Space-Efficient TREC for Enabling Deep Learning on Microcontrollers

Jiesong Liu
Renmin University of China
Beijing, China
liujiesong@ruc.edu.cn

Feng Zhang
Renmin University of China
Beijing, China
fengzhang@ruc.edu.cn

Jiawei Guan
Renmin University of China
Beijing, China
guanjw@ruc.edu.cn

Hsin-Hsuan Sung
North Carolina State University
Raleigh, North Carolina, USA
hsung2@ncsu.edu

Xiaoguang Guo
Renmin University of China
Beijing, China
xiaoguangguo@ruc.edu.cn

Xiaoyong Du
Renmin University of China
Beijing, China
duyong@ruc.edu.cn

Xipeng Shen
North Carolina State University
Raleigh, North Carolina, USA
xshen5@ncsu.edu

ABSTRACT

Deploying deep neural networks (DNNs) for a resource-constrained environment and achieving satisfactory performance is challenging. It is especially so on microcontrollers for their stringent space and computing power. This paper focuses on new ways to make TREC, an optimization recently proposed to enable computation reuse in DNNs, space and time efficient on Microcontrollers. The solution maximizes the performance benefits while keeping the DNN accuracy stable. Experiments show that the solution eliminates over 96% computations in DNNs and makes them fit well into microcontrollers, producing 3.4-5× speedups with only marginal accuracy loss.

CCS CONCEPTS

• **Computer systems organization** → **Real-time systems**; • **Software and its engineering** → *Compilers.*

KEYWORDS

real-time machine learning, compiler optimization

ACM Reference Format:
Jiesong Liu, Feng Zhang, Jiawei Guan, Hsin-Hsuan Sung, Xiaoguang Guo, Xiaoyong Du, and Xipeng Shen. 2023. Space-Efficient TREC for Enabling Deep Learning on Microcontrollers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3582016.3582062

1 INTRODUCTION

Deep neural networks (DNNs) are in high demand from servers to the edges and now to microcontroller-based devices. Microcontrollers dominate the computing engine market for small, low-cost or energy-efficient devices [6, 17, 30, 33]. Microcontrollers exist everywhere, from household appliances [42], to cars [47], consumer electronics [52], wearables [51] and so on. It is estimated that 250 billion microcontrollers are already in use [50]. Microcontrollers are very resource-constrained. For example, STM32 F469I contains only 324KB SRAM and 2048KB flash on-chip memory [7]. As a result, they have been widely regarded viable only for simple applications (e.g., keyword spotting [37, 65]), rather than complex DNN models.

But the demands for DNNs on microcontrollers [3, 13, 17, 30] keep growing, for three reasons. First, making DNNs runnable on microcontrollers can expand the range of AI-powered applications on more devices. Second, executing large DNNs locally on microcontrollers is essential for reducing energy consumption while increasing performance efficiency [7]. By eliminating the need for streaming from edge to the cloud [25], application latency can also be reduced and avoiding network congestion issues [24]. Finally, such designs eliminate many privacy concerns, as all user data is processed locally [26].

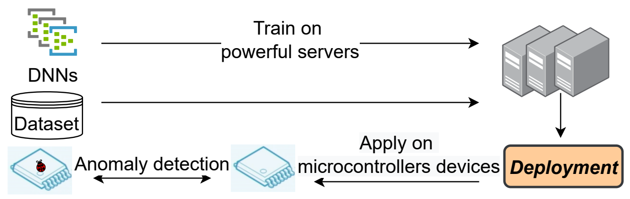


Figure 1: Use case of anomaly detection.

We show a use case of anomaly detection in Figure 1, which applies microcontrollers on a servo motor to retrieve vibration data [53]. With the help of microcontrollers, anomalies can be detected and reported in real time. In this particular case, the detection accuracy of 0.67 and a latency of over 1 second [12] are improved to 0.82 and 317ms when DNNs are moved from cloud to the device end [38]. In the same vein, many other tasks can benefit from on-device AI, ranging from exoskeletons [23], to voice activation [27, 59], object detection [4, 5, 49], and so forth.

There are some prior efforts trying to enable DNNs on microcontrollers, by creating light-weight neural networks [31, 60] and other designs [10, 14, 46]. However, the available DNNs that can be executed on microcontrollers are still very limited, and those that can run are often subject to long latency [36, 58].

Challenges to efficient DNN on microcontrollers lie in how to mitigate the tensions between performance and limited resources, shown in three aspects: (i) how to minimize the long execution time of DNN inferences under limited computing resources; (ii) how to host complicated DNNs in the limited memory including its inputs, outputs, weights, activation maps, and intermediate results; (iii) how to establish an optimal setting for minimizing the accuracy loss when optimizing the DNNs.

In this work, we propose a new approach, *space-efficient TREC*, to substantially improve the state of the art of DNNs on microcontrollers. TREC stands for *transient redundancy elimination-based convolution*. Its key idea is to identify similar elements in the input data of a convolution layer and avoid repeating similar computations on the fly. TREC makes the reuse mechanism a trainable component of a DNN. Because input data changes across inferences, the redundancy that TREC exploits is transient, hence the name.

The theoretical foundation TREC has been presented in a recent work [16], while this paper provides a deep examination of the challenges in making TREC effective for space-constraint devices, and details the optimizations that enable TREC to run efficiently on Microcontrollers. Specifically, we have made progress in three ways. First, as the new operator introduces additional space overhead, for space efficiency, we design a kernel reuse technique to reduce the space for storing parameters in the newly-built network. Second, we embed a two-step stack for storing clustering ID in TREC and use a reversed index to help locate the entries in the stack. Third, with the systematic design for the new network, we manage to integrate our techniques into back-propagation and keep marginal accuracy loss compared to the original TREC.

We evaluate our solution by applying it to three popular DNN networks, namely CifarNet [28], ZfNet [62], and SqueezeNet [21], on two microcontroller models. Our experiments show that, by implementing our solution on microcontrollers, we are able to avoid over 96% computations on convolution layers, and achieve an average of 3.4-5× reduction in the overall network latency with no or marginal accuracy loss.

Several previous studies have exploited similarities in inputs for DNNs. They either rely on special hardware [45] or use random hashing vectors that cause unstable inference accuracy [39]. None of them target microcontrollers or deal with the stringent space limitation. To our best knowledge, the solution from this work is the first that welds similarity-based computation reuse into DNN in a space-efficient manner for microcontrollers.

The main contributions of our work are as follows:

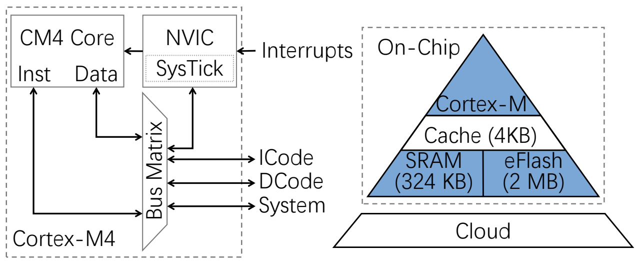
- We reveal the challenges of incorporating TREC for reducing computations in DNNs running on microcontrollers.
- We introduce a set of optimizations to mitigate the space overhead incurred by TREC.
- We empirically evaluate the effectiveness of the new solution on two models of microcontrollers, confirming the substantial benefits of the new solution in enabling efficient DNNs on microcontrollers.

2 BACKGROUND AND RELATED WORK

In this section, we introduce the deployment of neural networks on microcontrollers, the work related with input reuse and work related with model compression.

Microcontrollers. Microcontroller (MCU) is an energy-efficient processor that is ubiquitous in our lives. We show an example of MCU architecture and its memory hierarchy (STM32F469I in our case) in Figure 2. Figure 2 (a) shows that the Cortex-M4 core architecture consists of a 32-bit processor (CM4) and a small number of critical peripherals. The CM4 core is a Harvard-architecture, which means that it utilizes distinct interfaces to fetch instructions (Inst) and data (Data). This helps ensure that the CPU does not run out of memory, as it enables simultaneous access to the data and instruction memories. The special feature that distinguishes Cortex M4 from CM3 [9] is that CM4 includes, for the processor, single-instruction multiple-data (SIMD) extensions that are effective in achieving faster arithmetic computing performance in the CPU context. From the CM4's perspective, everything appears to be a memory.And the CM4 Core will only distinguish instruction fetches and data access. According to distinctive sections of memory, CM4 communicates with outer hardware through different memory buses (i.e., ICode, DCode, and System).

Figure 2 (b) shows its on-chip memory hierarchy consisting of very limited memory space. We can see that microcontrollers are typically comprised of a central processing unit (CPU), cached memory for frequently accessed data, static random-access memory (SRAM), and an on-chip flash memory for storage.



(a) Architecture of Cortex-M4. (b) Memory Hierachy for MCUs.

Figure 2: An illustration of the architecture and memory hierarchy for microcontrollers.

Microcontrollers are especially energy efficient with low-power (0.166W for F469I board) and cost effective (around $10) compared to common processors like CPU and GPU [63]. Given this, microcontrollers provide very limited computing resources and a limited volume of storage (around 1 cm^3) that developers need to take care of. In addition to physical microcontrollers, cloud functions, which can be seen as virtual resource-constrained devices in the cloud or the edge, are also widely used for cost-efficient computation and data processing tasks [19]. Optimizations that improve the space efficiency on microcontrollers may apply to cloud-function-based applications.

Work on input reuse. Several studies have explored similarities in inputs for DNN accelerations. *Deep reuse* [39, 55, 56, 64] is a pure software approach. It inspired this current work. *Deep reuse* deals with convolution as a General Matrix Multiplication (GEMM) in the form illustrated on the right side of Figure 3. After transformation, each row in X is a neuron vector. *Deep reuse* finds and clusters similar neuron vectors as a group. It thus avoids repeating similar computations by reusing the computation results of the cluster centroid as the results of the rest of the neuron vectors in the same group. The work uses some random locality-sensitive hashing vectors for the online clustering. It does not consider space efficiency; it actually increases the space usage substantially due to the extra space needed to store hashing vectors and cluster indices. The use of random hashing vectors also cause large fluctuations in the inference accuracy (detailed in Section 7). There also exist some data-effcienct-based works [54] that judiciously select a subset of train data (i.e. coreset) for training, which can much improve the efficiency while performing on par with the full train data. Some other studies [45] have designed special hardware to exploit input reuse.

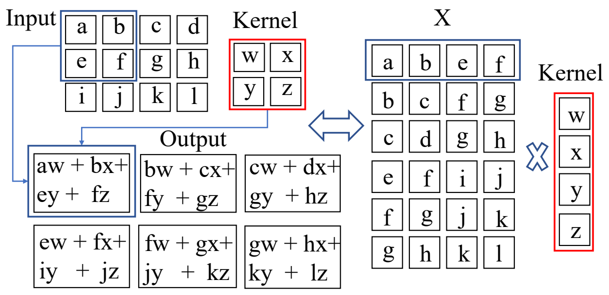


Figure 3: An illustration of convolution in *deep reuse.*

Relations with model compression approaches. Many studies on DNN compression [18] have exploited the redundancy among DNN parameters, which is orthogonal to input-level reuse. Examples include quantization [61], squeezing filter size [11], conducting feature compression to activation map [48], adding a downsampling [44], converting large models to light-weight counterparts through knowledge distillation [20], and applying adaptive LSH framework to reduce the size of model updates like Mongoose [8]. These techniques can significantly reduce the model size (i.e., weights and biases), the amount of computations, and data required to stream from edge to the cloud. For resource-stringent microcontrollers, input reuse and model compression have to be applied at the same time as shown later (TREC design in Sections 4&5 and model compression in Section 6) in this paper.

Locality Sensitive Hashing (LSH) LSH is an online clustering method used in multiple solutions (including TREC) for enabling computation reuse in DNN optimizations. As shown in Equation 1, for a parameter vector $\mathbf{v}$, the input vector $\mathbf{x}$ is transformed into 1 or 0 follow the hash function $h_{\mathbf{v}}$:

$$h_{\mathbf{v}}(\mathbf{x}) = \begin{cases} 1, & if \quad \mathbf{v} \cdot \mathbf{x} > 0 \\ 0, & if \quad \mathbf{v} \cdot \mathbf{x} \leq 0 \end{cases} \tag{1}$$

If H hash functions are used, for a given input vector, LSH maps it to a H-bit vector. Nearby input vectors often produce the same bit vector after hashing, and the number of hash functions, H, adjusts clustering roughness.

Each neuron vector then can be labeled an ID number according to the corresponding H-bit vector. And neuron vectors with the same ID can form a cluster and reuse the computing results from the centroid vector in place of the individual results for each vector.

3 OVERVIEW

Running large DNN inference on microcontrollers is especially challenging because of the strict resource constraints both in terms of computing capabilities and memory footprint. Removing computation redundancy and achieving space efficiency is thus essential for enabling efficient DNN deployment on microcontrollers.

Redundancy in DNNs can be categorized, based on where the redundancy originates, into *lasting redundancy* and *transient redundancy*. Lasting redundancy originates from the model parameters. As parameters are unchanged when applying inference, this kind of redundancy can be eliminated by methods such as pruning and quantization. Transient redundancy, however, exists in the form of similar tiles inside an input data or activation map [16].

There are many techniques that can be adopted for the elimination of lasting redundancy when running neural networks. For example, CMSIS-NN [33] applies quantization to the kernel weights and consequently avoids floating point computations for DNN networks, thus improving inference performance. However, ways of reducing transient redundancy on microcontrollers remain insufficiently explored.

To address this issue, we devise a principled way to efficiently detect and remove transient redundancy for DNNs on microcontrollers. It is based on Transient Redundancy Elimination-based Convolution (TREC), an idea of integrating reuse into a DNN as new kinds of DNN operators. This architecture has multiple benefits. First, by eliminating transient redundancy, TREC minimizes the computation volume, thus bringing the maximum computation-elimination benefits and improving the performance of the entire network. Second, TREC is compatible with both training and inference tasks, allowing for a simple plug-and-play replacement of convolutional layers in standard convolutional neural networks (CNNs) for training and inferencing. This approach guarantees highly robust model accuracy. Third, applying TREC in the model is orthogonal to methods targeting removal of lasting redundancy, which can bring further benefits when combined together. Section 4 gives more details of the TREC architecture.

Adding the new operators into the DNN inference stage reduces the amount of required computations significantly, but also introduces space overhead. To achieve space efficiency, we introduce *two-step stack substitution* for the DNN network to minimize the space occupancy for the clustering containers. Furthermore, for extreme cases where efficient space usage is of paramount importance (e.g., microcontrollers), we propose the *kernel reuse* technique to further remove the overhead while still eliminating the transient redundancy when performing DNN inference. Figure 4 outlines the overall process of space-efficient TREC. We next explain it in more details.

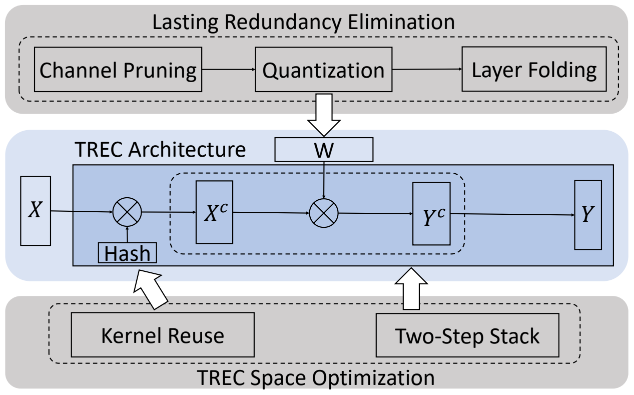


Figure 4: Overview of space-efficient TREC.

4 TREC ARCHITECTURE

To help understand the description on Space Efficient TREC, we first give a review of the basic architecture of TREC [16], its training and properties.

The main aim of TREC is grouping similar neuron vectors into clusters and then redirecting clustering results back to the DNN. To achieve this objective, it fuses the detection and avoidance of transient redundancy into DNN, making them part of its inherent architecture[16].

As illustrated in Figure 5, the original GEMM convolution is $\mathbf{y} = \mathbf{x} \cdot \mathbf{W}$. The TREC operator can be decomposed into four steps. First, the input matrix $\mathbf{x}$ passes through a clustering component. At a high level, TREC uses Locality-Sensitive Hashing (LSH) to do the clustering. In this example, $\mathbf{x} \in \mathbb{R}^{4\times4}$ is first sliced into two 4×2 sub-matrices, each of which has its own hash functions. And the 4 row vectors in each sub-matrix are grouped into two clusters. In the second step, for each $\mathbf{x}$, $\mathbf{x} \cdot \mathbf{W}$ is performed using the representative neuron vectors, allowing the matrix size for computing to be significantly reduced. In the third step, for each sub-matrix, vectors in the same cluster use the computation results from the corresponding centroid neuron vector to recover the full-sized output matrix. Finally, the final result is obtained by adding the results computed separately for each sub-matrix.

The main computation savings come from the clustering component. This is because in GEMM-based convolution, we should compute four results for each neuron vector. By using TREC, we use the result of the centroid computation in place of the per-row results for each of the row vector. Therefore, the number of vector-matrix computation reduces from four to two.

Benefits and the key conditions. By grouping neuron vectors into clusters, the size of the input matrix is significantly reduced, allowing for low computational complexity for the subsequent matrix multiplication. Consider that an input matrix X for GEMM (after im2col) is of $N \times K$ dimension and a weight matrix W is

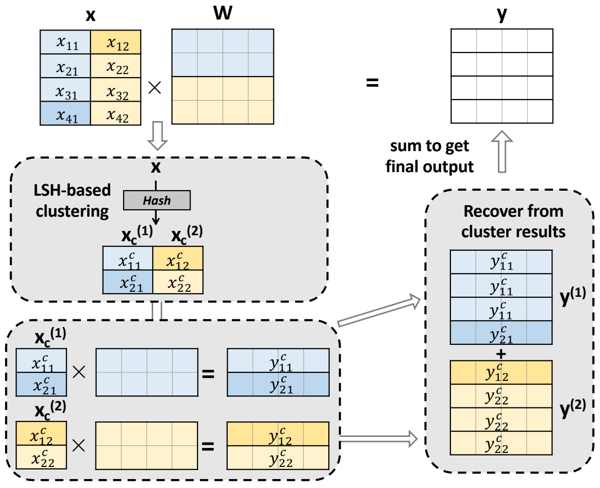


Figure 5: An illustration of TREC.

$K \times M$ size. The clustering step can be abstracted as applying a hash function matrix *Hash* to X. Let $K \times H$ be the dimension of the hash matrix. We can assume that the neuron vectors can be grouped into N_c clusters. For each input image or activation map, the benefits for removing the *transient redundancy* can thus be measured by a *redundancy radio*, i.e., $r_t = 1 - \frac{N_c}{N}$ as the reduction of input size. N is the total number of neuron vectors, and the number of centroid vectors is equal to the number of clusters, namely N_c. The number of required computations is thus reduced to N_c. We can conclude that r_t indicates the fraction of *transient redundancy* within input images or activation maps.

Using this measure, the total Floating-point Operations (Ops) for a conventional GEMM-based convolution is $N \cdot K \cdot M$, while the Ops for TREC will be $\left(\frac{H}{M} + 1 - r_t\right) \cdot N \cdot K \cdot M$, since TREC also involves a *Hash* matrix multiplication. Therefore, in order for TREC to expedite DNN inference (i.e., $\left(\frac{H}{M} + 1 - r_t\right) \cdot N \cdot K \cdot M < N \cdot K \cdot M$), the following *key condition* must hold: $\frac{H}{M} < r_t$. As shown in the experiments, the average transient redundancy elimination benefits r_t exceeds 96%.

Comparison with *deep reuse*. Prior studies like *deep reuse* have treated *transient redundancy* in an ad-hoc manner. *Deep reuse* takes place as extra operations outside DNN, using LSH with random hashing vectors for online data clustering. Such ad-hoc treatment causes severe uncertainty about the impact of the clustering errors on the DNN performance. Experimental results show that *deep reuse* causes significant (e.g., 5%) fluctuations on DNN accuracy.

TREC overcomes the limitations of *deep reuse*. The newly introduced DNN operator TREC incorporates transient redundancy detection and avoidance directly into DNN's inner architecture, while *deep reuse* functions outside the DNN. As a result, TREC achieves a consistent inference performance and accuracy.

Training TREC So far we have discussed how TREC can benefit from reusing the centroid results. We have also shown how TREC performs efficient convolutions in the following stages: 1) clustering: grouping similar neuron vectors through hash functions, 2) computing and reusing results from centroid vectors in each cluster in place of the results of each vector in the cluster, and 3) recovering back to final results. Now, we explain how to jointly learn the hash functions and kernel weight parameters, $Hash$ and W. Direct learning of TREC leads to problems: 1) discrete mapping problem, where we need to map values to 0 or 1 according to their signs, and 2) combinatorial optimization problem, where we have to compute the centroids for each cluster. To get around this, we reformulate the clustering stage.

We propose the following workflow for clustering in Figure 6 to make back-propagation work with the clustering in TREC. First, to solve the discrete mapping problem, after the input matrix $\mathbf{x}$ is multiplied by the $Hash$ matrix, the projected matrix is subjected to an element-wise sigmoid functioning as a binary classifier. In line with the proposed design, each neuron vector is thus transformed into a H-bit vector. In the next step, the bit vector is converted into a cluster ID, akin to converting a numerical value from its binary representation to the decimal counterpart. We then apply the mapper matrix to the ID vector and, after applying a Gaussian function, we obtain a bitmap of the neuron vector where each column indicates whether this vector belongs to a corresponding cluster. This bitmap helps to transform the computation of centroids from a combinatorial problem to a matrix multiplication.

After obtaining centroids, we can multiply the centroids with the weights W and perform the rest of the computation step of TREC as in Figure 4.

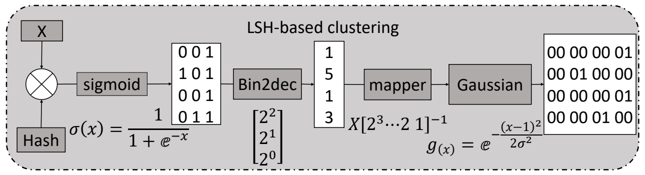

LSH-based clustering
X
Hash
sigmoid
$\sigma(x) = \frac{1}{1+e^{-x}}$
0 0 1
1 0 1
0 0 1
0 1 1
Bin2dec
$\begin{bmatrix}2^2\\2^1\\2^0\end{bmatrix}$
1
5
1
3
mapper
$X[2^3 \cdots 2\ 1]^{-1}$
Gaussian
$g_{(x)} = e^{-\frac{(x-1)^2}{2\sigma^2}}$
00 00 00 01
00 01 00 00
00 00 00 01
00 00 01 00

Figure 6: Training design for the clustering component in TREC.

Properties of TREC TREC has several appealing properties.

- **Accuracy.** Reusing results from the centroid vector for the vectors in the whole cluster is driven by the similarities among the vectors in the cluster. The learning process in TREC helps minimize the approximation errors. If we train the neural network with *random* LSH vectors, i.e., without learning the *Hash* matrix in the back propagation, accuracy could drop sharply. The *learned Hash* matrix in TREC makes a big difference in improving the accuracy of the network, as our experiments in Section 7 will show. The lesson is that the *Hash* matrix is learned and updated in a way that it resonates with the clustering process. In a sense, it chooses the optimal LSH vectors that are able to detect the similarities between vectors, and it helps group the similar vectors into the same cluster.
- **Robustness.** Traditional GEMM-based CNNs are robust in that they possess stability for small input perturbation. In order to assess the robustness of a neural network, we apply the *Lipschitz constant* L to analyze TREC. L gives the relations between the input perturbation ϵ and the output variations δ, namely $\delta \leq L\epsilon$. The *Lipschitz constant* for TREC is rigorously proved to be bounded by the GEMM-based Convolution, namely, $L\,(TREC) \leq L\,(Conv)$. Therefore, when the input image suffers from small perturbation, according to the properties of *Lipschitz continuous* of TREC, the output variation, denoted as δ_{TREC}, is bounded by the L_∞-norm of the weight matrix W, i.e., $\delta_{TREC} \leq L\,(TREC)\,\epsilon \leq L\,(Conv)\,\epsilon \leq \|W\|_\infty \epsilon$. This shows that TREC is robust to input perturbation.
- Convergence. TREC-equipped DNNs converge under reasonable assumptions: (a) The objective function $F\,(W, H)$, in which W and H correspond to the weight matrix and *Hash* matrix, is continuously differentiable, and (b) $F\,(W, H)$'s partial derivatives ∇F are *Lipschitz continuous* where their first and second moments meet certain limits. Given the above assumptions, we are able to obtain the fact that the expected change in F between two iterations are bounded. Therefore, for stochastic gradient descent optimization strategy, the partial derivatives of the objective function F converge to zero after many iterations. This is true to both fixed and diminishing stepsizes and thus proves that we have found the locally optimal solution [16].
- **Compatible with Sparse Matrices** TREC is applicable in the presence of sparse weights matrix or sparse input matrix, making it possible to be used together with other DNN optimizations (e.g., pruning). We provide details in Appendix A.

5 ACHIEVING SPACE EFFICIENCY

This section provides the techniques we propose that make TREC space-efficient, and explain how to make TREC-based DNN efficient on Microcontrollers.

5.1 Space Pressure For DNNs on Microcontrollers

In this section, we analyze the space pressure for DNNs on microcontrollers. Unlike desktop systems, microcontrollers have a rather flat memory system, as they are equipped with on-chip main memory only. To fit large DNNs into such a memory system, given the small size of microcontroller devices, we carefully analyze the space usage for the original network and the additional space required for TREC.

Figure 7 provides a typical memory breakdown for neural networks on the STM32F746ZG SRAM. The runtime overhead for the mbed-os is fairly small, requiring just 25 KB on SRAM. Loading buffers are needed for image input and are allocated. As CifarNet is equipped with only 2 convolutional layers, the full weights and biases are stored and buffered on the persistent buffers, where pointers to the intermediate buffers are also stored. Activation maps are also allocated on SRAM as well.

Additional space for TREC. Adding the TREC operator to the DNN can minimize the required computations for the whole network, allowing for efficient inference performance. However, it brings additional space overhead. Specifically, an extra *Hash* matrix and containers for holding cluster information are needed. Since there are at most 2^H different clusters, the additional space overhead for clustering computing is $O(L{\times}2^H)$, space required to be allocated on the *free* section in Figure 7. Consider TREC operators

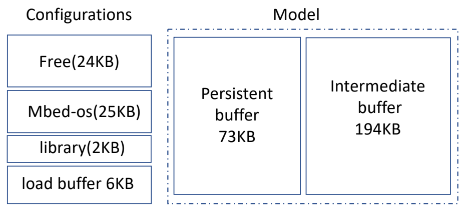


Figure 7: Breakdown of SRAM memory for an original DNN without TREC on STM32F746ZG.

in N layers, the total space overhead for the *Hash* matrix, however, is $\Sigma_{n=1}^{N} C_{i-1} \times k_i^2 \times H$, where k_i and C_i denotes the kernel size and the channel size of layer i, respectively.

Empirical analysis for space overhead. Space resource is very limited on microcontrollers. Specifically, SRAM is important as primary storage because direct computations are performed on it. Flash memory stores read-only data and uses SRAM to load data when doing computation. However, applying TREC can bring additional burden to SRAM spaces. Table 1 shows the SRAM size of four typical STM32 microcontrollers and the space ratio of TREC overheads to these boards. The corresponding L and H configurations for determining the TREC space overhead are displayed in Table 4 in Section 7.3. For each DNN network, *Hash* denotes the additional space required for the LSH matrix, while *Cluster* relates to the indexing and storing vectors for each cluster. For example, TREC requires an additional 16KB to conduct clustering operations and 41KB for hash tables for SqueezeNet.

Table 1: Space overheads introduced by TREC for the three networks and their ratio to the SRAM of four boards.

	Network	CifarNet		ZfNet		SqueezeNet	
		Hash	Cluster	Hash	Cluster	Hash	Cluster
	Space overhead	20 KB		40 KB		56 KB	
	Breakdown	60.6%	39.4%	65.8%	34.2%	91.1%	8.9%
Space ratio	F205 (64KB)	31.3%		62.5%		87.5%	
	F303 (80KB)	25.0%		50.0%		70.0%	
	F446 (128KB)	15.6%		31.3%		43.8%	
	F746 (320KB)	6.3%		12.5%		17.5%	

Space limitations. Original networks have large model sizes and need *lasting redundancy* elimination to fit in the microcontroller board. For example, the sizes of ZfNet and SqueezeNet are 5MB, which are reduced to below MB after pruning. TREC affects the footprint of the pruned model. In fact, Table 1 shows that TREC space overheads are substantial compared to the total SRAM capacities on microcontrollers. Therefore, the space overhead introduced by TREC is non-negligible and hinders running DNN applications on microcontrollers, considering facets such as larger batch sizes.

5.2 TREC Space Optimization

Although only trivial KB-level space occupancy overhead is introduced, which is negligible in the context of cloud computing, this capacity requirement poses challenges in the materialization of TREC on microcontrollers. Achieving space-efficient TREC is hence crucial.

Conventional optimization. Conventional implementations for space savings include DNN channel pruning, quantization, and layer folding. For these techniques, the goal is to prune model parameters and reduce the required static space for storing the model structure and parameters. Therefore, these *offline* pruning methods rely on delicate detection and examination of unnecessary parameters in the network, aiming at removing *lasting* redundancy (i.e., redundancy existing in DNN parameters). These techniques are discussed as implementation details in Section 6.1. Another category of potential space savings lies in reducing the *online* dynamic execution space occupancy of TREC, namely space savings for the clustering component. Emphases are placed on the *Hash* matrix and the choice of data structure facilitating storing and indexing vectors.

TREC optimizations. In the following, we propose two techniques for achieving space-efficient TREC. The first idea is to implement hashing in a space efficient way by re-using parts of a matrix that is already part of the DNN computation as the collection of vectors required to implement clustering. Vectors making up this matrix thus serve "double duty" as hashing vectors and matrix elements. The technique avoids huge memory overheads and achieves high speedups on real DNN computations.

For the second design, we embed a two-step stack for storing clustering ID in TREC and use a reversed index to help locate the entries in the stack. We will go through these two techniques in further depth.

5.3 Kernel Reuse

As discussed in Section 5.1, vanilla TREC increases the space usage due to extra space needed to store hashing vectors. We hereby propose *kernel reuse* as a technique to address the space issues for the *Hash* matrix on micrcocontrollers. The idea is to implement hashing by reusing parts of the weight matrix as the collection of vectors required to conduct clustering.

A closer look at the weight matrix. For a GEMM convolution, each filter slides through the input matrix as illustrated in Figure 8. There are six filters in total, and each filter is composed of one 3×3 kernel. We focus on the weight matrix W after *im2col*, where each filter is expanded to a column vector. Since there are six filters and nine elements in each filter, the weight matrix has a 9×6 size. Now, we consider the clustering step. Suppose we set $H = 3$, namely, the hashing comprises of three vectors for clustering. Immediately, we find it possible to take advantage of the first three columns of the weight matrix W and reuse them as the *Hash* matrix for clustering the rows of the X matrix. Specifically, an output matrix $Y^{n\times c}$ is obtained after $X^{n\times m}$ multiplies $W^{m\times c}$, thus the partial results from the first three columns, namely $Y^{n\times 3}$, serve double duty as matrix outputs and hashing outputs. This method avoids huge memory overheads and achieves time savings.

Detailed design. A detailed workflow of the proposed technique is illustrated in Figure 9. First, we reuse parts of the weight matrix as *Hash* matrix. Next, we multiply X with *Hash* and obtain a projection matrix. Based on this result, we project each row vector to a cluster. Specifically, for each row, i.e., a neuron vector $\mathbf{x}$, we apply $h_{\mathbf{v}_i}(\mathbf{x})$

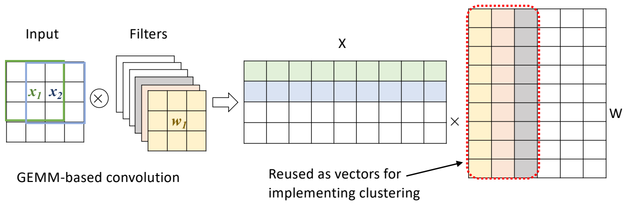


Figure 8: A closer look at the weight matrix.

for all $\mathbf{v}_i, 0 \le i < H$. In the end, the cluster ID of a neuron vector is encoded as a H bit integer $clusterId(\mathbf{x}) = (h_{\mathbf{v}_0}, h_{\mathbf{v}_1}, ...h_{\mathbf{v}_{H-1}})$. Vectors with the same cluster ID are grouped in the same cluster. Third, we compute a centroid vector for each cluster and compile them into a centroid matrix. Finally, we multiply the centroid matrix with the weight matrix W and obtain the result for each cluster.

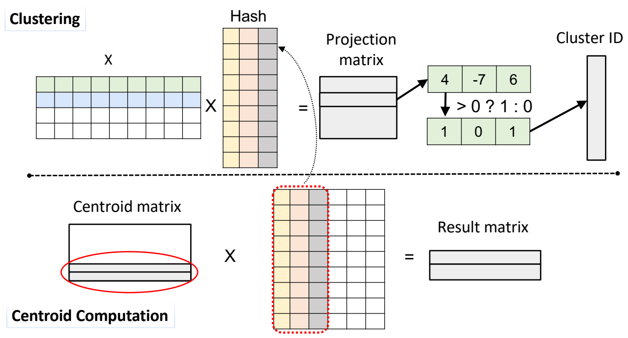


Figure 9: Illustration of kernel reuse.

Solving the dilemma for *kernel reuse* training. Training TREC with *kernel reuse* is one key challenge. *Kernel reuse* seeks ways to embed LSH vectors in the weight matrix. However, TREC requires separately updating *Hash* and the weight matrix W in each training iteration, which leads to inherent discrepancies between *Hash* and W. To get around this, we embed the LSH vectors in the weight matrix by binding *Hash* to the columns in W and sharing the gradients in each iteration. As illustrated in Figure 10, in the back propagation, instead of updating LSH vectors and the weight matrix independently, the gradients for *Hash* are copied directly from the gradients of the corresponding positions in the weight matrix. The *Hash* matrix is thus embedded in the weight matrix after each iteration.

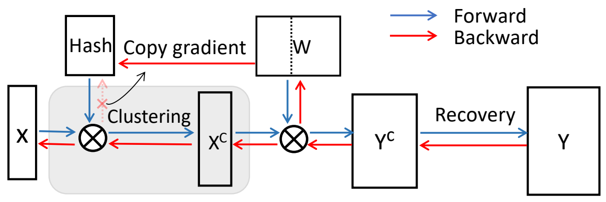


Figure 10: Illustration of *kernel reuse* training.

Maintaining properties of TREC. Another factor to note for building *kernel reuse* based TREC is to understand the impact of *kernel reuse* on the correctness, robustness, and convergentability of TREC.

Correctness. As stated above, *kernel reuse* introduces errors in clustering because gradients are directly copied from those of the weight matrix. This error is then passed on to the next forward propagation and got corrected in the back propagation. In this way, TREC is still able to exploit the similarities among the vectors in the cluster. Through training, *kernel reuse* minimizes errors while saving space.

Robustness and Convergence. Kernel reuse does not change the *Lipschitz constant* for TREC, and thus it keeps the robustness of TREC [16]. The convergence of *kernel reuse* can be proved following the same steps as those of TREC by viewing the objective function F as a function of W, namely, $F(W)$.

Space savings. *Kernel Reuse* removes the space overhead from *Hash* matrix and greatly saves space. This space saving does not sacrifice the expressive functionality of LSH to cluster vectors. This is because LSH vectors are kept to be mutually independent even by reusing parts of the weight matrix. Since filters are trained independently in the weight matrix, columns of the weight matrix are mutually independent. This guarantees that LSH vectors are independent from each other.

5.4 Two-Step Stack Substitution

General design. Another part of space overhead of TREC comes from the data structures for indexing vector entries. The space constraints on microcontrollers require efficiently storing and accessing neuron vectors in different clusters while minimizing the space occupancy. Therefore, with the help of a reversed ID table and a stack, a representative vector is stored in each cluster.

Detailed design. Suppose now we have each vector's *cluster* ID. The goal is to do clustering by grouping the vectors with the same ID together and finally get a centroid matrix containing the centroid vector for each cluster. However, storing vectors into groups according to cluster IDs and computing the centroids for each cluster is not space efficient. This is because pre-allocating fixed-size arrays for each cluster poses too much burden on the memory. To make it more space efficient, we form the centroid matrix in a streamlined way.

At any time, the space overhead is only a representative centroid vector for each cluster. These centroid vectors are stored in a stack with an index table for entry access. After finishing processing, these centroid vectors form a centroid matrix, and we can multiply it with the weight matrix. We then use the result of the centroid computation in place of the per-vector results for each vector in the cluster. A workflow sample is shown in Figure 11.

Each neuron vector is projected to a cluster ID Id. When it is the first vector with cluster ID = Id, we add this vector in the buffer stack and store a pointer to it in the *reverse ID table*. In fact, the x-entry in the reverse ID table stores the pointers, if existed, to the buffer stack for the cluster with $Id = x$. When another vector $\mathbf{v}_{new}$ with the same cluster Id arrives, we can directly find the representative centroid vector $\mathbf{v}_{old}$ in the stack and add the weight of this vector to it. Formally, if $count_x$ denotes the number of vectors in cluster x, we have

$$\mathbf{v}_{new} = \frac{\mathbf{v}_{old} \times count_x}{count_x + 1}, count_x = count_x + 1 \tag{2}$$

There are at most 2^H (Id ranges from 0 to $2^H - 1$) clusters, thus at most 2^H representative vectors in the stack. Finally, the centroid vectors form a centroid matrix.

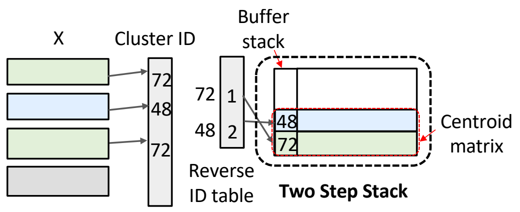


Figure 11: Stack for the representative vectors.

For each representative vector, we multiply it with the weight matrix to obtain a result vector, so that the vectors in the same cluster can reuse the result to speedup the computation. Note that the representative vectors stored in the stack actually form a matrix. Therefore, we only have to multiply the centroid matrix by the weight matrix, and the result vectors can be retrieved as the corresponding row in the result matrix.

6 IMPLEMENTATION

We discuss the general workflow of our deployment and other implementation details in this section.

6.1 Lasting Redundancy Elimination

To load a large DNN model onto Microcontrollers, the first steps are to eliminate lasting redundancy through *offline* methods so as to reduce the size of the model. These include an effective pruning to reduce the number of parameters in a model when performing inference (detailed in Section 6.1), quantization of transforming 32-bit floating point data into 8-bit integer (detailed in Section 6.1), and the use of batch norm layer folding into the convolution layer that proves efficiency and effectiveness (detailed in Section 6.1).

Parameterized model Pruning. The first and foremost method of reducing the size of a network is model pruning. An effective pruning method should first reduce the number of parameters in the network significantly and, at the same time, easy to maintain the architecture structure of the model.

Unstructured pruning. We apply PyTorch pruning [43] first and find huge potentials in reducing the size of the model. Specifically, this default pruning method masks partial elements of weight matrices thus converting these dense matrices into sparse matrices, so the network is transformed into a sparse network accordingly. That is to say, the entries with small values in the weight matrix will be set to zero. However, the sparse tensor operations are currently not supported for PyTorch. Therefore, we cannot see the acceleration effect this sparse network can achieve. Fortunately, we do benefit from such analysis; as an exploratory experiment, over 90% of the entries in a weight matrix can be masked without harming the accuracy of the whole network. This result shows huge potentials in reducing parameters, particularly the weight matrices, of the network.

Structured pruning. In this work, we identified torch-pruning as the pruning method. Torch-pruning is a widely used pruning method [2] and is special in that, instead of trying to mask certain entries in a weight matrix, it removes the whole channels from neural network for acceleration.

Network quantization. Floating point weights and activations are used to train neural networks. Previous research [32] has shown that fixed-point weights are enough for executing neural networks with minimum accuracy loss. This reduces weights from 32 bit to 8 bit, thus reducing the model size, and is good for microcontroller devices deployment. Furthermore, fixed-point integer operations in common microcontrollers are substantially faster than non-fixed point operation. This makes another justification for executing quantized models on microcontrollers.

Method. Assuming a fixed-point format and scaling by twos, the represented value is $A \times 2^n$, where A denotes the integer value and n is the radix point's location. Due to the power-of-two scaling, the scaling factors for the data (e.g., outputs or bias) are stored in the network as parameters. Because CM Core CPU can lack a floating-point unit (FPU), we utilize this fixed-point format rather than TensorFlows's method for 8-bit quantization.

Batch norm layer folding. When we apply quantization, problems arise for *batchnorm* layer, since the original floating point means that μ and standard deviation σ are required. To solve this problem, we fold batch norm into the convolution when performing inference for the batch norm layers in DNNs.

In batch norm folding, we replace the convolution followed by a batch normalization with just one convolution with different weights. This would remove the precision loss if we quantize μ and σ from 32 bit floating point to 8 bit integer. Furthermore, it reduces the number of operations to be performed at inference time, thereby speeding up the entire network.

Specifically, the convolution operator followed by a batch normalization can be expressed, for an input $\mathbf{x}$, as:

$$\begin{aligned} \mathbf{z} &= W * \mathbf{x} + \mathbf{b} \\ y_i &= \gamma \cdot \frac{z_i - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta \end{aligned} \tag{3}$$

When we rearrange the value of W and $\mathbf{b}$ such that:

$$\begin{aligned} W_{\text{folded}} &= \gamma \cdot \frac{W}{\sqrt{\sigma^2 + \epsilon}} \\ b_{\text{folded}} &= \gamma \cdot \frac{b - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta \end{aligned} \tag{4}$$

We manage to remove the whole batch norm layer and obtain the same result with only one convolution.

6.2 Meeting Accuracy Requirements

The input matrix X is divided into three smaller matrices and partial results are computed as $\mathbf{y}^{(1)}, \mathbf{y}^{(2)}$, and $\mathbf{y}^{(3)}$. For the normal full neuron vector granularity in microcontrollers, $\mathbf{y}^{(\mathbf{i})}$ is truncated and stored in an 8 bit integer. With smaller granularity, all $\mathbf{y}^{(\mathbf{i})}$ should be kept in 32-bit containers when the result $\mathbf{y} = \mathbf{y}^{(1)} + \mathbf{y}^{(2)} + \mathbf{y}^{(3)}$ is added up with all these partial results. After that, $\mathbf{y}$ gets scaled and truncated.

6.3 Implementing TREC

TREC design is at the core of our work in enabling efficient DNNs on microcontrollers. We need to pay special attention on the deployment of TREC, hardware architecture of the microcontrollers, and how the constraint space limits affect these above. We next explain each of these aspects.

The recover step in TREC. After computing the clustering results (i.e., partial results for each cluster), the final step is to recover the result matrix from the partial results we just obtained. As shown in Figure 12, the details are described as follows. We iterate each neuron vector and obtain their *clusterId*. Using this cluster *Id*, we can find the position of the representative vector of this cluster in the stack, thus obtaining the corresponding partial result vector. We retrieve this partial result vector and fill it in the result matrix. After going through all of the neuron vector, we finally manage to reconstruct the result matrix.

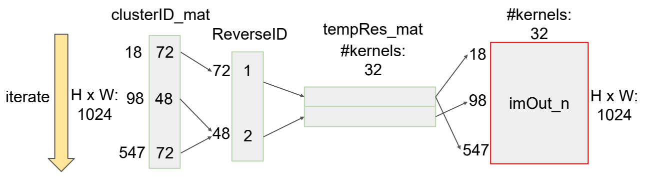


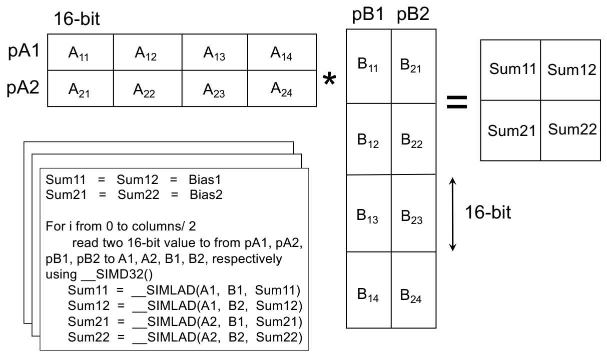
Figure 12: Recovery from the partial results.

Use of SIMD kernels. The 32-bit RISC processing cores in the CM (Cortex-M) series of processors have been energy-efficiency optimized. They are part of the ARM Cortex-M architecture and used for applications on microcontrollers [49]. In our work, we enable massive DNNs on CM-based platforms that have SIMD instructions, especially 16-bit Multiply-and-Accumulate (MAC) instructions (e.g. SMLAD). These are very helpful for running DNN.

In the Cortex-M processor, each word, namely 32 bit, can store four integers. Since we use the 16-bit MAC operators, we have to transform the quantized 8-bit data into 16-bit ones. These four 8-bit integer have to be first extended to 16-bit data type, and then reordered to abide with their original order stored in one word.

The most significant operator in the entire network is matrix multiplication. In particular, we have matrix multiplications when performing projection and obtaining partial results in reuse computation. Therefore, we pay special attention on optimizing the performance of matrix multiplication.

As depicted in Figure 13, the matrix multiplication kernel is accomplished with an emphasis on a 2x2 unit each time. This allows data reuse while also reducing the overall amount of load instructions. The MAC instruction __SMLAD is used to execute the computation. The accumulation is carried out using a 32-bit data type, and both operands are of the 16-bit data type. A bias value is assigned for each accumulator as the initial value.



```
Sum11  =  Sum12  =  Bias1
Sum21  =  Sum22  =  Bias2

For i from 0 to columns/ 2
      read two 16-bit value to from pA1, pA2,
pB1, pB2 to A1, A2, B1, B2, respectively
using __SIMD32()
     Sum11  =  __SIMLAD(A1,  B1,  Sum11)
     Sum12  =  __SIMLAD(A1,  B2,  Sum12)
     Sum21  =  __SIMLAD(A2,  B1,  Sum21)
     Sum22  =  __SIMLAD(A2,  B2,  Sum22)
```

Figure 13: The demonstration of the inner loop of matrix multiplication. Each time two columns and two rows are taken into consideration.

Compared to the original matrix multiplication kernel provided by Cortex-M library, this solution exhibits two major advantages. First, the original kernel requires matrix transpose to conduct multiplication, which does great harm to space locality. Second, the use of SIMD kernels brings about more efficient computations. Experiments show that our solution achieves good results. For instance, for projection in reuse time, we have 1.67× speedup compared to the original matrix multiplication of the Cortex-M library.

Other hardware characterization. We next describe other hardware characterizations of microcontrollers.

Methodology for placing weights and intermediate activations. For microcontrollers running DNNs, SRAM is used to allocate activation buffers, while Flash memory is used to allocate model weights and biases, as well as the network definition. Alternatively, weights can be stored in SRAM. However, we find that storing weights in Flash only results in a slight increase in end-to-end latency, roughly 1%. This can bring huge benefits because storing weights in Flash saves the limited space usage for activations, which can be placed in SRAM. Storing intermediate activations in Flash, on the other hand, renders a major drop-off in performance (over 20% increase in latency for a layer to store activation maps on the Flash in the current CifarNet setting through an explorative experiment). This is because of the need for block writes, rather than byte writes in Flash. Specifically, convolution layers write the output data to the Flash memory where the next layer reads and processes data. For efficient processing, studies are needed on modifications of kernel implementations, design, and testing of performance models.

Model latency. We next examine the hardware performance of the reuse-centered NN layers. Lai et al. [34] pointed out that, for typical NN layers, latency and energy consumption increases linearly in accord with the operation counts. This model, called latency/energy model, shows that the measured latency can be a function of the number of operations required. In contrast, the reuse-centered layers are able to reduce the total number of operations, thus decreasing the latency of the model.

7 EVALUATION

7.1 Experimental Setup

Methodology. To demonstrate the efficacy of TREC in enabling large DNN inference on microcontrollers, we first apply our approach to the entire DNN and measure end-to-end inference performance. Second, we apply TREC and measure the single-layer acceleration and accuracy for individual layers (specifically the densely-computed layers like convolutional layers). Third, we focus on the space savings of our method. Specifically, how much space is saved from the original DNN models with *offline* methods including pruning, quantization, and layer folding. Since we have shown the TREC space overhead for the three networks in Table 1, we examine the amount of space savings from the two ideas, i.e., *kernel reuse* and *two-step stack*. Fourth, we analyze the performance impact of the number of layers when applying *kernel reuse*. Fifth, we explore TREC performance on sparse inputs. For all the measurements, the SRAM and Flash occupancy is determined using the Mbed compiler (Mbed OS) and the microcontroller latency is measured using the Mbed Timer API.

Platform. DNN inference is performed on the STM32F469I and STM32F746ZG microcontroller featuring SIMD extensions. The STM32F469I platform comes with 324KB SRAM and 2048KB Flash as memory storage while the STM32F746ZG board has 320KB SRAM and 1024KB Flash. We also deploy CMSIS-NN kernels for optimized performance on the Cortex-M4 and Cortex-M7 CPU equipped on the STM32 board [33]. In addition, DNN training is performed on a server equipped by a 20-core 3.60GHz Intel Core i7-12700K CPU with 128GB RAM and an NVIDIA GeForce RTX A6000 GPU with 48 GB memory. We use PyTorch 1.10.1 (open-source software with a BSD license) as our frame for training.

Workloads. We evaluate our approach with the most popular compact DNNs that fit microcontrollers, namely CifarNet [1], ZfNet [62], and SqueezeNet, with and without complex bypass [22]. In all cases, we use the public dataset CIFAR-10 [29] for training and evaluation. All networks are optimized by SGD. The learning rate starts from 0.001 and decreases by 0.1 at 15- epochs intervals. The batch size, weight decay, and momentum are set to 256, 0.9, and 10^{-4}, respectively, and the maximum number of training iterations is set to 100.

7.2 Performance

we compare performance of TREC-equipped DNNs to conventional Convolutional DNNs, *deep reuse* featured DNNs, and LCNNs. Note that after channel pruning, the baseline accuracy of ZfNet is slightly lower than its standard accuracy of 83%, but the model size is 7.6% of the original model, which can well meet the memory constraint of the microcontrollers. Read-only data can be placed in flash memory for microcontroller storage to hold the entire network.

The conventional convolution implementation is from the CMSIS-NN library [33] that achieves the state-of-the-art performance for convolutional neural networks on edge devices. For the datasets, inputs are first converted from floating point representation to 8-bit integers. This implementation is also from the CMSIS-NN library. We compare TREC with LCNN. LCNN stores only a dictionary (which can be regarded as a subset of weight vectors) for reconstructing weights so as to save space and computations. This strategy is actually complementary to TREC. In particular, when we perform convolution in LCNN, the input tensor undergoes a matrix multiplication with dictionary vectors, where our LSH approach can fit in and speed up this matrix multiplication. Through experiments, as shown in Table 2, we find that LCNN experiences slowdown for CifarNet. This is because LCNN requires multiple memory accesses to the dictionary for reconstructing the original weight matrix, which are not suitable for microcontroller applications.

Table 2: Comparison of CMSIS-NN Convolution, LCNN, LCNN with TREC, and TREC for the CifarNet.

	Conv1 (ms)	Conv2 (ms)	Conv1 + Conv2 (ms)
CMSIS Convolution	96.64	57.4	154.69
LCNN	103.51	59.1	162.69
LCNN + TREC	93	49.46	143.18
TREC	53.4	37.76	91.35

For comparison, we also include the results from *deep reuse*, as this state-of-the-art method avoids transient redundancy in DNNs. Given the inherent randomness of *deep reuse*, we present its results in 150-run intervals. Note that *deep reuse* requires storing LSH vectors and index structures on microcontroller memory, which exceeds the memory capacity (denoted as "—" in Table 3 for SqueezeNet and SqueezeNet (Bypass)). For TREC-equipped DNNs, all the space-saving techniques are applied to the model. In Table 3, we have the following findings. First, compared to the conventional convolution (by CMSIS-NN library), TREC achieves 3.61× speedup on average. The same amount of speedups are observed on both microcontrollers, with STM32F746ZG's total end-to-end inference time half of that measured on STM32F469NI. The reason is that Cortex-M7 has a higher clock rate and can issue data loading and ALU instructions at the same time.

Second, TREC experiences only around 0.7% accuracy loss. *Deep reuse* [40] is subject to its inherent randomness, with accuracy loss reaching up to 7.6%. TREC is hence better in terms of stability and accuracy. Since both TREC and *deep reuse* exploit *transient redundancy* elimination, they have comparable inference time. The optimizations in TREC make it however more space efficient. It enables large DNNs (i.e., vanilla SqueezeNet and SqueezeNet with bypass) to run on microcontrollers, while *deep reuse* does not.

Third, when looking into the effects TREC bring to different networks, we find the most speedups in SqueezeNet. This is because there are more computation-intensive convolution layers in SqueezeNet. The more kernel channels are in a convolution layer, the more redundant computations can be detected and eliminated. We have a more detailed analysis in the following section.

7.3 Single Layer Speedup

We run experiments on each single convolutional layer with a different range of clustering configuration and collect the redundancy ratio. For the purpose of study, we find the configurations that can reduce the maximum amount of computations without threatening inference accuracy. Latencies are collected from the STM32F469NI board.

Table 3: End-to-end performance and accuracy comparison.

Target(s)	Convolution Methods	CifarNet	ZfNet	SqueezeNet	SqueezeNet (Bypass)
STM32F469NI	Latency (ms)-CMSIS Conv	217.32	3557.32	1639.51	1998.86
	Latency (ms)-*Deep reuse*	154.44	814.03	—	—
	Latency (ms)-TREC	153.92	814.01	327.9	543.71
	Speedup-TREC vs. CMSIS Conv.	1.412×	4.37×	4.98×	3.68×
STM32F746ZG	Latency (ms)-CMSIS Conv	120.62	1758.73	894.16	1152.46
	Latency (ms)-*Deep reuse*	98.44	525.03	—	—
	Latency (ms)-TREC	97.79	524.3	181.49	274.2
	Speedup-TREC vs. CMSIS Conv.	1.23×	3.35×	4.93×	4.20×
Both	Accuracy (%)-CMSIS Conv.	78.2	80.1	83.5	85.6
	Accuracy (%)-*Deep reuse*	73.2 ~ 76.1	72.5 ~ 76.6	79.8 ~ 81.9	80.5 ~ 83.1
	Accuracy (%)-TREC	76.5	78.9	83	85.3

We list in Table 4 the single-layer speedups and accuracy losses attained by substituting TREC for the traditional convolution. The following observations have been made. First, TREC finds and removes approximately 96.22% of the *transient redundancy* indicated by r_t. This is where the speedup comes from.

Second, with an average speedup of 4.28×, TREC significantly improves the performance of each convolution layer. Specifically, TREC speeds up SqueezeNet expand layers by up to 18.66×, showing that computation-intensive layers can particularly benefit from TREC. However, due to the additional cost from clustering, the speedup is smaller than what is suggested by the ratio of *transient redundancy* that TREC reduces.

Third, there is a limited and mixed impact of TREC on accuracy. It may result in a 0.8% accuracy decrease when applied to certain layers, but it may also result in a modest accuracy improvement when applied to some other layers. Overall, TREC keeps the accuracy much better than *deep reuse* does (around 3%-7% drop). The single-layer results show that TREC can operate at maximum efficiency, achieving notable speedups with the least amount of accuracy loss.

7.4 Space Saving Analysis

There are static space and dynamic space required for storing models. Specifically, static space refers to the space for weights and biases (from the original model). Dynamic space contains TREC space, namely, *Hash* matrix and index structures. The baseline size for original static and dynamic space occupancy is detailed in Table 5.

For static space, we apply existing techniques to remove *lasting redundancy*. For dynamic space, we use *kernel reuse* and *two-step stack* to eliminate *transient redundancy*. Table 5 summarizes how much space is saved for the original model size and TREC size, respectively. For static space, since there is no Batch norm layer in CifarNet Overall, the original model size witnesses an average of over 90% reduction as a result of these *lasting redundancy elimination* based methods. For dynamic space, we report the space reduction for *kernel reuse* with configurations where there is virtually no accuracy loss (< 0.1%), and provide a detailed analysis in Section 7.5. The dynamic space reduction exceeds 70% when we combine *kernel reuse* and *two-step stack*.

7.5 Detailed Analysis

Kernel reuse for reaching accuracy stability. *Kernel reuse* embeds the *Hash* matrix in the weight matrix W. Figure 14 shows how *kernel reuse* influences SqueezeNet in terms of both accuracy and space occupancy on the STM32F7 board. The x axis denotes the number of layers we apply *kernel reuse* for SqueezeNet. We first look at the accuracy. For comparison, we also include the accuracy of *deep reuse* at 150 runs. *Deep reuse* has random LSH vectors, and its accuracy varies between 80% ~ 83%. Accordingly, the two green lines show the accuracy range that *deep reuse* produces. In contrast, *kernel reuse* produces stable accuracy results up to 85.6%. Although there is accuracy loss as x increases, the network accuracy drops slowly and remains high (> 84%) for $x < 16$.

Second, *kernel reuse* brings huge space benefits. For example, when *kernel reuse* is in five layers, namely, $x = 5$, the space overhead is cut by over 50% with accuracy loss $\delta < 0.5\%$. That is to say, for large DNNs, applying *kernel reuse* to a small number of layers can dramatically reduce the total space overhead with marginal accuracy loss.

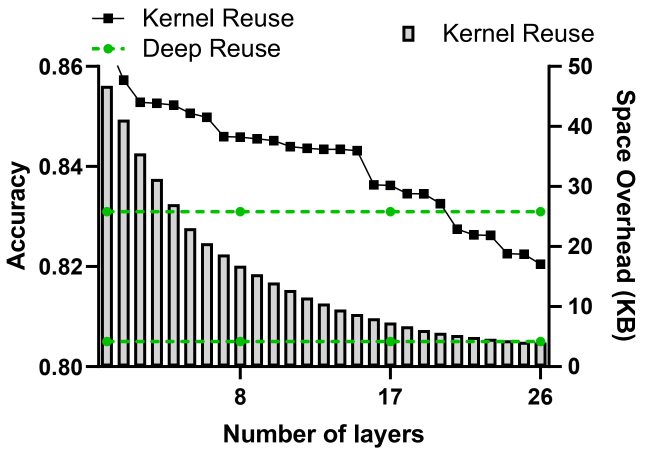


Figure 14: Influence of accuracy and space overhead from the number of layers applying kernel reuse.

Table 4: Single-layer performance benefits (TREC latency in ms). $Conf.$ **means configuration,** L **is the width of sub-matrices,** H **is the number of hash functions.** K **is the kernel size and** M **is the number of kernel channels.** r_t **represents the redundancy ratio, and** Δ**Acc is the accuracy difference between TREC and standard convolution.**

(a) Single-layer performance for CifarNet.

ConvLayer	K	M	Conf.		Latency	r_t	Speedup	ΔAcc
			L	H				
Conv1	75	64	15	5	53.4	0.943	1.81×	-0.0132
Conv2	1600	64	10	10	37.76	0.795	1.52×	-0.0077
Avg.						0.869	1.66×	-0.0105

(b) Single-layer performance for ZfNet.

ConvLayer	K	M	Conf.		Latency	r_t	Speedup	ΔAcc
			L	H				
Conv1	147	96	147	5	0.75	0.999	1.22×	-0.0011
Conv2	2400	256	300	5	45.63	0.997	4.69×	-0.0037
Conv3	2304	384	384	5	61.80	0.988	4.72×	-0.0076
Conv4	3456	384	432	5	69.97	0.998	6.23×	-0.0105
Conv5	3456	256	288	5	52.33	0.984	5.58×	-0.0068
Avg.						0.994	4.49×	-0.0059

(c) Single-layer performance for SqueezeNet.

ConvLayer	K	M	Conf.		Latency	r_t	Speedup	ΔAcc
			L	H				
Conv1	27	96	9	5	6.89	0.997	1.27×	0.0122
Fire2.squeeze.conv	96	16	96	4	19.74	0.993	1.07×	0.0144
Fire2.expand_1x1.conv	16	64	8	5	25.12	0.991	1.41×	0.0023
Fire2.expand_3x3.conv	144	64	48	5	48.79	0.990	5.90×	0.0002
Bypass1	96	128	32	5	28.58	0.994	2.45×	0.0071
Fire3.squeeze.conv	128	16	64	5	8.77	0.988	1.07×	-0.0051
Fire3.expand_1x1.conv	16	64	8	5	15.41	0.992	2.30×	-0.0033
Fire3.expand_3x3.conv	144	64	24	5	50.33	0.988	5.85×	-0.0013
Fire4.squeeze.conv	128	32	64	5	8.71	0.990	1.00×	-0.004
Fire4.expand_1x1.conv	128	128	4	8	12.07	0.988	1.45×	-0.0037
Fire4.expand_3x3.conv	32	128	16	5	22.98	0.988	6.36×	0.0012
Bypass2	288	256	48	5	26.47	0.993	2.53×	-0.0012
Fire5.squeeze.conv	256	32	64	4	2.46	0.982	1.84×	-0.0036
Fire5.expand_1x1.conv	32	128	16	5	3.89	0.953	4.51×	0.0016
Fire5.expand_3x3.conv	288	128	48	5	12.35	0.954	11.83×	0.0001
Fire6.squeeze.conv	256	48	128	5	6.62	0.972	1.01×	-0.0018
Fire6.expand_1x1.conv	256	192	32	5	5.97	0.963	4.40×	-0.0071
Fire6.expand_3x3.conv	48	192	32	5	17.48	0.953	12.50×	0.0002
Bypass3	432	384	72	5	9.88	0.977	5.29×	0.0021
Fire7.squeeze.conv	384	48	48	5	4.26	0.965	1.71×	0.0003
Fire7.expand_1x1.conv	48	192	24	5	6.57	0.957	4.00×	-0.0046
Fire7.expand_3x3.conv	432	192	48	5	23.35	0.953	9.36×	-0.0025
Fire8.squeeze.conv	192	64	64	4	19.01	0.969	1.03×	-0.0003
Fire8.expand_1x1.conv	384	256	128	5	16.25	0.965	2.15×	0.0012
Fire8.expand_3x3.conv	64	256	16	5	38.47	0.955	7.57×	-0.0022
Bypass4	576	512	64	5	62.08	0.961	1.12×	-0.0045
Fire9.squeeze.conv	512	64	128	4	2.19	0.894	1.01×	-0.0078
Fire9.expand_1x1.conv	64	256	16	5	5.7	0.839	1.53×	-0.0019
Fire9.expand_3x3.conv	576	256	96	5	6.25	0.841	11.64×	-0.0022
Conv10	512	10	4	5	2.1	0.944	1.29×	0.0057
Avg. (w/ bypass)						0.963	3.88×	0.0003
Avg. (w/o bypass)						0.963	4.89×	-0.0006

Table 5: Space savings by different techniques.

	Technique	CifarNet	ZfNet	SqueezeNet
Static Space	Baseline Size	420 KB	4 MB	5 MB
	Channel Pruning	45.1%	92.4%	90.8%
	CP + Q	86.8%	97.9%	97.4%
	CP + Q + LF	86.8%	98.9%	98.2%
Dynamic Space	Baseline Size	20 KB	40KB	56 KB
	Kernel Reuse	41.2%	68 .2%	71.9%
	KR + TSS	60.2%	73.4%	77.3%

CP: Channel Pruning, Q: Quantization, LF: Layer Foldings,
KR: Kernel Reuse, TSS: Two-Step Stack

8 CONCLUSION

In this work, we demonstrate that large DNNs can be deployed on resource-constrained microcontrollers and executed efficiently by the introduction of space-efficient TREC. It takes advantages of both computation reduction and space savings that make the best ability of the resource utilization of microcontrollers. As a result, the approach maximizes the performance benefits and obtain stable accuracy results. In the evaluation, our solution achieves 3.5×-5× speedups with virtually no accuracy loss.

ACKNOWLEDGMENTS

This material is supported by the National Natural Science Foundation of China (No. 62172419, 62072458, and 61732014) and Beijing Nova Program. Jiesong Liu, Feng Zhang, Jiawei Guan, Xiaoguang Guo, Xiaoyong Du are with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), and School of Information, Renmin University of China. Feng Zhang is the corresponding author of this paper.

A APPENDIX FOR SPARSE INPUT MATRICES: CONVERTING SPARSE CONVOLUTION TO DENSE CONVOLUTION

TREC is compatible with sparse input matrices through *sparse convolution*. Although this property is not directly used in this work, it could be an important property to exploit when a user would like to combine TREC with DNN sparse pruning or other optimizations. We explain this property in this part to offer a complete understanding of TREC.

One typical case where sparse matrix shows up is when weight pruning is used. We will draw on PatDNN [41] as our example. PatDNN prunes a DNN kernel via patterns. The left graph in Figure 15 shows a 4-element pattern left after a 3x3 kernel is pruned. In PatDNN, the sparse weight matrix is encoded in a Filter-Kernel-Weight (FKW) format [41]. When a convolution kernel slides through an image or activation map, only the pixels within the pattern's frame contribute to the output. In the example in Figure 15, only four elements in the input are used in the dot-product with the kernel in each step. TREC can be applied by regarding the four elements used in each step as a vector for clustering. If two of such vectors are similar and fall into the same cluster, TREC avoids the repeated dot-product calculations.

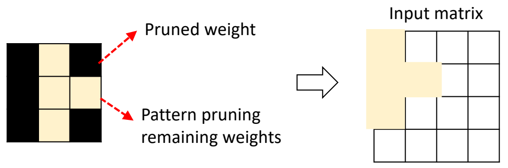


Figure 15: Illustration of a sparse pattern kernel.

We next look at how TREC can be applied if the input matrix is sparse. We discuss it in two scenarios. In the first scenario, the original algorithm simply treats the sparse input in the same way as it treats a dense input. Clearly, TREC can be applied as we have already described, and because there are many zero vectors, TREC would cluster them together and give significant speedups. In the second scenario, the original algorithm already tries to do calculations only on the non-zero parts of the input. *Sparse convolution reorganization* is one way to do that [15, 35, 57]. It groups non-zero elements together to form a dense matrix. This matrix is then multiplied with reorganized kernel matrices. In this way, it avoids the inefficiency of frequent irregular accesses. TREC can be applied to the matrix after the reorganization.

As Figure 16 illustrates, we have a 5×5 image with three channels. All the pixels are (0,0,0) except three points P_1, P_2, and P_3. The convolution kernel of sparse convolution is the same as traditional convolution. Each kernel size is 3×3. We can distinguish three filters according to the light and dark colors. In this example, we use convolution with the stride of one and padding of zero. There are three filters. Accordingly, the convolution will produce three 3×3 output activation maps (which are not displayed in the figures).

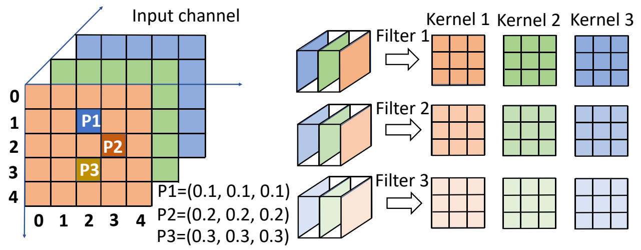


Figure 16: Illustration of input model and kernels.

We label the nine kernels as $kernel[k]_{i,j}, 1 \leq k \leq 9$. Considering the specific convolution, for each of the 3×3 kernel position $(i, j), 0 \leq i, j < 3$, we collect all the atomic operations for P_1, P_2, and P_3 with respect to $kernel[k]_{i,j}, 1 \leq k \leq 9$. For the kernel position $(1, 1)$, namely, F_4, we collect P_1, P_2, and P_3 as a reorganized matrix and compute them with F_4 together, as shown in Figure 17. For the kernel position F_0, the only generated operations are with P_1. Likewise, for the kernel position F_4, the generated operations are with P_1, P_2, and P_3.

Since there are three channels and three filters, we have a total of nine kernels. Therefore, for each of the kernel position $(i, j), 0 \leq i, j < 3$, the elements corresponding to (i, j) can form a 3×3 matrix. Accordingly, the direct convolution is constructed as a matrix multiplication. To this end, we can convert sparse convolution into a dense matrix multiplication through computing reorganization.

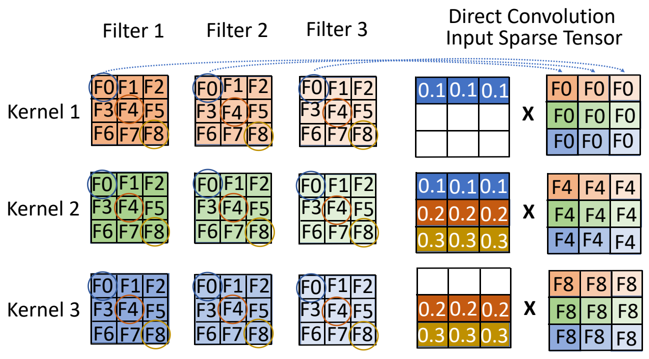


Figure 17: Direct convolution through dense matrix multiplication.

This matrix multiplication can be accelerated by incorporating LSH-based clustering.

We provide some experimental results on Cifarnet. Table 6 shows the performance for SparseConv and TREC. Input sparsity is set to 0.1 and batch size is 10. The speedup are 3.38× and 4.83× for the two convolution layers, respectively.

Table 6: Performance for sparse input matrix.

ConvLayer	Output Channels	Latency (ms) SparseConv	Latency (ms) TREC	Speedup
Conv1	32	21.98	6.51	3.38x
Conv2	64	42.96	8.89	4.83x

B ARTIFACT APPENDIX

B.1 Abstract

Space efficient TREC is a new form of convolution optimized for microcontrollers. It makes *trainsient redundancy* detection and avoidance an inherent part of the DNN architecture, and the determination of the best configurations for redundancy elimination part of DNN backward propagation.

TREC is currently implemented as a new lightweight high-level API of Pytorch for defining, training and evaluating complex models. This directory contains code for training and evaluating several compact Convolutional Neural Networks (CNNs) using TREC.

It contains scripts that will allow you to train models from scratch and evaluate models on both server and Microcontrollers.

B.2 Artifact Check-List (Meta-Information)

- **Program:** Arm Mbed CLI 1.10.5
- **Compilation:** Python 3.6, GNU Arm Embedded Toolchain 10.3
- **Data set:** Cifar-10
- **Run-time environment:** Ubuntu 20.04
- **Hardware:** An NVIDIA GeForce RTX A6000 GPU server with 20-core 3.60GHz Intel Core i7-12700K processor, 128GB of RAM, and 48GB of GPU memory.
  An STM32F469NI MCU with 324KB SRAM and 2MB Flash.

- **Experiments:** Benchmark: CMSIS-NN, Deep Reuse for CNNs on Microcontrollers
- **How much disk space required (approximately)?:** Less than 400MB.
- **How much time is needed to prepare workflow (approximately)?:** Around 30 minutes.
- **How much time is needed to complete experiments (approximately)?:** About five hours to train a model from scratch, and about five minutes for model inference only.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** MIT License.
- **Workflow framework used?:** No, but scripts are provided for automate training. Detailed workflow is in readme.md in the zipped file.
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.7702231

B.3 Description

B.3.1 How To Access. The artifact, which is zipped into one file, is available on Zenodo: https://doi.org/10.5281/zenodo.7702231. Source code, scripts, and instructions are zipped into TREC.zip.

B.3.2 Hardware Dependencies. For the training machine, we have An NVIDIA GeForce RTX A6000 GPU server with 20-core 3.60GHz Intel Core i7-12700K processor, 128GB of RAM, and 48GB of GPU memory. We use Pytorch-1.10.1 (open-source software with a BSD license).

For the deployment machine, users can use an STM32F469NI Microcontroller with 324KB SRAM and 2MB Flash.

B.3.3 Software Dependencies. For the embedded deep learning library, we adapted from CMSIS-NN kernel optimized for Arm Cortex-M devices. A more detailed instruction is included in the readme.md.

B.4 Installation

To use TREC on the server end, installing PyTorch is a prerequisite (available at the official website). TREC requires Python version 3.6 or later, and recommends PyTorch version torch==1.10.1+cu111. Then, users can install the TREC package by simply running:

```
python setup.py install
```

After a few minutes, TREC will exist in the user environment as a PyTorch extension package named *trec*. Now users can use TREC by importing both *torch* and *trec* packages in Python.

On the Microcontrollers end, users need to get the CMSIS-NN library and install mbed-cli. Specifically, first, clone the CMSIS-5 library, which consists of the optimized neural network kernels for Cortex-M. This can be done by running the following:

```
cd MCU_eval
git clone https://github.com/\
ARM-software/CMSIS_5.git
```

To install mbed-cli(available at this link) and its python dependencies, use:

```
pip install mbed-cli
```

B.5 Experiment Workflow

Training a model from scratch.

We provide an easy way to train a model from scratch using Cifar-10 dataset. The following example demonstrates how to train SqueezeNet using the default parameters.

```
TRAIN_DIR=/tmp/TREC/examples/EXP
DATASET_DIR=/tmp/TREC/data
python train_model.py \
    --checkpoint_path=${TRAIN_DIR} \
    --dataset_path=${DATASET_DIR} \
    --model_name=SqueezeNet
```

For simplicity, several scripts for training are put in the examples/scrips/ directory. Users can start training by simply executing:

```
cd examples/scrips/
bash train_squeeze_on_cifar10_template.sh
```

Pre-trained models of CifarNet and SqueezeNet (with and without bypass) under directory examples/pre_trained_models/.

Evaluating on Microcontrollers.

The first step in deploying the trained models on microcontrollers is quantization, which is described here https://github.com/ARM-software/ML-KWS-for-MCU/blob/master/Deployment/Quant_guide.md. This directory consists of example codes and steps for running a quantized DNN model on any Cortex-M board using mbed-cli and CMSIS-NN library. It also consists of an example of integration of the TREC model onto a Cortex-M development board to demonstrate real time inference on live streaming data.

We have our image data loaded to the 'camera_with_nn.cpp' file in this example, so inference is run on this input data. First, create a new project and install any python dependencies prompted when the project is created for the first time after the installation of mbed-cli.

```
mbed new trec --mbedlib
```

Then, fetch the required mbed libraries for compilation:

```
cd trec
mbed deploy
```

Now, users can have MCU board (in our case, it's DISCO_F469NI) connected to their computer. Then, compile and run the code for the mbed board. The inference time will show up on the screen of the MCU board.

```
mbed compile -t GCC_ARM -m DISCO_F469NI
--source .\
--source ../squeeze_complex_bypass
--source ../\
CMSIS_5/CMSIS/NN/Include --source ../\
CMSIS_5/CMSIS/NN/Source --source ../CMSIS_5/\
CMSIS/Core/Include --source ../CMSIS_5/\
CMSIS/DSP/Include --source ../CMSIS_5/\
CMSIS/DSP/Source --source ../\
CMSIS_5/CMSIS/DSP/PrivateInclude -j8\
--flash --sterm
```

B.6 Evaluation and Expected Results

Targets	CifarNet	SqueezeNet	SqueezeNet (Bypass)
STM32F469NI	153.92 ms	327.9 ms	543.71 ms
STM32F746ZG	97.79 ms	181.49 ms	274.2 ms
Accuracy	76.5%	83%	85.3%

REFERENCES

[1] 2020. CifarNet. http://places.csail.mit.edu/deepscene/small-projects/TRN-pytorch-pose/model_zoo/models/slim/nets/cifarnet.py.
[2] Peter Bajcsy and Michael Majurski. 2021. Baseline Pruning-Based Approach to Trojan Detection in Neural Networks. *arXiv preprint arXiv:2101.12016* (2021).
[3] Colby Banbury, Chuteng Zhou, Igor Fedorov, Ramon Matas, Urmish Thakker, Dibakar Gope, Vijay Janapa Reddi, Matthew Mattina, and Paul Whatmough. 2021. Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers. *Proceedings of Machine Learning and Systems* 3 (2021), 517–532.
[4] Jesús Benito-Picazo, Enrique Domínguez, Esteban J Palomo, Ezequiel López-Rubio, and Juan Miguel Ortiz-de Lazcano-Lobato. 2018. Deep learning-based anomalous object detection system powered by microcontroller for PTZ cameras. In *2018 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–7.
[5] Neel Bhave, Aniket Dhagavkar, Kalpesh Dhande, Monis Bana, and Jyoti Joshi. 2019. Smart Signal–Adaptive Traffic Signal Control using Reinforcement Learning and Object Detection. In *2019 Third International conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud)(I-SMAC)*. IEEE, 624–628.
[6] Dimosthenis E Bolanakis. 2019. A survey of research in microcontroller education. *IEEE Revista Iberoamericana de Tecnologias del Aprendizaje* 14, 2 (2019), 50–57.
[7] Gianmarco Cerutti, Renzo Andri, Lukas Cavigelli, Elisabetta Farella, Michele Magno, and Luca Benini. 2020. Sound event detection with binary neural networks on tightly power-constrained IoT devices. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*. 19–24.
[8] Beidi Chen, Zichang Liu, Binghui Peng, Zhaozhuo Xu, Jonathan Lingjie Li, Tri Dao, Zhao Song, Anshumali Shrivastava, and Christopher Re. 2021. {MONGOOSE}: A Learnable {LSH} Framework for Efficient Neural Network Training. In *International Conference on Learning Representations*. https://openreview.net/forum?id=wWK7yXkULyh
[9] Arm Company. 2010. Cortex®-M4 Technical Reference Manual. https://users.ece.utexas.edu/~valvano/EE345L/Labs/Fall2011/CortexM4_TRM_r0p1.pdf
[10] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang, et al. 2021. TensorFlow lite micro: Embedded machine learning for tinyml systems. *Proceedings of Machine Learning and Systems* 3 (2021), 800–811.
[11] Amir Erfan Eshratifar, Amirhossein Esmaili, and Massoud Pedram. 2019. Bottlenet: A deep learning architecture for intelligent mobile cloud computing services. In *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, 1–6.
[12] Derek Farren, Thai Pham, and Marco Alban-Hidalgo. 2016. Low latency anomaly detection and Bayesian network prediction of anomaly likelihood. *arXiv preprint arXiv:1611.03898* (2016).
[13] Igor Fedorov, Ryan P Adams, Matthew Mattina, and Paul Whatmough. 2019. Sparse: Sparse architecture search for cnns on resource-constrained microcontrollers. *Advances in Neural Information Processing Systems* 32 (2019).
[14] Igor Fedorov, Ryan P Adams, Matthew Mattina, and Paul Whatmough. 2019. Sparse: Sparse architecture search for cnns on resource-constrained microcontrollers. *Advances in Neural Information Processing Systems* 32 (2019).
[15] Benjamin Graham, Martin Engelcke, and Laurens Van Der Maaten. 2018. 3d semantic segmentation with submanifold sparse convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 9224–9232.
[16] Jiawei Guan, Feng Zhang, Jiesong Liu, Hsin-Hsuan Sung, Ruofan Wu, Xiaoyong Du, and Xipeng Shen. 2022. TREC: Transient Redundancy Elimination-based Convolution. In *Neural Information Processing Systems 35 (Neurips 2022)*.
[17] Chirag Gupta, Arun Sai Suggala, Ankit Goyal, Harsha Vardhan Simhadri, Bhargavi Paranjape, Ashish Kumar, Saurabh Goyal, Raghavendra Udupa, Manik Varma, and Prateek Jain. 2017. Protonn: Compressed and accurate knn for resource-scarce devices. In *International Conference on Machine Learning*. PMLR, 1331–1340.
[18] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).
[19] Bian Haoqiong, Sha Tiannan, and Anastasia Ailamaki. 2023. Using Cloud Functions as Accelerator for Elastic Data Analytics. In *SIGMOD*.
[20] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network (2015). *arXiv preprint arXiv:1503.02531* 2 (2015).
[21] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).
[22] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).
[23] Sunil Jacob, Varun G Menon, Fadi Al-Turjman, PG Vinoj, and Leonardo Mostarda. 2019. Artificial muscle intelligence system with deep learning for post-stroke assistance and rehabilitation. *Ieee Access* 7 (2019), 133463–133473.
[24] Jari Kaivo-oja. 2012. Weak signals analysis, knowledge management theory and systemic socio-cultural transitions. *Futures* 44, 3 (2012), 206–217.
[25] Kuljeet Kaur, Sahil Garg, Gagangeet Singh Aujla, Neeraj Kumar, Joel JPC Rodrigues, and Mohsen Guizani. 2018. Edge computing in the industrial internet of things environment: Software-defined-networks-based edge-cloud interplay. *IEEE communications magazine* 56, 2 (2018), 44–51.
[26] Dongyeon Kim, Kyuhong Park, Yongjin Park, and Jae-Hyeon Ahn. 2019. Willingness to provide personal information: Perspective of privacy calculus in IoT services. *Computers in Human Behavior* 92 (2019), 273–281.
[27] Aliaksei Kolesau and Dmitrij Šešok. 2020. Voice activation systems for embedded devices: Systematic literature review. *Informatica* 31, 1 (2020), 65–88.
[28] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).
[29] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).
[30] Ashish Kumar, Saurabh Goyal, and Manik Varma. 2017. Resource-efficient machine learning in 2 KB RAM for the internet of things. In *International Conference on Machine Learning*. PMLR, 1935–1944.
[31] Liangzhen Lai and Naveen Suda. 2018. Enabling deep learning at the LoT Edge. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–6.
[32] Liangzhen Lai, Naveen Suda, and Vikas Chandra. 2017. Deep convolutional neural network inference with floating-point weights and fixed-point activations. *arXiv preprint arXiv:1703.03073* (2017).
[33] Liangzhen Lai, Naveen Suda, and Vikas Chandra. 2018. Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus. *arXiv preprint arXiv:1801.06601* (2018).
[34] Liangzhen Lai, Naveen Suda, and Vikas Chandra. 2018. Not all ops are created equal! *arXiv preprint arXiv:1801.04326* (2018).
[35] Xuesong Li, Jose Guivant, Ngaiming Kwok, Yongzhi Xu, Ruowei Li, and Hongkun Wu. 2019. Three-dimensional backbone network for 3d object detection in traffic scenes. *arXiv preprint arXiv:1901.08373* (2019).
[36] Andrea Massa, Davide Marcantonio, Xudong Chen, Maokun Li, and Marco Salucci. 2019. DNNs as applied to electromagnetics, antennas, and propagation—A review. *IEEE Antennas and Wireless Propagation Letters* 18, 11 (2019), 2225–2229.
[37] Simon Mittermaier, Ludwig Kürzinger, Bernd Waschneck, and Gerhard Rigoll. 2020. Small-footprint keyword spotting on raw audio data with sinc-convolutions. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 7454–7458.
[38] Mao V Ngo, Hakima Chaouchi, Tie Luo, and Tony QS Quek. 2020. Adaptive anomaly detection for IoT data in hierarchical edge computing. *arXiv preprint arXiv:2001.03314* (2020).
[39] Lin Ning and Xipeng Shen. 2019. Deep reuse: streamline CNN inference on the fly via coarse-grained computation reuse. In *Proceedings of the ACM International Conference on Supercomputing*. 438–448.
[40] Lin Ning and Xipeng Shen. 2019. Deep Reuse: streamline CNN inference on the fly via coarse-grained computation reuse. In *Proceedings of the ACM International Conference on Supercomputing*. 438–448.
[41] Wei Niu, Xiaolong Ma, Sheng Lin, Shihao Wang, Xuehai Qian, Xue Lin, Yanzhi Wang, and Bin Ren. 2020. PatDNN: Achieving Real-Time DNN Execution on Mobile Devices with Pattern-Based Weight Pruning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 907–922. https://doi.org/10.1145/3373376.3378534
[42] Nefy Puteri Novani, Mohammad Hafiz Hersyah, and Ryon Hamdanu. 2020. Electrical Household Appliances Control using Voice Command Based on Microcontroller. In *2020 International Conference on Information Technology Systems and Innovation (ICITSI)*. IEEE, 288–293.
[43] Michela Paganini and Jessica Forde. 2020. Streamlining tensor and network pruning in pytorch. *arXiv preprint arXiv:2004.13770* (2020).
[44] Zheng Qin, Zhaoning Zhang, Xiaotao Chen, Changjian Wang, and Yuxing Peng. 2018. Fd-mobilenet: Improved mobilenet with a fast downsampling strategy. In *2018 25th IEEE International Conference on Image Processing (ICIP)*. IEEE, 1363–1367.

[45] Marc Riera, Jose-Maria Arnau, and Antonio Gonzalez. 2018. Computation Reuse in DNNs by Exploiting Input Similarity. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 57–68. https://doi.org/10.1109/ISCA.2018.00016

[46] Manuele Rusci, Alessandro Capotondi, and Luca Benini. 2020. Memory-driven mixed low precision quantization for enabling deep network inference on microcontrollers. *Proceedings of Machine Learning and Systems* 2 (2020), 326–335.

[47] Falk Salewski and Stefan Kowalewski. 2008. Hardware/software design considerations for automotive embedded systems. *IEEE Transactions on Industrial Informatics* 4, 3 (2008), 156–163.

[48] Jiawei Shao and Jun Zhang. 2020. Bottlenet++: An end-to-end approach for feature compression in device-edge co-inference systems. In *2020 IEEE International Conference on Communications Workshops (ICC Workshops)*. IEEE, 1–6.

[49] Prerna Sharma and Deepali Kamthania. 2019. Intelligent object detection and avoidance system. In *International Conference on Transforming IDEAS (Inter-Disciplinary Exchanges, Analysis, and Search) into Viable Solutions*. 342–351.

[50] Stanislava Soro. 2021. Tinyml for ubiquitous edge ai. *arXiv preprint arXiv:2102.01255* (2021).

[51] Srinivasa R Sridhara. 2011. Ultra-low power microcontrollers for portable, wearable, and implantable medical electronics. In *16th Asia and South Pacific Design Automation Conference (ASP-DAC 2011)*. IEEE, 556–560.

[52] Hidetoshi Teraoka, Fumiharu Nakahara, and Kenichi Kurosawa. 2017. Incremental update method for vehicle microcontrollers. In *2017 IEEE 6th Global Conference on Consumer Electronics (GCCE)*. IEEE, 1–2.

[53] Ching-Biau Tzeng. 2018. Vibration detection and analysis of wind turbine based on a wireless embedded microcontroller system. In *2018 IEEE International Conference on Applied System Invention (ICASI)*. IEEE, 133–136.

[54] Jiayi Wang, Chengliang Chai, Nan Tang, Jiabin Liu, and Guoliang Li. 2022. Coresets over Multiple Tables for Feature-rich and Data-efficient Machine Learning. *Proc. VLDB Endow.* 16, 1 (2022), 64–76. https://www.vldb.org/pvldb/vol16/p64-wang.pdf

[55] Ruofan Wu, Feng Zhang, Jiawei Guan, Zhen Zheng, Xiaoyong Du, and Xipeng Shen. 2022. Drew: Efficient winograd cnn inference with deep reuse. In *Proceedings of the ACM Web Conference 2022*. 1807–1816.

[56] Ruofan Wu, Feng Zhang, Zhen Zheng, Xiaoyong Du, and Xipeng Shen. 2021. Exploring deep reuse in winograd CNN inference. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 483–484.

[57] Yan Yan, Yuxing Mao, and Bo Li. 2018. Second: Sparsely embedded convolutional detection. *Sensors* 18, 10 (2018), 3337.

[58] Hyunho Yeo, Youngmok Jung, Jaehong Kim, Jinwoo Shin, and Dongsu Han. 2018. Neural adaptive content-aware internet video delivery. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 645–661.

[59] JZ Yi, YK Tan, ZR Ang, and SK Panda. 2007. Microcontroller based voice-activated powered wheelchair control. In *Proceedings of the 1st international convention on Rehabilitation engineering & assistive technology: in conjunction with 1st Tan Tock Seng Hospital Neurorehabilitation Meeting*. 67–72.

[60] Yunkai Yu, Zhihong Yang, Yuyang You, and Wenjing Shan. 2021. FASSNet: fast apnea syndrome screening neural network based on single-lead electrocardiogram for wearable devices. *Physiological Measurement* 42, 8 (2021), 085005.

[61] Jian Yuan, Kok Kiong Tan, Tong Heng Lee, and Gerald Choon Huat Koh. 2014. Power-efficient interrupt-driven algorithms for fall detection and classification of activities of daily living. *IEEE Sensors Journal* 15, 3 (2014), 1377–1387.

[62] Matthew D Zeiler and Rob Fergus. 2014. Visualizing and understanding convolutional networks. In *European conference on computer vision*. Springer, 818–833.

[63] Feng Zhang, Jidong Zhai, Bingsheng He, Shuhao Zhang, and Wenguang Chen. 2016. Understanding co-running behaviors on integrated CPU/GPU architectures. *IEEE Transactions on Parallel and Distributed Systems* 28, 3 (2016), 905–918.

[64] Feng Zhang, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. 2022. POCLib: a high-performance framework for enabling near orthogonal processing on compression. *IEEE Transactions on Parallel and Distributed Systems* 33, 2 (2022), 459–475.

[65] Yundong Zhang, Naveen Suda, Liangzhen Lai, and Vikas Chandra. 2017. Hello edge: Keyword spotting on microcontrollers. *arXiv preprint arXiv:1711.07128* (2017).

Received 2022-10-20; accepted 2023-01-19