Check for updates

Received: 8 December 2023 | Revised: 6 March 2024 | Accepted: 19 March 2024

DOI: 10.4218/etrij.2023-0522

ETRI Journal WILEY

ORIGINAL ARTICLE

Featured article

Automated deep-learning model optimization framework for microcontrollers

Seungtae Hong[1,2] | Gunju Park[1] | Jeong-Si Kim[1]

[1]Artificial Intelligence Computing Research Laboratory, Electronics and Telecommunications Research Institute, Daejeon, Republic of Korea

[2]University of Science and Technology, Daejeon, Republic of Korea

Correspondence
Seungtae Hong, Artificial Intelligence Computing Research Laboratory, Electronics and Telecommunications Research Institute, Daejeon, Republic of Korea.
Email: sthong@etri.re.kr

Funding information
This research was supported by the Challengeable Future Defense Technology Research and Development Program through the Agency for Defense Development (ADD) funded by the Defense Acquisition Program Administration (DAPA) in 2022 (No. 915062201).

Abstract

This paper introduces a framework for optimizing deep-learning models on microcontrollers (MCUs) that is crucial in today's expanding embedded device market. We focus on model optimization techniques, particularly pruning and quantization, to enhance the performance of neural networks within the limited resources of MCUs. Our approach combines automatic iterative optimization and code generation, simplifying MCU model deployment without requiring extensive hardware knowledge. Based on experiments with architectures, such as ResNet-8 and MobileNet v2, our framework substantially reduces the model size and enhances inference speed that are crucial for MCU efficiency. Compared with TensorFlow Lite for MCUs, our optimizations for MobileNet v2 reduce static random-access memory use by 51%–57% and flash use by 17%–62%, while increasing inference speed by approximately 1.55 times. These advancements highlight the impact of our method on performance and memory efficiency, demonstrating its value in embedded artificial intelligence and broad applicability in MCU-based neural network optimization.

KEYWORDS

automated framework, deep learning, memory efficiency, microcontrollers, model optimization

1 | INTRODUCTION

Deep-learning technology has recently gained popularity and has found applications in various fields, including image classification and object recognition [1, 2]. Advancements in the Internet of Things (IoT) device technology have also increased the demand for deep learning in low-end devices, such as wearable healthcare devices. Accordingly, research on utilizing deep-learning technology in ultralight and low-cost devices, such as microcontrollers (MCUs) with limited resources, is receiving increasing attention [3].

Compared with general-purpose personal computers (PCs) or mobile devices, MCUs offer a more limited resource environment, integrating the central processing unit (CPU), memory, and input/output modules into a single chip. Typically, MCUs are equipped with a single-core CPU that operates at relatively lowclock speeds

This is an Open Access article distributed under the term of Korea Open Government License (KOGL) Type 4: Source Indication + Commercial Use Prohibition + Change Prohibition (http://www.kogl.or.kr/info/licenseTypeEn.do).
1225-6463/$ © 2024 ETRI

ranging from a few MHz to hundreds of MHz. The capacity of RAM and flash memory in these systems is limited, typically ranging from a few kilobytes (kB) to megabytes (MB). This is in stark contrast to the massive capacities (in the range of GBs or TBs) in more powerful computing devices.

Despite these limitations, MCUs offer excellent reliability thanks to their simplicity and design specificity. Running single-threaded processes or simple multitasking with real-time constraints reduces the potential for errors and complexities that can plague more sophisticated systems. Designs are often streamlined for specific tasks, thus reducing the potential for crashes and operational errors. Additionally, MCUs are designed to operate in adverse conditions and can withstand extreme temperatures, electromagnetic interference, and physical shocks, which are important factors in fault-tolerant applications, such as those in medical devices or aerospace systems. As a result, this operational stability and robustness support the high reliability of MCUs. Accordingly, MCUs are attracting attention as new devices for popularizing deep-learning technology.

To enable deep learning on an MCU, it is essential to reduce the model size using techniques like pruning and quantization owing to the limited system resources. From this perspective, research is currently being conducted on deep-learning model optimization for MCUs [4, 5] and on generating inference operation code that supports memory-efficient and optimized inference on MCUs [6, 7].

Although the aforementioned research areas are very important for implementing deep learning on MCUs, each area is mainly studied independently owing to its distinctive requirements. Model optimization for MCU [8] requires a complex knowledge of neural network architecture, whereas code generation for inference tasks necessitates a deep understanding of MCU-specific hardware features, such as leveraging single-instruction multiple-data (SIMD) operations to accelerate computations.

Although these detailed approaches are effective in advancing individual aspects of deep-learning deployments [9, 10], they lack the cohesion needed for a streamlined, end-to-end solution. In particular, generating code for inference tasks that is both memory-efficient and speed-optimized on MCUs demands a high level of expertise regarding the device's hardware complexities. This situation can create a barrier for individuals who are proficient in neural networks but lack detailed knowledge of MCU hardware intricacies.

Traditionally, efforts expended to implement deep learning on resource-restricted devices (such as MCUs) have encountered several hurdles. First, the complexity associated with the reduction of model size through pruning and quantization requires deep knowledge of neural network architectures. Second, the absence of unified strategies, owing to separate research on model optimization and inference code generation, complicates end-to-end deployment. Last, the need for memory-efficient, high-performance inference code demands an in-depth understanding of the peculiarities of MCU hardware. This complexity often deters specialists in neural networks from delving into hardware-specific optimizations.

In response, our study proposes an end-to-end solution that seamlessly integrates the optimization of deep-learning models with the generation of inference task code. The proposed framework enables the deployment of deep-learning models that are optimized for the targeted hardware, circumventing the need for in-depth hardware knowledge on resource-constrained MCUs.

Our key contributions are the following:

- We propose an integrated framework for deep learning on MCUs that unifies model optimization with inference code generation, tailored for devices with stringent resource constraints. This end-to-end solution is designed to be agnostic of the underlying hardware, allowing for broader applicability.
- We describe the framework's capability to combine multiple optimization techniques, offering users the choice among various combinations or recommended settings based on the network structure and model compression. This feature simplifies the model optimization process for MCUs, especially for users with limited knowledge of model light-weighting and MCU inference coding, including SIMD utilization.
- Empirical evaluations across various MCU architectures reveal that our framework markedly outperforms traditional, standalone optimization methods. These studies demonstrate our framework's enhanced computational efficiency and reduced memory footprint.

2 | BACKGROUND AND RELATED WORKS

2.1 | Model optimization techniques

Representative model optimization techniques in deep learning include pruning and quantization. Pruning [11–17], a crucial technique for model efficiency, involves the reduction of the size of the neural network by eliminating less critical weights. This process not only decreases memory usage but also enhances computational efficiency. Various forms of pruning exist, such as

22337326, 2025, 2, Downloaded from https://onlinelibrary.wiley.com/doi/10.4218/etrij.2023-0522 by Electronics and Telecommunications, Wiley Online Library on [24/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

weight, unit, and structured pruning, with each form having its specific application and impact. The primary goal is to minimize the impact on model performance while effectively reducing the number of parameters.

Quantization [13, 16, 18–21] further complements these optimization efforts. This technique transforms model weights and activations into lower-bit precision formats, substantially reducing storage requirements and computational loads. Particularly crucial in environments with limited resources, quantization makes deep-learning models faster and more lightweight. It strikes a delicate balance between maintaining neural network performance and enhancing operational speed, thereby making it a vital component in optimizing models for MCUs.

These optimization techniques, while pivotal in adapting deep-learning models for MCUs, require substantial knowledge about neural network architectures. Understanding the intricacies of how pruning and quantization impact a neural network's functioning is crucial, especially in resource-constrained environments like MCUs. In these settings, it is imperative to balance carefully the reduction of model weights with the maintenance of the model's accuracy and effectiveness. Overly aggressive optimization may lead to considerable performance degradation, which can be particularly challenging when dealing with complex tasks on MCUs.

2.2 | Deep-learning frameworks for MCUs

In the realm of deploying convolutional neural networks (CNNs) on MCUs, encountering diverse challenges [3] is inevitable owing to the vast array of MCU manufacturers, each offering custom frameworks tailored to their hardware specifics, such as ARM CMSIS-NN [22], STM X-CUBE-AI [23, 24], and Renesas e2-studio [25].

Contrasting these proprietary solutions, universal frameworks like TensorFlow Lite for Microcontrollers (TFLM) [6] expand TensorFlow Lite's capabilities to suit resource-constrained MCUs by incorporating a hardware abstraction layer, thus facilitating model deployment across different MCU architectures without necessitating deep hardware knowledge. TFLM employs an interpreter-based inference engine, optimizing runtime memory and layer operations, even though it may introduce boot-time bottlenecks and increased memory consumption requirements compared with compiled methods, thereby challenging devices with stringent memory limits.

Similarly, Apache MicroTVM [26] aims to reduce platform dependence by compiling deep-learning computations from PyTorch [27], TensorFlow [28], and others, into operations by considering hardware constraints, generating inference code on the host for deployment on devices. However, its pursuit of universality may compromise optimization effectiveness akin to TFLM's limitations.

To mitigate these shortcomings, compiler-based, deep-learning inference engines like MCUNet's TinyEngine [4], Microsoft's Embedded Learning Library (ELL) [29], uTensor [30], and NNoM [31], ONNC [32] enhance optimization for specific platforms. Notably, engines targeting NAS-derived networks like MCUNet and MicroNet [5] perform optimizations within a confined search space and face limitations regarding the variety of supported operational kernels due to targeted optimization efforts.

3 | AUTOMATED DEEP-LEARNING MODEL OPTIMIZATION FRAMEWORK

3.1 | Motivation

Our framework simplifies the deployment of deep-learning models on MCUs, thus making the process straightforward and eliminating the need for specialized knowledge in neural network architecture and hardware programming. It connects model compression techniques with MCU-specific code generation, enabling users to deploy models with greater ease. The overall structure of the proposed framework is shown in Figure 1.

Using a pretrained model, refined through neural architecture search (NAS), and the specific resource constraints of the MCU, such as static random-access memory (SRAM) and flash memory sizes as inputs, our framework was structured based on the automatic iterative optimization and automatic code generation phases to facilitate a smooth transition of deep-learning models to MCUs.

During the automatic iterative optimization phase, the framework repeatedly applies optimization techniques, such as pruning and quantization, to tailor the model to the MCU's resource limitations while aiming to achieve optimal performance. This process involves iterative tuning of the model's architecture to identify a right balance between size, computational demand, and accuracy without extensive manual intervention.

The phase starts with the evaluation of the model's compatibility with the MCU's resources, pinpointing opportunities to enhance efficiency. The framework then applies a variety of pruning methods to eliminate unnecessary weights and optimizes the structure for the given

22337326, 2025, 2, Downloaded from https://onlinelibrary.wiley.com/doi/10.4218/etrij.2023-0522 by Electronics and Telecommunications, Wiley Online Library on [24/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

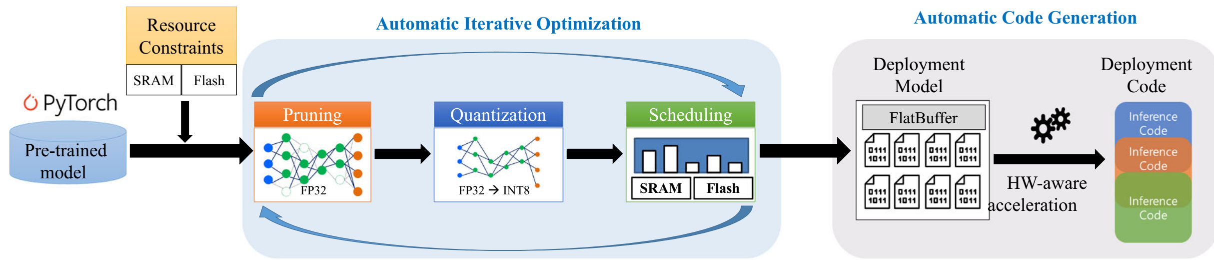

FIGURE 1 Overall structure of the proposed framework.

constraints. Following pruning, quantization reduces the precision of the model's weights to fit the memory limits, with constant evaluation to maintain accuracy. This ensures that the optimized model is suitable for deployment on an MCU, optimizing resource use and computational efficiency.

The automatic code generation phase translates the optimized model into executable code that fits the MCU's specifications. This phase produces code that is aligned with the computational and memory capabilities of the MCU, leveraging any available hardware acceleration to ensure efficient operation.

Our framework surpasses existing solutions, like MCUNet, by integrating advanced optimization techniques such as quantization-aware training (QAT) and post-training quantization (PTQ), enabling more accurate and efficient deep-learning model deployment on MCUs. This approach not only achieves higher accuracy with minimal performance loss but also ensures that users, regardless of their deep learning or MCU expertise, can effectively leverage AI technologies.

3.2 | Automatic iterative optimization

The automatic iterative optimization phase automates the intricate processes of pruning, quantization, and memory scheduling to facilitate deploying deep-learning models on MCUs. This phase greatly reduces the need for in-depth knowledge of neural networks or the hardware specifics of MCUs, thus simplifying the optimization process.

Initially, the framework evaluates a pretrained model against the resource constraints of SRAM and flash memory, which are typical of MCUs. It methodically assesses the model based on a variety of pruning strategies, each designed to achieve an ideal balance between model sparsity and performance integrity. Subsequently, a fine-tuning process was employed to reclaim, and potentially improve, the model's accuracy following pruning.

ALGORITHM 1 Model Optimization Algorithm

Input **model – a pretrained deep-learning model**
constraints – resource constraints (static random-access memory (SRAM) and flash)
Output **optimized_model – a lightweight model optimized for deployment**

```
1   val_accuracy ← MeasureValidationAccuracy(model)
2   candidates ← DefineCandidateSets()
3   best_model ← model
4   best_val_accuracy ← val_accuracy
5   best_resource_usage ← MeasureResourceUsage(model)
6
7   for each candidate_set in candidates do
8     pruned_model ← Pruning(model, candidate_set)
9     pruned_model ← FineTune(pruned_model)
10    quantized_model ← Quantization(pruned_model)
11    for scheduling_round in SchedulingRounds do
12      scheduled_model ← MemoryScheduling(quantized_model)
13      sram_usage, flash_usage
            ← MeasureResourceUsage(scheduled_model)
14      val_accuracy
            ← MeasureValidationAccuracy(scheduled_model)
15      if sram_usage <= constraints.SRAM and
          flash_usage <= constraints.flash then
16        if val_accuracy > best_val_accuracy then
17          best_model ← scheduled_model
18          best_val_accuracy ← val_accuracy
19          best_resource_usage ← {sram_usage, flash_usage}
20        end if
21      end if
22    end for
23  end for
24  return best_model
```

Quantization is crucial to adapt to the limited computational resources of the MCUs. It mainly involves two techniques: QAT and PTQ. QAT is typically recognized for its superior performance, particularly in retaining high accuracy in models, thus making it a preferred choice for performance-critical applications [18].

22337326, 2025, 2, Downloaded from https://onlinelibrary.wiley.com/doi/10.4218/etrij.2023-0522 by Electronics and Telecommunications, Wiley Online Library on [24/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Although QAT is typically employed as the default method for its superior performance in retaining high-model accuracy, we also recognize the utility of PTQ in specific scenarios, particularly with models that possess lower levels of compression. Unlike the MCUNet, which primarily focuses on architectural searches with subsequent post-training quantization, our methodology embraces a broader spectrum of optimization techniques. By incorporating a dual approach to quantization, including both QAT and PTQ, our framework ensures that models are not only compact but also retain high levels of accuracy, thus making them more adaptable to the diverse requirements of MCU-based applications.

The optimization algorithm also includes an automated scheduling feature, which refines the model's memory usage in accordance with the MCU's specifications. This component intelligently orchestrates kernel operations within memory constraints, optimizing SRAM and flash usage while maintaining validation accuracy. Models that successfully navigate this stringent scheduling process are deemed fit for deployment.

The steps of this optimization are meticulously encapsulated in the pseudocode of Algorithm 1, thus demonstrating the algorithm's effectiveness in generating models that satisfy the stringent efficiency and performance criteria required by MCUs. This automated procedure adeptly circumvents the common performance trade-offs associated with the optimization of models for resource-limited settings.

We refine the model optimization algorithm to achieve the transformation from an initial pretrained model M to its optimized counterpart M_{OPT}, which is suitable for MCU deployment,

$$M_{\mathrm{OPT}} = \arg\max_{M*} A(M^*), \text{subject to}\, M^* = Q(P[M_{\mathrm{INIT}}, \theta_{\mathrm{P}}], \theta_{\mathrm{Q}}), \tag{1}$$

where

- $A(M^*)$ denotes the accuracy metric of the optimized model M^*, emphasizing its performance after undergoing optimization;
- M_{INIT} represents the original, pretrained model before any optimization;
- $P\,(M_{\mathrm{INIT}}, \theta_{\mathrm{P}})$ signifies the pruning operation applied to M_{INIT}, with θ_{P} as the threshold for pruning, aiming to eliminate weights selectively;
- Q (pruned_model, θ_{Q}) is the quantization function applied to the pruned model, where θ_{Q} is a parameter that varies depending on the quantization technique, such as PTQ or QAT;
- M_{OPT} symbolizes the final, optimized model, showcasing the effectiveness of combining pruning with advanced quantization strategies to meet the operational constraints of MCUs.

To conclude, the automatic iterative optimization phase executes a complex balance of pruning, quantization, and memory scheduling to arrive at an optimally configured model that complies with the predetermined resource constraints of SRAM and flash memory. The entire process is autonomous, eliminating the need for user intervention in determining the most suitable model parameters. This autonomous algorithm traverses the optimization landscape, thus ensuring that the resultant model is not only compact and efficient but is also specifically optimized to function within the stringent resource boundaries of MCUs. This systematic strategy broadens the application of deep learning within the MCU arena, empowering the use of advanced AI technologies irrespective of specialized hardware knowledge.

In the specialized area of MCU-based deep-learning model optimization, selecting the most appropriate pruning technique is essential to utilize effectively hardware acceleration capabilities and ensure efficient inference. Our framework concentrates on selecting filter pruning methods for its candidate sets, as these methods work well with hardware acceleration techniques. This approach guarantees that our model optimization is not only efficient but also aligns with the unique hardware attributes of MCUs, thereby facilitating quicker and more resource-efficient inferences.

Our framework incorporates both one-shot channel pruning [11] and alternating direction method of multipliers (ADMM) pruning [33], each offering unique benefits for MCU optimization. One-shot channel pruning is recognized for its speed and efficiency in reducing redundant channels, thus making it invaluable in the context of resource-limited MCUs. It adeptly reduces model size while minimally impacting performance, which is crucial for devices with constrained processing capabilities. Conversely, ADMM pruning is known for its iterative refinement process, which progressively compresses the model to balance compactness and accuracy. This method is especially advantageous for MCUs, allowing for incremental adjustments that conform to specific memory and computational limitations, ensuring that the model remains both efficient and precise.

In our framework, users can select between one-shot channel pruning and ADMM pruning according to their optimization needs and model characteristics. One-shot channel pruning is ideal for rapid model size reduction

22337326, 2025, 2, Downloaded from https://onlinelibrary.wiley.com/doi/10.4218/etrij.2023-0522 by Electronics and Telecommunications, Wiley Online Library on [24/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

and time efficiency, whereas ADMM pruning is better suited for deeper model compaction in resource-limited environments. Our workflow, particularly in Algorithm 1, offers a range of pruning methods when defining candidate sets ("candidates ← DefineCandidateSets()"). Users also benefit from heuristic recommendations based on model complexity and the desired compression level. This approach not only provides flexibility in choosing the most suitable pruning method but also ensures optimal model performance by balancing model efficacy with MCU resource constraints.

In addition, our framework employs a range of metrics like random, l1_norm, l2_norm, and fine-grained pruning of the geometric median (FPGM) to evaluate and select the optimal pruning candidates. This ensures that the chosen channels for pruning have the least impact on the model's performance. The framework also offers varied sparsity levels to cater to different model compaction requirements: 0.25, 0.5, 0.75, and 0.875 for ADMM pruning and one-shot channel pruning. This nuanced approach to sparsity enables the framework to adapt to a broad spectrum of MCU specifications, tailoring the optimization process to the specific requirements of each deployment scenario.

In the automatic iterative optimization phase of our framework, we propose a memory scheduling algorithm based on the principles of the FirstFit algorithm, similarly utilized in MCUNet's TinyEngine. This approach leverages the effectiveness of the FirstFit algorithm for managing the memory constraints characteristic of MCUs. Chosen for its straightforward and efficient memory allocation method, the FirstFit algorithm scans from the start of the memory space, allocating the first suitable block for each memory requirement. This strategy is especially appropriate for the constraints of MCU environments, where efficient use of limited memory resources like SRAM and flash is crucial.

As outlined in the memory scheduling algorithm (Algorithm 2), our process begins by initializing slots to represent the total available SRAM, achieved by the *InitializeSlots* function. This creates a comprehensive structure for managing SRAM allocation. For each layer of the pruned and quantized deep-learning model, the *CalculateMemoryUsage* function determines the SRAM and flash memory requirements. The algorithm finds the best slot for each layer's SRAM using the *FindLowestSuitableSlot* function and assesses the slot's suitability. The starting point of each SRAM allocation is tracked and appended to the allocation report. Simultaneously, the flash memory requirement for each layer is recorded, reflecting a comprehensive understanding of the model's total memory footprint in both SRAM and flash.

The process iterates through each layer of the model, summarizing in a detailed allocation report the allocations of both SRAM and flash for each layer. This detailed reporting, as returned by Algorithm 2, ensures efficient utilization of memory, minimizes fragmentation, and aligns with the operational sequence of the model. It effectively demonstrates our framework's capability to manage complex memory allocation challenges in MCUs, thereby enhancing the deployment efficiency of deep-learning models in these resource-constrained environments.

ALGORITHM 2 Memory Scheduling Algorithm

Input **model – a pruned and quantized deep-learning model**
resource_constraints – resource constraints (SRAM and flash)
Output **allocation_report – report of SRAM and flash allocations for each layer**

```
1   slots ← InitializeSlots(resource_constraints.SRAM)
2   allocation_report ← {SRAM: [], flash: []}
3
4   for each layer in the model do
5     sram_size, flash_size ← CalculateMemoryUsage(layer)
6     if sram_size > 0 then
7       suitable_slot ← FindLowestSuitableSlot(slots, sram_size)
8       AssertSlotExists(suitable_slot)
9       slots ← AdjustSlots(slots, suitable_slot, sram_size)
10      allocation_report.SRAM.append(GetSlotStart(suitable_slot))
11    end if
12    if flash_size > 0 then
13      allocation_report.flash.append(flash_size)
14    end if
15  end for
16  return allocation_report
```

We refine the memory scheduling algorithm, underscoring its essential role in optimizing M_{OPT}'s deployment on MCUs, balancing efficiency and performance within tight memory constraints. This strategic adjustment paves the way for

$$\text{Minimize Overflow}\,(M_{\text{OPT}}) + \lambda \text{Efficiency}\,(M_{\text{OPT}}), \text{subject to MS}\,(M_{\text{OPT}}, R_{\text{CONST}}), \tag{2}$$

where

- Overflow (M_{OPT}) quantifies the challenge of fitting M_{OPT} within the strict memory limitations of MCUs, aiming to minimize this metric;
- Efficiency (M_{OPT}) characterizes the essential balance between running the optimized model effectively and

22337326, 2025, 2, Downloaded from https://onlinelibrary.wiley.com/doi/10.4218/etrij.2023-0522 by Electronics and Telecommunications, Wiley Online Library on [24/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

adhering to the MCU's memory constraints using λ as the coefficient to mediate this balance;
- MS (M_{OPT}, R_{CONST}) delineates the memory scheduling strategy, designed to align M_{OPT}'s deployment with the available SRAM and flash memory, ensuring optimal performance within the confines of MCU resources.

3.3 | Automatic code generation

After the automatic iterative optimization phase, the framework enters the automatic code generation phase. This phase bridges the gap between optimized deep-learning models and their practical applications in MCUs. The goal is to transform the model into an executable format that not only adapts to the hardware specifications of the MCU but also takes full advantage of its computational potential.

The automatic code generation phase includes several key steps:

- **Buffer code generation**: This step involves the creation of buffer definitions for the SRAM and flash memory within the MCU's memory constraints. In flash, read-only model parameters are stored, such as weights, biases, and quantizer parameters, including scales and zero points. Within the SRAM, dedicated read-write buffers are allocated for each layer's input and output activations, as well as a scratch buffer, which is essential for temporarily holding intermediate computation results.
- **Header file generation**: Following buffer definition, the routine generates header files that incorporate essential declarations and macro definitions for the inference code.
- **Invoke function creation**: The culmination of this step is attained when an invoke function is developed that sequentially calls the deep-learning model's operation functions to ensure accurate inference execution on the MCU.

During this phase, the optimized model from the automatic iterative optimization phase is converted to the Tensorflow Lite format, which by default uses FlatBuffer [34] for model representation. FlatBuffer was chosen for its efficiency in memory usage and its capability to promote fast computations, thus making it ideal for resource-constrained MCU environments. Additionally, during the conversion to the TensorFlow Lite model, quantization is performed by considering preprocessing operations, such as image pixel value normalization. This eliminates bottlenecks in image preprocessing during MCU runtime, thus enabling efficient inferences.

Typically, for deep-learning model inference on MCUs, TFLM is employed. TFLM operates as an inference engine using an interpreter approach like that shown in Figure 2A, which includes various tasks for model execution at runtime, considering the execution of generic TensorFlow Lite models. However, bottlenecks occurring during runtime in resource-constrained MCU environments notably impact inference efficiency. To overcome this, there are compile-based inference engines like those shown in Figure 2B that perform (offline) tasks prone to runtime bottlenecks, such as memory allocation, operator setup, and optimal kernel search.

This approach generates an optimized deep-learning C-code engine. Open-source tools like microTVM, ONNC, and IDE from MCU manufacturers like STM and Renesas employ this compile-based inference engine. Our code generator was also designed as a compile-based inference engine based on a similar approach. Figure 3 exemplifies the kernel-level optimization in neural network operators that is crucial for efficient code generation. It contrasts the original MatMul kernel in ARM CMSIS-NN (A) with the operator-level optimized MatMul kernel (B). These pseudocodes illustrate how our framework refines the computational kernels to enhance performance on MCUs. These optimizations include the reduction of computational complexity and memory usage, which are critical in resource-constrained environments like MCUs.

Further, to accelerate inference on MCU devices, we utilized some additional acceleration techniques that

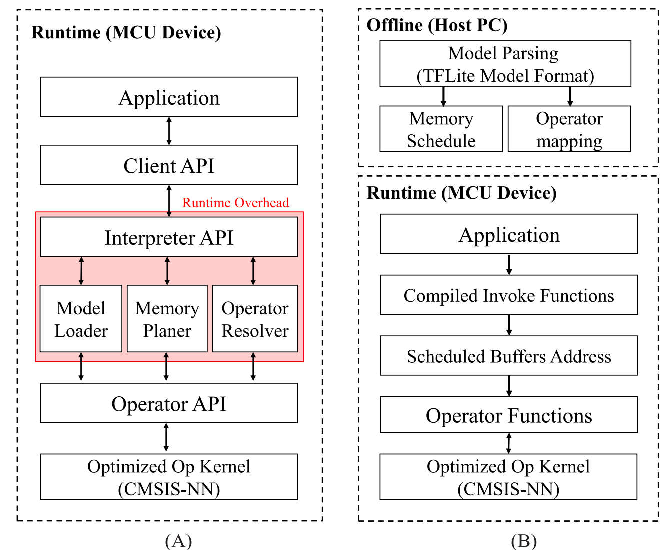

FIGURE 2 Comparative diagram of neural network deployment engines: (A) depicts the interpreter-based engine with its runtime overhead components, and (B) illustrates the compiled-based engine, including the offline preparation and runtime processes.

22337326, 2025, 2, Downloaded from https://onlinelibrary.wiley.com/doi/10.4218/etrij.2023-0522 by Electronics and Telecommunications, Wiley Online Library on [24/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

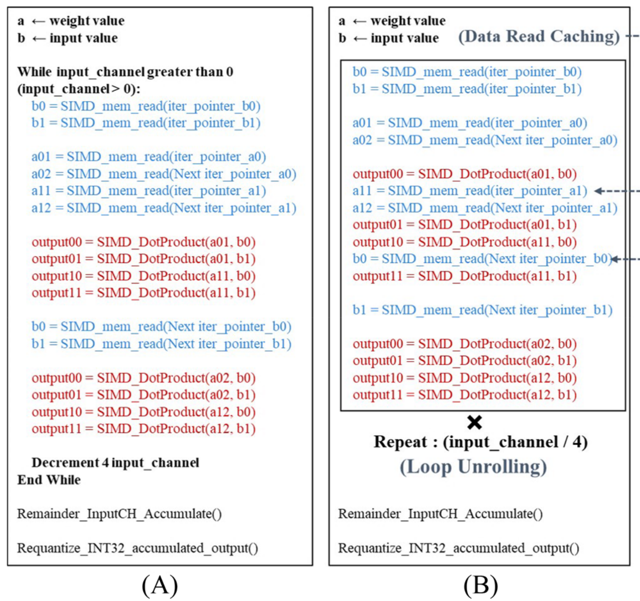

FIGURE 3 NN operators kernel-level optimization pseudo codes. (A) Original CMSIS-NN MatMul kernel and (B) operator-level optimized MatMul kernel.

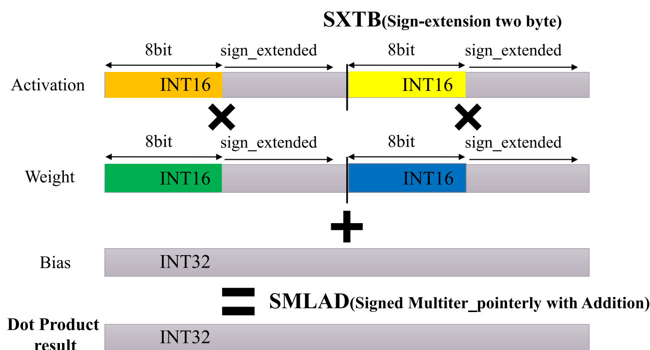

FIGURE 4 Examples of ARM-CMSIS single-instruction multiple-data (SIMD) instructions for microcontrollers (MCU).

leveraged hardware features, including SIMD. For this purpose, our framework utilized the computational kernel implemented in MCUNet's TinyEngine. The TinyEngine employs technologies, such as data caching and loop unrolling based on ARM CMSIS-NN, and it is currently demonstrating excellent performance in ARM Cortex-M MCUs. The key to this phase is to create custom code generation routines tailored to generate efficient C/C++ code that can run smoothly on the MCU. Figure 4 shows an example of ARM CMSIS SIMD instructions for an MCU. These instructions are integral to our framework's approach to leverage hardware acceleration features. By utilizing SIMD instructions, our framework substantially increases the speed of computations, thus making it more suitable for real-time applications on MCUs. This illustrates how deep-learning models can be effectively executed on MCUs, taking full advantage of their limited, yet powerful processing capabilities.

By leveraging SIMD for hardware acceleration, our framework's automatic code generation phase supports essential convolutions, broadening the range of networks like ResNet-8 executable on MCUs. This not only sets our framework apart from MCUNet but also enhances the deployment of deep-learning models in limited-resource settings. It reflects our dedication to advancing embedded AI, enabling sophisticated model operations on MCUs for real-time applications with unparalleled efficiency and flexibility.

4 | EXPERIMENTS

4.1 | Experimental setup

This section evaluates our framework's effectiveness in optimizing deep-learning models for MCUs. We used two MCU boards, namely, F746G-Disco and H747I-Disco, to demonstrate this. The F746G-Disco, a single-CPU board, and the H747I-Disco, a dual-CPU board, are summarized in Table 1. These boards were chosen to test our framework in MCU environments with varying resource profiles in different hardware contexts.

Table 2 lists the various optimization parameters that were explored in this study. This includes a range of pruning methods, such as one-shot and ADMM, applied with different metrics like L1-norm, L2-norm, FPGM, and random. Additionally, the table lists various levels of sparsity and quantization methods (QAT and PTQ), which were pivotal in fine-tuning the models for optimal performance on MCUs. The selection of these parameters was critical in ensuring that the models were not only size-efficient but also maintained high accuracy, a balance that is essential for effective deployment on resource-limited MCUs.

Model optimization was performed on a PC with a Nvidia RTX 4090 graphics processing unit by utilizing the PyTorch framework renowned for its efficient processing of complex models. Once optimized, these models undergo conversion for MCU compatibility using Alibaba's TinyNeuralNetwork [35], which translates them into TensorFlow Lite format. This step is crucial for ensuring that the models can function effectively on MCU platforms.

Following this, the automatic code generation phase begins, where the TensorFlow Lite models are used to generate inference code tailored for MCUs. Our experimental setup includes two distinct CNN architectures: ResNet-8, which relies on standard 3×3 convolutions,

22337326, 2025, 2, Downloaded from https://onlinelibrary.wiley.com/doi/10.4218/etrij.2023-0522 by Electronics and Telecommunications, Wiley Online Library on [24/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

22337326, 2025, 2, Downloaded from https://onlinelibrary.wiley.com/doi/10.4218/etrij.2023-0522 by Electronics and Telecommunications, Wiley Online Library on [24/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 1 Detailed specifications of the microcontrollers used for experiments.

Type	Central processing unit (maximum clock)	Static random-access memory (SRAM)	Flash	Quad-SPI (bandwidth)
STM32 F746G-Disco	Cortex M7 (216 MHz)	340 kB	1 MB	128 Mbits
STM32 H747I-Disco	Cortex M7(480 MHz) Cortex M4(240 MHz)	1 MB	2 MB	512 Mbits

TABLE 2 Optimization parameter candidates set for experiments.

Pruning method	Pruning metrics	Sparsity	Quantization method
• One-shot • ADMM	• L1-norm • L2-norm • FPGM • Random	• 0.25 • 0.5 • 0.75 • 0.875	• QAT • PTQ

and MobileNet v2, noted for its depthwise separable structure. These architectures are evaluated using the Cifar10 dataset, aligned with the MLPerf-Tiny benchmark [36], to assess thoroughly the optimization techniques across diverse neural network configurations. The choice of this benchmark is particularly relevant in the context of TinyML, where the focus is to enable AI computations on compact IoT devices like MCUs.

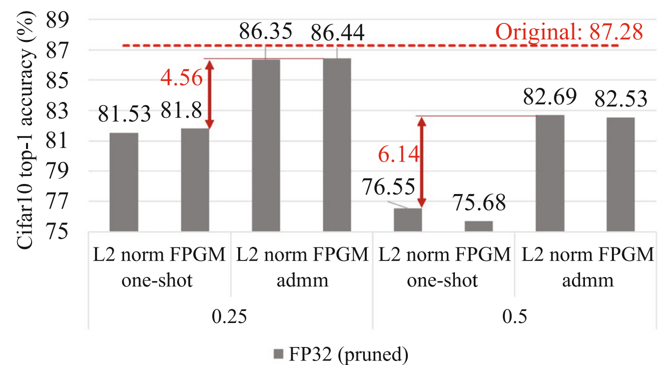

FIGURE 5 ResNet-8 Cifar10 accuracy after pruning.

4.2 | Pruning and quantization analysis

4.2.1 | Pruning method comparison

In this section, we evaluate and compare the effectiveness of one-shot and ADMM on two different CNN architectures, ResNet-8 and MobileNet v2.

Figure 5 presents the comparative results obtained after the application of two prominent pruning metrics, L2-norm and FPGM, to the ResNet-8 network, which is based on a standard 3×3 convolutional structure. Our experiments on the ResNet-8 network show that the ADMM method is more effective than the one-shot method at maintaining model accuracy after pruning. The ADMM approach resulted in a notably lower accuracy drop, exemplified by a 4.56% decrease at a sparsity level of 0.25, compared with a more significant reduction of 6.14% at a sparsity level of 0.5 obtained in the one-shot method case. These findings emphasize the strength of the ADMM method in preserving a higher degree of accuracy in standard convolutional networks even after extensive pruning.

In contrast, the MobileNet v2 architecture, which is premised on 3×3 depthwise separable convolution blocks, offers a distinctive scenario. As Figure 6 shows, although the difference in model accuracy postpruning is subtle between the one-shot and ADMM methods for MobileNet v2 at moderate sparsity levels, there is a notable trend as sparsity increases. The data suggest that although both pruning methods are effective at maintaining accuracy near the original model's performance, the FPGM metric exhibits a more stable accuracy preservation compared with the L2-norm, especially at higher sparsity levels, affirming FPGM's robustness in depthwise separable convolution-based architectures.

Nevertheless, when considering the computational cost of pruning, the single-shot channel pruning method typically requires condiderably less time than that required by ADMM. For example, at a sparsity level of 0.25, the single-shot approach requires considerably less time compared with ADMM, which translates to improved temporal efficiency. This efficiency makes the single-shot pruning method a more practical choice for architectures like MobileNet v2.

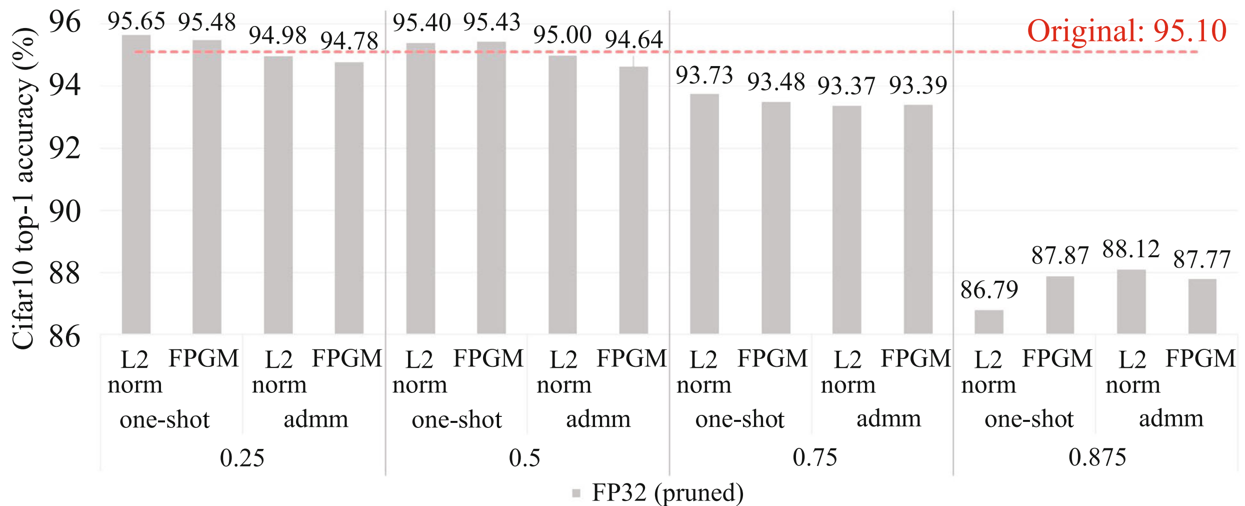

FIGURE 6 MobileNet v2 Cifar10 accuracy after pruning (one-shot, ADMM). In pruned model with sparsity (0.25, 0.5, 0.75), one-shot method with a slightly better performance. In the case of the pruned model with a sparsity of 0.875, the ADMM method yields a slightly better performance than that of the one-shot method.

4.2.2 | Quantization methods comparison

In this section, we present a detailed comparison of two quantization methods, QAT and PTQ, as applied to the MobileNet v2 model after one-shot channel pruning.

Table 3 demonstrates that as the sparsity in the pruned model increases, QAT tends to achieve higher accuracy compared with PTQ. This pattern is consistently observed across various sparsity levels and metrics. For example, at a sparsity level of 0.25 using the L1 norm, the accuracy of QAT is 95.24%, which is 0.42% higher than that of PTQ. Similarly, at a sparsity level of 0.875, QAT reaches an accuracy of 86.46%, outperforming PTQ by 1.33%.

For models with lower levels of compression, PTQ is found to be more beneficial, particularly when considering the reduced time and computational resources required. PTQ notably outpaces QAT in terms of computational cost. Therefore, our framework adopts a strategy that selects the quantization method in accordance with the level of model compression, ensuring that performance efficiency is maintained—vital in the context of MCUs where resource optimization is paramount.

TABLE 3 Comparative analysis of the accuracies of QAT and PTQ on MobileNet v2 at various sparsity levels.

		Cifar10 Top-1 accuracy		
Sparsity	Metric	PTQ	QAT	QAT-PTQ
0.25	L1 norm	94.82	**95.24**	+0.42
	L2 norm	94.71	95.06	+0.35
	FPGM	**94.91**	95.01	+0.1
0.5	L1 norm	94.91	94.75	−0.16
	L2 norm	94.69	**94.86**	+0.17
	FPGM	**94.98**	94.81	−0.17
0.75	L1 norm	**92.84**	**92.99**	+0.15
	L2 norm	92.37	92.71	+0.34
	FPGM	92.58	92.74	+0.16
0.875	L1 norm	85.13	86.46	+1.33
	L2 norm	84.8	85.9	+1.1
	FPGM	**85.78**	**86.77**	+0.99

4.3 | Interpreter versus compile-based engine comparison on MCU device

We evaluate herein the inference speed and memory usage efficiency in the MCU environment by comparing interpreter-based (TFLM) and compiler-based (our engine) models, both utilizing the MobileNet v2 architecture. Our selection of MobileNet v2 as the performance evaluation model was based on its robustness to various pruning methodologies compared with ResNet-8, thus maintaining superior accuracy performance across different pruning approaches. This robust performance underlines the suitability of MobileNet v2 for our comparative analysis, aiming to showcase the efficiency of compiler-based optimization compared with traditional interpreter-based models in resource-constrained MCU settings.

22337326, 2025, 2, Downloaded from https://onlinelibrary.wiley.com/doi/10.4218/etrij.2023-0522 by Electronics and Telecommunications, Wiley Online Library on [24/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 4 MobileNet v2 inference time on MCU (STM32H747I).

Sparsity	Inference time [ms] (speed-up)		
	TFLM	Our method (without optimization)	Our method
0.25	850	780 (**1.09**×)	703 (**1.21**×)
0.5	457	410 (**1.11**×)	373 (**1.23**×)
0.75	186	144 (**1.29**×)	134 (**1.39**×)
0.875	85	57 (**1.49**×)	55 (**1.55**×)

4.3.1 | Inference speed

In this section, we explore the relationship between model compression and inference acceleration.

The listings in Table 4 demonstrate the correlation between the increased model sparsity and enhanced inference speed, thus showcasing a marked improvement compared with that achieved by TFLM. This improvement is consistent across all performance evaluation targets, with our framework, both in optimized ("Our method") and non-optimized forms ("Our method without optimization"), consistently outperforming TFLM. Unpruned original models cannot be executed as they exceed the maximum SRAM size of the MCU and require runtime memory. However, models pruned to a sparsity of 0.25 or higher are capable of performing inference at rates >1 frames per second (FPS), and those pruned to a sparsity of 0.875 can achieve up to approximately 18 FPS. Additionally, this is particularly significant in our optimized framework, which exhibits a speedup ranging from approximately 1.21 times (at lower sparsity levels) to values as high as 1.55 times (at the highest sparsity level) compared with TFLM. This underscores the effectiveness of our optimization strategies, especially when dealing with higher levels of model compression.

Furthermore, the implementation of loop unrolling and data caching, as detailed in Section 3.3, plays a pivotal role in enhancing these gains. This demonstrates the compound effect of our comprehensive optimization approach, which includes both the automated iterative optimization phase and the automatic code generation phase that effectively leverages hardware acceleration features.

This enhancement highlights the success of our framework in reducing the time required to reach inferences, thus markedly outpacing TFLM and demonstrating the compound effect of our optimization techniques. The automated iterative optimization phase refines the model structure, and the automatic code generation phase leverages hardware acceleration features, such as SIMD operations, to generate highly efficient inference code for the MCU platform. These strategic advancements are critical in greatly boosting inference speeds, thus showcasing the comprehensive design of our framework that optimizes both the model and the execution code for maximum performance on targeted hardware.

4.3.2 | Memory scheduling

This section presents a detailed analysis of runtime memory usage for deep-learning models on MCUs.

Table 5 provides a comparative overview of the runtime memory requirements in terms of SRAM and flash memory when using our framework versus the TFLM. The data show that our framework consistently achieves a higher compression ratio, which directly translates to lower memory usage on the MCU. For instance, at a sparsity level of 0.25, our framework required only 221 kB of SRAM and 1325 kB of flash, which represent respective reductions of 51% and 17% compared with TFLM. This trend of reduced memory footprint continues as the sparsity level increases, highlighting the framework's effectiveness in memory optimization processes.

Furthermore, at higher sparsity levels, the benefits become even more pronounced. At a sparsity level of 0.875, our framework necessitates only 38.5 kB of SRAM and 182.2 kB of flash, underscoring impressive respective decreases of 53% and 43% in memory usage compared with those related to TFLM. These results not only showcase our framework's ability to minimize memory consumption substantially but also its potential to enable more complex models to run on MCUs with limited memory resources.

The integration of this efficient memory management into our framework ensures that it is well-suited for various real-world applications where memory constraints are a critical factor. By optimizing the trade-off between runtime memory and model complexity, our framework provides a path to realize the full potential of MCUs in executing deep-learning tasks with limited overhead.

22337326, 2025, 2, Downloaded from https://onlinelibrary.wiley.com/doi/10.4218/etrij.2023-0522 by Electronics and Telecommunications, Wiley Online Library on [24/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 5 Comparative analysis of MCU runtime memory for MobileNetV2-Cifar10 inference between TFLM and our method.

	SRAM (kB)			Flash (kB)		
Sparsity	**Ours**	**TFLM**	**Compression ratio**	**Ours**	**TFLM**	**Compression ratio**
0.25	221.0	451	51%	1325.0	1591.3	17%
0.5	147.3	310	52%	623.8	845.0	26%
0.75	74.7	160	53%	182.2	320.0	43%
0.875	38.5	90	57%	58.7	154.9	62%

5 | CONCLUSION

In this study, we focused on optimizing deep-learning models for MCUs, a rapidly growing field owing to the proliferation of embedded devices. The heart of our work involved the development and evaluation of advanced optimization techniques tailored to neural networks operating in resource-limited environments.

Key findings include the effective optimization of architectures such as ResNet-8 and MobileNet v2, utilizing advanced pruning and quantization strategies. Notably, these techniques collectively yielded substantial reductions in model size and enhancements in inference speed that are essential for efficient MCU performance. A prime example of this impact is observed in the application of MobileNet v2. When compared with TFLM, our optimizations for MobileNet v2 resulted in significant improvements in speed and memory utilization, with runtime memory usage in SRAM reduced by 51% to 57% and flash memory savings ranging from 17% to 62%, alongside a reduction in inference speed by approximately 1.55 times. These results demonstrate that our optimization approach is highly effective in MCU environments, enhancing both computational efficiency and resource utilization.

The study's outcomes show promising directions for embedded AI, thus suggesting their potential for real-world applications and increased usage efficiencies. Our future goal will be the extension of this research to a broader array of neural network architectures. This expansion aims to validate our methods across various scenarios and contribute meaningfully to the field of deep-learning model optimization in embedded AI, particularly for MCUs. Additionally, acknowledging and addressing the challenges faced during this research will refine our approaches and strategies in future work.

ACKNOWLEDGMENTS

This research was supported by the Challengeable Future Defense Technology Research and Development Program through the Agency for Defense Development (ADD) funded by the Defense Acquisition Program Administration (DAPA) in 2022 (No. 915062201).

CONFLICT OF INTEREST STATEMENT

The authors declare that there are no conflicts of interest.

ORCID

Seungtae Hong https://orcid.org/0000-0003-0273-3542

REFERENCES

1. D. Lee, S.-J. Han, K.-W. Min, J. CHoi, and C. H. Park, *EMOS: Enhanced moving object detection and classification via sensor fusion and noise filtering*, ETRI J. **45** (2023), no. 5, 847–861.
2. S. Seo and H. Jung, *A robust collision prediction and detection method based on neural network for autonomous delivery robots*, ETRI J. **45** (2023), no. 2, 329–337.
3. F. Svoboda, J. Fernández-Marqués, E. Liberis, and N. D. Lane, *Deep learning on microcontrollers: A study on deployment costs and challenges* (Proc. 2nd Eur. Workshop Mach. Learn. Syst., Rennes, France), 2022. DOI 10.1145/3517207.3526978
4. J. Lin, W. M. Chen, Y. Lin, C. Gan, and S. Han, *MCUNet: Tiny deep learning on IoT devices*, Adv. Neural Inf. Process. Syst. **33** (2020), 11711–11722.
5. C. Banbury, C. Zhou, I. Fedorov, R. Matas, U. Thakker, D. Gope, V. J. Reddi, M. Mattina, and P. N. Whatmough, *Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers*, Proc. Mach. Learn. Syst. **3** (2021), 517–532.
6. R. David, J. Duke, A. Jain, V. Janapa Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, S. Regev, R. Rhodes, T. Wang, and P. Warden, *Tensorflow lite micro: embedded machine learning for TinyML systems*, Proc. Mach. Learn. Syst. **3** (2021), 800–811.
7. J. Lin, W. M. Chen, H. Cai, C. Gan, and S. Han, *MCUNetv2: Memory-efficient patch-based inference for tiny deep learning*, (Ann. Conf. Neural Inf. Process. Syst. (NeurIPS)), 2021.
8. P. E. Novac, G. B. Hacene, A. Pegatoquet, B. Miramond, and V. Gripon, *Quantization and deployment of deep neural networks on microcontrollers*, Sensors **21** (2021), no. 9. DOI 10.3390/s21092984
9. M. Bechtel, Q. T. Weng, and H. Yun, *DeepPicarMicro: applying TinyML to autonomous cyber physical systems*, (2022 IEEE 28th Int. Conf. Embedded Real-Time Comput. Syst. Appl. (RTCSA), Taipei, Taiwan), 2022. DOI 10.1109/RTCSA55878.2022.00019

22337326, 2025, 2, Downloaded from https://onlinelibrary.wiley.com/doi/10.4218/etrij.2023-0522 by Electronics and Telecommunications, Wiley Online Library on [24/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

10. H. K. Mean, X. Diao, H. Shi, H. DIng, H. Zhou, and C. Vaulx, *Trends and challenges in AIoT/IIoT/IoT implementation*, Sensors **23** (2023), no. 11. DOI 10.3390/s23115074
11. C. Li, Z. Wang, X. Wang, and H. Qi, *Single-shot channel pruning based on alternating direction method of multipliers*, arXiv Preprint, 2019. DOI 10.48550/arXiv.1902.06382
12. Y. He, X. Zhang, and J. Sun, *Channel pruning for accelerating very deep neural networks* (IEEE Int. Conf. Computer Vision, Venice, Italy), 2017. DOI 10.1109/ICCV.2017.155
13. S. Han, H. Mao, and W. J. Dally, *Deep compression: compressing deep neural networks with pruning, trained quantization and Huffman coding*, (Int. Conf. Learning Representations, San Juan, Puerto Rico), 2016.
14. Liu Zhuang, M. Sun, T. Zhou, G. Huang, and T. Darrell, *Rethinking the value of network pruning*, (Int. Conf. Learning Representations, New Orleans, LA, USA), 2018.
15. N. Liu, X. Ma, Z. Xu, Y. Wang, J. Tang, and J. Ye, *AutoCompress: An automatic dnn structured pruning framework for ultra-high compression rates*, Proc. AAAI COnf. Artif. Intell. **34**, (2020), 4876–4883.
16. G. Amir, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keytzer, *A survey of quantization methods for efficient neural network inference*, in low-power computer vision, in *Low-power computer vision*, Chapman and Hall/CRC, City, State, Country, 2022, pp. 291–326.
17. L. Tailin, J. Glossner, L. Wang, S. Shii, and X. Ahzng, *Pruning and quantization for deep neural network acceleration: a survey*, Neurocomput. **461** (2021), 370–403.
18. C. Sakr, S. Dai, R. Venkatesan, B. Zimmer, W. J. Dally, and B. Khailany, *Optimal clipping and magnitude-aware differentiation for improved quantization-aware training*, (Int. Conf. Mach. Learning, Baltimore, MA, USA) PMLR, 2022.
19. B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, *Quantization and training of neural networks for efficient integer-arithmetic-only inference* (Proc. IEEE Conf. Comput. Vision Pattern Recognit, Salt Lake City, UT, USA), 2018. DOI 10.1109/CVPR.2018.00286
20. M. Nagel, M. V. Baalen, T. Blankwvoort, and M. Welling, *Data-free quantization through weight equalization and bias correction* (Proc. IEEE/CVF Int. Conf. Comput, Vision, Seoul, Rep. of Korea), 2019. DOI 10.1109/ICCV.2019.00141
21. R. Banner, Y. Nahshan, and D. Soudry, *Post training 4-bit quantization of convolutional networks for rapid deployment*, Adv. Neural Inf. Process. Syst. **32** (2019), 7950–7958.
22. ARM, CMSIS-NN, Available from: https://github.com/ARM-software/CMSIS-NN [last accessed April 2024].
23. STM, STM32CudeIDE, Available from: https://www.st.com/en/development-tools/stm32cubeide.html [last accessed April 2024].
24. STM XCUBE-AI, Available from: https://www.st.com/en/embedded-software/x-cube-ai.html [last accessed April 2024].
25. Renesas E2Studio, Available from: https://www.renesas.com/us/en/software-tool/e-studio [last accessed April 2024].
26. Apache, microTVM, Available from: https://tvm.apache.org/docs/topic/microtvm/index.html [last accessed April 2024].
27. A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Rasion, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, *Pytorch: an imperative style, high-performance deep learning library* (Adv. Neural Inf. Process. Syst., Vancouver, Canada), 2019, 8026–8037.
28. M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, *Tensorflow: large-scale machine learning on heterogeneous distributed systems*, arXiv Preprint, 2016. DOI 10.48550/arXiv.1603.04467
29. Microsoft Embedded Learning Library (ELL), Available from: https://github.com/microsoft/ELL [last accessed April 2024].
30. Micro Tensor, Available from: https://github.com/uTensor/uTensor [last accessed April 2024].
31. J. Ma, *A higher-level neural network library on microcontrollers (NNoM)*, 2020.
32. ONNC, Available from: https://github.com/ONNC/onnc [last accessed April 2024].
33. J. Zeng, S.-B. Lin, Y. Yao, and D.-X. Zhou, *On ADMM in deep learning: convergence and saturation-avoidance*, J. Mach. Learning Res. **22** (2021), no. 1, 9024–9090.
34. Google, FlatBuffer, Available from: https://github.com/google/flatbuffers [last accessed April 2024].
35. Alibaba, TinyNeuralNetwork, Available from: https://github.com/alibaba/TinyNeuralNetwork [last accessed April 2024].
36. MLPerf, *Tiny deep learning benchmarks for embedded devices*, Available from: https://github.com/mlcommons/tiny [last accessed April 2024].

AUTHOR BIOGRAPHIES

Seungtae Hong received his BS, MS, and PhD degrees from the Department of Computer Science and Engineering at the Chonbuk National University, Jeonju, Republic of Korea, in 2008, 2010, and 2015, respectively. Since 2015, he has been with the Electronics and Communications Research Institute, where he is currently a senior researcher. His research interests include on-device deep learning, resource-aware computing, and embedded systems.

Gunju Park received his BS and MS degrees in Electrical and Computer Engineering from the University of Seoul, Republic of Korea, in 2019 and 2021, respectively. He has been a researcher at the Electronics and Communications Research Institute since 2023. His research interests include on-device deep learning.

22337326, 2025, 2, Downloaded from https://onlinelibrary.wiley.com/doi/10.4218/etrij.2023-0522 by Electronics and Telecommunications, Wiley Online Library on [24/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Jeong-Si Kim received her BS, MS, and PhD degrees in Computer Science from the Gyeongsang National University, JinJu, Republic of Korea, in 1992, 1995, and 1999, respectively. Since 2000, she has worked for the Electronics and Telecommunications Research Institute, Daejeon, South Korea, where she is now serving as a principal member of the engineering staff. Her research interests include on-device artificial intelligence, embedded systems, and debugging race conditions.

How to cite this article: S. Hong, G. Park, and J.-S. Kim, *Automated deep-learning model optimization framework for microcontrollers*, ETRI Journal **47** (2025), 179–192, DOI 10.4218/etrij.2023-0522

22337326, 2025, 2, Downloaded from https://onlinelibrary.wiley.com/doi/10.4218/etrij.2023-0522 by Electronics and Telecommunications, Wiley Online Library on [24/04/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License