# APQ: Automated DNN Pruning and Quantization for ReRAM-Based Accelerators

Siling Yang, Shuibing He, *Member, IEEE*, Hexiao Duan, Weijian Chen, Xuechen Zhang, *Member, IEEE*, Tong Wu, and Yanlong Yin

***Abstract*—Emerging ReRAM-based accelerators support in-memory computation to accelerate deep neural network (DNN) inference. Weight matrix pruning is a widely used technique to reduce the size of DNN models, thereby reducing the resource and energy consumption of ReRAM-based accelerators. However, existing pruning works for ReRAM-based accelerators have three major issues. First, they use heuristics or rules from domain experts to prune the weights, leading to sub-optimal pruning policies. Second, they use row or column-level coarse-granularity methods to prune weights, resulting in poor compression rates with model accuracy constraints. Third, they only apply the weight pruning technique individually, losing the compression opportunity of both pruning and quantization. In this article, we propose an Automated DNN Pruning and Quantization framework, named APQ, for ReRAM-based accelerators. First, APQ adopts reinforcement learning (RL) to automatically determine the pruning policy for DNN layers for a global optimum. Second, it prunes and maps weight matrices to a ReRAM-based accelerator in a finer granularity of column-vector, which improves the compression rates with the accuracy constraints. To address the dislocation problem, it uses a new data path in ReRAM-based accelerators to correctly index and feed input to matrix-vector computation. Third, to further reduce resource consumption, APQ also leverages reinforcement learning to automatically determine the quantization bitwidth of each layer of the pruned DNN model. Experimental results show that, APQ achieves up to 4.52X compression rate, 4.11X area efficiency, and 4.51X energy efficiency with similar or even higher model accuracy, compared to the state-of-the-art work.**

***Index Terms*—ReRAM-based accelerator, pruning, quantization, reinforcement learning.**

## I. Introduction

DEEP neural networks (DNNs) have become the dominant approach to solving a variety of computing problems in computer vision, natural language processing, robotics among many other fields. Leveraging emerging devices and non-traditional computing systems [1], [2], [3], [4] is an ideal approach to accelerate DNN training and inference. Resistive random access memory (ReRAM) is an attractive candidate for DNN accelerators because of its superior characteristics of extremely low energy leakage, high-density storage, and high-parallel in-situ computation, compared to other NVM devices such as spin-transfer torque MRAM (STT-MRAM) and phase-change memory (PCM).

ReRAM-based accelerators can perform matrix-vector multiplication efficiently in the convolutional (CONV) layers and fully-connected (FC) layers of DNNs [5], [6]. They store weight matrices of DNN filters in crossbar arrays. The weights are represented as the conductances, which conduct dot-product with the voltages converted from the input feature maps. The current at the end of each bitline can be summed up and the dot-product operations are executed simultaneously, thus reducing the massive amount of data movements between memory and arithmetic units required in the von Neumann computer architecture. However, considering the limited resource and energy supply of modern ReRAM-based accelerators, it is inefficient to directly store the whole weight matrix on the accelerator.

A prevalent solution is to compress DNN models before mapping them to the hardware. Recent researches show that weight sparsity increases as the bits-per-cell decreases [7]. After applying weight sparsifying algorithms (e.g., quantization [8] and low-rank matrix factorization [9]) during training, up to 78% of crossbar cells may store zero weights. As a result, pruning these values before mapping to hardware saves the usage of crossbars and removes unnecessary computations. However, existing DNN pruning algorithms designed for ReRAM-based accelerators have the following three major issues.

First, *these algorithms prune the weights using heuristics or rules*. For example, they typically used heuristics (e.g., patterns and all-zero rows/columns) to direct the pruning process [10], [11], [12]. Because of the nature of heuristics-based algorithms, they may prune some nontrivial weights or preserve some trivial weights. Besides, they mainly aim to maximize the compression rate but ignore its impact on model accuracy [7], [10]. Therefore, it would be very difficult for them to find a pruning policy on ReRAM-based accelerators that achieves a global optimum for DNN models.

Second, *they usually use coarse-granularity methods to prune DNN models*. To reduce the hardware design complexity of

Manuscript received 21 April 2022; revised 14 June 2023; accepted 20 June 2023. Date of publication 27 June 2023; date of current version 10 July 2023. This work was supported in part by the National Key Research and Development Program of China under Grant 2021ZD0110700, in part by the Program of Zhejiang Province Science and Technology under Grant 2022C01044, in part by the National Science Foundation of China under Grant 62172361, and in part by the Zhejiang Lab Research Project under Grant 2020KC0AC01. Recommended for acceptance by A. Sussman. *(Corresponding author: Shuibing He.)*

Siling Yang, Shuibing He, Hexiao Duan, Weijian Chen, Tong Wu, and Yanlong Yin are with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China, and also with Zhejiang Laboratory, Hangzhou 311100, China (e-mail: slingzjunet@zju.edu.cn; heshuibing@zju.edu.cn; duanhexiao@zju.edu.cn; weijianchen@zju.edu.cn; wu.tong@zju.edu.cn; yinyanlong@gmail.com).

Xuechen Zhang is with the School of Engineering and Computer Science, Washington State University Vancouver, Vancouver, WA 98686 USA (e-mail: xuechen.zhang@wsu.edu).

Digital Object Identifier 10.1109/TPDS.2023.3290010

ReRAM-based accelerators, existing algorithms tend to prune weight matrices in a coarse granularity, i.e., rows or columns of DNN weight matrices. SNrram [13] and XCS [14] prune the unimportant columns of the weight matrix to exploit the sparsity. SRE [7] prunes all-zero vectors in weight matrices in either row or column direction for OU-based ReRAM-based accelerators. However, as some nontrivial elements in the row or column are also removed, these approaches may lead to a low model compression rate with the model accuracy constraint.

Third, *they do not take the advantages of both pruning and quantization.* Quantization [15], [16] is another technique that uses low bitwidth instead of full precision to represent weight matrices for DNN compression. However, existing pruning works [10], [11] only focus on removing unimportant elements but ignore the opportunity of quantization. Although other studies [17] pay attention to quantization, they omit the effects of weight pruning. Therefore, none of them obtain the most compacted DNN models.

In this paper, we propose an Automated DNN Pruning and Quantization framework, named APQ, for ReRAM-based accelerators. First, it adopts reinforcement learning (RL) to automatically make the pruning policy for each DNN layer for a global optimum. The RL agent receives the configuration and characteristic of the layer as observation. Then it outputs the expected pruning rate of the weight matrix for the layer. After the pruning rates of all the layers are decided, we leverage the simulator of ReRAM-based accelerators as the environment to obtain feedback. The RL agent then uses the feedback to compute reward of this pruning policy (i.e., a list of pruning rates for all layers). After multiple epochs of searching both locally and globally, the reward converges and the optimal pruning policy is decided.

Second, APQ prunes and maps weight matrices on ReRAM-based accelerators in a finer granularity of column-vector. It removes less important column-vectors and shifts the remaining vectors left. Then it maps the pruned weight matrices to crossbar arrays. To solve the dislocation problem, we add a weight indexing structure to the data path of the architecture of ReRAM-based accelerators. The control unit can feed matching input to conduct matrix-vector multiplication at the level of the operation unit (OU) [7].

Third, to take the advantages of both pruning and quantization, we also leverage another reinforcement learning algorithm to automatically determine the quantization bitwidth of each layer of the pruned DNN model. It takes the bitwidth in the action space and also uses the feedback of the simulator to compute the reward of the quantization policy (i.e., a list of bitwidths for all layers). Compared to AUTO-PRUNE [18] (the conference version) that only performs pruning, APQ supports both pruning and quantization, thus further saving resources and energy.

In summary, this paper offers the following contributions:

- We design an automated pruning framework for ReRAM-based accelerators, APQ, which searches for a global optimum pruning policy for each DNN layer without requiring rule-based heuristics and domain experts.
- We prune and map weight matrices in a finer granularity of column-vector for improved compression rate and accuracy. We also design a new data path to support the column-vector pruning using the OU mechanism.
- To further compress the model, we propose another reinforcement learning algorithm to automatically determine the quantization bitwidth of each DNN layer based on the pruned result of the model.
- We evaluate APQ with three DNN models including AlexNet [19], VGG16 [20], and Plain20 [21] on two datasets, i.e., CIFAR10 [22] and MNIST [23]. Compared to the state-of-the-art PIM-Prune [10], APQ achieves up to 4.52X compression rate, 4.11X area efficiency, and 4.53X energy efficiency with a similar or even higher accuracy.

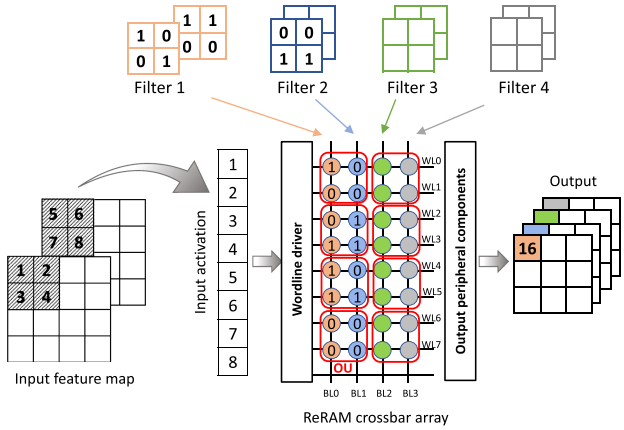



Fig. 1. Illustration of mapping filter weights to a crossbar array used in the architecture of ReRAM-based accelerators. BL: bitline; WL: wordline; OU: operation unit.

## II. Background and Related Work

### A. Mapping Filter Weights of DNNs in ReRAM-Based Accelerators

The architecture of ReRAM-based accelerators consists of crossbar arrays and input/output peripheral components. In a crossbar array, each bitline is connected to each wordline through a ReRAM cell. Input peripheral circuits (e.g., wordline decoders) convert inputs to the voltage pulses and feed them into the corresponding wordlines. Each ReRAM cell conveys the inner product between the driving voltage and the cell conductance and generates current which reaches the output peripheral circuits, e.g., analog-to-digital converters (ADCs). The accumulated current at the end of each bitline is converted by ADCs to the digital values representing the partial sum of a convolution operation. Because of the overhead from ADC, matrix-vector multiplication in ReRAM-based accelerators must be executed at a smaller granularity, called an operation unit (OU) [7]. Fig. 1 shows a simplified example of an $8 \times 4$ crossbar array. When OU is enabled, only two wordlines and two bitlines are turned on concurrently within the crossbar array in one cycle.

When a DNN model is mapped onto a ReRAM crossbar array, the synaptic weights of neurons are encoded as the conductances of ReRAM cells in crossbars. Fig. 1 demonstrates a mapping scheme for ReRAM-based accelerators. It shows the convolution operations between four $2 \times 2 \times 2$ filters and one $4 \times 4 \times 2$

input feature map in a convolution layer. Each element of the filter is mapped to one bitline of the crossbar array. A set of $2 \times 2 \times 2$ activation values derived from the input feature map is sent to eight wordlines of the crossbar array after being converted to the input voltages. Because only two wordlines and two bitlines in one OU are activated in one cycle, the four convolution operations through the four bitlines are completed in eight cycles. At the end of each bitline, the accumulated currents are converted to the digital value by ADC, which corresponds to an element in one channel of the output feature map. Then, the input sliding window moves right (or down) and the corresponding elements in the input feature maps are fed into the crossbar array in the next cycles. When all the elements of the input feature map complete convolution operations with the four filters in this layer, the output feature map of $3 \times 3 \times 4$ can be stored in buffers and used as the input for the next layer.

### B. Weight Pruning for ReRAM-Based Accelerators

Filter weight matrices of DNN models are sparse because they often store zero or redundant weights, which have a trivial impact on the accuracy of the models [13]. Therefore, before mapping the weight matrices to crossbars of ReRAM-based accelerators, they need to be pruned to reduce the number of occupied crossbars and their energy consumption. Researchers have designed several weight pruning schemes considering the tightly coupled crossbar structure in ReRAM-based accelerators [7], [10], [11], [24]. They can be used to effectively reduce the number of trivial elements exploiting the sparsity of the weight matrices.

The tightly coupled crossbar structure makes it difficult to exploit the sparsity of neural networks for ReRAM-based accelerators. ReCom is the first to exploit the sparsity of neural networks for ReRAM-based accelerators [25]. It explores the weight sparsity only in the granularity of matrix-row and crossbar-row. SNrram [13] and XCS [14] prune the unimportant columns of the weight matrix to exploit the sparsity. Lin et al. proposed to exchange columns of weight matrices to move non-zero elements together and store them in clusters separated from the clusters of zero elements [24]. Then the clusters of zero elements can be pruned. Yang et al. designed a sparse ReRAM engine that prunes all-zero vectors in weight matrices in either row or column direction for OU-based ReRAM-based accelerators [7]. They exploited the sparsity of weight matrices at the granularity of row/column vectors for a higher compression rate. For the pruning schemes designed for OU-based ReRAM-based accelerators, additional indexing tables are required in the data path to active correct OUs in subsequent cycles. PIM-Prune exploits the sparsity at the level of blocks of weight matrices [10]. It prunes the elements in both row and column directions. Most recently, pattern pruning uses patterns that represent irregular vectors of a particular shape to identify more zero elements for pruning [11].

### C. Weight Quantization for ReRAM-Based Accelerator

DNN models typically use data types of certain lengths to represent their parameters and inputs/outputs. Recent research [15], [16], [26], [27], [28], [29] shows that we can quantize the high bitwidth data to a low bitwidth format with no accuracy drop. Using quantization, the physical crossbars needed to store the given weights will be reduced. As a result, the memory usage will be saved and the data movement during training or inference will be reduced. As different layers of DNNs may have different sensitivity to quantization effects, it is better to adopt a non-uniform quantization strategy for different DNN layers. For example, Han et al. specified distinct bitwidths for CONV and FC layers [8]. Choi et al. performed 2-bit quantization for non-convolutional layers, leaving convolution layers in floating format. However, empirical or rule-based methods to determine the quantization strategy for a neural network may not work with another network. Hence, some works use genetic algorithms to automatically search the proper quantization policy for a given network [30], [31].

We compare APQ to the major compression schemes in Table I. APQ has four major differences from them. First, the existing schemes only consider the pruning policy, APQ leverages the poential of both pruning and quantization. Second, the existing schemes use the proposed heuristics or rules to find a compression policy. Because of the nature of heuristics-based algorithms, it would be very difficult to find a global optimum for the DNN model. APQ searches for an optimal solution globally based on reinforcement learning. Third, they were designed to maximize the compression rate of weight matrices. They may not meet the accuracy requirements. In contrast, APQ involves direct feedback from the ReRAM-based accelerators in the design loop, which makes better trade-offs between compression rate and accuracy. Fourth, existing approaches usually prune weight matrices in the granularity of matrix, block, or crossbar row/column. APQ performs the pruning in a finer granularity of column-vector based on the OU mechanism. It delivers a higher compression rate with the accuracy constraint.

### D. Accelerator Design Using Reinforcement Learning

AutoML based on reinforcement learning is designed to release human labor on searching configurations while there are vast search space and limited computational budgets. It is a popular search method with good performance, less assistance from humans, and high computational efficiency [17], [32]. Therefore, AutoML is widely used in neural architecture search. Inspired by the AutoML framework, AMC compresses DNN models automatically [21]. It achieves a higher compression rate and preserves better accuracy than heuristics-based model compression algorithms. More importantly, it does not require human expertise in the design. Recently, HAQ [33] was designed to decide a hardware-aware quantization policy for DNNs using AutoML. They targeted FPGA and ASCI-based accelerators. Similar to them, APQ uses AutoML to determine the pruning and quantization policies of weight matrices of DNN models.

## III. Design of APQ

The design objective of APQ is to automatically compress the original DNN weight matrix and map it to the ReRAM-based accelerator with reduced amount of crossbars and energy consumption considering the accuracy constraint. To reduce the crossbar consumption, APQ considers both pruning and

TABLE I
COMPARISON OF APQ WITH EXISTING WEIGHT MATRIX COMPRESSION SCHEMES FOR RERAM-BASED ACCELERATORS

| *Compression technique* | *Pruning Pattern* | *Consider Quantization* | *Automation* | *Hardware feedback* | *OU* |
|---|---|---|---|---|---|
| Lin et al. [24] | Unimportant weight groups | No | No | No | No |
| SRE [7] | All-zero row/column vectors | No | No | No | Yes |
| PIM-Prune [10] | Unimportant rows and columns | No | No | No | No |
| Pattern pruning [11] | Patterns | No | No | No | Yes |
| AUTO-PRUNE [18] | Unimportant column-vectors | No | Yes | Yes | Yes |
| APQ | Unimportant column-vectors | Yes | Yes | Yes | Yes |

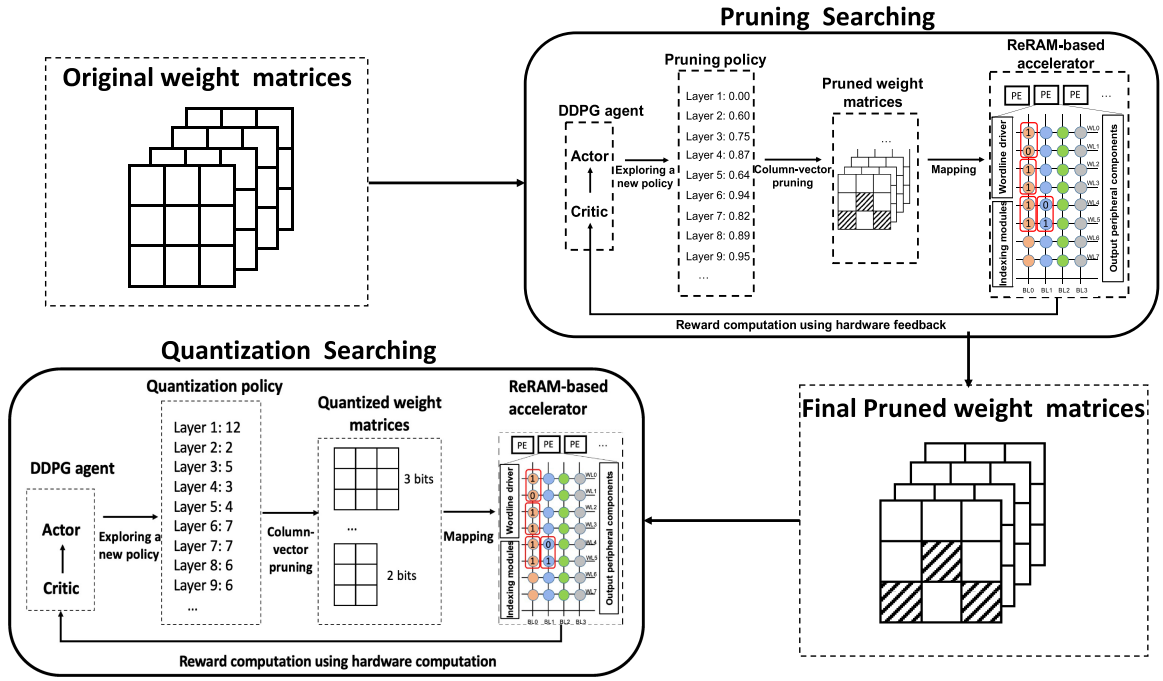



Fig. 2. Overview of the APQ framework.

quantization optimizations. While it is theoretically feasible to integrate these two aspects into a single reinforcement learning (RL) algorithm to achieve near-optimal compression decisions, doing so will introduce an immense search space, resulting in an excessively long execution time for the approach. To overcome this issue, we adopt a greedy approach to obtain the local optimal decisions for pruning and quantization separately with two individual RL algorithms. The goal of the first RL algorithm is to learn a pruning policy that minimizes the number of occupied crossbars used in ReRAM-based accelerators with the accuracy constraint. Based on the final result of the pruning searching, we then run the second RL algorithm to obtain a quantization policy that further reduces the number of occupied crossbars with the accuracy constraint. Fig. 2 shows the general architecture of the APQ framework, which includes the following components.

*DDPG Agent for Pruning:* APQ uses deep deterministic policy gradient (DDPG) to generate a pruning action, i.e., pruning policy for all layers of the DNN. The agent is a pair of actor-critic network [34]. The *actor* network predicts a new pruning policy for hardware feedback given the input of a state vector consisting of features of DNN models (e.g., the number of input channels) and ReRAM-based accelerators (e.g., the number of crossbars required before pruning). The *critic* network evaluates the importance of the action-state pair using Q-function [35].

*Pruning Policy:* The agent needs to explore a large action space of pruning policies. We use a list of pruning rates, each corresponds to a layer of the DNN model, to denote one pruning policy. In Fig. 2, the policy to be evaluated by the hardware assigns the pruning rate of 0.6 to layer 2, meaning that APQ will need to prune 60% of the elements in the filter weight matrix of layer 2 before mapping it to crossbars.

*Pruned Weight Matrices:* They are a list of filter weight matrices to be mapped to ReRAM-based accelerators after pruning. Different pruning policies may result in different pruned weight matrices, leading to the accuracy variation of the pruned DNN models and the changing resource consumption of hardware.

*ReRAM-Based Accelerator:* It is simulated using MNSIM [36]. Because APQ leverages the OU mechanism to explore fine-grained pruning, we add indexing modules to the data path of the processing elements (PEs) of ReRAM-based accelerators in the simulator.

*DDPG Agent for Quantization:* It is used to generate a quantization action, i.e., quantization policy for all layers of the DNN

TABLE II
SYMBOLS USED IN THE DDPG ALGORITHM

| Symbol | Meaning |
|---|---|
| $k$ | layer index |
| $t$ | layer type: CONV:1 ; FC: 0 |
| $inc$ | number of channels in the input feature map |
| $outc$ | number of channels produced by the convolution |
| $ks$ | number of elements of a convolving kernel |
| $h$ | height of the input feature maps |
| $w$ | width of the input feature maps |
| $s$ | stride of the convolution |
| $xb[k]$ | number of crossbars required for mapping layer $k$ |
| $xb_{saved}[k]$ | accumulated number of the crossbar saved from the first layer to layer $k-1$ |
| $xb_{rest}[k]$ | number of crossbars required from layer $k+1$ to the last layer |
| $size_{xb}$ | length of the crossbar size |
| $acc_{reram}$ | accuracy reported by the ReRAM-based accelerator simulator |
| $a_{k-1}$ | action from the last time step in the pruning plicy searching process |
| $b_{k-1}$ | action from the last time step in the quantization policy searching process |

model. It has a similar input as the DDPG agent for pruning, but the output is different.

*Quantization Policy:* Similar to the pruning policy, we use a list of quantization bitwidths, each related to a layer of a DNN model, to denote one quantization policy. In Fig. 2, the quantization policy assigns 5 to layer 3, meaning that we will use 5-bit width to represent the elements in layer 3 when mapping them to crossbars.

To efficiently support the weight quantization, we use the existing approach in MNSIM [36] to map multiple-bit weights to the crossbars in a flexible manner with low hardware overhead, as shown in Section III-E.

### A. DDPG Algorithm for Pruning

In this section, we describe the DDPG algorithm to search for an optimal pruning policy given direct hardware feedback. In the design of a DDPG agent, we need to define its *state space*, *action space*, and *reward function*.

*State Space:* For each layer $k$, we use a 12-dimensional feature vector as our observation. Specifically, the state vector $S_k$ for layer $k$ is defined as

$$(k, t, inc, outc, ks, h, w, s, xb[k], xb_{saved}[k], xb_{rest}[k], a_{k-1}) \tag{1}$$

where all the features are defined in Table II. It is worth noting that for fully-connected layers their $inc$ and $outc$ are equal to the number of input and output neurons respectively. To make the reinforcement learning model effective for both convolutional layers and fully-connected layers, we set both $ks$ and $s$ to 1 for the fully-connected layers even though they do not have such attributes.

*Action Space:* In order to achieve a fine-grained pruning decision, we choose the Actor's action space $a_k \in (0, 1]$ and prune the current layer with a pruning rate $a_k$ at the granularity of column-vectors. We describe the pruning algorithm in Section III-B.

*Reward Function:* As direct feedbacks, the simulator passes the number of occupied crossbars and the accuracy of the compressed DNN models to the DDPG agent to compute the reward for reinforcement learning given the current pruning policy. Specifically, we define our reward function to be related to both compression rate and DNN model accuracy in (2). It is challenging to design an effective reward function. Our research shows that directly adopting a linear form only achieves a suboptimal performance as discussed in Section IV-F. In this paper, we design the reward function empirically. The agent uses the reward function to obtain a reward and adjusts its policy based on the reward. The reward function is also designed to balance the impact of the compression rate and the accuracy of pruned DNN models. As a result, we can avoid the situation of a significant accuracy drop when exploring pruning policies to maximize the compression rate of crossbars.

$$Reward = \left(1 - \frac{1}{rate^{xb}_{compression}}\right)^{\alpha} \times acc_{reram} \tag{2}$$

In the equation, $\alpha$ is a scaling factor which is set to 2 in our experiments. $rate^{xb}_{compression}$ is a rate of the number of occupied crossbars for all the layers without pruning ($xb_{ori}$) to the number of occupied crossbars using the current pruning policy ($xb_{cur}$) as shown in (3).

$$rate^{xb}_{compression} = \frac{xb_{ori}}{xb_{cur}} \tag{3}$$

$$xb_{ori} = \sum_{k=0}^{L-1}(xb_{ori}[k]) \tag{4}$$

$$xb_{cur} = \sum_{k=0}^{L-1}(xb_{cur}[k]) \tag{5}$$

$xb_{ori}[k]$ and $xb_{cur}[k]$ are the number of occupied crossbars for layer $k$ without pruning and with the current pruning policy respectively. $xb_{cur}[k]$ is obtained from the simulator. $xb_{ori}[k]$ is calculated using the following equations.

For convolutional layer $k$,

$$xb_{ori}[k] = \left\lceil \frac{ks \times inc}{size_{xb}} \right\rceil \times \left\lceil \frac{outc}{size_{xb}} \right\rceil \tag{6}$$

For FC layer $k$,

$$xb_{ori}[k] = \lceil inc/size_{xb} \rceil \times \lceil outc/size_{xb} \rceil \tag{7}$$

### B. Column-Vector Based Pruning and OU Formation

Given a pruning policy generated by the DDPG actor, APQ needs to run pruning algorithms to mark nontrivial elements in filter weight matrices based on their weights. It only maps the nontrivial elements to crossbars. To save the resource and energy consumption of ReRAM-based accelerators, APQ prunes weight matrices in a granularity of column-vector and adjacently places the remaining vectors on the crossbars by shifting them. APQ's pruning policy differs from PIM-Prune [10] in two-fold. First, it prunes the model with a finer granularity (i.e., column-vector) while PIM-Prune prunes at the level of the rows/columns

**Algorithm 1:** The Column-Vector Based Pruning Algorithm.

**Require:** $g$: the granularity of the column-vector;
$rate_{pruning}$: the pruning rate generated by DDPG agents;
$outc$: number of output channels of a layer;
$inc$: number of channels of input feature map of the layer;
$ks$: kernel size of the layer;
$W$: 2D-weight matrix of the layer;

1: $num \leftarrow \lfloor ks \times inc/g \rfloor$
2: $dict \leftarrow \{\}, list \leftarrow []$
3: **for** $i \leftarrow 0, 1,..., (outc-1)$ **do**
4: **for** $j \leftarrow 0, 1,..., num-1$ **do**
5: $tmp\_sum \leftarrow 0$
6: **for** $k \leftarrow 0, 1, \ldots, g$ **do**
7: $tmp\_sum \leftarrow tmp\_sum + abs(W[g \times j + k][i])$
8: **end for**
9: $dict$.append$(key : (j, i), value : tmp_sum)$
10: **end for**
11: **end for**
12: $sorted\_list \leftarrow ascend\_sort\_by\_value(dict)$
13: **for** $i \leftarrow \lceil rate_{pruning} \times num \times outc \rceil - 1$,..., len(sorted_list)-1 **do**
14: $list$.append$(sorted_list[i].key)$
15: **end for**
16: **return** $list$

**Algorithm 2:** Vector Sequence Creation Algorithm.

**Require:** $h$: the number of column-vectors in each OU;
$list[i]$: the list of index pairs from Algorithm 1;

1: $Index \leftarrow \{\}$
2: $L \leftarrow getIndexLen(list)$
3: $cnt \leftarrow 0$
4: **for** $i$ =0, 1, ..., $(L-1)$ **do**
5: $tag[i] \leftarrow 1$
6: **end for**
7: **for** $i$ =0, 1, ..., $(L-1)$ **do**
8: **if** $tag[i] == 1$ **then**
9: $Index$.append$(list[i])$
10: $tag[i] \leftarrow 0$
11: $cnt \leftarrow 1$
12: **for** $j = i+1, \ldots, (L-1)$ **do**
13: **if** $list[i].x == list[j].x$ and $tag[j] == 1$ **then**
14: $Index$.append$(list[j])$
15: $tag[j] \leftarrow 0$
16: $cnt \leftarrow cnt + 1$
17: **if** $cnt == h$ **then**
18: break
19: **end if**
20: **end if**
21: **end for**
22: **end if**
23: **end for**
24: **return** $Index$

of a logical block (whose size is larger than the array size but smaller than the weight matrix size). Second, it supports a dynamic pruning rate for different layers while PIM-Prune is executed with a fixed pruning rate (i.e., the block size/array size).

We use Algorithm 1 for matrix pruning in APQ. Specifically, it calculates the number of column-vectors in a column of the weight matrix given the kernel size of the layer, the number of input channels, and the granularity of column-vectors (Line #1). Then for each column in the matrix, it scans through all the column-vectors and records their coordinates and the accumulated weights of the vectors in a dictionary (Line #3-11). Then we sort the vectors based on their weights (Line #14). Finally, based on the pruning rate, the vectors consisting of nontrivial weight elements are selected and returned (Line #15-18).

We use an example to illustrate the pruning and OU formation process in Fig. 3. In the example, we assume the granularity of a column-vector is 2. The weight matrix $W$ can be divided into 18 column-vectors. Each column-vector is indexed using its coordinate in the vector space. For example, the coordinate of the vector $[1, 6]^T$ in filter $F1$ (column 1) is (3, 1), where 3 represents that it is located in the third row in the vector space and 1 represents that it is in the first column in the vector space. After running the algorithm for the pruned weight matrix $W'$, nine vectors are pruned to achieve a pruning rate of 50% because their accumulated weights are smaller than those of the remaining vectors. One observation from the example is that the pruned vectors are randomly located in the weight matrix. In order to exploit the sparsity, APQ accumulates the remaining vectors to remove the holes. Then it leverages the support of OUs in ReRAM-based accelerators to perform matrix-vector multiplication.

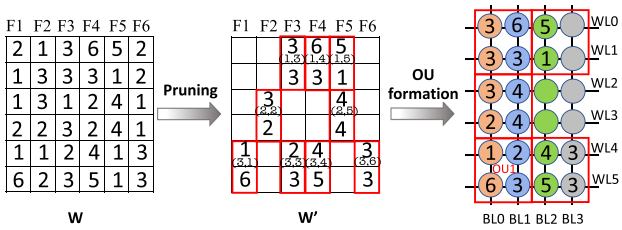



Fig. 3. Example of the pruning and OU formation process. The value in each cell represents the weight of the corresponding element in the filter. (x,y) represents the coordinate of a vector in the vector space. $W$ and $W'$ denote the original and pruned weight matrices respectively.

For the formation of OUs after pruning, we need to create a list of indexes of column-vectors based on the order of their access in crossbars in the sequence of computation of OUs. Then we will add a corresponding indexing module to the data path of ReRAM-based accelerators as discussed in Section III-C. We use Algorithm 2 to form OUs and create the index. Each OU has a fixed number of column-vectors $h$. In the vector space, we need to find $h$ vectors that are next to each other (Line #11-20). These $h$ vectors are computed together in one OU. The algorithm assigns all vectors in the list output from Algorithm 1 to their

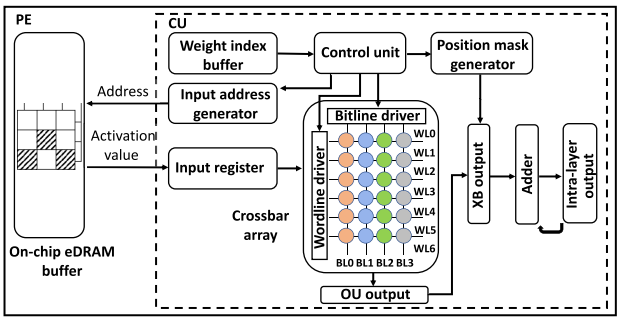



Fig. 4. Overview of the data path for the weight pruning approach.

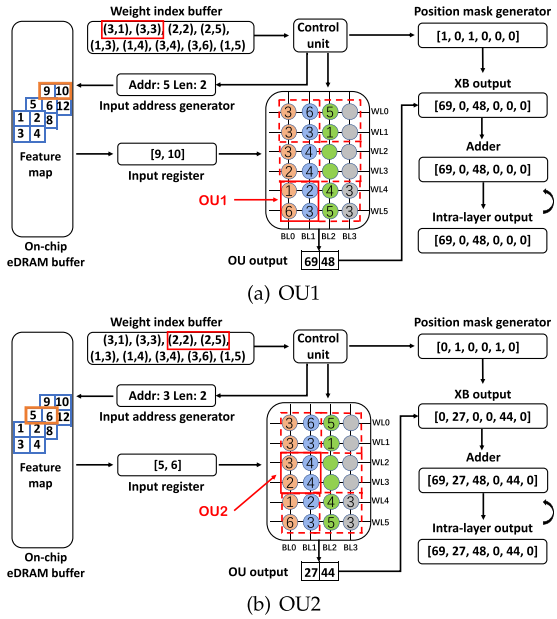



Fig. 5. States of data path components after the execution of OU1 and OU2.

corresponding OUs. The output of Algorithm 2 is an indexing list to be used by data paths of ReRAM-based accelerators. We continue using the example in Fig. 3 for illustration. The index output of Algorithm 2 is {(3, 1), (3, 3), (2, 2), (2, 5), (1, 3), (1, 4), (3, 4), (3, 6), (1, 5)}. We assume that the OU size is $2 \times 2$ and two column-vectors are assigned to one OU. The data path hardware in ReRAM-based accelerators will map the weight vectors onto 5 OUs. For example, $[1, 6]^T$ at (3, 1) in $F1$ and $[2, 3]^T$ at (3, 3) in $F3$ are assigned to OU1.

### C. Data Path for Column-Vector Based Pruning

Because we use semi-structural pruning in APQ, vectors of different filters may be placed to the same crossbar-column. As shown in Fig. 3, $[3, 3]^T$ of F3, $[3, 2]^T$ of F2, and $[1, 6]^T$ of F1 are placed to the first column in the crossbar, resulting in the dislocation problem [10]. In this section, we introduce a novel data path to solve the dislocation problem in ReRAM-based accelerators.

Fig. 4 shows the data path designed for the column-vector pruning algorithm used in the APQ framework. Because the crossbar arrays only store nontrivial weights, we only need to fetch the input activations corresponding to the weights. *Weight index buffers* store the index of column-vectors generated by Algorithm 2. In each cycle, control units fetch a few index tuples having the same $x$ coordinate from the weight index buffer. The number of tuples fetched should be smaller than or equal to the number of column vectors in one OU. *Input address generators* generate the address of an activation vector in the input feature map to be accessed by the crossbar given the $x$ coordinate of index tuples from the weight index buffer. Specifically, the buffer address of the activation vector is $g \times (x - 1) + 1$ where $g$ is the granularity of column-vectors. *Input registers* store the input to the crossbar array. To perform computation on OUs, *control units* issue commands to active corresponding wordlines and bitlines given the coordinates of column-vectors. Then we need to place the output of crossbars at a matching location in the output. For this purpose, we use *position mask generators* to produce position masks whose length is equal to the number of columns of the original weight matrix. The currents from crossbar arrays are converted by ADCs and then stored in *OU outputs*. Then *XB (crossbar) outputs* recover the final output by padding the value from the OU output with zeros according to the position mask. Then *adders* add the previous output from *intra-layer output* and the new value from XB output. Finally, they store the partial sum to the intra-layer output.

We continue to use the example in Fig. 3 for illustration. Given the output of Algorithm 2, the weight index buffer stores {(3, 1), (3, 3), (2, 2), (2, 5), (1, 3), (1, 4), (3, 4), (3, 6), (1, 5)}. In OU1, two vectors $[1, 6]^T$ and $[2, 3]^T$ with indexes (3, 1) and (3, 3) respectively (marked in the red square in Fig. 5(a)) are selected to perform computation. For the selected weights, the input address generator outputs buffer address 5 which is equal to $2 \times$ (3 -1)+ 1. Then given the address, [9, 10] in the feature map are fetched to the input register. The crossbar array then performs multiplication between [9, 10] and the selected weights in OU1 and then outputs [69, 48]. However, because $[1, 6]^T$ and $[2, 3]^T$ are from F1 and F3 respectively as shown in Fig. 3, we cannot store it in the intra-layer output directly. Instead, we need to find the corresponding positions of 69 and 48 in the XB output. The positions indicated by the position mask are $[1, 0, 1, 0, 0, 0]$ because the accumulated sum for Filter 1 and Filter 3 should be placed at positions 1 and 3 respectively. As a result, the adder adds $[69, 0, 48, 0, 0, 0]$ from the XB output and $[0, 0, 0, 0, 0, 0]$ from the intra-layer output and then sends the result $[69, 0, 48, 0, 0, 0]$ to the intra-layer output.

In OU2, two vectors $[3, 2]^T$ and $[4, 4]^T$ with indexes (2, 2) and (2, 5) respectively (marked in the red square in Fig. 5(b)) are selected to perform computation. For the selected weights, the input address generator outputs buffer address 3. Then given the address, [5, 6] in the feature map is fetched to the input register. The crossbar array then performs multiplication between [5, 6] and the selected weights in OU2 and outputs [27, 44]. However, because $[3, 2]^T$ and $[4, 4]^T$ are from F2 and F5 respectively as shown in Fig. 3, we need to find the corresponding positions of 27 and 44 in the XB output. The position mask is $[0, 1, 0, 0, 1, 0]$ because the accumulated sum for Filter 2 and Filter 5 should be placed at positions 2 and 5 respectively. As a

result, the adder adds $[0, 27, 0, 0, 44, 0]$ from the XB output and $[69, 0, 48, 0, 0, 0]$ from the intra-layer output and then sends the result $[69, 27, 48, 0, 44, 0]$ to the intra-layer output.

### D. DDPG Algorithm for Quantization

To further compress the DNN model, APQ trains another reinforcement learning agent to determine the optimal quantization policy for the final pruned matrices. It explores the search space layer by layer. In this section, we describe its *state space*, *action space*, *reward function*, and *agent*.

*State Space:* For each layer $k$, we use a 12-dimensional feature vector as our observation. Specifically, the state vector $S_k$ for layer $k$ is defined as

$$(k, t, inc, outc, ks, h, w, s, xb[k], xb_{saved}[k], xb_{rest}[k], b_{k-1}) \tag{8}$$

where all the features are defined in Table II. It is worth noting that for fully-connected layers their $inc$ and $outc$ are equal to the number of input and output neurons respectively. To make the reinforcement learning model effective for both convolutional layers and fully-connected layers, we set both $ks$ and $s$ to 1 for the fully-connected layers even though they do not have such attributes.

*Action Space:* The action space is formulated with the action which the agent makes at each time step. As the quantization decision is executed layer by layer, which may be time-consuming, we limit the bitwidth candidates for all layers with a $bound\_list$ using a profiling process to accelerate the quantization exploration. Each $bound\_list$ includes $N$ [$lbound$, $rbound$] vectors, where $N$ denotes the number of layers that the DNN model has, $lbound$ and $rbound$ represent the min bitwidth and max bitwidth of the corresponding DNN layer, respectively.

We shrink the bitwidth range for each layer because we observe that (1) bitwidths larger than a high threshold always bring such a high model accuracy that they are not necessary for model representation, and (2) bitwidths less than a low threshold usually render such a high accuracy loss that they are unacceptable. For example, for the first layer of VGG16 on CIFAR10, bitwidths larger than 12 always yield an accuracy loss below 0.75% and those less than 3 incur accuracy losses more than 5%, as shown in Fig. 6. We have similar observations for other layers. As a result, we set the $bound\_list$ for all layers as [[3, 12], [4, 12], [3, 12], [3, 12], [3, 12], [3, 12], [2, 12], [3, 12], [2, 12], [3, 12], [3, 12], [4, 12], [3, 11], [2, 12], [2, 12], [3, 12]].

Since a bitwidth should be an integer, we convert the continuous decision space into a discrete space to represent the range of available bitwidths. Specifically, at the $i_{th}$ time step, we get the action $b_i$ with a continuous range of [0,1], and then round it into the integer bitwidth $Q_i$ by using $Q_i = round(b_i \times (rbound - lbound + 1) + lbound - 0.5)$.

*Reward Function:* In the quantization decision process, the reward function of APQ aims to maximize the compression rate of crossbars while preserving high model accuracy, as shown in (9).

$$Reward = (acc_{reram} - acc_{ori}) \times \theta - \log(rate^{xb}_{compression}) \times \gamma \tag{9}$$

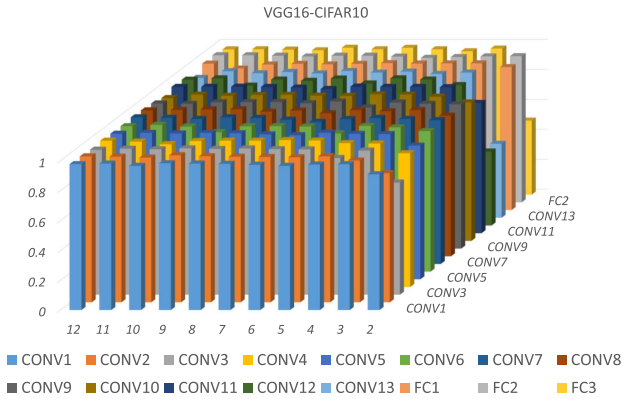



Fig. 6. Quantization bitwidth impact of each layer on the model accuracy for VGG16 on CIFAR10.

In the equation, $acc_{reram}$ and $rate^{xb}_{compression}$ are the same as those symbols in the pruning searching process. $\theta$ and $\gamma$ are the factors of bias between the compression rate of crossbars and the accuracy of quantized DNN models in the searching process. $acc_{ori}$ is the accuracy on ReRAM-based accelerator without quantization.

*Agent:* As for the RL agent, we leverage DDPG , which is an actor-critic algorithm for continuous control problems. The actor network is fed with the state embedding to make an action. The critic network takes the last action and current state embedding as input and computes the Q-function to get the maximal Q-value (i.e., the minimal loss).

First, for each time step, the transition tuple $(S_t, b_t, r, S_{t+1})$ is put into the replay buffer. In the transition tuple, $S_t$ is the state embedding at the $t_{th}$ time step and $S_{t+1}$ is of the next time step. $b_t$ is the action and $r$ is the reward of this action.

Next, $N$ transition tuples are sampled in the history buffer. The actor computes the policy gradient and the critic updates the loss.

Finally, the actor network and the critic network are updated according to the policy gradient and loss, respectively. Specifically, the loss function is defined as:

$$Q'_t = r_t - \varepsilon + \xi \times Q(S_{t+1}, A(S_{t+1}|\theta^q)) \tag{10}$$

$$Loss = \frac{1}{N}\sum_i (Q'_t - Q(S_t, b_t|\theta^f))^2 \tag{11}$$

where $Q'_t$ is the Q-value at the $t_{th}$ time step, $\varepsilon$ is an exponential moving average of all previous rewards to reduce the variance of the gradient estimation, and $\xi$ is the discount factor which we set to 1 with the assumption that the action made for each layer should contribute equally to the reward. $Q(S_t, b_t|\theta^f)$ is the critic network with parameter $\theta^f$. $N$ is the number of sampled data from the replay buffer.

### E. Data Path Design for Weight Quantization

To enable weight quantization, APQ needs to map weights in different layers to crossbars with different bitwidths. The weights in the same layer use the same bitwidth. There are two

TABLE III
STRUCTURE OF NEURAL NETWORKS

| Network | Structure |
|---|---|
| AlexNet | $C3$-64, $C3$-192, $C3$-384, $2 \times C3$-256, F4096, F4096, F10 |
| VGG16 | $2 \times C3$-64, $2 \times C3$-128, $3 \times C3$-256,$3 \times C3$-512, $3 \times C3$-512, F4096, F1000, F10 |
| Plain20 | $7 \times C3$-16, $6 \times C3$-32, $6 \times C3$-64, F10 |

The symbol $a \times Cb$-$c$ represents that $a$ convolutional layers with $b \times b$-sized kernels and $C$ output channels. The symbol $Fd$ means a fully-connected layer with $d$ neurons.

kinds of methods that can be adopted to map weights. The first one is to map each quantized weight to the continuous ReRAM cells on the same crossbar [3], [4] and we then merge the results of adjacent columns to get the final results. As pointed out by [36], the peripheral circuits connected to each bit line need to be *customized* to support shift-and-add (S&A) operations for different bitwidths, thus increasing the overhead of peripheral circuits and lacking reconfigurability. Another recent approach is to use multiple crossbars to store the quantized weight [36]. Each crossbar only stores one bit of the whole bitwidth. Since it does not need to customize and reconfigure the peripheral circuits of crossbars (e.g., S&As and ADCs), we adopt this approach to support the proposed weight quantization in APQ.

## IV. EVALUATION

### A. Experimental Setup

*Experimental Platform:* We implement the proposed compression and mapping framework for ReRAM-based accelerators in Python. We use a ReRAM simulator in the experiments because commercial ReRAM devices are unavailable to us. We use MNSIM [36] as the simulator of ReRAM-based accelerators owing to its efficiency, flexibility, and simplicity. To support mixed-precision quantization, MNSIM stores multi-bit weights in multiple crossbars, improving the flexibility of computing units while reducing the overhead of peripheral modules [36].

We use the hardware model in MNSIM to evaluate the area and energy consumption of crossbars and other modules (including DAC, ADC, IR, OR, and other digital parts). In the setting of the model, each memristor cell stores one bit, and both ADC and DAC are set to be 1 bit. By default, each weight of the DNN models uses 9-bit quantization, which means 9 crossbars are needed to represent a weight. We set the crossbar size as $128 \times 128$ and the granularity of column-vector ($g$) as 32. The operation unit (OU) size is set to $g \times g$. All other configurations are the same as the default ones used in MNSIM.

*Workloads and Datasets:* We evaluate APQ with three DNN models, including AlexNet [19], VGG16 [20], and Plain20 [21]. Table III shows the structures of these DNN models. We execute the inference of the models on two datasets, i.e., CIFAR10 [22] and MNIST [23]. The CIFAR10 dataset consists of 60, 000 colorful images, each with $32 \times 32 \times 3$ pixels. There are 50, 000 images for training and 10, 000 images for testing. The MNIST dataset consists of 70, 000 grayscale $28 \times 28 \times 1$ images. It includes 60, 000 images for training and 10, 000 images for testing.

TABLE IV
CROSSBAR COMPRESSION RATE (CR) COMPARISON OF DIFFERENT FRAMEWORKS ON CIFAR10

| Network | Method | CR | Acc1 | Acc Drop |
|---|---|---|---|---|
| AlexNet | Naive | 1 | 81.58% | |
| | PIM-Prune | 4.3 | 76.76% | 4.82% |
| | Pattern-Prune | 1.1 | 56.81% | 24.77% |
| | **Auto-Prune** | 14.3 | 81.14% | 0.44% |
| | **APQ** | **23.6** | **80.62%** | **0.96%** |
| VGG16 | Naive | 1 | 81.61% | |
| | PIM-Prune | 6.1 | 78.81% | 2.8% |
| | Pattern-Prune | 2.6 | 77.52% | 4.09% |
| | **Auto-Prune** | 11.9 | 81.43% | 0.18% |
| | **APQ** | **13.9** | **81.20%** | **0.41%** |
| Plain20 | Naive | 1 | 74.15% | |
| | PIM-Prune | 7.3 | 60.50% | 13.65% |
| | Pattern-Prune | 1.2 | 66.71% | 7.44% |
| | **Auto-Prune** | 10.3 | 73.57% | 0.58% |
| | **APQ** | **14.6** | **73.29%** | **0.86%** |

*Compared Systems:* We compare APQ with the state-of-the-art compression and mapping frameworks for ReRAM architectures: PIM-Prune [10], Pattern-Prune [11], and AUTO-PRUNE [18] (Our conference version). We do not compare APQ with SRE [7] because SRE has not shown the inference accuracy of the pruned networks in the paper, making the comparison unfair as APQ provides both low accuracy losses and high compression rates. Neither do we evaluate Lin et al. [24] and XCS [14] because the evaluation results of PIM-Prune [10] have shown PIM-Prune significantly outperforms them, making the comparisons redundant.

As PIM-Prune and Pattern-Prune are not open-source, we re-implement them as faithfully as possible according to the descriptions in their papers respectively. Although PIM-Prune supports three pruning methods (i.e., SC+XRS, SR+XCS, and block-based pruning), we only implement the block-based pruning method because it exploits the fine-grained block-level sparsity in both directions (row and column) while the other two methods can only exploit the sparsity in one direction. PIM-Prune prunes the weights in each block within a fixed compression rate, which may lead to sub-optimal pruning performance. Pattern-Prune uses the pattern pruning algorithm [12] to remove unimportant weights and then applies kernel-reordering methods to map the pruned weight matrices to crossbar arrays. Pattern-Prune only focuses on convolutional layers which usually have a large number of patterns. It fails to utilize the sparsity in fully-connected layers. Consequently, the compression rate of the whole network models may be very low using Pattern-Prune. AUTO-PRUNE only considers the weight pruning policy but loses the potential of weight quantization. Furthermore, we compare APQ to the original system (Naive) where DNN models are not compressed and directly mapped to ReRAM crossbar arrays.

### B. General Results

*Compression Rate:* Table IV shows the compression rate (CR) comparison of different compression and mapping methods on

TABLE V
COMPRESSION RATE COMPARISON OF DIFFERENT FRAMEWORKS ON MNIST

| Network | Method | CR | Acc1 | Acc Drop |
|---|---|---|---|---|
| AlexNet | Naive | 1 | 98.89% | |
| | PIM-Prune | 13.6 | 98.41% | 0.48% |
| | Pattern-Prune | 1.1 | 97.78% | 1.11% |
| | **Auto-Prune** | **21.4** | **98.49%** | **0.40%** |
| | **APQ** | **32.2** | **98.10%** | **0.79%** |
| VGG16 | Naive | 1 | 98.67% | |
| | PIM-Prune | 13.3 | 98.56% | 0.11% |
| | Pattern-Prune | 2.8 | 98.34% | 0.33% |
| | **Auto-Prune** | **16.0** | **97.90** | **0.77%** |
| | **APQ** | **17.5** | **99.24%** | **-0.57%** |
| Plain20 | Naive | 1 | 98.13% | |
| | PIM-Prune | 5.0 | 97.91% | 0.22% |
| | Pattern-Prune | 1.2 | 97.42% | 0.71% |
| | **Auto-Prune** | **6.2** | **98%** | **0.13%** |
| | **APQ** | **9.9** | **98.29%** | **-0.16%** |

the CIFAR10 dataset. The CR is defined as the rate of the number of occupied crossbars (XBs) before compression to that after compression. We can see that APQ achieves the highest compression rate among the five methods within the accuracy drop of 1%. More specifically, compared to Naive, APQ can get 23.6X, 13.9X, and 14.6X compression rates for AlexNet, VGG16, and Plain20, respectively, with 0.96%, 0.41%, and 0.86% accuracy drops. Its compression rate is up to 4.9X, 20.5X, and 0.7X higher than that of PIM-Prune, Pattern-Prune, and AUTO-PRUNE respectively. AUTO-PRUNE has the best pruning performance excluding APQ because of its automated pruning method using reinforcement learning and a fine-grained mapping mechanism. APQ has the best compression performance because it further utilizes quantization to compress the model besides pruning.

We also notice that Pattern-Prune has the lowest compression rate. This is because Pattern-Prune is efficient only for convolutional layers but inefficient for fully-connected layers, which have high weight sparsity. In contrast, PIM-Prune, AUTO-PRUNE, and APQ work effectively for all the layers of the networks. We also observe that AlexNet has the highest compression rate among the networks. The reason is that AlexNet has 5% and 9% higher weight sparsity than VGG16 and Plain20 respectively in our experimental results.

Table V shows the compression rates of different compression methods on the MNIST dataset. APQ also outperforms PIM-Prune, Pattern-Prune, and AUTO-PRUNE. Compared to Naive, APQ achieves 32.2X, 17.5X and 9.9X compression rates with 0.79%, -0.57%, and -0.16% accuracy drops for AlexNet, VGG16, and Plain20 respectively, while PIM-Prune, Pattern-Prune, and AUTO-PRUNE achieve up to 13.6X, 2.8X, and 21.4X compression rates with at least 0.11%, 0.33%, and 0.13% accuracy drops. APQ has the highest compression rates among all approaches because of its automated pruning policy, fine-grained mapping mechanism, and joint compression with quantization. We observe that the compression rates of APQ are up to 36% higher on MNIST than those on CIFAR10 for AlexNet and VGG16. This is because the images in the MNIST dataset are easier to be classified than those in the CIFAR10 dataset. The trained networks using the MNIST dataset require

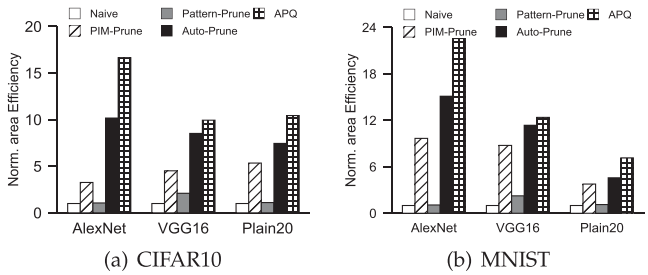



Fig. 7. Results of area efficiency on different datasets.

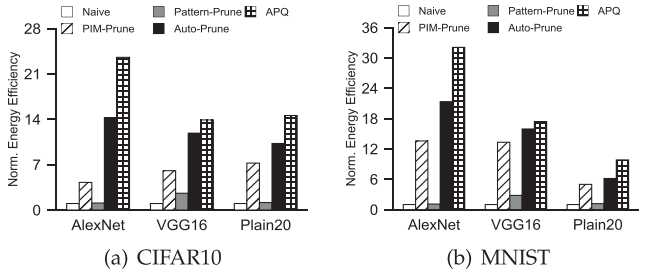



Fig. 8. Results of energy efficiency on different datasets.

a fewer number of parameters to achieve similar accuracy. Consequently, more network weights can be pruned on MNIST. We also notice that the compression rates of Plain20 on MNIST are lower than those on CIFAR10. This is because (1) we choose Acc1 on MNIST which is more sensitive to pruning than Acc5 and (2) the accuracy of Plain20 is more sensitive to pruning than the other two networks.

*Area Efficiency:* Fig. 7(a) shows the crossbar area efficiency of different methods on CIFAR10. The area efficiency is normalized to that of the network without pruning (Naive). From Fig. 7(a), we can observe that APQ achieves the best area efficiency among the five schemes for all three networks. More specifically, APQ improves the area efficiency by 16.7X, 9.9X, and 10.4X on AlexNet, VGG16, and Plain20, respectively, while PIM-Prune, Pattern-Prune, and AUTO-PRUNE can improve by up to 5.3X, 2.1X, and 10.2X. These results demonstrate that our proposed method is beneficial to improving the area efficiency using joint pruning and quantization.

Fig. 7(b) shows the results of ReRAM crossbar area efficiency on MNIST. APQ improves the area efficiency by 22.6X, 12.4X, and 7.1X for AlexNet, VGG16, and Plain20, respectively. The area efficiency is up to 2.3X, 21.1X, and 1.6X higher than those of PIM-Prune, Pattern-Prune, and AUTO-PRUNE, respectively.

*Energy Efficiency:* Fig. 8 shows the energy efficiency comparison among the compression schemes for the networks on CIFAR10 and MNIST. The energy efficiency is normalized to that of Naive. We can observe that APQ achieves the best energy efficiency among the five schemes for all three networks. More specifically, APQ improves the area efficiency against Naive by 23.6X, 13.9X, and 14.6X on AlexNet, VGG16, and Plain20, respectively. Compared to PIM-Prune, Pattern-Prune, and AUTO-PRUNE, APQ achieves up to 5.5X, 21.8X, and 1.7X

TABLE VI
NUMBER OF OCCUPIED CROSSBARS FOR EACH LAYER OF ALEXNET ON CIFAR10

| Method | # of occupied crossbars | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | $L_1$ | $L_2$ | $L_3$ | $L_4$ | $L_5$ | $L_6$ | $L_7$ | $L_8$ | Total |
| Naive | 8 | 80 | 336 | 432 | 288 | 2048 | 8192 | 256 | 11640 |
| PIM-Prune | 8 | 24 | 112 | 104 | 72 | 480 | 1800 | 120 | 2720 |
| Pattern-Prune | conv:280 | | | | | fc:10496 | | | 10776 |
| Auto-Prune | 8 | 40 | 208 | 216 | 120 | 72 | 80 | 72 | 816 |
| APQ | 12 | 30 | 130 | 135 | 60 | 63 | 70 | 54 | 554 |

$L_i$ means layer $i$ of ALEXNET.

TABLE VII
NUMBER OF OCCUPIED CROSSBARS FOR EACH LAYER OF ALEXNET ON MNIST

| Method | # of occupied crossbars | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | $L_1$ | $L_2$ | $L_3$ | $L_4$ | $L_5$ | $L_6$ | $L_7$ | $L_8$ | Total |
| Naive | 8 | 80 | 336 | 432 | 288 | 2048 | 8192 | 256 | 11640 |
| PIM-Prune | 8 | 16 | 32 | 56 | 40 | 128 | 512 | 64 | 856 |
| Pattern-Prune | conv:80 | | | | | fc:10496 | | | 10576 |
| Auto-Prune | 8 | 8 | 24 | 16 | 16 | 96 | 360 | 16 | 544 |
| APQ | 11 | 8 | 21 | 10 | 10 | 36 | 315 | 10 | 421 |

$L_i$ means layer $i$ of ALEXNET.

higher energy efficiency on the CIFAR10 dataset. For MNIST, APQ achieves 32.1X, 17.4X, and 8.3X energy efficiency improvement for AlexNet, VGG16, and Plain20 respectively. We also observe that AUTO-PRUNE is more energy efficient than PIM-Prune and Pattern-Prune because AUTO-PRUNE requires up to 95% fewer ReRAM crossbar arrays, making it more energy efficient since fewer bitlines, wordlines, ADCs, and DACs are activated. However, because APQ uses the automated quantization technique to further reduce crossbars and other peripheral circuits, it is more energy efficient than AUTO-PRUNE.

### C. Occupied Crossbar Analysis

Tables VI and VII show the number of occupied crossbars for each layer of the networks on CIFAR10 and MNIST with different compression schemes. Due to the space limitation, we only show the results of AlexNet. We have similar observations on other networks. AlexNet has eight layers, where the first five layers are convolutional layers and the last three are fully-connected layers. Because Pattern-Prune may map multiple layers to one crossbar, we cannot compute the number of crossbars for each layer individually. As a result, for Pattern-Prune, we only record the total numbers of occupied crossbars for all convolutional layers and all fully-connected layers respectively.

Table VI shows that different pruning schemes lead to different numbers of occupied crossbars for each layer of AlexNet on CIFAR10. However, APQ always occupies the fewest crossbars for the whole network. Compared to Naive, APQ reduces the total number of occupied crossbars for by 95%, while PIM-Prune, Pattern-Prune, and AUTO-PRUNE only reduce it by 77%, 7%, and 93% respectively. Reducing the number of occupied crossbars results in 15.7X and 21.8X higher area efficiency and energy efficiency of ReRAM-based accelerators as shown in Figs. 7(a) and 8(a).

We also observe that AUTO-PRUNE performs 70% better than PIM-Prune, especially for the fully-connected layers (i.e., $L_6$, $L_7$, and $L_8$). This is because the fully-connected layers have higher sparsity (as shown in Fig. 11 in Section IV-E) than other layers. The RL agent prunes more unimportant weights in the fully-connected layers with a higher pruning rate to achieve a global optimum. In contrast, PIM-Prune uses a fixed pruning rate for all layers without considering the sparsity variation of individual layers, resulting in more occupied crossbar arrays to realize the similar accuracy. As APQ inherits the advantage of AUTO-PRUNE and its RL-based quantization technique can further reduce crossbars, APQ performs better than AUTO-PRUNE. Furthermore, we find that Pattern-Prune is inefficient for fully-connected layers because it cannot reduce the number of crossbars for such layers.

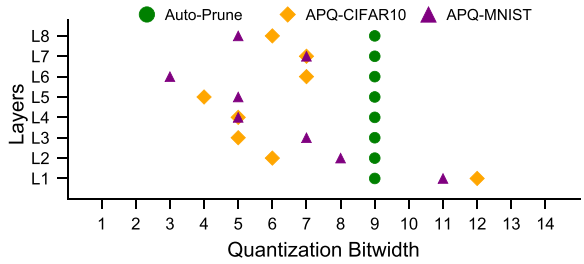



Fig. 9. Quantization policies for each layer of AlexNet with different methods on CIFAR10 and MNIST. The $x$-axis denotes the quantization bitwidth and $y$-axis represents all the layers in the model. The orange diamond and the purple triangle denote the bitwidths of a layer under APQ on CIFAR10 and MNIST, respectively. The green circle represents the quantization policy of AUTO-PRUNE.

Table VII shows that APQ has better performance than other schemes on the MNIST dataset. Specifically, it reduces the total number of occupied crossbars by 96% compared to Naive, while PIM-Prune, Pattern-Prune, and AUTO-PRUNE only reduce it by 93%, 9%, and 95% respectively.

We also note that all compression schemes occupy fewer crossbars on MNIST than those on CIFAR10 for the same network. For example, the total number of occupied crossbars for APQ reduces from 554 to 421 when the dataset is switched from CIFAR10 to MNIST. This is because MNIST is easier to be classified than CIFAR10, thus APQ identifies more trivial elements for pruning in weight matrices on MNIST. In addition, we observe that APQ usually requires more crossbars for the first layer of the network compared to AUTO-PRUNE. This is because the weight matrix of the first layer usually retrieves more non-trivial input information, which contributes more to the model accuracy. Therefore, APQ quantizes them with a higher bitwidth to reduce the overall quantization bitwidths globally.

### D. Quantization Bitwidth Analysis

To understand the behavior of the automated quantization in APQ, Fig. 9 shows the quantized bitwidths for each layer of AlexNet under APQ and AUTO-PRUNE on CIFAR10 and MNIST. Due to the space limitation, we exclude the results of VGG16 and Plain20. AUTO-PRUNE maps the elements for all the layers to the ReRAM-based accelerator using a fixed 9-bit width for quantization on both datasets, as the default setting in the MNSIM simulator. In contrast, APQ uses an automated

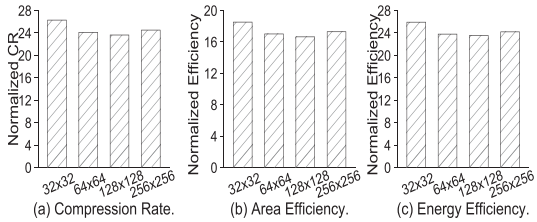



Fig. 10. Compression rate, area efficiency and energy efficiency for AlexNet on CIFAR10 with various crossbar sizes.

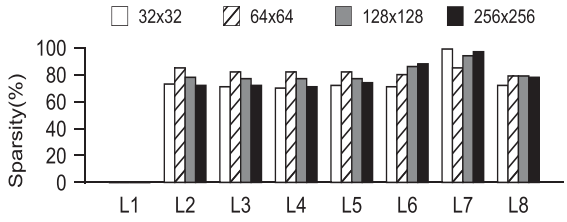



Fig. 11. Layer-by-layer sparsity of AlexNet with various crossbar sizes after pruning.

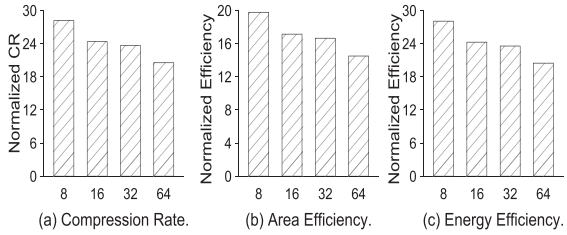



Fig. 12. Compression rate, area efficiency, and energy efficiency for AlexNet with various granularities of column-vectors.

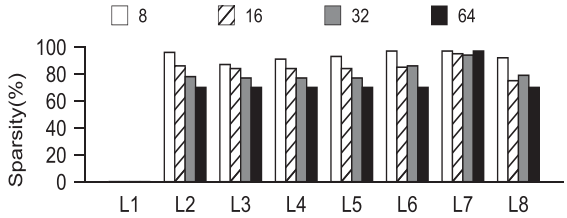



Fig. 13. Layer-by-layer sparsity of AlexNet with various granularities of column-vectors after pruning.

mixed bitwidth policy for quantization. We have the following three observations.

First, APQ usually assigns the quantization bitwidths less than the default bitwidth, i.e., 9, to most of the layers of the models. For example, the bitwidths of all layers except layer 1 of AlexNet are between 3 and 8 on CIFAR10 and MNIST. Compared to the fixed policy in AUTO-PRUNE, these fewer quantization bitwidths lead to fewer occupied crossbars, as shown in Tables VI and VII.

Second, APQ may assign a bitwidth larger than the default bitwidth to some layers of the models, especially the first several layers. This is because the weight matrices of these first layers can retrieve more important information, thus representing them with higher bitwidths not only yields a positive effect on model accuracy but also brings the opportunity to compress the weight matrices of the rest of layers in a more aggressive manner to further compress the whole model.

### *E. Sensitivity Studies*

*Crossbar Size:* Fig. 10 shows the normalized compression rate (CR), area efficiency, and energy efficiency of APQ with various crossbar sizes. We only show the results of AlexNet with CIFAR10 because other models on CIFAR10 or MNIST show similar trends. For all the four crossbar sizes, the normalized compression rates are consistently larger than 23 with less than 1% accuracy drops. Further, as shown in Fig. 10(a), APQ achieves the highest CR with the crossbar size $32 \times 32$. This is because a small crossbar size usually makes a high crossbar utilization, thus requiring less physical memory capacity for the pruned model. The reduced crossbar resource also improves the area/energy efficiency, as shown in Fig. 10(b) and (c). However, the CR is affected by multiple factors, including the crossbar size, the model sparsity, the dataset type, the compression algorithm, and the mapping scheme. Therefore, the CR varies non-linearly as we increase the crossbar size from $32 \times 32$ to $256 \times 256$. So do the area efficiency and energy efficiency.

To understand the characteristics of APQ with different crossbar sizes, Fig. 11 shows the layer sparsity of AlexNet. For the same crossbar size, we can see that $L_6$, $L_7$, and $L_8$ have higher sparsity on average compared to the other layers, meaning that the fully-connected layers are more sparse and prone to be pruned. For the same layer in AlexNet, the pruning rates decided by the reinforcement learning algorithm are also varied with different crossbar sizes. To preserve the features in the original input feature map as much as possible, APQ is designed not to prune the first layer. As a result, the sparsity of $L_1$ equals zero in Fig. 11.

*The Pruning Granularity:* Fig. 12 shows the impact of pruning granularity on system performance. As the granularity of column-vector increases, the compression rate decreases. So do the area efficiency and energy efficiency. When the granularity of column-vectors is 8, APQ achieves the highest compression rate. This is because using finer-grained column-vectors helps exploit the sparsity of weight matrices, improving the chances of pruning unimportant weights. However, the fine granularity requires a large index structure, incurring high indexing overhead. In this paper, we choose 32 as the granularity of column-vectors in order to balance performance benefits and hardware overhead.

To understand the characteristics of APQ with various pruning granularities, Fig. 13 shows the layer sparsity of AlexNet on CIFAR10 when the granularity increases from 8 to 64. For each layer except $L_1$ and $L_7$, as the granularity increases, the layer sparsity decreases. When the granularity of the column-vector is 8, the sparsity of each layer is above 87% and achieves the highest.

### *F. Other Tests*

*Reward Function:* The reward functions in (2) and (9) are magic functions. In principle, the reward should be increased with a higher model accuracy and a larger compression rate. We normalize the value of compression rate in the functions for

TABLE VIII
IMPACT OF DIFFERENT REWARD FUNCTIONS

| Reward Function | Acc. | CR |
|---|---|---|
| $(acc_{reram} - acc_{ori}) \times \theta - \log(rate^{xb}_{compression}) \times \gamma$ | 98.1% | 32.2 |
| $(acc_{reram} - acc_{ori}) + rate^{xb}_{compression}$ | 97.8% | 50.2 |
| $100 \times (acc_{reram} - acc_{ori}) + 0.01 \times rate^{xb}_{compression}$ | 97.4% | 27.9 |

TABLE IX
COMPARASION OF RL WITH OTHER SEARCH METHODS

| Model | EA | | BO | | RL | |
|---|---|---|---|---|---|---|
| | Acc. | CR | Acc. | CR | Acc. | CR |
| VGG16 | 98.3% | 8.0 | 96.0% | 12.3 | 97.9% | 16 |
| AlexNet | 98.0% | 20.2 | 98.3% | 17.5 | 98.49% | 21.4 |
| Plain20 | 97.8% | 9.1 | 96.6% | 10.1 | 98% | 6.2 |

the convenience of algorithmic design. We empirically choose the reward functions used in (2) and (9). To show the efficiency of our proposed reward functions, we added an experiment to compare three possible functions listed in Table VIII for (9). As Table VIII shows, the first one (adopted in our design) provides the best tradeoff between accuracy and compression rate when training AlexNet on MNIST.

*Why Choose RL:* We choose RL rather than evolutionary algorithm (EA) and Bayesian optimization (BO) for two reasons. First, RL usually requires a relatively smaller search space (i.e., decision time). This is because RL makes an action according to the state of one layer each time while EA and BO make an action for all the layers in one epoch. For example, when training VGG16, the search space in one epoch is $100 \times 11$ in RL and $100^{11}$ in EA and BO. Second, RL achieves better performance as it can accurately perceive the state of the environment while EA and BO cannot do this. To verify this, we added new experiments to compare RL with an EA [37] and an BO algorithm [38]. As Table IX shows, RL achieves a higher compression rate with less accuracy loss compared with EA and BO in most of the cases.

*The Training Time for RL:* The RL training is an offline process. The training time can be well amortized over the usage of the compressed models. The actual training time varies with different models and datasets. For example, for AlexNet and Plain20, the training times are 7.7 and 11.3 hours, respectively.

*Scalability Discussion:* We also try to evaluate APQ on large models and datasets. However, APQ may not scale very efficiently on larger problems. For example, the RL algorithm with ResNet152 model on ImageNet dataset does not converge even after searching for 10 days. This is because the existing PIM simulators (e.g., NeuroSim, MNISM) take about 10 minutes to evaluate the performance of a single NN model [39] and the large model on a large dataset requires more epochs to converge. However, with the emergence of real ReRAM-based hardware in the near future, we can evaluate the compression decision on the real hardware instead of a simulator to significantly reduce the search time. While APQ provides weaker scalability in performance for large problems than existing heuristic approaches, it provides a better crossbar compression rate and accuracy loss because of the inherent nature of existing rule-based heuristic approaches.

## V. CONCLUSION

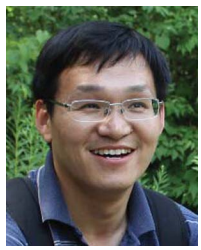
In this paper, we propose an Automated DNN Pruning and Quantization framework, APQ, for ReRAM-based accelerators. First, APQ leverages reinforcement learning to automatically determine a global optimum pruning policy given the constraint of accuracy loss. Second, it prunes weight matrices in a column-vector finer granularity and maps the nontrivial weights to crossbars. To enable the fine-grained pruning, it devises a new data path to correctly index and feed input to matrix-vector computation. Third, it leverages another reinforcement learning scheme to automatically determine the quantization bitwidth of each layer based on the result of DNN pruning. Experimental results show that APQ achieves up to 4.52X compression rate, 4.11X area efficiency, and 4.51X energy efficiency compared to PIM-Prune while maintaining a similar or even higher accuracy.

## REFERENCES

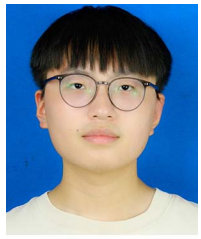
[1] Y. Chen et al., "DaDianNao: A machine-learning supercomputer," in *Proc. IEEE/ACM 47th Annu. Int. Symp. Microarchitecture*, 2014, pp. 609–622.
[2] S. Han et al., "EIE: Efficient inference engine on compressed deep neural network," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 243–254, 2016.
[3] 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.
[4] A. Shafiee et al., "ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 14–26, 2016.
[5] S. Mittal, "A survey of ReRAM-based architectures for processing-in-memory and neural networks," *Mach. Learn. Knowl. Extraction*, vol. 1, no. 1, pp. 75–114, 2019.
[6] L. Song, X. Qian, H. Li, and Y. Chen, "PipeLayer: A pipelined ReRAM-based accelerator for deep learning," in *Proc. IEEE Int. Symp. High Perform. Comput. Architecture*, 2017, pp. 541–552.
[7] 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. Architecture*, 2019, pp. 236–249.
[8] 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*.
[9] M. Denil, B. Shakibi, L. Dinh, M. Ranzato, and N. de Freitas, "Predicting parameters in deep learning," in *Proc. 26th Int. Conf. Neural Inf. Process. Syst.*, 2013, pp. 2148–2156.
[10] C. Chu et al., "PIM-Prune: Fine-grain DCNN pruning for crossbar-based process-in-memory architecture," in *Proc. ACM/IEEE 57th Des. Automat. Conf.*, 2020, pp. 1–6.
[11] S. Yu et al., "High area/energy efficiency RRAM CNN accelerator with kernel-reordering weight mapping scheme based on pattern pruning," 2020, *arXiv: 2010.06156*.
[12] J. Wang et al., "High PE utilization CNN accelerator with channel fusion supporting pattern-compressed sparse neural networks," in *Proc. 57th ACM/IEEE Des. Automat. Conf.*, 2020, pp. 1–6.
[13] 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 ACM/ESDA/IEEE Des. Automat. Conf.*, 2018, pp. 1–6.
[14] L. Liang et al., "Crossbar-aware neural network pruning," *IEEE Access*, vol. 6, pp. 58324–58337, 2018.
[15] A. Mishra and D. Marr, "Apprentice: Using knowledge distillation techniques to improve low-precision network accuracy," 2017, *arXiv: 1711.05852*.
[16] C. Baskin et al., "UNIQ: Uniform noise injection for non-uniform quantization of neural networks," *ACM Trans. Comput. Syst.*, vol. 37, no. 1–4, pp. 1–15, 2021.
[17] S. Qu, B. Li, Y. Wang, D. Xu, X. Zhao, and L. Zhang, "RaQu: An automatic high-utilization CNN quantization and mapping framework for general-purpose RRAM accelerator," in *Proc. ACM/IEEE 57th Des. Automat. Conf.*, 2020, pp. 1–6.

[18] S. Yang, W. Chen, X. Zhang, S. He, Y. Yin, and X.-H. Sun, "AUTO-PRUNE: Automated DNN pruning and mapping for ReRAM-based accelerator," in *Proc. ACM Int. Conf. Supercomputing*, 2021, pp. 304–315.
[19] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, pp. 84–90, 2017.
[20] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*.
[21] Y. He et al., "AMC: Automl for model compression and acceleration on mobile devices," in *Proc. Eur. Conf. Comput. Vis.*, 2018, pp. 784–800.
[22] A. Krizhevsky, "Learning multiple layers of features from tiny images," 2009.
[23] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.
[24] 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. Automat. Conf.*, 2019, pp. 639–644.
[25] H. Ji, L. Song, L. Jiang, H. Li, and Y. Chen, "ReCom: An efficient resistive accelerator for compressed deep neural networks," in *Proc. Des. Automat. Test Europe Conf. Exhib.*, 2018, pp. 237–240.
[26] 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*.
[27] 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.*, 2018, pp. 365–382.
[28] Y. Zhou, S.-M. Moosavi-Dezfooli, N.-M. Cheung, and P. Frossard, "Adaptive quantization for deep neural network," in *Proc. AAAI Conf. Artif. Intell.*, 2018, pp. 4596–4604.
[29] J. Choi, S. Venkataramani, V.V. Srinivasan, K. Gopalakrishnan, Z. Wang, and P. Chuang, "Accurate and efficient 2-bit quantized neural networks," in *Proc. Mach. Learn. Syst.*, vol. 1, pp. 348–359, 2019.
[30] Y. Long, E. Lee, D. Kim, and S. Mukhopadhyay, "Q-PIM: A genetic algorithm based flexible DNN quantization method and application to processing-in-memory platform," in *Proc. ACM/IEEE 57th Des. Automat. Conf.*, 2020, pp. 1–6.
[31] S. Huang et al., "Mixed precision quantization for ReRAM-based DNN inference accelerators," in *Proc. 26th Asia South Pacific Des. Automat. Conf.*, 2021, pp. 372–377.
[32] Q. Yao et al., "Taking human out of learning applications: A survey on automated machine learning," 2018, *arXiv: 1810.13306*.
[33] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "HAQ: Hardware-aware automated quantization with mixed precision," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2019, pp. 8612–8620.
[34] T. P. Lillicrap et al., "Continuous control with deep reinforcement learning," 2015, *arXiv:1509.02971*.
[35] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *Proc. 31st Int. Conf. Int. Conf. Mach. Learn.*, 2014, pp. I-387–I-395.
[36] Z. Zhu et al., "A configurable multi-precision CNN computing framework based on single bit RRAM," in *Proc. ACM/IEEE 56th Des. Automat. Conf.*, 2019, pp. 1–6.
[37] S. Mirjalili and S. Mirjalili, "Genetic algorithm," in *Evolutionary Algorithms and Neural Networks: Theory and Applications*. Berlin, Germany: Springer, 2019, pp. 43–55.
[38] P. I. Frazier, "A tutorial on Bayesian optimization," 2018, *arXiv: 1807.02811*.
[39] H. Sun et al., "Gibbon: Efficient co-exploration of NN model and processing-in-memory architecture," in *Proc. Des. Automat. Test Europe Conf. Exhib.*, 2022, pp. 867–872.

**Siling Yang** is currently working toward the PhD degree with the College of Computer Science and Technology, Zhejiang University, China. Her research focuses on intelligent computing and processing-in-memory.

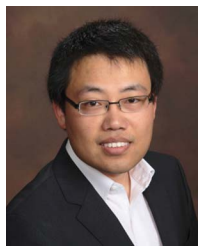


**Shuibing He** (Member, IEEE) received the PhD degree in computer science and technology from the Huazhong University of Science and Technology, in 2009. He is now a ZJU100 Young professor with the College of Computer Science and Technology, Zhejiang University, China. His research areas include intelligent computing, memory and storage systems, processing-in-memory. He is a member of ACM.



**Hexiao Duan** is currently working toward the master degree with the College of Computer Science and Technology, Zhejiang University. His research interests include neural network acceleration and processing in memory.

**Weijian Chen** is currently working toward the PhD degree with the College of Computer Science and Technology, Zhejiang University, China. His research focuses on intelligent computing, memory, and storage systems.



**Xuechen Zhang** (Member, IEEE) received the MS and PhD degrees in computer engineering from Wayne State University. He is currently an associate professor with the School of Engineering and Computer Science, Washington State University Vancouver. His research interests include the areas of storage systems and high-performance computing. He is a member of ACM.

**Tong Wu** is currently working toward the PhD degree with the College of Computer Science and Technology, Zhejiang University, China. Her research focuses on intelligent computing and processing-in-memory.

**Yanlong Yin** received the PhD degree in computer science from the Illinois Institute of Technology, in 2014. He is now an adjunct professor with Zhejiang University, China. His research interests include intelligent computing, parallel computing, and parallel file systems.