2019

Deep Model Compression for Mobile Platforms: A Survey

Kaiming Nan  
*School of Computer Science and Technology, Xidian University, Xi’an 710071, China.*

Sicong Liu  
*School of Computer Science and Technology, Xidian University, Xi’an 710071, China.*

Junzhao Du  
*School of Software and Institute of Software Engineering, Xidian University, Xi’an 710071, China.*

Hui Liu  
*School of Software and Institute of Software Engineering, Xidian University, Xi’an 710071, China.*

Follow this and additional works at: https://tsinghuauniversitypress.researchcommons.org/tsinghua-science-and-technology

Part of the Computer Sciences Commons, and the Electrical and Computer Engineering Commons

**Recommended Citation**

Kaiming Nan, Sicong Liu, Junzhao Du et al. Deep Model Compression for Mobile Platforms: A Survey. Tsinghua Science and Technology 2019, 24(06): 677-693.

This Research Article is brought to you for free and open access by Tsinghua University Press: Journals Publishing. It has been accepted for inclusion in Tsinghua Science and Technology by an authorized editor of Tsinghua University Press: Journals Publishing.
Deep Model Compression for Mobile Platforms: A Survey

Kaiming Nan, Sicong Liu, Junzhao Du, and Hui Liu*

Abstract: Despite the rapid development of mobile and embedded hardware, directly executing computation-expensive and storage-intensive deep learning algorithms on these devices’ local side remains constrained for sensory data analysis. In this paper, we first summarize the layer compression techniques for the state-of-the-art deep learning model from three categories: weight factorization and pruning, convolution decomposition, and special layer architecture designing. For each category of layer compression techniques, we quantify their storage and computation tunable by layer compression techniques and discuss their practical challenges and possible improvements. Then, we implement Android projects using TensorFlow Mobile to test these 10 compression methods and compare their practical performances in terms of accuracy, parameter size, intermediate feature size, computation, processing latency, and energy consumption. To further discuss their advantages and bottlenecks, we test their performance over four standard recognition tasks on six resource-constrained Android smartphones. Finally, we survey two types of run-time Neural Network (NN) compression techniques which are orthogonal with the layer compression techniques, run-time resource management and cost optimization with special NN architecture, which are orthogonal with the layer compression techniques.

Key words: deep learning; model compression; run-time resource management; cost optimization

1 Introduction

Recently, the development of hardware, such as Graphic Processing Unit (GPU)[1] and Tensor Processing Unit (TPU)[2], and the success of deep learning algorithms, such as AlexNet[3], the 16-layer VGG[4], and the 152-layer ResNet[5], hastened the popularization of deep learning based applications in a wide range of areas[4–7]. The deep learning based applications, such as computer vision[8], speech recognition[9], user-input recognition[10], natural language processing[11], and the recommend system[12], have been well understood. Specifically, the Neural Networks (NNs) build a non-linear map using multiple hidden layers to extract increasingly better features, resulting in considerable technological improvements for the artificial intelligence.

However, these powerful deep models along with the added resource cost in latency, storage, computation, and energy, cause difficulty in realizing off-line deep awareness in the resource-constrained mobile and embedded devices. For example, the well-performed VGG-16 model[4] quantified with 8 bits trained by ImageNet datasets[13] requires $1.5 \times 10^{10}$ of Multiply-Accumulate (MAC) operations, $1.4 \times 10^8$ of parameters, cost average of 1650 ms latency, and 397.7 mJ energy on RedMi 3S Android platforms. Thus, compressing/simplifying the NN’s parameter/computation substantially aids in fitting the deep learning algorithms into the mobile platforms.
for deep awareness applications. Several typical deep learning based mobile device applications, such as Siri, facial, fingerprint unlock\textsuperscript{[14]}, and speech-to-text tools, also show considerable research potential in deep learning based mobile applications.

The NN layer compression is a commonly employed technique to trim down the NN’s complexity to fit the computation-intensive deep models for mobile devices; this method can be performed by reducing the weight precision or the number of operations or both. We first summarize three categories of layer compression techniques: weight compression, convolution decomposition, and compact architectures (see details in Section 2). Deep NNs (DNNs) are compressed either during or after training to be executed on resource-constrained mobile devices. However, the existing layer compression techniques only investigate the one-for-all scheme, e.g., how to reduce the NN complexity for unique objectives using one compression technique, and neglect the various resource constraints across different deployment platforms. For each technique, we quantify two performance metrics, parameter size and computation amount, which are tunable by the applied compression techniques (Table 1) and impact the latency and energy cost. Meanwhile, we discuss the drawbacks and possible improvement directions for the layer compression techniques.

According to the orthogonal to layer compression techniques described above, NNs can also be optimized at run-time to preserve their execution accuracy and reduce the unnecessary resource cost. We briefly summarize two classes of run-time NN compression techniques: cost optimization for multiple NNs and cost optimization with special NN architecture. Then, we discuss the challenges and improvement directions in Section 4.

### 2 Layer Compression Techniques

An NN consists of multiple and different types of layers. A typical DNN includes an input layer, multiple Fully-Connected (FC) layers, and an output layer,
whereas a typical Convolutional NN (CNN) contains an input layer, multiple Convolution (CONV) layers and pooling layers, several FC layers, and finally an output layer. Computation and memory of NNs are dominated by FC and CONV layers. This section reviews the state-of-the-art compression techniques for FC and CONV layers. We summarize these techniques into three categories: weight compression, convolution decomposition, and special layer architecture.

Table 1 compares and quantifies the weight size and the computation operation amount of one FC or CONV layer using different types of layer compression techniques. As the parameter size and the MAC amount of the compressed layer are tunable by different compression methods, they serve as performance metrics that influence the other metrics.

Let the input and output sizes of an FC layer be $A$ and $B$, respectively. Then, the weight size and computation operation amount of the FC layer are all $AB$. Similarly, denote the input and output sizes of a CONV layer as $M_{in}W_{in}$ and $N_{out}W_{out}$, respectively. In this case, $M$ and $N$ are the channel number of input and output, $H_{in}$ and $W_{in}$ refer to the shape of input features, and $H_{out}$ and $W_{out}$ show the shape of output features. The weight size and computation amount in terms of MAC amounts of the CONV layer are $M_{in}N_{out}W_{in}W_{out}$, respectively, where $U$ is kernel height and $V$ is kernel width. Next, we discuss each compression technique in detail.

2.1 Weight compression

Weight compression removes redundant weights to reduce the size of NNs. This technique can be applied to compress both FC and CONV layers in the NNs. Three popular weight compression techniques are normally used.

2.1.1 Tech1: Singular value decomposition based weight compression

As shown in Fig. 1, the Singular Value Decomposition (SVD) based weight factorization method introduces an intermediate layer $L'$ between two adjacent layers, $L$ and $L + 1$, to compress their trained weights $W$ by SVD$^{[15, 23]}$. This method requires no retraining, and thus, it can be adopted for run-time layer compression. The weight matrix $W_{A \times B}^{L+1}$ of two adjacent layers is decomposed by SVD into two components, and it is further approximated by keeping $k$ closest to the number of eigenvalues of the matrix. Thus, the weight matrix can be computed by the dot-product of two weight matrices $W_{L'}^L W_{L+1}^{L+1}$. The dot-product of matrix is commutable, and the map relationships between the three layers are as follows:

$$y = (W_{L'}^L x) + b = (W_{L}^L W_{L+1}^{L+1} x) + b = (W_{L}^{L+1})^T (W_{L'}^L x) + b$$

That is to say, the layers $L$ and $L'$ are connected by $y = (W_{L'}^L W_{L+1}^{L+1} x)$ without bias, and the mapping relationship between layer $L'$ and $(L + 1)$ is $y = (W_{L}^{L+1})^T y' + b$, with the same bias $b$ as the original connection. After compression, both the weight size and computation are reduced to $(A + B)k$.

2.1.2 Tech2: Weight compression based on sparse coding

Weight compression based on sparse coding decomposes the original weight matrix of an FC/CONV layer into two matrices using sparse dictionary learning, where the dictionary with $k$ basis is learned from the original weight $W^{[23]}$. With the learned dictionary $B$ and the sparse code $C$, we obtain the sparse factorization of the weight $W^{L+1}$:

$$W^{L+1} \approx \sum_{i=1}^{k} B_i C_i$$

The mapping relationship is similar with the above SVD-based compression method. With a $k$-basis dictionary, the weight size and computation of the original layer are both reduced to $(A + B)k$.

2.1.3 Tech3: Weight pruning

This method introduces a three-phase compression pipeline (pruning, trained quantization, and Huffman coding) to reduce the storage and energy consumption without affecting the accuracy$^{[17, 24]}$. Specifically, weight pruning first prunes the trained network by removing the unimportant (small-weight) connections, i.e., weights below a certain threshold, and then retrain the network to tune weights for the remaining sparse connections. Next, the method uses K-means clustering to identify the weight sharing to reduce the number
of weights for storage. Then, Huffman coding is adopted to reduce the number of bits that represent each connection. However, the clustering and Huffman coding processes of weights require an underlying modification to the storage Application Programming Interface (API) of TensorFlow when loaded on an Android platform. Tech3 is applicable for both FC and CONV layers. The compressed weight size and computation for FC layers are $AB - n_{[\text{weight}]<\epsilon}$ and $AB - n_{[\text{weight}]<\epsilon}$, respectively. Here $n_{[\text{weight}]<\epsilon}$ denotes the number of unimportant weights below the threshold $\epsilon$. For the CONV layers, the compressed weight size and computation are $MNUV - n_{[\text{weight}]<\epsilon}$ and $[MNUV - n_{[\text{weight}]<\epsilon}]H_{\text{out}}W_{\text{out}}$, respectively.

2.1.4 Discussion and improvements

Discussion. The developers can select the proper techniques for specific awareness applications. The Tech3 usually achieves a higher compression ratio than Tech1 and Tech2, but it requires resource-intensive retraining. The retraining is unsuitable for the run-time compression requirements.

Improvements on Tech1 and Tech2: Runtime hyper-parameter setup for low-rank weight factorization. Both Tech1 and Tech2 factorize the original weights by adding a new layer through different factorization methods, and their compression performance is limited by an estimated hyper-parameter $k$ for factorization. The intuitive method sets up the parameter $k$ according to the testing results of massive experiment comparisons. However, the choice of hyper-parameters depends more on human experience, which may cause the poor performance of the model.

DeepMind presents a run-time parameter estimation mechanism that can automatically select the hyper-parameters for different NN architectures rather than conduct a random or grid search but disregards the resource limitation of diverse devices.

Improvement to Tech3: Inactive weight pruning. The generally used weight pruning method sets the negligible weights, whose values are below a user-defined threshold, to be zero. We can further study an efficient mechanism to determine the active weights instead of blindly removing the small-value weights. One possible solution includes the usage of the Hashing method to determine the approximate active sets of neurons (weights) which can be the top $p\%$ nodes with significant activation. Spring and Shrivastava[26] presented a novel hashing-based technique to combine recent ideas from adaptive dropouts and randomized hashing for maximum inner product search to select the nodes with the highest activation efficiently. Another possible solution is to understand the high-level features by inverting them, as the inactive weights constantly feature less contributions to the feature abstraction and the following classification. Specifically, we can study the locality of the active node information stored in the representations by reconstructing the input from selected groups of neurons, either spatially or by channel through the AutoEncoder. Mahendran and Vedaldi[27] conducted a direct analysis of the visual information contained in representations by inverting representations, and they observed that a portion of the layers in CNNs retains photographically accurate image information, with different degrees of geometric and photo-metric invariance. Raghu et al.[28] proposed the Singular Vector Canonical Correlation Analysis (SVCCA) for quick comparison of two representations to measure the intrinsic dimensionality of the layers, showing in certain cases needless over-parameterization and formation of class-specific information in networks. Thus, we can follow this thread of findings to study the locality of the active node information stored in the representations through reconstructing the input from the selected groups of neurons or SVCCA.

Improvement to Tech3: Universal sparse storage and computation techniques. (1) EIE[29] is a dedicated accelerator to implement the sparse NN and the weight sharing NN. However, this technique is difficult to deploy on ubiquitous low-end Android platforms owing to the special usage of process units. The simple storage mechanism without hardware change, such as directly skipping the zero-weight operation at FC layers, is necessary to be compatible with TensorFlow Lite[30].

(2) The FC layers dominate the weight size given their lack of the weight-sharing mechanism. Tech3 uses K-means clustering to identify the shared weights for each layer of a trained network, and all the weights that fall into the same cluster will share the same weight. On one hand, the number of clusters and the algorithm of clustering are necessary for each layer in NN and may lead to various performance. Thus, selecting hyper-parameters and retraining could require considerable time. On the other hand, the method of storage with Huffman coding in bits necessitates restoring in Android platforms for inference, thus wasting time for the awareness of mobile devices. In the future work, we can study the novel storage and
computation methods for FC layer weight sharing with less underlying modification.

### 2.2 Convolution decomposition

Convolution operations involve calculations among multiple channels, incurring intensive computation and memory. Convolution decomposition techniques compress CONV layers leveraging different types of sparsity in the convolution operation. We review four representative convolution decomposition methods for CONV layer compression as follows.

#### 2.2.1 Tech4: Sparse decomposition of convolution kernel

Sparse decomposition maximizes the sparsity of the kernels in convolution operations\(^{18}\). Specifically, a computation-intensive convolution is replaced by two-stage decomposition by using the sparse dictionary learning algorithms, e.g., principal component analysis or multiplication with identity matrices. Next, retraining the weights reduces the accuracy loss caused by the above decomposition. Notably, the loss function in this retraining phase is modified to involve the loss of several components. The compressed weight size and computation are \(\frac{1}{2(MN_{UV})}\) and \(\frac{1}{H_{out}W_{out}}\), respectively, where \(\theta\) is the sparsity coefficient (Table 1).

#### 2.2.2 Tech5: Direct sparse convolution

To reduce the MAC computation amount of traditional convolution, the direct sparse convolution method, under which only the terms corresponding to the non-zero weights need to be computed, is presented\(^{20,25}\). When the \(K_{(n,c,h,w)}\) is a zero value, then the result of \(K_{(n,c,h,w)} \cdot I_{(c, xs+h, ys+w)}\) is also a zero value, which will never affect the result of convolution operation and can be ignored. The variable \(s\) represents the stride, \((h, w)\) is the position of conv kernel value. When calculating the output value at \((x, y)\) position of the \(n\)-th output channel, the direct sparse convolution algorithm needs only to determine and compute the non-zero values from the \(n\)-th filter. The convolution kernel is a four-mode tensor \(K\), the input feature map is a three-mode tensor \(I\), and the output feature is a three-mode tensor \(O\). Then, the output value at \((x, y)\) of \(n\)-th output channel is computed by the following:

\[
O_{(n,x,y)} = \sum_{c=0}^{C-1} \sum_{h=0}^{H-1} \sum_{w=0}^{W-1} K_{(n,c,h,w)} \cdot I_{(c, xs+h, ys+w)}
\]

(3)

As shown in Fig. 2, the convolution kernel \(K\) is sparse. Numbers 3 and 4 are the valid weights in kernel contributing to the final output. Thus, the convolution operation can be treated as the sum of two matrices, each of which is the product of a number and a matrix generated from the input feature map.

#### 2.2.3 Tech6: Depth-wise separable convolution

Depth-wise convolution decomposes the standard convolution into a depth-wise convolution and a point-wise convolution\(^{19}\). Specifically, this method presents two multipliers to drop certain weights: width multiplier \(\gamma\) and resolution multiplier \(\rho\). The width multiplier \(\gamma\) is used to thin a network uniformly (drop the input channels and output channels) at each layer, and the resolution parameter \(\rho\) is applied to the input and internal features to reduce the resolution of the input features of each layer. To simplify implementation, we can only implement the width multiplier by dropping out several channels of the convolution kernels. The performance of Tech6 is controlled by the width multiplier \(\gamma\). Given a width multiplier \(\gamma\), the compressed weight size and computation are \(\gamma M_{UV} + \gamma^2 M_{MN}\) and \(\gamma U V_{MH_{in} W_{in}} + \gamma^2 M_{NH_{in} W_{in}}\), respectively.

#### 2.2.4 Tech7: Sparse random

For this technique, the key idea is to replace the dense connections between a small number of channels with sparse connections and between a large number of

![Fig. 2 Example of direct sparse convolution computation (Tech5).](image-url)
channels for convolution operations\textsuperscript{[31]}. Based on the basic sparsification principle of the Tech6, random connection dropout schemes at the CONV layers are selected across spatial dimensions. That is, each output channel connects to a (possibly) different subset of channels. Given a sparsity coefficient $\theta$, the compressed weight size and computation are $(1 - \theta)MNU$ and $(1 - \theta)H_{out}W_{out}UVMN$, respectively.

### 2.2.5 Discussion and improvements

**Discussion.** Tech4 and Tech5 can aggressively compress the CONV layers in CNNs, whereas Tech6 and Tech7 are useful for large-scale NNs. In our implementation, given its simplicity, we use Tech6 to compute the sparse matrix generated by Tech4; on the other hand, the other sparse matrix multiplication techniques are not universal enough to involve the machine instruction Advanced Vector Extension\textsuperscript{[18]}.

**Improvement to convolution decomposition:**

**Useful sparsity bound determination.** When we use Tech6 and Tech7 to remove the random or width-wise part of channels at the CONV layers, the useful sparsity range for CONV weights should be further studied. For example, a sparsity higher than the upper bound or lower than the lower bound will provide no contribution to the cost (latency, storage, or energy consumption) reduction nor cause considerable accuracy loss. These bounds on useful sparsity can provide run-time guidelines for simultaneously tuning the DNN accuracy, speed, and model size of sparse CNN.

### 2.3 Layer architecture modification

Instead of layer compression, researchers have also proposed more compact layer architectures.

#### 2.3.1 Tech8: Fire layer

Fire layers are proposed to replace the CONV layers\textsuperscript{[21]}. Such technique employs three specific strategies: (1) replacing $3 \times 3$ filters with $1 \times 1$ filters; (2) decreasing the number of input channels to $3 \times 3$ filters; (3) simple downsampling late in the network to obtain large activation maps for the convolution layers. Thus, a fire layer includes a squeeze layer with one filter $M_1N_1 \times 1 \times 1$ and an expanded layer with two filters $M_2N_2 \times 1 \times 1$ and $M_3N_3 \times 3 \times 3$, respectively. These filters are denoted by $s_1 \times 1$, $e_1 \times 1$, and $e_3 \times 3$, whose output sizes are $N_1H_{out1}W_{out1}$, $N_2H_{out2}W_{out2}$, and $N_3H_{out3}W_{out3}$, respectively. Thus, the compressed weight size and computation are $M_1N_1 + M_2N_2 + (3 \times 3 \times M_3N_3)$, and $H_{out1}W_{out1}M_1N_1 + H_{out2}W_{out2}M_2N_2 + 3 \times 3 \times H_{out3}W_{out3}M_3N_3$, respectively.

#### 2.3.2 Tech9: MlpConv layer

The MlpConv layer is proposed to increase the depth of CNNs with several parameters leveraging a micro multi-layer perceptron embedded with multiple small kernel conv layers\textsuperscript{[22]}. Specifically, the MlpConv layer is embedded with multiple CONV layers with $1 \times 1$ filters (Fig. 3). For the specific tasks of mobile computing and the resource constraints of devices, the number of internal micro network can be flexibly adjusted. Assume two layers ($i = 1, 2$) with the filter size $M_iN_iU_iV_i$ and output size $H_{out}W_{out}N_iM_i$ in the micro network; their compressed weight size and computation are $M_1N_1 + M_2N_2$ and $\sum_{i=1}^{2}H_{out}W_{out}U_iV_iM_iN_i$, respectively.

#### 2.3.3 Tech10: Global average pooling layer

The global average is introduced to replace the FC layers in the CNN to avoid over-fitting\textsuperscript{[22]}, which is necessary for mobile computing scenes to avoid the weight learning strict rules based on limited training datasets as the test data are sensed by the different mobile devices with diverse sampling rate, hardware precision, and environmental noise. Specifically, instead of adding the FC layers over the feature map, this method extracts the average of each feature map and passes the result vector to the output (softmax) layer. Thus, Tech10 can replace the FC layer over the feature map to generate classification results. The compressed weight size and computation are $MN$ and $H_{out}W_{out}MN$, respectively.

#### 2.3.4 Discussion and improvements

**Discussion.** Different compression techniques are suitable for different cases according to the diverse network architecture and performance demands. Specifically, the SVD-based weight compression

![Fig. 3 An example of Tech9.](image-url)
(Tech1), the sparse-coding compression (Tech2), and the global average pooling (Tech10) can dramatically reduce the parameter size especially for large-scale networks given the high proportion of FC layer weights in the whole model. Weight pruning (Tech3), sparse and direct sparse compression (Tech4 and Tech5, respectively), and sparse random compression (Tech7) depend more on the hardware computation chip because the sparse matrix calculation needs a subtle operation flow control to achieve better performance. Depth-wise separable (Tech6), fire module (Tech8), and MlpConv module (Tech9) redefine the network structure to decouple the traditional convolution method; these strategies also achieve high accuracy and low energy consumption. However, they need to be trained thoroughly, making the process time-consuming and more difficult to execute off-line. Also, the fire module (Tech8) and MlpConv module (Tech9) are more suitable to improve the generalization of large CNNs. The global average pooling (Tech10) can avoid over-fitting of the FC layers and is fit for smaller NNs.

**Improvement with composite CONV layers.** An important advantage of NN for mobile applications is the separability and generalization capability of feature extraction. The generalization capability of the deep feature is proportional to the non-linearity of the feature extractor. Both Tech8 and Tech9 build a baseline to increase the number of layers with fewer weights to improve the non-linearity of feature mapping. More special architectures with several parallel and sequential $1 \times 1$, $3 \times 3$, and $5 \times 5$ filter kernels can be studied to achieve better performance. For example, we can combine the strategies in Tech8 and Tech9 to design a special layer: multiple sequential CONV layers with $1 \times 1$ filters that are finally merged with $3 \times 3$ CONV.

**Improvement with combination of the above techniques.** All these NN layer compression techniques are presented for a part of resource cost optimization objectives (e.g., energy, latency, computation, or parameter size). Thus, they experience bottlenecks. To optimize all of the resource costs, we can further study the underlying relationships between different techniques when they are applied for different layers and combine them to simultaneously compress both the CONV and FC layers. Thus, the combinations of these special layers can further improve the compression performance. For example, the SqueezeNet$^{[16]}$ uses Tech2 and convolution separation to simultaneously compress the CONV and FC layers. For example, we can combine two or three compression techniques to achieve better trade-off between accuracy, storage, latency, computation, and energy consumption than all of the single compression techniques. Figures 4 and 5 show the network architecture.

The existing compression techniques investigate a one-for-all scheme, e.g., how to reduce NN complexity using one compression technique, and neglect the various resource constraints across different deployment platforms. In the future work, we can further study the automatic selection of the best combination of different compression techniques that balance the application-driven system performance and the platform-imposed resource constraints for different devices and recognition tasks.

### 3 Evaluation

In this section, we conduct exhaustive experiments to evaluate the performance of state-of-the-art NN compression methods and show our findings.

#### 3.1 Experiment setup

We will describe the experimental setup from three aspects: software implementation, recognition tasks, datasets, and models, and mobile platforms.

**Software implementation.** We train the NNs using TensorFlow library on the server sides, such as cloud server or super-microcomputers, because of their powerful GPU computing and storage$^{[32]}$. Then, the trained DNNs are loaded on Android platforms for off-line recognition with the TensorFlow Library$^{[30]}$.

To execute NNs on Android platforms, we can use either TensorFlow Mobile or TensorFlow Lite$^{[30]}$. The TensorFlow Lite is published on November 15, 2017;

![Fig. 4 Novel combination of compression techniques in LeNet.](image)

![Fig. 5 Novel combination of compression techniques in AlexNet.](image)
it is the lightweight TensorFlow solution for mobile and embedded devices to achieve low latency and a small storage size. Its release shows that the design of lightweight NNs for mobile and embedded devices is a hot topic in both the research and industrial communities. The TensorFlow Lite will transfer the “pb” file into “tflite” file and then be distributed on the Android platform. However, to date (Feb. 2018), the TensorFlow Lite is still a preview version and cannot support certain functions and operations, for which we suggest developers to use the TensorFlow Mobile Library. In the process of application development, we only need to integrate the TensorFlow Mobile Library and the “pb” model in assets directory. Then, we invoke the model using TensorFlow Java API to perform prediction or classification.

In our implementation, the NN model is stored in the cache on Android platforms once the mobile application is loaded. LruCache is a cache tool supported by Android API to use the Least Recently Used algorithm. This cache stores the most recently used object in a “strong reference” in the LinkedHashMap and removes the least recently used object from memory before the cache value reaches the preset value. Then, the Android projects can directly call the LruCache.get() function to load the NN model file from the cache.

In our evaluation, we select two state-of-the-art networks, LeNet\cite{33} and AlexNet\cite{3}. To further cut down the parameter storage, we leverage the quantization techniques\cite{17} supported by TensorFlow to reduce the bit width of weight from 32 bits to 8 bits. We refer the readers to Ubiear for more details in implementing the deep learning based project on Android platforms.

### Recognition tasks, datasets, and models.

| No. | Task    | Model | Dataset             |
|-----|---------|-------|---------------------|
| T1  | Digit   | LeNet | 60000 pieces of data from MNIST\cite{34} |
| T2  | Image   | LeNet | 60000 pieces of data from CIFAR-10\cite{35} |
| T3  | Image   | AlexNet | 60000 of pieces data from CIFAR-10\cite{35} |
| T4  | Audio   | LeNet | 7500 pieces of data Ubisound\cite{36} |

Mobile platforms. Table 3 summarizes their resource constraints, which are tested by the GeeBench4 Tool\cite{37}, a professional software which measures the processor performance through a comprehensive set of benchmarks. These universal mobiles and embedded devices are equipped with different process power, diverse battery, and memory/cache capacity. The trained NNs are stored in the L2 cache of mobile platforms.

#### 3.2 Micro-benchmark of NN compression techniques

To determine the appropriate hyper-parameters for NN compression techniques described in Section 2, we first compare various metrics under different hyper-parameters setup on the basis of LeNet model as shown in Fig. 6.

**k value in SVD-based compression technique (Tech1).** The SVD decomposition methods can be used in the CONV and FC layers. When we adopt SVD decomposition to the final FC layers, the value of $k$ stands for the neuron numbers at the inserted layer between FC1 and FC2. Assuming that both FC1 and FC2 contain $m$ neurons, we compare the following three options: $k_1 = m/12$, $k_2 = m/6$, and $k_3 = m/4$. Figure 6a shows the following results: all these settings present competitive accuracy. As the
Fig. 6 Impact of various compression hyper-parameters to the metrics for different techniques. (a) Different $k$ values on Tech1 at FC layers, (b) different $k$ values on Tech1 at CONV layers, (c) different $k$ values on Tech2 at FC layers, (d) different $γ$ values on Tech6 at CONV layers, and (e) different $θ$ values on Tech7 at CONV layers. In this figure, $A$ means accuracy, $S_p$ is parameter size, $S_f$ is the intermediate feature size, $T$ stands for latency, and $E$ and $C$ are energy cost and MAC number, respectively. x-axis shows a different parameter setup, and y-axis is the accuracy/cost ratio of compressed layer to origin layers.

$k_1 = m/12$ setting exhibits the best performance, we configure the value of $k$ in SVD decomposition to be $k = m/12$. Specifically, the parameter size is reduced to around 15%, process delay is shortened to 20%, energy consumption is decreased to 56%, and the MAC computation amount is lowered to 80% under $k_1 = m/12$. This condition incurs an 8% increase in the output features owing to the inserted FC layer. Figure 6b shows the results of adopting the SVD decomposition technique at the CONV layers. All these settings can dramatically reduce the parameter size, latency, energy consumption, and MAC amount. Specifically, $k_2$ and $k_3$ lead to few accuracy loss (less than 8%), and $k_2$ leads to the less intermediate feature size (132%). Thus, we set the value of $k$ in sparse-coding as $k_2$ when we use Tech2 to the CONV layers.

**k value in sparse-coding compression technique (Tech2).** The sparse-coding methods can be used to divide the weights into the matrix multiplication of coefficient and basis. We test three parameter settings, $k_1 = m/12$, $k_2 = m/6$, and $k_3 = m/4$, at the CONV layers. Figure 6c shows the impact of basis number $k$ on the NN performance. All the compressed layers under three $k$ value settings yield an equivalent accuracy (about 98%) with the original layer. The $k_1$ can achieve 9× parameter storage and 3× latency reduction, 2× energy, and 5× computational efficiency. Therefore, we set $k = m/6$ for the FC layers in the following experiments.

**Depth multiplier in depth-wise separable technique (Tech6).** When depth-wise separable technique (Tech6) compresses the CONV layers, the depth multiplier $γ$ serves as an important hyper-parameter. Figure 6d compares the compressed NN’s performance under three settings: $γ_1 = 1$, $γ_2 = 0.75$, and $γ_3 = 0.5$. All these setups lead to minimal accuracy loss, and $γ_3 = 0.5$ exhibits the best performance that reduces the parameter storage, latency, energy, and computation cost by 51%, 41%, 40%, and 76%, respectively. Thus, we set $γ = 0.5$ in the following experiments.

**Sparse random multiplier in sparse random compression technique (Tech7).** Figure 6e shows the NNs’ accuracy and cost under $θ_1 = 0.25$, $θ_2 = 0.5$, and
The values $\theta_1$ and $\theta_2$ witness less performance improvement, i.e., they still cost about 80% energy and 80% computation. The value $\theta_3 = 0.75$ causes 75%, 62%, 29%, and 72% reduction in parameter storage, latency, energy cost, and computation, respectively, with only 0.4% accuracy loss. Thus, we select $\theta = 0.75$ to be our experimental setting.

### 3.3 Baseline comparison

In this section, we conduct adequate experiments to test the performance of different compression techniques and determine underlying relationships of the compressed NN to its accuracy, model size, energy cost, latency, and computation on Android platforms.

We train a 7-layer LeNet (CONV1, pool1, CONV2, pool2, FC1, FC2, and output) using the MNIST dataset and a 12-layer AlexNet (CONV1, pool1, CONV2, pool2, CONV3, CONV4, CONV5, pool3, FC1, FC2, FC3, and output) using the CIFAR-10 dataset and test their execution accuracy and cost on the RedMi 3S platform. We perform no compression techniques for the first two layers (CONV1 and pool1) to maintain additional details of the real world at the first several layers to preserve inference accuracy. SVD-based compression (Tech1), sparse-coding compression (Tech2), weight pruning (Tech3), and global average pooling (Tech10) are used to compress the FC layers. On other hand, the SVD-based compression (Tech1), sparse and direct sparse compression (Tech4 and Tech5, respectively), depth-wise separable technique (Tech6), sparse random compression (Tech7), fire module (Tech8), and MlpConv module (Tech9) are adopted to compress the CONV layers.

Figure 7 shows the comparison results for the special layers in the LeNet and AlexNet model. The $y$-axis stands for the ratio of the compressed layer’s cost to that of the origin layer. We apply SVD-based compression (Tech1) in the first FC layer and the second CONV layer. The other techniques only target at one certain layer type. Sparse-coding compression (Tech2) and weight pruning (Tech3) are used for the
first FC layer and global average pooling (Tech10) are used for all the FC layers. Meanwhile, sparse and direct sparse technique (Tech4 and Tech5), depth-wise separable (Tech6), sparse random compression (Tech7), fire module (Tech8), and MlpConv module (Tech9) are used for the second CONV layer. Thus, all of these layer compression techniques can reduce execution cost with minimal accuracy loss.

Specifically, SVD-based compression (Tech1) and sparse-coding compression (Tech2) adopt low-rank factorization method to reduce the weight size by 50%. As the CONV layer is weight-shared, the change for convolution kernel will lose larger accuracy than the FC layer in this strategy. Weight pruning (Tech3), sparse and direct sparse compression (Tech4 and Tech5, respectively), and sparse random compression (Tech7) adopt matrix sparsity method to achieve model compression. Their parameters amounts can be reduced by 70%, relative to the given sparsity. Depth-wise separable (Tech6), fire module (Tech8), and MlpConv module (Tech9) adopt a special convolution structure to achieve higher accuracy and lower parameter values. Global average pooling (Tech10) replaces the FC layers with global average layer to avoid huge weights of the FC layers. We can see Tech10 reduces the weight parameter to 1%. Meanwhile, the SVD-based compression (Tech1), sparse-coding compression (Tech2), and fire module (Tech8) increase the storage usage of the intermediate features, which may lead to more memory access energy cost; the overall energy consumption of these layers are reduced by 40%, 40%, and 10%, respectively. These findings are attributed to the dramatically reduced parameter storage and MAC computation, which both contribute to energy efficiency. The MAC computation is decreased to less than 20% by using SVD-based compression (Tech1) at the FC layers, SVD-based compression (Tech1) at the CONV layers, sparse and direct sparse compression (Tech4 and Tech5, respectively), depth-wise separable (Tech6), fire module (Tech8), MlpConv module (Tech9), and global average pooling (Tech10). Sparse-coding compression (Tech2), weight pruning (Tech3), and sparse random compression (Tech7) reduce around 40% of MAC computation amount.

To show how the layer compression techniques affect the accuracy/cost of the total NN model, we perform another series of experiments (Fig. 8). In general, the compression techniques at the FC layers lead to much considerable parameter reduction than CONV layer compression, for example SVD-based compression (Tech1), sparse-coding compression (Tech2), weight pruning (Tech3), and global average pooling (Tech10) at the FC layers reduce about 60% of parameter storage; SVD-based compression (Tech1), sparse and direct sparse compression (Tech4 and Tech5, respectively), and sparse random compression (Tech7) at the CONV layers decrease 10% of the parameter value. The CONV compression methods incur more computational efficiency than the compression at the FC layers.

Table 4 summarizes the performance of the compression methods. First, the low-rank factorization and the matrix sparsity all yield low accuracy, whereas the special network structures are even more accurate than the initial model. Global average pooling can dramatically reduce the parameter size and energy cost. Thus, the most recent novel networks all replace the final FC layers with the global average pooling layer. We think this process is an important solution for deploying the NN on mobile or embedded devices.

**Summary and discussion.** First, although the layer compression techniques, such as the SVD-based compression (Tech1) and sparse-coding compression (Tech2), increase certain layers, they cause no dramatically impact on the sum of the intermediate features of the entire NN. Second, the energy consumption not only includes the MAC consumption, but also the memory access cost to the intermediate features and the parameters (Fig. 8). Third, as for the inference latency of the NN on Android platforms, we test NNs on the same Android device with unchanged mobile processor, and observe that the delay forms no direct relationship with the MAC amount or the number of memory access, and it is jointly impacted by the NN’s execution computation and storage and the dynamic CPU usage of the devices.

**3.4 Performance over different recognition tasks**

In this section, we compare the performances of the compressed NN with different compression techniques over different recognition tasks (T1–T4). The weighted sum of five metrics is as follows:

\[
\text{Arg min } \text{Sum} = \sum_{i=1}^{2} p_i \cdot r_i
\]

where \( r_i \) shows the performance ratio of the compressed NNs to the origin model on the RedMi 3S smartphone. Specifically, 
\[
\begin{align*}
& r_1 = \frac{A_{compressed}}{A_{origin}}, \\
& r_2 = \frac{S_{compressed}}{S_{origin}}, \\
& r_3 = \frac{T_{compressed}}{T_{origin}}, \\
& r_4 = \frac{C_{compressed}}{C_{origin}}, \\
& r_5 = \frac{E_{compressed}}{E_{origin}}
\end{align*}
\]
Fig. 8 Comparison of the impact of different compression techniques to the whole model. (a) LeNet model and (b) AlexNet model. Tech1 is used for FC1 and CONV2, Tech2 and Tech3 act on FC1, Tech4 and Tech5, Tech6, Tech7, Tech8, and Tech9 target at CONV2, and Tech10 is used for all FC layers. The y-axis shows the overhead of compressed model compared with the original model.

Table 4 Performance of compression techniques on LeNet + MNIST and AlexNet + CIFAR-10, as evaluated on RedMi 3S phone (Device1).

| Compression technique                  | AlexNet + CIFAR-10 | LeNet + MNIST |
|----------------------------------------|--------------------|---------------|
| Initial model                          | 82.12, 54.39, 180, 65.23 | 99.3, 12.49, 32, 2.30 |
| SVD-based compression for FC (Tech1)   | 77.16, 44.02, 230, 61.29 | 99.14, 4.03, 12, 1.51 |
| SVD-based compression for CONV (Tech1) | 60.32, 52.68, 128, 43.25 | 92.19, 12.32, 31, 1.97 |
| Sparse-coding compression (Tech2)      | 79.12, 44.02, 230, 61.29 | 98.78, 4.30, 12, 1.51 |
| Weight pruning (Tech3)                 | 83.79, 1.56, 185, 30.72 | 97.89, 3.63, 32, 1.05 |
| Sparse and direct sparse (Tech4 and Tech5) | 75.63, 52.19, 210, 93.48 | 98.75, 12.29, 20.5, 1.73 |
| Depth-wise separable (Tech6)           | 78.26, 52.22, 122, 37.21 | 98.7, 12.30, 33, 1.75 |
| Sparse random (Tech7)                  | 57.31, 52.63, 195, 47.34 | 98.9, 12.44, 25, 2.16 |
| Fire module (Tech8)                    | 80.24, 52.28, 120, 37.21 | 99.1, 12.34, 31, 2.07 |
| MlpConv module (Tech9)                 | 83.07, 58.55, 330, 112.81 | 98.9, 12.32, 31, 2.08 |
| Global average pool (Tech10)           | 85.00, 2.35, 160, 44.90 | 97.1, 0.25, 4, 1.10 |

and we set their coefficients as follows: $p_1 = 0.4$, $p_2 = 0.2$, $p_3 = 0.1$, $p_4 = 0.1$, and $p_5 = 0.2$. These coefficients represent the importance of five metrics in our experiments and can be manually changed according to the performance requirements and device resource limitation. Given the different device resources and user demands, the settings of these coefficients vary. We can set the high proportion of accuracy to guarantee the correct prediction on low resource limitation devices. In addition, we can achieve the deployment of NN with a minimal loss of accuracy. Table 5 shows the best compression technique for each recognition task with the minimal weighted sum of multiple performance metrics.
Discussion. Different compression techniques are appropriate for different recognition tasks. We select the compression techniques for each recognition task according to a simple scheme, i.e., the weighted sum of the performance ratio described in Eq. (4). In the future work, we can design the efficient and adaptive selection rules for the compression techniques.

3.5 Performance on different resource constraints

As mentioned in Section 3.1, different devices feature diverse process capabilities, storage power, and battery capacities, which lead to various resource constraints, thus affecting the compression ratio. Thus, in this section, we use different NN compression techniques to compress the LeNet (MNIST) model and test their practical performances on six Android devices. We select the compression technique for each device according to Eq. (4). The device resource limitations will affect the coefficient $p_1 - p_5$ settings. For example, the Device1 presents the lowest MAC process rate (691.3 Mflops), the largest battery power (4100 mAh), and a medium L2-Cache capacity (2 MB), we can set the related coefficients to the compression ratio of computation, energy, and parameter storage and give the best performance under this mobile setting. Table 6 shows the details of metric coefficient settings for different devices and the final selected compression techniques based on min Eq. (4). Among these results, Device2 and Device3 have near performance about the memory and CPU, thus these two settings all choose the Depth-wise separable technique. For other devices, the choice of compression techniques has strong relation with the coefficients which means the trade-off of the device’s hardware performance.

Discussion. Although the coefficients are diverse, several compression techniques still cannot achieve the best joint performance, given that all of these single compression techniques can exhibit different levels of performance improvement and bottlenecks. For example, fire module (Tech8) yields the best accuracy but achieves the lowest storage reduction, whereas sparse and direct sparse compression (Tech4 and Tech5),

### Table 5 Performance of different recognition tasks which choose best compression techniques with Eq. (4).

| Metric ratio                        | Accuracy loss ratio $r_1 = A_{\text{origin}} / A_{\text{compressed}}$ | Parameter compression ratio $r_2 = S_{\text{origin}} / S_{\text{compressed}}$ | Latency compression ratio $r_3 = T_{\text{origin}} / T_{\text{compressed}}$ | MAC amount compression ratio $r_4 = C_{\text{origin}} / C_{\text{compressed}}$ | Energy cost compression ratio $r_5 = E_{\text{origin}} / E_{\text{compressed}}$ |
|-------------------------------------|-------------------------------------------------|---------------------------------|---------------------------------|---------------------------------|---------------------------------|
| T1 (Weight pruning (Tech3))         | 1.01                                            | 0.21                            | 0.44                            | 0.30                            | 0.45                            |
| T2 (Global average pool (Tech10))   | 1.07                                            | 0.02                            | 0.58                            | 0.62                            | 0.48                            |
| T3 (Depth-wise separable (Tech6))   | 1.04                                            | 0.32                            | 0.23                            | 0.13                            | 0.38                            |
| T4 (Sparse and direct sparse (Tech4 and Tech5)) | 1.01 | 0.41 | 0.43 | 0.35 | 0.39 |

### Table 6 Performance of NN compression techniques on different resource-constrained Android devices.

| Technique with mini Eq. (4) | Accuracy loss | Parameter compression | Latency compression | MAC amount compression | Energy cost compression |
|-----------------------------|---------------|-----------------------|---------------------|------------------------|------------------------|
| Device1 (Weight pruning (Tech3)) | 1.01 ($p_1 = 0.5$) | 0.21 ($p_2 = 0.2$) | 0.44 ($p_3 = 0.1$) | 0.30 ($p_4 = 0.1$) | 0.45 ($p_5 = 0.1$) |
| Device2 (Depth-wise separable (Tech6)) | 1.04 ($p_1 = 0.3$) | 0.32 ($p_2 = 0.3$) | 0.23 ($p_3 = 0.1$) | 0.13 ($p_4 = 0.1$) | 0.38 ($p_5 = 0.2$) |
| Device3 (Depth-wise separable (Tech6)) | 1.04 ($p_1 = 0.4$) | 0.32 ($p_2 = 0.2$) | 0.23 ($p_3 = 0.1$) | 0.13 ($p_4 = 0.1$) | 0.38 ($p_5 = 0.2$) |
| Device4 (Sparse and Direct sparse (Tech4 and Tech5)) | 1.01 ($p_1 = 0.2$) | 0.41 ($p_2 = 0.2$) | 0.43 ($p_3 = 0.1$) | 0.35 ($p_4 = 0.2$) | 0.39 ($p_5 = 0.3$) |
| Device5 (Sparse coding compression (Tech2)) | 1.01 ($p_1 = 0.3$) | 0.18 ($p_2 = 0.2$) | 0.22 ($p_3 = 0.1$) | 0.35 ($p_4 = 0.2$) | 0.25 ($p_5 = 0.2$) |
| Device6 (Fire Module (Tech8)) | 0.94 ($p_1 = 0.4$) | 0.82 ($p_2 = 0.1$) | 0.23 ($p_3 = 0.1$) | 0.12 ($p_4 = 0.2$) | 0.42 ($p_5 = 0.2$) |
respectively) feature the largest latency and energy cost decrease with a medium storage reduction ratio. Overall, no compression technique can achieve the best reduction of all cost metrics simultaneously. Thus, we can further design complex compression techniques to address the performance bottlenecks of these single NN compression techniques and carry forward their advantages.

4 Run-Time NN Compression Techniques

Orthogonal to the compression techniques described above, NNs can also be optimized at run-time to preserve their execution accuracy and reduce unnecessary resource cost on energy, latency, storage, or computation. The run-time NN compression techniques can be implemented through either scheduling the layer compression techniques presented in Section 2 to further balance the utility and cost of compressed NN on mobile devices or designing novel network architectures, such as combining with the residual network and multiple-branch network to save the execution NN cost.

4.1 Cost optimization for multiple NNs

4.1.1 Resource management for multiple NNs

This section presents the resource (energy, storage, latency, and cloud cost) management problem, that is, approximate model scheduling, and presents the design and implementation of an optimizing compiler and run-time scheduler, that is, MCDNN, to address this problem. Specifically, the MCDNN adaptively selects the compressed NN for each NN using existing layer compression methods to maximize the reference accuracy within pre-defined budgets of multiple NNs’ energy, memory, latency, and cloud cost based on the statistic performance catalog of the layer compression techniques. To guide the resource allocation between several NNs, it presents the heterogeneous streaming setting for NN execution and introduces two NN compression methods that exploit it.

4.1.2 Heterogeneous resource management for multiple concurrently running NNs

Different from existing schedulers/offloaders which typically emphasize one primary offloading aspect, LEO studies the heterogeneous resource management for concurrent sensor app execution on a single resource. LEO designs a Low-Power Unit (LPU) scheduler to maximize the energy efficiency and promptness for the unique workload of continuous and intermittent mobile apps without accuracy loss. This algorithm uses domain-specific signal processing knowledge to smartly distribute the sensor processing tasks across heterogeneous computational resources, i.e., CPU, co-processor, GPU, and the cloud, of high-end smartphones. To exploit short-lived but substantial optimization opportunities and remain responsive to the needs of near real-time apps, such as voice-based natural user interfaces, LEO runs as a service on an LPU to perform both frequent and joint schedule optimization for concurrent pipelines depending on the workload and network conditions.

4.2 Cost optimization with special NN architecture

4.2.1 Compressor-critic framework

The compressor-critic framework proposes DeepIoT, a unified approach that compresses all commonly used deep learning structures, including both FC and CONV, and their combinations. This approach compresses NNs into small dense matrices by finding the minimum number of non-redundant hidden elements, such as filters and dimensions required by each layer, with minimal accuracy loss. DeepIoT borrows the idea of dropping hidden elements from a widely-used regularization method called dropout, and it determines the optimal dropout probability for each hidden element by a compressor NN. The compressor NN is jointly optimized through a compressor-critic framework, which emulates the idea of the actor-critic algorithm from reinforcement learning, to minimize the loss function. The compressed model generated by DeepIoT can directly use the existing TensorFlow library that runs on embedded and mobile systems without further modifications.

4.2.2 Fast inference via early exiting from NNs

This section presents a novel augmented Deep Learning (DL) architecture, BranchyNet, with several branches to allow test samples to exit the network early to save execution overhead. Specifically, BranchyNet is end-to-end trainable by solving a joint optimization problem on the weighted sum of the loss functions associated with the exit points. Once the network is trained, it allows the test samples to exit the NN early via multiple branches when the input data can already be inferred with high confidence. BranchyNet finds that the features learned at the early layers are often sufficient for the classification of a number of test data. A small number of difficult data will be used further to provide accurate
predictions.

4.2.3 Unnecessary computation pruning
This process introduces the Dynamic DNN (D2NN), a new type of feed-forward NN with a controller that allows selective execution to prune unnecessary computation depending on the resolution of input. Each controller module is a Q-network whose output is a decision that determines whether other modules can be executed. A D2NN (regular NN and controller NN) is trained end-to-end to optimize both the accuracy and computational efficiency by integrating back-propagation with reinforcement learning. The network uses the reward function in reinforcement learning to maximize the accuracy and efficiency of regular NNs.

4.3 Discussion
We study the key ideas in the above run-time NN compression approaches and ask the following questions: Can we design a run-time resource management algorithm to schedule both NN compression and execution to maximize resource utilization from multiple concurrent NNs on mobile devices? Can we study the estimation models for both hardware and software cost of NN execution on mobile devices to guide run-time NN compression? The above scheduling approaches have been proposed, but they either feature different optimization goals or fail to fully address the above questions.

The minimal depth of layers, the number of neurons at each layer, filter numbers, and channel numbers needed to preserve the accuracy for mobile applications are still an open problem. For example, the depth can increase the NN’s non-linear feature extraction capability, but deeper layers cannot constantly guarantee a better accuracy owing to the vanishing/exploding gradient problem. The special units and architectures, such as residual unit in ResNet, are introduced to solve this problem: the identified mapping reveals that deeper networks cannot always extract more features. The deep networks are difficult to deploy on mobile platforms or embedding devices. Thus, we can further study other general compression techniques to remove redundant layers, neurons, filters, and channels with competitive accuracy.

5 Conclusion
The interest in deep model compression and deployment on mobile devices is growing rapidly. In the last few years, various novel methods were presented to fit the DNNs into the mobile or embedded devices. At the same time, the number of research articles related to model compression area has been increasing exponentially. In this paper, we surveyed several compression techniques and deployed the compressed models on different Android platforms to compare six metrics: accuracy, parameter size, intermediate feature size, computation, latency, and energy consumption. Our survey shows the impact of different compression categories on the resource limitation. However, despite the advances in compression techniques, the current mechanism for balancing the resource limitation in mobile devices still requires optimization. Finally, we have discussed the potential new direction in both techniques and hardware. We hope that the issues presented in this paper will advance the discussion to speed up the process of edge computing.

Acknowledgment
This work was partially supported by the National Key Research and Development Program of China (No. 2018YFB1003605), Foundations of CARCH (No. CARCH201704), the National Natural Science Foundation of China (No. 61472312), Foundations of Shaanxi Province and Xi’an Science and Technology Plan (Nos. B018230008 and BD34017020001), and the Foundations of Xidian University (No. JBJZ171002).

References
[1] E. Lindholm, I. Nickolls, S. Oberman, and J. Montrym, NVIDIA Tesla: A unified graphics and computing architecture. IEEE Micro, vol. 28, no. 2, pp. 39–55, 2008.
[2] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al., In-datacenter performance analysis of a tensor processing unit, in Proc. 44th Annu. Int. Symp. Computer Architecture, Toronto, Canada, 2017, pp. 1–12.
[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, ImageNet classification with deep convolutional neural networks, in Proc. 25th Int. Conf. Neural Information Processing Systems, Lake Tahoe, NV, USA, 2012.
[4] K. Simonyan and A. Zisserman, Very deep convolutional networks for large-scale image recognition, arXiv preprint arXiv:1409.1556, 2015.
[5] K. M. He, X. Y. Zhang, S. Q. Ren, and J. Sun, Deep residual learning for image recognition, in Proc. IEEE Conf. Computer Vision and Pattern Recognition, Las Vegas, NV, USA, 2016, pp. 770–778.
[6] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, Rethinking the inception architecture for computer vision, in Proc. IEEE Conf. Computer Vision and Pattern Recognition, Las Vegas, NV, USA, 2016, pp. 2818–2826.
and A. Krishnamurthy, MCDNN: An approximation-based execution framework for deep stream processing under resource constraints, in Proc. 14th Annu. Int. Conf. Mobile Systems, Applications, and Services, Singapore, 2016, pp. 123–136.

[39] P. Georgiev, N. D. Lane, K. K. Rachuri, and C. Mascolo, LEO: Scheduling sensor inference algorithms across heterogeneous mobile processors and network resources, in Proc. 22nd Annu. Int. Conf. Mobile Computing and Networking, New York, NY, USA, 2016, pp. 320–333.

[40] S. C. Yao, Y. R. Zhao, A. Zhang, L. Su, and T. Aldelzaher, DeepIoT: Compressing deep neural network structures for sensing systems with a compressor-critic framework, in Proc. 15th ACM Conf. Embedded Network Sensor Systems, Delft, Netherlands, 2017, p. 4.

Kaiming Nan received the BS degree from Xidian University in 2017. He is currently a master student at Xidian University. His research interests include mobile computing, mobile data collection, and energy consumption optimization in mobile deep learning.

Junzhao Du received the BS, MS, and PhD degrees from Xidian University in 1997, 2000, and 2008, respectively. He is currently a professor and PhD advisor at Xidian University. His research interests include mobile computing, cloud computing, and IoT systems. He is the member of ACM/IEEE, senior member of CCF, and vice secretary of ACM Xi’an Chapter.

Sicong Liu received the BS and MS degrees from Xidian University in 2013 and 2016, respectively. She is currently a PhD student at Xidian University, and a visiting student at Rice University. Her research interests include mobile computing, adaptive and automatic deep model compression and optimization, and mobile computing.

Hui Liu received the BS, MS, and PhD degrees from Xidian University in 1998, 2003, and 2011, respectively. She is currently an associate professor in School of Software at Xidian University. Her research interests include big data analysis, task scheduling, and mobile computing. She is the member of ACC, IEEE, and CCF.

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

[42] V. R. Konda and J. N. Tsitsiklis, Actor-critic algorithms, in Proc. 13th Int. Conf. Neural Information Processing Systems, Denver, CO, USA, 2000, pp. 1008–1014.

[43] S. Teerapittayanon, B. McDanel, and H. T. Kung, BranchyNet: Fast inference via early exiting from deep neural networks, in Proc. 23rd Int. Conf. Pattern Recognition, Cancun, Mexico, 2016, pp. 2464–2469.

[44] L. L. Liu and J. Deng, Dynamic deep neural networks: Optimizing accuracy-efficiency trade-offs by selective execution, arXiv preprint arXiv:1701.00299, 2018.