Lightweight Run-Time Working Memory Compression for Deployment of Deep Neural Networks on Resource-Constrained MCUs

Zhepeng Wang
University of Pittsburgh
Pittsburgh, Pennsylvania, US
zhepeng.wang@pitt.edu

Yawen Wu
University of Pittsburgh
Pittsburgh, Pennsylvania, US
yawen.wu@pitt.edu

Zhenge Jia
University of Pittsburgh
Pittsburgh, Pennsylvania, US
zhenge.jia@pitt.edu

Yiyu Shi
University of Notre Dame
Notre Dame, Indiana, US
yshi4@nd.edu

Jingtong Hu
University of Pittsburgh
Pittsburgh, Pennsylvania, US
jthu@pitt.edu

ABSTRACT

This work aims to achieve intelligence on embedded devices by deploying deep neural networks (DNNs) onto resource-constrained microcontroller units (MCUs). Apart from the low frequency (e.g., 1-16 MHz) and limited storage (e.g., 16KB to 256KB ROM), one of the largest challenges is the limited RAM (e.g., 2KB to 64KB), which is needed to save the intermediate feature maps of a DNN. Most existing neural network compression algorithms aim to reduce the model size of DNNs so that they can fit into limited storage. However, they 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 in MCUs even after compression. To address this problem, this work proposes a technique to dynamically prune the activation values of the intermediate output feature maps in the runtime to ensure that they can fit into limited RAM. The results of our experiments show that this method could significantly reduce the working memory of DNNs to satisfy the hard constraint of RAM size, while maintaining satisfactory accuracy with relatively low overhead on memory and run-time latency.

CCS CONCEPTS

• **Computer systems organization** → **Embedded software**; • **Computing methodologies** → *Neural networks*; Supervised learning by classification.

KEYWORDS

Neural Network Deployment, Neural Network Compression, Edge Computing, Artificial Intelligence of Things (AIoT)

ASPDAC '21, January 18–21, 2021, Tokyo, Japan

ACM ISBN 978-1-4503-7999-1/21/01…$15.00
https://doi.org/10.1145/3394885.3439194

ACM Reference Format:
Zhepeng Wang, Yawen Wu, Zhenge Jia, Yiyu Shi, and Jingtong Hu. 2021. Lightweight Run-Time Working Memory Compression for Deployment of Deep Neural Networks on Resource-Constrained MCUs. In *26th Asia and South Pacific Design Automation Conference (ASPDAC '21), January 18–21, 2021, Tokyo, Japan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3394885.3439194

1 INTRODUCTION

In the fast-growing world of Internet of Things (IoT), which connects and shares data across a vast network of devices or "things", analytics is the key to extract the most valuable information from raw data in a broad spectrum of applications – from manufacturing and retailing to energy, smart cities, health care and beyond. For its ability to make rapid decisions and uncover deep insights as it learns from massive volumes of IoT data, Artificial Intelligence (AI) is an essential form of analytics to expand the value of IoT. In Artificial Intelligence of Things (AIoT), AI adds value to IoT through machine learning and improves decision making while IoT adds value to AI through connectivity, signaling, and data exchange. According to a recent market research report, embedded AI in support of IoT Things/Objects will reach $4.6B globally by 2024 [14].

In recent years, deep neural networks (DNNs) have been proved to be a promising technique of AI with its powerful capability to make accurate inferences based on complex and noisy inputs. While many dedicated DNN accelerators such as TPU [9] have been designed, the majority of IoT devices are still using low-cost, low-power, and resource-constrained microcontroller units (MCUs). Therefore, to realize the vision of AIoT, it is essential to deploy 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 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, neural network compression techniques such as [1, 5] have been adopted by some works such as GENESIS [4] to deploy DNNs on resource-constrained MCUs. However, even if the DNN could be

fit into the limited storage with compression, it still cannot run successfully if the intermediate data (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 [4], it is either too expensive or 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. We use Ω_l to denote the memory requirement of layer l of a specific DNN. It is defined as:

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

where $|x_l|$ denotes the number of activation values of input feature maps of layer l, and $|y_l|$ denotes the number 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 $\max_{l \in 1,\ldots,L} \Omega_l$. And the working memory Ω of a specific DNN is oversized when Ω exceeds the RAM size of the target MCU.

According to Eq. (1), we can conclude that Ω_l is related to the shape and number of filters of layer l. Existing neural network compression techniques such as fine-grained unstructured pruning in [5] 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 [11], which removes 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 reduce the working memory. This pruning operation might not affect the inference of some inputs like input 0. However, the removed filter might be very important for the inference of some inputs like input 1. And the removal of this filter could degrade the accuracy of the inference of this kind of inputs. Thus, static structured pruning would weaken the representation capability of the original DNNs especially those already small DNNs designed for MCUs, and thus lower the accuracy by a large margin. Although [12] proposes dynamic structured pruning to mitigate the loss of accuracy incurred by this coarse-grained pruning, it is not suitable to be used in 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 [16] is another compression technique that could reduce working memory by using fewer bits to represent the activation values of output feature maps. However, the working memory of lots of DNNs could not satisfy the constraint of RAM even after their activation values being quantized to one byte, which will be shown in Section 4. [16] proposes to quantize the activation values to less than one byte. However, this kind of quantization is not hardware friendly and might cause problems in accessing the memory of MCUs without extra hardware support. Moreover, quantization is also static and thus insensitive to the inputs in the runtime.

To deploy a given DNN that could not directly run on MCUs due to the limited size of RAMs, 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

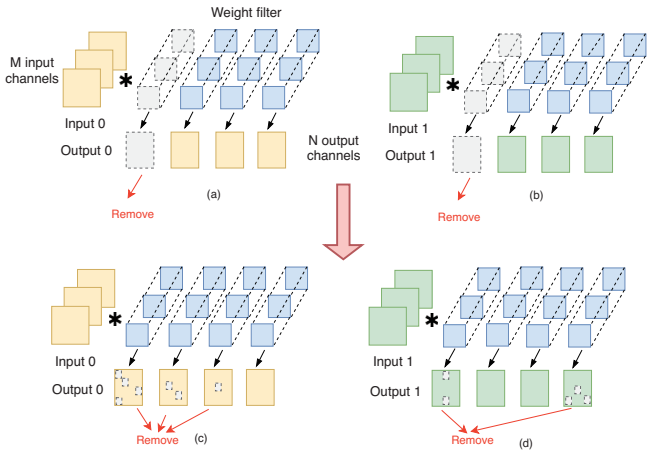

Figure 1: Static Pruning vs. Dynamic Pruning

without degrading accuracy significantly for certain inputs. The main idea is shown in Figure 1. Instead of removing certain filters statically (Figure 1 (a)(b)), our method could dynamically remove the insignificant activation values (white 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 minimize the accuracy degradation incurred by 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 reduced compared with static pruning that modifies the architecture of original DNNs, 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 Section 4.

Besides, the DNN running with our compression algorithm is also a good complement to the recent neural architecture search (NAS) specified for MCUs [3], which will be illustrated in Section 2.

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

The remainder of the paper is organized as follows. Section 2 reviews the related works and Section 3 describes the run-time working memory compression in detail. Experimental results are given in Section 4 and the concluding remarks are given in Section 5.

2 RELATED WORK

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 its complexity and the limited resources of MCUs. However, more attention has been paid to running DNNs on resource-constrained MCUs in recent years. [7] designs and deploys a DNN on MCUs to detect ventricular arrhythmias and achieves better performance than conventional algorithms. However, the simple architecture of the DNN hinders it from being applied to more complex tasks such as image classification. [4] is the first work to successfully deploy DNNs on energy harvesting powered MCUs. However, instead of ROM, this kind of MCUs use 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 data 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, [19] proposes a framework for the efficient on-device training of DNNs. Based on their framework, the online training of LeNet could be achieved on MCUs, which helps to enable more use cases of DNNs on MCUs.

DNN Compression. DNN compression is a technique to reduce the size of a specific DNN so that it could fit into the memory of mobile or embedded devices with negligible loss of accuracy [5, 6, 11, 12, 16]. Pruning is one of the common compression techniques, which could be divided into structured pruning [6, 11, 12] and unstructured pruning [5]. However, directly applying unstructured pruning to DNNs on MCUs could not solve the problem of oversized working memory, as we discussed in Section 1. Structured pruning and quantization [16] could reduce the working memory of a given DNNs since they effectively decrease the size of intermediate feature maps. And [18] is the first work to combine these two techniques to deploy DNNs on energy harvesting powered MCUs. However, the framework they propose 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.

Hardware-Aware Neural Architecture Search (NAS). Hardware-aware NAS is an emerging technique that could automatically generate the architecture of DNNs with the best accuracy for a particular application while satisfying the hardware constraints of target platforms [2, 8, 17]. While most of the existing works focus on mobile devices or FPGAs, little attention has been paid to the design of DNNs on resource-constrained MCUs. Recently, [3] proposes a hardware-aware NAS customized for MCUs. It takes the model size and working memory of DNNs into consideration in the search process. It could eliminate the DNNs with oversized working memory since they are regarded as not being able to run on MCUs by default in the search process. However, equipped with our run-time working memory compression, this kind of eliminated DNNs actually can run on the target MCUs successfully. And they might have better accuracy compared with those DNNs having smaller working memory since a larger working memory is usually related to 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 [3] and thus expand their search space, which could lead to better results. Therefore, we can conclude that the DNN running with our compression algorithm is a good complement to [3].

3 RUN-TIME WORKING MEMORY COMPRESSION

Traditional static structured pruning [11] aims to reduce the computational cost by removing certain filters on selected layers of DNNs, which could reduce the memory requirement 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 memory requirements are sensitive to pruning, the algorithm could choose to prune less or even no filters at those layers in order to maintain the accuracy. However, the limited RAM size in 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 memory requirements exceed the RAM size by a large margin. Therefore, we propose a run-time working memory (WM) compression specified for the deployment of DNNs 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 as much as possible 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 memory requirement 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. 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 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 related 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 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 RAM appropriately and run successfully on the target MCU.

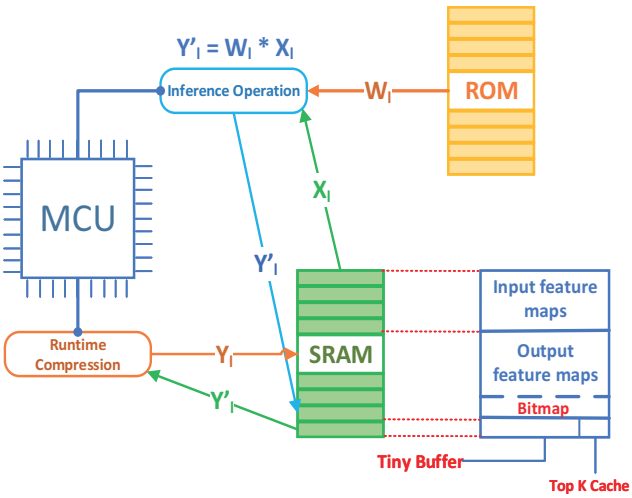

Figure 2: System Overview of Run-Time WM Compression

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 amount of activation values to prune in the 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 reduce 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 map must be able to fit into RAM. For the fully connected (FC) layer, the size of its output is much smaller than that of the convolution layer. Thus, there is little possibility that the memory requirement 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 different cases. The first one is the hard constraint of RAM size S_r. When the memory requirement of layer l exceeds S_r, according to the memory layout shown in Figure 2, the hard constraint could be formulated as

$$S_r = S_{in}^{(l)} + S_{out}^{(l)} - D_p^{(l)} + S_{bm} + S_b + S_{tkc}, \tag{2}$$

where $S_{in}^{(l)}$ is the size of 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} is the size of bitmap, where $S_{bm} = \lceil \frac{S_{out}[l]}{bw} \rceil$ and bw is the bit width of an activation value. S_{tkc} is the size of Top K cache, where $S_{tkc} = 2 * \frac{(S_{out}^{(l)} - D_p^{(l)}) * S_b}{S_{out}^{(l)}}$. Since we need to record both the indices and the values in the Top K cache, there is a multiplier 2 to calculate S_{tkc}. Based on Eq. (2), 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. (2) 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,

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

$D_p^{(l)}$ acquired based on Eq. (3), 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

$$S_{in}^{(l+1)} = \min\left\{S_{out}^{(l)}, S_{pool}\right\}, \tag{4}$$

where S_{pool} 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.

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 of specific layer l with

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

$S'_{in} \leftarrow$ **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
    end
  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
    $num \leftarrow 0$;
  end
  $D_p[l] \leftarrow \lceil \frac{num}{dem} \rceil$;
  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
      $S'_{in}[l+1] \leftarrow S'_{out}[l]$;
    end
  end
end

$D_p^{(l)} > 0$ for X_1. For layer l, the activation values are pruned in the unit of batch B, whose size is S_b. The amount of values pruned P_b for a batch 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 **Conv(**X_l, M, l, s, e**)** calculates the activation values starting from $s + 1$ 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 descending 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 $\mathcal{K}$ through function **TopKAdd**$(\mathcal{K}_{id}, \mathcal{K}, Y_b^j, j)$. If Y_b^j is larger 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 it 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 the 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 pruned values Y_b in 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 of the remaining fully connected layers in the deployed DNN. And the final prediction Y would be calculated through function **FC(**X_{N+1}, M**)**.

4 EXPERIMENTAL RESULTS

This section reports the experimental results of our proposed run-time working memory compression on CIFAR-10. The results show that our methods 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. 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 the impact of them on the performance of our algorithm.

4.1 Experimental Setup

Dataset. The dataset we used in the following experiments is CIFAR-10, which contains 60000 images in 10 classes. In our experiments, the size of training set, validation set and test set are 45000, 5000, and 10000, respectively. The pre-processing performed on the images are mean subtraction and division by the standard deviation.
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 [10], SpArSeNet [3] and SonicNet [4], respectively. The details of these DNNs are listed in Table 1. MS represents the model size of the DNN while MS represents its working memory. # Filters records the number of filters for each convolution layer and kernel shape records the shape of kernels of the corresponding filters.

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

$P_{cur} \leftarrow 0$;
$S_{out} \leftarrow$ **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**
        $P_b \leftarrow D_p[l] - P_{cur}$;
        $B \leftarrow S_{out}[l] \mod S_b$;
      end
      $s, e \leftarrow (i-1) * S_b, \min\{i * S_b, S_{out}[l]\}$;
      $Y_b \leftarrow$ **Conv**(X_l, M, l, s, e);
      $\mathcal{K}_{id}, \mathcal{K} \leftarrow$ **Sort**$(Y_b[0, P_b])$;
      $C_0 \leftarrow 0$;
      for $j = 1, ..., B$ **do**
        TopKAdd$(\mathcal{K}_{id}, \mathcal{K}, Y_b[j], j)$;
        if $Y_b[j] < \tau$ **then**
          SetBitmap$(BM, i * (S_b - 1) + j)$;
          $Y_b[j] \leftarrow 0$;
          $C_0 \leftarrow C_0 + 1$;
        end
      end
      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$(BM, i * (S_b - 1) + k)$;
          $Y_b[k] \leftarrow 0$;
        end
        $P_{cur} \leftarrow P_{cur} + P_b$;
      end
      $Y_l[s : e] \leftarrow Y_b$;
    end
  else
    $Y_l \leftarrow$ **Conv**$(X_l, M, l, 0, S_{out}[l])$;
  end
  if *there is a pooling layer after layer* l **then**
    $Y_l \leftarrow$ **Pool**(Y_l, M, l);
  end
  $X_{l+1} \leftarrow Y_l$;
end
$Y \leftarrow$ **FC**(X_{N+1}, M);

Note that all of the convolution layers of the evaluated DNNs use valid padding with stride equal to 1. Pool position is a list of the placement of pooling layers, i.e., the indices of convolution layers followed by the pooling layers. FC config provides the size of output features of each fully connected layer, which is 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 is global average pooling. The other pooling layers of the evaluated DNNs are max-pooling by default. LeNet-A has three fully-connected layers after all the convolution and pooling layers. And the output feature of them are 120, 84, 10, respectively. Moreover, the weights and the activation values of these three DNNs are all quantized to one byte under Arm configuration.

Baseline. According to Section 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 implement the method in [11] as the baseline. Note that in [11], 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 memory requirement exceeds RAM size.

Output Threshold α. Output threshold α is a predefined hyperparameter in our run-time working memory compression, which is mentioned in Section 3.2. In our experiments, 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 Evaluation of Run-Time WM Compression

Table 2 shows the experimental results of the evaluation of the three mentioned DNNs running with our proposed run-time working memory compression (RTWMC) and compares it with the baseline, i.e., static structured pruning (SSP). The basic memory configurations of the target MCUs are shown with storage size and RAM size. After quantization, all of the three DNNs could fit into the storage of the target MCUs while the working memory of them still exceeds the corresponding RAM size. Therefore, to run on the target MCUs, pruning the intermediate feature maps is necessary. Original ACC represents the accuracy of the DNNs without pruning, while pruned ACC is the accuracy after pruning. According to Table 2, 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 our method could reduce the accuracy loss incurred by pruning as much as possible. The main overheads of RTWMC are the overhead on memory and run-time latency. For

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]

SSP, all of the space of RAM are 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 0.02 KB, 0.09 KB and 0.07 KB, respectively. And it is negligible compared with the corresponding RAM size. Moreover, the accuracy of our RTWMC with these memory overheads is still much higher than that of SSP as shown in Table 2. Besides, we also estimate 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 1.08x, 1.26x and 1.17x, respectively, which is moderate for running DNNs on MCUs. Therefore, we can claim that our RTWMC is lightweight for the deployment of DNNs on resource-constrained MCUs.

4.3 Sensitivity Analysis of Hyperparameters

In this section, we report the experimental results about the sensitivity analysis of hyperparameters in RTWMS. Section 4.3.1 shows the analysis of pruning threshold τ, while Section 4.3.2 is about the analysis of buffer size S_b.

4.3.1 Sensitivity Analysis of Pruning Threshold τ. Figure 3 shows the impact of pruning threshold τ on the top-1 accuracy of the DNNs pruned by RTWMC, 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, 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 18.32% and 14.59% higher than the baseline, respectively. And for SpArSeNet-A, it is only 0.78% lower than the baseline. This result shows the advantage of fine-grained dynamic unstructured pruning over the coarse-grained static structured pruning. 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 at 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 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 dynamic structured pruning.

4.3.2 Sensitivity Analysis of Buffer Size S_b. Figure 4 shows the impact of buffer size S_b on the top-1 accuracy of the DNNs pruned by RTWMC, 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. 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, a bigger buffer size is helpful to make wiser decisions about pruning. But if the buffer size is too large, the memory overhead and the run-time overhead will be increased, which leads to the larger total number of activation values to be pruned and longer run-time latency. Thus, we need to choose an appropriate buffer size in order to get the best performance. The highest accuracy is 60.04%, 72.01% and 71.11% for LeNet-A, SpArSeNet and SonicNet, which is achieved with the buffer size of 10 B, 40 B and 30 B, respectively, For LeNet-A, the best performance could be achieved with quite small buffer size while for SpArSeNet and SonicNet, increasing the buffer size could improve the accuracy at the early stage. When the best accuracy is achieved, the trend of the three curves in Figure 4 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 conclude 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

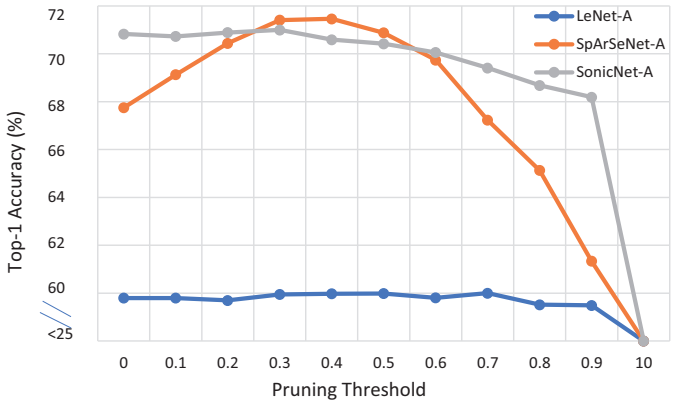

Figure 3: Top-1 Accuracy of Run-Time Working Memory Compression (RTWMC) with Different Pruning Threshold τ on Three Evaluated DNNs.

.

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

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	1.26x	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	-	-	-	-

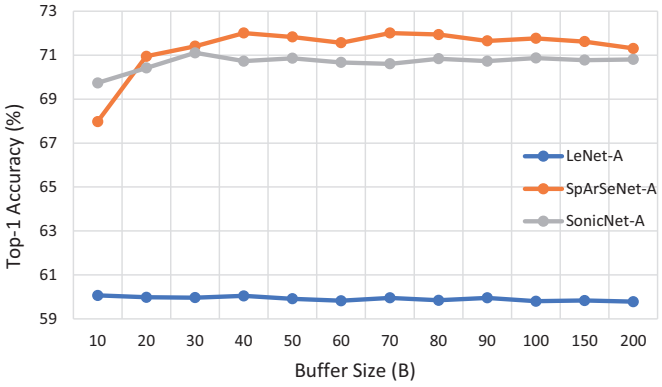

Figure 4: Top-1 Accuracy of Run-Time Working Memory Compression (RTWMC) with Different Buffer Size S_b on Three Evaluated DNNs.

5 CONCLUSIONS

This work aims to enable the deployment of DNNs on resource-constrained MCUs when their working memory exceeds the RAM size. It proposes a lightweight run-time working memory compression to dynamically prune the activation values on the intermediate output feature maps, such that the working memory is 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.

ACKNOWLEDGMENTS

This research 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 through the resources provided. It used the Extreme Science and Engineering Discovery Environment (XSEDE) [15], 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) [13].

REFERENCES

[1] Sourav Bhattacharya and Nicholas D Lane. 2016. 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*. 176–189.

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

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

[4] Graham Gobieski, Brandon Lucia, and Nathan Beckmann. 2019. 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*. ACM, 199–213.

[5] Song Han, Huizi Mao, and William J. Dally. 2016. 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*.

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

[7] Zhenge Jia, Zhepeng Wang, Feng Hong, Lichuan Ping, Yiyu Shi, and Jingtong Hu. 2020. Personalized Deep Learning for Ventricular Arrhythmias Detection on Medical IoT Systems. *arXiv preprint arXiv:2008.08060* (2020).

[8] Weiwen Jiang, Xinyi Zhang, Edwin H-M Sha, Lei Yang, Qingfeng Zhuge, Yiyu Shi, and Jingtong Hu. 2019. Accuracy vs. efficiency: Achieving both through fpga-implementation aware neural architecture search. In *Proceedings of the 56th Annual Design Automation Conference 2019*. 1–6.

[9] Norman Jouppi, Cliff Young, Nishant Patil, and David Patterson. 2018. Motivation for and evaluation of the first tensor processing unit. *IEEE Micro* 38, 3 (2018), 10–19.

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

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

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

[13] Nicholas A Nystrom, Michael J Levine, Ralph Z Roskies, and J Ray Scott. 2015. 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*. 1–8.

[14] RESEARCH and MARKETS. 2019. Artificial Intelligence (AI) in Big Data, Data as a Service (DaaS), AI Supported IoT (AIoT), and AIoT DaaS 2019 - 2024. Retrieved Nov. 27, 2019 from https://www.researchandmarkets.com/reports/4858130/artificial-intelligence-ai-in-big-data-data-as

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

[16] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. 2019. HAQ: Hardware-Aware Automated Quantization with Mixed Precision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 8612–8620.

[17] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. 2019. 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)*.

[18] Yawen Wu, Zhepeng Wang, Zhenge Jia, Yiyu Shi, and Jingtong Hu. 2020. Intermittent Inference with Nonuniformly Compressed Multi-Exit Neural Network for Energy Harvesting Powered Devices. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC18072.2020.9218526

[19] Yawen Wu, Zhepeng Wang, Yiyu Shi, and Jingtong Hu. 2020. 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 (2020), 3445–3457. https://doi.org/10.1109/TCAD.2020.3012216