MCUNet: Tiny Deep Learning on IoT Devices

Ji Lin[1] **Wei-Ming Chen**[1,2] **Yujun Lin**[1] **John Cohn**[3] **Chuang Gan**[3] **Song Han**[1]
[1]MIT [2]National Taiwan University [3]MIT-IBM Watson AI Lab
https://tinyml.mit.edu

Abstract

Machine learning on tiny IoT devices based on microcontroller units (MCU) is appealing but challenging: the memory of microcontrollers is 2-3 orders of magnitude smaller even than mobile phones. 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 engine 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×, and accelerating the inference by 1.7-3.3× compared to TF-Lite Micro [3] and CMSIS-NN [28]. MCUNet is the first to achieves >70% ImageNet top1 accuracy on an off-the-shelf commercial microcontroller, using 3.5× less SRAM and 5.7× 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× faster than MobileNetV2 and ProxylessNAS-based solutions with 3.7-4.1× smaller peak SRAM. Our study suggests that the era of always-on tiny machine learning on IoT devices has arrived.

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, 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 [21] exceeds the storage limit by 100×, MobileNetV2 [44] exceeds the peak memory limit by 22×. Even the int8 quantized version of MobileNetV2 still exceeds the memory limit by 5.3×*, showing a big gap between the desired and available hardware capacity.

Tiny AI is fundamentally different from cloud AI and mobile AI. Microcontrollers are bare-metal devices that do not have an operating system, nor do they have DRAM. There-

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

34th Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada.

Table 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.

	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

fore, 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 [25, 44, 49] and neural architecture search methods [45, 6, 48, 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. In fact, we find that at similar ImageNet accuracy (70%), MobileNetV2 [44] reduces the *model size* by 4.6× compared to ResNet-18 [21] (Figure 1), but the peak *activation size* increases by 1.8×, making it even more difficult to fit the SRAM on microcontrollers. There is limited literature [16, 31, 43, 29] 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.

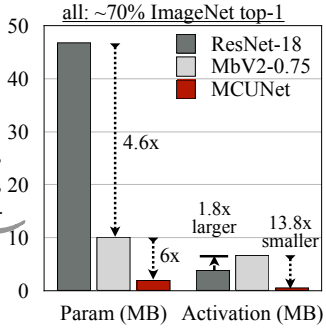
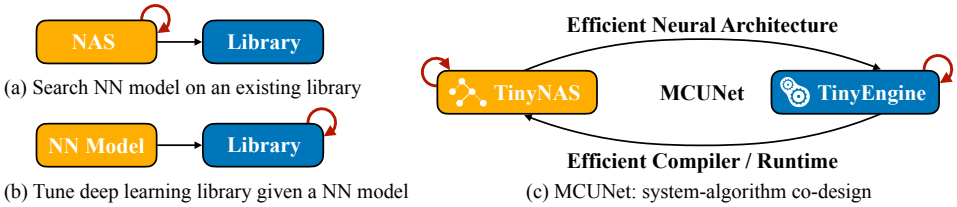


Figure 1. MobileNetV2 reduces model size but not peak memory, while MCUNet effectively reduces both parameter size and activation size.

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 [39], 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 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.

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.



(a) Search NN model on an existing library

(b) Tune deep learning library given a NN model

(c) MCUNet: system-algorithm co-design

Figure 2. 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.

2 Background

Microcontrollers have tight memory: for example, only 320kB SRAM and 1MB Flash for a popular ARM Cortex-M7 MCU STM32F746. 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 constrains the activation size (read&write); Flash constrains the model size (read-only).

Deep Learning Inference on Microcontrollers. Deep learning inference on microcontrollers is a fast-growing area. Existing frameworks such as TensorFlow Lite Micro [3], CMSIS-NN [28], CMix-NN [8], and MicroTVM [9] 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.

Efficient Neural Network Design. 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 [20, 23, 32, 35, 22, 34] and quantization [19, 51, 40, 50, 13, 11, 46] to remove redundancy and reduce complexity. Tensor decomposition [30, 17, 26] also serves as an effective compression method. Another way is to directly design an efficient and mobile-friendly network [25, 44, 37, 49, 37]. Recently, neural architecture search (NAS) [52, 53, 33, 6, 45, 48] dominates efficient network design.

The performance of NAS highly depends on the quality of the search space [39]. Traditionally, people follow manual design heuristics for NAS search space design. For example, the widely used mobile-setting search space [45, 6, 48] originates from MobileNetV2 [44]: 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.

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 2). 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) [45, 6, 48], 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.

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

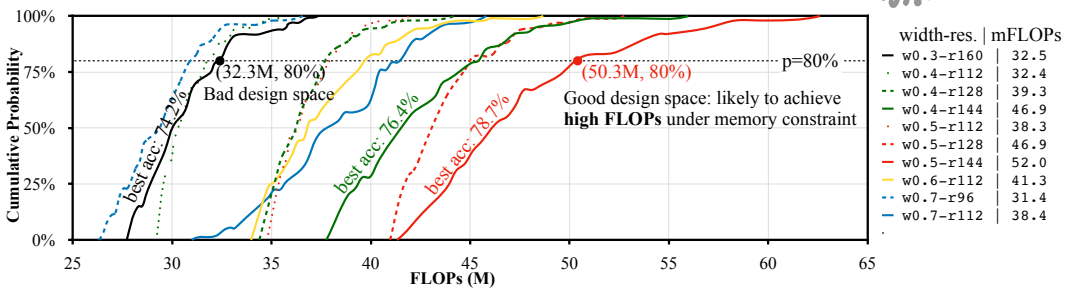


Figure 3. TinyNAS selects the best search space by analyzing the FLOPs CDF of different search spaces. Each curve represents a design space. Our insight is that the design space that is more likely to produce *high FLOPs* models under the memory constraint gives higher model capacity, thus more likely 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}`.

width multiplier of the mobile search space [45]. 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 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* [38], which is computationally heavy due to tremendous training, we only collect the CDF of *FLOPs* (see Figure 3). The intuition is that, within the same model family, the accuracy is usually positively related to the computation [7, 22]. 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 4.5.

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. 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 green space (average FLOPs 46.9M), showing the effectiveness of automated search space optimization. We will elaborate more on the ablations in Section 4.5.

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, 18] 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 TinyML: the efficiency of the inference library matters a lot to both the latency and accuracy of the searched model. Specifically,

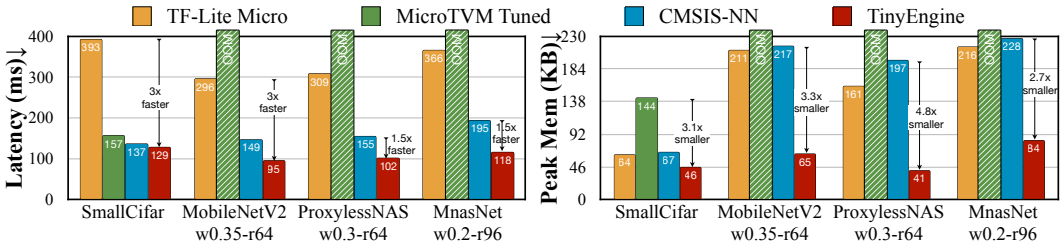


Figure 4. TinyEngine achieves higher inference efficiency than existing inference frameworks while reducing the memory usage. **Left**: TinyEngine is 3× and 1.6× faster than TF-Lite Micro and CMSIS-NN, respectively. Note that if the required memory exceeds the memory constraint, it is marked with "OOM" (out of memory). **Right**: 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. We scale the width multiplier and input resolution so that most libraries can fit the neural network (denoted by w{}-r{}).

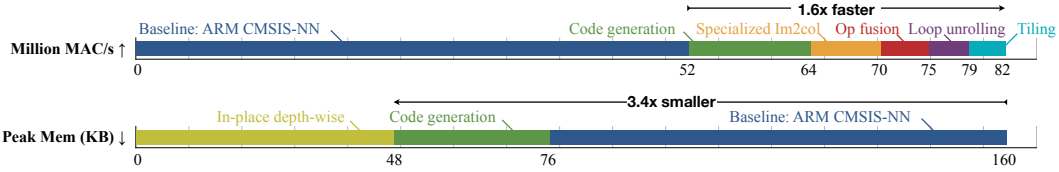


Figure 5. TinyEngine outperforms existing libraries by eliminating runtime overheads, specializing each optimization technique, and adopting in-place depth-wise convolution. This effectively enlarges design space for TinyNAS under a given latency/memory constraint.

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.

From interpretation to code generation. 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 runtime memory, the most expensive resource in MCU, to store the meta-information (such as model structure parameters). Instead, TinyEngine offloads these operations from runtime to compile time, and only generates the code that will be executed by the TinyNAS model. Thanks to the algorithm and system co-design, we have full control over what model to run, and the generated code is fully specialized for TinyNAS models. It not only avoids the time for runtime interpretation, but also frees up the memory usage to allow larger models to run. Compared to CMSIS-NN, TinyEngine reduced memory usage by 2.1× and improve inference efficiency by 22% via code generation, as shown in Figures 4 and 5.

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 6, such model-adaptive compilation reduces code size by up to 4.5× and 5.0× compared to TF-Lite Micro and CMSIS-NN, respectively.

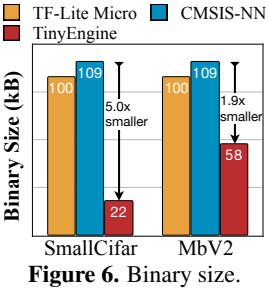


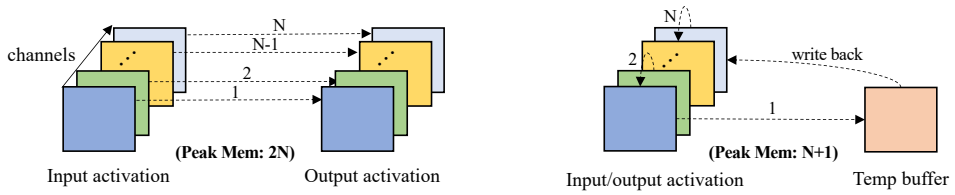
Figure 6. Binary size.

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 $\boldsymbol{L}$,

$$M = \max\left(\text{kernel size}^2_{L_i} \cdot \text{in channels}_{L_i}; \forall L_i \in \boldsymbol{L}\right). \quad (1)$$

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,

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



(a) Depth-wise convolution (b) In-place depth-wise convolution

Figure 7. 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 5, the model-adaptive im2col operation improved inference efficiency by 13%.

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 5.

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 7. This method reduces the measured memory usage by 1.6× as shown in Figure 5.

4 Experiments

4.1 Setups

Datasets. We used 3 datasets as benchmark: ImageNet [14], Visual Wake Words (VWW) [12], and Speech Commands (V2) [47]. 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 [27] 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 Flash). By default, we use STM32F746 to report the results unless otherwise specified. All the latency is normalized to STM32F746 with 216MHz CPU.

Table 2. System-algorithm co-design (TinyEngine + TinyNAS) achieves the highest ImageNet accuracy of models runnable on a microcontroller.

Library \ Model	S-MbV2	S-Proxyless	TinyNAS
CMSIS-NN [28]	35.2%	49.5%	55.5%
TinyEngine	47.4%	56.4%	**61.8%**

Table 3. MCUNet outperforms the baselines at various latency requirements. Both TinyEngine and TinyNAS bring significant improvement on ImageNet.

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

	Quantization	STM32F412 (256kB, 1MB)	STM32F746 (320kB, 1MB)	STM32F765 (512kB, 1MB)	STM32H743 (512kB, 2MB)
Rusci *et al.* [43]	Mixed	60.2%	-	62.9%	68.0%
MCUNet	4-bit	**62.0%**	**63.5%**	**65.9%**	**70.7%**

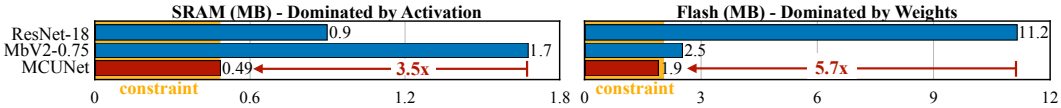


Figure 8. MCUNet reduces the the SRAM memory by **3.5**× and Flash usage by **5.7**× compared to MobileNetV2 and ResNet-18 (8-bit), while achieving better accuracy (70.7% *vs*. 69.8% ImageNet top-1).

4.2 Large-Scale Image Recognition on Tiny Devices

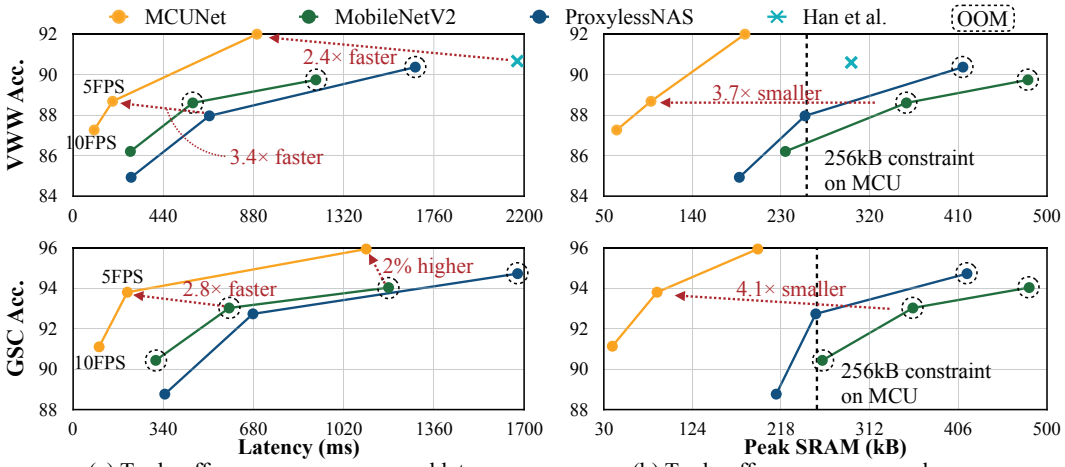
With our system-algorithm co-design, we achieve record high accuracy (70.7%) on large-scale ImageNet recognition on microcontrollers. We co-optimize TinyNAS and TinyEngine to find the best runnable network. We compare our results to several baselines. We generate the *best scaling* of MobileNetV2 [44] (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 [24]-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 2, when running on a tight budget of 320kB SRAM and 1MB Flash, the optimal scaling of MobileNetV2 and ProxylessNAS models only achieve 35.2% and 49.5% top-1 accuracy on ImageNe using CMSIS-NN [28]. 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 3). 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 4. MCUNet can handle diverse hardware resources on different MCUs with Cortex-M4 (F412) and M7 (F746, F765, H743) core.

[†] *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.



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

Figure 9. 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.

Table 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.

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

Without mixed-precision, we can already outperform the existing state-of-the-art [43] on microcontrollers, showing the effectiveness 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 8) to fit the tiny memory size on microcontrollers.

4.3 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 9. 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 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.

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

Table 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.

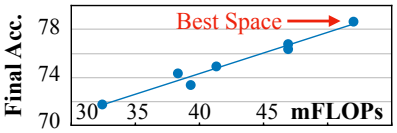
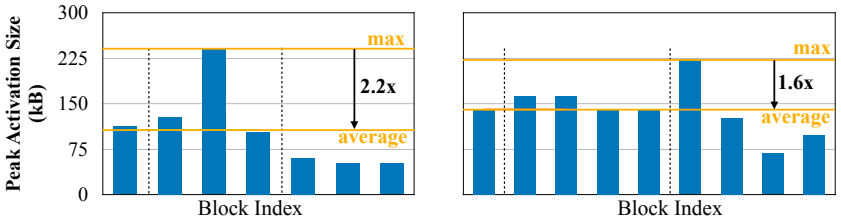


Figure 10. Search space with higher mean FLOPs leads to higher final accuracy.



(a) 0.3x MbV2: highly imbalanced activation (b) TinyNAS: more balanced activation

Figure 11. The peak memory of MobileNetV2 is *dominated* by one block, while the MCUNet searched by TinyNAS has more *balanced* activation size per block, fitting a higher model capacity at the same SRAM budget.

4.4 Object Detection on MCUs

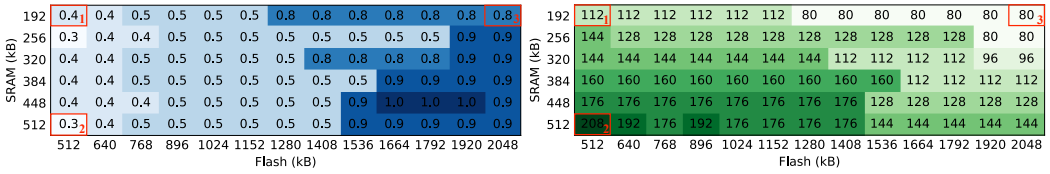
To show the generalization ability of MCUNet framework across different tasks, we apply MCUNet to object detection. Object detection is particularly challenging for memory-limited MCUs: a high-resolution input is usually required to detect the relatively small objects, which will increase the peak performance significantly. We benchmark the object detection performance of our MCUNet and scaled MobileNetV2+CMSIS-NN on on Pascal VOC [15] dataset. We used YOLOv2 [41] as the detector; other more advanced detectors like YOLOv3 [42] 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. Under tight memory budget (only 512kB SRAM and 2MB Flash), MCUNet significantly improves the mAP by 20%, which makes AIoT applications more accessible.

4.5 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. 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 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 10. We can see a clear positive relationship, which backs our algorithm.

Per-block peak memory analysis. We compare the peak memory distribution of scaled-down MobileNetV2 (0.3×) and TinyNAS searched model under 320kB SRAM limit in Figure 11. We plot the per block activation size (not including other runtime buffers) of the first two-stages, which have the biggest activation size as the memory bottleneck. MobileNetV2 has a highly *imbalanced* peak activation size: one single block has 2.2× peak activation size compared to the average. To scale down the network and fit the SRAM constraints, other blocks are forced to scale to a very small capacity. On the other hand, MCUNet searched by TinyNAS has more a balanced peak memory size, leading to a overall higher network capacity. The memory allocation is automatically discovered when optimizing the accuracy/memory trade-off by TinyNAS (Section 3.1), without human heuristics on the memory distribution.



(a) Best width setting. **(b)** Best resolution setting.

Figure 12. Best search space configurations under different SRAM and Flash constraints.

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 12. 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.

Evolution search. The curve of evolution search on different inference library is in Figure 13. 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.

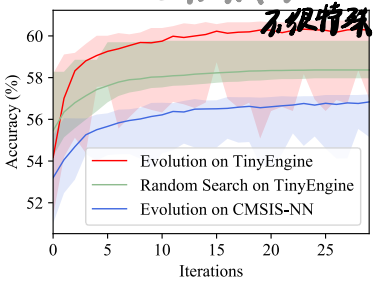


Figure 13. Evolution progress.

5 Conclusion

We propose MCUNet to jointly design the neural network architecture (TinyNAS) and the inference library (TinyEngine), enabling deep learning on tiny hardware resources. We achieved a record ImageNet accuracy (70.7%) on off-the-shelf microcontrollers, and accelerated the inference of wake word applications by 2.4-3.4×. Our study suggests that the era of always-on tiny machine learning on IoT devices has arrived.

Statement of Broader Impacts

Our work is expected to enable tiny-scale deep learning on microcontrollers and further democratize deep learning applications. Over the years, people have brought down the cost of deep learning inference from $5,000 workstation GPU to $500 mobile phones. We now bring deep learning to microcontrollers costing $5 or even less, which greatly expands the scope of AI applications, making AI much more accessible.

Thanks to the low cost and large quantity (250B) of commercial microcontrollers, we can bring AI applications to every aspect of our daily life, including personalized healthcare, smart retail, precision agriculture, smart factory, *etc*. People from rural and under-developed areas without Internet or high-end hardware can also enjoy the benefits of AI. Our method also helps combat COVID-19 by providing affordable deep learning solutions detecting face masks and people gathering on edge devices without sacrificing privacy.

With these always-on low-power microcontrollers, we can process raw sensor data right at the source. It helps to protect privacy since data no longer has to be transmitted to the cloud but processed locally.

Acknowledgments

We thank MIT Satori cluster for providing the computation resource. We thank MIT-IBM Watson AI Lab, Qualcomm, NSF CAREER Award #1943349 and NSF RAPID Award #2027266 for supporting this research.

References

[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] 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.

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

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

[18] 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.

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

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

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

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

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

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

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

[30] 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.

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

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

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

[34] 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.

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

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

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

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

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

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

[41] 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.

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

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

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

[45] 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.

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

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

[48] 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.

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

[50] 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.

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

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

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

A Demo Video

We release a demo video of MCUNet running the visual wake words dataset[12] in this link. MCUNet with TinyNAS and TinyEngine achieves **12%** higher accuracy and **2.5**× faster speed compared to MobilenetV1 on TF-Lite Micro [3].

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.

B Profiled Model Architecture Details

We provide the details of the models profiled in Figure 4.

SmallCifar. SmallCifar is a small network for CIFAR [27] 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 1024×10 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 [44] and ProxylessNAS [6], and 96×96 for MnasNet [45]. The width multipliers are 0.35 for MobileNetV2, 0.3 for ProxylessNAS and 0.2 for MnasNet.

C 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.

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 [45] 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 14).

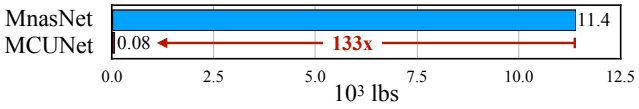


Figure 14. Total CO_2 emission (klbs) for model design. MCUNet saves the design cost by orders of magnitude, allowing model specialization for different deployment scenarios.

D 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 [45, 6, 48, 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 [20]), 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,

[‡] https://tvm.apache.org/2020/06/04/tinyml-how-tvm-is-taming-tiny

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.

E 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 [36] 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 [47] 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.

F Changelog

v1 Initial preprint release.

v2 NeurIPS 2020 camera ready version. We add the in-place depth-wise convolution technique to TinyEngine (Figure 7), which further reduces the peak memory size for inference. Part of the results in Table 2, 3, 4 are updated since the new version of TinyEngine can hold a larger model capacity now. The peak SRAM statistics in Figure 9 are also reduced. We provide the per-block peak memory distribution comparison (Figure 11) to provide network design insights.