Enabling Deep Neural Networks with Oversized Working Memory on Resource-Constrained MCUs

by

Zhepeng Wang

Bachelor of Engineering, Harbin Institute of Technology, 2018

Submitted to the Graduate Faculty of

the Swanson School of Engineering in partial fulfillment

of the requirements for the degree of

Master of Science

University of Pittsburgh

2021

UNIVERSITY OF PITTSBURGH

SWANSON SCHOOL OF ENGINEERING

This thesis was presented

by

Zhepeng Wang

It was defended on

July 12, 2021

and approved by

Jingtong Hu, PhD, Associate Professor, Department of Electrical and Computer Engineering

Zhi-Hong Mao, PhD, Professor, Department of Electrical and Computer Engineering

Samuel Dickerson, PhD, Associate Professor, Department of Electrical and Computer Engineering

Thesis Advisor: Jingtong Hu, PhD, Associate Professor, Department of Electrical and Computer Engineering

Enabling Deep Neural Networks with Oversized Working Memory on Resource-Constrained MCUs

Zhepeng Wang, M.S.

University of Pittsburgh, 2021

Deep neural networks (DNNs) have shown their great power in effectively extracting features and making predictions from noisy input data, which makes them the most widely used algorithm in artificial intelligence (AI) applications. In the meantime, microcontroller units (MCUs) have become the most common processors in our daily life. Therefore, integrating DNNs into MCUs will definitely make a huge impact on the real world. Despite its importance, little attention has been paid to the deployment of DNNs onto MCUs yet. DNNs are usually resource-intensive while MCUs are resource-constrained, which often makes it infeasible to directly run DNNs on MCUs. Apart from the low frequency (1-16 MHz) and limited storage (e.g., 64KB to 256KB ROM), one of the biggest challenges is the small RAM size (e.g., 2KB to 16KB), which is needed to save the intermediate feature maps of a DNN in the runtime. Most existing DNN compression algorithms aim to reduce the model size so that the model can fit into limited storage. However, these algorithms do not reduce the size of intermediate feature maps significantly, which is referred to as working memory and might exceed the capacity of RAM. Therefore, it is possible that DNNs cannot run on MCUs even after compression. To address this problem, this work proposes a technique to dynamically prune the activation values of the output feature maps in the runtime if necessary, such that intermediate feature maps can fit into limited RAM. Experimental results on SVHN and CIFAR-10 have shown that the proposed algorithm could significantly reduce the working memory of a DNN to satisfy the hard constraint of RAM size while maintaining satisfactory accuracy with relatively low overhead on memory and run-time latency.

Table of Contents

List of Tables

List of Figures

Preface

The thesis is based on my published paper [20], which was finished during my graduate years at University of Pittsburgh. The original work [20] was supported in part by the National Science Foundation under Grant CNS-2007274 and in part by the University of Pittsburgh Center for Research Computing (CRC) through providing computing resources. It also used the computing resources of the Extreme Science and Engineering Discovery Environment (XSEDE) [18], which is supported by National Science Foundation grant number ACI-1548562. Specifically, it used the Bridges system, which is supported by NSF award number ACI-1445606, at the Pittsburgh Supercomputing Center (PSC) [16].

I would like to thank the following people, without whom I would not have been able to complete this research, and without whom I would not have made it through my masters degree. First of all, my parents and friends, who were always willing to help and support me with love and understanding whenever I was depressed. And secondly, my advisor, Professor Hu, whose insight and knowledge into my research topic steered me through this research. Also, I cannot express enough thanks to my committee for providing feedback and guidance to my research and thesis. Thank you all for your support!

1.0 Introduction

The past few years have witnessed the rapid development of artificial intelligence (AI) technology, where computer vision (CV) applications like face recognition and natural language processing (NLP) applications like voice assistant are everywhere in our daily life. In the meantime, we are also living in a fast-growing world of Internet of Things (IoT), which connects and shares data across a vast network of devices or "things" in a broad spectrum of applications from manufacturing and retailing to energy, smart cities, health care and beyond. The vast volumes of data provided by IoT could enable the AI algorithms to learn the complexity of the real world while the great analytical ability of the AI algorithms could significantly expand the value of IoT. Therefore, the vision of artificial intelligence of things (AIoT), which aims to inject AI algorithms into IoT devices (i.e., embedded devices), has been proposed. According to a recent market research report, embedded AI in support of IoT Things/Objects will reach $4.6B globally by 2024 [17].

In recent years, deep neural networks (DNNs) have become the mainstream of AI algorithms when we tried to apply AI technology to real-world applications due to its great power in making rapid decisions and uncovering deep insights based on noisy input data. While many works have been done, which focus on deploying DNNs on low-end devices such as mobile devices [14, 15, 24] and FPGAs [9, 10], little attention has been paid to the low-cost, low-power, and resource-constrained microcontroller units (MCUs), which are the majority of IoT devices. Therefore, to realize the vision of AIoT, it is essential to inject intelligence into the prolific embedded devices via deploying DNNs on MCUs. However, typical MCUs are resource-constrained, which have limited storage (e.g., ROM and Flash memory) capacity and run in low frequency (several or tens of MHz), while a typical DNN is resource-intensive, which usually has tens of millions of weights and uses billions of operations to finish one inference. Even a lightweight DNN (e.g., MobileNetV2 [15]) has over a million weights and millions of operations. The gap between the model size of DNNs and the storage capacity of MCUs makes the deployment of DNNs onto MCUs infeasible. Therefore, techniques for DNN compression such as [2, 6] have been adopted by some works such as GENESIS [5] to

deploy DNNs on resource-constrained MCUs. However, even if the DNN could be fit into the limited storage after compression, it still cannot run successfully if the size of intermediate results (i.e., feature maps) exceeds the size of limited RAM (e.g., 2KB - 64KB). Although we can constantly spill out the intermediate data to fast non-volatile memories such as FRAM in some cases [5], it is either too expensive or even infeasible for Flash memory or ROM in most of the off-the-shelf commercial MCUs.

The necessary space to save the intermediate results of a DNN is referred to as working memory Ω of the DNN. And we use Ω_l to denote the memory requirement of layer l of the DNN, which is also referred to as the working memory of layer l. It is defined as,

$$\Omega_l = |x_l| + |y_l|\,, \tag{1–1}$$

where $|x_l|$ denotes the size of activation values of input feature maps of layer l, and $|y_l|$ denotes the size of activation values of output feature maps of layer l, which is equivalent to $|x_{l+1}|$. For a DNN consisting of L layers, its working memory Ω is defined as $\max_{l\in\{1,\ldots,L\}} \Omega_l$. Note that the working memory Ω of a specific DNN is oversized when Ω exceeds the RAM size of the target MCU.

According to Eq. (1–1), we can conclude that Ω_l is related to the shape and number of filters of layer l. Existing DNN compression techniques such as fine-grained unstructured pruning in [6] focus on pruning the insignificant weights of filters, which could not reduce the working memory of a DNN since it does not change the shape or number of filters. Structured pruning [12], which removes a certain number of filters in each layer (as shown in Figure 1(a)(b)), could lead to the reduction of working memory. However, this method is not only coarse-grained but also static and invariant to different inputs. For instance, in Figure 1(a)(b), the filter in grey color is removed permanently and thus reduces the working memory. This pruning operation might not affect the inference of some inputs like input 0 shown in Figure 1(a). However, the features in the removed filter might be very important for the inference of some inputs like input 1 shown in Figure 1(b). And the removal of this filter could degrade the accuracy of the inference for this kind of inputs. Therefore, static structured pruning would weaken the representation capability of the original DNNs especially for those already small DNNs designed for MCUs, and thus lower the accuracy by

a large margin. Although [13] proposes dynamic structured pruning to mitigate the loss of accuracy by introducing dynamics to the coarse-grained structured pruning, it is infeasible to be applied to MCUs since it needs an extra DNN to decide the policy of pruning in the runtime, which makes it prohibitive for resource-constrained MCUs. Quantization [19] is another type of DNN compression technique that could reduce working memory by using fewer bits to represent each activation value in the output feature maps. However, the working memory of lots of DNNs could not satisfy the constraint of RAM size of MCUs even after their activation values are quantized to only one byte, which will be shown in Chapter 4. [19] proposes to quantize the activation values to less than one byte. However, this kind of quantization is not hardware-friendly and might cause problems when accessing the memory of MCUs without extra hardware support. Moreover, quantization is also static and thus insensitive to the inputs in the runtime.

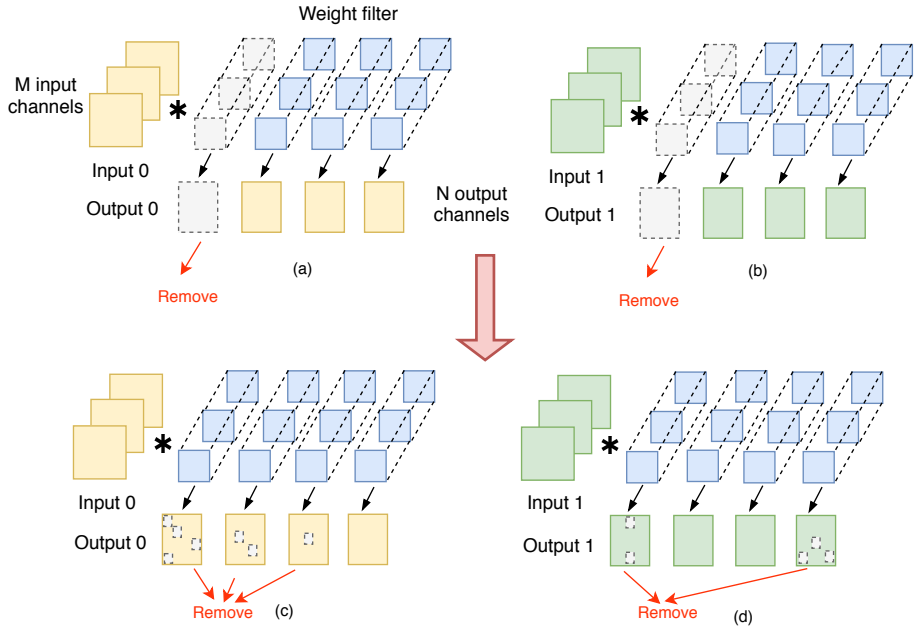

Figure 1: Static Pruning vs. Dynamic Pruning

To deploy a given DNN that could not directly run on MCUs due to its oversized working memory, we developed a lightweight run-time working memory compression algorithm to dynamically prune the intermediate output feature maps such that they could fit into RAMs without degrading accuracy significantly for certain inputs. The main idea is shown in

Figure 1. Instead of removing certain filters statically (grey filter in Figure 1 (a)(b)), our method could dynamically remove the insignificant activation values (white small squares in Figure 1 (c)(d)) in the output feature maps in the runtime. Since which values to be pruned are dynamically decided based on the current input, the method is sensitive to the input and thus minimizes the accuracy degradation incurred by the pruning for each input.

The main advantages of our method are as follows.

- **Effectiveness**. To the best of our knowledge, this is the first work to guarantee that a specific DNN with oversized working memory could fit into resource-constrained MCUs without changing the architecture of the deployed DNN. Since the complete architecture is reserved, the loss of accuracy incurred by pruning is usually reduced compared with static pruning that modifies the architecture of the original DNN, which shows the effectiveness of our algorithm.
- **Simplicity**. The method we proposed could be implemented easily on the off-the-shelf commercial MCUs without any extra hardware support.
- **Lightweight**. The method is also lightweight since the incurred memory overhead is negligible and the overhead on the run-time latency is moderate, which will be shown in Chapter 4.

Besides, the DNN running with our compression algorithm is also a good complement to the recent neural architecture search (NAS) algorithm designed specifically for MCUs [4], which will be illustrated in Chapter 2.

According to the experimental results shown in Section 4.2, our method could guarantee that the DNN with oversized working memory could fit into the limited RAM of the target MCU while maintaining satisfactory accuracy with relatively low overhead on memory and run-time latency.

Note that this thesis is based on my published paper [20], which means I reused most of the content in [20]. The remainder of the thesis is organized as follows. Chapter 2 reviews the related works and Chapter 3 describes our run-time working memory compression in detail. Experimental details are given in Chapter 4 and the concluding remarks are given in Chapter 5.

2.0 Related Work

This chapter will review the related work on DNN compression and the techniques developed for deploying DNNs on MCUs.

2.1 DNN Compression

DNN compression is a technique to reduce the model size of a specific DNN so that it could fit into the memory of mobile or embedded devices with negligible loss of accuracy [7, 19, 6, 12, 13]. Pruning is one of the common compression techniques, which could be divided into structured pruning [12, 13, 7] and unstructured pruning [6]. However, directly applying unstructured pruning to DNNs on MCUs could not solve the problem of oversized working memory, as we discussed in Chapter 1. Structured pruning and quantization [19] could reduce the working memory of a given DNNs since they effectively decrease the size of intermediate feature maps. And [22] is the first work to combine these two techniques to deploy DNNs on energy harvesting powered MCUs. However, the framework they proposed is optimized for energy harvesting settings and specified for a special kind of DNNs, i.e., multi-exit DNNs. To satisfy the more strict energy consumption constraint of energy harvesting powered devices, the accuracy of DNNs is sacrificed. Therefore, a more general method is needed to solve the problem of oversized working memory of DNNs on MCUs while maintaining the accuracy as much as possible.

2.2 Deployment of DNNs on Resource-Constrained MCUs

DNNs were once thought to be unsuitable for deployment on resource-constrained MCUs due to the gap between their complexity and the limited resources of MCUs. However, more attention has been paid to running DNNs on resource-constrained MCUs in recent

years. [8] designed and deployed a DNN on MCUs to detect ventricular arrhythmias and achieved better performance than conventional algorithms. However, the simple architecture of the proposed DNN hinders it from being applied to more complex tasks such as image classification. [5] is the first work to successfully deploy DNNs on energy harvesting powered MCUs. However, instead of ROM, this kind of MCUs uses FRAM as the storage component, which is more expensive and allows frequent writing operations. The problem of oversized working memory was overcome by constantly spilling out intermediate results from SRAM to FRAM. However, this strategy is either too expensive or infeasible for the MCUs with Flash memory or ROM. Different from the above works that focus on the inference of DNNs on MCUs, [23] proposed a framework for the efficient on-device training of DNNs. Based on their framework, the on-device training of LeNet could be achieved on MCUs, which helps to enable more use cases of DNNs on MCUs.

2.3 Hardware-Aware Neural Architecture Search (NAS)

Hardware-aware NAS is an emerging technology that could automatically generate the architecture of DNNs with the best accuracy for a particular application while satisfying the hardware constraints of target platforms [3, 10, 21]. While most of the existing hardware-aware NAS works focus on mobile devices or FPGAs, little attention has been paid to the design of DNNs on resource-constrained MCUs. Recently, [4] proposes a hardware-aware NAS customized for MCUs. It takes the memory usage, i.e., the model size and working memory of DNNs, into consideration in the search process. [1] is another hardware-aware NAS work for MCUs, which considered the requirement of latency in addition to the constraints on memory usage. Both of these works could eliminate the DNNs with oversized working memory in the search process since this kind of DNNs is regarded as not being able to run on MCUs by default. However, equipped with our run-time working memory compression, this kind of eliminated DNNs actually can run on the target MCU successfully. And they might have better accuracy compared with those DNNs having smaller working memory since a larger working memory usually implies a more complicated architecture with

stronger representation capability. Due to the simplicity of our compression algorithm, it is convenient to be merged into the NAS framework in [1, 4] and thus expanding the search space of NAS, which could lead to better results. Therefore, we can conclude that the DNN running with our compression algorithm is a good complement to the current hardware-aware NAS algorithms for MCUs.

3.0 Run-time Working Memory Compression

Traditional static structured pruning [12] aims to reduce the computational cost by removing certain filters on selected layers of DNNs, which could reduce the working memory of the corresponding layers at the same time. However, since it is usually applied to DNNs on mobile or cloud platforms, where the memory to store the intermediate data is sufficient, the reduction of working memory is only a side effect of this method and thus it is not optimized for the saving of working memory. In the original setting of structured pruning, if some layers with large working memory are sensitive to pruning, the algorithm could choose to prune less or even no filters in those layers in order to maintain the accuracy. However, the limited RAM size of MCUs poses hard constraints on the working memory of the deployed DNNs. Even if some layers are sensitive to pruning, they will have to be pruned heavily if their working memory exceeds the RAM size by a large margin. Therefore, we propose a run-time working memory (WM) compression specified for the deployment of DNNs with oversized WM on resource-constrained MCUs. It could reserve the complete architecture of the deployed DNNs if their weights could fit into the storage and dynamically prune the insignificant activation values of intermediate output feature maps in the runtime to satisfy the hardware constraint of RAM size. Therefore, our method could make full use of the representation capability of the original DNN and thus lower the accuracy loss incurred by pruning for some layers sensitive to it.

3.1 System Overview and Execution Model

Our run-time WM compression mainly consists of two parts, i.e., the offline part and the online part, which will be illustrated in Section 3.2 and 3.3, respectively. The offline part will decide the amount of activation values to prune in each layer of the DNN before deployment. If the working memory does not exceed the RAM size for a specific layer, no activation values need to be pruned and thus the WM compression will not be triggered in that layer in the

runtime, which reduces the online overhead as much as possible. The system overview of the online part is shown in Figure 2. And the online part will decide which activation values to prune dynamically in the runtime according to the output feature maps of the specific layer. Note that the choices could be distinct for different input data. When the online part is triggered for layer l, it means that RAM cannot hold the complete output feature maps from layer l. Therefore, we reserve a tiny buffer, which occupies a small space in RAM, to process the output feature maps progressively. The calculated activation values in Y_l' will be sent to a tiny buffer first after the inference operation. When the tiny buffer is full, the MCU will be notified to execute the code of the online part of our run-time WM compression. The program will decide k, the minimum number of activation values to prune for the specific data in the buffer. We introduce a mechanism with threshold τ to adapt k in the runtime, which will be discussed in Section 3.3. Therefore, we need a tiny space called Top K cache, to keep track of the smallest k activation values among the data in the buffer. Then, at least these k values are pruned and only the remaining part of the activation values are saved in the space for output feature maps. Besides, there is a bitmap related to the pruned output feature maps Y_l. It uses one bit to indicate whether the corresponding activation values are pruned in Y_l. The bit will be set to one if the corresponding activation value is pruned. When the MCU accesses the input feature maps of the next layer $l + 1$, it will first query the bitmap, and use zero to represent the activation value whose corresponding bit is set to one for the following inference operations. Otherwise, it will employ the original values saved in Y_l. By introducing a tiny buffer and considering all the incurred memory overhead in the offline part of our algorithm, our run-time WM compression can guarantee that the DNN with oversized working memory could fit into the limited RAM appropriately and run successfully on the target MCU.

3.2 Offline Part of Compression Algorithm

Algorithm 1 presents the steps of the offline part of run-time WM compression in detail. The purpose of this part is to decide the number of activation values to prune in the

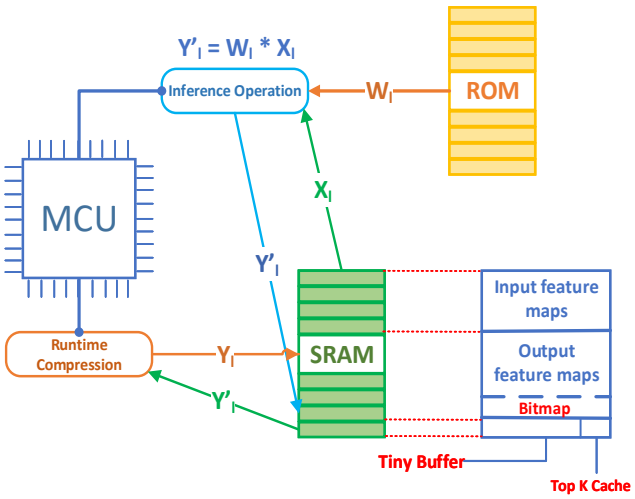

Figure 2: System Overview of Run-Time WM Compression

corresponding output feature maps of each layer in the DNN with an online pruning array D_p as output. Since this process is done before the deployment of a DNN on the MCU, it is offline and thus reduces the online overhead. Note that in our WM compression, we only need to consider the convolution layers for compression. For the pooling layer, it is used to downsample the output Y from the convolution layer. If Y could fit into RAM, then the downsampled feature maps must be able to fit into RAM. For the fully connected (FC) layer, the size of its output is usually much smaller than that of the convolution layer. Thus, there is little possibility that the working memory of the FC layer could exceed RAM size. As for the input of Algorithm 1, RAM size S_r and tiny buffer size S_b are both converted to the number of activation values they can hold.

In the statements of Algorithm 1, function **GetInSize(M)** returns an array containing the number of activation values in the input feature maps of each convolution layer of DNN M, while function **GetOutSize(M)** returns an array with the number of activation values in the output feature maps of each convolution layer. The decision process of D_p iterates through the N convolution layers. For a specific layer l, there are two constraints on the output feature maps and only one of them would be the bottleneck in a specific case. The first one is the hard constraint of RAM size S_r. When the working memory of layer l exceeds S_r,

according to the memory layout shown in Figure 2, the hard constraint could be formulated as,

$$S_r = S_{in}^{(l)} + S_{out}^{(l)} - D_p^{(l)} + S_{bm}^{(l)} + S_b + S_{tkc}^{(l)}, \tag{3–1}$$

where $S_{in}^{(l)}$ is the size of the input of layer l, while $S_{out}^{(l)}$ is the original size of output of layer l. S_b denotes the size of tiny buffer. And $S_{bm}^{(l)}$ is the size of bitmap, where $S_{bm}^{(l)} = \lceil \frac{S_{out}[l]}{bw} \rceil$ and bw is the bit width of an activation value. $S_{tkc}^{(l)}$ is the size of Top K cache, where $S_{tkc}^{(l)} = 2 * \frac{D_p^{(l)} * S_b}{S_{out}^{(l)}}$. Since we need to record both the indices and the values in the Top K cache, there is a multiplier of 2 to calculate S_{tkc}. Based on Eq. (3–1), we can get $D_p^{(l)}$, which records the number of activation values to prune for layer l. $D_p^{(l)}$ obtained in this case corresponds to the option 1 (Opt. 1) for num to calculate $D_p^{(l)}$ in Algorithm 1.

Another constraint is from the observation that if $S_{in}^{(l)}$ is small and $S_{out}^{(l)}$ is quite large for layer l, the remaining spaces for $S_{out}^{(l+1)}$ might be very tight if we only consider the hard constraint in Eq. (3–1) for pruning and thus degrading the accuracy significantly. Therefore, we introduce a predefined output threshold α to ensure that the space occupied by the output of layer l does not exceed $\alpha * S_r$ after pruning. And the constraint could be formulated as,

$$\alpha * S_r = S_{out}^{(l)} - D_p^{(l)} + S_{bm}^{(l)} + S_{tkc}^{(l)}. \tag{3–2}$$

$D_p^{(l)}$ acquired based on Eq. (3–2), corresponds to the option 2 (Opt. 2) for num to calculate $D_p^{(l)}$ in Algorithm 1. Besides, if there is a pooling layer after layer l, $S_{in}^{(l+1)}$ might not be equal to $S_{out}^{(l)}$. And we have

$$S_{in}^{(l+1)} = \min \left\{ S_{out}^{(l)} - D_p^{(l)}, S_{pool}^{(l)} \right\}, \tag{3–3}$$

where $S_{pool}^{(l)}$ is the size of the output from the pooling layer following convolution layer l, returned by the function **GetPoolSize(**M, l). After N iterations, the resulted D_p is the output of Algorithm 1 and would be used as one of the inputs of the online part of run-time WM compression.

Algorithm 1: Offline Part of Run-Time WM Compression

Input: Original DNN M with N convolution layers, output threshold α, RAM size S_r, tiny buffer size S_b, bit width bw of an activation value

Output: Online pruning array D_p

Begin

$S'_{in} \leftarrow$ **GetInSize**(M);

$S_{out} \leftarrow$ **GetOutSize**(M);

$S'_{out} \leftarrow S_{out}$;

for $l = 1, ..., N$ **do**

$S_{bm} \leftarrow \lceil \frac{S_{out}[l]}{bw} \rceil$;

$S'_{out}[l] \leftarrow S'_{out}[l] + S_{bm}$;

$S_{wm} \leftarrow S'_{in}[l] + S'_{out}[l]$;

$dem \leftarrow 1 - 2 * (S_b / S_{out}[l])$;

$p \leftarrow$ **False**;

if $S'_{out} > \alpha * S_r$ **then**

$p \leftarrow$ **True**;

if $S'_{in}[l] + S_b \geq (1 - \alpha) * S_r$ **then**

$num \leftarrow S'_{in}[l] + S'_{out}[l] + S_b - S_r$; // Opt. 1

else

$num \leftarrow S'_{out}[l] - \alpha * S_r$; // Opt. 2

else if $S_{wm} > S_r$ **then**

$p \leftarrow$ **True**;

$num \leftarrow S'_{in}[l] + S'_{out}[l] + S_b - S_r$; // Opt. 1

else

$num \leftarrow 0$;

$D_p[l] \leftarrow \lceil \frac{num}{dem} \rceil$;

for_continue

  if p **then**

    $S'_{out}[l] \leftarrow S'_{out}[l] - D_p[l]$;

    if *there is a pooling layer after layer* l **then**

      $S_{pool} \leftarrow$ **GetPoolSize**(M, l);

      $S'_{in}[l+1] \leftarrow \min\{S'_{out}[l], S_{pool}\}$;

    else

      $S'_{in}[l+1] \leftarrow S'_{out}[l]$;

3.3 Online Part of Compression Algorithm

Algorithm 2 presents the steps of the online part of run-time WM compression in detail for a specific input data X_1. The purpose of this part is to decide which activation values to prune dynamically according to the output feature maps X_l of specific layer l with $D_p^{(l)} > 0$. For layer l, the activation values are pruned in the unit of batch B, whose size is S_b. The amount of activation values to prune within a batch (i.e., P_b) is the mean of $D_p^{(l)}$ over all the T batches initially. Note that although the output feature maps and their related bitmaps are made up of three dimensions, they are flattened to one dimension in Algorithm 2 for the simplicity of index. Function $\mathbf{Conv}(X_l, M, l, s, e)$ calculates the activation values of layer l starting from index $s+1$ to e through convolution operations, which might include batch normalization and ReLU functions. And the result Y_b is saved in the tiny buffer. The first P_b activation values and their corresponding indices in the batch are sorted in ascending order and used to initialize the Top K cache as $\mathcal{K}$ and $\mathcal{K}_{id}$, respectively. Then each value Y_b^j in Y_b is compared with the values in $\mathcal{K}$ through function $\mathbf{TopKAdd}(\mathcal{K}_{id}, \mathcal{K}, Y_b^j, j)$. If Y_b^j is less than any value in $\mathcal{K}$, then Y_b^j and j will be added to $\mathcal{K}$ and $\mathcal{K}_{id}$, respectively. And the last value in $\mathcal{K}$ and its corresponding index will be removed from Top K cache. Otherwise, $\mathcal{K}_{id}$ and $\mathcal{K}$ will keep unchanged. Besides, Y_b^j should also be compared with pruning threshold τ.

If Y_b^j is less than τ, then it will be pruned and its corresponding bit in bitmap BM will be set to one through function **SetBitmap**$(BM, i * (S_b - 1) + j)$, where i is the index of the batch for Y_b^j.

After all of the activation values in the batch are processed by the steps mentioned above, if the activation values pruned by the threshold τ are more than the preset number P_b, then we can reduce P_b for the following batches, where P_b is the average number of the total amount of activation values to prune in the rest of the output feature maps. Therefore, we can reserve more important features in the following batches and thus mitigate the loss of accuracy. Otherwise, the program will prune all the activation values recorded in the Top K cache and set the corresponding bits to one in bitmap BM. Then, the activation values Y_b after pruning within the batch will be moved to the corresponding position in the space for compressed output feature maps Y_l. After all the batches are processed, our run-time WM compression for convolution layer l is finished. If there is a pooling layer after layer l, Y_l will be downsampled through the function **Pool(**Y_l, M, l**)**. And the downsampled Y_l will be used as the input X_{l+1} for the next layer $l + 1$.

In the end, all of the N convolution layers are processed by our compression method. The generated features X_{N+1} would be the input to the remaining fully connected layers in the deployed DNN. And the final prediction Y would be calculated through function **FC(**X_{N+1}, M**)**.

Algorithm 2: Online Part of Run-time WM Compression

Input: Input data X_1, original DNN M with N convolution layers, online pruning array D_p, pruning threshold τ, tiny buffer size S_b, bitmap BM initialized with zeros

Output: Prediction Y

Begin

$P_{cur} \leftarrow 0$;
$S_{out} \leftarrow \textbf{GetOutSize}(M)$;
for $l = 1, ..., N$ **do**
  if $D_p[l] > 0$ **then**
    $T \leftarrow \lceil \frac{S_{out}[l]}{S_b} \rceil$;
    $P_b \leftarrow \lceil \frac{D_p[l]}{T} \rceil$;
    $B \leftarrow S_b$;
    for $i = 1, ..., T$ **do**
      if $i == T$ **then**
        $P_b \leftarrow D_p[l] - P_{cur}$;
        $B \leftarrow S_{out}[l] \mod S_b$;
      $s, e \leftarrow (i-1) * S_b, \min\{i * S_b, S_{out}[l]\}$;
      $Y_b \leftarrow \textbf{Conv}(X_l, M, l, s, e)$;
      $\mathcal{K}_{id}, \mathcal{K} \leftarrow \textbf{Sort}(Y_b[0, P_b])$;
      $C_0 \leftarrow 0$;
      for $j = 1, ..., B$ **do**
        $\textbf{TopKAdd}(\mathcal{K}_{id}, \mathcal{K}, Y_b[j], j)$;
        if $Y_b[j] < \tau$ **then**
          $\textbf{SetBitmap}(BM, i * (S_b - 1) + j)$;
          $Y_b[j] \leftarrow 0$;
          $C_0 \leftarrow C_0 + 1$;

for_continue

for_continue

if $C_0 > P_b$ **then**

$P_{cur} \leftarrow P_{cur} + C_0$;

$P_b \leftarrow \lceil \frac{D_p[l] - P_{cur}}{T - i} \rceil$;

else

for $k \in \mathcal{K}_{id}$ **do**

SetBitmap$(BM, i * (S_b - 1) + k)$;

$Y_b[k] \leftarrow 0$;

$P_{cur} \leftarrow P_{cur} + P_b$;

$Y_l[s:e] \leftarrow Y_b$;

if $P_{cur} \geq D_p$ **then**

break;

else

$Y_l \leftarrow$ **Conv**$(X_l, M, l, 0, S_{out}[l])$;

if *there is a pooling layer after layer* l **then**

$Y_l \leftarrow$ **Pool**(Y_l, M, l);

$X_{l+1} \leftarrow Y_l$;

$Y \leftarrow$ **FC**(X_{N+1}, M);

4.0 Experiments

This chapter reports the experimental results of our proposed run-time working memory compression on SVHN and CIFAR-10. The results show that our method could reduce the working memory of a given DNN effectively with accuracy higher than the original DNN or with acceptable accuracy loss. Moreover, the accuracy of the DNN compressed by our method outperforms the DNN compressed with static structured pruning by a large margin in most cases. Besides, we also conduct sensitivity analysis for the two hyperparameters in our algorithm, i.e., tiny buffer size S_b and pruning threshold τ, to explore their impact on the performance of our algorithm.

4.1 Experimental Setup

4.1.1 Dataset

The datasets we used in the following experiments are SVHN and CIFAR-10. For SVHN, it has 99289 images of digits from 0 to 9. In our experiments, the size of training set, validation set and testing set is 65932, 7325 and 26032, respectively. For CIFAR-10, it contains 60000 images in 10 classes, which consist of airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. And the size of training set, validation set and testing set are 45000, 5000, and 10000, respectively. Besides, the images in both datasets are RGB images with a resolution of 32 by 32. Compared with SVHN, CIFAR-10 is more challenging for DNNs to learn.

4.1.2 Evaluated DNNs

The DNNs evaluated in our experiments are LeNet-A, SpArSeNet-A and SonicNet-A, which are the adapted versions of three lightweight DNNs suitable for resource-constrained MCUs, i.e., LeNet [11], SpArSeNet [4] and SonicNet [5], respectively. The details of these

DNNs are listed in Table 1. MS represents the model size of the DNN while WM denotes its working memory. # Filters records the number of filters for each convolution layer and kernel shape records the shape of the square kernels of the corresponding filters. Note that all of the convolution layers of the evaluated DNNs use valid padding with a stride equal to 1. Pool position is a list of the placement of pooling layers, i.e., the indices of the convolution layers followed by the pooling layers. FC config provides the list of the size of output features of fully connected layers, which are placed after the series of convolution layers and pooling layers. For example, LeNet-A has two convolution layers. Both of them have 5×5 kernels. The first layer has 6 filters while the second one has 32 filters. They are both followed by a pooling layer. Note that only the second pooling layer of LeNet-A uses global average pooling. The other pooling layers of the evaluated DNNs use max-pooling by default. LeNet-A has three fully connected layers after all the convolution and pooling layers. And the size of output features of them is 120, 84, 10, respectively. Moreover, the weights and the activation values of these three DNNs are all quantized to one byte under Arm configuration in Pytorch. In addition, the evaluated DNNs are implemented in Pytorch and trained on a single Nvidia RTX 2080 Ti GPU.

Table 1: Configuration of Evaluated DNNs

Network	MS (KB)	WM (KB)	# Filters	Kernel Shape	Pool Position	FC Config
LeNet-A	19.79	4.59	[6, 32]	[5, 5]	[1, 2]	[120, 84, 10]
SpArSeNet-A	24.33	15.74	[9, 11, 17, 39]	[3, 4, 1, 5]	[2, 4]	[10]
SonicNet-A	60.17	15.31	[20, 80]	[5, 5]	[1, 2]	[10]

4.1.3 Baseline

According to Chapter 2, static structured pruning and quantization could also effectively reduce the working memory of DNNs on resource-constrained MCUs. Since we have already quantized the DNNs to one byte in our experiments, only static structured pruning could reduce working memory further. Therefore, we implemented the method in [12] as the baseline. Note that in [12], the layers to prune are decided manually since the goal of their

work is to reduce the computational cost and there is no memory constraint in their work. But in our implementations, pruning will be triggered when the working memory of the running layer exceeds RAM size. Besides, the baseline and our run-time working memory compression are both implemented in Pytorch.

4.1.4 Hyperparameters of Run-Time WM Compression

The hyperparameters of our run-time working memory compression include buffer size S_b, pruning threshold τ and output threshold α. For the first two hyperparameters, their values are changeable in different experiments since we want to explore the relationship between their setting and the accuracy of pruned DNNs, which will be shown in Section 4.2.2. As for the output threshold α, its value is fixed in our experiments. And we set the value of α to 0.8, 0.5 and 0.8 for LeNet-A, SpArSeNet-A and SonicNet-A, respectively.

4.2 Experimental Results

4.2.1 Evaluation of Run-Time WM Compression

Table 2 and Table 3 show the experimental results on SVHN and CIFAR-10, respectively. Our experiments evaluated the three mentioned DNNs running with our proposed run-time working memory compression (RTWMC) and compared it with the corresponding baselines, i.e., static structured pruning (SSP). The basic memory configurations of the target MCUs are listed as storage size and RAM size. After quantization, all of the three DNNs could fit into the storage of the target MCUs while their working memory still exceeds the corresponding RAM size. Therefore, to run on the target MCUs, pruning the intermediate feature maps is necessary. In the mentioned two tables, original ACC represents the accuracy of the DNNs without pruning, while pruned ACC is the accuracy after pruning. Note that the pruned ACC we showed for RTWMC in the two tables is the best accuracy when buffer size is less than 50 bytes, which corresponds to the cases where the overhead on memory and latency is relatively small.

Table 2: Evaluation of the Three DNNs with Run-Time WM Compression on SVHN

Network	Storage Size (KB)	RAM Size (KB)	Original Acc (%)	Pruning Method	Pruned ACC (%)	Memory Overhead (KB)	Estimated Runtime Latency	Buffer Size (B)	Pruning Threshold
LeNet-A	32	4	81.57	RTWMC	**81.45**	0.08	1.24x	40	0.2
				SSP	72.56	-	-	-	-
SpArSeNet-A	32	8	89.69	RTWMC	85.59	0.09	1.27x	40	0.8
				SSP	**87.55**	-	-	-	-
SonicNet-A	64	8	88.80	RTWMC	**87.68**	0.10	1.22x	40	0.5
				SSP	86.76	-	-	-	-

According to Table 2, for LeNet-A and SonicNet-A, the accuracy after pruning with our proposed method (i.e., RTWMC) is only 0.12% and 1.12% lower than the original accuracy, respectively. Moreover, the accuracy of our method on LeNet-A and SonicNet-A outperforms that of the baseline (i.e., SSP) by 8.89% and 0.92%, respectively. As for SpArSeNet-A, our pruning method incurs 4.1% accuracy loss and is 1.96% lower than the baseline. The main reason for the underperformance is due to the simplicity of SVHN, which means that a naive and compact architecture is sufficient to get good performance for such a simple dataset. Therefore, the advantage brought by compactness is over the limitation on representation capability. The main overheads of RTWMC are the overhead on memory and run-time latency. For SSP, all of the space of RAM is used to store the intermediate feature maps, while for RTWMC, some of them are reserved for the tiny buffer and Top K cache as shown in Figure 2. But the total memory overhead is quite small, which are 0.08 KB, 0.09 KB and 0.10 KB, respectively. And it is negligible compared with the corresponding RAM size. Besides, we also estimated the runtime latency of RTWMC, which is represented in the form of the ratio of the latency of RTWMC to that of running the DNNs without pruning. The ratio is 1.24x, 1.27x and 1.22x, respectively, which is moderate for deploying DNNs on MCUs.

According to Table 3, for LeNet-A and SonicNet-A, the accuracy after pruning with our proposed method (i.e., RTWMC) is 0.62% and 2.3% higher than the original accuracy, respectively. It might be due to the regularization functionality provided by our pruning method, which could improve the generalization capability of the original DNNs. In addition,

the accuracy of our method on LeNet-A and SonicNet-A outperforms that of the baseline (i.e., SSP) by 18.58% and 14.87%, respectively. As for SpArSeNet-A, our pruning method incurs 3.61% accuracy loss, while the accuracy loss of the baseline is 7.59%. The accuracy of our method outperforms the baseline by 3.98%. It shows that in CIFAR-10, a more complicated data set than SVHN, our method could reduce the accuracy loss incurred by pruning as much as possible. The overhead on memory and run-time is still relatively small. The total memory overhead is 0.02 KB, 0.09 KB and 0.07 KB, respectively. And the ratio for estimated latency is 1.08x, 1.26x and 1.17x, respectively.

Table 3: Evaluation of the Three DNNs with Run-Time WM Compression on CIFAR-10

Network	Storage Size (KB)	RAM Size (KB)	Original ACC (%)	Pruning Method	Pruned ACC (%)	Memory Overhead (KB)	Estimated Runtime Latency	Buffer Size (B)	Pruning Threshold
LeNet-A	32	4	59.44	RTWMC	**60.06**	0.02	1.08x	10	0.3
				SSP	41.48	-	-	-	-
SpArSeNet-A	32	8	75.62	RTWMC	**72.01**	0.09	1.27x	40	0.3
				SSP	68.03	-	-	-	-
SonicNet-A	64	8	68.81	RTWMC	**71.11**	0.07	1.17x	30	0.1
				SSP	56.24	-	-	-	-

Based on the experimental results in Table 2 and Table 3, we can claim that our RTWMC could maintain the accuracy of the original DNNs as much as possible in most cases, especially for complicated datasets like CIFAR-10. In the meantime, it is lightweight for the deployment of DNNs on resource-constrained MCUs.

4.2.2 Sensitivity Analysis of Hyperparameters

In this section, we report the experimental results about the sensitivity analysis of the two hyperparameters in RTWMS. Section 4.2.2.1 shows the analysis of pruning threshold τ, while Section 4.2.2.2 is about the analysis of buffer size S_b.

4.2.2.1 Sensitivity Analysis of Pruning Threshold

Figure 3 shows the impact of pruning threshold τ on the top-1 accuracy of the DNNs pruned by RTWMC on SVHN when the buffer size S_b is fixed. For LeNet-A, SpArSeNet

and SonicNet, the buffer size is 50 B, 30 B and 60 B, respectively. When the pruning threshold is 0, each output feature map is pruned equally, which means the number of activation values to be pruned is the same for each feature map. And the achieved accuracy is already relatively high. For LeNet-A and SonicNet-A, it is 8.85% and 0.7% higher than the baseline, respectively. And for SpArSeNet-A, it is 2.47% lower than the baseline. This result shows the advantage of fine-grained dynamic unstructured pruning over coarse-grained static structured pruning. Moreover, the accuracy could be further improved with an appropriate pruning threshold. The highest accuracy is 81.42%, 85.08% and 87.81% for LeNet-A, SpArSeNet and SonicNet, which is achieved when pruning threshold τ equals 0.5, 0.8 and 0.4, respectively. Besides, when the threshold is large (i.e., 10 in our experiments), the accuracy of the evaluated DNNs drops dramatically and is less than 45% for all these three DNNs. More specifically, the corresponding accuracy is 43.59%, 22.67% and 19.12% for LeNet-A, SpArSeNet and SonicNet-A, respectively. In this case, the first several output feature maps are almost removed completely for inference, which reduces the accuracy significantly. This case is quite similar to the case where the structured pruning is applied in the run-time and thus without the chance to retrain the weights. Therefore, this result could imply the advantage of fine-grained dynamic unstructured pruning over the naive coarse-grained online structured pruning.

Figure 4 shows the impact of pruning threshold τ on the top-1 accuracy of the DNNs pruned by RTWMC on CIFAR-10 when the buffer size S_b is fixed. For LeNet-A, SpArSeNet and SonicNet, the buffer size is 70 B, 30 B and 90 B, respectively. When the pruning threshold is 0, the achieved accuracy is already relatively high. For LeNet-A and SonicNet-A, it is 18.32% and 14.59% higher than the baseline, respectively. And for SpArSeNet-A, it is only 0.78% lower than the baseline. Moreover, the accuracy could be further improved with an appropriate pruning threshold. The highest accuracy is 60%, 71.46% and 71% for LeNet-A, SpArSeNet and SonicNet, which is achieved when pruning threshold τ equals 0.7, 0.4 and 0.3, respectively. And all of them outperform the baseline by a large margin. Besides, when the threshold is large (i.e., 10 in our experiments), the accuracy of the evaluated DNNs drops dramatically and is less than 25% for all these three DNNs. More specifically, the corresponding accuracy is 20.28%, 11.22% and 17.31% for LeNet-A, SpArSeNet and

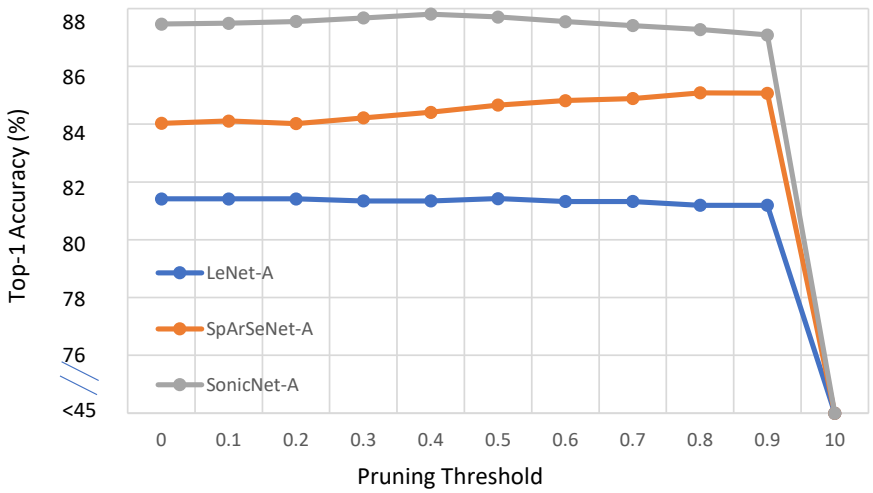

Figure 3: Top-1 Accuracy of Run-Time Working Memory Compression (RTWMC) with Different Pruning Threshold on Three Evaluated DNNs on SVHN.

SonicNet-A, respectively.

In conclusion, when the buffer size is fixed, simply using a small pruning threshold τ such as 0.1 is enough to get a satisfactory accuracy. But an appropriate choice of the pruning threshold τ could lead to better accuracy.

.

4.2.2.2 Sensitivity Analysis of Buffer Size

Figure 5 shows the impact of buffer size S_b on the top-1 accuracy of the DNNs pruned by RTWMC on SVHN, when the pruning threshold τ is fixed. For LeNet-A, SpArSeNet and SonicNet, the pruning threshold τ is 0.7, 0.9 and 0.5, respectively. Since the complete output feature maps could not fit into the RAM, pruning could not be executed based on the global information of the feature maps. And in RTWMC, pruning is done in the unit of buffer size. If the buffer size is too small, the decision of pruning is only based on the information of a small number of activation values, which might lead to suboptimal solutions. Therefore, bigger buffer size is helpful to make wiser decisions about pruning. In Figure 5, the best

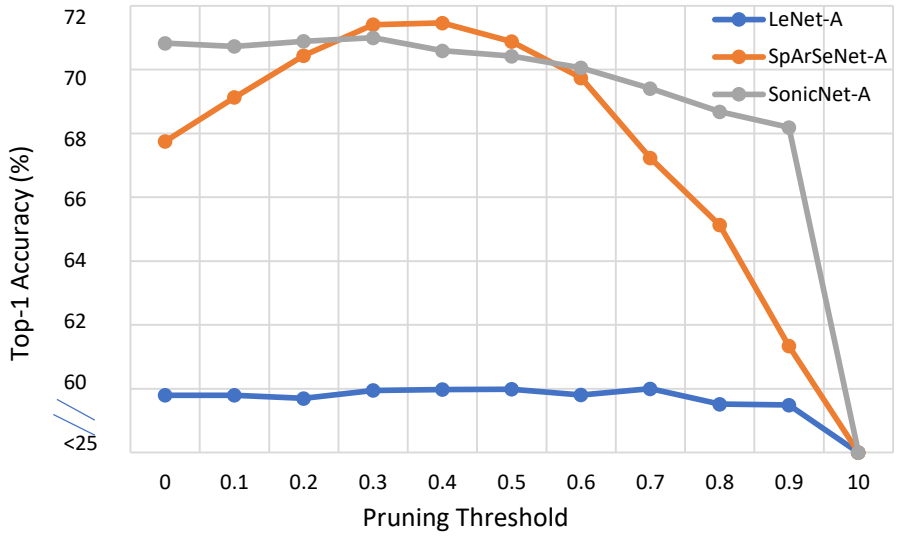

Figure 4: Top-1 Accuracy of Run-Time Working Memory Compression (RTWMC) with Different Pruning Threshold on Three Evaluated DNNs on CIFAR-10.

accuracy is 81.52%, 86.37% and 88.37% for LeNet-A, SpArSeNet and SonicNet, which is all achieved when the buffer size S_b is equal to 200, the largest evaluated buffer size. These results show the advantage of a large buffer size. But note that when the buffer size S_b equals 40, the achieved accuracy is already high, which is only 0.07%, 0.91% and 0.69% lower than the best accuracy, respectively. And the accuracy is improved the most when the buffer size S_b is increased from 10 to 20, which is 0.40%, 5.09% and 1.25%, respectively.

Figure 6 shows the impact of buffer size S_b on the top-1 accuracy of the DNNs pruned by RTWMC on CIFAR-10 when the pruning threshold τ is fixed. For LeNet-A, SpArSeNet and SonicNet, the pruning threshold τ is 0.3, 0.3 and 0.1, respectively. Although a large buffer size is useful to improve accuracy as shown in the experiments on SVHN, larger buffer size is not always better. If the buffer size is too large, the memory overhead and the run-time overhead will be increased significantly. Thus, we need to choose an appropriate buffer size in order to get the best trade-off between accuracy and overhead. For LeNet-A, SpArSeNet and SonicNet, the best accuracy is 60.04%, 72.01% and 71.11%, which is achieved when the buffer size equals 10 B, 40 B and 30 B, respectively, For LeNet-A, the best performance could

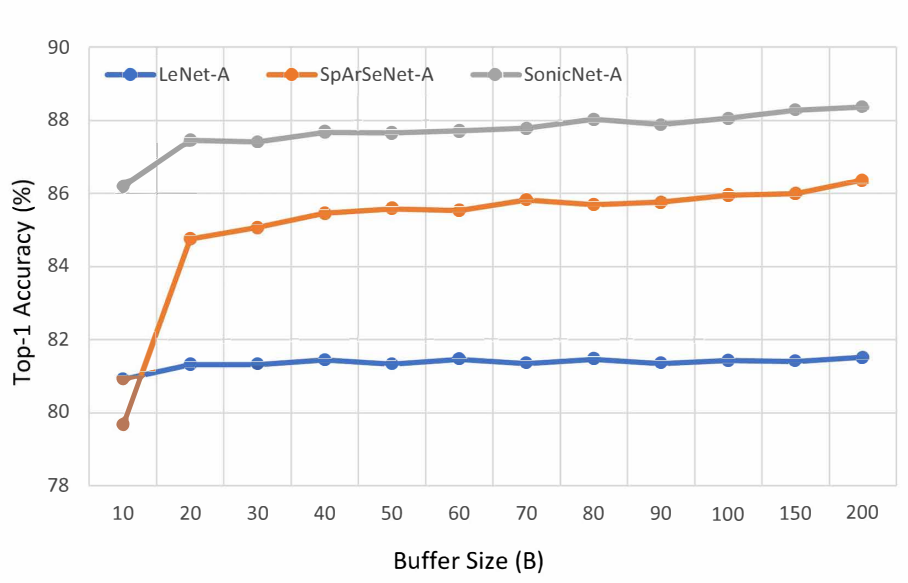

Figure 5: Top-1 Accuracy of Run-Time Working Memory Compression (RTWMC) with Different Buffer Size on Three Evaluated DNNs on SVHN.

be achieved with quite a small buffer size while for SpArSeNet and SonicNet, increasing the buffer size could improve the accuracy only at the early stage. When the best accuracy is achieved, the trend of the three curves in Figure 6 becomes stable with little change in the accuracy. It means that the positive and negative impacts brought by a larger buffer size achieve a subtle balance. And thus, increasing the buffer size is not necessary for better accuracy. Therefore, we can claim that for RTWMC, the best performance could be achieved with a relatively small buffer size, which justifies the small memory overhead and the moderate run-time overhead we claimed in Section 4.2.1.

In conclusion, a larger buffer size is helpful to improve the accuracy of RTWMC, especially at the early stage when we increase the buffer size from a small number. But a huge buffer is not necessary, and the conducted experiments have proved that a satisfactory accuracy could be achieved with relatively small buffer size.

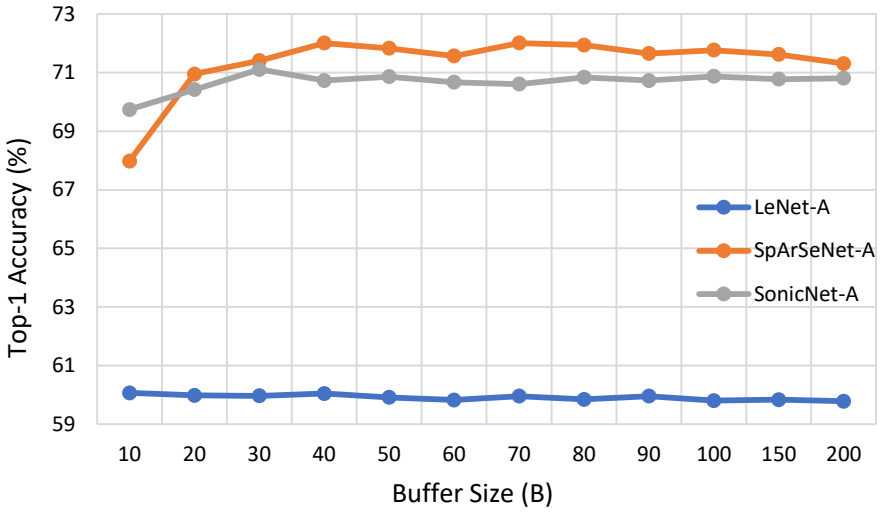

Figure 6: Top-1 Accuracy of Run-Time Working Memory Compression (RTWMC) with Different Buffer Size on Three Evaluated DNNs on CIFAR-10.

5.0 Conclusions

This work aims to enable the deployment of DNNs on resource-constrained MCUs when their working memory exceeds the RAM size of the target MCU. It proposes a lightweight run-time working memory compression to dynamically prune the activation values on the intermediate output feature maps of the deployed DNN when the working memory of specific layers is oversized, such that the working memory could be reduced to a size lower than the RAM size. Experimental results show that without incurring heavy overhead on memory and run-time latency, the compressed DNNs could maintain the original accuracy or run with moderate accuracy loss.

Bibliography

[1] Colby Banbury, Chuteng Zhou, Igor Fedorov, Ramon Matas, Urmish Thakker, Dibakar Gope, Vijay Janapa Reddi, Matthew Mattina, and Paul Whatmough. Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers. *Proceedings of Machine Learning and Systems*, 3, 2021.

[2] Sourav Bhattacharya and Nicholas D Lane. Sparsification and separation of deep learning layers for constrained resource inference on wearables. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*, pages 176–189, 2016.

[3] Han Cai, Ligeng Zhu, and Song Han. Proxylessnas: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332*, 2018.

[4] Igor Fedorov, Ryan P Adams, Matthew Mattina, and Paul Whatmough. Sparse: Sparse architecture search for cnns on resource-constrained microcontrollers. In *Advances in Neural Information Processing Systems*, pages 4977–4989, 2019.

[5] Graham Gobieski, Brandon Lucia, and Nathan Beckmann. Intelligence beyond the edge: Inference on intermittent embedded systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 199–213. ACM, 2019.

[6] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico*, 2016.

[7] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 784–800, 2018.

[8] Zhenge Jia, Zhepeng Wang, Feng Hong, Lichuan Ping, Yiyu Shi, and Jingtong Hu. Personalized deep learning for ventricular arrhythmias detection on medical iot systems. *arXiv preprint arXiv:2008.08060*, 2020.

[9] Weiwen Jiang, Edwin H.-M. Sha, Xinyi Zhang, Lei Yang, Qingfeng Zhuge, Yiyu Shi, and Jingtong Hu. Achieving super-linear speedup across multi-fpga for real-time dnn inference. *ACM Trans. Embed. Comput. Syst.*, 18(5s), October 2019. doi:10.1145/3358192.

[10] Weiwen Jiang, Xinyi Zhang, Edwin H-M Sha, Lei Yang, Qingfeng Zhuge, Yiyu Shi, and Jingtong Hu. Accuracy vs. efficiency: Achieving both through fpga-implementation

aware neural architecture search. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

[11] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[12] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

[13] Ji Lin, Yongming Rao, Jiwen Lu, and Jie Zhou. Runtime neural pruning. In *Advances in neural information processing systems*, pages 2181–2191, 2017.

[14] Wei Niu, Zhenglun Kong, Geng Yuan, Weiwen Jiang, Jiexiong Guan, Caiwen Ding, Pu Zhao, Sijia Liu, Bin Ren, and Yanzhi Wang. A compression-compilation framework for on-mobile real-time bert applications. *arXiv preprint arXiv:2106.00526*, 2021.

[15] Wei Niu, Zhengang Li, Xiaolong Ma, Peiyan Dong, Gang Zhou, Xuehai Qian, Xue Lin, Yanzhi Wang, and Bin Ren. Grim: A general, real-time deep learning inference framework for mobile devices based on fine-grained structured weight sparsity. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.

[16] Nicholas A Nystrom, Michael J Levine, Ralph Z Roskies, and J Ray Scott. Bridges: a uniquely flexible hpc resource for new communities and data analytics. In *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, pages 1–8, 2015.

[17] RESEARCH and MARKETS. Artificial intelligence (ai) in big data, data as a service (daas), ai supported iot (aiot), and aiot daas 2019 - 2024, November 2019. URL: `https://www.researchandmarkets.com/reports/4858130/artificial-intelligence-ai-in-big-data-data-as`.

[18] John Towns, Timothy Cockerill, Maytal Dahan, Ian Foster, Kelly Gaither, Andrew Grimshaw, Victor Hazlewood, Scott Lathrop, Dave Lifka, Gregory D Peterson, et al. Xsede: Accelerating scientific discovery computing in science & engineering, 16 (5): 62–74, sep 2014. *URL https://doi. org/10.1109/mcse*, 2014.

[19] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. Haq: Hardware-aware automated quantization with mixed precision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8612–8620, 2019.

[20] Zhepeng Wang, Yawen Wu, Zhenge Jia, Yiyu Shi, and Jingtong Hu. Lightweight runtime working memory compression for deployment of deep neural networks on resource-constrained mcus. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, ASPDAC '21, page 607–614, New York, NY, USA, 2021. Association for Computing Machinery. `doi:10.1145/3394885.3439194`.

[21] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[22] Yawen Wu, Zhepeng Wang, Zhenge Jia, Yiyu Shi, and Jingtong Hu. Intermittent inference with nonuniformly compressed multi-exit neural network for energy harvesting powered devices. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020. doi:10.1109/DAC18072.2020.9218526.

[23] Yawen Wu, Zhepeng Wang, Yiyu Shi, and Jingtong Hu. Enabling on-device cnn training by self-supervised instance filtering and error map pruning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3445–3457, 2020. doi:10.1109/TCAD.2020.3012216.

[24] Yawen Wu, Zhepeng Wang, Dewen Zeng, Yiyu Shi, and Jingtong Hu. Enabling on-device self-supervised contrastive learning with selective data contrast. *arXiv preprint arXiv:2106.03796*, 2021.