for each OU column due to the row change. The nonzero inputs can be organized as multiple OU-size input groups to sequentially activate the corresponding rows while the zero inputs are skipped. Bit decomposition is further used to fit the device precision limitation and create more bit-level sparsity. By doing these, the same crossbar resource can perform more operations with fewer cycles. Besides inference, the structured gradient sparsity during training has also been leveraged to mitigate the limited endurance of eNVM devices for lifetime improvement [254].

F. Performance Summary

Table 16 lists the performance of some typical neural network accelerators using compression techniques. Note that the performance variance is largely due to different compression configurations, processing architectures, and device technologies. Therefore, it is quite hard to provide a very fair lateral comparison, and we just intend to give a high-level overview. Apparently, the compression techniques can help accelerators achieve a significant performance improvement, especially reflected from recent architectures with binary/ternary quantization. Whereas, how to reduce the accuracy loss caused by extreme compression on large-scale networks still needs further efforts.

V. SUMMARY AND DISCUSSION

In this article, we provide a comprehensive survey on neural network compression and acceleration from both algorithm and hardware perspectives. We individually introduce four categories of compression approaches: compact model, tensor decomposition, data quantization, and network sparsification, as well as the joint-way compression. We systematically present how to compress, what has

Table 16 Accelerators With Neural Network Compression

Reference	Substrate	Compression Approach	Execution Performance
ESE (2017) [255]	FPGA	S(W) & Q(W/A:12-16b)	61.5 GOPS/W (effective)
EIE (2016) [286]	ASIC	S(W/A) & Q(W:4b)	5 TOPS/W (effective)
Eyeriss (2017) [261]	ASIC	S(A)	246 GOPS/W
Combricon-X (2016) [281]	ASIC	S(W)	570 GOPS/W
Q-table (2017) [287]	ASIC	Q(W:4-16b) & S(A)	1.1 TOPS/W
TNN (2017) [300]	FPGA	Q(W/A:ternary/8b)	1.29 TOPS/W
DNPU (2017) [256]	ASIC	Q(W:4-16b)	2.1-8.1 TOPS/W
BRein Memory (2018) [273]	ASIC	Q(W/A:binary-ternary)	2.3∼6 TOPS/W
NullHop (2018) [285]	ASIC	S(A)	3 TOPS/W
UNPU (2018) [12]	ASIC	Q(W:1-16b)	3.1-50.6 TOPS/W
YodaNN (2018) [274]	ASIC	Q(W:binary)	61.2 TOPS/W
TIE (2019) [11]	ASIC	D (W:TT)	72.9 TOPS/W
XNORBIN (2018) [275]	ASIC	Q(W/A:binary)	95 TOPS/W
Parallelizing SRAM (2018) [295]	ASIC	Q(W/A:binary)	50∼100 TOPS/W
XNE (2018) [276]	ASIC	Q(W/A:binary)	112 TOPS/W
Mixed-signal (2018) [294]	ASIC	Q(W/A:binary)	772 TOPS/W

Note Abbreviations: D-decomposition, Q-quantization, S-sparsification. We summarize the references with explicitly reported performance values in tables or texts, not including those with only implicit values in figures. In the sole quantization scenario, we do not list the works using ≥8b quantization.

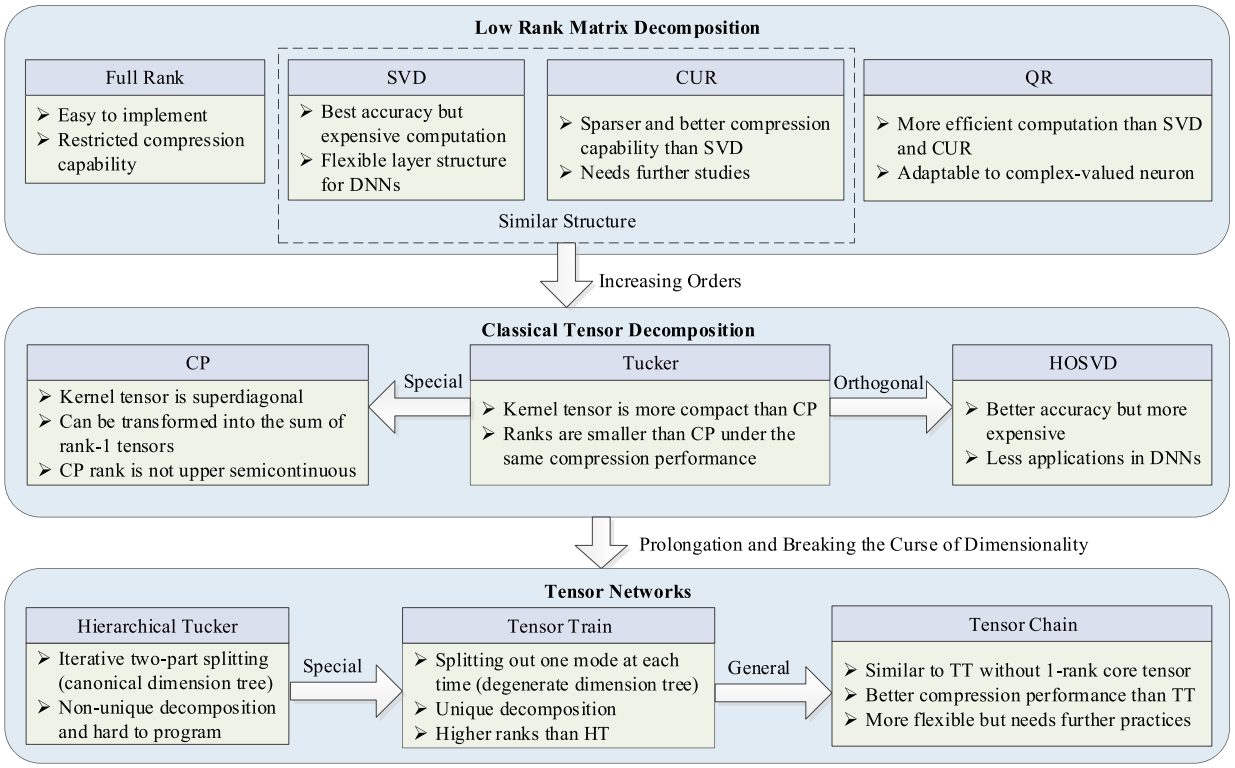


Fig. A1. ***Vein of tensor (including matrix) decomposition methods.***

been done, state-of-the-art performance, and model sensitivity analysis. On the hardware side, we mainly explain the reason for demanding domain-specific accelerators for neural networks and how to exploit compression techniques in hardware design. Through the review of typical works and performance comparison, we observe that compression is able to boost the execution performance such as throughput and energy.

Next, we try to discuss several interesting issues and opportunities in this field:

A. Training Compression and Acceleration

Compression and acceleration for training are more challenging than inference. The underlying reason is that the iterative manner in training makes it impractical to use complicated optimization algorithms. Although these algorithms are widely used for the inference compression, they often make the training too complicated to speed up. Moreover, the possible heuristic compression usually meets difficulty in maintaining the model accuracy. The compact model seems promising for accelerating both training and inference. Whereas, the search of optimal neural architectures with both high accuracy and regular execution pattern is time-consuming, which needs fast searching algorithms. For other compression approaches, how to balance the model size and the accuracy loss via training-friendly methods should be taken seriously. Currently, individual compression methods for both single-card training and distributed training exist, while how to make them compatible is still unclear and explorable.

B. Fair Comparison Between Compression Algorithms

In the neural network compression domain, various algorithms come out continuously. Besides the different solution paths (e.g., compact model, decomposition, quantization, and sparsification), there are even countless methods within each path. A consequent question is how to compare these methods, especially how to make the comparison fair. In our opinion, we should carefully consider the following aspects:

1) Training Versus Inference: Due to the different challenging degree in compressing training and inference to achieve speedup but maintain accuracy, a separate comparison is recommended for these two different application scenarios.

2) Compression Data Objects: Different compression data objects provide discrepant benefits in simplifying the computation pattern and memory cost. Besides, they also present very different sensitivity on accuracy loss. The same data objects should remain when presenting lateral comparison.

3) Compression Ratio Versus Model Accuracy: To achieve a higher compression ratio and a lower accuracy loss is what we always expect but an intractable tradeoff.

Table A1 Accuracy of Representative Low-Bit (≤2b) Quantization Works for CNNs

References	Configuration and Accuracy
BinaryConnect (2015) [150]	**W(binary)/det.**: MNIST-MLP (98.71%); SVHN-VGG9 (97.7%); CIFAR10-VGG9 (90.1%)
	W(binary)/sto.: MNIST-MLP (98.82%); SVHN-VGG9 (97.85%); CIFAR10-VGG9 (91.73%)
BNN (2016/2017) [5, 152]	**W(binary)/A(binary)/ det.**: MNIST-MLP (98.6%); SVHN-VGG9 (97.47%); CIFAR10-VGG9 (89.85%)
	W(binary)-det./A(binary)-sto./Shift BN: MNIST-MLP (99.04%); SVHN-VGG9 (97.2%); CIFAR10-VGG9 (88.60%)
	W(binary)/A(binary)/det. on ImageNet, AlexNet (top1-41.8%/top5-67.1%), GoogleNet (top1-47.1%/top5-69.1%)
	W(binary)/A(2b)/det. on ImageNet, AlexNet (top1-51.03%/top5-73.67%)
XNOR-Net (2016) [157]	**W(binary)** on ImageNet, AlexNet (top1-56.8%/top5-79.4%), ResNet18 (top1-60.8%/top5-83%), GoogLeNet (top1-65.5%/86.1%)
	W(binary)/A(binary) on ImageNet, AlexNet (top1-44.2%/top5-69.2%), ResNet18 (top1-51.2%/top5-73.2%)
DoReFa-Net (2016) [147]	**W(binary)** on SVHN, **A(binary)**: 97.1%; **A(binary)/E(2b)**: 93.4%; **A(2b)**: 97.6%
	SVHN, **W(binary)/A(2b)/E(2b)**: 93%; **W(2b)/A(binary)/E(2b)**: 91.8%; **W(2b)/A(2b)/E(2b)**: 91.8%
	W(binary)/pre-trained on ImageNet-AlexNet, **A(binary)**: top1-43.6%; **A(2b)**: top1-49.8%
BinaryNet (2017) [158]	**W(binary)/A(2b)** on ImageNet, AlexNet (top1-46.6%/top5-71.1%), NINNet (top1-51.4%/top5-75.6%)
TrueNorth (2016) [159]	**W(ternary)/A(binary)**: SVHN (97.46%); CIFAR10 (89.32%); CIFAR100 (65.48%)
HWGQ (2017) [160]	**W(binary)** on ImageNet: AlexNet (top1-52.4%/top5-75.9%), ResNet18 (top1-61.3%/top5-83.6%), VGG-Variant (top1-65.5%/top5-86.5%)
	A(ternary) on ImageNet: AlexNet (top1-49.5%/top5-73.7%), ResNet18 (top1-37.5%/top5-61.9%), VGG-Variant (top1-48.3%/top5-72.3%)
	W(binary)/A(binary) on ImageNet: AlexNet (top1-39.5%/top5-63.6%), ResNet18 (top1-42.1%/top5-67.1%), VGG-Variant (top1-50.1%/top5-74.3%)
	W(binary)/A(ternary) on ImageNet: AlexNet (top1-52.7%/top5-76.3%), VGG-Variant (top1-64.1%/top5-85.6%), GoogleNet (top1-63%/top5-84.9%)
	W(binary)/A(ternary) on ImageNet: ResNet18/34/50 (top1-59.6%/top5-82.2%)/(top1-64.3%/top5-85.7%)/(top1-64.6%/top5-85.9%)
	W(binary)/A(2b) on ImageNet: AlexNet (top1-50.6%/top5-74.3%), ResNet18 (top1-57.6%/top5-81%)
TWN (2016) [161]	**W(ternary)**: MNIST-LeNet5 (99.35%); CIFAR10-VGG8 (92.56%)
	W(ternary) on ImageNet, ResNet18 (top1-61.8%/top5-84.2%), ResNet18B (top1-65.3%/top5-86.2%)
TTQ (2016) [162]	**W(ternary)**: CIFAR10-ResNet20/32/44/56 (91.13%/92.37%/92.98%/93.56%); ImageNet-AlexNet (top1-57.5%/top5-79.7%)
Balanced DoReFa-Net (2017) [169]	**A(2b)** on ImageNet, AlexNet (top1-56.5%/top5-79%), ResNet18 (top1-61.1%/top5-82.7%)
	W(2b)/A(2b) on ImageNet, AlexNet (top1-54.7%/top5-77.7%), ResNet18 (top1-57.7%/top5-80.8%)
	W(2b)/A(2b)/balanced on ImageNet, AlexNet (top1-55.7%/top5-78%), ResNet18 (top1-59.4%/top5-82%)
WRPN (2017) [170]	**W(binary)/A(binary)** on ImageNet, AlexNet (inflation 1x/2x) (top1-44.2%/48.3%), ResNet34 (inflation 1x/2x/3x) (top1-60.54%/69.85%/72.38%)
	W(binary)/A(binary) on ImageNet, Inception-BN (inflation 2x) (top1-65.02%); **W(2b)/A(2b)** on ImageNet, AlexNet (inflation 1x/2x) (top1-51.3%/55.8%)
	W(2b)/A(2b) on ImageNet, Inception-BN (inflation 2x) (top1-70.75%), ResNet34 (inflation 2x) (top1-73.32%)
INQ (2017) [172]	**W(ternary)** on ImageNet, ResNet18 (top1-66.02%/top5-87.13%)
VNQ (2018) [163]	**W(ternary)**: MNIST-LeNet5 (99.27%); CIFAR10-DenseNet (91.22%)
BinaryRelax (2018) [164]	**W(binary)**, on CIFAR10, VGG11/16(89.28%/91.98%), ResNet18/20/32/34(94.19%/87.82%/90.65%/94.66%)
	W(binary), on CIFAR100, VGG11/16(63.82%/70.14%), ResNet18/34/56/110(74.04%/75.62%/67.65%/69.85%); on ImageNet, ResNet18(top1-63.2%/top5-85.1%)
	W(ternary) on CIFAR10, VGG11/16(91.01%/93.2%), ResNet18/20/32/34(94.98%/90.07%/92.04%/95.07%)
	W(ternary) on CIFAR100, VGG11/16(65.87%/72.1%), ResNet18/34/56/110(75.24%/76.16%/69.83%/72.32%); on ImageNet, ResNet18(top1-66.5%/top5-87.3%)
ADMM (2018) [165]	**W(binary)** on ImageNet, AlexNet (top1-57%/top5-79.7%), VGG16 (top1-68.9%/top5-88.7%), GoogLeNet (top1-60.3%/top5-83.2%)
	ImageNet, **W(binary)/C1_int8**: GoogLeNet (top1-65.4%/top5-86.7%); **W(binary)**: ResNet18/50 (top1-64.8%/top5-86.2%)/(top1-68.7%/top5-88.6%)
	W(ternary) on ImageNet, AlexNet (top1-58.2%/top5-80.6%), VGG16 (top1-70%/top5-89.6%), GoogLeNet (top1-63.1%/top5-85.4%)
	ImageNet, **W(ternary)/C1_int8**: GoogleNet (top1-66.7%/top5-87.7%); **W(ternary)**: ResNet18/50 (top1-67%/top5-87.5%)/(top1-72.5%/top5-90.7%)
Alternating (2018) [144]	**W(2b)/A(binary)**: MNIST-MLP (98.87%); CIFAR10-VGG9 (88.3%)
TBN (2018) [166]	**W(binary)/A(ternary)**: MNIST-LeNet5 (99.38%); SVHN-VGG8 (97.27%); CIFAR10-VGG8 (90.85%)
	W(binary)/A(ternary) on ImageNet: AlexNet (top1-49.7%/top5-74.2%), ResNet18/34 (top1-55.6%/top5-79%)/(top1-58.2%/top5-81%)
Regularization (2018) [173]	**W(binary)/A(binary)** on ImageNet: AlexNet (top1-43.9%/top5-69%), ResNet18 (top1-47.2%/top5-73%)
	W(binary)/A(2b) on ImageNet: AlexNet (top1-53%/top5-76.8%), ResNet18 (top1-60.4%/top5-83.3%)
	W(2b)/A(2b) on ImageNet: AlexNet (top1-54.1%/top5-77.9%), ResNet18 (top1-61.7%/top5-84.4%)
Group-Net (2018) [174]	ImageNet, **W(binary)**: ResNet18 (top1-69.6%/top5-89.1%); **W(binary)/A(binary)**: ResNet18/50 (top1-65.2%/top5-85.6%)/(top1-70.4%/top5-89%)
	W(binary)/A(2b) on ImageNet, AlexNet (top1-57.3%/top5-80.1%), ResNet18/50 (top1-67.6%/top5-87.8%)/(top1-73.4%/top5-90.8%)
ABC-Net (2017) [142]	ImageNet, **W(binary)/A(binary)**: ResNet18/34 (top1-42.7%/top5-67.6%)/(top1-52.4%/top5-76.5%)
LQ-Nets (2018) [143]	VGG-small on CIFAR10, **W(binary)**: (93.5%); **W(2b)**: (93.8%); **W(binary)/A(2b)**: (93.4%); **W(2b)/A(2b)**: (93.5%)
	ResNet20 on CIFAR10, **W(binary)**: (90.1%); **W(2b)**: (91.8%); **W(binary)/A(2b)**: (88.4%); **W(2b)/A(2b)**: (90.2%)
	AlexNet on ImageNet, **W(2b)**: (top1-60.5%/top5-82.7%); **W(binary)/A(2b)**: (top1-55.7%/top5-78.8%); **W(2b)/A(2b)**: (top1-57.4%/top5-80.1%)
	ImageNet, VGG-Variant, **W(binary)/A(2b)**: (top1-67.1%/top5-87.6%), **W(2b)/A(2b)**: (top1-68.8%/top5-88.6%); DenseNet121, **W(2b)/A(2b)**: (top1-69.6%/top5-89.1%)
	GoogLeNet-Variant on ImageNet, **W(binary)/A(2b)**: (top1-65.6%/top5-86.4%), **W(2b)/A(2b)**: (top1-68.2%/top5-88.1%)
	ResNet18 on ImageNet, **W(binary)**: (top1-68%/top5-88%); **W(binary)/A(2b)**: (top1-62.6%/top5-84.3%); **W(2b)/A(2b)**: (top1-64.9%/top5-85.9%)
	ResNet34 on ImageNet, **W(binary)/A(2b)**: (top1-66.6%/top5-86.9%), **W(2b)/A(2b)**: (top1-69.8%/top5-89.1%)
	ResNet50 on ImageNet, **W(binary)**: (top1-75.1%/top5-92.3%); **W(binary)/A(2b)**: (top1-68.7%/top5-88.4%); **W(2b)/A(2b)**: (top1-71.5%/top5-90.3%)
QIL (2019) [146]	**W(ternary)/A(ternary)** on ImageNet, AlexNet (top1-58.1%), ResNet18/34 (top1-66.1%)/(top1-70.6%)
TernGrad (2017) [167]	**G(ternary)/sto.** on ImageNet, AlexNet (top1-57.61%/top5-80.47%), GoogleNet (top5-86.77%)
TQN (2017) [168]	**W(binary)/U(binary)/det.** on CIFAR10, VGG9-1 (76.01%), VGG9-2 (78.12%), ResNet56 (66.44%), WRN56-2 (72.10%)
	W(binary)/U(binary)/det.: CIFAR100-ResNet56 (64.66%); ImageNet-ResNet18 (47.89%)
	W(binary)/U(binary)/sto. on CIFAR10, VGG9-1 (76.67%), VGG9-2 (79.44%), ResNet56 (73.51%), WRN56-2 (78.42%)
	W(binary)/U(binary)/sto.: CIFAR100-ResNet56 (41.94%); ImageNet-ResNet18 (11.14%)
	W(binary)/U(binary)/sto./big batch on CIFAR10, VGG9-1 (83.05%), VGG9-2 (83.23%), ResNet56 (81.16%), WRN56-2 (83.96%)
	W(binary)/U(binary)/sto. /big batch: CIFAR100-ResNet56 (49.21%); ImageNet-ResNet18 (22.32%)
	W(binary)/sto. on CIFAR10, VGG9-1 (89.64%), VGG9-2 (91.79%), ResNet56 (91.17%), WRN56-2 (92.83%);
	W(binary)/sto.: CIFAR100-ResNet56 (64.66%); ImageNet-ResNet18 (47.89%)
GXNOR-Net (2018) [151]	**W(ternary)/A(ternary)/U(ternary)**: MNIST-LeNet5 (99.32%); SVHN-VGG8 (97.37%); CIFAR10-VGG8 (92.5%)
QBP2 (2018) [180]	**W(binary)/A(binary)**: CIFAR10-ResNet18 (inflation 5) (89.3%)
	W(binary)/A(binary)/E(ternary) on CIFAR10, VGG9 (inflation 3x) (90.47%), ResNet18 (inflation 5x) (89.2%)

The compression ratio and the model accuracy must be provided together. The sensitivity curve ("accuracy versus compression ratio") and variable controlling ("keep one the same and compare the other") are recommended when compared with prior work.

4) Compression Ratio Versus Practical Benefits: Sometimes, the compression ratio cannot accurately reflect how much practical benefits (e.g., latency and energy reduction) we can obtain. For example, structured sparsification is much easier to produce higher speedup compared to the

Table A2 Accuracy of Representative Multibit (>2b) Quantization Works for CNNs

References	Configuration and Accuracy
QBP1 (2015) [156]	**W(1b)/A(3b)**: MNIST-MLP (98.71%); SVHN-VGG9 (97.52%); CIFAR10-VGG8 (87.92%)
	W(ternary)/A(3b): MNIST-MLP (98.85%); SVHN-VGG9 (97.58%); CIFAR10-VGG8 (87.99%)
BNN (2017) [152]	ImageNet-GoogleNet, **W(4b)/A(4b)/ det.**: top1-66.5%/top5-83.4%; **W(6b)/A(6b)/E(6b)/det.**: top1-66.4%/top5-83.1%
DoReFa-Net (2016) [147]	**W(1b)** on SVHN, **A(3b)/E(3b)**: 96.8%; **A(3b)/E(6b)**: 97.7%; **A(4b)/E(4b)**: 97.5%; **A(4b)/E(8b)**: 97.7%
	W(1b) on ImageNet-AlexNet, **A(3b)/A(6b)**: top1-47.1%; **A(4b)/E(6b)**: top1-48.2%
	ImageNet-AlexNet, **W(1b)/A(4b)/pre-trained**: top1-53%; **W(8b)/A(8b)/E(8b)**:top1-53%
Balanced DoReFa-Net (2017) [169]	**W(4b)/A(4b)/balanced** on ImageNet-GoogLeNet: top1-67.7%/top5-87.3%
WRPN (2017) [170]	**W(2b)/A(4b)** on ImageNet, AlexNet (inflation 1x/2x) (top1-50.5%/57.2%), ResNet34 (inflation 2x) (top1-73.58%), Inception-BN (inflation 2x) (top1-71.61%)
	W(4b)/A(2b) on ImageNet, AlexNet (inflation 1x/2x) (top1-52.4%/57.3%), ResNet34 (inflation 2x) (top1-73.5%)
	W(4b)/A(4b) on ImageNet, AlexNet (inflation 1x/2x) (top1-54.4%/58.6%), ResNet34 (inflation 2x) (top1-74.52%), Inception-BN (inflation 2x) (top1-71.63%)
Sketching (2017) [171]	**Conv-W(3b)/FC-W(1b)/pre-trained** on ImageNet-AlexNet (top1-55.2%/top5-78.8%)
INQ (2017) [172]	**W(5b)** on ImageNet, AlexNet (top1-57.39%/top5-80.46%), VGG16 (top1-71.82%/top5-90.3%), GoogleNet (top1-69.02%/top5-89.28%), ResNet50 (top1-74.81%/top5-92.45%)
	ImageNet-ResNet18, **W(5b)** (top1-68.98%/top5-89.1%); **W(4b)** (top1-68.89%/top5-89.01%); **W(3b)** (top1-68.08%/top5-88.36%)
HWGQ (2017) [160]	**W(binary)/A(3b/4b)** on ImageNet: AlexNet (top1-52.4%/top5-75.8%)/(top1-52.6%/top5-76.2%), ResNet18 (top1-60.3%/top5-82.8%)/(top1-60.8%/top5-83.4%)
ADMM (2018) [165]	**W(5 levels)** on ImageNet, AlexNet (top1-59.2%/top5-81.8%), VGG16 (top1-71.7%/top5-90.7%), GoogLeNet (top1-65.9%/top5-87.3%)
	ImageNet, **W(5 levels)/C1_int8**: GoogLeNet (top1-67.4%/top5-88.3%); **W(5 levels)**: ResNet18/50 (top1-67.5%/top5-87.9%)/(top1-73.9%/top5-91.5%)
	W(7 levels) on ImageNet, AlexNet (top1-60%/top5-82.2%), VGG16 (top1-72.2%/top5-90.9%), GoogLeNet (top1-66.3%/top5-87.5%)
	ImageNet, **W(7 levels)/C1_int8**: GoogLeNet (top1-67.6%/top5-88.3%); **W(7 levels)**: ResNet18/50 (top1-68%/top5-88.3%)/(top1-74%/top5-91.6%)
Regularization (2018) [173]	**W(4b)/A(4b)** on ImageNet: AlexNet (top1-56.6%/top5-79.8%), ResNet18 (top1-67.3%/top5-87.9%)
	W(8b)/A(8b) on ImageNet: AlexNet (top1-57.2%/top5-79.9%), ResNet18 (top1-68.1%/top5-88.2%)
Group-Net (2018) [174]	**W(binary)/A(4b)** on ImageNet, ResNet18/50 (top1-69.2%/top5-88.5%)/(top1-75.2%/top5-91.7%)
ABC-Net (2017) [142]	ImageNet, **W(3b)/A(3b)**: ResNet18/34 (top1-61%/top5-83.2%)/(top1-66.7%/top5-87.4%); **W(5b)/A(5b)**: ResNet18/34/50 (top1-65%/top5-85.9%)/(top1-68.4%/top5-88.2%)/(top1-70.1%/top5-89.7%)
LQ-Nets (2018) [143]	VGG-small on CIFAR10, **W(3b)**: (93.8%); **W(2b)/A(3b)**: (93.8%); **W(3b)/A(3b)**: (93.8%)
	ResNet20 on CIFAR10, **W(3b)**: (92%); **W(2b)/A(3b)**: (91.1%), **W(3b)/A(3b)**: (91.6%)
	ResNet18 on ImageNet, **W(3b)**: (top1-69.3%/top5-88.8%); **W(4b)**: (top1-70%/top5-89.1%); **W(3b)/A(3b)**: (top1-68.2%/top5-87.9%); **W(4b)/A(4b)**: (top1-69.3%/top5-88.8%)
	ImageNet, ResNet34, **W(3b)/A(3b)**: (top1-71.9%/top5-90.2%); ResNet50, **W(4b)**: (top1-76.4%/top5-93.1%), **W(3b)/A(3b)**: (top1-74.2%/top5-91.6%), **W(4b)/A(4b)**: (top1-75.1%/top5-92.4%)
QIL (2019) [146]	**W(3b)/A(3b)** on ImageNet, AlexNet (top1-61.3%), ResNet18/34 (top1-69.2%)/(top1-73.1%)
	W(4b)/A(4b) on ImageNet, AlexNet (top1-62%), ResNet18/34 (top1-70.1%)/(top1-73.7%)
	W(5b)/A(5b) on ImageNet, AlexNet (top1-61.9%), ResNet18/34 (top1-70.4%)/(top1-73.7%)
UNIQ (2018) [175]	**W(4b)/A(8b)** on ImageNet, ResNet18/34/50 (top1-67.02%)/(top1-71.09%)/(top1-73.37%), MobileNet (top1-66%)
	W(5b)/A(8b) on ImageNet, ResNet18/34 (top1-68%)/(top1-72.6%), MobileNet (top1-67.5%)
	ImageNet, **W(4b)**: ResNet34/50 (top1-73.1%)/(top1-74.84%); **W(8b)/A(8b)**: MobileNet (top1-68.25%)
FAQ (2018) [149]	**W(4b)/A(4b)** on ImageNet: ResNet18/34/50 (top1-69.82%/top5-89.1%)/(top1-73.31%/top5-91.32%)/(top1-76.27%/top5-92.89%)
	W(8b)/A(8b) on ImageNet: ResNet18/34/50 (top1-70.02%/top5-89.32%)/(top1-73.71%/top5-91.63%)/(top1-76.52%/top5-93.09%)
	W(8b)/A(8b) on ImageNet: Inception-v3 (top1-77.6%/top5-93.59%), VGG16 (top1-73.66%/top5-91.52%)
	W(8b)/A(8b) on ImageNet: ResNet152 (top1-78.54%/top5-94.07%), DenseNet161 (top1-77.84%/top5-93.91%)
MP (2017) [177]	**W(FP16)/A(FP16)/E(FP16)** on ImageNet: AlexNet (top1-56.93%), VGG16 (top1-65.43%), GoogLeNet-v1/v2/v3 (top1-68.43%/70.02%/74.13%)
	W(FP16)/A(FP16)/E(FP16)/loss scale: ImageNet-ResNet50 (top1-76.04%); VOC2007-Faster R-CNN (69.7%); VOC2007/2012-Multibox SSD (77.1%)
MP-INT (2018) [179]	**W(DFP16)/A(DFP16)/E(DFP16)** on ImageNet: AlexNet (top1-56.94%/top5-80.06%), VGG16 (top1-68.12%/top5-88.18%)
	W(DFP16)/A(DFP16)/E(DFP16) on ImageNet: GoogleNet (top1-69.34%/top5-89.31%), ResNet50 (top1-75.77%/top5-92.84%)
DST (2018) [181]	**W(7b)/U(7b)**: MNIST-MLP (98.93%); **W(8b)/U(8b)**: SVHN-VGG8 (97.55%); **W(9b)/U(9b)**: CIFAR10-VGG8 (88.16%)
WAGE (2018) [182]	**W(ternary)/A(8b)/E(8b)/G(8b)/U(8b)**: MNIST-LeNet5 (99.4%); SVHN-VGG8 (98.08%); ImageNet-AlexNet (top1-48.4%/top5-72.2%)
	W(ternary)/A(8b)/E(8b)/G(12b/8b/4b/2b)/U(12b/8b/4b/2b) on CIFAR10-VGG8: 92.43%/93.22%/71.78%/45.78%
QBP2 (2018) [180]	**W(1b)/A(2b/4b)/E(4b)/BN(8b)** on ImageNet-AlexNet (inflation 3x): top1-49.57%/53.3%
8b Training (2018) [184]	**W(FP8)/A(FP8)/E(FP8)/G(FP8)/U(FP8)/loss scale** on CIFAR10: CIFAR10-CNN (81.85%), CIFAR10-ResNet (92.21%); on BN50: BN50-DNN (39.92%)
	W(FP8)/A(FP8)/E(FP8)/G(FP8)/U(FP8)/loss scale on ImageNet: AlexNet (top1-57.55%), ResNet18/50 (top1-66.95%/71.72%)

irregular one even if at the same compression ratio [201], or compressing the energy-hungry layers first can reduce more energy [231]. The practical benefits rely on hardware support such as efficient hardware simulator that can quickly estimate the execution latency and energy. However, we still recommend to explicitly give more details of the pruning strategy (e.g., sparsification structure) to help reflect the execution performance, even if in an algorithm article where the hardware simulation or measurement is unavailable.

C. Testing Workloads

Besides the easy-to-compress small models (e.g., on MNIST, SVHN, and CIFAR10 data sets), most compression studies often benchmark AlexNet and VGG16 on ImageNet. However, as well known, these two networks are more redundant, which eases compression with insignificant accuracy degradation. Recent works begin to evaluate on modern networks such as ResNet and GoogleNet, and evidence that their compression is more intractable due to the natural compactness and the fragile C1 $\times$ 1 path. Furthermore, the compression of very deep networks on large data sets (e.g., >50 layers on ImageNet) has yet to be studied well (except for few works on quantization [149] and sparsification [209]). The layer-wise error accumulation might eventually cause a large accuracy loss or even nonconvergence. In the future, extending current compression methods to more compact or deep networks seems attractive but challenging.

As for RNNs, first, the testing models are relatively smaller and more divergent than those for CNNs. This brings difficulty in assessing compression effectiveness and conducting a fair comparison, which requires further extension and standardization. Second, the neuron sparsity is not fully utilized due to the lack of natural zero activations without the ReLU function. How to leverage the

Table A3 Accuracy of Representative Quantization Works for RNNs

References	Configuration and Accuracy
TernaryConnect (2016) [188]	**W(unlimited)/exp.** on VRNN2048×1: text8 (1.639 BPC); PTB (1.372 BPC)
	W(ternary) on TIDIGITS-GRU200×1: **det.** (top1-96.02%); **sto.** (top1-96.55%); **pow2** (top1-99.18%)
	W(unlimited)/exp. on TIDIGITS-GRU200×1: top1-99.1%; **W(ternary)/pow2** on WSJ-LSTM250×4: 10.49 WER
BNN (2017) [152]	PTB-VRNN2048×1, **W(3b)/A(3b)**: 1.81 BPC; **W(2b)/A(4b)**: 1.67 BPC; **W(3b)/A(4b)**: 1.11 BPC
	PTB-LSTM300×1, **W(2b)/A(3b)**: 220 PPW; **W(3b)/A(4b)**: 110 PPW; **W(4b)/A(4b)**: 100 PPW
Balanced DoReFa-Net (2016) [153]	**balanced**, PTB-GRU300×1, **W(1b)/A(2b)**: 285 PPW; **W(1b)**: 178 PPW; **W(2b)/A(2b)**: 150 PPW
	balanced, PTB-GRU300×1, **W(2b)/A(3b)**: 128 PPW; **W(3b)/A(3b)**: 109 PPW; **W(4b)/A(4b)**: 104 PPW
	balanced, PTB-LSTM300×1, **W(1b)/A(2b)**: 257 PPW; **W(1b)**: 198 PPW; **W(2b)/A(2b)**: 152 PPW
	balanced, PTB-LSTM300×1, **W(2b)/A(3b)**: 142 PPW; **W(3b)/A(3b)**: 120 PPW; **W(4b)/A(4b)**: 114 PPW
	balanced, IMDB-GRU512×1, **W(1b)/A(2b)**: 0.8684; **W(2b)/A(2b)**: 0.8708; **W(4b)/A(4b)**: 0.88132
	balanced, IMDB-LSTM512×1, **W(1b)/A(2b)**: 0.87888; **W(2b)/A(2b)**: 0.8812; **W(4b)/A(4b)**: 0.88476
Balanced DoReFa-Net (2017) [169]	**balanced**, PTB-GRU300×1, **W(2b)/A(2b)**: 142 PPW; **W(4b)/A(4b)**: 116 PPW
	balanced, PTB-LSTM300×1, **W(2b)/A(2b)**: 126 PPW; **W(2b)/A(3b)**: 123 PPW; **W(4b)/A(4b)**: 114 PPW
Neuron Increase (2017) [190]	PTB-LSTM300×1, **W(2b)/A(2b)**: 152.2 PPW; **W(2b)/A(4b)**: 138.2 PPW; **W(4b)/A(2b)**: 116.7 PPW; **W(4b)/A(4b)**: 115.2 PPW
	PTB-LSTM450×1, **W(2b)/A(2b)**: 148.2 PPW; **W(2b)/A(4b)**: 130.6 PPW; **W(4b)/A(2b)**: 111.7 PPW; **W(4b)/A(4b)**: 108.8 PPW
	PTB-LSTM1000×1, **W(2b)/A(2b)**: 148 PPW; **W(2b)/A(4b)**: 128.4 PPW; **W(4b)/A(2b)**: 113.1 PPW; **W(4b)/A(4b)**: 108.4 PPW
	TED-LIUM-LSTM2048×1-5 FCs, **FC W(4b)**: ↑5.9% WER; **FC W(4b)/↑25% FC-1-2-5 neuron**: ↑3.4% WER
	TED-LIUM-LSTM2048×1-5 FCs, **FC W(8b)**: ↑3.3% WER; **FC W(8b)/↑25%/50% FC-1-2-5 neuron**: ↑2.1% WER/↓1.7% WER
	TED-LIUM-LSTM2048×1-5 FCs, **LSTM A(4b)**: ↑2.1% WER; **LSTM A(8b)**: ↑0.4% WER
MP (2017) [177]	**W(FP16)/A(FP16)/E(FP16)**: WSJ-DeepSpeech2 (1.99 CER); Mandarin-DeepSpeech2 (15.01 CER)
HitNet (2018) [191]	**W(ternary)/A(ternary)**, PTB, LSTM300×1: 110.3 PPW; GRU300×1: 113.5 PPW
	W(ternary)/A(ternary), WikiText2, LSTM512×1: 126.72 PPW; GRU512×1: 132.49 PPW
	W(ternary)/A(ternary), Text8, LSTM1024×1: 169.1 PPW; GRU1024×1: 172.6 PPW
Alternating (2018) [144]	PTB-LSTM300×1/**pre-trained**, **W(2b)**: 103.1 PPW; **W(3b)**: 93.8 PPW; **W(4b)**: 91.4 PPW
	PTB-GRU300×1/**pre-trained**, **W(2b)**: 110.3 PPW; **W(3b)**: 97.3 PPW; **W(4b)**: 95.2 PPW
	PTB-LSTM300×1/**pre-trained**, **W(2b)/A(2b)**: 95.8 PPW; **W(2b)/A(3b)**: 91.9 PPW; **W(3b)/A(3b)**: 87.9 PPW
	PTB-GRU300×1/**pre-trained**, **W(2b)/A(2b)**: 101.2 PPW; **W(2b)/A(3b)**: 97 PPW; **W(3b)/A(3b)**: 92.9 PPW
	WikiText2-LSTM512×1/**pre-trained**, **W(2b)/A(2b)**: 106.1 PPW; **W(2b)/A(3b)**: 102.7 PPW; **W(3b)/A(3b)**: 98.7 PPW
	WikiText2-GRU512×1/**pre-trained**, **W(2b)/A(2b)**: 113.7 PPW; **W(2b)/A(3b)**: 110.2 PPW; **W(3b)/A(3b)**: 106.4 PPW
	Text8-LSTM1024×1/**pre-trained**, **W(2b)/A(2b)**: 108.8 PPW; **W(2b)/A(3b)**: 105.1 PPW; **W(3b)/A(3b)**: 98.8 PPW
	Text8-GRU1024×1/**pre-trained**, **W(2b)/A(2b)**: 124.5 PPW; **W(2b)/A(3b)**: 118.7 PPW; **W(3b)/A(3b)**: 114 PPW

Note Abbreviations: A-hidden state (usually not including the unbounded cellular state in LSTM), WER-word error rate, BPC-bits per character, PPW-perplexity per word, CER-character error rate.

rich saturation regions of the sigmoid and tanh functions for RNN compression seems interesting.

D. Influence on Adversarial Attack

Adversarial attack is one of the most important neural network attack models [301]: the adversaries manipulate the output of the neural network by inserting small perturbations into the input data that remain imperceptible to human vision. Taking the sparsification as an example, there are some recent publications discovering that sparsification helps improve the robustness of neural network models against adversarial attack. Previous work predicts that there may exist a sparse architecture that takes only the most relevant parts of the input data. It possibly reduces the space that an adversarial attack can affect, and therefore, the sparsification improves the model robustness. Marzi *et al.* [302] explored the impact of the sparsification against adversarial attack from both experimental and theoretical perspectives on linear classifiers and discuss the possibility of extending such technique to DNNs. Wang *et al.* [303] used defensive dropout to strengthen DNNs. Ji *et al.* [304] proposed a method to defend the adversarial examples by removing the unnecessary features. In addition to the research on sparsification techniques, Galloway *et al.* [305] empirically showed that stochastic quantization can also reduce the impact of attack based on experiments. In summary, existing works evidence the positive impact of sparsification and quantization techniques on neural network security. More insights from other compression approaches still remain quite open that merits further investigation.

E. Automatic Compression

In most compression methods, some configuration constraints (e.g., search space in NAS; rank value, bitwidth, or sparsity of different layers in decomposition, quantization, or sparsification, respectively) are given in advance

Table A4 Accuracy of Representative Sparsification Works for CNNs

References	Configuration, Sparsity, and Accuracy (default: top-1)
Deep Compression (2015) [185]	**W(Element)** on MNIST, LeNet-300-100(Paras-S:92%, Acc:98.42%), LeNet-5(Paras-S:92%, Acc:99.26%)
	W(Element) on ImageNet, AlexNet(Paras-S:89%, Acc:57.22%), VGG16(Paras-S:92.5%, ACC:68.83%)
Prune or Not (2017) [198]	**W(Element)** on ImageNet, Inception V3(Paras-S:87.5%, Acc:74.6%), MobileNet(Paras-S:50%, Acc:69.5%; Paras-S:90%, Acc:61.8%; Paras-S:95%, Acc:53.6%)
FDNP (2018) [199]	**W(Element)** (frequency-domain) on MNIST, LeNet-5(Paras-S:99.35%, Acc:99.08%); on ImageNet, AlexNet(Paras-S:95.57%, Acc:56.82%)
Filter Pruning (2016) [200]	**W(Filter)** on CIFAR10, VGG16(Paras-S:64%, OPs-S:34.2%, Acc:93.4%), ResNet56(Paras-S:13.7%, OPs-S:27.6%, Acc:93.06%), ResNet110(Paras-S:32.4%, OPs-S:38.6%, Acc:93.3%)
	W(Filter) on ImageNet, ResNet34(Paras-S:10.8$, OPs-S:24.2%, Acc:72.17%)
meProp (2017) [202]	**E(Element)** on MNIST, MLP500×2(E-S:95%, Acc:98.32%; E-S:98%, BP-Speedup(CPU Xeon):41.7x, Acc:98%), MLP 500×5(E-S:95%, Acc:98.21%)
	E(Element) on PTB, MLP500×2(E-S:92%, Acc:91.99%; E-S:96%, BP-Speedup(CPU Xeon):18.6x, Acc:90.01%)
DGC (2017) [203]	**G(Element)** on ImageNet, AlexNet(G-S:99.83%, Acc:58.2%), ResNet50(G-S:99.64%, Acc:76.15%)
Taylor Expansion (2016) [204]	**A(FM)** on Flowers-102, AlexNet(OPs-S:72.6%, Speedup(GPU Titan X/TX1):2.4x/2.3x, Acc-top5: 79.8%; OPs-S:86.3%, Speedup(GPU Titan X/TX1):3.2x/2.9x, Acc-top5: 74.1%)
	A(FM) on ImageNet, VGG16(OPs-S:62.86%, Speedup(GPU Titan X/TX1):2.2x/2.5x, Acc-top5: 87%; OPs-S:74.16%, Speedup(GPU Titan X/TX1):3.4x/3.3x, Acc-top5: 84.5%)
	A(FM) on nvGesture, R3DCNN(OPs-S:92.06%, Speedup(GPU GT 730M):5.2x, Acc-top5: 78.2%)
ThiNet (2017) [205][1]	**A(FM)** on ImageNet, VGG16(Paras-S:93.99%, OPs-S:69.81%, Speedup(GPU M40):2.75x, Acc:67.34%; Paras-S:99.05%, OPs-S:93.5%, Speedup(GPU M40):7x, Acc:59.34%)
	A(FM) on ImageNet, ResNet50(Paras-S:51.56%, OPs-S:55.83%, Speedup(GPU M40):1.25x, Acc:71.01%; Paras-S:66.12%, OPs-S:71.5%, Speedup(GPU M40):1.33x, Acc:68.42%)
	A(FM) on CUB-200, VGG16(Paras-S:94.14%, OPs-S:69.8%, Acc:69.43%; Paras-S:99.17%, OPs-S:93.5%, Acc:65.45%)
	A(FM) on Indoor-67, VGG16(Paras-S:94.17%, OPs-S:69.8%, Acc:70.22%; Paras-S:99.2%, OPs-S:93.5%, Acc:62.84%)
Channel Pruning (2017) [206]	**A(FM)** on ImageNet, VGG16(OPs-S:50%/80%, Acc-top5:89.9%/88.2%; OPs-S:75%, Speedup(GPU Titan X):2.5x, Acc-top5:88.9%)
	A(FM), on CIFAR10, ResNet56(OPs-S:50%, Acc:91.8%); on ImageNet, ResNet50(OPs-S:50%, Acc-top5:90.8%), Xception(OPs-S:50%, Acc-top5:91.8%)
Slimming (2017) [207]	**A(FM)** on MNIST, LeNet-500-300(Paras-S:84.4%, Acc:98.51%); on ImageNet, VGG-A(Paras-S:82.5%, OPs-S:30.4%, Acc:63.34%)
	A(FM) on SVHN, VGGNet(Paras-S:84.8%, OPs-S:50.1%, Acc:97.94%), DenseNet40(Paras-S:56.6%, OPs-S:49.8%, Acc:98.19%), ResNet164(Paras-S:34.3%, OPs-S:54.9%, Acc:98.19%)
	A(FM) on CIFAR10, VGGNet(Paras-S:88.5%, OPs-S:51%, Acc:93.8%), DenseNet40(Paras-S:65.2%, OPs-S:55%, Acc:94.35%), ResNet164(Paras-S:35.2%, OPs-S:44.9%, Acc:94.73%)
	A(FM) on CIFAR100, VGGNet(Paras-S:75.1%, OPs-S:37.1%, Acc:73.48%), DenseNet40(Paras-S:54.6%, OPs-S:47.1%, Acc:74.28%), ResNet164(Paras-S:29.7%, OPs-S:50.6%, Acc:76.09%)
NISP (2018) [208]	**A(FM)**, CIFAR10, ResNet56/110(Paras-S:42.6%/43.25%, OPs-S:43.61%/43.78%, Acc:↓0.03%/↓0.18%); ImageNet, AlexNet(Paras-S:33.77%, OPs-S:67.85%, Acc:↓1.43%)
	A(FM) on ImageNet, ResNet34/50(Paras-S:43.68%/43.82%, OPs-S:43.76%/44.01%, Acc:↓0.92%/↓0.89%), GoogleNet(Paras-S:33.76%, OPs-S:58.34%, Acc:↓0.21%)
ISTA Pruning (2018) [209]	**A(FM)** on CIFAR10, ConvNet(Paras-S:84.4%/89.6%/92.7%, Acc:89.5%/87.6%/86%), ResNet20(Paras-S:37.2%/67.8%, Acc:90.9%/88.8%)
	A(FM) on ImageNet, ResNet101(Paras-S:47%/61%, Acc:75.27%/74.56%)
AutoPrunner (2018) [210]	**A(FM)** on ImageNet, VGG16(OPs-S:73.6%, Acc:68.43%), ResNet50(OPs-S:51.3%, Acc:74.22%; OPs-S:65.8%, Acc:72.53%)
	A(FM) on CUB-200, VGG16(OPs-S:68.9%, Acc:73.45%; OPs-S:91.5%, Acc:65.06%)
Layer-compensated (2018) [213]	**W(Filter)** on CIFAR10, ResNet56(OPs-S:30%/40%/50%, Acc:93.79%/93.65%/93.41%); **W(Filter)/(Naive)** on ImageNet, ResNet50(OPs-S:50%, Acc:74.46%)
AMC (2018) [214]	**W(Channel)** on CIFAR10, Plain-20(OPs-S:50%, Acc:90.2%), ResNet50(Paras-S:60%, Acc:93.55%), ResNet56(OPs-S:50%, Acc:91.9%)
	W(Element) on ImageNet, ResNet50(Paras-S:80%, Acc:76.11%)
	W(Channel) on ImageNet, VGG16(OPs-S:80%, Acc:69.1%), MobileNet V1(OPs-S:50%/60%, Acc:70.5%/69.2%), MobileNet V2(OPs-S:50%, Acc:70.8%)
	W(Channel) on ImageNet, MobileNet V1(OPs-S:50%, Speedup(GPU Titan Xp/Google Pixel 1) 1.43x/1.81x, Acc:70.5%)
	W(Channel) on ImageNet, MobileNet V1(Time-50%(object), Speedup(GPU Titan Xp/Google Pixel 1) 1.53x/1.95x, Acc:70.2%)
NestedNet (2018) [215]	CIFAR10, **W(Element)**, ResNet56(Paras-S:50%/66.7%, Acc:93.4%/92.8%); **W(Kernel)**, ResNet56(Paras-S:50%, Acc:92.9%)
	CIFAR10, **W(Kernel)**, WRN-32-4(Paras-S:93.6%, Acc:67.1%); **W(Layer)**, WRN-32-4(Paras-S:63%, Acc:74.3%); **W(Kernel/Layer)**, WRN-32-4(Paras-S:97.6%, Acc:64.3%)
Hybrid Pruning (2018) [216]	**W(Filter)**, on CIFAR10, ResNet56(Paras-S:59%, Acc:92.67%); on ImageNet, ResNet50(Paras-S:32.5%, Acc:74.87%)
	W(Filter & Element), on CIFAR10, ResNet56(Paras-S:78%, Acc:92.48%); on ImageNet, ResNet50(Paras-S:72.9%, Acc:74.32%)
NeST (2019) [217]	**W(Element)/A(FM)** on MNIST, LeNet-300-100(Paras-S:98.58%, OPs-S:98.74%, Acc:98.42%), LeNet-5(Paras-S:98.65%, OPs-S:97.71%, Acc:99.23%)
	W(Element)/A(FM) on ImageNet, AlexNet(Paras-S:93.63%, OPs-S:78.26%, Acc:57.24%)
	W(Element)/A(FM) on ImageNet, VGG16(Paras-S:92.81%, OPs-S:79.59%, Acc:71.94%; Paras-S:96.69%, OPs-S:88.37%, Acc:69.28%)
Channel Pruning (2018) [218]	**W(Kernel)** on SVHN, VGG16(Paras-S:88.4%, OPs-S:66.4%, Acc:96.13%), ResNet50(Paras-S:64.8%, OPs-S:45.3%, Acc:96.52%)
	W(Kernel) on CIFAR10, VGG16(Paras-S:84%, OPs-S:56.2%, Acc:92.74%), ResNet50(Paras-S:74.9%, OPs-S:49.1%, Acc:94.15%)
	W(Kernel) on CIFAR100, VGG16(Paras-S:64.8%, OPs-S:37.4%, Acc:72.01%), ResNet50(Paras-S:64%, OPs-S:45.3%, Acc:74.1%)
	W(Kernel) on ImageNet, VGG16(Paras-S:5.52%, OPs-S:69.32%, Acc:70.42%)
Joint Sparsity (2018) [221][2]	**W(Kernel)** on ImageNet, ResNet18(Paras-S:80%, Ops-S:64.3%, Acc:67.8%)
Synaptic Strength (2018) [222]	**W(Kernel)** on CIFAR10, VGGNet(Paras-S:96%, Ops-S:76.38%, Acc:93.77%), DenseNet40(Paras-S:80%, Ops-S:74.82%, Acc:94.42%)
	W(Kernel), on CIFAR10, ResNet18(Paras-S:90%/95%, Ops-S:75.32%/84.14%, Acc:94.94%/94.2%); on ImageNet, ResNet50(Paras-S:76.95%, Acc:74.68%)
ADMM (2018) [7, 223–225]	**W(Element)** on MNIST, LeNet-300-100(Paras-S:95.63%, Acc:98.4%), LeNet5(Paras-S:98.8%/99.4%, Acc:99.2%/99%)
	W(Element) on ImageNet, AlexNet(Conv2-5)(Paras-S:83.05%/93.79%/95.89%/96.63%/97.53%, Acc:58.67%/57.5%/57.1%/56.8%/55.38%), AlexNet(Paras-S:96.7%, Acc-top5:80.2%)
	W(Element) on ImageNet, VGG16(Paras-S:96.7%/97%, Acc-top5:88.7%/88.2%)
	W(Element) on ImageNet, ResNet50(Paras-S:85.9%/89.1%/94.3%, Acc:↓0.0%/↓0.3%/↓0.8%), ResNet50(Paras-S:94.9%, Acc-top5:92%)
	W(Fiber/Filter) on ImageNet, AlexNet(Conv2-5)(Paras-S:33.33%/79.17%/84.38%/92.42%, Acc:59.04%/57.47%/56.65%/55.33%)

Note Abbreviations: Paras-parameters, OPs-operations, S-sparsity or compression percentage, Acc-accuracy.
[1]For VGG16, the last max pooling before FC layer is replaced with a global average pooling (GAP) layer for parameter reduction.
[2]We only list its results in common spatial-domain convolution without quantization for fair comparison with others.

for implementation simplicity. This hand-crafted configuration relies on domain experiences and limits compression performance. Automatic compression is a promising direction to free human labor and search the optimal solution. RL seems an effective way toward this goal, the learning results of which can give useful design insights. Although existing RL-based trials [74], [176], [214] have demonstrated superior performance compared with conventional solutions and have presented good compatibility with practical rewards of hardware execution cost (e.g., latency and energy), they mainly use the naive heuristic compression criteria. How to combine with the optimization methods to boost compression performance is an interesting topic.

F. Framework-Level Support

Deep learning frameworks significantly improve the programmability for algorithm designers, so that the programmers can simply explore models using the basic

Table A5 Accuracy of Representative Sparsification Works for RNNs

References	Configuration, Sparsity, and Accuracy
Soft Pruning (2017) [228]	**W(Element)** on Deep Speech 2, RNN1760×7(Paras-S:87.61%, CER:↓20.71%), RNN2560×7(Paras-S:92.13%, CER:↑0.75%), GRU2560×3(Paras-S:88.7%, CER:↓13.82%)
Prune or Not (2017) [198]	**W(Element)** on PTB, LSTM650×2(Paras-S:80%, P:83.87%; Paras-S:95%, PPW:96.3%; Paras-S:97.5%, PPW:113.6%)
	W(Element) on PTB, LSTM1500×2(Paras-S:80%, PPW:77.52%; Paras-S:95%, PPW:87.83%; Paras-S:97.5%, PPW:103.2%)
	W(Element) on NMT, LSTM1500×2(Paras-S:80%, EN-DE BLEU:26.86%, DE-EN BLEU:29.5%; Paras-S:90%, EN-DE BLEU:26.19%, DE-EN BLEU:28.81%)
ISS (2017) [232]	**W(Vector)** on PTB, LSTM1500×2(Paras-S:61.82%, Ops-S:80.64%, Speedup(CPU Xeon E5):7.1x, PPW:76.03)
	W(Vector) on PTB, LSTM1500×2(Paras-S:66.97%, Ops-S:86.63%, Speedup(CPU Xeon E5):10.59x, PPW:78.65)
	W(Vector) on PTB, RHN830×10(Paras-S:52.77%, PPW:65.4; Paras-S:75.74%, PPW:71.2)
	W(Vector) on SQuAD, BiDAF(Paras-S:44.98%, Speedup(CPU Xeon E5):1.37x, EM:66.59, F1:76.4; Paras-S:62.45%, Speedup(CPU Xeon E5):1.77x, EM:64.81, F1:75.22)
meProp (2017) [202]	**E(Element)** on WSJ, LSTM500×1(E-S:96%, Acc:97.31%; E-S:98%, BP-Speedup(CPU Xeon):69.2x, Acc:97.25%)
DGC (2017) [203]	**G(Element)** on PTB, LSTM1500×2(G-S:99.78%, PPW:72.24); on LibriSpeech, GRU1200×7(G-S:99.85%, WER(clean/other):9.06%/27.04%)
Sparse Gate Gradients (2018) [233]	**E(Block)** on PTB, LSTM512×2(E-S:50%, PPW:96.41%; E-S:75%, PPW:108.05); on MSCOCO, Inception V3-LSTM512×1(E-S:50%, BLEU-4:30.6)
	E(Block) on NMT, LSTM512×6(E-S:50%, BP-Speedup(GPU Tesla V100):1.36x, BLEU:19.92%)

Note Abbreviations: RHN-recurrent highway network, Paras-parameters, Ops-operations, S-sparsity or compression percentage, PPW-perplexity per word, BLEU-BLEU score, CER-character error rate, WER-word error rate, EM-ExactMatch score, F1-F1 score.

operator library. For example, Novikov *et al.* [306] proposed a Tensorflow library for TT decomposition which makes machine learning relying on tensor decomposition easier to implement. By observing the effectiveness and broad use of sparsification techniques, many frameworks, libraries, and automated optimization tools are developed. There are also some commonly used libraries to accelerate the sparse matrix operations, such as SparseBLAS/cuSPARSE by NVIDIA and MKL by Intel. Moreover, Tensorflow provides a quantization library for quantization-aware training and inference [307]. Xilinx further proposes the FINN-R framework, an end-to-end tool to explore the design space of quantization techniques and to automate the design of inference engine on FPGA [308]. In the future, more framework-level supports are expected to ease the training with given performance requirements.

G. Hardware-Level Support

Another reason that the compression ratio cannot accurately reflect practical benefits is because the architecture matters. To fully utilize the resources, GPUs favor regular, large, and high-precision operations. Although compact models introduce tight connections and operations, they still remain the dense execution pattern and normal data precision which can be supported by general-purpose processors. For other compression approaches such as high-dimensional tensor computation, sparse processing, and low-precision operation, they cannot obtain ideal acceleration on general-purpose processors. Therefore, designing efficient specialized architectures for compressed neural networks becomes a promising solution to take full advantage of their acceleration potential. Recently, as aforementioned, various accelerators report impressive performance via the incorporation of compression techniques. Whereas, when comparing with other architectures, most works do not present a very fair comparison. For example, sometimes only the running performance is emphasized while the accuracy result is understated or even skipped. Another issue is that we should use the "variable-controlling" method to clearly show how many benefits come from the model compression rather than the device improvement, but most comparisons often conceal this point.

An interesting topic is to answer which compression approach can bring the most benefits for hardware performance. In the future, this might be achievable through a series of comparison experiments on a similar substrate but amenable to be compatible with different compression techniques. From the existing works, we give a summary here. For tensor decomposition, the specialized accelerators [11] are rare. Although the regular computation pattern can be maintained for efficient running, the low effectiveness on Conv layers impedes its wide deployment. Tensor decomposition might be useful for compression of more high-dimensional weight tensor such as that in 3DCNNs, which demands more practice. It looks like the performance improvement benefited from data quantization is the most significant (see Table 16), since the memory and compute costs greatly decrease as the bitwidth decreases. The remaining of the regular execution pattern in data quantization further simplifies its architectural design. However, the significant accuracy loss in the cases of extremely low-bit quantization must be improved in the future. For network sparsification, the index overhead and compute/access irregularity are the most challenging issues. Even though the structured sparsification can help, how to maintain accuracy also needs more efforts. The emerging ADMM-based optimization method [7], [165], [224] seems promising in low-bit quantization and structured sparsification with high accuracy; the learnable compression [146] and the automatic compression [176], [214] are also attractive. At last, we believe the integration of the aggressive joint-way compression into hardware design is a promising topic to pursue ultrahigh execution performance.

In summary, although much work has been done, the neural network compression field still affords opportunities in following directions: 1) joint-way compression for ultracompact models and efficient accelerators; 2) lightweight compression for training; 3) fair benchmarks to

compare different algorithms and architectures; 4) automatic compression to find the optimal solution; 5) framework- and hardware-level supports for easy programming and efficient execution; and 6) the influence of model compression (e.g., on security). These opportunities require innovations at various levels and need continuous efforts from the entire community. We anticipate that a close interaction between these wide levels would give birth to powerful systems for embedded AI applications. ■

APPENDIX A
TENSOR DECOMPOSITION

See Fig. A1.

APPENDIX B
DATA QUANTIZATION

See Tables A1–A3.

APPENDIX C
NETWORK SPARSIFICATION

See Tables A4 and A5.

Acknowledgment

The authors would like to thank S. Wu, D. Wang, L. Liang, H. Cai, and X. Hu for their contributions in preparing the contents of compact model, tensor decomposition and compact RNNs, data collection, neural architecture search, and influence on adversarial attack, respectively.

REFERENCES

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

[2] L. Xiao, Y. Bahri, J. Sohl-Dickstein, S. S. Schoenholz, and J. Pennington, "Dynamical isometry and a mean field theory of CNNs: How to train 10,000-layer vanilla convolutional neural networks," 2018, *arXiv:1806.05393*. [Online]. Available: http://arxiv.org/abs/1806.05393

[3] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 4510–4520.

[4] A. Novikov, D. Podoprikhin, A. Osokin, and D. P. Vetrov, "Tensorizing neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2015, pp. 442–450.

[5] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or −1," 2016, *arXiv:1602.02830*. [Online]. Available: http://arxiv.org/abs/1602.02830

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

[7] A. Ren *et al.*, "ADMM-NN: An algorithm-hardware co-design framework of DNNs using alternating direction methods of multipliers," in *Proc. 24th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2019, pp. 925–938.

[8] Y. Chen *et al.*, "DaDianNao: A machine-learning supercomputer," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2014, pp. 609–622.

[9] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proc. ACM/IEEE 44th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2017, pp. 1–12.

[10] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," 2018, *arXiv:1807.07928*. [Online]. Available: http://arxiv.org/abs/1807.07928

[11] C. Deng, F. Sun, X. Qian, J. Lin, Z. Wang, and B. Yuan, "TIE: Energy-efficient tensor train-based inference engine for deep neural network," in *Proc. 46th Int. Symp. Comput. Archit.*, 2019, pp. 264–278.

[12] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H.-J. Yoo, "UNPU: An energy-efficient deep neural network accelerator with fully variable weight bit precision," *IEEE J. Solid-State Circuits*, vol. 54, no. 1, pp. 173–185, Jan. 2019.

[13] X. Zhou *et al.*, "Cambricon-S: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2018, pp. 15–28.

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

[15] Y. Guo, "A survey on methods and theories of quantized neural networks," 2018, *arXiv:1808.04752*. [Online]. Available: http://arxiv.org/abs/1808.04752

[16] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, "Model compression and acceleration for deep neural networks: The principles, progress, and challenges," *IEEE Signal Process. Mag.*, vol. 35, no. 1, pp. 126–136, Jan. 2018.

[17] R. Pilipović, P. Bulić, and V. Risojević, "Compression of convolutional neural networks: A short survey," in *Proc. 17th Int. Symp. INFOTEH-JAHORINA (INFOTEH)*, Mar. 2018, pp. 1–6.

[18] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[19] K. Cho *et al.*, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," 2014, *arXiv:1406.1078*. [Online]. Available: http://arxiv.org/abs/1406.1078

[20] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.

[21] W. Luo, Y. Li, R. Urtasun, and R. Zemel, "Understanding the effective receptive field in deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 4898–4906.

[22] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 2818–2826.

[23] B. Wu *et al.*, "Shift: A zero FLOP, zero parameter alternative to spatial convolutions," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 9127–9135.

[24] M. Holschneider, R. Kronland-Martinet, J. Morlet, and P. Tchamitchian, "A real-time algorithm for signal analysis with the help of the wavelet transform," in *Wavelets*. Berlin, Germany: Springer, 1990, pp. 286–297.

[25] F. Yu and V. Koltun, "Multi-scale context aggregation by dilated convolutions," in *Proc. ICLR*, May 2016.

[26] J. Dai *et al.*, "Deformable convolutional networks," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Oct. 2017, pp. 764–773.

[27] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. Int. Conf. Learn. Represent.*, 2015.

[28] R. Kumar Srivastava, K. Greff, and J. Schmidhuber, "Highway networks," 2015, *arXiv:1505.00387*. [Online]. Available: http://arxiv.org/abs/1505.00387

[29] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2015, pp. 1–9.

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

[31] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proc. CVPR*, 2017, vol. 1, no. 2, pp. 4700–4708.

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

[33] S. Xie, R. Girshick, P. Dollar, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 5987–5995.

[34] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 1251–1258.

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

[36] G. Huang, S. Liu, L. van der Maaten, and K. Q. Weinberger, "CondenseNet: An efficient densenet using learned group convolutions," *Group*, vol. 3, no. 12, p. 11, 2017.

[37] X. Zhang, X. Zhou, M. Lin, and J. Sun, "ShuffleNet: An extremely efficient convolutional neural network for mobile devices," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 6848–6856.

[38] H. Gao, Z. Wang, and S. Ji, "ChannelNets: Compact and efficient convolutional neural networks via channel-wise convolutions," in *Proc. Adv. Neural Inf. Process. Syst.*, 2018, pp. 5199–5207.

[39] T. Zhang, G.-J. Qi, B. Xiao, and J. Wang, "Interleaved group convolutions," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Oct. 2017, pp. 4373–4382.

[40] G.-B. Zhou, J. Wu, C.-L. Zhang, and Z.-H. Zhou, "Minimal gated unit for recurrent neural networks," *Int. J. Autom. Comput.*, vol. 13, no. 3, pp. 226–234, Jun. 2016.

[41] Z. Wu and S. King, "Investigating gated recurrent networks for speech synthesis," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, Mar. 2016, pp. 5140–5144.

[42] J. van der Westhuizen and J. Lasenby, "The unreasonable effectiveness of the forget gate," 2018, *arXiv:1804.04849*. [Online]. Available: http://arxiv.org/abs/1804.04849

[43] D. Neil, M. Pfeiffer, and S.-C. Liu, "Phased LSTM: Accelerating recurrent network training for long or event-based sequences," in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 3882–3890.

[44] A. G. Howard *et al.*, "MobileNets: Efficient convolutional neural networks for mobile vision applications," 2017, *arXiv:1704.04861*. [Online]. Available: http://arxiv.org/abs/1704.04861

[45] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," in *Proc. ICLR*, Apr. 2017.

[46] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with LSTM," *Neural Comput.*, vol. 12, no. 10, pp. 2451–2471, Oct. 2000.

[47] R. Jozefowicz, W. Zaremba, and I. Sutskever, "An empirical exploration of recurrent network architectures," in *Proc. Int. Conf. Mach. Learn.*, 2015, pp. 2342–2350.

[48] S. Li, W. Li, C. Cook, C. Zhu, and Y. Gao, "Independently recurrent neural network (IndRNN): Building a longer and deeper RNN," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 5457–5466.

[49] J. Bradbury, S. Merity, C. Xiong, and R. Socher, "Quasi-recurrent neural networks," *Proc. ICLR*, Apr. 2017.

[50] S. I. W. H. D. Tao Lei, Y. Zhang, and Y. Artzi, "Simple recurrent units for highly parallelizable recurrence," in *Proc. EMNLP*, Oct. 2018.

[51] M. Arjovsky, A. Shah, and Y. Bengio, "Unitary evolution recurrent neural networks," in *Proc. Int. Conf. Mach. Learn.*, 2016, pp. 1120–1128.

[52] H. Sak, A. Senior, and F. Beaufays, "Long short-term memory recurrent neural network architectures for large scale acoustic modeling," in *Proc. INTERSPEECH*, Sep. 2014.

[53] O. Kuchaiev and B. Ginsburg, "Factorization tricks for LSTM networks," 2017, *arXiv:1703.10722*. [Online]. Available: http://arxiv.org/abs/1703.10722

[54] Y. Wu *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," 2016, *arXiv:1609.08144*. [Online]. Available: http://arxiv.org/abs/1609.08144

[55] S. Zhang *et al.*, "Architectural complexity measures of recurrent neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 1822–1830.

[56] N. Kalchbrenner, I. Danihelka, and A. Graves, "Grid long short-term memory," 2015, *arXiv:1507.01526*. [Online]. Available: http://arxiv.org/abs/1507.01526

[57] Z. He, S. Gao, L. Xiao, D. Liu, H. He, and D. Barber, "Wider and deeper, cheaper and faster: Tensorized LSTMs for sequence learning," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 2017, pp. 1–11.

[58] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 8697–8710.

[59] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean, "Efficient neural architecture search via parameter sharing," 2018, *arXiv:1802.03268*. [Online]. Available: http://arxiv.org/abs/1802.03268

[60] H. Liu, K. Simonyan, and Y. Yang, "DARTS: Differentiable architecture search," 2018, *arXiv:1806.09055*. [Online]. Available: http://arxiv.org/abs/1806.09055

[61] D. R. So, C. Liang, and Q. V. Le, "The evolved transformer," 2019, *arXiv:1901.11117*. [Online]. Available: http://arxiv.org/abs/1901.11117

[62] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," in *Proc. AAAI Conf. Artif. Intell.*, vol. 33, Jul. 2019, pp. 4780–4789.

[63] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," 2017, *arXiv:1710.05941*. [Online]. Available: http://arxiv.org/abs/1710.05941

[64] B. Baker, O. Gupta, N. Naik, and R. Raskar, "Designing neural network architectures using reinforcement learning," in *Proc. ICLR*, Apr. 2017.

[65] Z. Zhong, J. Yan, W. Wu, J. Shao, and C.-L. Liu, "Practical block-wise neural network architecture generation," 2017, *arXiv:1708.05552*. [Online]. Available: http://arxiv.org/abs/1708.05552

[66] C. Liu *et al.*, "Progressive neural architecture search," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2018, pp. 19–34.

[67] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu, "Hierarchical representations for efficient architecture search," 2017, *arXiv:1711.00436*. [Online]. Available: http://arxiv.org/abs/1711.00436

[68] H. Cai, T. Chen, W. Zhang, Y. Yu, and J. Wang, "Efficient architecture search by network transformation," in *Proc. AAAI*, Feb. 2018.

[69] H. Cai, J. Yang, W. Zhang, S. Han, and Y. Yu, "Path-level network transformation for efficient architecture search," 2018, *arXiv:1806.02639*. [Online]. Available: http://arxiv.org/abs/1806.02639

[70] T. Elsken, J. Hendrik Metzen, and F. Hutter, "Efficient multi-objective neural architecture search via lamarckian evolution," 2018, *arXiv:1804.09081*. [Online]. Available: http://arxiv.org/abs/1804.09081

[71] A. Canziani, A. Paszke, and E. Culurciello, "An analysis of deep neural network models for practical applications," 2016, *arXiv:1605.07678*. [Online]. Available: http://arxiv.org/abs/1605.07678

[72] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han, "Once for all: Train one network and specialize it for efficient deployment," in *Proc. ICLR*, Apr. 2020. [Online]. Available: https://openreview.net/forum?id=HylxE1HKwS

[73] A. Brock, T. Lim, J. M. Ritchie, and N. Weston, "SMASH: One-shot model architecture search through HyperNetworks," 2017, *arXiv:1708.05344*. [Online]. Available: http://arxiv.org/abs/1708.05344

[74] H. Cai, L. Zhu, and S. Han, "ProxylessNAS: Direct neural architecture search on target task and hardware," 2018, *arXiv:1812.00332*. [Online]. Available: http://arxiv.org/abs/1812.00332

[75] V. Klema and A. Laub, "The singular value decomposition: Its computation and some applications," *IEEE Trans. Autom. Control*, vol. AC-25, no. 2, pp. 164–176, Apr. 1980.

[76] T. N. Sainath, B. Kingsbury, V. Sindhwani, E. Arisoy, and B. Ramabhadran, "Low-rank matrix factorization for deep neural network training with high-dimensional output targets," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process.*, May 2013, pp. 6655–6659.

[77] Y. Liu, S. Yang, P. Wu, C. Li, and M. Yang, "L_1-norm low-rank matrix decomposition by neural networks and mollifiers," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 27, no. 2, pp. 273–283, Feb. 2016.

[78] C. Tai, T. Xiao, Y. Zhang, and X. Wang, "Convolutional neural networks with low-rank regularization," in *Proc. ICLR*, May 2016.

[79] M. Denil, B. Shakibi, L. Dinh, M. Ranzato, and N. D. Freitas, "Predicting parameters in deep learning," in *Proc. Adv. Neural Inf. Process. Syst.*, 2013, pp. 2148–2156.

[80] C. H. Li and S. C. Park, "Neural network for text classification based on singular value decomposition," in *Proc. 7th IEEE Int. Conf. Comput. Inf. Technol. (CIT)*, Oct. 2007, pp. 47–52.

[81] J. Xue, J. Li, D. Yu, M. Seltzer, and Y. Gong, "Singular value decomposition based low-footprint speaker adaptation and personalization for deep neural network," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, May 2014, pp. 6359–6363.

[82] H. T. Huynh and Y. Won, "Training single hidden layer feedforward neural networks by singular value decomposition," in *Proc. 4th Int. Conf. Comput. Sci. Converg. Inf. Technol.*, 2009, pp. 1300–1304.

[83] M. Masana, J. V. D. Weijer, L. Herranz, A. D. Bagdanov, and J. M. Alvarez, "Domain-adaptive deep network compression," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Oct. 2017, pp. 4289–4297.

[84] T. Kumamoto, M. Suzuki, and H. Matsueda, "Singular-value-decomposition analysis of associative memory in a neural network," *J. Phys. Soc. Jpn.*, vol. 86, no. 2, Feb. 2017, Art. no. 024005.

[85] T. Deb, A. K. Ghosh, and A. Mukherjee, "Singular value decomposition applied to associative memory of Hopfield neural network," *Mater. Today, Proc.*, vol. 5, no. 1, pp. 2222–2228, 2018.

[86] S. Xue, H. Jiang, L. Dai, and Q. Liu, "Unsupervised speaker adaptation of deep neural network based on the combination of speaker codes and singular value decomposition for speech recognition," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, Apr. 2015, pp. 4555–4559.

[87] Z. Zou and Z. Shi, "Ship detection in spaceborne optical image with SVD networks," *IEEE Trans. Geosci. Remote Sens.*, vol. 54, no. 10, pp. 5832–5845, Oct. 2016.

[88] X. Zhang, J. Zou, X. Ming, K. He, and J. Sun, "Efficient and accurate approximations of nonlinear convolutional networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2015, pp. 1984–1992.

[89] X. Zhang, J. Zou, K. He, and J. Sun, "Accelerating very deep convolutional networks for classification and detection," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 38, no. 10, pp. 1943–1955, Oct. 2016.

[90] K. Shim, M. Lee, I. Choi, Y. Boo, and W. Sung, "SVD-softmax: Fast softmax approximation on large vocabulary neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 5463–5473.

[91] M. Yu *et al.*, "GradiVeQ: Vector quantization for bandwidth-efficient gradient aggregation in distributed CNN training," in *Proc. Adv. Neural Inf. Process. Syst.*, 2018, pp. 5125–5135.

[92] N. Kishore Kumar and J. Schneider, "Literature survey on low rank approximation of matrices," *Linear Multilinear Algebra*, vol. 65, no. 11, pp. 2212–2244, Dec. 2016.

[93] C. Trabelsi *et al.*, "Deep complex networks," in *Proc. ICLR*, Apr. 2018.

[94] I. Aizenberg, A. Luchetta, and S. Manetti, "A modified learning algorithm for the multilayer neural network with multi-valued neurons based on the complex QR decomposition," *Soft Comput.*, vol. 16, no. 4, pp. 563–575, Aug. 2011.

[95] C. Thurau, K. Kersting, and C. Bauckhage, "Deterministic CUR for improved large-scale data analysis: An empirical study," in *Proc. SIAM Int. Conf. Data Mining*, Dec. 2013, pp. 684–695.

[96] A. Gittens and M. W. Mahoney, "Revisiting the Nyström method for improved large-scale machine learning," *J. Mach. Learn. Res.*, vol. 17, no. 1, pp. 3977–4041, 2016.

[97] N. Lee and A. Cichocki, "Fundamental tensor operations for large-scale data analysis using tensor network formats," *Multidimensional Syst. Signal Process.*, vol. 29, no. 3, pp. 921–960, Mar. 2017.

[98] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Rev.*, vol. 51, no. 3, pp. 455–500, Aug. 2009.

[99] A. Cichocki *et al.*, "Tensor decompositions for signal processing applications: From two-way to multiway component analysis," *IEEE Signal Process. Mag.*, vol. 32, no. 2, pp. 145–163, Mar. 2015.

[100] Y.-D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin, "Compression of deep convolutional neural networks for fast and low power mobile applications," in *Proc. ICLR*, May 2016.

[101] J.-T. Chien and Y.-T. Bao, "Tensor-factorized neural networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 29, no. 5, pp. 1998–2011, May 2018.

[102] J. Kossaifi, A. Khanna, Z. Lipton, T. Furlanello, and A. Anandkumar, "Tensor contraction layers for parsimonious deep nets," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. Workshops (CVPRW)*, Jul. 2017, pp. 26–32.

[103] J. Kossaifi, Z. C. Lipton, A. Khanna, T. Furlanello, and A. Anandkumar, "Tensor regression networks," 2017, *arXiv:1707.08308*. [Online]. Available: http://arxiv.org/abs/1707.08308

[104] M. Janzamin, H. Sedghi, and A. Anandkumar, "Beating the perils of non-convexity: Guaranteed training of neural networks using tensor

methods," 2015, *arXiv:1506.08473*. [Online]. Available: http://arxiv.org/abs/1506.08473

[105] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky, "Speeding-up convolutional neural networks using fine-tuned CP-decomposition," in *Proc. ICLR*, May 2015.

[106] M. Astrid and S.-I. Lee, "CP-decomposition with tensor power method for convolutional neural networks compression," in *Proc. IEEE Int. Conf. Big Data Smart Comput. (BigComp)*, Feb. 2017, pp. 115–118.

[107] Y. Chen, X. Jin, B. Kang, J. Feng, and S. Yan, "Sharing residual units through collective tensor factorization to improve deep neural networks," in *Proc. 27th Int. Joint Conf. Artif. Intell.*, Jul. 2018, pp. 635–641.

[108] D. T. Tran, A. Iosifidis, and M. Gabbouj, "Improving efficiency in convolutional neural networks with multilinear filters," *Neural Netw.*, vol. 105, pp. 328–339, Sep. 2018.

[109] K. T. Schütt, F. Arbabzadah, S. Chmiela, K. R. Müller, and A. Tkatchenko, "Quantum-chemical insights from deep tensor neural networks," *Nature Commun.*, vol. 8, no. 1, Jan. 2017, Art. no. 13890.

[110] M. Zhou, Y. Liu, Z. Long, L. Chen, and C. Zhu, "Tensor rank learning in CP decomposition via convolutional neural network," *Signal Process., Image Commun.*, vol. 73, pp. 12–21, Apr. 2019.

[111] S. Oymak and M. Soltanolkotabi, "End-to-end learning of a convolutional neural network via deep tensor decomposition," 2018, *arXiv:1805.06523*. [Online]. Available: http://arxiv.org/abs/1805.06523

[112] J. Ye *et al.*, "Learning compact recurrent neural networks with block-term tensor decomposition," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 9378–9387.

[113] L. De Lathauwer, B. De Moor, and J. Vandewalle, "A multilinear singular value decomposition," *SIAM J. Matrix Anal. Appl.*, vol. 21, no. 4, pp. 1253–1278, 2000.

[114] A. Rövid, L. Szeidl, and P. Várlaki, "On tensor-product model based representation of neural networks," in *Proc. 15th IEEE Int. Conf. Intell. Eng. Syst.*, Jun. 2011, pp. 69–72.

[115] L. Li and Q. Huang, "HRTF personalization modeling based on RBF neural network," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process.*, May 2013, pp. 3707–3710.

[116] A. Cichocki, "Tensor networks for dimensionality reduction, big data and deep learning," in *Advances in Data Analysis with Computational Intelligence Methods* (Studies in Computational Intelligence), vol. 738. Cham, Switzerland: Springer, 2018, pp. 3–49.

[117] L. Grasedyck, "Hierarchical singular value decomposition of tensors," *SIAM J. Matrix Anal. Appl.*, vol. 31, no. 4, pp. 2029–2054, Jan. 2010.

[118] Q. Zhao, M. Sugiyama, L. Yuan, and A. Cichocki, "Learning efficient tensor representations with ring-structured networks," in *Proc. ICASSP IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, May 2019, pp. 8608–8612.

[119] H. Huang, L. Ni, K. Wang, Y. Wang, and H. Yu, "A highly parallel and energy efficient three-dimensional multilayer CMOS-RRAM accelerator for tensorized neural network," *IEEE Trans. Nanotechnol.*, vol. 17, no. 4, pp. 645–656, Jul. 2018.

[120] J. Su, J. Li, B. Bhattacharjee, and F. Huang, "Tensorial neural networks: Generalization of neural networks and application to model compression," 2018, *arXiv:1805.10352*. [Online]. Available: http://arxiv.org/abs/1805.10352

[121] T. Garipov, D. Podoprikhin, A. Novikov, and D. Vetrov, "Ultimate tensorization: Compressing convolutional and FC layers alike," 2016, *arXiv:1611.03214*. [Online]. Available: http://arxiv.org/abs/1611.03214

[122] A. Tjandra, S. Sakti, and S. Nakamura, "Compressing recurrent neural network with tensor train," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, May 2017, pp. 4451–4458.

[123] A. Tjandra, S. Sakti, and S. Nakamura, "Tensor decomposition for compressing recurrent neural network," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Jul. 2018, pp. 1–8.

[124] Y. Yang, D. Krompass, and V. Tresp, "Tensor-train recurrent neural networks for video classification," in *Proc. 34th Int. Conf. Mach. Learn.*, vol. 70, 2017, pp. 3891–3900.

[125] J. West, "Generating trees and the catalan and Schröder numbers," *Discrete Math.*, vol. 146, nos. 1–3, pp. 247–262, Nov. 1995.

[126] M. Hou and B. Chaib-draa, "Hierarchical tucker tensor regression: Application to brain imaging data analysis," in *Proc. IEEE Int. Conf. Image Process. (ICIP)*, Sep. 2015, pp. 1344–1348.

[127] I. Perros, R. Chen, R. Vuduc, and J. Sun, "Sparse hierarchical tucker factorization and its application to healthcare," in *Proc. IEEE Int. Conf. Data Mining*, Nov. 2015, pp. 943–948.

[128] N. Cohen and A. Shashua, "Convolutional rectifier networks as generalized tensor decompositions," in *Proc. 33rd Int. Conf. Mach. Learn.*, vol. 48, 2016, pp. 955–963.

[129] N. Cohen, O. Sharir, and A. Shashua, "On the expressive power of deep learning: A tensor analysis," *J. Mach. Learn. Res., Workshop Conf. Proc.*, vol. 49, pp. 1–31, Jun. 2016.

[130] I. V. Oseledets, "Tensor-train decomposition," *SIAM J. Sci. Comput.*, vol. 33, no. 5, pp. 2295–2317, Jan. 2011.

[131] N. Lee and A. Cichocki, "Regularized computation of approximate pseudoinverse of large matrices using low-rank tensor train decompositions," *SIAM J. Matrix Anal. Appl.*, vol. 37, no. 2, pp. 598–623, Jan. 2016.

[132] L. Grasedyck and W. Hackbusch, "An introduction to hierarchical ($\mathcal{H}$-) rank and TT-rank of tensors with examples," *Comput. Methods Appl. Math.*, vol. 11, no. 3, pp. 291–304, 2011.

[133] V. Khrulkov, A. Novikov, and I. Oseledets, "Expressive power of recurrent neural networks," in *Proc. ICLR*, Apr. 2018.

[134] H. Rauhut, R. Schneider, and V. Z. Stojanac, "Tensor completion in hierarchical tensor representations," in *Compressed Sensing and Its Applications*. Cham, Switzerland: Springer, 2015, pp. 419–450.

[135] I. V. Oseledets, "Approximation of $(2^d \times 2^d)$ matrices using tensor decomposition," *SIAM J. Matrix Anal. Appl.*, vol. 31, no. 4, pp. 2130–2145, 2010.

[136] M. Espig, K. K. Naraparaju, and J. Schneider, "A note on tensor chain approximation," *Comput. Vis. Sci.*, vol. 15, no. 6, pp. 331–344, 2012.

[137] Q. Zhao, G. Zhou, S. Xie, L. Zhang, and A. Cichocki, "Tensor ring decomposition," 2016, *arXiv:1606.05535*. [Online]. Available: http://arxiv.org/abs/1606.05535

[138] R. Yu, S. Zheng, A. Anandkumar, and Y. Yue, "Long-term forecasting using higher order tensor RNNs," 2017, *arXiv:1711.00073*. [Online]. Available: http://arxiv.org/abs/1711.00073

[139] W. Wen, C. Xu, C. Wu, Y. Wang, Y. Chen, and H. Li, "Coordinating filters for faster deep neural networks," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Oct. 2017, pp. 658–666.

[140] L. Ge, H. Liang, J. Yuan, and D. Thalmann, "3D convolutional neural networks for efficient and robust hand pose estimation from single depth images," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 5679–5688.

[141] K. K. Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation*. Hoboken, NJ, USA: Wiley, 1999.

[142] X. Lin, C. Zhao, and W. Pan, "Towards accurate binary convolutional neural network," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 345–353.

[143] D. Zhang, J. Yang, D. Ye, and G. Hua, "LQ-Nets: Learned quantization for highly accurate and compact deep neural networks," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2018, pp. 365–382.

[144] C. Xu *et al.*, "Alternating multi-bit quantization for recurrent neural networks," 2018, *arXiv:1802.00150*. [Online]. Available: http://arxiv.org/abs/1802.00150

[145] A. Polino, R. Pascanu, and D. Alistarh, "Model compression via distillation and quantization," 2018, *arXiv:1802.05668*. [Online]. Available: http://arxiv.org/abs/1802.05668

[146] S. Jung *et al.*, "Learning to quantize deep networks by optimizing quantization intervals with task loss," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2019, pp. 4350–4359.

[147] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "DoReFa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," 2016, *arXiv:1606.06160*. [Online]. Available: http://arxiv.org/abs/1606.06160

[148] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional neural networks using logarithmic data representation," 2016, *arXiv:1603.01025*. [Online]. Available: http://arxiv.org/abs/1603.01025

[149] J. L. McKinstry *et al.*, "Discovering low-precision networks close to full-precision networks for efficient embedded inference," 2018, *arXiv:1809.04191*. [Online]. Available: http://arxiv.org/abs/1809.04191

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

[151] L. Deng, P. Jiao, J. Pei, Z. Wu, and G. Li, "GXNOR-Net: Training deep neural networks with ternary weights and activations without full-precision memory under a unified discretization framework," *Neural Netw.*, vol. 100, pp. 49–58, Apr. 2018.

[152] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *J. Mach. Learn. Res.*, vol. 18, no. 1, pp. 6869–6898, 2017.

[153] Q. He *et al.*, "Effective quantization methods for recurrent neural networks," 2016, *arXiv:1611.10176*. [Online]. Available: http://arxiv.org/abs/1611.10176

[154] P. Yin, J. Lyu, S. Zhang, S. Osher, Y. Qi, and J. Xin, "Understanding straight-through estimator in training activation quantized neural nets," 2019, *arXiv:1903.05662*. [Online]. Available: http://arxiv.org/abs/1903.05662

[155] Z.-G. Liu and M. Mattina, "Learning low-precision neural networks without straight-through estimator (STE)," 2019, *arXiv:1903.01061*. [Online]. Available: http://arxiv.org/abs/1903.01061

[156] Z. Lin, M. Courbariaux, R. Memisevic, and Y. Bengio, "Neural networks with few multiplications," 2015, *arXiv:1510.03009*. [Online]. Available: http://arxiv.org/abs/1510.03009

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

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

[159] S. K. Esser *et al.*, "Convolutional networks for fast, energy-efficient neuromorphic computing," *Proc. Nat. Acad. Sci. USA*, vol. 113, no. 41, pp. 11441–11446, Sep. 2016.

[160] Z. Cai, X. He, J. Sun, and N. Vasconcelos, "Deep learning with low precision by half-wave Gaussian quantization," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 5918–5926.

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

[162] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," 2016, *arXiv:1612.01064*.

[Online]. Available: http://arxiv.org/abs/1612.01064

[163] J. Achterhold, J. M. Koehler, A. Schmeink, and T. Genewein, "Variational network quantization," in *Proc. ICLR*, Apr. 2018.

[164] P. Yin, S. Zhang, J. Lyu, S. Osher, Y. Qi, and J. Xin, "BinaryRelax: A relaxation approach for training deep neural networks with quantized weights," *SIAM J. Imag. Sci.*, vol. 11, no. 4, pp. 2205–2223, Oct. 2018.

[165] C. Leng, Z. Dou, H. Li, S. Zhu, and R. Jin, "Extremely low bit neural network: Squeeze the last bit out with admm," in *Proc. AAAI*, Feb. 2018.

[166] D. Wan *et al.*, "Tbn: Convolutional neural network with ternary inputs and binary weights," *Matrix*, vol. 1, no. 2, pp. 0–6, 2018.

[167] W. Wen *et al.*, "Terngrad: Ternary gradients to reduce communication in distributed deep learning," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 1509–1519.

[168] H. Li, S. De, Z. Xu, C. Studer, H. Samet, and T. Goldstein, "Training quantized nets: A deeper understanding," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 5811–5821.

[169] S.-C. Zhou, Y.-Z. Wang, H. Wen, Q.-Y. He, and Y.-H. Zou, "Balanced quantization: An effective and efficient approach to quantized neural networks," *J. Comput. Sci. Technol.*, vol. 32, no. 4, pp. 667–682, Jul. 2017.

[170] A. Mishra, E. Nurvitadhi, J. J Cook, and D. Marr, "WRPN: Wide reduced-precision networks," 2017, *arXiv:1709.01134*. [Online]. Available: http://arxiv.org/abs/1709.01134

[171] Y. Guo, A. Yao, H. Zhao, and Y. Chen, "Network sketching: Exploiting binary structure in deep CNNs," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, vol. 2, Jul. 2017.

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

[173] Y. Choi, M. El-Khamy, and J. Lee, "Learning low precision deep neural networks through regularization," 2018, *arXiv:1809.00095*. [Online]. Available: http://arxiv.org/abs/1809.00095

[174] B. Zhuang, C. Shen, and I. Reid, "Training compact neural networks with binary weights and low precision activations," 2018, *arXiv:1808.02631*. [Online]. Available: http://arxiv.org/abs/1808.02631

[175] C. Baskin *et al.*, "UNIQ: Uniform noise injection for non-uniform quantization of neural networks," 2018, *arXiv:1804.10969*. [Online]. Available: http://arxiv.org/abs/1804.10969

[176] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "HAQ: hardware-aware automated quantization with mixed precision," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2019, pp. 8612–8620.

[177] P. Micikevicius *et al.*, "Mixed precision training," 2017, *arXiv:1710.03740*. [Online]. Available: http://arxiv.org/abs/1710.03740

[178] U. Köster *et al.*, "Flexpoint: An adaptive numerical format for efficient training of deep neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 1742–1752.

[179] D. Das *et al.*, "Mixed precision training of convolutional neural networks using integer operations," 2018, *arXiv:1802.00930*. [Online]. Available: http://arxiv.org/abs/1802.00930

[180] R. Banner, I. Hubara, E. Hoffer, and D. Soudry, "Scalable methods for 8-bit training of neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2018, pp. 5145–5153.

[181] G. Li *et al.*, "Training deep neural networks with discrete state transition," *Neurocomputing*, vol. 272, pp. 154–162, Jan. 2018.

[182] S. Wu, G. Li, F. Chen, and L. Shi, "Training and inference with integers in deep neural networks," 2018, *arXiv:1802.04680*. [Online]. Available: http://arxiv.org/abs/1802.04680

[183] C. Sakr and N. Shanbhag, "Per-tensor fixed-point quantization of the back-propagation algorithm," 2018, *arXiv:1812.11732*. [Online]. Available: http://arxiv.org/abs/1812.11732

[184] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan, "Training deep neural networks with 8-bit floating point numbers," in *Proc. Adv. Neural Inf. Process. Syst.*, 2018, pp. 7685–7694.

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

[186] J. Wu, Y. Wang, Z. Wu, Z. Wang, A. Veeraraghavan, and Y. Lin, "Deep k-means: Re-training and parameter sharing with harder cluster assignments for compressing deep convolutions," 2018, *arXiv:1806.09228*. [Online]. Available: http://arxiv.org/abs/1806.09228

[187] K. Hwang and W. Sung, "Fixed-point feedforward deep neural network design using weights $+1$, 0, and -1," in *Proc. IEEE Workshop Signal Process. Syst. (SiPS)*, Oct. 2014, pp. 1–6.

[188] J. Ott, Z. Lin, Y. Zhang, S.-C. Liu, and Y. Bengio, "Recurrent neural networks with limited numerical precision," 2016, *arXiv:1608.06902*. [Online]. Available: http://arxiv.org/abs/1608.06902

[189] J. Cheng, J. Wu, C. Leng, Y. Wang, and Q. Hu, "Quantized CNN: A unified approach to accelerate and compress convolutional networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 29, no. 10, pp. 4730–4743, Oct. 2018.

[190] S. Kapur, A. Mishra, and D. Marr, "Low precision RNNs: Quantizing RNNs without losing accuracy," 2017, *arXiv:1710.07706*. [Online]. Available: http://arxiv.org/abs/1710.07706

[191] P. Wang, X. Xie, L. Deng, G. Li, D. Wang, and Y. Xie, "Hitnet: Hybrid ternary recurrent neural network," in *Proc. Adv. Neural Inf. Process. Syst.*, 2018, pp. 601–611.

[192] Y. Bai, Y.-X. Wang, and E. Liberty, "ProxQuant: Quantized neural networks via proximal operators," 2018, *arXiv:1810.00861*. [Online]. Available: http://arxiv.org/abs/1810.00861

[193] X. Jia *et al.*, "Highly scalable deep learning training system with mixed-precision: Training ImageNet in four minutes," 2018, *arXiv:1807.11205*. [Online]. Available: http://arxiv.org/abs/1807.11205

[194] S. Wu *et al.*, "L_1-norm batch normalization for efficient training of deep neural networks," *IEEE Trans. neural Netw. Learn. Syst.*, vol. 30, no. 7, pp. 2043–2051, Jul. 2019.

[195] E. Hoffer, R. Banner, I. Golan, and D. Soudry, "Norm matters: Efficient and accurate normalization schemes in deep networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2018, pp. 2160–2170.

[196] S. Chetlur *et al.*, "CuDNN: Efficient primitives for deep learning," 2014, *arXiv:1410.0759*. [Online]. Available: http://arxiv.org/abs/1410.0759

[197] A. Ardakani, C. Condo, and W. J. Gross, "Sparsely-connected neural networks: Towards efficient VLSI implementation of deep neural networks," 2016, *arXiv:1611.01427*. [Online]. Available: http://arxiv.org/abs/1611.01427

[198] M. Zhu and S. Gupta, "To prune, or not to prune: Exploring the efficacy of pruning for model compression," 2017, *arXiv:1710.01878*. [Online]. Available: http://arxiv.org/abs/1710.01878

[199] Z. Liu, J. Xu, X. Peng, and R. Xiong, "Frequency-domain dynamic pruning for convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2018, pp. 1049–1059.

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

[201] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 2074–2082.

[202] X. Sun, X. Ren, S. Ma, and H. Wang, "meprop: Sparsified back propagation for accelerated deep learning with reduced overfitting," in *Proc. 34th Int. Conf. Mach. Learn. (JMLR)*, vol. 70, 2017, pp. 3299–3308.

[203] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, "Deep gradient compression: Reducing the communication bandwidth for distributed training," 2017, *arXiv:1712.01887*. [Online]. Available: http://arxiv.org/abs/1712.01887

[204] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient inference," 2016, *arXiv:1611.06440*. [Online]. Available: http://arxiv.org/abs/1611.06440

[205] J.-H. Luo, J. Wu, and W. Lin, "ThiNet: A filter level pruning method for deep neural network compression," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Oct. 2017, pp. 5058–5066.

[206] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Oct. 2017, vol. 2, no. 6, pp. 1389–1397.

[207] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, "Learning efficient convolutional networks through network slimming," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Oct. 2017, pp. 2755–2763.

[208] R. Yu *et al.*, "NISP: Pruning networks using neuron importance score propagation," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 9194–9203.

[209] J. Ye, X. Lu, Z. Lin, and J. Z. Wang, "Rethinking the smaller-norm-less-informative assumption in channel pruning of convolution layers," 2018, *arXiv:1802.00124*. [Online]. Available: http://arxiv.org/abs/1802.00124

[210] J.-H. Luo and J. Wu, "AutoPruner: An end-to-end trainable filter pruning method for efficient deep model inference," 2018, *arXiv:1805.08941*. [Online]. Available: http://arxiv.org/abs/1805.08941

[211] C. Min, A. Wang, Y. Chen, W. Xu, and X. Chen, "2PFPCE: Two-phase filter pruning based on conditional entropy," 2018, *arXiv:1809.02220*. [Online]. Available: http://arxiv.org/abs/1809.02220

[212] Y. He, G. Kang, X. Dong, Y. Fu, and Y. Yang, "Soft filter pruning for accelerating deep convolutional neural networks," 2018, *arXiv:1808.06866*. [Online]. Available: http://arxiv.org/abs/1808.06866

[213] T.-W. Chin, C. Zhang, and D. Marculescu, "Layer-compensated pruning for resource-constrained convolutional neural networks," 2018, *arXiv:1810.00518*. [Online]. Available: http://arxiv.org/abs/1810.00518

[214] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han, "AMC: AutoML for model compression and acceleration on mobile devices," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2018, pp. 784–800.

[215] E. Kim, C. Ahn, and S. Oh, "NestedNet: Learning nested sparse structures in deep neural networks," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 8669–8678.

[216] X. Xu, M. Sun Park, and C. Brick, "Hybrid pruning: Thinner sparse networks for fast inference on edge devices," 2018, *arXiv:1811.00482*. [Online]. Available: http://arxiv.org/abs/1811.00482

[217] X. Dai, H. Yin, and N. K. Jha, "NeST: A neural network synthesis tool based on a grow-and-prune paradigm," *IEEE Trans. Comput.*, vol. 68, no. 10, pp. 1487–1497, Oct. 2019.

[218] Y. Hu, S. Sun, J. Li, X. Wang, and Q. Gu, "A novel channel pruning method for deep neural network compression," 2018, *arXiv:1805.11394*. [Online]. Available: http://arxiv.org/abs/1805.11394

[219] L. Liang *et al.*, "Crossbar-aware neural network pruning," *IEEE Access*, vol. 6, pp. 58324–58337, 2018.

[220] Y. Ji, L. Liang, L. Deng, Y. Zhang, Y. Zhang, and Y. Xie, "Tetris: Tile-matching the tremendous

irregular sparsity," in *Proc. Adv. Neural Inf. Process. Syst.*, 2018, pp. 4115–4125.

[221] Y. Choi, M. El-Khamy, and J. Lee, "Compression of deep convolutional neural networks under joint sparsity constraints," 2018, *arXiv:1805.08303*. [Online]. Available: http://arxiv.org/abs/1805.08303

[222] C. Lin, Z. Zhong, W. Wei, and J. Yan, "Synaptic strength for convolutional neural network," in *Proc. Adv. Neural Inf. Process. Syst.*, 2018, pp. 10169–10178.

[223] T. Zhang *et al.*, "A systematic DNN weight pruning framework using alternating direction method of multipliers," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2018, pp. 184–199.

[224] T. Zhang *et al.*, "StructADMM: A systematic, high-efficiency framework of structured weight pruning for DNNs," 2018, *arXiv:1807.11091*. [Online]. Available: http://arxiv.org/abs/1807.11091

[225] S. Ye *et al.*, "Progressive weight pruning of deep neural networks using ADMM," 2018, *arXiv:1810.07378*. [Online]. Available: http://arxiv.org/abs/1810.07378

[226] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 1990, pp. 598–605.

[227] B. Hassibi and D. G. Stork, "Second order derivatives for network pruning: Optimal brain surgeon," in *Proc. Adv. Neural Inf. Process. Syst.*, 1993, pp. 164–171.

[228] S. Narang, E. Elsen, G. Diamos, and S. Sengupta, "Exploring sparsity in recurrent neural networks," 2017, *arXiv:1704.05119*. [Online]. Available: http://arxiv.org/abs/1704.05119

[229] S. Alford, R. Robinett, L. Milechin, and J. Kepner, "Training behavior of sparse neural network topologies," 2018, *arXiv:1810.00299*. [Online]. Available: http://arxiv.org/abs/1810.00299

[230] X. Dai, H. Yin, and N. K. Jha, "Grow and prune compact, fast, and accurate LSTMs," 2018, *arXiv:1805.11797*. [Online]. Available: http://arxiv.org/abs/1805.11797

[231] T.-J. Yang, Y.-H. Chen, and V. Sze, "Designing energy-efficient convolutional neural networks using energy-aware pruning," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 5687–5695.

[232] W. Wen *et al.*, "Learning intrinsic sparse structures within long short-term memory," 2017, *arXiv:1709.05027*. [Online]. Available: http://arxiv.org/abs/1709.05027

[233] M. Zhu, J. Clemons, J. Pool, M. Rhu, S. W. Keckler, and Y. Xie, "Structurally sparsified backward propagation for faster long short-term memory training," 2018, *arXiv:1806.00512*. [Online]. Available: http://arxiv.org/abs/1806.00512

[234] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing DNN pruning to the underlying hardware parallelism," *ACM SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 548–560, Jun. 2017.

[235] S. Anwar, K. Hwang, and W. Sung, "Structured pruning of deep convolutional neural networks," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 13, no. 3, pp. 1–18, Feb. 2017.

[236] B. Liu, M. Wang, H. Foroosh, M. Tappen, and M. Penksy, "Sparse convolutional neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2015, pp. 806–814.

[237] A. Makhzani and B. J. Frey, "Winner-take-all autoencoders," in *Proc. Adv. Neural Inf. Process. Syst.*, 2015, pp. 2791–2799.

[238] A. S. Morcos, D. G. T. Barrett, N. C. Rabinowitz, and M. Botvinick, "On the importance of single directions for generalization," 2018, *arXiv:1803.06959*. [Online]. Available: http://arxiv.org/abs/1803.06959

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

[240] J. Ba and B. Frey, "Adaptive dropout for training deep neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2013, pp. 3084–3092.

[241] R. Spring and A. Shrivastava, "Scalable and sustainable deep learning via randomized hashing," in *Proc. 23rd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining (KDD)*, 2017, pp. 445–454.

[242] V. Akhlaghi, A. Yazdanbakhsh, K. Samadi, R. K. Gupta, and H. Esmaeilzadeh, "SnaPEA: Predictive early activation for reducing computation in deep convolutional neural networks," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2018.

[243] Y. Lin, C. Sakr, Y. Kim, and N. Shanbhag, "PredictiveNet: An energy-efficient convolutional neural network via zero prediction," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2017, pp. 1–4.

[244] M. Song, J. Zhao, Y. Hu, J. Zhang, and T. Li, "Prediction based execution on deep neural networks," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2018, pp. 752–763.

[245] C. Kim, D. Shin, B. Kim, and J. Park, "Mosaic-CNN: A combined two-step zero prediction approach to trade off accuracy and computation energy in convolutional neural networks," *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 8, no. 4, pp. 770–781, Dec. 2018.

[246] L. Liu *et al.*, "Dynamic sparse graph for efficient deep learning," 2018, *arXiv:1810.00859*. [Online]. Available: http://arxiv.org/abs/1810.00859

[247] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell, "Rethinking the value of network pruning," 2018, *arXiv:1810.05270*. [Online]. Available: http://arxiv.org/abs/1810.05270

[248] P. Jiang and G. Agrawal, "A linear speedup analysis of distributed deep learning with sparse and quantized communication," in *Proc. Adv. Neural Inf. Process. Syst.*, 2018, pp. 2526–2537.

[249] Y. Guo, A. Yao, and Y. Chen, "Dynamic network surgery for efficient DNNs," in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 1379–1387.

[250] M. Rhu, M. O'Connor, N. Chatterjee, J. Pool, Y. Kwon, and S. W. Keckler, "Compressing DMA engine: Leveraging activation sparsity for training deep neural networks," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2018, pp. 78–91.

[251] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko, "Gist: Efficient data encoding for deep neural network training," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2018, pp. 776–789.

[252] S. Ambrogio *et al.*, "Equivalent-accuracy accelerated neural-network training using analogue memory," *Nature*, vol. 558, no. 7708, pp. 60–67, Jun. 2018.

[253] M. Cheng *et al.*, "Time: A training-in-memory architecture for RRAM-based deep neural networks," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 5, pp. 834–847, Apr. 2018.

[254] Y. Cai *et al.*, "Long live time: Improving lifetime for training-in-memory engines by structured gradient sparsification," in *Proc. 55th Annu. Design Autom. Conf.*, 2018, p. 107.

[255] S. Han *et al.*, "ESE: Efficient speech recognition engine with sparse LSTM on FPGA," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, 2017, pp. 75–84.

[256] D. Shin, J. Lee, J. Lee, and H.-J. Yoo, "DNPU: An 8.1 TOPS/W reconfigurable CNN-RNN processor for general-purpose deep neural networks," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2017, pp. 240–241.

[257] S. Yin *et al.*, "A high energy efficient reconfigurable hybrid neural network processor for deep learning applications," *IEEE J. Solid-State Circuits*, vol. 53, no. 4, pp. 968–982, 2018.

[258] A. Ankit *et al.*, "PUMA: A programmable ultra-efficient memristor-based accelerator for machine learning inference," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Oper. Syst.*, 2019, pp. 715–731.

[259] D. Hammerstrom, "A VLSI architecture for high-performance, low-cost, on-chip learning," in *Proc. IJCNN Int. Joint Conf. Neural Netw.*, 1990, pp. 537–544.

[260] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "NeuFlow: A runtime reconfigurable dataflow processor for vision," in *Proc. CVPR WORKSHOPS*, Jun. 2011, pp. 109–116.

[261] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.

[262] A. Parashar *et al.*, "Scnn: An accelerator for compressed-sparse convolutional neural networks," *ACM SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 27–40, 2017.

[263] H. Kung, B. McDanel, and S. Q. Zhang, "Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Oper. Syst.*, 2019, pp. 821–834.

[264] J. Pei *et al.*, "Towards artificial general intelligence with hybrid Tianjic chip architecture," *Nature*, vol. 572, no. 7767, pp. 106–111, Jul. 2019.

[265] L. Deng *et al.*, "Tianjic: A unified and scalable chip bridging spike-based and continuous neural computation," *IEEE J. Solid-State Circuits*, to be published.

[266] B. Zimmer *et al.*, "A 0.11 pJ/Op, 0.32-128 TOPS, scalable multi-chip-module-based deep neural network accelerator with ground-reference signaling in 16 nm," in *Proc. Symp. VLSI Circuits*, Jun. 2019, pp. C300–C301.

[267] Y. S. Shao *et al.*, "Simba: Scaling deep-learning inference with multi-chip-module-based architecture," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2019, pp. 14–27.

[268] P. Chi *et al.*, "Prime: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 27–39, 2016.

[269] S. Yin, X. Sun, S. Yu, and J.-s. Seo, "High-throughput in-memory computing for binary deep neural networks with monolithically integrated RRAM and 90nm CMOS," 2019, *arXiv:1909.07514*. [Online]. Available: http://arxiv.org/abs/1909.07514

[270] C. Eckert *et al.*, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," in *Proc. 45th Annu. Int. Symp. Comput. Archit.*, 2018, pp. 383–396.

[271] S. Li *et al.*, "SCOPE: A stochastic computing engine for DRAM-based *in-situ* accelerator," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, Oct. 2018, pp. 696–709.

[272] X. Guo *et al.*, "Fast, energy-efficient, robust, and reproducible mixed-signal neuromorphic classifier based on embedded NOR flash memory technology," in *IEDM Tech. Dig.*, Dec. 2017, pp. 5–6.

[273] K. Ando *et al.*, "BRein memory: A single-chip binary/ternary reconfigurable in-memory deep neural network accelerator achieving 1.4 TOPS at 0.6 w," *IEEE J. Solid-State Circuits*, vol. 53, no. 4, pp. 983–994, Apr. 2018.

[274] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "YodaNN: An architecture for ultralow power binary-weight CNN acceleration," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 1, pp. 48–60, Jan. 2018.

[275] A. Al Bahou, G. Karunaratne, R. Andri, L. Cavigelli, and L. Benini, "XNORBIN: A 95 TOp/s/W hardware accelerator for binary convolutional neural networks," in *Proc. IEEE Symp. Low-Power High-Speed Chips (COOL CHIPS)*, Apr. 2018, pp. 1–3.

[276] F. Conti, P. D. Schiavone, and L. Benini, "XNOR neural engine: A hardware accelerator IP for 21.6-fJ/op binary neural network inference," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2940–2951, Nov. 2018.

[277] A. Jafari, M. Hosseini, A. Kulkarni, C. Patel, and T. Mohsenin, "BiNMAC: Binarized neural network manycore accelerator," in *Proc. Great Lakes Symp. VLSI*, 2018, pp. 443–446.

[278] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, May 2016, pp. 1–12.

[279] H. Sharma *et al.*, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2018.

[280] J. Su *et al.*, "Accuracy to throughput trade-offs for reduced precision neural networks on reconfigurable logic," in *Appl. Reconfigurable Comput. Archit., Tools, Appl., 14th Int. Symp. ARC*, vol. 10824, Santorini, Greece. Cham, Switzerland: Springer, May 2018, p. 29.

[281] S. Zhang *et al.*, "Cambricon-X: An accelerator for sparse neural networks," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchit.*, Oct. 2016, pp. 1–12.

[282] C. Deng, S. Liao, Y. Xie, K. K. Parhi, X. Qian, and B. Yuan, "PermDNN: Efficient compressed DNN architecture with permuted diagonal matrices," in *Proc. 51th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2018, pp. 189–202.

[283] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 1–13, 2016.

[284] B. Reagen *et al.*, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 267–278, 2016.

[285] A. Aimar *et al.*, "NullHop: A flexible convolutional neural network accelerator based on sparse representations of feature maps," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 30, no. 3, pp. 644–656, Mar. 2019.

[286] S. Han *et al.*, "EIE: Efficient inference engine on compressed deep neural network," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 243–254.

[287] J. Lee, D. Shin, and H.-J. Yoo, "A 21 mW low-power recurrent neural network accelerator with quantization tables for embedded deep learning applications," in *Proc. IEEE Asian Solid-State Circuits Conf. (A-SSCC)*, Nov. 2017, pp. 237–240.

[288] K. Hegde, J. Yu, R. Agrawal, M. Yan, M. Pellauer, and C. W. Fletcher, "UCNN: Exploiting computational reuse in deep neural networks via weight repetition," in *Proc. 45th Annu. Int. Symp. Comput. Archit.*, 2018, pp. 674–687.

[289] E. Park, D. Kim, and S. Yoo, "Energy-efficient neural network accelerator based on outlier-aware low-precision computation," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2018, pp. 688–698.

[290] Y. Cai, T. Tang, L. Xia, B. Li, Y. Wang, and H. Yang, "Low bit-width convolutional neural network on rram," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, to be published.

[291] S. Jain, A. Sengupta, K. Roy, and A. Raghunathan, "RxNN: A framework for evaluating deep neural networks on resistive crossbars," 2018, *arXiv:1809.00072*. [Online]. Available: https://arxiv.org/abs/1809.00072

[292] T. Tang, L. Xia, B. Li, Y. Wang, and H. Yang, "Binary convolutional neural network on RRAM," in *Proc. 22nd Asia South Pacific Des. Autom. Conf. (ASP-DAC)*, 2017, pp. 782–787.

[293] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "DRISA: A DRAM-based reconfigurable *in-situ* accelerator," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchit.*, Oct. 2017, pp. 288–301.

[294] D. Bankman, L. Yang, B. Moons, M. Verhelst, and B. Murmann, "An always-on 3.8 μJ/86% CIFAR-10 mixed-signal binary CNN processor with all memory on chip in 28-nm CMOS," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2018, pp. 222–224.

[295] R. Liu, X. Peng, X. Sun, W. S. Khwa, and S. Yu, "Parallelizing SRAM arrays with customized bit-cell for binary neural networks," in *Proc. Design Autom. Conf.*, 2018, pp. 1–6.

[296] T.-H. Yang *et al.*, "Sparse ReRAM engine: Joint exploration of activation and weight sparsity in compressed neural networks," in *Proc. 46th Int. Symp. Comput. Archit.*, 2019, pp. 236–249.

[297] P. Wang, Y. Ji, C. Hong, Y. Lyu, D. Wang, and Y. Xie, "SNrram: An efficient sparse neural network computation architecture based on resistive random-access memory," in *Proc. 55th Annu. Design Autom. Conf.*, 2018, p. 106.

[298] J. Lin, Z. Zhu, Y. Wang, and Y. Xie, "Learning the sparsity for ReRAM: Mapping and pruning sparse neural network for ReRAM based accelerator," in *Proc. 24th Asia South Pacific Des. Autom. Conf.*, 2019, pp. 639–644.

[299] H. Ji, L. Song, L. Jiang, H. H. Li, and Y. Chen, "ReCom: An efficient resistive accelerator for compressed deep neural networks," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 237–240.

[300] A. Prost-Boucle, A. Bourge, F. Pétrot, H. Alemdar, N. Caldwell, and V. Leroy, "Scalable high-performance architecture for convolutional ternary neural networks on FPGA," in *Proc. 27th Int. Conf. Field Program. Logic Appl. (FPL)*, Sep. 2017, pp. 1–7.

[301] A. Kurakin, I. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," 2016, *arXiv:1607.02533*. [Online]. Available: https://arxiv.org/abs/1607.02533

[302] Z. Marzi, S. Gopalakrishnan, U. Madhow, and R. Pedarsani, "Sparsity-based defense against adversarial attacks on linear classifiers," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jun. 2018, pp. 31–35.

[303] S. Wang *et al.*, "Defensive dropout for hardening deep neural networks under adversarial attacks," in *Proc. Int. Conf. Comput.-Aided Design*, 2018, p. 71.

[304] G. Ji, W. Beilun, and Q. Yanjun, "DeepCloak: Masking deep neural network models for robustness against adversarial samples," 2017, *arXiv:1702.06763*. [Online]. Available: https://arxiv.org/abs/1702.06763

[305] A. Galloway, G. W. Taylor, and M. Moussa, "Attacking binarized neural networks," 2017, *arXiv:1711.00449*. [Online]. Available: http://arxiv.org/abs/1711.00449

[306] A. Novikov, P. Izmailov, V. Khrulkov, M. Figurnov, and I. Oseledets, "Tensor train decomposition on TensorFlow (T3F)," 2018, *arXiv:1801.01928*. [Online]. Available: https://arxiv.org/abs/1801.01928

[307] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," 2018, *arXiv:1806.08342*. [Online]. Available: http://arxiv.org/abs/1806.08342

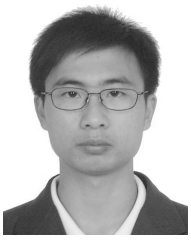
[308] M. Blott *et al.*, "*FINN-R*: an end-to-end deep-learning framework for fast exploration of quantized neural networks," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 11, no. 3, pp. 1–23, 2018.

ABOUT THE AUTHORS

Lei Deng (Member, IEEE) received the B.E. degree from the University of Science and Technology of China, Hefei, China, in 2012, and the Ph.D. degree from Tsinghua University, Beijing, China, in 2017.

He is currently a Postdoctoral Fellow with the Department of Electrical and Computer Engineering, University of California at Santa Barbara, Santa Barbara, CA, USA. He has authored or coauthored over 40 refereed publications. His research interests include brain-inspired computing, machine learning, neuromorphic chip, computer architecture, tensor analysis, and complex networks.

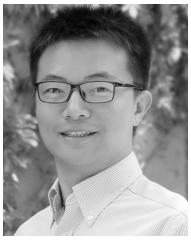
Dr. Deng was a recipient of the MIT Technology Review Innovators Under 35 China 2019. He was a PC Member for International Symposium on Neural Networks (ISNN) 2019. He currently serves as a Guest Associate Editor for *Frontiers in Neuroscience* and *Frontiers in Computational Neuroscience*, and a reviewer for a number of journals and conferences.



Guoqi Li (Member, IEEE) received the B.E. degree from the Xi'an University of Technology, Xi'an, China, in 2004, the M.E. degree from Xi'an Jiaotong University, Xi'an, in 2007, and the Ph.D. degree from Nanyang Technological University, Singapore, in 2011.

He was a Scientist with the Data Storage Institute and the Institute of High Performance Computing, Agency for Science, Technology and Research (ASTAR), Singapore, from 2011 to 2014. He is currently an Associate Professor with the Center for Brain Inspired Computing Research (CBICR), Tsinghua University, Beijing, China. His current research interests include machine learning, brain-inspired computing, neuromorphic chip, complex systems, and system identification.

Dr. Li was a recipient of the 2018 First Class Prize in Science and Technology of the Chinese Institute of Command and Control, Best Paper Awards (EAIS 2012 and NVMTS 2015), and the 2018 Excellent Young Talent Award of the Beijing Natural Science Foundation. He is also an Editorial Board Member for *Control and Decision* and a Guest Associate Editor for *Frontiers in Neuroscience*, *Neuromorphic Engineering*.



Song Han received the B.S. degree in microelectronics from Tsinghua University, Beijing, China, in 2012, and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, USA, in 2014 and 2017, respectively.

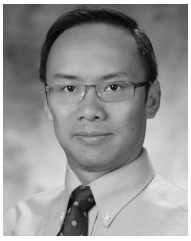
He was a Co-Founder and the Chief Scientist of DeePhi Tech., Beijing, which has been acquired by Xilinx in 2018 for the technologies in deep compression and system-level optimization for neural networks. He is currently an Assistant Professor with the Electrical Engineering and Computer Science Department, Massachusetts Institute of Technology (MIT), Cambridge, MA, USA. His research interests include co-designing efficient algorithms and hardware systems for machine learning.

Dr. Han has served as a PC Member for HPCA 2018 and SysML 2019. He received the Best Paper Awards, ICLR 2016, FPGA 2017, the 2018 SONY Faculty Award, and the 2018 Amazon Machine Learning Research Award. He has served as the Area Chair for ICLR 2019 and a reviewer for top conferences and journals.

Luping Shi received the B.S. and M.S. degrees in physics from Shandong University, Jinan, China, in 1981 and 1988, respectively, and the Ph.D. degree in physics from the University of Cologne, Cologne, Germany, in 1992.

He was a Postdoctoral Fellow with the Fraunhofer Institute for Applied Optics and Precision Instrument, Jena, Germany, in 1993, and a Research Fellow with the Department of Electronic Engineering, City University of Hong Kong, Hong Kong, from 1994 to 1996. From 1996 to 2013, he was with the Data Storage Institute (DSI), Singapore, as a Senior Scientist and Division Manager, and led research studies on nonvolatile solid-state memory (NVM), artificial cognitive memory (ACM), and optical storage. He joined Tsinghua University, Beijing, China, as a National Distinguished Professor, in 2013. By integrating seven departments, he established the Center for Brain Inspired Computing Research (CBICR), Tsinghua University, in 2014, where he served as the Director. He has published more than 200 articles in prestigious journals including *Nature*, *Science*, *Nature Photonics*, *Advanced Materials*, and *Physical Review Letters*. His research interests include brain-inspired computing, neuromorphic chip, memory devices, and so on.

Dr. Shi is a Fellow of SPIE. He was a recipient of the 2004 National Technology Award of Singapore, the unique awardee that year. He has served as the General Co-Chair for the Asia-Pacific Conference on Near-Field Optics in 2013, NVMTS 2011-2015, East-West Summit on Nanophotonics and MetalMaterials 2009 and ODS 2009. He is an Editorial Board Member of *Scientific Reports* and an Associate Editor for *Frontiers in Neuroscience*, *Neuromorphic Engineering*.



Yuan Xie (Fellow, IEEE) received the B.S. degree in electronic engineering from Tsinghua University, Beijing, China, in 1997, and the M.S. and Ph.D. degrees in electrical engineering from Princeton University, Princeton, NJ, USA, in 1999 and 2002, respectively.

He was an Advisory Engineer with the IBM Microelectronic Division, Burlington, NJ, USA, from 2002 to 2003. He was a Full Professor with Pennsylvania State University, State College, PA, USA, from 2003 to 2014. He was a Visiting Researcher with the Interuniversity Microelectronics Center (IMEC), Louvain, Belgium, from 2005 to 2007 and in 2010. He was a Senior Manager and a Principal Researcher with the AMD Research China Laboratory, Beijing, from 2012 to 2013. He is currently a Professor with the Department of Electrical and Computer Engineering, University of California at Santa Barbara, Santa Barbara, CA, USA. His research interests include VLSI design, electronics design automation (EDA), computer architecture, and embedded systems.

Dr. Xie serves as a Committee Member for the IEEE Design Automation Technical Committee (DATC). He is a Fellow of AAAS/ACM and an Expert in computer architecture who has been inducted into ISCA/MICRO/HPCA Hall of Fame. He was a recipient of the Best Paper Awards at HPCA 2015, ICCAD 2014, GLSVLSI 2014, ISVLSI 2012, ISLPED 2011, ASPDAC 2008, and ASICON 2001; the Best Paper Nominations at ASPDAC 2014, MICRO 2013, DATE 2013, ASPDAC 2010–2009, and ICCAD 2006; the 2016 IEEE Micro Top Picks Award at the 2008 IBM Faculty Award, and the 2006 NSF CAREER Award. He has served as the TPC Chair for ICCAD 2019, HPCA 2018, ASPDAC 2013, ISLPED 2013, and MPSOC 2011. He serves as the Editor-in-Chief for *ACM Journal on Emerging Technologies in Computing Systems* and as an Associate Editor for the *ACM Transactions on Design Automations for Electronics Systems*, the IEEE TRANSACTIONS ON COMPUTERS, the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, *IEEE Design and Test*, and *IET Computers and Design Techniques*. Through extensive collaboration with industry partners, AMD, HP, Honda, IBM, Intel, Google, Samsung, IMEC, Qualcomm, Alibaba, Seagate, Toyota, and so on, he has helped the transition of research ideas to industry.