Efficient Algorithms and Systems for Tiny Deep Learning

by

Ji Lin

B.Eng., Tsinghua University (2019)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2021

© Massachusetts Institute of Technology 2021. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
May 20, 2021

Certified by . . . .
Song Han
Assistant Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Efficient Algorithms and Systems for Tiny Deep Learning

by

Ji Lin

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2021, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

Tiny machine learning on IoT devices based on microcontroller units (MCUs) enables various real-world applications (*e.g.*, keyword spotting, anomaly detection). However, deploying deep learning models to MCUs is challenging due to the limited memory size: the memory of microcontrollers is 2-3 orders of magnitude smaller even than mobile phones. In this thesis, we study efficient algorithms and systems for tiny-scale deep learning. We propose MCUNet, a framework that jointly designs the efficient neural architecture (TinyNAS) and the lightweight inference engine (TinyEngine), enabling ImageNet-scale inference on microcontrollers. TinyNAS adopts a two-stage neural architecture search approach that first optimizes the search space to fit the resource constraints, then specializes the network architecture in the optimized search space. TinyNAS can automatically handle diverse constraints (*i.e.* device, latency, energy, memory) under low search costs. TinyNAS is co-designed with TinyEngine, a memory-efficient inference library to expand the search space and fit a larger model. TinyEngine adapts the memory scheduling according to the *overall* network topology rather than *layer-wise* optimization, reducing the memory usage by $3.4\times$, and accelerating the inference by 1.7-3.3$\times$ compared to TF-Lite Micro [3] and CMSIS-NN [31]. For vision applications on MCUs, we diagnosed and found that existing convolutional neural network (CNN) designs have an imbalanced peak memory distribution: the first several layers have much higher peak memory usage than the rest of the network. Based on the observation, we further extend the framework to support patch-based inference to break the memory bottleneck of the initial stage. MCUNet is the first to achieves $>$70% ImageNet top1 accuracy on an off-the-shelf commercial microcontroller, using 3.5$\times$ less SRAM and 5.7$\times$ less Flash compared to quantized MobileNetV2 and ResNet-18. On visual&audio wake words tasks, MCUNet achieves state-of-the-art accuracy and runs 2.4-3.4$\times$ faster than MobileNetV2 and ProxylessNAS-based solutions with 3.7-4.1$\times$ smaller peak SRAM. Our study suggests that the era of always-on tiny machine learning on IoT devices has arrived.

Thesis Supervisor: Song Han
Title: Assistant Professor of Electrical Engineering and Computer Science

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my advisor, Professor Song Han. Through my undergraduate study, gap year, and graduate study, he has provided me invaluable guidance, not only in academic aspects but also in life. With his comprehensive understanding of algorithms and hardware and deep insights from academia and industry, Professor Han has inspired me to conduct impactful research and become a better researcher.

I sincerely appreciate my undergraduate advisors, Professor Yu Wang, for introducing me into the world of efficient deep learning and hardware acceleration, and Professor Jiwen Lu for leading me into computer vision research. I would like to thank Dr. Huazhe Xu and Dr. Yang Gao for guiding my undergraduate research. I would like to thank Dr. Jia Li for the mentorship during my gap year. I would also like to thank Prof. Jun-Yan Zhu and Dr. Richard Zhang for the wonderful summer at Adobe.

This thesis would not have been possible without my collaborators. In particular, I would like to thank Wei-Ming Chen for system-related support and experiments. I would like to thank Han Cai, Yujun Lin, Dr. John Cohn, and Dr. Chuang Gan for their valuable contribution to this thesis. Thanks also go to other labmates, Zhijian Liu, Hanrui Wang, Kuan Wang, and Ligeng Zhu, for the constructive feedback and inspiring discussions.

Part of this work was published in the 2020 Conference on Neural Information Processing Systems. I would like to thank MIT-IBM Watson AI Lab, Qualcomm, NSF CAREER Award #1943349 and NSF RAPID Award #2027266 for supporting this research, and MIT Satori cluster for generously providing the computation resource.

Finally, I would like to thank my partner Siyu Yao for her accompany and support along the journey, helping me through all the difficult times. I would like to thank my parents, Zhuang Lin and Huating Yan, for their lifelong unconditional love. All my achievements would not be possible without them.

Contents

1 Introduction **17**

2 Background **21**

2.1 Microcontroller Basics 21

2.2 Deep Learning Inference on Microcontrollers 21

2.3 Efficient Neural Network 22

3 MCUNet: System-Algorithm Co-Design **25**

3.1 TinyNAS: Two-Stage NAS for Tiny Memory Constraints 25

3.1.1 Automated search space optimization. 25

3.1.2 Resource-constrained model specialization. 27

3.2 TinyEngine: A Memory-Efficient Inference Library 27

3.2.1 Code generator-based compilation. 28

3.2.2 Model-adaptive memory scheduling. 29

3.2.3 Computation kernel specialization. 30

3.2.4 In-place depth-wise convolution 30

4 MCUNetV2: Memory-Efficient Patched-based Inference **33**

4.1 Rethinking the Memory Bottleneck of TinyML 33

4.1.1 Imbalanced Memory Usage Distribution 33

4.1.2 Reducing Memory with Per-patch Computation 35

4.1.3 Memory Saving & Computation Overhead 36

4.2 Reducing Computation Overhead by Redistributing Receptive Field 38

4.3 Joint Search Space Scaling and Neural Architecture Search 39

5 Results **41**

5.1 Setups 41

5.2 Large-Scale Image Recognition on Tiny Devices 42

5.3 Visual&Audio Wake Words 44

5.4 Reduce Memory Usage with Patch-based Inference 46

5.5 Object Detection on MCUs 47

5.6 Analysis 48

6 Applications **53**

7 Conclusion **57**

A Appendix **59**

List of Figures

1-1 Example applications of TinyML on MCUs. Deploying tiny deep learning models on MCUs can enable various low-power, low-cost AI applications, handling sensor inputs of different modalities. 18

1-2 MobileNetV2 [49] has a very *imbalanced* memory usage distribution. The peak memory is determined by first several blocks with high memory usage, while the later blocks all share a small memory usage. By using re-computation (2×2 granularity), we are able to significantly reduce the memory usage of the first 5 blocks, and reduce the overall peak memory by 11×, fitting under 256kB memory budget (a common microcontroller configuration like STM32F412). The memory usage is measured using FP32. 19

3-1 MCUNet jointly designs the neural architecture and the inference scheduling to fit the tight memory resource on microcontrollers. TinyEngine makes full use of the limited resources on MCU, allowing a larger design space for architecture search. With a larger degree of design freedom, TinyNAS is more likely to find a high accuracy model compared to using existing frameworks. 26

3-2 TinyNAS selects the best search space by analyzing the FLOPs CDF of different search spaces. Each curve represents a design space. Our insight is that the design space that is more likely to produce *high FLOPs* models under the memory constraint gives higher model capacity, thus more likely to achieve high accuracy. For the solid **red** space, the top 20% of the models have >50.3M FLOPs, while for the solid **black** space, the top 20% of the models only have >32.3M FLOPs. Using the solid **red** space for neural architecture search achieves 78.7% final accuracy, which is 4.5% higher compared to using the **black** space. The legend is in format: `w{width}-r{resolution}|{mean FLOPs}`. 27

3-3 TinyEngine achieves higher inference efficiency than existing inference frameworks while reducing the memory usage. **Up**: TinyEngine is 3× and 1.6× faster than TF-Lite Micro and CMSIS-NN, respectively. Note that if the required memory exceeds the memory constraint, it is marked with "OOM" (out of memory). **Down**: By reducing the memory usage, TinyEngine can run various model designs with tiny memory, enlarging the design space for TinyNAS under the limited memory of MCU. Model details in supplementary. 28

3-4 TinyEngine outperforms existing libraries by eliminating runtime overheads, specializing each optimization technique, and adopting in-place depth-wise convolution. This effectively enlarges design space for TinyNAS under a given latency/memory constraint. 29

3-5 Binary size. 29

3-6 TinyEngine reduces peak memory by performing in-place depth-wise convolution. **Left:** Conventional depth-wise convolution requires 2N memory footprint for activations. **Right:** in-place depth-wise convolution reduces the memory of depth-wise convolutions to N+1. Specifically, the output activation of the first channel is stored in a temporary buffer. Then, for each following channel, the output activation overwrites the input activation of its previous channel. Finally, the output activation of the first channel stored in the buffer is written back to the input activation of the last channel. . . . 30

4-1 Apart from MobileNetV2 [49] (Figure 1-2), ProxylessNAS-Mobile [6] and MCUNet [35] also suffer from a highly imbalanced memory usage distribution. Note that the memory profiling is done with fp32. The MCUNet model was designed to fit the memory constraint when quantized in int8. . 34

4-2 Object detection (PASCAL VOC [16]) suffers a larger degradation from reduced resolutions than image classification (ImageNet [14]). 35

4-3 Per-patch computation can significantly reduce the peak memory required to execute a sequence of convolutional layers. Here we study two convolutional layers (with stride 1 and 2). Under per-layer computation (a), the first convolution has large input/output activation size, dominating the peak memory requirement. With per-patch computation (b), we allocate the buffer to host the final output activation, and compute the results *patch-by-patch*. We only need to store the activation from *one patch* but not the entire feature map, reducing the peak memory usage significantly (the first input is the input image, which can be partially decoded from a compressed format like JPEG) 35

4-4 Ablation study on spatial partial computation. (a) The peak memory generally goes down with more blocks being partially computed, and then goes up due to the large patch size. The optimal index is chosen when the output feature map is spatially down-sampled by 8×. (b) The receptive field $R^{5\to0}$ (also the patch size) grows with the output patch size (OP). For $OP \leq 7$, the peak memory is bounded by the rest of the network; for $OP \geq 14$, the patch is too large, causing peak memory increase. We choose $OP = 7$ to achieve reduced peak memory and minimum computation overhead. 37

4-5 The original MobileNetV2 [49] (upper) and the one with redistributed receptive field (lower). We reduce the receptive field in the initial stage with large peak memory (left), leading to smaller repeated computation with per-patch inference. We then increase the receptive field in the rest of the network to maintain the overall performance. 39

5-1 MCUNet reduces the the SRAM memory by **3.5**× and Flash usage by **5.7**× compared to MobileNetV2 and ResNet-18 (8-bit), while achieving better accuracy (70.7% *vs*. 69.8% ImageNet top-1). 43

5-2 Accuracy *vs*. latency/SRAM memory trade-off on VWW (**top**) and Speech Commands (**down**) dataset. MCUNet achieves better accuracy while being 2.4-3.4× faster at 3.7-4.1× smaller peak SRAM. 44

5-3 (a) With patch-based inference ("MCUNetV2"), we are can greatly improve the accuracy *vs*. peak SRAM trade-off compared to per-layer execution ("per-layer") and existing state-of-the-art methods. Under the same peak SRAM, MCUNetV2 improves the accuracy by 3.3%. It can also achieve >90% accuracy under 64kB memory, 1.9× smaller than per-layer execution, paving the way for even smaller hardware resource. (b) MCUNetV2 also has a better accuracy *vs*. latency trade-off thanks to the broadened search space. Compared to MCUNet, MCUNetV2 can obtain higher accuracy at 1.5× faster inference; compared to MobileNetV2+TF-Lite Micro, MCUNetV2 improves the accuracy by 4.7% at the same latency. 46

5-4 Search space with higher mean FLOPs leads to higher final accuracy. 48
5-5 Best search space configurations under different SRAM and Flash constraints. 48
5-6 Evolutionary search can find specialized networks with better accuracy compared to random search. Evolving on TinyEngine provides a larger search space, enabling us to find better models compared to CMSIS-NN. . . 49

6-1 MCUNet for visual wake words on STM32F746 MCU. MCUNet can achieve 12% higher accuracy at 2.5× faster inference speed compared to the widely used industrial solution (MobileNet [28]+TF-Lite Micro [3]). The complete demo video can be found at https://youtu.be/YvioBgtec4U. 54
6-2 MCUNet for face and mask detection. The model can detect multiple human faces appearing in the image, and whether they are wearing face masks properly (green bounding boxes) or not (red bounding boxes). The complete demo video can be found at https://www.youtube.com/watch?v=AFsExPbxIiA. 55
6-3 MCUNet for person detection on STM32F746 MCU. MCUNet can detect multiple persons at different sizes and scales. The complete demo video can be found at https://www.youtube.com/watch?v=cr5y_AJ-LM4. 56

A-1 Total CO_2 emission (klbs) for model design. MCUNet saves the design cost by orders of magnitude, allowing model specialization for different deployment scenarios. 60

List of Tables

1.1 **Left**: Microcontrollers have 3 orders of magnitude *less* memory and storage compared to mobile phones, and 5-6 orders of magnitude less than cloud GPUs. The extremely limited memory makes deep learning deployment difficult. **Right**: The peak memory and storage usage of widely used deep learning models. ResNet-50 exceeds the resource limit on microcontrollers by $100\times$, MobileNet-V2 exceeds by $20\times$. Even the int8 quantized MobileNetV2 requires $5.3\times$ larger memory and can't fit a microcontroller. 18

5.1 System-algorithm co-design (TinyEngine + TinyNAS) achieves the highest ImageNet accuracy of models runnable on a microcontroller. 42

5.2 MCUNet outperforms the baselines at various latency requirements. Both TinyEngine and TinyNAS bring significant improvement on ImageNet. . . . 42

5.3 MCUNet can handle diverse hardware resource on different MCUs. It outperforms [48] without using advanced mixed-bit quantization (8/4/2-bit) policy under different resource constraints, achieving a record ImageNet accuracy (>70%) on microcontrollers. 43

5.4 MCUNetV2 with patch-based inference further improves the accuracy of ImageNet classification under various MCU hardware constraints, leading to 3% better accuracy under the same quantization scheme. 45

5.5 MCUNet improves the detection mAP by 20% on Pascal VOC under 512kB SRAM constraint. With MCUNet, we are able to fit a model with much larger capacity and computation FLOPs at a smaller peak memory. MobileNet-v2 + CMSIS-NN is bounded by the memory consumption: it can only fit a model with 34M FLOPs even when the peak memory slightly exceeds the budget, leading to inferior detection performance. 47

5.6 Our search space achieves the best accuracy, closer to ResNet-18@224 resolution (OOM). Randomly sampled and a huge space (contain many configs) leads to worse accuracy. 48

5.7 Per-patch inference can reduce the peak memory by 8× for MobileNetV2 [49] (1372kB to 172kB), but it increases the computation by 13% (300M to 339M) due to a large patch receptive field (RF). Splitting input image into non-overlapping patches (Input split) can retain the computation, but it breaks the translation-invariance of CNN and leads to a large degradation in object detection. Re-distributing receptive field (Re-distribute) can reduce receptive field and computation overhead (only 3.7%) while getting same performance compared to the original network. We denote degraded items with red color. 50

5.8 Our joint search algorithm outperforms existing state-of-the-art tiny networks in terms of *computation-accuracy* trade-off, especially under tiny computation setting (<50M). All of ours models are derived from *the same search space*, while obtaining the best accuracy at the same computation. For models with *, we re-measure the MACs and parameters using our profiler. 51

6.1 Statistics of the deployed models for different applications. 53

Chapter 1

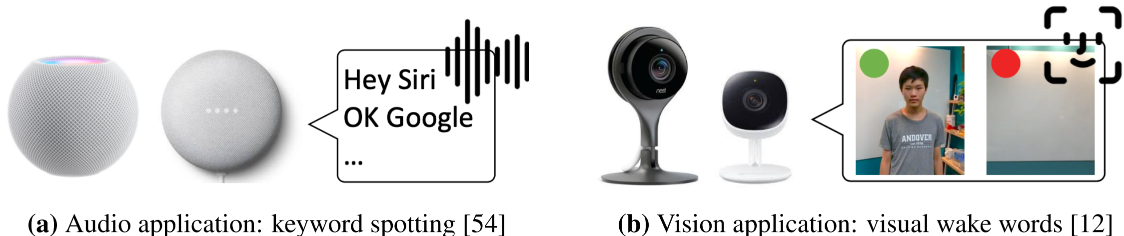
Introduction

The number of IoT devices based on always-on microcontrollers is increasing rapidly at a historical rate, reaching 250B [2], enabling numerous applications, including smart manufacturing, personalized healthcare, precision agriculture, automated retail, *etc*. These low-cost, low-energy microcontrollers give rise to a brand new opportunity of tiny machine learning (TinyML). By running deep learning models on these tiny devices, we can directly perform data analytics near the sensor, thus dramatically expand the scope of AI applications.

However, microcontrollers have a very limited resource budget, especially memory (SRAM) and storage (Flash). The on-chip memory is 3 orders of magnitude smaller than mobile devices, and 5-6 orders of magnitude smaller than cloud GPUs, making deep learning deployment extremely difficult. As shown in Table 1.1, a state-of-the-art ARM Cortex-M7 MCU only has 320kB SRAM and 1MB Flash storage, which is impossible to run off-the-shelf deep learning models: ResNet-50 [23] exceeds the storage limit by $100\times$, MobileNetV2 [49] exceeds the peak memory limit by $22\times$. Even the int8 quantized version of MobileNetV2 still exceeds the memory limit by $5.3\times$*, showing a big gap between the desired and available hardware capacity.

Different from the cloud and mobile devices, microcontrollers are bare-metal devices that do not have an operating system. Therefore, we need to jointly design the deep learning model and the inference library to efficiently manage the tiny resources and fit the tight memory&storage budget. Existing efficient network design [28, 49, 56] and neural

*Not including the runtime buffer overhead (*e.g.*, Im2Col buffer); the actual memory consumption is larger.



(a) Audio application: keyword spotting [54] **(b)** Vision application: visual wake words [12]

Figure 1-1. Example applications of TinyML on MCUs. Deploying tiny deep learning models on MCUs can enable various low-power, low-cost AI applications, handling sensor inputs of different modalities.

	Cloud AI (NVIDIA V100)		**Mobile AI** (iPhone 11)		**Tiny AI** (STM32F746)		**ResNet-50**	**MobileNetV2**	**MobileNetV2 (int8)**
Memory	16 GB	4× →	4 GB	3100× →	**320 kB**	← gap →	7.2 MB	6.8 MB	1.7 MB
Storage	TB~PB	1000× →	>64 GB	64000× →	**1 MB**	← gap →	102MB	13.6 MB	3.4 MB

Table 1.1. Left: Microcontrollers have 3 orders of magnitude *less* memory and storage compared to mobile phones, and 5-6 orders of magnitude less than cloud GPUs. The extremely limited memory makes deep learning deployment difficult. **Right**: The peak memory and storage usage of widely used deep learning models. ResNet-50 exceeds the resource limit on microcontrollers by 100×, MobileNet-V2 exceeds by 20×. Even the int8 quantized MobileNetV2 requires 5.3× larger memory and can't fit a microcontroller.

architecture search methods [51, 6, 55, 5] focus on GPU or smartphones, where both memory and storage are abundant. Therefore, they only optimize to reduce FLOPs or latency, and the resulting models cannot fit microcontrollers. There is limited literature [17, 34, 48, 32] that studies machine learning on microcontrollers. However, due to the lack of system-algorithm co-design, they either study tiny-scale datasets (*e.g.*, CIFAR or sub-CIFAR level), which are far from real-life use case, or use weak neural networks that cannot achieve decent performance.

In this paper, we propose MCUNet, a system-model co-design framework that enables ImageNet-scale deep learning on off-the-shelf microcontrollers. To handle the scarce on-chip memory on microcontrollers, we jointly optimize the deep learning model design (TinyNAS) and the inference library (TinyEngine) to reduce the memory usage. TinyNAS is a two-stage neural architecture search (NAS) method that can handle the tiny and diverse memory constraints on various microcontrollers. The performance of NAS highly depends on the search space [44], yet there is little literature on the search space design heuristics at the tiny scale. TinyNAS addresses the problem by first optimizing the search space

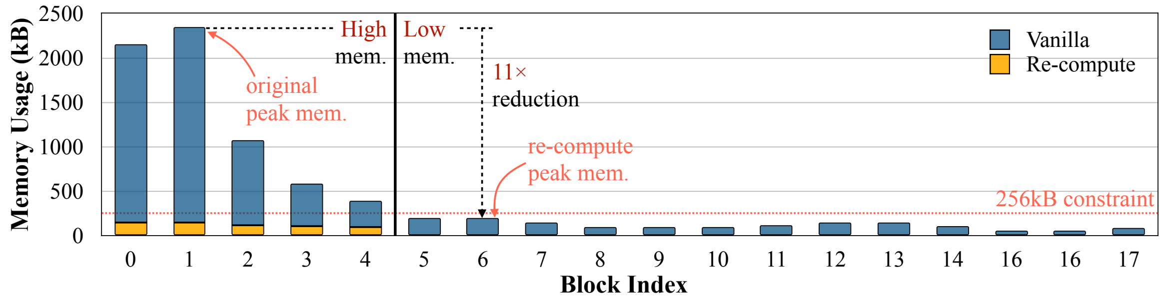


Figure 1-2. MobileNetV2 [49] has a very *imbalanced* memory usage distribution. The peak memory is determined by first several blocks with high memory usage, while the later blocks all share a small memory usage. By using re-computation (2×2 granularity), we are able to significantly reduce the memory usage of the first 5 blocks, and reduce the overall peak memory by 11×, fitting under 256kB memory budget (a common microcontroller configuration like STM32F412). The memory usage is measured using FP32.

automatically to fit the tiny resource constraints, then performing neural architecture search in the optimized space. Specifically, TinyNAS generates different search spaces by scaling the input resolution and the model width, then collects the computation FLOPs distribution of satisfying networks within the search space to evaluate its priority. TinyNAS relies on the insight that *a search space that can accommodate higher FLOPs under memory constraint can produce better model*. Experiments show that the optimized space leads to better accuracy of the NAS searched model. To handle the extremely tight resource constraints on microcontrollers, we also need a memory-efficient inference library to eliminate the unnecessary memory overhead, so that we can expand the search space to fit larger model capacity with higher accuracy. TinyNAS is co-designed with TinyEngine to lift the ceiling for hosting deep learning models. TinyEngine improves over the existing inference library with code generator-based compilation method to eliminate memory overhead . It also supports model-adaptive memory scheduling: instead of *layer-wise* optimization, TinyEngine optimizes the memory scheduling according to the *overall* network topology to get a better strategy. Finally, it performs *specialized* computation kernel optimization (*e.g.*, loop tiling, loop unrolling, op fusion, *etc.*) for different layers, which further accelerates the inference.

We further analyze convolutional neural networks (CNNs), the widely used backbone for computer vision-based applications. We find that existing CNNs suffer from an *imbalanced* memory distribution problem. Take MobileNetV2 [49] as an example: the peak memory usage of the early stage is 11× higher compared to the later blocks (Figure 1-2), which

becomes the memory bottleneck of the entire network. When executing the neural network *layer-by-layer*, the majority of the network backbone (after block index 5) has a small memory usage, which can even fit a common microcontroller model with 256kB memory. If we can find a way to "bypass" the early stage, we can reduce the peak memory by 11× times (denoted by the yellow bars). To this end, we proposed MCUNetV2, which breaks the memory bottleneck of the initial stage with *patch-by-patch* inference. Such a computation order executes the initial stage on only a small patch of the full image at a time (*e.g.*, 56×56 *vs*. 224×224), effectively reducing the memory required for inference. However, patch-based inference brings spatial overlapping between patches, leading to repeated computation. For MobileNetV2, the largest overhead can increase the overall computation from 300M MACs to 738M MACs. We propose techniques to re-distribute the receptive field distribution of the backbone to reduce the computation overhead to 3%. To automate the re-distribution process and balance between accuracy and cost, we propose joint search space scaling and neural architecture search to find a good architecture for tiny hardware constraints automatically.

MCUNet dramatically pushes the limit of deep network performance on microcontrollers. TinyEngine reduces the peak memory usage by 3.4× and accelerates the inference by 1.7-3.3× compared to TF-Lite and CMSIS-NN, allowing us to run a larger model. With system-algorithm co-design, MCUNet (TinyNAS+TinyEngine) achieves a record ImageNet top-1 accuracy of 70.7% on an off-the-shelf commercial microcontroller. On visual&audio wake words tasks, MCUNet achieves state-of-the-art accuracy and runs 2.4-3.4× faster than existing solutions at 3.7-4.1× smaller peak SRAM. For interactive applications, our solution achieves 10 FPS with 91% top-1 accuracy on Speech Commands dataset. Our study suggests that the era of tiny machine learning on IoT devices has arrived.

Chapter 2

Background

2.1 Microcontroller Basics

Microcontrollers have tight memory, making it difficult to host deep learning models. For example, a popular ARM Cortex-M7 MCU STM32F746 only has 320kB SRAM and 1MB Flash. Other MCUs with Cortex-M4 CPUs have even lower memory budgets. Therefore, we have to carefully design the inference library and the deep learning models to fit the tight memory constraints. In deep learning scenarios, SRAM is related to the activation size (read&write); Flash is related to the model size (read-only). Specifically, SRAM stores the input and output activation of the current layer, the buffer to store partial weights, the buffers for optimization (*e.g.*, Im2col buffer), *etc*. Flash is used to host the weight and architecture of the deep learning models, the program code (instructions, constant variables), and the framework interpreter (e.g., TensorFlow-Lite [3] interpreter). Some microcontrollers support external memory like SD cards. However, accessing the external memory is slow and consumes $100\times$ more energy compared to on-chip memory [26], which is prohibitive under a low-power setting.

2.2 Deep Learning Inference on Microcontrollers

Deep learning inference on microcontrollers is a fast-growing area. TensorFlow Lite [3] is among the first deep learning framework to support bare-metal microcontrollers. CMSIS-NN [31] implements optimized kernels to improve inference speed and energy efficiency.

CMix-NN [8] supports mixed precision on MCU to reduce memory footprint. Recently, TVM [9] and AutoTVM [10] also supports microcontrollers (named as μTVM/microTVM).

Existing frameworks have several limitations: 1. Most frameworks rely on an interpreter to interpret the network graph at runtime, which will consume a lot of SRAM and Flash (up to 65% of peak memory) and increase latency by 22%. 2. The optimization is performed at layer-level, which fails to utilize the overall network architecture information to further reduce memory usage.

2.3 Efficient Neural Network

Network efficiency is very important for the overall performance of the deep learning system. One way is to compress off-the-shelf networks by pruning [22, 25, 36, 40, 24, 39] and quantization [21, 58, 45, 57, 13, 11, 53] to remove redundancy and reduce complexity. Tensor decomposition [33, 18, 29] also serves as an effective compression method. Researchers also propose automated methods to achieve high accuracy in model compression [24, 53, 39]. Another way is to directly design an efficient and mobile-friendly network [28, 49, 42, 56, 42].

Recently, neural architecture search (NAS) [59, 60, 37, 6, 51, 55] dominates efficient network design. Designing neural network architecture is quite time-consuming and requires domain expertise. NAS can automate the process and design high-accuracy model architectures with reinforcement learning [59, 60, 51], evolutionary search [19, 5], gradient-based method [37, 6], *etc*. Recently, researchers develop NAS methods to automatically design efficient and hardware-friendly models [51, 6, 55, 5], facilitating edge deployment.

The performance of NAS highly depends on the quality of the search space [44]. Traditionally, people follow manual design heuristics for NAS search space design. For example, the widely used mobile-setting search space [51, 6, 55] originates from MobileNetV2 [49]: they both use 224 input resolution and a similar base channel number configurations, while searching for kernel sizes, block depths, and expansion ratios. However, there lack standard model designs for microcontrollers with limited memory, so as the search space design. One possible way is to manually tweak the search space for each microcontroller. But manual tuning through trials and errors is labor-intensive, making it prohibitive for a large number

of deployment constraints (*e.g.*, STM32F746 has 320kB SRAM/1MB Flash, STM32H743 has 512kB SRAM/2MB Flash, latency requirement 5FPS/10FPS). Therefore, we need a way to automatically optimize the search space for tiny and diverse deployment scenarios.

Chapter 3

MCUNet: System-Algorithm Co-Design

We propose MCUNet, a system-algorithm co-design framework that jointly optimizes the NN architecture (TinyNAS) and the inference scheduling (TinyEngine) in a same loop (Figure 3-1). Compared to traditional methods that either (a) optimizes the neural network using neural architecture search based on a given deep learning library (*e.g.*, TensorFlow, PyTorch) [51, 6, 55], or (b) tunes the library to maximize the inference speed for a given network [9, 10], MCUNet can better utilize the resources by system-algorithm co-design.

3.1 TinyNAS: Two-Stage NAS for Tiny Memory Constraints

TinyNAS is a two-stage neural architecture search method that first optimizes the search space to fit the tiny and diverse resource constraints, and then performs neural architecture search within the optimized space. With an optimized space, it significantly improves the accuracy of the final model.

3.1.1 Automated search space optimization.

We propose to optimize the search space automatically at *low cost* by analyzing the computation distribution of the satisfying models. To fit the tiny and diverse resource constraints of different microcontrollers, we scale the *input resolution* and the *width multiplier* of the mobile search space [51]. We choose from an input resolution spanning $R = \{48, 64, 80, ..., 192, 208, 224\}$ and a width multiplier $W = \{0.2, 0.3, 0.4, ..., 1.0\}$ to cover a wide spectrum of resource constraints. This leads to $12 \times 9 = 108$ possible search

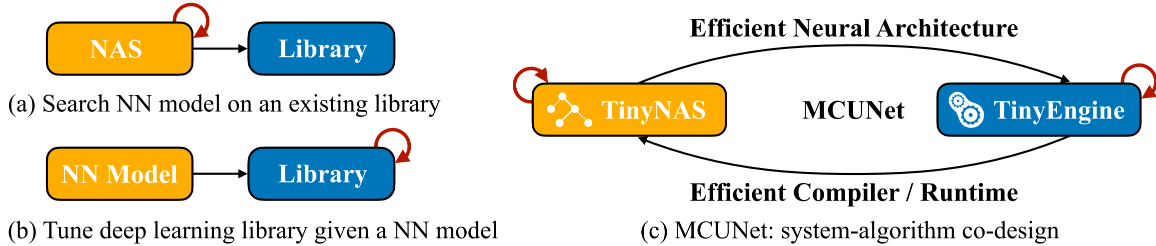


Figure 3-1. MCUNet jointly designs the neural architecture and the inference scheduling to fit the tight memory resource on microcontrollers. TinyEngine makes full use of the limited resources on MCU, allowing a larger design space for architecture search. With a larger degree of design freedom, TinyNAS is more likely to find a high accuracy model compared to using existing frameworks.

space configurations $S = W \times R$. Each search space configuration contains 3.3×10^{25} possible sub-networks. Our goal is to find the best search space configuration S^* that contains the model with the highest accuracy while satisfying the resource constraints.

Finding S^* is non-trivial. One way is to perform neural architecture search on each of the search spaces and compare the final results. But the computation would be astronomical. Instead, we evaluate the quality of the search space by randomly sampling m networks from the search space and comparing the distribution of satisfying networks. Instead of collecting the Cumulative Distribution Function (CDF) of each satisfying network's *accuracy* [43], which is computationally heavy due to tremendous training, we only collect the CDF of *FLOPs* (see Figure 3-2). The intuition is that, within the same model family, the accuracy is usually positively related to the computation [7, 24]. A model with larger computation has a larger capacity, which is more likely to achieve higher accuracy. We further verify the the assumption in Section 5.6.

As an example, we study the best search space for ImageNet-100 (a 100 class classification task taken from the original ImageNet) on STM32F746. We show the FLOPs distribution CDF of the top-10 search space configurations in Figure 3-2. We sample $m = 1000$ networks from each space and use TinyEngine to optimize the memory scheduling for each model. We only keep the models that satisfy the memory requirement at the best scheduling. To get a quantitative evaluation of each space, we calculate the average FLOPs for each configuration and choose the search space with the largest average FLOPs. For example, according to the experimental results on ImageNet-100, using the solid red space (average FLOPs 52.0M) achieves 2.3% better accuracy compared to using the solid

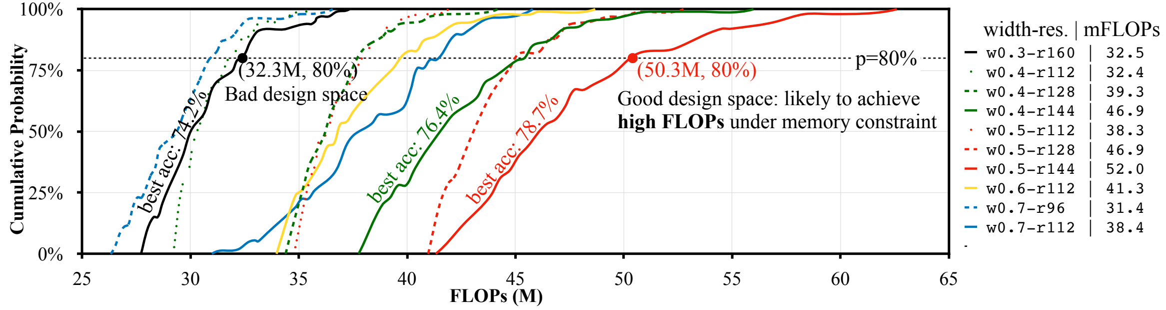


Figure 3-2. TinyNAS selects the best search space by analyzing the FLOPs CDF of different search spaces. Each curve represents a design space. Our insight is that the design space that is more likely to produce *high FLOPs* models under the memory constraint gives higher model capacity, thus more likely to achieve high accuracy. For the solid **red** space, the top 20% of the models have >50.3M FLOPs, while for the solid **black** space, the top 20% of the models only have >32.3M FLOPs. Using the solid **red** space for neural architecture search achieves 78.7% final accuracy, which is 4.5% higher compared to using the **black** space. The legend is in format: `w{width}-r{resolution}|{mean FLOPs}`.

green space (average FLOPs 46.9M), showing the effectiveness of automated search space optimization. We will elaborate more on the ablations in Section 5.6.

3.1.2 Resource-constrained model specialization.

To specialize network architecture for various microcontrollers, we need to keep a low neural architecture search cost. After search space optimization for each memory constraint, we perform one-shot neural architecture search [4, 19] to efficiently find a good model, reducing the search cost by 200× [6]. We train one super network that contains all the possible sub-networks through *weight sharing* and use it to estimate the performance of each sub-network. We then perform evolution search to find the best model within the search space that meets the on-board resource constraints while achieving the highest accuracy. For each sampled network, we use TinyEngine to optimize the memory scheduling to measure the optimal memory usage. With such kind of co-design, we can efficiently fit the tiny memory budget. The details of super network training and evolution search can be found in the supplementary.

3.2 TinyEngine: A Memory-Efficient Inference Library

Researchers used to assume that using different deep learning frameworks (libraries) will only affect the *inference speed* but not the *accuracy* . However, this is not the case for

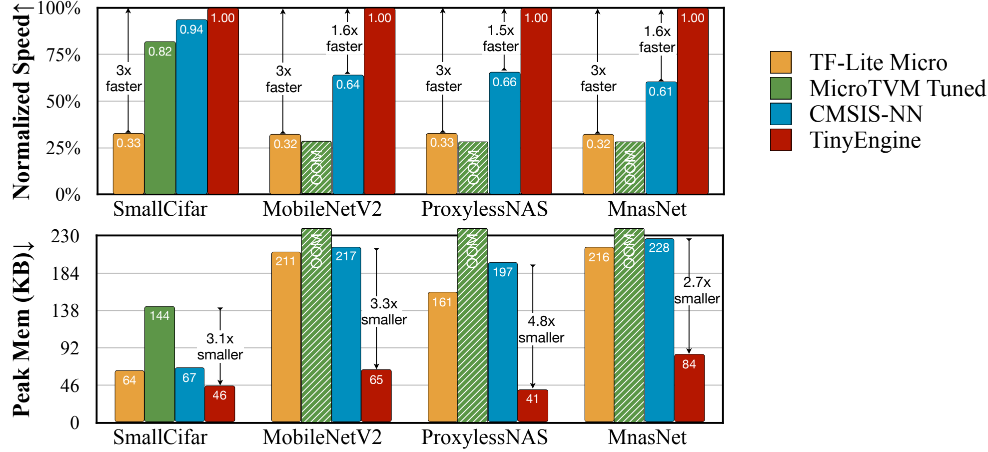


Figure 3-3. TinyEngine achieves higher inference efficiency than existing inference frameworks while reducing the memory usage. **Up**: TinyEngine is 3× and 1.6× faster than TF-Lite Micro and CMSIS-NN, respectively. Note that if the required memory exceeds the memory constraint, it is marked with "OOM" (out of memory). **Down**: By reducing the memory usage, TinyEngine can run various model designs with tiny memory, enlarging the design space for TinyNAS under the limited memory of MCU. Model details in supplementary.

TinyML: the efficiency of the inference library matters a lot to both the latency and accuracy of the searched model. Specifically, a good inference framework will make full use of the limited resources in MCU, avoiding waste of memory, and allow a larger search space for architecture search. With a larger degree of design freedom, TinyNAS is more likely to find a high accuracy model. Thus, TinyNAS is co-designed with a memory-efficient inference library, TinyEngine.

3.2.1 Code generator-based compilation.

Most existing inference libraries (*e.g.*, TF-Lite Micro, CMSIS-NN) are interpreter-based. Though it is easy to support cross-platform development, it requires extra memory, the most expensive resource in MCU, to store the meta-information (such as model structure parameters). Instead, TinyEngine only focuses on MCU devices and adopts code generator-based compilation. It not only avoids the time for runtime interpretation, but also frees up the memory usage to allow design and inference of larger models. Compared to CMSIS-NN, TinyEngine reduced memory usage by 2.1× and improve inference efficiency by 22% via code generation, as shown in Figures 3-3 and 3-4.

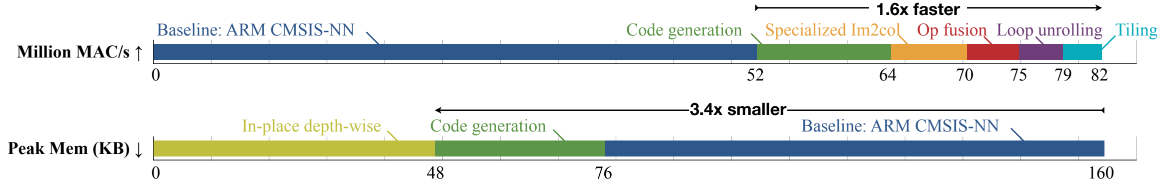


Figure 3-4. TinyEngine outperforms existing libraries by eliminating runtime overheads, specializing each optimization technique, and adopting in-place depth-wise convolution. This effectively enlarges design space for TinyNAS under a given latency/memory constraint.

The binary size of TinyEngine is light-weight, making it very memory-efficient for MCUs. Unlike interpreter-based TF-Lite Micro, which prepares the code for *every* operation (e.g., conv, softmax) to support cross-model inference even if they are not used, which has high redundancy. TinyEngine only compiles the operations that are used by a given model into the binary. As shown in Figure 3-5, such model-adaptive compilation reduces code size by up to 4.5× and 5.0× compared to TF-Lite Micro and CMSIS-NN, respectively.

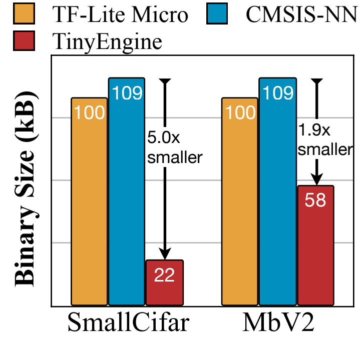


Figure 3-5. Binary size.

3.2.2 Model-adaptive memory scheduling.

Existing inference libraries schedule the memory for each layer solely based on the layer itself: in the very beginning, a large buffer is designated to store the input activations after im2col; when executing each layer, only one column of the transformed inputs takes up this buffer. This leads to poor input activation reuse. Instead, TinyEngine smartly adapts the memory scheduling to the model-level statistics: the *maximum* memory M required to fit exactly one column of transformed inputs over all the layers $\boldsymbol{L}$,

$$M = \max\left(\text{kernel size}_{L_i}^2 \cdot \text{in channels}_{L_i}; \forall L_i \in \boldsymbol{L}\right). \tag{3.1}$$

For each layer L_j, TinyEngine tries to tile the computation loop nests so that, as many columns can fit in that memory as possible,

$$\text{tiling size of feature map width}_{L_j} = \lfloor M / \left(\text{kernel size}_{L_j}^2 \cdot \text{in channels}_{L_j}\right) \rfloor. \tag{3.2}$$

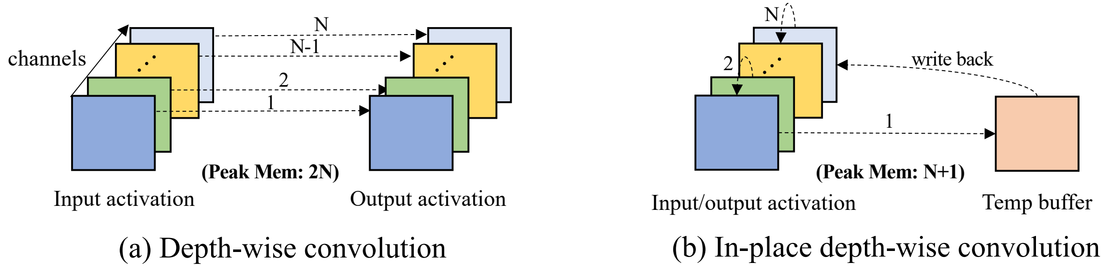


Figure 3-6. TinyEngine reduces peak memory by performing in-place depth-wise convolution. **Left:** Conventional depth-wise convolution requires 2N memory footprint for activations. **Right:** in-place depth-wise convolution reduces the memory of depth-wise convolutions to N+1. Specifically, the output activation of the first channel is stored in a temporary buffer. Then, for each following channel, the output activation overwrites the input activation of its previous channel. Finally, the output activation of the first channel stored in the buffer is written back to the input activation of the last channel.

Therefore, even for the layers with the same configuration (*e.g.*, kernel size, #in/out channels) in two different models, TinyEngine will provide different strategies. Such adaption fully uses the available memory and increases the input data reuse, reducing the runtime overheads including the memory fragmentation and data movement. As shown in Figure 3-4, the model-adaptive im2col operation improved inference efficiency by 13%.

3.2.3 Computation kernel specialization.

TinyEngine *specializes* the kernel optimizations for different layers: loops tiling is based on the kernel size and available memory, which is different for each layer; and the inner loop unrolling is also specialized for different kernel sizes (*e.g.*, 9 repeated code segments for 3×3 kernel, and 25 for 5×5) to eliminate the branch instruction overheads. Operation fusion is performed for Conv+Padding+ReLU+BN layers. These specialized optimization on the computation kernel further increased the inference efficiency by 22%, as shown in Figure 3-4.

3.2.4 In-place depth-wise convolution

We propose *in-place* depth-wise convolution to further reduce peak memory. Different from standard convolutions, depth-wise convolutions do not perform filtering across channels. Therefore, once the computation of a channel is completed, the input activation of the channel can be overwritten and used to store the output activation of another channel,

allowing activation of depth-wise convolutions to be updated in-place as shown in Figure 3-6. This method reduces the measured memory usage by 1.6$\times$ as shown in Figure 3-4.

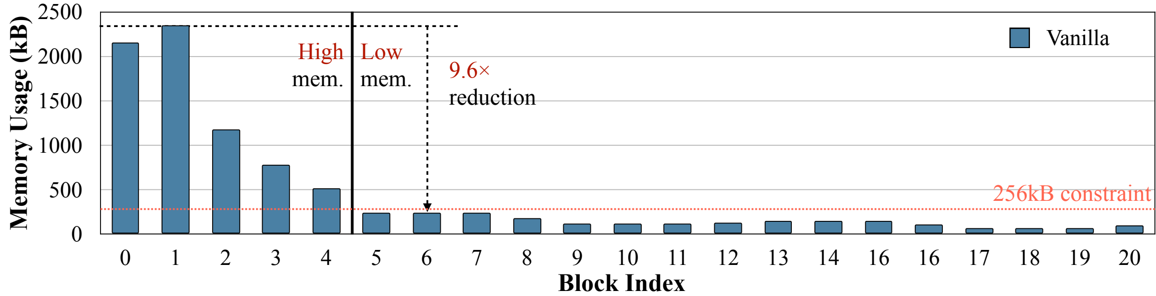
Chapter 4

MCUNetV2: Memory-Efficient Patched-based Inference

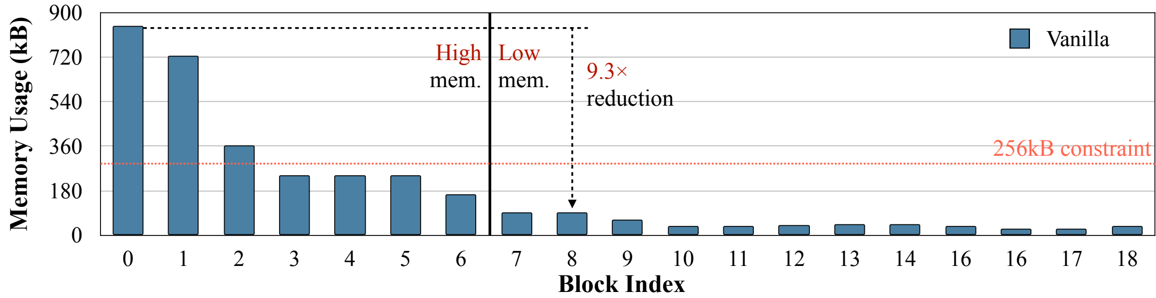
4.1 Rethinking the Memory Bottleneck of TinyML

4.1.1 Imbalanced Memory Usage Distribution

Tiny deep learning on MCUs is challenging due to the limited memory size. Even efficient deep network designs like MobileNetV2 [49]) exceed the memory limit of a popular MCU model (STM32F412 with 256kB SRAM and 1MB Flash) by $9\times$, making the deployment difficult. However, we find that the memory bottleneck is largely due to the *imbalanced peak memory distribution* in general CNN backbone designs. We analyze the per-block peak memory usage of MobileNetV2 [49] in Figure 1-2. We can see that the memory usage is highly imbalanced: the first 5 blocks consume much larger memory compared to the rest of the network. In fact, the second block has $11\times$ larger memory usage than the rest of the network. To execute the model on an MCU, the *peak* occupied memory usage during inference must be lower than the SRAM size; thus, the second block becomes the *memory bottleneck* of the whole network. We also provide the memory distribution of a NAS model ProxylessNAS-Mobile [6] and MCUNet model [35] in Figure 4-1. Both of them suffer from a similar imbalanced distribution. Notice that though MCUNet reduces the overall peak memory, the memory usage distribution is still imbalanced due to the CNN backbone shape.



(a) Memory usage of ProxylessNAS Mobile



(b) Memory usage of MCUNet

Figure 4-1. Apart from MobileNetV2 [49] (Figure 1-2), ProxylessNAS-Mobile [6] and MCUNet [35] also suffer from a highly imbalanced memory usage distribution. Note that the memory profiling is done with fp32. The MCUNet model was designed to fit the memory constraint when quantized in int8.

Actually, this situation applies to almost all single-branch CNN designs (like ResNet [23] or MobileNet [49]) due to the hierarchical structure*: after each stage, the image resolution is down-sampled by half, leading to $4\times$ fewer pixels, while the channel number increases only by $2\times$ [50, 23, 28] or by an even smaller ratio [49, 27, 52], resulting in a decreasing activation size. Therefore, the memory bottleneck tends to appear at the very early stage of the network, after which the peak memory usage is much smaller. Any sequential CNN network with the above hierarchical backbone design will suffer from a similar imbalanced memory distribution.

The imbalanced memory distribution significantly limits the model capacity runnable under limited memory: in order to accommodate the initial memory-intensive stage, the whole network needs to be scaled down (resolution or channel) by a large ratio even though the majority of the network actually has a small memory usage. It also makes resolution-sensitive tasks (*e.g.*, object detection) difficult: as shown in Figure 4-2, object detection

*some CNN designs have highly complicated branching structure (*e.g.*, NASNet [60]), but they are generally less efficient for inference; thus not widely used for edge computing.

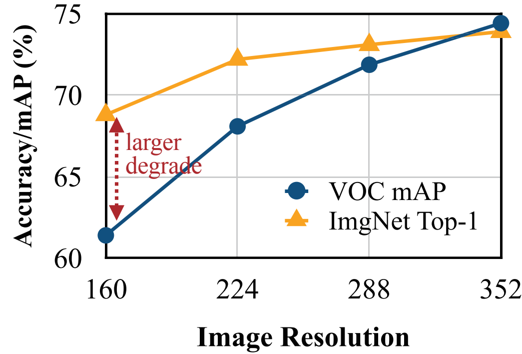


Figure 4-2. Object detection (PASCAL VOC [16]) suffers a larger degradation from reduced resolutions than image classification (ImageNet [14]).

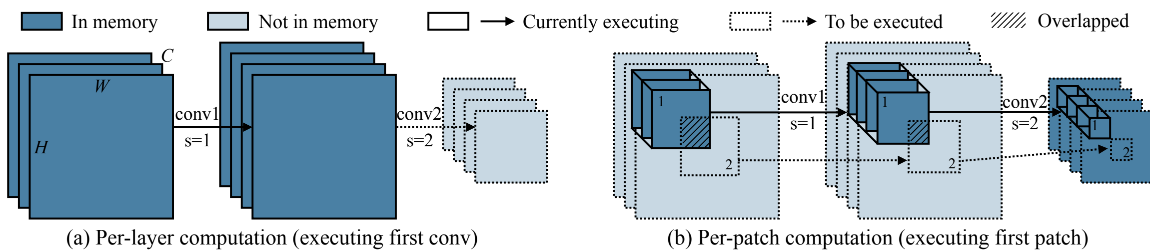


Figure 4-3. Per-patch computation can significantly reduce the peak memory required to execute a sequence of convolutional layers. Here we study two convolutional layers (with stride 1 and 2). Under per-layer computation (a), the first convolution has large input/output activation size, dominating the peak memory requirement. With per-patch computation (b), we allocate the buffer to host the final output activation, and compute the results *patch-by-patch*. We only need to store the activation from *one patch* but not the entire feature map, reducing the peak memory usage significantly (the first input is the input image, which can be partially decoded from a compressed format like JPEG) .

suffers a larger degradation from reduced resolutions compared to image classification. Suppose the first convolutional layer has 3 input channels and 16 output channels, the largest allowable resolution for this layer is 117×117 under 256kB SRAM (with int8 quantization, $\frac{3\times117^2+16\times117^2}{1024} = 254$kB), which may not be enough to capture smaller objects.

4.1.2 Reducing Memory with Per-patch Computation

We argue that such a dilemma is due to the *per-layer* execution commonly adopted in deep learning inference frameworks like TensorFlow Lite Micro [3], CMSIS-NN [31], TinyEngine [35], microTVM [9], *etc*. For each convolutional layers, the inference library first allocates the input and output activation buffer in SRAM and releases the input buffer after the *whole* layer computation is finished (if the input is not used by other layers). Such an implementation makes inference optimization easy (*e.g.*, im2col, loop tiling, loop

unrolling, *etc.*), but the SRAM has to hold the entire input and output activation for each layer, which is prohibitively large for the initial stage of the network.

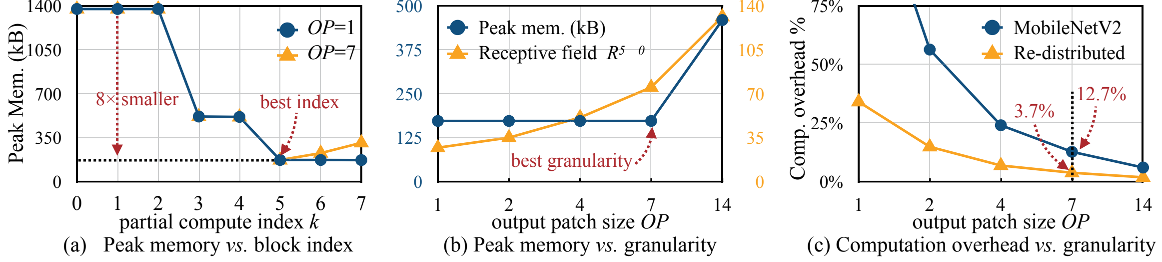
As shown in Figure 1-2, although the first several blocks have large input and output activation, the output activation of the fifth block is small. If we can find a way to "bypass" the initial memory-heavy stage, we can greatly reduce the memory required for inference. A possible solution is to reorder the computation of convolutions and only compute a subset of output at a time (also for all intermediate activations). The size of activation in CNNs is determined by the spatial and channel dimension size ($C \times HW$). A normal convolutional layer (with group size 1) has fully connected channel dependencies, which means the output is dependent on all input channels, making it impossible to compute only a subset of channels. Therefore, we choose to perform *per-patch* computation to reduce the memory usage.

As shown in Figure 4-3, we show an example of two convolutional layers (with stride 1 and 2). For conventional per-layer computation, the first convolutional layer has high input and output memory usage, leading to a large peak memory usage. With per-patch computation, we first allocate the buffer for the final output and compute its values *patch-by-patch*. In this manner, we only need to store the activation from *one patch* instead of the *whole feature map*, which greatly reduces the peak memory requirement. Note that the first input activation is the input image, which can be partially decoded from a compressed format like JPEG, so we do not need to fully store it.

4.1.3 Memory Saving & Computation Overhead

Per-patch computation can greatly reduce the peak memory, but it also introduces computation overhead due to *spatial overlapping*. Due to the increased receptive field, the non-overlapping patches in output activation have overlapping correspondence in input activation (shadowed area in Figure 4-3(b)), leading to repeated computation. We analyze the memory saving and computation overhead of the MobileNetV2 [49] backbone.

Notations. For a CNN with n convolutional layers, we denote each layer as $l^i, i \in \{0, 1, ..., n-1\}$, whose kernel size is k^i, and stride is s^i (assume symmetry). We assume the pooling is done with convolutions with stride=2 (as in MobileNetV2). The input



(a) Peak memory *vs.* block index (b) Peak memory *vs.* granularity (c) Computation overhead *vs.* granularity

Figure 4-4. Ablation study on spatial partial computation. (a) The peak memory generally goes down with more blocks being partially computed, and then goes up due to the large patch size. The optimal index is chosen when the output feature map is spatially down-sampled by $8\times$. (b) The receptive field $R^{5\to0}$ (also the patch size) grows with the output patch size (OP). For $OP \leq 7$, the peak memory is bounded by the rest of the network; for $OP \geq 14$, the patch is too large, causing peak memory increase. We choose $OP = 7$ to achieve reduced peak memory and minimum computation overhead.

activation of layer i is denoted as $\mathbf{x}^i$, and $\mathbf{x}^{i+1} = l^i(\mathbf{x}^i)$. For a pixel in $\mathbf{x}^i$, we denote its receptive field on an earlier activation $\mathbf{x}^j, j < i$ as $R^{i\to j}$, which can be computed iteratively:

$$R^{i\to i} = 1, \quad R^{i\to m-1} = (R^{i\to m} - 1) * s^{m-1} + k^{m-1}. \tag{4.1}$$

Similarly, we can calculate the receptive field of a coarser region by setting $R^{i\to i} = 2/4/etc$. $R^{i\to i}$ is the output patch size OP, $R^{i\to 0}$ is the input patch size IP extracted from the input image.

Suppose we perform per-patch computation for the first k blocks to reduce the peak memory, and compute the rest of the network (blocks $> k$) with normal per-layer computation. We suppose the output of block k has resolution $H_k \times W_k$. Without loss of generality, we assume $H_k = W_k$. For each time, we can partially compute a small spatial region with resolution $h \times w, h = w = OP$. [†].

Memory saving. We show the reduced peak memory *vs.* per-patch computed block index k in Figure 4-4(a). We followed [35] to compute the memory usage in int8 precision to better represent the actual edge deployment scenario. When performing per-patch computation, we need to hold the final output activation and the activation for each patch (Figure 4-3(b)). With a larger k, the former generally becomes smaller, but the latter becomes larger since

[†] We suppose the partial computation is performed on a square patch each time. We can also compute a partial output in other orders like row-by-row, which also follows a similar analysis.

the receptive field grows with k, making input patch size IP larger (for the same OP). Therefore, the peak memory first decreases and then increases. $k = 5$ is the best index, after which the feature map is down-sampled by $8\times$ (from 224^2 to 28^2). At this point, the peak memory can be reduced by $8\times$ compared to per-layer execution. For the following analysis, we use $k = 5$.

Given a fixed k, the peak memory is also related to the output patch size OP. As shown in Figure 4-4(b), the input patch size IP, *i.e.*, receptive field $R^{5\to0}$ increases with OP. For $OP \leq 7$, the patch size is small enough, and the peak memory is dominated by the rest of the network. But for $OP \geq 14$, the patch is too large, increasing the peak memory usage.

Computation overhead. Per-patch inference could bring computation overhead from spatial overlapping. Due to the increased receptive field from convolutions, the input patches need to be *overlapped* to produce the corresponding *non-overlapping* output patches (see the shadowed area in Figure 4-3(b)). The larger the receptive field, the larger the overlapping area and repeated computation. Given a fixed k, the computation overhead decreases as OP increases (Figure 4-4). For $OP = 1$, the computation overhead is 145% (738M *vs*. 300M), which is prohibitively large. For $OP = 7$, we can achieve the best memory reduction and at an overhead of 13% (339M *vs*. 300M), which is much smaller but still a significant increase for edge devices. We will discuss how to further reduce the overhead in the next section.

4.2 Reducing Computation Overhead by Redistributing Receptive Field

As discussed above, to reduce the repeated computation, we need to reduce the input patch size IP at the same output patch size OP, requiring to reduce the receptive field of the initial stage. The receptive field comes from two components: (1) feature map down-sampling with convolutions of stride=2; (2) convolutions with kernel size>1. The former determines the backbone shape of the CNN and contributes to the shrinking feature map size; thus cannot be changed. Therefore, we focus on changing the convolutional kernels to reduce the receptive field of the patch-based stage.

We propose to *redistribute* the receptive field of the CNN to reduce computation overhead.

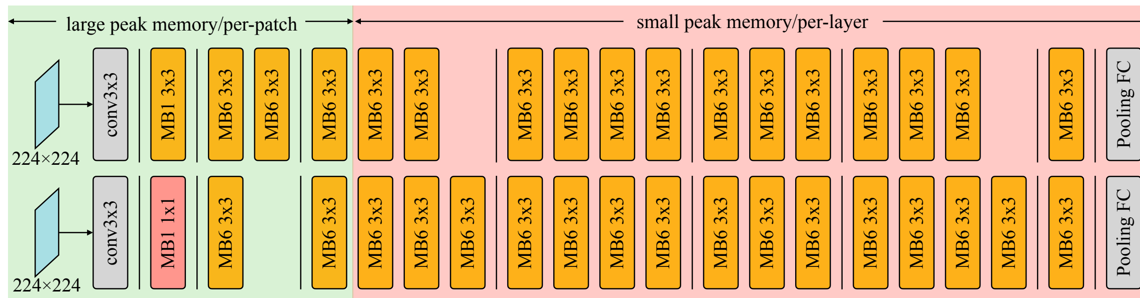


Figure 4-5. The original MobileNetV2 [49] (upper) and the one with redistributed receptive field (lower). We reduce the receptive field in the initial stage with large peak memory (left), leading to smaller repeated computation with per-patch inference. We then increase the receptive field in the rest of the network to maintain the overall performance.

The basic idea is to *reduce the receptive field of the patch-based initial stage and increase the receptive field of the later stage*, which maintains the performance of the network (some tasks require a large receptive field of CNN to get decent performance). We follow the below design guideline:

- We do not change the down-sampling design of the network to keep the original backbone shape (*i.e.*, different stages design).
- For convolutional layers or blocks with stride=2, we keep kernel size=3; otherwise, 3/4 of the information will be lost.
- We remove other convolutional layers and blocks in the per-patch inference stage and increase the convolutions in the later stage to keep the overall computation and model size.

We take MobileNetV2 as a baseline network and re-design its initial stage. The architecture comparison is shown in Figure 4-5. As shown in Figure 4-4(c), the computation overhead drops from 12.7% to 3.7% after redistribution, which is negligible considering the benefits in peak memory reduction.

4.3 Joint Search Space Scaling and Neural Architecture Search

Redistributing receptive field can reduce the computation overhead, but the design varies case-by-case for different networks: we need to carefully select which of the stages to

increase receptive field so that the overall computation remains the same and the performance does not degrade. To automate the process, we used neural architecture search to automatically find a network with minimal computation overhead and best accuracy under patch-based inference.

A proper search space design is fundamental to the final performance of neural architecture search, especially for tight hardware constraints [35]: an improper search space may not contain suitable models at all. MCUNet [35] automatically optimizes the search space design by analyzing the FLOPs distribution under specific hardware constraints, where a space of higher computation models is preferred. However, we argue that the search space for NAS should be not only *device-aware* but also *task-aware*. For example, some tasks may need higher input resolutions than others, like object detection *vs*. image classification. When scaling the search space, we should also consider task performance as an important metric.

We propose to treat the search space scaling itself as part of the neural architecture search process. Existing MobileNetV2-space [6, 35, 51] or MobileNetV3-space [27, 5] search spaces consist of different micro structures like kernel sizes k (3/5/7), expansion ratios e (3/4/6), depth per stage d (2/3/4). We extend them to also support search space-wise macro structures like different input resolutions r and global width multipliers w. Supporting input different input resolutions and width multipliers can help scale the overall computation and hardware usage of the sub-network to fit hardware of diverse resources, which is extremely helpful on microcontrollers with limited memory (Section 5.4). In later experiments, we will show that supporting flexible per-block w can further improve the network performance compared to a global w.

Chapter 5

Results

5.1 Setups

Datasets. We used 3 datasets as benchmark: ImageNet [14], Visual Wake Words (VWW) [12], and Speech Commands (V2) [54]. ImageNet is a standard large-scale benchmark for image classification. VWW and Speech Commands represent popular microcontroller use-cases: VWW is a vision based dataset identifying whether a person is present in the image or not; Speech Commands is an audio dataset for keyword spotting (*e.g.*, "Hey Siri"), requiring to classify a spoken word from a vocabulary of size 35. Both datasets reflect the always-on characteristic of microcontroller workload. We did not use datasets like CIFAR [30] since it is a small dataset with a limited image resolution (32×32), which cannot accurately represent the benchmark model size or accuracy in real-life cases.

During neural architecture search, in order not to touch the validation set, we perform validation on a small subset of the training set (we split 10,000 samples from the training set of ImageNet, and 5,000 from VWW). Speech Commands has a separate validation&test set, so we use the validation set for search and use the test set to report accuracy. The training details are in the supplementary material.

Model deployment. We perform int8 linear quantization to deploy the model. We deploy the models on microcontrollers of diverse hardware resource, including STM32F412 (Cortex-M4, 256kB SRAM/1MB Flash), STM32F746 (Cortex-M7, 320kB/1MB Flash), STM32F765 (Cortex-M7, 512kB SRAM/1MB Flash), and STM32H743 (Cortex-M7, 512kB SRAM/2MB

Library Model	S-MbV2	S-Proxyless	TinyNAS
CMSIS-NN [31]	35.2%	49.5%	55.5%
TinyEngine	47.4%	56.4%	**61.8%**

Table 5.1. System-algorithm co-design (TinyEngine + TinyNAS) achieves the highest ImageNet accuracy of models runnable on a microcontroller.

Latency Constraint	**N/A**	**5FPS**	**10FPS**
S-MbV2+CMSIS	39.7%	39.7%	28.7%
S-MbV2+TinyEngine	47.4%	41.6%	34.1%
MCUNet	**61.8%**	**49.9%**	**40.5%**

Table 5.2. MCUNet outperforms the baselines at various latency requirements. Both TinyEngine and TinyNAS bring significant improvement on ImageNet.

Flash). By default, we use STM32F746 to report the results unless otherwise specified. All the latency is normalized to STM32F746 with 216MHz CPU.

5.2 Large-Scale Image Recognition on Tiny Devices

With our system-algorithm co-design, we achieve record high accuracy (70.7%) on large-scale ImageNet recognition on microcontrollers. We co-optimize TinyNAS and TinyEngine to find the best runnable network. We compare our results to several baselines. We generate the *best scaling* of MobileNetV2 [49] (denoted as **S-MbV2**) and ProxylessNAS Mobile [6] (denoted as **S-Proxyless**) by compound scaling down the width multiplier and the input resolution until they meet the memory requirement. We train and evaluate the performance of all the satisfying scaled-down models on the Pareto front [*], and then report the highest accuracy as the baseline. The former is an efficient manually designed model, the latter is a state-of-the-art NAS model. We did not use MobileNetV3 [27]-alike models because the hard-swish activation is not efficiently supported on microcontrollers.

Co-design brings better performance. Both the inference library and the model design help to fit the resource constraints of microcontrollers. As shown in Table 5.1, when running on a tight budget of 320kB SRAM and 1MB Flash, the optimal scaling of MobileNetV2 and

[*] *e.g.*, if we have two models (w0.5, r128) and (w0.5, r144) meeting the constraints, we only train and evaluate (w0.5, r144) since it is strictly better than the other; if we have two models (w0.5, r128) and (w0.4, r144) that fits the requirement, we train both networks and report the higher accuracy.

	Quantization	STM32F412 (256kB, 1MB)	STM32F746 (320kB, 1MB)	STM32F765 (512kB, 1MB)	STM32H743 (512kB, 2MB)
Rusci *et al.* [48]	Mixed	60.2%	-	62.9%	68.0%
MCUNet	4-bit	**62.0%**	**63.5%**	**65.9%**	**70.7%**

Table 5.3. MCUNet can handle diverse hardware resource on different MCUs. It outperforms [48] without using advanced mixed-bit quantization (8/4/2-bit) policy under different resource constraints, achieving a record ImageNet accuracy (>70%) on microcontrollers.

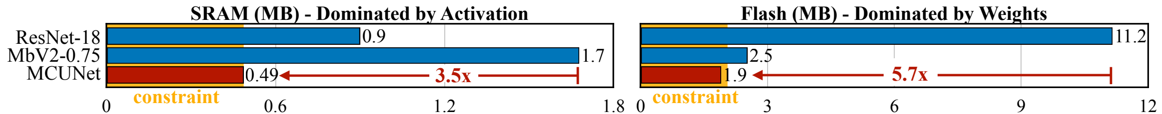


Figure 5-1. MCUNet reduces the the SRAM memory by **3.5**× and Flash usage by **5.7**× compared to MobileNetV2 and ResNet-18 (8-bit), while achieving better accuracy (70.7% *vs*. 69.8% ImageNet top-1).

ProxylessNAS models only achieve 35.2% and 49.5% top-1 accuracy on ImageNe using CMSIS-NN [31]. With TinyEngine, we can fit larger models that achieve higher accuracy of 47.4% and 56.4%; with TinyNAS, we can specialize a more accurate model under the tight memory constraints to achieve 55.5% top-1 accuracy. Finally, with system-algorithm co-design, MCUNet further advances the accuracy to 61.8%, showing the advantage of joint optimization.

Co-design improves the performance at various latency constraints (Table 5.2). TinyEngine accelerates inference to achieve higher accuracy at the same latency constraints. For the optimal scaling of MobileNetV2, TinyEngine improves the accuracy by 1.9% at 5 FPS setting and 5.4% at 10 FPS. With MCUNet co-design, we can further improve the performance by 8.3% and 6.4%.

Diverse hardware constraints & lower bit precision. We used int8 linear quantization for both weights and activations, as it is the industrial standard for faster inference and usually has negligible accuracy loss without fine-tuning. We also performed 4-bit linear quantization on ImageNet, which can fit larger number parameters. The results are shown in Table 5.3. MCUNet can handle diverse hardware resources on different MCUs with Cortex-M4 (F412) and M7 (F746, F765, H743) core. Without mixed-precision, we can already outperform the existing state-of-the-art [48] on microcontrollers, showing the effectiveness

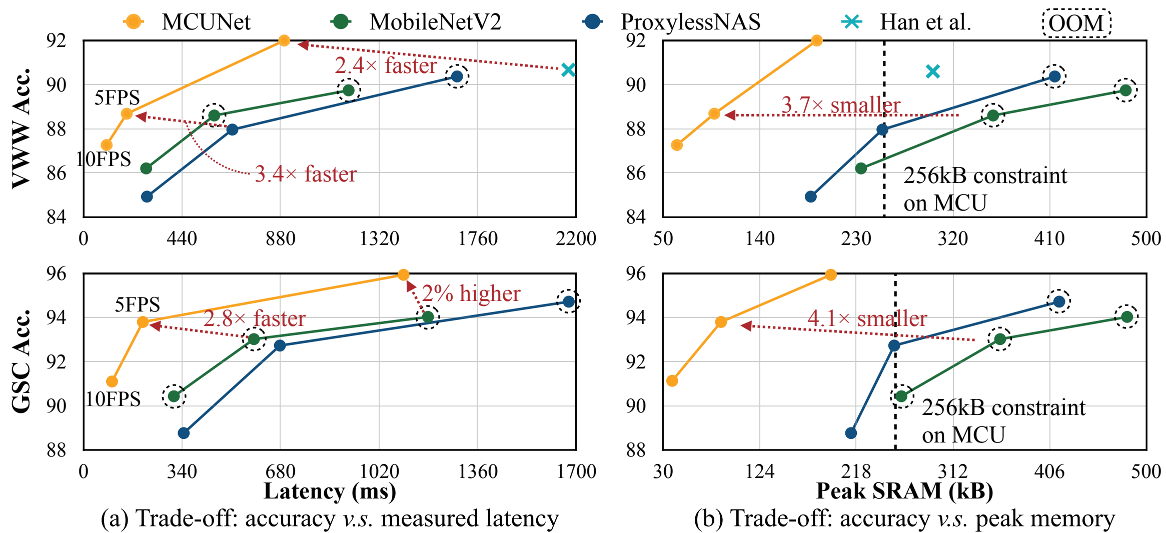


Figure 5-2. Accuracy *vs.* latency/SRAM memory trade-off on VWW (**top**) and Speech Commands (**down**) dataset. MCUNet achieves better accuracy while being 2.4-3.4× faster at 3.7-4.1× smaller peak SRAM.

of system-algorithm co-design. We believe that we can further advance the Pareto curve in the future with mixed precision quantization.

Notably, our model achieves a record ImageNet top-1 accuracy of 70.7% on STM32H743 MCU. To the best of our knowledge, we are the first to achieve $> 70\%$ ImageNet accuracy on off-the-shelf commercial microcontrollers. Compared to ResNet-18 and MobileNetV2-0.75 (both in 8-bit) which achieve a similar ImageNet accuracy (69.8%), our MCUNet reduces the the memory usage by 3.5× and the Flash usage by 5.7× (Figure 5-1) to fit the tiny memory size on microcontrollers.

5.3 Visual&Audio Wake Words

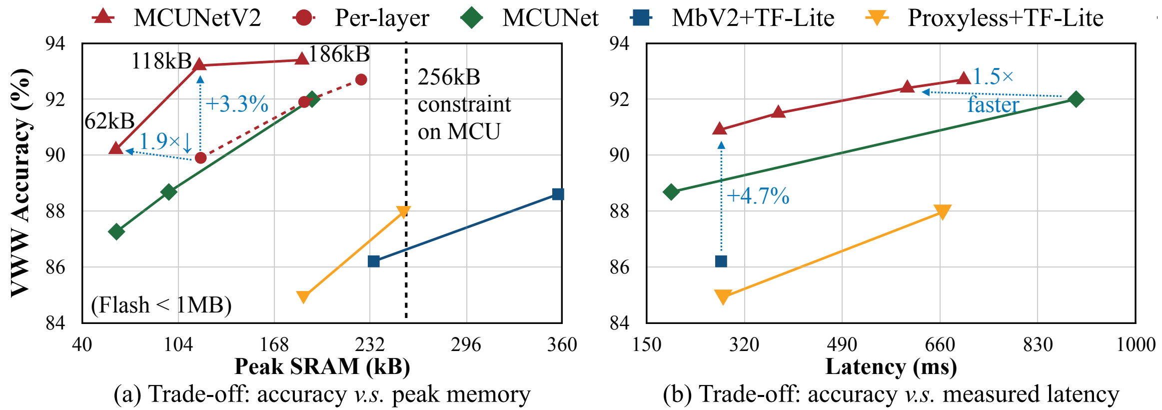
We benchmarked the performance on two wake words datasets: Visual Wake Words [12] (VWW) and Google Speech Commands (denoted as GSC) to compare the accuracy-latency and accuracy-peak memory trade-off. We compared to the optimally scaled MobileNetV2 and ProxylessNAS running on TF-Lite Micro. The results are shown in Figure 5-2. MCUNet significantly advances the Pareto curve. On VWW dataset, we can achieve higher accuracy at 2.4-3.4× faster inference speed and 3.7× smaller peak memory. We also compare our results to the previous first-place solution on VWW challenge [1] (denoted as Han *et al.*). We scaled the input resolution to tightly fit the memory constraints of 320kB and re-trained

Model / Library	Quantize	SRAM	Flash	Top-1	Top-5
STM32F412 (256kB SRAM, 1MB Flash)					
MbV1 0.5× (r=192) [28] / Rusci *et al*. [48]	mixed	<256kB	<1MB	60.2%	
MCUNet (TinyNAS / TinyEngine) [35]	int8	238kB	1007kB	60.3%	-
MCUNet (TinyNAS / TinyEngine) [35]	int4	233kB	1008kB	62.0%	-
MCUNetV2	int8	228kB	1010kB	**63.4%**	**85.4%**
STM32F746 (320kB SRAM, 1MB Flash)					
MbV2 0.35× (r=144) [49] / TinyEngine [35]	int8	308kB	862kB	49.0%	73.8%
Proxyless 0.25× (r=112) [6] / TF-Lite [3]	int8	288kB	860kB	43.8%	69.0%
Proxyless 0.3× (r=176) [6] / TinyEngine [35]	int8	292kB	892kB	56.2%	79.7%
MCUNet (TinyNAS / TinyEngine) [35]	int8	293kB	897kB	61.8%	84.2%
MCUNet (TinyNAS / TinyEngine) [35]	int4	282kB	1010kB	63.5%	-
MCUNetV2	int8	297kB	1010kB	**64.9%**	**86.2%**
STM32H743 (512kB SRAM, 2MB Flash)					
MbV1 0.75× (r=224) [28] / Rusci *et al*. [48]	mixed	<512kB	<2MB	68.0%	
MCUNet (TinyNAS / TinyEngine) [35]	int8	452kB	2014kB	68.5%	-
MCUNet (TinyNAS / TinyEngine) [35]	int4	498kB	2000kB	70.7%	-
MCUNetV2	int8	465kB	2032kB	**71.8%**	**90.7%**

Table 5.4. MCUNetV2 with patch-based inference further improves the accuracy of ImageNet classification under various MCU hardware constraints, leading to 3% better accuracy under the same quantization scheme.

it under the same setting like ours. We find that MCUNet achieves 2.4× faster inference speed compared to the previous state-of-the-art. Interestingly, the model from [1] has a much smaller peak memory usage compared to the biggest MobileNetV2 and ProxylessNAS model, while having a higher computation and latency. It also shows that a smaller peak memory is the key to success on microcontrollers.

On the Speech Commands dataset, MCUNet achieves a higher accuracy at 2.8× faster inference speed and 4.1× smaller peak memory. It achieves 2% higher accuracy compared to the largest MobileNetV2, and 3.3% improvement compared to the largest runnable ProxylessNAS under 256kB SRAM constraint.



(a) Trade-off: accuracy *v.s.* peak memory (b) Trade-off: accuracy *v.s.* measured latency

Figure 5-3. (a) With patch-based inference ("MCUNetV2"), we are can greatly improve the accuracy *vs*. peak SRAM trade-off compared to per-layer execution ("per-layer") and existing state-of-the-art methods. Under the same peak SRAM, MCUNetV2 improves the accuracy by 3.3%. It can also achieve >90% accuracy under 64kB memory, 1.9× smaller than per-layer execution, paving the way for even smaller hardware resource. (b) MCUNetV2 also has a better accuracy *vs*. latency trade-off thanks to the broadened search space. Compared to MCUNet, MCUNetV2 can obtain higher accuracy at 1.5× faster inference; compared to MobileNetV2+TF-Lite Micro, MCUNetV2 improves the accuracy by 4.7% at the same latency.

5.4 Reduce Memory Usage with Patch-based Inference

With MCUNetV2, we can further improve the accuracy and reduce the peak memory usage by using patch-based inference. Here we show the results of MCUNetV2 running on microcontrollers with TinyEngine.

Improving ImageNet classification. We show the results of ImageNet classification under different hardware constraints in Table 5.4. Compared to MCUNet with the same quantization scheme (int8), MCUNetV2 greatly improves the accuracy by 3%, thanks to the patch-based inference scheme. Specifically, we are able to get better accuracy on STM32F412 than previous work on STM32F746. Note that quantization with a lower bit-precision (*e.g.*, 4) or mixed precisions can further improve the accuracy-peak memory trade-off (marked in gray), but the instruction set on MCU usually does not support low-bit acceleration, leading to a lower inference speed due to higher MACs. Therefore, we only show the results of int8 quantization and left other settings to future work.

Visual wake words under 64kB SRAM. With patch-based inference, we can greatly improve the accuracy *vs*. peak SRAM trade-off compared to per-layer execution. As shown

	resolution	FLOPs	#Param	peak SRAM	mAP
MbV2+CMSIS	128	34M	0.87M	519kB (OOM)	31.6%
MCUNet	224	168M	1.20M	466kB	**51.4%**

Table 5.5. MCUNet improves the detection mAP by 20% on Pascal VOC under 512kB SRAM constraint. With MCUNet, we are able to fit a model with much larger capacity and computation FLOPs at a smaller peak memory. MobileNet-v2 + CMSIS-NN is bounded by the memory consumption: it can only fit a model with 34M FLOPs even when the peak memory slightly exceeds the budget, leading to inferior detection performance.

in Figure 5-3(a), MCUNetV2 can improve the accuracy by 3.3% under the same peak memory compared to per-layer execution. It also achieves >90% accuracy using only 62kB SRAM memory, which is 1.9× smaller compared to per-layer inference and enables tiny-scale deep learning at an even smaller hardware budget.

Patch-based inference allows a broadened search space, which also gives better accuracy-latency trade-off. As shown in Figure 5-3(b), MCUNetV2 can obtain higher accuracy at 1.5× faster inference compared to MCUNet. Compared to MobileNetV2+TF-Lite Micro, we can improve the accuracy by 4.7% at the same latency.

5.5 Object Detection on MCUs

To show the generalization ability of MCUNet framework across different tasks, we apply MCUNet to object detection. Object detection is particularly challenging for memory-limited MCUs: a high-resolution input is usually required to detect the relatively small objects, which will increase the peak performance significantly. We benchmark the object detection performance of our MCUNet and scaled MobileNetV2+CMSIS-NN on on Pascal VOC [16] dataset. We used YOLOv2 [46] as the detector; other more advanced detectors like YOLOv3 [47] use multi-scale feature maps to generate the final prediction, which has to keep intermediate activations in the SRAM, increasing the peak memory by a large margin. The results on H743 are shown in Table 5.5. Under tight memory budget (only 512kB SRAM and 2MB Flash), MCUNet significantly improves the mAP by 20%, which makes AIoT applications more accessible.

	R-18@224	Rand Space	Huge Space	Our Space
Acc.	80.3%	74.7±1.9%	77.0%	**78.7%**

Table 5.6. Our search space achieves the best accuracy, closer to ResNet-18@224 resolution (OOM). Randomly sampled and a huge space (contain many configs) leads to worse accuracy.

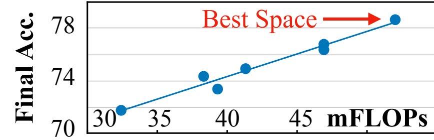
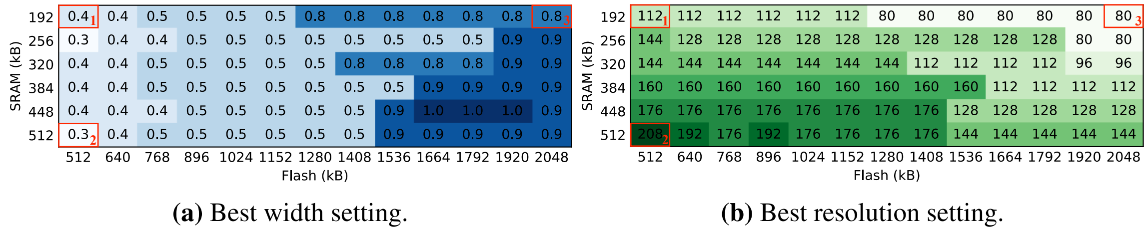


Figure 5-4. Search space with higher mean FLOPs leads to higher final accuracy.



(a) Best width setting.

(b) Best resolution setting.

Figure 5-5. Best search space configurations under different SRAM and Flash constraints.

5.6 Analysis

Search space optimization matters. Search space optimization significantly improves the NAS accuracy. We performed an ablation study on ImageNet-100, a subset of ImageNet with 100 randomly sampled categories. The distribution of the top-10 search spaces is shown in Figure 3-2. We sample several search spaces from the top-10 search spaces and perform the whole neural architecture search process to find the best model inside the space that can fit 320kB SRAM/1MB Flash.

We compare the accuracy of the searched model using different search spaces in Table 5.6. Using the search space configuration found by our algorithm, we can achieve 78.7% top-1 accuracy, closer to ResNet-18 on 224 resolution input (which runs out of memory). We evaluate several randomly sampled search spaces from the top-10 spaces; they perform significantly worse. Another baseline is to use a very large search space supporting variable resolution (96-176) and variable width multipliers (0.3-0.7). Note that this "huge space" contains the best space. However, it fails to get good performance. We hypothesize that using a super large space increases the difficulty of training super network and evolution search. We plot the relationship between the accuracy of the final searched model and the mean FLOPs of the search space configuration in Figure 5-4. We can see a clear positive relationship, which backs our algorithm.

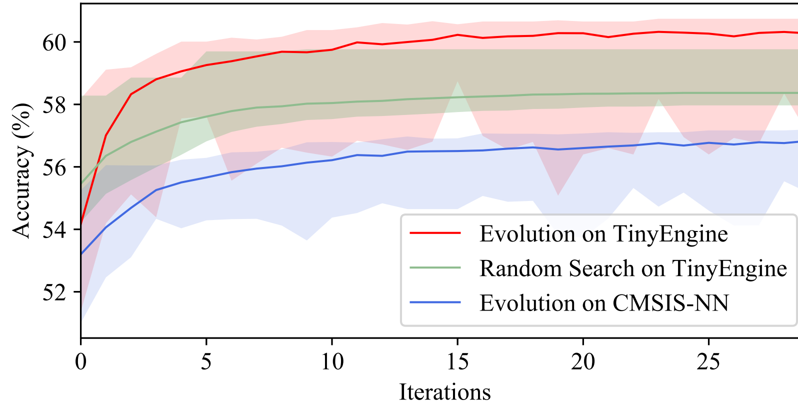


Figure 5-6. Evolutionary search can find specialized networks with better accuracy compared to random search. Evolving on TinyEngine provides a larger search space, enabling us to find better models compared to CMSIS-NN.

Sensitivity analysis on search space optimization. We inspect the results of search space optimization and find some interesting patterns. The results are shown in Figure 5-5. We vary the SRAM limit from 192kB to 512kB and Flash limit from 512kB to 2MB, and show the chosen width multiplier and resolution. Generally, with a larger SRAM to store a larger activation map, we can use a higher input resolution; with a larger Flash to store a larger model. we can use a larger width multiplier. When we increase the SRAM and keep the Flash from point 1 to point 2 (red rectangles), the width is not increased as Flash is small; the resolution increases as the larger SRAM can host a larger activation. From point 1 to 3, the width increases, and the resolution actually decreases. This is because a larger Flash hosts a wider model, but we need to scale down the resolution to fit the small SRAM. Such kind of patterns is non-trivial and hard to discover manually.

Evolutionary search. The curve of evolutionary search on different inference library is in Figure 5-6. The solid line represents the average value, while the shadow shows the range of (min, max) accuracy. On TinyEngine, evolution clearly outperforms random search, with 1% higher best accuracy. The evolution on CMSIS-NN leads to much worse results due to memory inefficiency: the library can only host a smaller model compared to TinyEngine, which leads to lower accuracy.

Receptive field redistribution *vs*. input splitting. Apart from redistributing the receptive field, here, we also consider a baseline by splitting input images into non-overlapping

Model	Inference	Invariant	Patch RF	MACs	Peak SRAM	ImgNet Top-1	VOC mAP
MbV2 [49]	Per-layer	✓	-	300M	1372kB	72.2%	75.4%
	Per-patch (OP=7)	✓	75^2	339M	172kB		
Input split	Per-patch (OP=7)	✗	56^2	300M	172kB	71.8%	73.9%
Re-distribute	Per-patch (OP=7)	✓	63^2	309M	172kB	72.1%	75.7%

Table 5.7. Per-patch inference can reduce the peak memory by 8× for MobileNetV2 [49] (1372kB to 172kB), but it increases the computation by 13% (300M to 339M) due to a large patch receptive field (RF). Splitting input image into non-overlapping patches (Input split) can retain the computation, but it breaks the translation-invariance of CNN and leads to a large degradation in object detection. Re-distributing receptive field (Re-distribute) can reduce receptive field and computation overhead (only 3.7%) while getting same performance compared to the original network. We denote degraded items with red color.

patches and directly run the per-patch stage on each patch as done in [15]. Such a practice does not incur extra computation (since the patches are not overlapped), but it breaks the feature propagation between patches and also breaks the translation-invariant property of CNNs. The comparison is provided in Table 5.7. We can find that per-patch inference reduces the peak SRAM by 8× for all methods, but the original MobileNetV2 design has 13% computation overhead, which is undesired for edge devices. Splitting the input image into non-overlapping patches ("Input split") can retain the computation, but it breaks the translation-invariance of CNN and leads to performance degradation, especially for object detection on Pascal VOC [16] dataset. A similar phenomenon is observed in [38], showing cross-patch communication is essential for dense prediction tasks. Redistributing the receptive field ("Redistribute") can reduce the input patch size from 75 to 63 while maintaining similar performance.

Joint search space scaling and NAS improves tinyML. Existing efficient neural architecture search methods for edge deployment mostly focus on mobile settings. To fit tiny hardware, the searched models are usually scaled down to meet a smaller computation budget (*e.g.*, <50M MACs), as the original search space does not contain small enough models. However, such NAS+scaling procedure is sub-optimal: the searched model may not be optimal after scaling down. Our extended search space with joint search scaling can directly specialize models under very small computational budgets and provide decent performance. Here we perform experiments based on the MobileNetV3 [27] search space

Budget	Model	Setting	MACs	Top-1	Top-5
100M MACs	MobileNetV1 0.5× (r=192) [28]	Manual+Scale	110M	61.7%	83.6%
	MobileNetV2 0.75×(r=160) [49]	Manual+Scale	107M	66.4%	87.3%
	MobileNetV3 Small 1.25× [27]	NAS+Scale	91M	70.4%	-
	EfficientNet-B^{-2} [52, 20]	NAS+Scale	98M	70.5%	89.5%
	TinyNet-C [20] *	NAS+Scale	103M	71.2%	89.7%
	Ours (global w)	Joint Search	98M	**72.3%**	**90.6%**
	Ours (flexible w)	Joint Search	99M	**72.3%**	90.5%
50M MACs	MobileNetV2 0.35× [49]	Manual+Scale	59M	60.3%	82.9%
	MnasNet-A1 0.35× [51]	NAS+Scale	63M	64.1%	85.1%
	MnasNet-search1 [51]	NAS	65M	64.9%	-
	EfficientNet-B^{-3} [52, 20]	NAS+Scale	51M	65.0%	85.2%
	TinyNet-D [20] *	NAS+Scale	53M	67.0%	87.1%
	MobileNetV3 Small 1.0× [27]	NAS	56M	67.4%	-
	Ours (global w)	Joint Search	50M	67.9%	87.7%
	Ours (flexible w)	Joint Search	50M	**68.8%**	**88.2%**
25M MACs	MobileNetV2 0.35× (r=160) [49]	Manual+Scale	30M	55.7%	79.1%
	MnasNet-A1 0.57× (r=128) [51]	NAS+Scale	22M	54.8%	78.1%
	EfficientNet-B^{-4} [52, 20]	NAS+Scale	24M	56.7%	79.8%
	MobileNetV3 Small 0.5× [27]	NAS+Scale	23M	58.0%	-
	TinyNet-E [20] *	NAS+Scale	25M	59.9%	81.1%
	Ours (global w)	Joint Search	25M	63.2%	84.7%
	Ours (flexble w)	Joint Search	25M	**63.9%**	**84.9%**

Table 5.8. Our joint search algorithm outperforms existing state-of-the-art tiny networks in terms of *computation-accuracy* trade-off, especially under tiny computation setting (<50M). All of ours models are derived from *the same search space*, while obtaining the best accuracy at the same computation. For models with *, we re-measure the MACs and parameters using our profiler.

extended with different input resolutions (128-256) and width multipliers (0.5×, 0.75×, 1×) and compare with other state-of-the-art methods in Table 5.8. Our joint search algorithm achieves the best accuracy under various computation budgets, despite all the networks are derived from the same search space. The improvement is most significant under a tiny computation setting: our method can achieve 4% higher accuracy compared to existing methods at 25M MACs. We also compared using a global width multiplier w for all the blocks, or choosing a flexible w for each block. We can find that allowing flexible w can further improve the accuracy under tiny computation budgets ($\leq$ 50M MACs). Therefore, we enable flexible w support for MCUNetV2 by default.

Chapter 6

Applications

In this chapter, we deploy our MCUNet models to actual MCU hardware to showcase its real-life applications. All the models are deployed on STM32F746 MCU with 320kB SRAM, 1MB Flash, and a 216MHz CPU. We provide the hardware statistics of each deployed model below:

Application	MACs	#Param	Peak SRAM	Flash	Latency
Visual wake words	6.0M	0.42M	63kB	466kB	90ms
Face & mask detection	24.5M	0.045M	217kB	91kB	395ms
Person detection	21.9M	0.082M	217kB	131kB	374ms

Table 6.1. Statistics of the deployed models for different applications.

Visual Wake Words

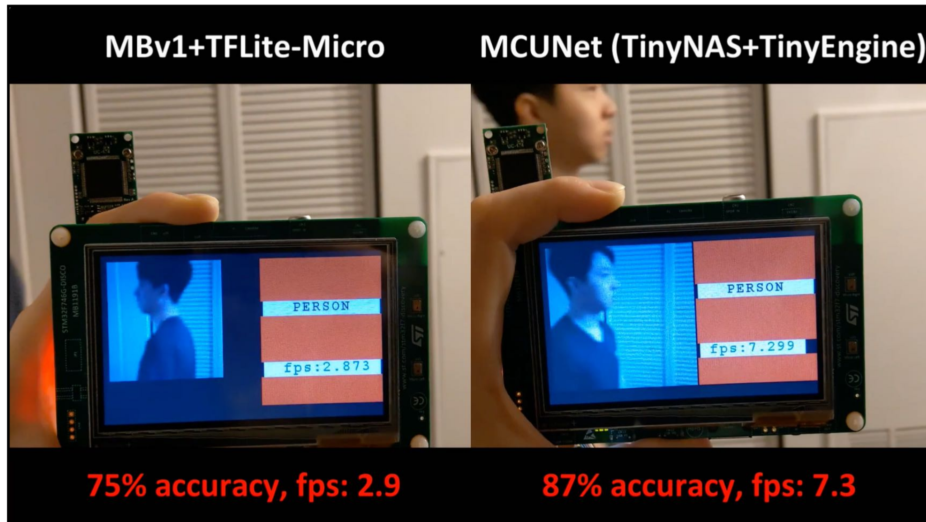
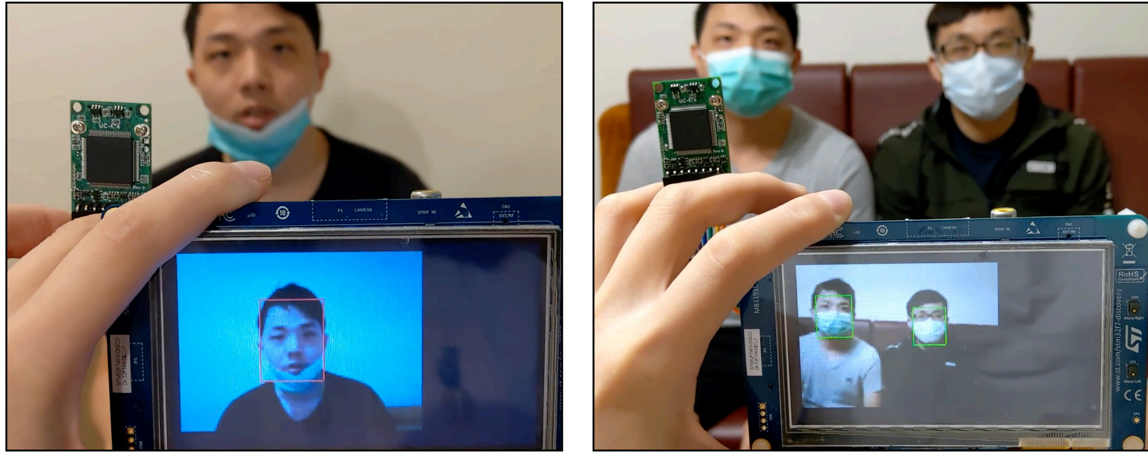


Figure 6-1. MCUNet for visual wake words on STM32F746 MCU. MCUNet can achieve 12% higher accuracy at 2.5× faster inference speed compared to the widely used industrial solution (MobileNet [28]+TF-Lite Micro [3]). The complete demo video can be found at `https://youtu.be/YvioBgtec4U`.

We deployed the models trained on the Visual Wake Words dataset [12] to STM32F746 MCU. It can detect whether a person is in front of the camera or not. It can act as an always-on, energy-efficient switch to trigger the more heavy machine learning model when a person is present. As shown in Figure 6-1, MCUNet with TinyNAS and TinyEngine achieves **12%** higher accuracy and **2.5**× faster speed compared to MobilenetV1 on TF-Lite Micro [3]. The complete demo video can be found here.

Note that we show the actual frame rate in the demo video, which includes frame capture latency overhead from the camera (around 30ms per frame). Such camera latency slows down the inference from 10 FPS to 7.3 FPS.

Face & Mask Detection



(a) Detecting one face with no mask (b) Detecting two faces with masks

Figure 6-2. MCUNet for face and mask detection. The model can detect multiple human faces appearing in the image, and whether they are wearing face masks properly (green bounding boxes) or not (red bounding boxes). The complete demo video can be found at `https://www.youtube.com/watch?v=AFsExPbxIiA`.

Wearing masks is essential to preventing the spread of COVID-19 and other respiratory infectious diseases. Here we deploy a face detection model with mask/no-mask attributes to provide an efficient and low-cost solution for monitoring face mask wearing. The model is trained on a dataset constructed following this link. As shown in Figure 6-2, with MCUNet, we can detect *multiple* faces appearing in the image and whether they are wearing face masks properly (marked with green bounding boxes) or not (marked with red bounding boxes). With our technique, we can provide a low-cost solution to help combat COVID-19.

Person Detection

(a) Detecting indoor persons

(b) Detecting outdoor persons

Figure 6-3. MCUNet for person detection on STM32F746 MCU. MCUNet can detect multiple persons at different sizes and scales. The complete demo video can be found at `https://www.youtube.com/watch?v=cr5y_AJ-LM4`.

We further train MCUNet to perform person detection on STM32F746. The model is trained on Pascal VOC [16] dataset, where only the "person" class is selected for training. As shown in Figure 6-3, MCUNet can detect multiple persons of different sizes and scales (even when truncated or occluded), providing an affordable solution for social distancing monitoring and person counting. The complete demo video can be found at this link.

Chapter 7

Conclusion

In this thesis, we study efficient algorithms and systems for tiny deep learning deployment on microcontrollers. To overcome the limited memory size, we propose MCUNet to jointly design the neural network architecture with TinyNAS and the inference library with TinyEngine. TinyNAS adopts a two-stage neural architecture search approach that first optimizes the search space to fit the resource constraints, then specializes the network architecture in the optimized search space. It is co-designed with TinyEngine, a memory-efficient inference library to expand the search space and fit a larger model. We further analyzed the memory bottleneck of convolutional neural networks and pointed out the imbalanced distribution of peak memory usage. Based on the observation, we introduced MCUNetV2 with a patch-based inference scheme to significantly reduce the peak memory required for inference.

MCUNet is the first to achieves $>$70% ImageNet top1 accuracy (71.8%) on an off-the-shelf commercial microcontroller, using 3.5$\times$ less SRAM and 5.7$\times$ less Flash compared to quantized MobileNetV2 and ResNet-18. On visual&audio wake words tasks, MCUNet achieves state-of-the-art accuracy and runs 2.4-3.4$\times$ faster than MobileNetV2 and ProxylessNAS-based solutions with 3.7-4.1$\times$ smaller peak SRAM. We are also the first to study large-scale object detection on microcontrollers. Finally, we deploy MCUNet on actual hardware to showcase real-life applications: visual wake words, face & mask detection, and person detection. Our study suggests that the era of always-on tiny machine learning on IoT devices has arrived.

Appendix A

Appendix

Profiled Model Architecture Details

We provide the details of the models profiled in Figure 3-3.

SmallCifar. SmallCifar is a small network for CIFAR [30] dataset used in the MicroTVM/μTVM post*. It takes an image of size 32×32 as input. The input image is passed through $3\times$ $\{$convolution (kernel size 5×5), max pooling$\}$. The output channels are 32, 32, 64 respectively. The final feature map is flattened and passed through a linear layer of weight 1024$\times$10 to get the logit. The model is quite small. We mainly use it to compare with MicroTVM since most of ImageNet models run OOM with MicroTVM.

ImageNet Models. All other models are for ImageNet [14] to reflect a real-life use case. The input resolution and model width multiplier are scaled down so that they can run with most of the libraries profiled. We used input resolution of 64×64 for MobileNetV2 [49] and ProxylessNAS [6], and 96×96 for MnasNet [51]. The width multipliers are 0.35 for MobileNetV2, 0.3 for ProxylessNAS and 0.2 for MnasNet.

Design Cost

There are billions of IoT devices with drastically different constraints, which requires different search spaces and model specialization. Therefore, keeping a low design cost is important.

*`https://tvm.apache.org/2020/06/04/tinyml-how-tvm-is-taming-tiny`

MCUNet is efficient in terms of neural architecture design cost. The search space optimization process takes negligible cost since no training or testing is required (it takes around 2 CPU hours to collect all the FLOPs statistics). The process needs to be done only once and can be reused for different constraints (*e.g.*, we covered two MCU devices and 4 memory constraints in Table 4). TinyNAS is an one-shot neural architecture search method without a meta controller, which is far more efficient compared to traditional neural architecture search method: it takes 40,000 GPU hours for MnasNet [51] to design a model, while MCUNet only takes 300 GPU hours, reducing the search cost by 133×. With MCUNet, we reduce the CO_2 emission from 11,345 lbs to 85 lbs per model (Figure A-1).

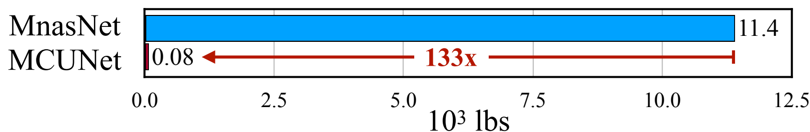


Figure A-1. Total CO_2 emission (klbs) for model design. MCUNet saves the design cost by orders of magnitude, allowing model specialization for different deployment scenarios.

Resource-Constrained Model Specialization Details

For all the experiments in our paper, we used the same training recipe for neural architecture search to keep a fair comparison.

Super network training. We first train a super network to contain all the sub-networks in the search space through *weight sharing*. Our search space is based on the widely-used mobile search space [51, 6, 55, 5] and supports variable kernel sizes for depth-wise convolution (3/5/7), variable expansion ratios for inverted bottleneck (3/4/6) and variable stage depths (2/3/4). The input resolution and width multiplier is chosen from search the space optimization technique proposed in section 3.1. The number of possible sub-networks that TinyNAS can cover in the search space is large: 2×10^{19}.

To speed up the convergence, we first train the largest sub-network inside the search space (all kernel size 7, all expansion ratio 6, all stage depth 4). We then use the trained weights to initialize the super network. Following [5], we sort the channels weights according to their importance (we used L-1 norm to measure the importance [22]), so that the most important channels are ranked higher. Then we train the super network to support different

sub-networks. For each batch of data, we randomly sample 4 sub-networks, calculate the loss, backpropogate the gradients for each sub-network, and update the corresponding weights. For weight sharing, when select a smaller kernel, *e.g.*, kernel size 3, we index the central 3×3 window from the 7×7 kernel; when selecting a smaller expansion ratio, *e.g.* 3, we index the first $3n$ channels from the $6n$ channels (n is #block input channels), as the weights are already sorted according to importance; when using a smaller stage depth, *e.g.* 2, we calculate the first 2 blocks inside the stage the skip the rest. Since we use a fixed order when sampling sub-networks, we keep the same sampling manner when evaluating their performance.

Evolution search. After super-network training, we use evolution to find the best sub-network architecture. We use a population size of 100. To get the first generation of population, we randomly sample sub-networks and keep 100 satisfying networks that fit the resource constraints. We measure the accuracy of each candidate on the independent validation set split from the training set. Then, for each iteration, we keep the top-20 candidates in the population with highest accuracy. We use crossover to generate 50 new candidates, and use mutation with probability 0.1 to generate another 50 new candidates, which form a new generation of size 100. We measure the accuracy of each candidate in the new generation. The process is repeated for 30 iterations, and we choose the sub-network with the highest validation accuracy.

Training&Testing Details

Training. The super network is trained on the training set excluding the split validation set. We trained the network using the standard SGD optimizer with momentum 0.9 and weight decay 5e-5. For super network training, we used cosine annealing learning rate [41] with a starting learning rate 0.05 for every 256 samples. The largest sub-network is trained for 150 epochs on ImageNet [14], 100 epochs on Speech Commands [54] and 30 epochs on Visual Wake Words [12] due to different dataset sizes. Then we train the super network for twice training epochs by randomly sampling sub-networks.

Validation. We evaluate the performance of each sub-network on the independent validation set split from the training set in order not to over-fit the real validation set. To evaluate each sub-network's performance during evolution search, we index and inherit the partial weights from the super network. We re-calibrate the batch normalization statistics (moving mean and variance) using 20 batches of data with a batch size 64. To evaluate the final performance on the real validation set, we also fine-tuned the best sub-network for 100 epochs on ImageNet.

Quantization. For most of the experiments (except Table 4), we used TensorFlow's int8 quantization (both activation and weights are quantized to int8). We used post-training quantization without fine-tuning which can already achieve negligible accuracy loss. We also reported the results of 4-bit integer quantization (weight and activation) on ImageNet (Table 4 of the paper). In this case, we used quantization-aware fine-tuning for 25 epochs to recover the accuracy.

Bibliography

[1] Solution to visual wakeup words challenge'19 (first place). https://github.com/mit-han-lab/VWW.

[2] Why tinyml is a giant opportunity. https://venturebeat.com/2020/01/11/why-tinyml-is-a-giant-opportunity/.

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, 2016.

[4] Gabriel Bender, Pieter-Jan Kindermans, Barret Zoph, Vijay Vasudevan, and Quoc Le. Understanding and simplifying one-shot architecture search. In *ICML*, 2018.

[5] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once for All: Train One Network and Specialize it for Efficient Deployment. In *ICLR*, 2020.

[6] Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware. In *ICLR*, 2019.

[7] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. An analysis of deep neural network models for practical applications. *arXiv preprint arXiv:1605.07678*, 2016.

[8] Alessandro Capotondi, Manuele Rusci, Marco Fariselli, and Luca Benini. Cmix-nn: Mixed low-precision cnn library for memory-constrained edge devices. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 67(5):871–875, 2020.

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *OSDI*, 2018.

[10] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *NeurIPS*, 2018.

[11] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. Pact: Parameterized clipping activation for quantized neural networks. *arXiv preprint arXiv:1805.06085*, 2018.

[12] Aakanksha Chowdhery, Pete Warden, Jonathon Shlens, Andrew Howard, and Rocky Rhodes. Visual wake words dataset. *arXiv preprint arXiv:1906.05721*, 2019.

[13] Matthieu Courbariaux and Yoshua Bengio. Binarynet: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[14] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR*, 2009.

[15] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[16] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *International journal of computer vision*, 88(2):303–338, 2010.

[17] Igor Fedorov, Ryan P Adams, Matthew Mattina, and Paul Whatmough. Sparse: Sparse architecture search for cnns on resource-constrained microcontrollers. In *NeurIPS*, 2019.

[18] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014.

[19] Zichao Guo, Xiangyu Zhang, Haoyuan Mu, Wen Heng, Zechun Liu, Yichen Wei, and Jian Sun. Single Path One-Shot Neural Architecture Search with Uniform Sampling. *arXiv*, 2019.

[20] Kai Han, Yunhe Wang, Qiulin Zhang, Wei Zhang, Chunjing Xu, and Tong Zhang. Model rubik's cube: Twisting resolution, depth and width for tinynets. *Advances in Neural Information Processing Systems*, 33, 2020.

[21] Song Han, Huizi Mao, and William J Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. In *ICLR*, 2016.

[22] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both Weights and Connections for Efficient Neural Networks. In *NeurIPS*, 2015.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *CVPR*, 2016.

[24] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. AMC: AutoML for Model Compression and Acceleration on Mobile Devices. In *ECCV*, 2018.

[25] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *ICCV*, 2017.

[26] Mark Horowitz. 1.1 computing's energy problem (and what we can do about it). In *ISSCC*. IEEE, 2014.

[27] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. Searching for MobileNetV3. In *ICCV*, 2019.

[28] Andrew G. Howard, Menglong Zhu, Bo Chen, Dimitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv*, 2017.

[29] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv preprint arXiv:1511.06530*, 2015.

[30] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[31] Liangzhen Lai, Naveen Suda, and Vikas Chandra. Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus. *arXiv preprint arXiv:1801.06601*, 2018.

[32] Tom Lawrence and Li Zhang. Iotnet: An efficient and accurate convolutional neural network for iot devices. *Sensors*, 19(24):5541, 2019.

[33] Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan Oseledets, and Victor Lempitsky. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. *arXiv preprint arXiv:1412.6553*, 2014.

[34] Edgar Liberis and Nicholas D Lane. Neural networks on microcontrollers: saving memory at inference via operator reordering. *arXiv preprint arXiv:1910.05110*, 2019.

[35] Ji Lin, Wei-Ming Chen, Yujun Lin, John Cohn, Chuang Gan, and Song Han. Mcunet: Tiny deep learning on iot devices. In *NeurIPS*, 2020.

[36] Ji Lin, Yongming Rao, Jiwen Lu, and Jie Zhou. Runtime neural pruning. In *NeurIPS*, 2017.

[37] Haoxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: Differentiable Architecture Search. In *ICLR*, 2019.

[38] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. *arXiv preprint arXiv:2103.14030*, 2021.

[39] Zechun Liu, Haoyuan Mu, Xiangyu Zhang, Zichao Guo, Xin Yang, Kwang-Ting Cheng, and Jian Sun. MetaPruning: Meta Learning for Automatic Neural Network Channel Pruning. In *ICCV*, 2019.

[40] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *ICCV*, 2017.

[41] Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016.

[42] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design. In *ECCV*, 2018.

[43] Ilija Radosavovic, Justin Johnson, Saining Xie, Wan-Yen Lo, and Piotr Dollár. On network design spaces for visual recognition. In *ICCV*, 2019.

[44] Ilija Radosavovic, Raj Prateek Kosaraju, Ross Girshick, Kaiming He, and Piotr Dollár. Designing network design spaces. *arXiv preprint arXiv:2003.13678*, 2020.

[45] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *ECCV*, 2016.

[46] Joseph Redmon and Ali Farhadi. Yolo9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7263–7271, 2017.

[47] Joseph Redmon and Ali Farhadi. YOLOv3: An Incremental Improvement. *arXiv*, 2018.

[48] Manuele Rusci, Alessandro Capotondi, and Luca Benini. Memory-driven mixed low precision quantization for enabling deep network inference on microcontrollers. In *MLSys*, 2020.

[49] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *CVPR*, 2018.

[50] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[51] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. MnasNet: Platform-Aware Neural Architecture Search for Mobile. In *CVPR*, 2019.

[52] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114. PMLR, 2019.

[53] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. HAQ: Hardware-Aware Automated Quantization with Mixed Precision. In *CVPR*, 2019.

[54] Pete Warden. Speech commands: A dataset for limited-vocabulary speech recognition. *arXiv preprint arXiv:1804.03209*, 2018.

[55] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. FBNet: Hardware-Aware Efficient ConvNet Design via Differentiable Neural Architecture Search. In *CVPR*, 2019.

[56] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. In *CVPR*, 2018.

[57] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

[58] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. Trained ternary quantization. *arXiv preprint arXiv:1612.01064*, 2016.

[59] Barret Zoph and Quoc V Le. Neural Architecture Search with Reinforcement Learning. In *ICLR*, 2017.

[60] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning Transferable Architectures for Scalable Image Recognition. In *CVPR*, 2018.