

Tiny Machine Learning: Progress and Futures

Ji Lin Ligeng Zhu Wei-Ming Chen Wei-Chen Wang Song Han
Massachusetts Institute of Technology
<https://tinymml.mit.edu>

Abstract—Tiny Machine Learning (TinyML) is a new frontier of machine learning. By squeezing deep learning models into billions of IoT devices and microcontrollers (MCUs), we expand the scope of AI applications and enable ubiquitous intelligence. However, TinyML is challenging due to hardware constraints: the tiny memory resource makes it difficult to hold deep learning models designed for cloud and mobile platforms. There is also limited compiler and inference engine support for bare-metal devices. Therefore, we need to co-design the algorithm and system stack to enable TinyML. In this review, we will first discuss the definition, challenges, and applications of TinyML. We then survey the recent progress in TinyML and deep learning on MCUs. Next, we will introduce MCUNet, showing how we can achieve ImageNet-scale AI applications on IoT devices with *system-algorithm co-design*. We will further extend the solution from *inference to training* and introduce tiny on-device training techniques. Finally, we present future directions in this area. Today’s “large” model might be tomorrow’s “tiny” model. The scope of TinyML should evolve and adapt over time.

Index Terms—TinyML, Efficient Deep Learning, On-Device Training, Learning on the Edge

I. OVERVIEW OF TINY MACHINE LEARNING

Machine learning (ML) has made significant impacts on various fields, including vision, language, and audio. However, state-of-the-art models often come at the cost of high computation and memory, making them expensive to deploy. To address this, researchers have been working on efficient algorithms, systems, and hardware to reduce the cost of machine learning models in various deployment scenarios. There are two main subdomains of efficient ML: EdgeML and CloudML (Figure 1). While CloudML focuses on improving latency and throughput on cloud servers, EdgeML focuses on improving energy efficiency, latency, and privacy on edge devices. These two domains also intersect in areas such as hybrid inference [1, 2], over-the-air (OTA) updates, and federated learning between the edge and cloud [3]. In recent years, there has been significant progress in extending the scope of EdgeML to ultra-low-power devices such as IoT devices and microcontrollers, known as TinyML.

TinyML has several key advantages. It enables machine learning using only a few hundred kilobytes of memory which greatly reduces the cost. With billions of IoT devices producing more and more data in our daily lives, there is a growing need for low-power, always-on, on-device AI. By performing on-device inference near the sensor, TinyML enables better

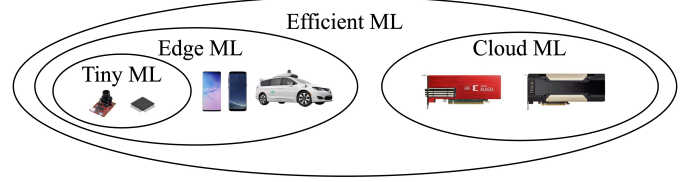


Fig. 1. Efficiency is critical for CloudML, EdgeML, and TinyML. CloudML targets high-throughput accelerators like GPUs, while EdgeML focuses on portable devices like mobile phones. TinyML further pushes the efficiency boundary, enabling powerful ML models to run on ultra-low-power devices such as microcontrollers.

responsiveness and privacy while reducing the energy cost associated with wireless communication. On-device processing of data can be beneficial for applications where real-time decision-making is crucial, such as autonomous vehicles.

In addition to inference, we push the frontier of TinyML to enable on-device training on IoT devices. It revolutionizes EdgeAI through continuous and lifelong learning. Edge device can finetune the model on itself rather than transmitting data to cloud servers, which protects privacy. On-device learning has numerous benefits and a variety of applications. For example, home cameras can continuously recognize new faces, and email clients can gradually improve their prediction by updating customized language models. It also enables IoT applications that do not have a physical connection to the internet to adapt to the environment, such as precision agriculture and ocean sensing.

In this review, we will first discuss the definition and challenges of TinyML, analyzing why we can’t directly scale mobile ML or cloud ML models for tinyML. Then we delve into the importance of system-algorithm co-design in TinyML. We will then survey recent literature and the progress of the field, presenting a holistic survey and comparison in Tables II and III. Next, we will introduce our TinyML project, MCUNet, which combines efficient system and algorithm design to enable TinyML for both inference to training. Finally, we will discuss several emerging topics for future research directions in the field.

A. Challenges of TinyML

The success of deep learning models often comes at the cost of high computation, which is not feasible for use in TinyML applications due to the strict resource constraints of devices such as microcontrollers. Deploying and training AI models on MCU is extremely hard: No DRAM, no operating systems (OS), and strict memory constraints (SRAM is smaller than 256kB, and FLASH is read-only). The available resources on these devices are orders of magnitude smaller than those

TABLE I. 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.

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

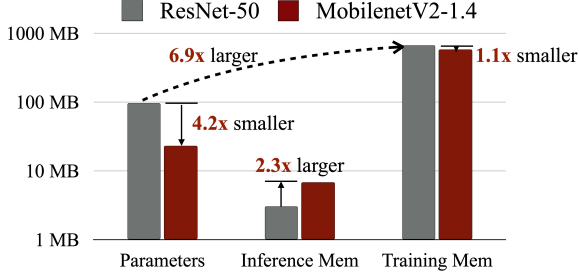


Fig. 2. We can't directly scale mobile ML or cloud ML models for TinyML. MobilenetV2 [4] with a width of 1.4 was used for the experiments. The batch size was set to 1 for inference and 8 for training. While MobilenetV2 reduces the number of parameters by 4.2 \times compared to ResNet, the peak memory usage increases by 2.3 \times for inference and only improves by 1.1 \times for training. Additionally, the total required training memory is 6.9 \times larger than the memory needed for inference. These results demonstrate the significant memory bottleneck for TinyML, and the bottleneck is the activation memory, not the number of parameters.

available on mobile platforms (see Table I). Previous work in the field has either (I) focused on reducing model parameters without addressing the real bottleneck of activations, or (II) only optimized operator kernels without considering improving the network architecture design. Neither of which considers the problem from a co-design perspective, and this has led to less optimal solutions for TinyML applications. We observe several unique challenges of TinyML and postulate how they might be overcome:

1) Models designed for mobile platforms does not fit TinyML

There has been a lot of effort optimizing deep learning models for mobile platforms like MobileNets [5, 4] and ShuffleNet [6]. However, since mobile devices have sufficient memory resources (Table I), the model designs focus on parameters/FLOPs/latency reduction but not peak memory usage. As shown in Figure 2 Left and Middle, comparing two models with the same level of ImageNet accuracy, MobileNetV2-1.4 has 4.2 \times smaller model size compared to ResNet-50, but its peak memory even larger by 2.3 \times . Using MobileNet designs does not adequately address the SRAM limit, instead, it actually makes the situation even worse compared to ResNet. Therefore, we need to rethink the model design principles for TinyML.

2) Directly adapting models for inference does not work for tiny training.

Training poses an even greater challenge in terms of resource constraints, as intermediate activations must be stored in

order to compute backward gradients. When moving from inference to training with full backpropagation, the required memory increases by a factor of 6.9. As shown in Figure 2, the training memory requirements of MobileNets are not much better than ResNets (improved by only 10%). Tiny IoT devices such as microcontrollers typically have a limited SRAM size, such as 256KB, which is barely enough for the inference of deep learning models, let alone training. Previous work in the cloud and mobile AI has focused on reducing FLOPs [5, 4, 7] or only optimizing inference memory [8, 9]. However, even using memory-efficient inference models such as MCUNet [8] to bridge the three orders of magnitude gap, training is still too expensive for tiny platforms. If we follow conventional full model update schemes, the model must be scaled down significantly to fit within the tight memory constraints, resulting in low accuracy. This highlights the need to redesign backpropagation schemes and investigate new learning algorithms to reduce the main activation memory bottleneck and enable fast and accurate training on tiny devices. In Section IV, we will discuss this issue in detail and introduce the concept of sparse layer and sparse tensor updates.

3) Co-design is necessary for TinyML

Co-design is necessary for TinyML because it allows us to fully customize the solutions that are optimized for the unique constraints of tiny devices. Previous neural architectures like MobileNets [5, 4], and ResNets [10] are designed for mobile/cloud scenarios but not well-suited for tiny hardware. Therefore, we need to design neural architectures that are suitable for TinyML applications. On the other hand, existing deep training frameworks are optimized for cloud servers and lack support for memory-efficient forward and backward, thus cannot fit into tiny devices. The huge gap (>1000 \times) between the resources of tiny IoT devices and the requirements of current frameworks prohibits the usage. To address these challenges, it is necessary to develop algorithms, systems, and training techniques that are specifically tailored to the settings of these tiny platforms.

B. Applications of TinyML

By democratizing costly deep learning models to IoT devices, TinyML has many practical applications. Some example applications include:

- **Personalized healthcare:** TinyML can allow wearable devices, such as smartwatches, to continuously track the activities and oxygen saturation status of the user in order to provide health suggestions [11, 12, 13, 14]. Body

pose estimation is also a crucial application for elderly healthcare [15].

- Wearable applications: TinyML can assist people with wearable or IoT devices for speech applications, *e.g.*, keyword spotting, automatic speech recognition, and speaker verification [16, 17, 18].
- Smart home: TinyML can enable object detection, image recognition, and face detection on IoT devices to build smart environments, such as smart homes and hospitals [19, 20, 21, 22, 23].
- Human-machine interface: TinyML can enable human-machine interface applications, like hand gesture recognition [24, 25, 26, 27]. TinyML is also capable of predicting and recognizing sign languages [28].
- Smart vehicle and transportation: TinyML can perform object detection, lane detection, and decision making without a cloud connection, achieving high-accuracy and low-latency results for autonomous driving scenarios [29, 30, 31].
- Anomaly detection: TinyML can equip robots and sensors with the capability to perform anomaly detection to reduce human efforts [32, 33, 34].
- Ecology and agriculture: TinyML can also help with ecological, agricultural, environmental, and phenomics applications so as to conserve endangered species or forecast weather activities [35, 36, 37, 38, 39, 40].

Overall, the potential applications of TinyML are diverse and numerous, and will expand as the field continues to advance.

II. RECENT PROGRESS IN TINYML

A. Recent Progress on TinyML Inference

TinyML and deep learning on MCUs have seen rapid growth in industry and academia in recent years. The primary challenge of deploying deep learning models on MCUs for inference is the limited memory and computation available on these devices. For example, a popular ARM Cortex-M7 MCU, the STM32F746, has only 320KB of SRAM and 1MB of flash memory. In deep learning scenarios, SRAM limits the size of activations (read and write) while flash memory limits the size of the model (read-only). In addition, the STM32F746 has a processor with a clock speed of 216 MHz, which is 10 to 20 times lower than laptops. To enable deep learning inference on MCUs, researchers have proposed various designs and solutions to address these issues. Table II summarizes the recent related studies on TinyML targeting MCUs, including both algorithm solutions and system solutions. In Table III, we measured three different metrics (*i.e.*, latency, peak memory, and flash usage) of four representative related studies (*i.e.*, CMSIS-NN [41], X-Cube-AI [42], TinyEngine [8], and TF-Lite Micro [47]) on an identical MCU (STM32H743) and identical datasets (VWW and Imagenet), in order to provide a more accurate and transparent comparison.

a) Algorithm Solutions

The importance of neural network's efficiency to the overall performance of a deep learning system cannot be overstated. Compressing off-the-shelf networks by removing redundancy and reducing complexity through pruning [57, 58, 59, 60,

61, 62] and quantization [63, 64, 65, 66, 67, 68, 69, 70] are two popular methods to improve network efficiency. Tensor decomposition [71, 72, 73] is also an efficient compression technique. In order to enhance network efficiency, knowledge distillation is also a method to transfer information learned from one teacher model to another student model [74, 75, 76, 77, 78, 79, 80, 81]. Another method is to directly design tiny and efficient network structures [5, 4, 6, 7]. Recently, neural architecture search (NAS) has dominated the design of efficient networks [82, 83, 84, 85, 86, 87].

To make deep learning feasible on MCUs, researchers have proposed various algorithm solutions. Rusci et al. proposed a rule-based quantization strategy that minimizes the bit precision of activations and weights in order to reduce memory usage [45]. Depending on the memory constraints of various devices, this method can quantize activations and weights with 8 bits, 4 bits, or 2 bits of mixed precision. On the other hand, although neural architecture search (NAS) has been successful in finding efficient network architectures, its effectiveness is highly dependent on the quality of the search space [88]. For MCUs with limited memory, standard model designs and appropriate search spaces are especially lacking. To address this, TinyNAS, proposed as part of MCUNet, employs a two-step NAS strategy that optimizes the search space according to the available resources [8]. TinyNAS then specializes network architectures within the optimized search space, allowing it to automatically deal with a variety of constraints (*e.g.*, device, latency, energy, memory) at low search costs. MicroNets observed that the inference latency of networks in the NAS search space for MCUs varies linearly with the number of FLOPs in the model [48]. As a result, it proposed differentiated NAS, which treats the FLOPs as a proxy for latency in order to achieve both low memory consumption and high speed. MCUNetV2 identified that the imbalanced memory distribution is the primary memory bottleneck in most convolutional neural network designs, where the memory usage of the first few blocks is an order of magnitude greater than the rest of the network [9]. As a result, this study proposed receptive field redistribution to shift the receptive field and FLOPs to a later stage, reducing the halo's computation overhead. To minimize the difficulty of manually redistributing the receptive field, this study also automated the neural architecture search process to simultaneously optimize the neural architecture and inference scheduling. UDC explored a broader design search space to generate compressible neural networks with high accuracy for neural processing units (NPU), which can address the memory problem by exploiting model compression with a broader range of weight quantization and sparsity [51].

b) System Solutions

In recent years, popular training frameworks such as PyTorch [89], TensorFlow [90], MXNet [91], and JAX [92] have contributed to the success of deep learning. However, these frameworks typically rely on a host language (*e.g.*, Python) and various runtime systems, which adds significant overhead and makes them incompatible with tiny edge devices. Emerging frameworks such as TVM [93], TF-Lite [94], MNN [95], NCNN [96], TensorRT [97], and OpenVino [98]

TABLE II. Specification and performance comparison of recent progress on TinyML research targeting microcontrollers.

	CMSIS-NN arXiv’18 [41]	X-Cube-AI [42]	MicroTVM [43]	Liberis et al. MLSys’20 [44]	Rusci et al. MLSys’20 [45]	CMix-NN TCAS’20 [46]
On-Device Training or Inference	Inference	Inference	Inference	Inference	Inference	Inference
Measured Device	MCU (STM32H743)	MCU (STM32H743)	MCU (STM32F746)	MCU (STM32F767)	MCU (STM32H743)	MCU (STM32H743)
Dataset	ImageNet	ImageNet	CIFAR-10	VWW	ImageNet	ImageNet
Model	MobileNetV1	MobileNetV1	SmallCifar	MobileNetV1	MobileNetV1	MobileNetV1
Input Resolution	192	192	32	96	224	192
Width Multiplier	0.5	0.5	1.0	0.25	0.75	0.5
Data Bitwidth	INT8	INT8	INT8	INT8	Mixed	INT8
Latency	510 ms ¹	437 ms ¹	157 ms ³	1325 ms	1860 ms	677 ms
Peak Memory	< 1 MB ²	< 1 MB ²	144 KB ³	55 KB	< 512 KB ²	< 512 KB ²
Flash Usage	1.4 MB ¹	1.4 MB ¹	< 1 MB ²	< 2 MB ²	2 MB	1.4 MB
Energy Consumption	135 mJ ¹	115 mJ ¹	-	735 mJ	491 mJ ⁴	179 mJ ⁴
Top-1 Accuracy	59.5% ¹	59.5% ¹	-	~76%	68.2%	62.9%
	MCUNetV1 NeurIPS’20 [8]	TF-Lite Micro MLSys’21 [47]	MicroNets MLSys’21 [48]	MCUNetV2 NeurIPS’21 [9]	TinyOps CVPRW’22 [49]	TinyMaix [50]
On-Device Training or Inference	Inference	Inference	Inference	Inference	Inference	Inference
Measured Device	MCU (STM32H743)	MCU (STM32F746)	MCU (STM32F746)	MCU (STM32H743)	MCU (STM32F746)	MCU (STM32H750)
Dataset	ImageNet	ImageNet	VWW	ImageNet	ImageNet	VWW
Model	MCUNet	MobileNetV2	MicroNet-VWW-1	MCUNet	MNASNet	MobileNetV1
Input Resolution	160	64	160	224	96	96
Width Multiplier	N/A	0.35	N/A	N/A	1.0	0.25
Data Bitwidth	INT8	INT8	INT8	INT8	INT8	Mixed
Latency	463 ms	296 ms ³	1133 ms	859 ms	866 ms	64 ms
Peak Memory	416 KB	211 KB ³	285 KB	434 KB	397 KB	54 KB
Flash Usage	1.7 MB	< 1 MB ²	0.8 MB	1.8 MB	4.7 MB	0.2 MB
Energy Consumption	-	-	479 mJ	-	546 mJ	-
Top-1 Accuracy	68.0%	-	88.0%	71.8%	64.0%	~76%
	UDC NeurIPS’22 [51]	TinyTL NeurIPS’20 [52]	TinyOL IJCNN’21 [53]	POET ICML’22 [54]	MiniLearn EWSN’22 [55]	MCUNetV3 NeurIPS’22 [56]
On-Device Training or Inference	Inference	Training	Training	Training	Training	Training
Measured Device	N/A (Simulation)	N/A (Simulation)	MCU (nRF52840)	MCU (nRF52840)	MCU (nRF52840)	MCU (STM32F746)
Dataset	ImageNet	CIFAR-10	Self-Collected	CIFAR-10	KWS-subset	VWW
Model	UDC	ProxylessNAS-Mobile	Autoencoder	ResNet-18	Customized	MCUNet
Data Bitwidth	Mixed	FP32	FP32	FP32	Mixed	INT8
Latency	-	-	-	49 ms	93 ms	546 ms
Peak Memory	-	65 MB	< 256 KB ²	271 KB	196 KB	173 KB
Flash Usage	1.27 MB	-	< 1 MB ²	< 1 MB ²	0.9 MB	0.7MB
Energy Consumption	-	-	-	868 mJ	1486 mJ	-
Top-1 Accuracy	72.1%	96.1%	-	95.5%	88.5%	89.3%

¹Measured by CMix-NN paper [46]. ²Speculated by the specification of the corresponding MCU. ³Measured by MCUNet paper [8]. ⁴In a private email on Dec. 22, 2022, the authors of the papers replied that the energy consumption should be interpreted as mJ instead of μ J in their papers.

offer lightweight runtime systems for edge devices such as mobile phones, but they are not yet small enough for MCUs.

TABLE III. Performance comparison of various tiny models and inference frameworks on STM32H743, which runs at 480MHz with the resource constraint of 512 KB peak memory and 2 MB storage.

	CMSIS-NN arXiv'18 [41]	X-Cube-AI [42]	TinyEngine NeurIPS'20 [8]	TF-Lite Micro MLSys'21 [47]
Dataset: VWW; Model: mcunet-vww0 ; Input Resolution: 64; Width Multiplier: N/A; Top-1 Accuracy: 87.3%				
Latency	53 ms	32 ms	27 ms	587 ms
Peak Memory	163 KB	88 KB	59 KB	163 KB
Storage usage	646 KB	463 KB	453 KB	627 KB
Dataset: VWW; Model: mcunet-vww1 ; Input Resolution: 80; Width Multiplier: N/A; Top-1 Accuracy: 88.9%				
Latency	97 ms	57 ms	51 ms	1120 ms
Peak Memory	220 KB	113 KB	92 KB	220 KB
Storage usage	736 KB	534 KB	521 KB	718 KB
Dataset: VWW; Model: mcunet-vww2 ; Input Resolution: 144; Width Multiplier: N/A; Top-1 Accuracy: 91.8%				
Latency	478 ms	269 ms	234 ms	5310 ms
Peak Memory	390 KB	201 KB	174 KB	385 KB
Storage usage	1034 KB	774 KB	741 KB	1016 KB
Dataset: ImageNet; Model: mcunet-in0 ; Input Resolution: 48; Width Multiplier: N/A; Top-1 Accuracy: 40.4%				
Latency	51 ms	35 ms	25 ms	596 ms
Peak Memory	161 KB	69 KB	49 KB	161 KB
Storage usage	1090 KB	856 KB	842 KB	1072 KB
Dataset: ImageNet; Model: mcunet-in1 ; Input Resolution: 96; Width Multiplier: N/A; Top-1 Accuracy: 49.9%				
Latency	103 ms	63 ms	56 ms	1227 ms
Peak Memory	219 KB	106 KB	96 KB	219 KB
Storage usage	956 KB	737 KB	727 KB	937 KB
Dataset: ImageNet; Model: mcunet-in2 ; Input Resolution: 160; Width Multiplier: N/A; Top-1 Accuracy: 60.3%				
Latency	642 ms	351 ms	280 ms	6463 ms
Peak Memory	469 KB	238 KB	215 KB	460 KB
Storage usage	1102 KB	849 KB	830 KB	1084 KB
Dataset: ImageNet; Model: mcunet-in3 ; Input Resolution: 176; Width Multiplier: N/A; Top-1 Accuracy: 61.8%				
Latency	770 ms	414 ms	336 ms	7821 ms
Peak Memory	493 KB	243 KB	260 KB	493 KB
Storage usage	1106 KB	867 KB	835 KB	1091 KB
Dataset: ImageNet; Model: mcunet-in4 ; Input Resolution: 160; Width Multiplier: N/A; Top-1 Accuracy: 68.0%				
Latency	OOM	516 ms	463 ms	OOM
Peak Memory	OOM	342 KB	416 KB	OOM
Storage usage	OOM	1843 KB	1825 KB	OOM
Dataset: ImageNet; Model: proxyless-w0.3; Input Resolution: 64; Width Multiplier: 0.3; Top-1 Accuracy: 37.0%				
Latency	54 ms	35 ms	23 ms	512 ms
Peak Memory	136 KB	97 KB	35 KB	128 KB
Storage usage	1084 KB	865 KB	777 KB	1065 KB
Dataset: ImageNet; Model: proxyless-w0.3; Input Resolution: 176; Width Multiplier: 0.3; Top-1 Accuracy: 56.2%				
Latency	380 ms	205 ms	176 ms	3801 ms
Peak Memory	453 KB	221 KB	259 KB	453 KB
Storage usage	1084 KB	865 KB	779 KB	1065 KB
Dataset: ImageNet; Model: mbv2-w0.3; Input Resolution: 64; Width Multiplier: 0.3; Top-1 Accuracy: 34.1%				
Latency	43 ms	29 ms	23 ms	467 ms
Peak Memory	173 KB	88 KB	61 KB	173 KB
Storage usage	959 KB	768 KB	690 KB	940 KB

¹All the inference frameworks used in this measurement are the latest versions as of Dec. 19, 2022. ²The measurement of X-Cube-AI (v7.3.0) is with the default compilation setting, *i.e.*, balanced optimization. ³OOM denotes Out Of Memory. ⁴All the models are available on: <https://github.com/mit-han-lab/mcunet>.

These frameworks cannot accommodate IoT devices and MCUs with limited memory.

CMSIS-NN implements optimized kernels to increase inference speed, minimize memory footprint, and enhance the energy

efficiency of deep learning models on ARM Cortex-M processors [41]. X-Cube-AI, designed by STMicroelectronics, enables the automatic conversion of pre-trained deep learning models to run on STM MCUs with optimized kernel libraries [42]. TVM [93] and AutoTVM [99] also supports microcontrollers (referred to as μ TVM/microTVM [43]). Compilation techniques can also be employed to reduce memory requirements. For instance, Stoutchinin et al. propose to improve deep learning performance on MCU by optimizing the convolution loop nest [100]. Liberis et al. and Ahn et al. present to reorder the operator executions to minimize peak memory [44, 101], whereas Miao et al. seek to achieve better memory utilization by temporarily swapping data off SRAM [102]. With a similar goal of reducing peak memory, other researchers further propose computing partial spatial regions across multiple layers [103, 104, 105]. Additionally, CMix-NN supports mixed-precision kernel libraries of quantized activation and weight on MCU to reduce memory footprint [46]. TinyEngine, as part of MCUNet, is proposed as a memory-efficient inference engine for expanding the search space and fitting a larger model [8]. TinyEngine transfers the majority of operations from runtime to compile time before generating only the code that will be executed by the TinyNAS model. In addition, TinyEngine adapts memory scheduling to the overall network topology as opposed to layer-by-layer optimization. TensorFlow-Lite Micro (TF-Lite Micro) is among the first deep-learning frameworks to support bare-metal microcontrollers in order to enable deep-learning inference on MCUs with tight memory constraints [47]. However, the aforementioned frameworks only support per-layer inference, which limits the model capacity that can be executed with only a small amount of memory and makes higher-resolution input impossible. Hence, MCUNetV2 proposes a generic patch-by-patch inference scheduling, which operates on a small spatial region of the feature map and drastically reduces peak memory usage, and thus makes the inference with high-resolution input on MCUs feasible [9]. TinyOps combines fast internal memory with an additional slow external memory through Direct Memory Access (DMA) peripheral to enlarge memory size and speed up inference [49]. TinyMaix, similar to CMSIS-NN, is an optimized inference kernel library, but it eschews new but rare features and seeks to preserve the readability and simplicity of the codebase [50].

B. Recent Progress on TinyML Training

On-device training on small devices is gaining popularity, as it enables machine learning models to be trained and refined directly on small and low-power devices. On-device training offers several benefits, including the provision of personalized services and the protection of user privacy, as user data is never transmitted to the cloud. However, on-device training presents additional challenges compared to on-device inference, due to larger memory footprints and increased computing operations needed to store intermediate activations and gradients.

Researchers have been investigating ways to reduce the memory footprint of training deep learning models. One kind of approach is to design lightweight network structures manually or by utilizing NAS [85, 106, 107]. Another common approach is to trade computation for memory efficiency, such as freeing

up activation during inference and recomputing discarded activation during the backward propagation [108, 109]. However, such an approach comes at the expense of increased computation time, which is not affordable for tiny devices with limited computation resources. Another approach is layer-wise training, which can also reduce the memory footprint compared to end-to-end training. However, it is not as effective at achieving high levels of accuracy [110]. Another approach reduces the memory footprint by building a dynamic and sparse computation graph for training by activation pruning [111]. Some researchers propose different optimizers [112]. Quantization is also a common approach that reduces the size of activation during training by reducing the bitwidth of training activation [113, 114].

Due to limited data and computational resources, on-device training usually focuses on transfer learning. In transfer learning, a neural network is first pre-trained on a large-scale dataset, such as ImageNet [115], and used as a feature extractor [116, 117, 118]. Then, only the last layer needs to be fine-tuned on a smaller, task-specific dataset [119, 120, 121, 122]. This approach reduces the memory footprint by eliminating the need to store intermediate activations during training, but due to the limited capacity, the accuracy can be poor when the domain shift is large [52]. Fine-tuning all layers can achieve better accuracy but requires large memory to store activation, which is not affordable for tiny devices [117, 116]. Recently, several memory-friendly on-device training frameworks were proposed [123, 124, 125], but these frameworks targeted larger edge devices (i.e., mobile devices) and cannot be adopted on MCUs. An alternative approach is only updating the parameters of batch normalization layers [126, 127]. This reduces the number of trainable parameters, which however does not translate to memory efficiency [52] because the intermediate activation of batch normalization layers still needs to be stored in the memory.

It has been shown that the activation of a neural network is the main factor limiting the ability to train on small devices. Tiny-Transfer-Learning (TinyTL) addresses this issue by freezing the weights of the network and only fine-tuning the biases, which allows intermediate activations to be discarded during backward propagation, reducing peak memory usage [52]. TinyOL trains only the weights of the final layer, allowing for weight training while keeping the activation small enough to fit on small devices [53]. This enables incremental on-device streaming of data for training. However, fine-tuning only the biases or the last layer may not provide sufficient precision. To train more layers on devices with limited memory, POET (Private Optimal Energy Training) [54] introduces two techniques: rematerialization, which frees up activations early at the cost of recomputation, and paging, which allows activations to be transferred to secondary storage. POET uses an integer linear program to find the energy-optimal schedule for on-device training. To further reduce the memory required to store trained weights, MiniLearn applies quantization and dequantization techniques to store the weights and intermediate output in integer precision and dequantizes them to floating-point precision during training [55]. When deployed on tiny devices, deep learning models are often quantized to reduce

TinyML Techniques

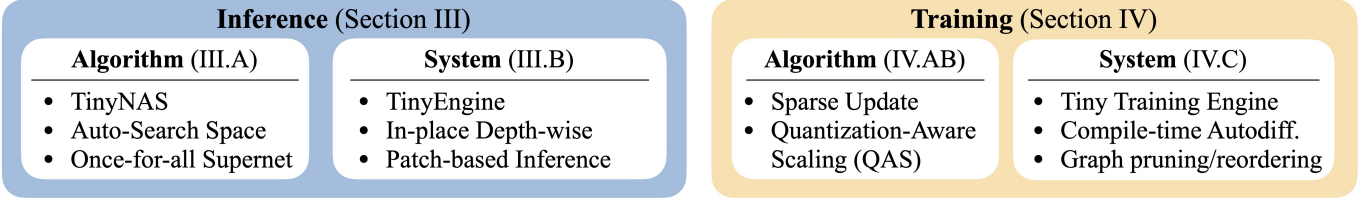


Fig. 3. Techniques specifically designed for tiny devices. In order to fully leverage the limited available resources, we need to take careful consideration of both the system and the algorithm. The co-design approach not only enables practical AI applications on a wide range of IoT platforms (*inference*), but also allows AI to continuously learn over time, adapting to a world that is changing fast (*training*).

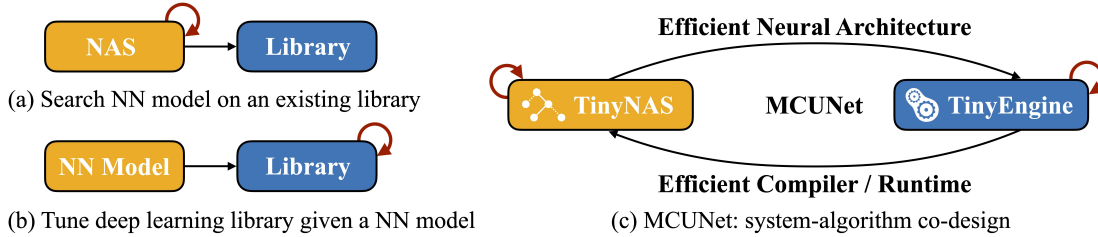


Fig. 4. MCUNet jointly designs the neural architecture and the inference scheduling to fit the tight memory resource on microcontrollers. TinyEngine makes full use of the limited resources on MCU, allowing a larger design space for architecture search. With a larger degree of design freedom, TinyNAS is more likely to find a high accuracy model compared to using existing frameworks.

the memory usage of parameters and activations. However, even after quantization, the parameters may still be too large to fit in the limited hardware resources, preventing full back-propagation. To address these challenges, MCUNetV3 proposes an algorithm-system co-design approach [56]. The algorithm part includes Quantization-Aware Scaling (QAS) and the sparse update. QAS calibrates the gradient scales and stabilizes 8-bit quantized training, while the sparse update skips the gradient computation of less important layers and sub-tensors. The system part includes the Tiny Training Engine (TTE), which has been developed to support both QAS and the sparse update, enabling on-device learning on microcontrollers with limited memory, such as those with 256KB or even less.

III. TINY INFERENCE

In this section, we discuss our recent work, MCUNet family [8, 9], a system-algorithm co-design framework that jointly optimizes the NN architecture (TinyNAS) and the inference scheduling (TinyEngine) in the same loop (Figure 4). Compared to traditional methods that either (a) optimize the neural network using neural architecture search based on a given deep learning library (*e.g.*, TensorFlow, PyTorch) [86, 85, 87], or (b) tune the library to maximize the inference speed for a given network [93, 99], MCUNet can better utilize the resources by system-algorithm co-design, enabling a better performance on microcontrollers. The design space of the inference part is listed in Figure 3 (left).

A. TinyNAS: Automated Tiny Model Design

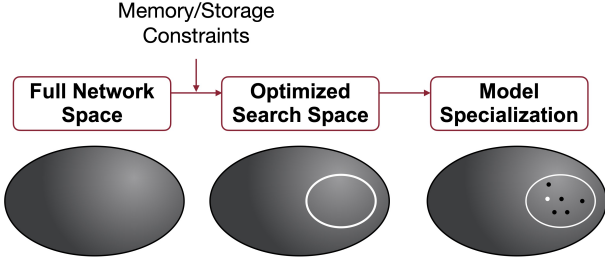
TinyNAS is a two-stage neural architecture search method that first optimizes the search space to fit the tiny and diverse resource constraints, and then performs neural architecture

search within the optimized space. By optimizing the search space, it significantly improves the accuracy of the final model.

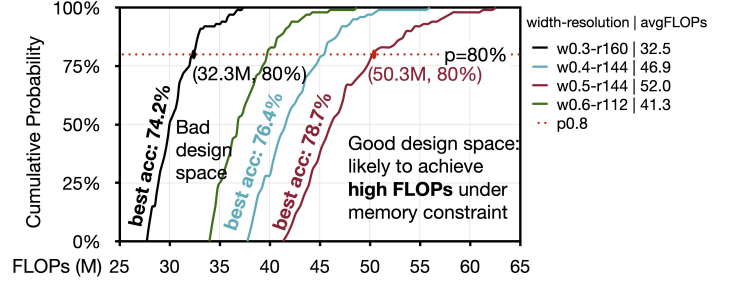
1) Automated search space optimization.

TinyNAS proposes 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, TinyNAS scales the *input resolution* and the *width multiplier* of the mobile search space [86]. It chooses from an input resolution spanning $R = \{48, 64, 80, \dots, 192, 208, 224\}$ and a width multiplier $W = \{0.2, 0.3, 0.4, \dots, 1.0\}$ 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. The 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, TinyNAS evaluates 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* [88], which is computationally heavy due to tremendous training, it only collects the CDF of *FLOPs* (see Figure 5(b)). The intuition is that, within the same model family, the accuracy is usually positively related to the computation [128, 61]. A model with larger computation has a larger capacity, which is more likely to achieve higher accuracy.



(a) TinyNAS: two-stage neural architecture search



(b) Search space optimization by analyzing FLOPs CDF

Fig. 5. (a) TinyNAS is a two-stage neural architecture search method. It first specifies a sub-space according to the constraints, and then performs model specialization. (b) 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.

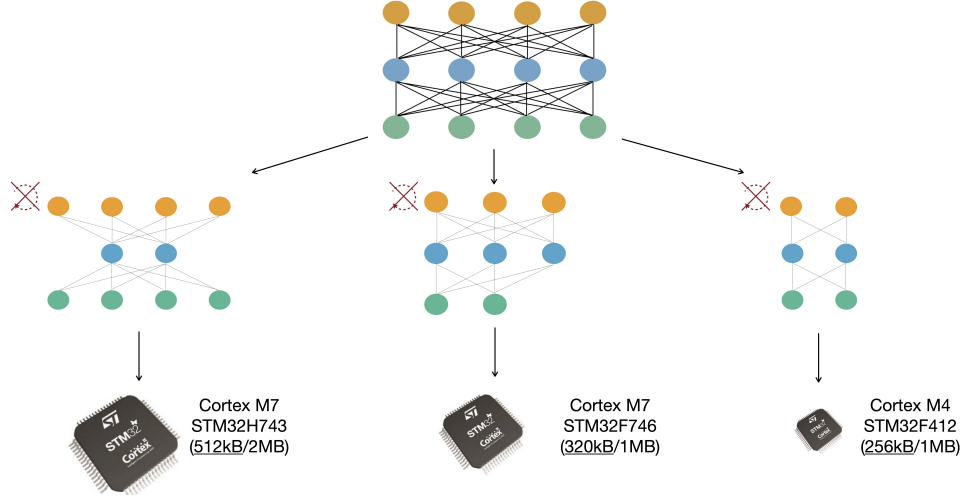


Fig. 6. Once-For-All [129] trains one single super network that supports a wide range of sub-networks through weight sharing, and specializes different sub-network architectures for different MCU hardware.

Take the study of the best search space for ImageNet-100 (a 100-class classification task taken from the original ImageNet) on STM32F746 as an example. We show the FLOPs distribution CDF of the top-10 search space configurations in Figure 5(b). Only the models that satisfy the memory requirement at the best scheduling from TinyEngine are kept. 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.

2) Resource-constrained model specialization with Once-For-All NAS.

To specialize network architecture for various microcontrollers, we need to keep a low neural architecture search cost. Given an optimized search space, TinyNAS further performs one-shot neural architecture search [130, 131] to efficiently find a good model. Specifically, it follows Once-For-All (OFA) NAS [129] to perform network specialization (Figure 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. The search space is based

on the widely-used mobile search space [86, 85, 87, 129] 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 number of possible sub-networks that TinyNAS can cover in the search space is large: 2×10^{19} . For each batch of data, it randomly samples 4 sub-networks, calculates the loss, backpropagates the gradients for each sub-network, and updates the corresponding weights. It then performs an evolution search to find the best model within the search space that meets the onboard resource constraints while achieving the highest accuracy. For each sampled network, it uses TinyEngine to optimize the memory scheduling to measure the optimal memory usage. With such a kind of co-design, we can efficiently fit the tiny memory budget.

B. 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, a good inference framework will make full use of the limited resources in MCUs, avoiding waste of memory, and allowing

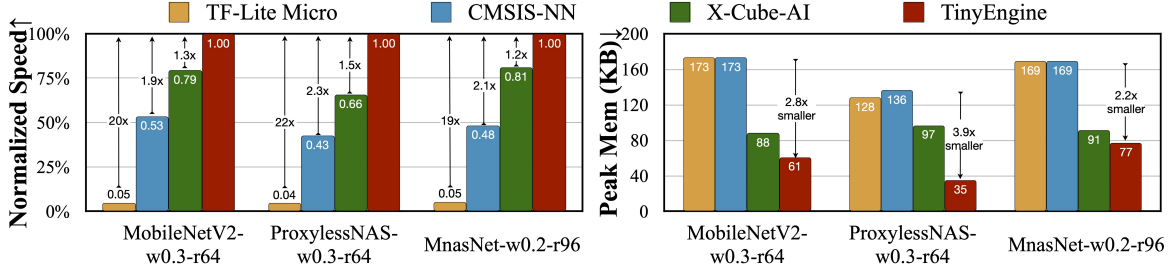


Fig. 7. TinyEngine achieves higher inference efficiency than existing inference frameworks while reducing memory usage. **Left:** TinyEngine is up to 22 \times , 2.3 \times , and 1.5 \times faster than TF-Lite Micro, CMSIS-NN, and X-Cube-AI, respectively. **Right:** By reducing the memory usage, TinyEngine can run various model designs with tiny memory, enlarging the design space for TinyNAS under the limited memory of MCU.

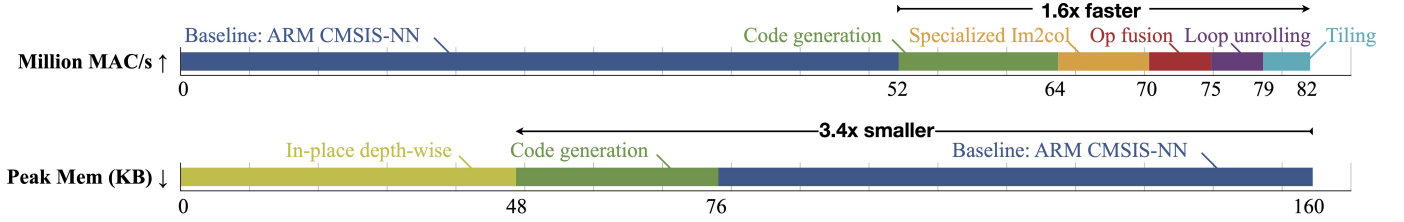


Fig. 8. TinyEngine outperforms existing libraries by eliminating runtime overheads, specializing each optimization technique, and adopting in-place depth-wise convolution. This effectively enlarges design space for TinyNAS under a given latency/memory constraint.

a 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.

1) 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 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. This 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 \times and improve inference efficiency by 22% via code generation, as shown in Figures 7 and 8.

The binary size of TinyEngine is lightweight, making it very memory-efficient for MCUs. The model directly compiled by well-known programming languages for deep learning (e.g., Python, Cython, etc.) cannot be run on MCUs as the size of their dependencies and packages are already larger than the Flash size of MCUs, let alone the size of the compiled model. Besides, 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. That is, the reduction of binary size of the model

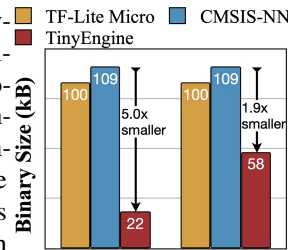


Fig. 9. Binary size.

compiled by TinyEngine comes from not only the benefit of compilation over interpretation but also the model-specific optimization/specialization. As shown in Figure 9, such model-adaptive compilation reduces code size by up to 4.5 \times and 5.0 \times compared to TF-Lite Micro and CMSIS-NN, respectively.

2) In-place depth-wise convolution

TinyEngine supports *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 10. This method reduces the measured memory usage by 1.6 \times as shown in Figure 8.

3) Patched-based Inference

TinyNAS and TinyEngine have significantly reduced the peak memory at the same level of accuracy. But we still notice a very imbalanced peak memory usage per block.

Imbalanced memory distribution. As an example, the per-block peak memory usage of MobileNetV2 [4] is shown in Figure 11. The profiling is done in `int8`. There is a clear pattern of *imbalanced memory usage distribution*. The first 5 blocks have large peak memory, exceeding the memory constraints of MCUs, while the remaining 13 blocks easily fit 256KB memory constraints. The third block has 8 \times larger memory usage than the rest of the network, becoming the memory bottleneck. There are similar patterns for other efficient network designs, which is quite common across different CNN backbones, even for models specialized for memory-limited microcontrollers [8].

The phenomenon applies to most single-branch or residual

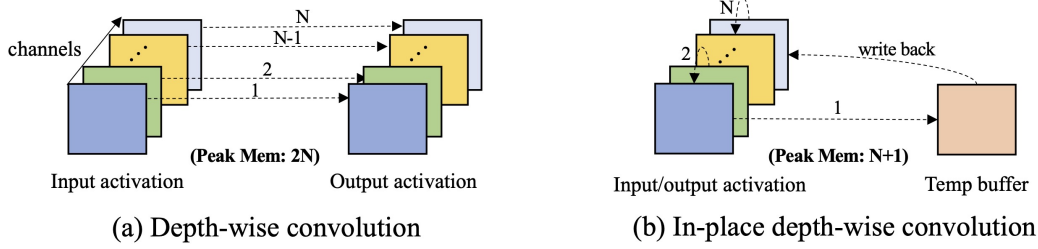


Fig. 10. TinyEngine reduces peak memory by performing in-place depth-wise convolution. **Left:** Conventional depth-wise convolution requires $2N$ memory footprint for activations. **Right:** in-place depth-wise convolution reduces the memory of depth-wise convolutions to $N+1$. Specifically, the output 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.

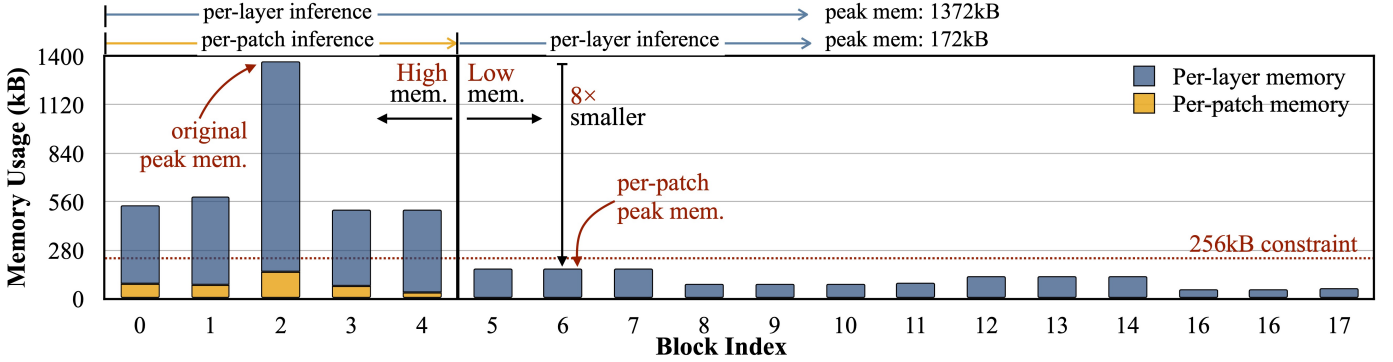


Fig. 11. MobileNetV2 [4] has a very *imbalanced memory usage distribution*. The peak memory is determined by the first 5 blocks with high peak memory, while the later blocks all share a small memory usage. By using per-patch inference (4×4 patches), we are able to significantly reduce the memory usage of the first 5 blocks, and reduce the overall peak memory by 8×, fitting MCUs with a 256kB memory budget. Notice that the model architecture and accuracy are not changed for the two settings. The memory usage is measured in `int8`.

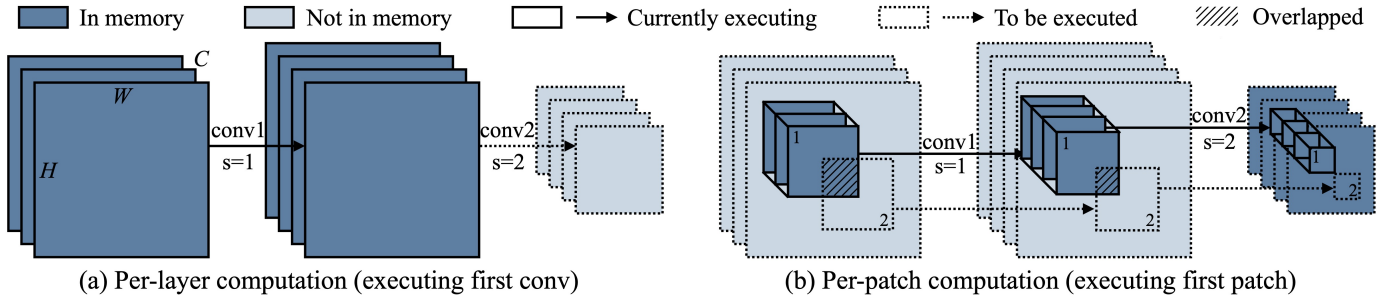


Fig. 12. Per-patch inference can significantly reduce the peak memory required to execute a sequence of convolutional layers. We study two convolutional layers (stride 1 and 2). Under per-layer computation (a), the first convolution has a large input/output activation size, dominating the peak memory 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 (the first input is the image, which can be partially decoded from a compressed format like JPEG).

CNN designs 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 $2\times$ [132, 10, 5] or by an even smaller ratio [4, 133, 106], resulting in a decreasing activation size. Therefore, the memory bottleneck tends to appear at the early stage of the network, after which the peak memory usage is much smaller.

Breaking the Memory Bottleneck with Patch-based Inference. TinEngine breaks the memory bottleneck of the

initial layers with *patch-based inference* (Figure 12). Existing deep learning inference frameworks (e.g., TensorFlow Lite Micro [90], TinyEngine [8], microTVM [93], etc.) use a *layer-by-layer* execution. For each convolutional layer, 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. The patch-based inference runs the initial memory-intensive stage in a *patch-by-patch* manner. For each time, it only runs the model on a small spatial region ($>10\times$ smaller than the whole area), which effectively cuts down the peak memory usage. After this stage is finished, the rest of the network with a small peak memory is executed in a

*some CNN designs have highly complicated branching structure (e.g., NASNet [83]), but they are generally less efficient for inference [6, 86, 85]; thus not widely used for edge computing.

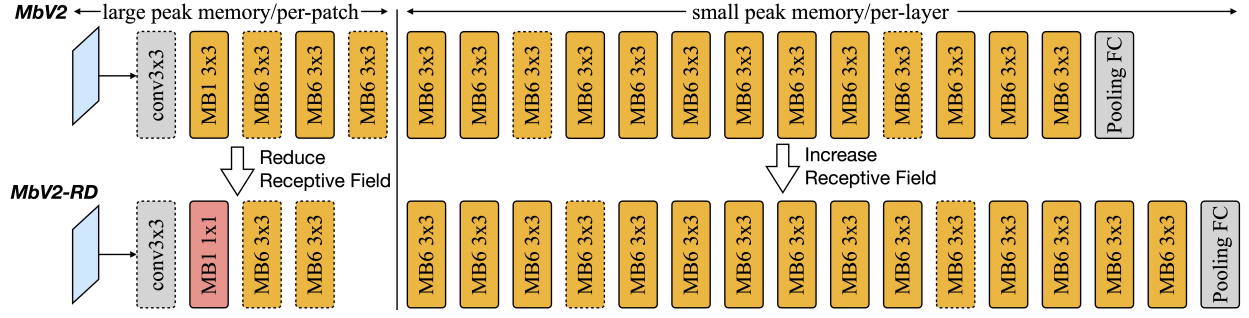


Fig. 13. The redistributed MobileNetV2 (MbV2-RD) has reduced receptive field for the per-patch inference stage and increased receptive field for the per-layer stage. The two networks have the same level of performance, but MbV2-RD has a smaller overhead under patch-based inference. The mobile inverted block is denoted as MB{expansion ratio} {kernel size}. The dashed border means stride=2.

normal layer-by-layer manner (upper notations in Figure 11).

An example of two convolutional layers (with stride 1 and 2) is shown in Figure 12. For conventional per-layer computation, the first convolutional layer has large input and output activation size, leading to a high peak memory. With spatial partial computation, it allocates the buffer for the final output and computes its values *patch-by-patch*. In this manner, it only needs to store the activation from *one patch* instead of the *whole feature map*.

Reducing Computation Overhead by Redistributing the Receptive Field. The significant memory saving comes at the cost of computation overhead. To maintain the same output results as per-layer inference, the non-overlapping output patches correspond to overlapping patches in the input image (the shadow area in Figure 12(b)). This is because convolutional filters with kernel size >1 contribute to increasing receptive fields. The computation overhead is related to the receptive field of the patch-based initial stage. Consider the output of the patch-based stage, the larger receptive field it has on the input image, the larger resolution for each patch, leading to a larger overlapping area and repeated computation. There are some focusing on addressing the issue from the hardware perspective [103]. However, since such practices may not be general to all devices, TinyEngine solves the problem from the network architecture side.

MCUNet proposes to *redistribute* the receptive field (RF) of the CNN to reduce computation overhead. The basic idea is: (1) *reduce the receptive field of the patch-based initial stage*; (2) *increase the receptive field of the later stage*. Reducing RF for the initial stage helps to reduce the size of each input patch and repeated computation. However, some tasks may have degraded performance if the overall RF is smaller (e.g., detecting large objects). Therefore, it further increases the RF of the later stage to compensate for the performance loss. A manually tuned example of MobileNetV2 is shown in Figure 13. After redistributing the receptive field (“MbV2-RD”), the computation overhead is negligible.

MCUNet automates the process with joint search (introduced in the next section).

C. Co-Design: Joint Neural Architecture and Inference Scheduling Search

1) Co-Design Loop

The optimization algorithms for model architectures and inference engines are tightly coupled. For example, redistributing the receptive field allows us to enjoy the benefit of memory reduction at minimal computation/latency overhead, which allows larger freedom when designing the backbone architecture (e.g., we can now use a larger input resolution). To explore such a large design space, MCUNet jointly optimizes the *neural architecture* and the *inference scheduling* in an automated manner. Given a certain dataset and hardware constraints (SRAM limit, Flash limit, latency limit, etc.), our goal is to achieve the highest accuracy while satisfying all the constraints. For the model optimization, it uses NAS to find a good candidate network architecture; for the scheduling optimization, it optimizes the knobs like the patches number p and the number of blocks n to perform patch-based inference, and other knobs in TinyEngine [8].

There are some trade-offs during the co-design. For example, given the same constraints, it can choose to use a smaller model that fits per-layer execution ($p = 1$, no computation overhead), or a larger model and per-patch inference ($p > 1$, with a small computation overhead). Therefore, MCUNet puts both sides in the same loop and uses evolutionary search to find the best set of $(k_{\lfloor \cdot \rfloor}, e_{\lfloor \cdot \rfloor}, d_{\lfloor \cdot \rfloor}, w_{\lfloor \cdot \rfloor}, r, p, n)$ satisfying constraints. Therefore, The two dimensions are jointly searched in the same loop with evolutionary search.

2) Experimental Results

a) Pushing the ImageNet record on MCUs.

With joint optimization of neural architecture and inference scheduling, MCUNet significantly pushes the state-of-the-art results for MCU-based tiny image classification.

We compared MCUNet with existing state-of-the-art solutions on ImageNet classification under two hardware settings: 256kB SRAM/1MB Flash and 512kB SRAM/2MB Flash. The former represents a widely used Cortex-M4 microcontroller; the latter corresponds to a higher-end Cortex-M7. The goal is to achieve the highest ImageNet accuracy on resource-constrained MCUs (Table IV). MCUNet significantly improves the ImageNet accuracy of tiny deep learning on microcontrollers. Under

TABLE IV. MCUNet significantly improves the ImageNet accuracy on microcontrollers, outperforming the state-of-the-arts by **4.6%** under 256kB SRAM and **3.3%** under 512kB. Lower or mixed precisions (marked gray) are orthogonal techniques, which we leave for future work. Out-of-memory (OOM) results are ~~struck out~~.

Model / Library	Quant.	MACs	SRAM	Flash	Top-1	Top-5
<i>STM32F412 (256kB SRAM, 1MB Flash)</i>						
MbV2 $0.35\times$ ($r=144$) [4] / TinyEngine [8]	int8	24M	308kB	862kB	49.0%	73.8%
Proxyless $0.3\times$ ($r=176$) [85] / TinyEngine [8]	int8	38M	292kB	892kB	56.2%	79.7%
MbV1 $0.5\times$ ($r=192$) [5] / Rusci <i>et al.</i> [45]	mixed	110M	<256kB	<1MB	60.2%	-
MCUNet (TinyNAS / TinyEngine) [8]	int8	68M	238kB	1007kB	60.3%	-
MCUNet (TinyNAS / TinyEngine) [8]	int4	134M	233kB	1008kB	62.0%	-
MCUNet-M4 (w/ patch)	int8	119M	196kB	1010kB	64.9%	86.2%
<i>STM32H743 (512kB SRAM, 2MB Flash)</i>						
MbV1 $0.75\times$ ($r=224$) [5] / Rusci <i>et al.</i> [45]	mixed	317M	<512kB	<2MB	68.0%	-
MCUNet (TinyNAS / TinyEngine) [8]	int8	126M	452kB	2014kB	68.5%	-
MCUNet (TinyNAS / TinyEngine) [8]	int4	474M	498kB	2000kB	70.7%	-
MCUNet-H7 (w/ patch)	int8	256M	465kB	2032kB	71.8%	90.7%

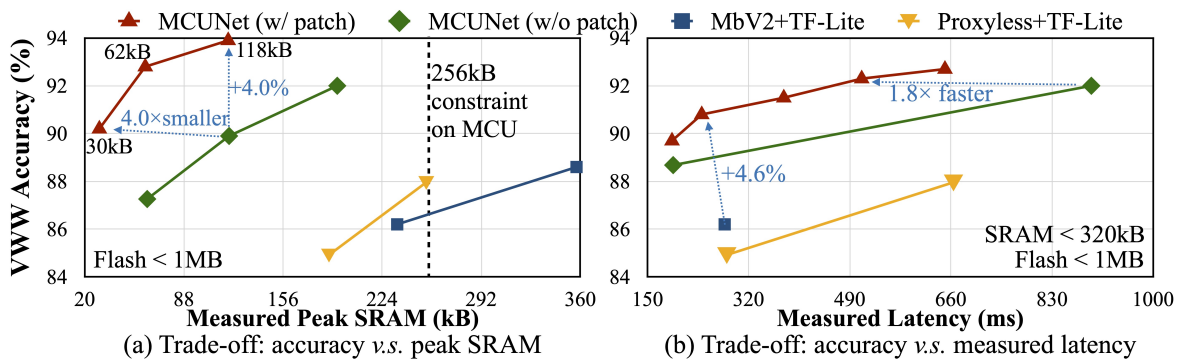


Fig. 14. **Left:** MCUNet has better visual wake word (VWW) accuracy vs. peak SRAM trade-off. Compared to MCUNet [8], MCUNet achieves better accuracy at 4.0 \times smaller peak memory. It achieves >90% accuracy under <32kB memory, facilitating deployment on extremely small hardware. **Right:** patch-based method expands the search space that can fit the MCU, allowing better accuracy vs. latency trade-off.

256kB SRAM/1MB Flash, MCUNet outperforms the state-of-the-art method [8] by 4.6% at 18% lower peak SRAM. Under 512kB SRAM/2MB Flash, MCUNet achieves a new *record* ImageNet accuracy of 71.8% on commercial microcontrollers, which is 3.3% compared to the best solution under the same quantization policy. Lower-bit (int4) or mixed-precision quantization can further improve the accuracy (marked in gray in the table).

b) Visual Wake Words under 32kB SRAM.

Visual wake word (VWW) reflects the low-energy application of TinyML. MCUNet allows running a VWW model with a modest memory requirement. As in Figure 14[†], MCUNet outperforms state-of-the-art method [8] for both accuracy vs. peak memory and accuracy vs. latency trade-off. Compared to per-layer inference, MCUNet can achieve better accuracy using 4.0 \times smaller memory. Actually, it can achieve >90% accuracy under 32kB SRAM requirement, allowing model deployment on low-end MCUs like STM32F410 costing only \$1.6. Per-patch inference also expands the search space, giving us more freedom to find models with better accuracy vs. latency trade-off.

[†]Note that MCUNetV2 refers to the version w/ patch-based inference, while MCUNet refers to per-layer inference.

c) MCU-based detection on Pascal VOC.

Object detection is sensitive to a smaller input resolution [9]. Current state-of-the-art [8] cannot achieve a decent detection performance on MCUs due to the resolution bottleneck. MCUNet breaks the memory bottleneck for detectors and improves the mAP by double digits.

The object detection results on Pascal VOC trained with YOLOv3 [135] are shown in Table V, including results for M4 MCU with 256kB SRAM and H7 MCU with 512kB SRAM. On H7 MCU, MCUNet-H7 improves the mAP by 16.7% compared to the state-of-the-art method MCUNet [8]. It can also scale down to fit a cheaper commodity Cortex-M4 MCU with only 256kB SRAM, while still improving the mAP by 13.2% at 1.9 \times smaller peak SRAM. Note that MCUNet-M4 shares a similar computation with MCUNet (172M vs. 168M) but a much better mAP. This is because the expanded search space from patch-based inference allows us to choose a better configuration of larger input resolution and smaller models.

d) Memory-efficient face detection.

The performance of MCUNet for memory-efficient face detection on WIDER FACE [136] dataset are shown in Table VI. The analytic memory usage of the detector backbone in fp32 is reported following [105]. The models are trained with S3FD

TABLE V. MCUNet significantly improves Pascal VOC [134] object detection on MCU by allowing a higher input resolution. Under STM32H743 MCU constraints, MCUNet-H7 improves the mAP by 16.9% compared to [8], achieving a record performance on MCU. It can also scale down to cheaper MCU STM32F412 with only 256kB SRAM while still improving mAP by 13.2% at $1.9\times$ smaller peak SRAM and a similar computation.

MCU Model	Constraint	Model	#Param	MACs	peak SRAM	VOC mAP	Gain
H743 ($\sim \$7$)	SRAM $<512\text{kB}$	MbV2+CMSIS [8]	0.87M	34M	519kB	31.6%	-
		MCUNet [8]	1.20M	168M	466kB	51.4%	0%
		MCUNet-H7	0.67M	343M	438kB	68.3%	+16.9%
F412 ($\sim \$4$)	$<256\text{kB}$	MCUNet-M4	0.47M	172M	247kB	64.6%	+13.2%

TABLE VI. MCUNet outperforms existing methods for memory-efficient face detection on WIDER FACE [136] dataset. Compared to RNNPool-Face-C [105], MCUNet-L can achieve similar mAP at $3.4\times$ smaller peak SRAM and $1.6\times$ smaller computation. The model statistics are profiled on 640×480 RGB input images following [105].

Method	MACs \downarrow	Peak Memory \downarrow (fp32)	mAP \uparrow			mAP (≤ 3 faces) \uparrow		
			Easy	Medium	Hard	Easy	Medium	Hard
EXTD [137]	8.49G	18.8MB ($9.9\times$)	0.90	0.88	0.82	0.93	0.93	0.91
LFFD [138]	9.25G	18.8MB ($9.9\times$)	0.91	0.88	0.77	0.83	0.83	0.82
RNNPool-Face-C [105]	1.80G	6.44MB ($3.4\times$)	0.92	0.89	0.70	0.95	0.94	0.92
MCUNet-L	1.10G	1.89MB ($1.0\times$)	0.92	0.90	0.70	0.94	0.93	0.92
EagleEye [139]	0.08G	1.17MB ($1.8\times$)	0.74	0.70	0.44	0.79	0.78	0.75
RNNPool-Face-A [105]	0.10G	1.17MB ($1.8\times$)	0.77	0.75	0.53	0.81	0.79	0.77
MCUNet-S	0.11G	672kB ($1.0\times$)	0.85	0.81	0.55	0.90	0.89	0.87

face detector [140] following [105] for a fair comparison. The reported mAP is calculated on samples with ≤ 3 faces, which is a more realistic setting for tiny devices. MCUNet outperforms existing solutions under different scales. MCUNet-L achieves comparable performance at $3.4\times$ smaller peak memory compared to RNNPool-Face-C [8] and $9.9\times$ smaller peak memory compared to LFFD [138]. The computation is also $1.6\times$ and $8.4\times$ smaller. MCUNet-S consistently outperforms RNNPool-Face-A [105] and EagleEye [139] at $1.8\times$ smaller peak memory.

IV. TINY TRAINING

In addition to inference, tiny on-device training is a growing direction that allows us to *adapt* the pre-trained model to newly collected sensory data *after* deployment. By training and adapting *locally* on the edge, the model can learn to continuously improve its predictions and perform lifelong learning and user customization. By bringing training closer to the sensors, it also helps to protect user privacy when handling sensitive data (e.g., healthcare).

However, on-device training on tiny edge devices is extremely challenging and fundamentally different from cloud training. Tiny IoT devices (e.g., microcontrollers) typically have a limited SRAM size like 256KB. Such a small memory budget is hardly enough for the *inference* of deep learning models [8, 9, 48, 141, 142, 143, 44, 45], let alone the *training*, which requires extra computation for the backward and extra memory for intermediate activation [109]. On the other hand, modern deep training frameworks (e.g., PyTorch [89], TensorFlow [144]) are usually designed for cloud servers and require a large memory footprint ($>300\text{MB}$) even when training a small model (e.g., MobileNetV2-w0.35 [4]) with batch size 1. The huge gap ($>1000\times$) makes it impossible to run on tiny IoT

devices. Furthermore, devices like microcontrollers are bare-metal and do not have an operational system and the runtime support needed by existing training frameworks. Therefore, we need to jointly design the *algorithm* and the *system* to enable tiny on-device training.

Deep learning training systems such as PyTorch [89], TensorFlow [144], JAX [92], MXNet [91], *etc.* do not consider the tight resources on edge devices. Edge deep learning inference frameworks like TVM [93], TF-Lite [94], NCNN [97], *etc.* provide a slim runtime, but lack the support for back-propagation. There are low-cost efficient transfer learning algorithms like training only the final classifier layer, bias-only update [52], *etc.* However, the existing training systems can not realize the theoretical saving into measured savings. The downstream accuracy of such update schemes is also low (Figure 18). There is a need for training systems that can effectively utilize the limited resources on edge devices."

In order to bridge the gap and enable tiny on-device training with algorithm-system co-design, there are two unique challenges in tiny on-device training: (1) the model is quantized on edge devices. A *real* quantized graph is difficult to optimize due to mixed-precision tensors and the lack of Batch Normalization layers [145]; (2) the limited hardware resource (memory and computation) of tiny hardware does not allow full back-propagation, as the memory usage can easily exceed the SRAM of microcontrollers by more than an order of magnitude. To cope with the difficulties, TinyTraining proposes the following designs

A. Quantization Aware Scaling

Neural networks usually need to be quantized to fit the limited memory of edge devices [8, 146]. For a fp32 linear layer $\mathbf{y}_{\text{fp32}} = \mathbf{W}_{\text{fp32}}\mathbf{x}_{\text{fp32}} + \mathbf{b}_{\text{fp32}}$, the int8 quantized

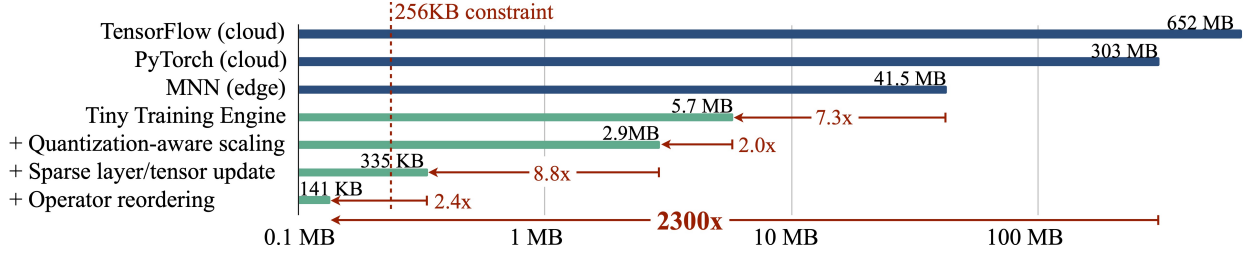


Fig. 15. Algorithm and system co-design reduces the training memory from 303MB (PyTorch) to 149KB with the same transfer learning accuracy, leading to 2077 \times reduction. The numbers are measured with MobilenetV2-w0.35 [4], batch size 1 and resolution 128 \times 128. It can be deployed to a microcontroller with 256KB SRAM.

counterpart is:

$$\bar{\mathbf{y}}_{\text{int8}} = \text{cast2int8}[s_{\text{fp32}} \cdot (\bar{\mathbf{W}}_{\text{int8}} \bar{\mathbf{x}}_{\text{int8}} + \bar{\mathbf{b}}_{\text{int32}})], \quad (1)$$

where $\bar{\cdot}$ denotes the tensor being quantized to fixed-point numbers, and s is a floating-point scaling factor to project the results back into int8 range. The gradient update for the weights can be presented as: $\bar{\mathbf{W}}'_{\text{int8}} = \text{cast2int8}(\bar{\mathbf{W}}_{\text{int8}} - \alpha \cdot \mathbf{G}_{\bar{\mathbf{W}}})$, where α is the learning rate, and $\mathbf{G}_{\bar{\mathbf{W}}}$ is the gradient of the weights. After applying the gradient update, the weights are rounded back to 8-bit integers.

Gradient Scale Mismatch: Unlike fine-tuning floating-point model on the cloud, training with a *real* quantized graph[‡] is difficult: the quantized graph has tensors of different bit-precisions (int8 , int32 , fp32 , shown in Equation 1) and lacks Batch Normalization [145] layers (fused), leading to unstable gradient update.

Optimizing a quantized graph often leads to lower accuracy compared to the floating-point counterpart. A possible hypothesis is that the quantization process distorts the gradient update. To verify the idea, Figure 16 plot the ratio between weight norm and gradient norm (*i.e.*, $\|\mathbf{W}\|/\|\mathbf{G}\|$) for each tensor at the beginning of the training on the CIFAR dataset [147]. The ratio curve is very different after quantization: (1) the ratio is much larger (could be addressed by adjusting the learning rate); (2) the ratio has a different pattern after quantization. Take the highlighted area (red box) as an example, the quantized ratios have a zigzag pattern, differing from the floating-point curve. If

[‡]Note that this is contrary to the *fake* quantization graph, which is widely used in quantization-aware training [146].

all the tensors are updated with a fixed learning rate, then the update speed of each tensor would be very different compared to the floating-point case, leading to inferior accuracy. Even adaptive-learning rate optimizers like Adam [148] cannot fully address the issue, as shown in Table VII.

Hyperparameter-Free Gradient Scaling. To address the problem, a hyper-parameter-free learning rate scaling rule, QAS, is proposed. Consider a 2D weight matrix of a linear layer $\mathbf{W} \in \mathbb{R}^{c_1 \times c_2}$, where c_1, c_2 are the input and output channel. To perform per-tensor quantization[§], a scaling rate $s_{\mathbf{W}} \in \mathbb{R}$ is computed, such that $\bar{\mathbf{W}}$'s largest magnitude is $2^7 - 1 = 127$:

$$\mathbf{W} = s_{\mathbf{W}} \cdot (\mathbf{W}/s_{\mathbf{W}}) \stackrel{\text{quantize}}{\approx} s_{\mathbf{W}} \cdot \bar{\mathbf{W}}, \quad \mathbf{G}_{\bar{\mathbf{W}}} \approx s_{\mathbf{W}} \cdot \mathbf{G}_{\mathbf{W}}, \quad (2)$$

The process (roughly) preserves the mathematical functionality during the forward (Equation 1), but it distorts the magnitude ratio between the weight and its corresponding gradient:

$$\|\bar{\mathbf{W}}\|/\|\mathbf{G}_{\bar{\mathbf{W}}}\| \approx \|\mathbf{W}/s_{\mathbf{W}}\|/\|s_{\mathbf{W}} \cdot \mathbf{G}_{\mathbf{W}}\| = s_{\mathbf{W}}^{-2} \cdot \|\mathbf{W}\|/\|\mathbf{G}_{\mathbf{W}}\|. \quad (3)$$

The weight and gradient ratios are off by $s_{\mathbf{W}}^{-2}$, leading to the distorted pattern in Figure 16: (1) the scaling factor is far smaller than 1, making the weight-gradient ratio much larger; (2) weights and biases have different data type (int8 vs. int32) and thus have scaling factors of very different magnitude, leading to the zigzag pattern. To solve the issue, Quantization-Aware Scaling (QAS) is proposed by compensat-

[§]For simplicity. In practice, per-channel quantization [146] is used and the scaling factor is a vector of size c_2 .

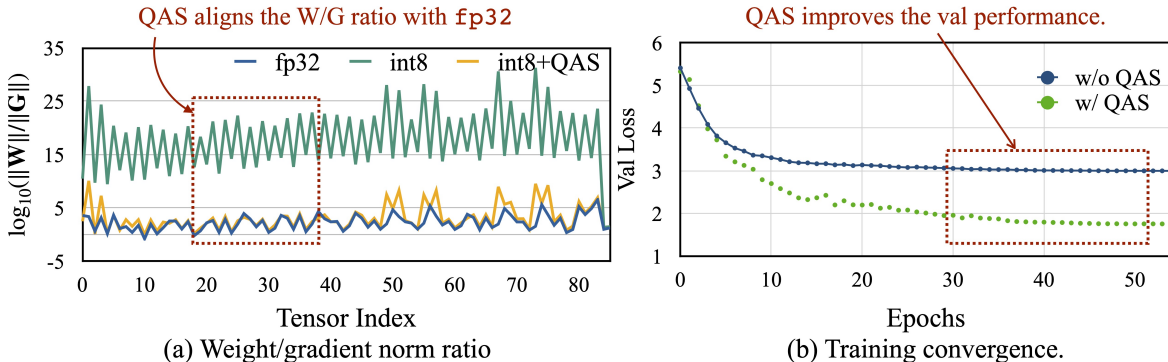


Fig. 16. *Left:* The quantized model has a very different weight/gradient norm ratio (*i.e.*, $\|\mathbf{W}\|/\|\mathbf{G}\|$) compared to the floating-point model at training time. QAS stabilizes the $\|\mathbf{W}\|/\|\mathbf{G}\|$ ratio and helps optimization. *Right:* The validation loss curves w/ and w/o QAS. QAS effectively helps convergence, leading to better accuracy. The results are from updating the last two blocks of the MCUNet model on the Cars dataset.

TABLE VII. Updating real quantized graphs (int8) with SGD is difficult: the transfer learning accuracy falls behind the floating-point counterpart (fp32), even with adaptive learning rate optimizers like Adam [148] and LARS [149]. QAS helps to bridge the accuracy gap without memory overhead (slightly higher). The numbers are for updating the last two blocks of MCUNet-5FPS [8] model.

Precision	Optimizer	Accuracy (%) (MCUNet backbone: 23M MACs, 0.48M Param)								Avg Acc.
		Cars	CF10	CF100	CUB	Flowers	Food	Pets	VWW	
fp32	SGD-M	56.7	86.0	63.4	56.2	88.8	67.1	79.5	88.7	73.3
int8	SGD-M	31.2	75.4	54.5	55.1	84.5	52.5	81.0	85.4	64.9
	Adam [148]	54.0	84.5	61.0	58.5	87.2	62.6	80.1	86.5	71.8
	LARS [149]	5.1	64.8	39.5	9.6	28.8	46.5	39.1	85.0	39.8
	SGD-M+QAS	55.2	86.9	64.6	57.8	89.1	64.4	80.9	89.3	73.5

ing the gradient of the quantized graph according to Equation 3:

$$\tilde{\mathbf{G}}_{\mathbf{W}} = \mathbf{G}_{\mathbf{W}} \cdot s_{\mathbf{W}}^{-2}, \quad \tilde{\mathbf{G}}_{\mathbf{b}} = \mathbf{G}_{\mathbf{b}} \cdot s_{\mathbf{W}}^{-2} \cdot s_{\mathbf{x}}^{-2} = \mathbf{G}_{\mathbf{b}} \cdot s^{-2} \quad (4)$$

where $s_{\mathbf{x}}^{-2}$ is the scaling factor for quantizing input \mathbf{x} (a scalar following [146], note that $s = s_{\mathbf{W}} \cdot s_{\mathbf{x}}$ in Equation 1). $\|\mathbf{W}\|/\|\mathbf{G}\|$ curve with QAS is plotted in Figure 16 (int8+scale). After scaling, the gradient ratios match the floating-point counterpart. It also improves transfer learning accuracy (Table VII), matching the accuracy of the floating-point counterpart without incurring memory overhead.

Experiment Results. The last two blocks in Table VII show the fine-tuning results (simulating low-cost fine-tuning) of MCUNet on various downstream datasets. With momentum SGD, the training accuracy of the quantized model (int8) falls behind the floating-point counterpart due to the difficulty in optimization. Adaptive learning rate optimizers like Adam [148] can improve the accuracy, but it is still lower than the fp32 fine-tuning results. It also consumes 3 times more memory due to second-order momentum, which is not desired in TinyML settings. LARS [149] does not converge well on most datasets despite extensive hyperparameter tuning (of both the learning rate and the "trust coefficient"). The aggressive gradient scaling rule of LARS makes the training unstable. The accuracy gap is closed when applying QAS, achieving the same accuracy as floating-point training with no extra memory cost. Figure 16 shows the training curve of TinyTraining on the Cars dataset with and without QAS. QAS effectively improves optimization.

B. Memory-Efficient Sparse Update

Though QAS makes optimizing a quantized model possible, updating the whole model (or even the last several blocks) requires a large amount of memory, which is not affordable for the TinyML setting. To address this, sparsely updating the layers and the tensors is proposed

Sparse Layer/Tensor Update. Pruning techniques prove to be quite successful for achieving sparsity and reducing model size [63, 61, 59, 58, 60, 62]. Instead of pruning *weights* for inference, the *gradient* during backpropagation, and updating the model sparsely are pruned. Given a tight memory budget, updates of the *less important* parameters are skipped to reduce memory usage and computation cost. When updating a linear layer $\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$ (similar analysis applies to convolutions), the gradient update is $\mathbf{G}_{\mathbf{W}} = f_1(\mathbf{G}_{\mathbf{y}}, \mathbf{x})$ and $\mathbf{G}_{\mathbf{b}} = f_2(\mathbf{G}_{\mathbf{y}})$, given the output gradient $\mathbf{G}_{\mathbf{y}}$ from the later layer. Notice that updating the biases does not require saving the intermediate

activation \mathbf{x} , leading to a lighter memory footprint [52][¶]; while updating the weights is more memory-intensive but also more expressive. For hardware like microcontrollers, an extra copy is needed for the updated parameters since the original ones are stored in read-only FLASH [8]. Given the different natures of updating rules, three aspects of the sparse update rule are considered (Figure 17): (1) *Bias update*: how many layers should be backpropagated to and update the biases (bias update is cheap, the bias terms can be always updated if the layer is backpropagated). (2) *Sparse layer update*: select a subset of layers to update the corresponding weights. (3) *Sparse tensor update*: further allow updating a subset of weight channels to reduce the cost.

However, finding the right sparse update scheme under a memory budget is challenging due to the large combinational space. For MCUNet [8] model with 43 convolutional layers and weight update ratios from $\{0, 1/8, 1/4, 1/2, 1\}$, the combination is about 10^{30} , making exhaustive search impossible.

Automated Selection with Contribution Analysis. *contribution analysis* is proposed to automatically derive the sparse update scheme by counting the contribution of each parameter (weight/bias) to the downstream accuracy. Given a convolutional neural network with l layers, the accuracy improvement is measured from (1) biases: the improvement of updating *last* k biases $\mathbf{b}_l, \mathbf{b}_{l-1}, \dots, \mathbf{b}_{l-k+1}$ (bias-only update) compared to only updating the classifier, defined as $\Delta\text{acc}_{\mathbf{b}[:k]}$; (2) weights: the improvement of updating the weight of one extra layer \mathbf{W}_i (with a channel update ratio r) compared to bias-only update, defined as $\Delta\text{acc}_{\mathbf{W}_i, r}$. An example of the contribution analysis can be found in Figure 18 Left (MCUNet on Cars [150] dataset; After finding $\Delta\text{acc}_{\mathbf{b}[:k]}$ and $\Delta\text{acc}_{\mathbf{W}_i}$ ($1 \leq k, i \leq l$), an optimization problem is solved to find:

$$k^*, \mathbf{i}^*, \mathbf{r}^* = \max_{k, \mathbf{i}, \mathbf{r}} (\Delta\text{acc}_{\mathbf{b}[:k]} + \sum_{i \in \mathbf{i}, r \in \mathbf{r}} \Delta\text{acc}_{\mathbf{W}_i, r}) \quad (5)$$

$$\text{s.t. Memory}(k, \mathbf{i}, \mathbf{r}) \leq \text{constraint},$$

where \mathbf{i} is a collection of layer indices whose weights are updated, and \mathbf{r} is the corresponding update ratios (1/8, 1/4, 1/2, 1). Intuitively, by solving this optimization problem, the combination of (#layers for bias update is found, the subset of weights to update), such that the total contribution is maximized while the memory overhead does not exceed the constraint. The problem can be efficiently solved with the evolutionary search. Sparse update assumes that the accuracy contribution of

[¶]If many layers are updated, the intermediate activation could consume a large memory [109].

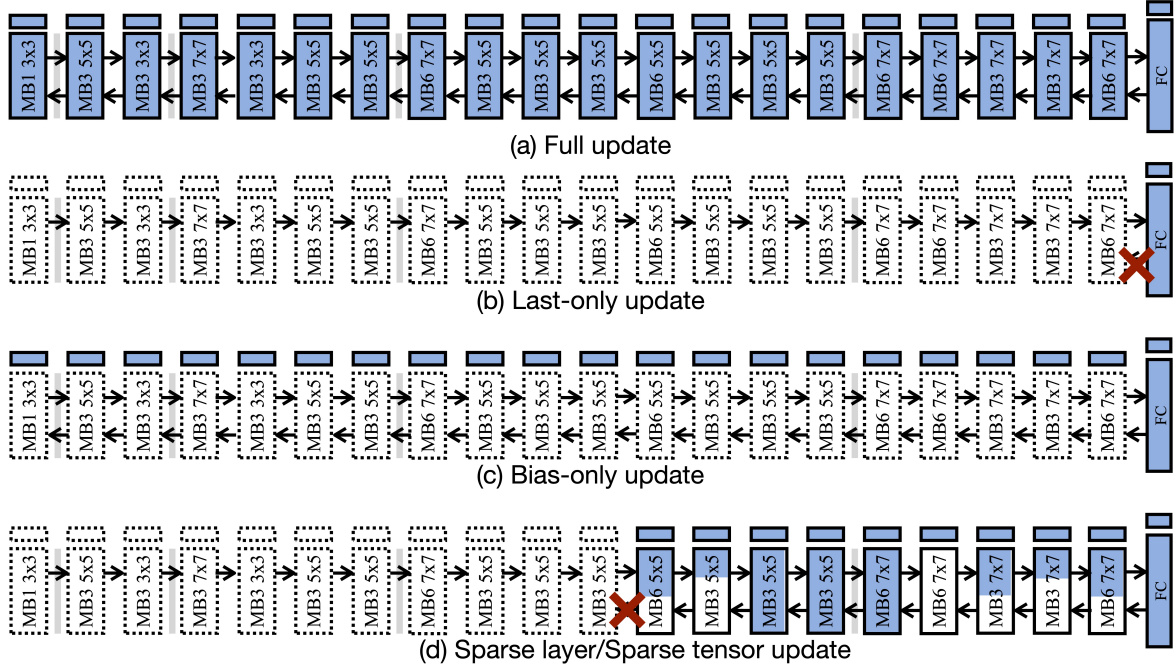


Fig. 17. Different learning schemes on ProxyllessNAS-Mobile [85]. Full update (a) consumes a lot of memory thus cannot fit TinyML. Efficient learning methods like last-only (b) / bias-only (c) save the memory but cannot match the baseline performance. Sparse update (d) only performs partial back-propagation, leading to less memory usage and computation with comparable accuracy on downstream tasks.

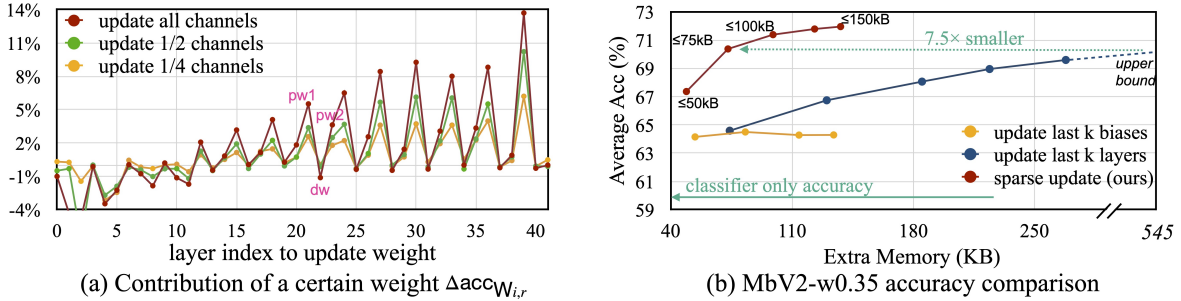


Fig. 18. *Left:* Contribution analysis of updating biases and weights. For updating the weight of a specific layer, the later layers appear to be more important; the first point-wise conv (pw1) in an inverted bottleneck block [4] appears to be more important; and the gains are bigger with more channels updated. *Right:* Sparse update can achieve higher transfer learning accuracy using 4.5-7.5 \times smaller extra memory (analytic) compared to updating the last k layers. For classifier-only update, the accuracy is low due to limited capacity. Bias-only update can achieve a higher accuracy but plateaus soon.

each tensor (Δacc) can be summed up. Such an approximation proves to be quite effective in our experiments.

Sparse Update Obtains Better Accuracy at Lower Memory.

The performance of our searched sparse update schemes is compared to two baseline methods: fine-tuning only the biases of the last k layers and fine-tuning the weights and biases of the last k layers. For each configuration, the average accuracy is measured on 8 downstream datasets, and the extra memory usage is calculated analytically. Figure 18 compares the results with a simple baseline of fine-tuning only the classifier. The accuracy of classifier-only update is low due to the limited learning capacity. Updating only the classifier is not enough; the backbone also needs updates. Bias-only update outperforms classifier-only update, but the accuracy quickly plateaus and does not improve even when more biases are tuned. For updating the last k layers, the accuracy generally improves as more layers are tuned; however, it has a very large memory

footprint. For example, updating the last two blocks of MCUNet leads to an extra memory usage exceeding 256KB, making it infeasible for IoT devices/microcontrollers. Our sparse update scheme can achieve higher downstream accuracy at a much lower memory cost. Compared to updating the last k layers, the sparse update can achieve higher downstream accuracy with 4.5-7.5 times smaller memory overhead. The highest accuracy is achievable by updating the last k layers^{||} as the baseline upper bound (denoted as "upper bound"). Interestingly, our sparse update achieves a better downstream accuracy compared to the baseline best statistics. The sparse update scheme alleviates over-fitting or makes momentum-free optimization easier.

^{||}Note that fine-tuning the entire model does not always lead to the best accuracy. The best k on Cars dataset is obtained via grid search: $k=36$ for MobileNetV2, 39 for ProxyllessNAS, 12 for MCUNet, and apply it to all datasets.

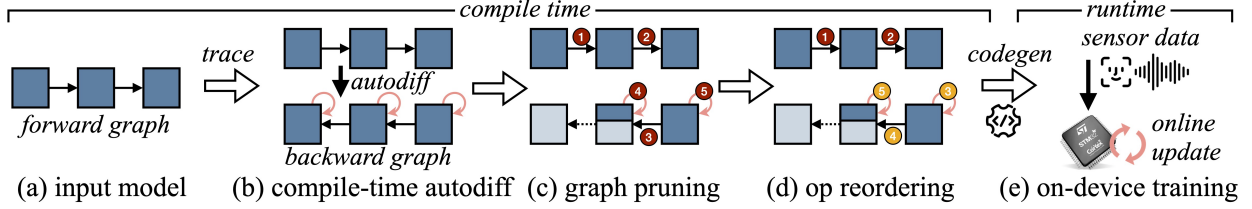


Fig. 19. The workflow of our Tiny Training Engine (TTE). (a,b) Our engine traces the forward graph for a given model and derives the corresponding backward graph at compile time. The red cycles denote the gradient descent operators. (c) To reduce memory requirements, nodes related with frozen weights (colored in light blue) are pruned from backward computation. (d) To minimize memory footprint, the gradient descent operators are re-ordered to be interlaced with backward computations (colored in yellow). (e) TTE compiles forward and backward graphs using code generation and deploys training on tiny IoT devices (best viewed in colors).

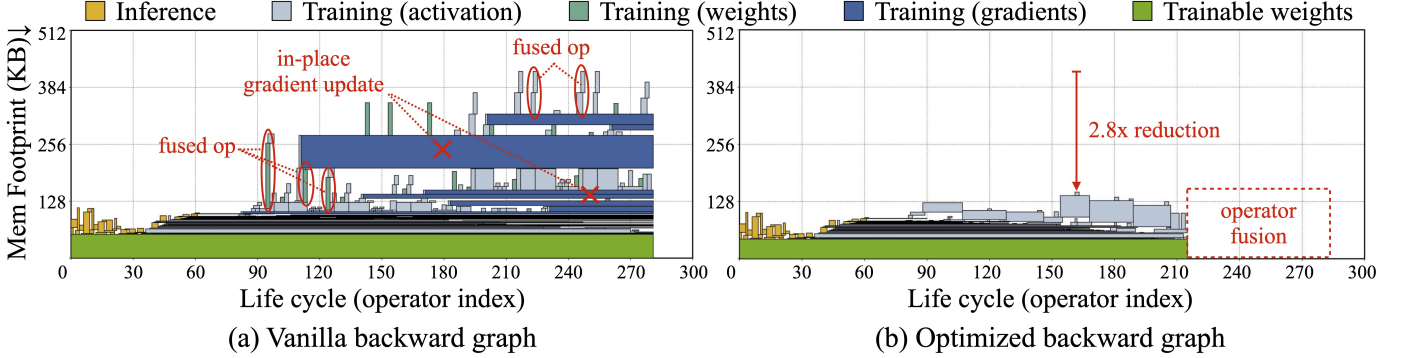


Fig. 20. Memory footprint reduction by operator reordering. With operator reordering, TTE can apply in-place gradient update and perform operator fusion to avoid large intermediate tensors to reduce memory footprint. We profiled MobileNetV2-w0.35 in this figure (same as Figure 15).

C. Tiny Training Engine (TTE)

The theoretical saving from real quantized training and sparse update does not translate to measured memory saving in existing deep learning frameworks, due to the redundant runtime and the lack of graph pruning. MCUNetV3 co-designed an efficient training system, Tiny Training Engine (TTE), to transform the above algorithms into slim binary codes (Figure 19).

Compile-time Differentiation and Code Generation. TTE offloads the auto-differentiation from the runtime to the compile-time, generating a static backward graph that can be pruned and optimized (see below) to reduce the memory and computation. TTE is based on code generation: it compiles the optimized graphs to executable binaries on the target hardware, which minimizes the runtime library size and removes the need for host languages like Python (typically uses Megabytes of memory).

Backward Graph Pruning for Sparse Update. TTE prunes the redundant nodes in the backward graph before compiling it to binary codes. For sparse layer update, TTE prunes away the gradient nodes of the frozen weights, only keeping the nodes for bias update. Afterward, TTE traverses the graph to find unused intermediate nodes due to pruning (e.g., saved input activation) and apply dead-code elimination (DCE) to remove the redundancy. For sparse tensor update, TTE introduces a *sub-operator slicing* mechanism to split a layer’s weights into trainable and frozen parts; the backward graph of the frozen subset is removed. TTE’s compilation translates the sparse update algorithm into the measured memory saving,

reducing the training memory by 7-9 \times without losing accuracy (Figure 21(a)).

Operator Reordering and Graph Optimization. The execution order of different operations affects the life cycle of tensors and the overall memory footprint. This has been well-studied for inference [101, 44] but not for training due to the extra complexity. Traditional training frameworks usually derive the gradients of all the trainable parameters before applying the update. Such a practice leads to significant memory waste for storing the gradients. By reordering operators, the gradient update to a specific tensor can immediately be applied (in-place update) before back-propagating to earlier layers, so that the gradient can be released. As such, TTE trace the dependency of all tensors (weights, gradients, activation) and reorder the operators, so that some operators can be fused to reduce memory footprint (by 2.4-3.2 \times , Figure 21(a)). Figure 20 provides an example to reflect the memory saving from reordering.

Memory Saving & Faster Training Figure 21(a) shows the training memory of three models on STM32F746 MCU to compare the memory saving from TTE. The sparse update effectively reduces peak memory by 7-9 \times compared to the full update thanks to the graph pruning mechanism, while achieving the same or higher transfer learning accuracy (compare the data points connected by arrows in Figure 18). The memory is further reduced with operator reordering, leading to 20-21 \times total memory saving. With both techniques, the training of all 3 models fits 256KB SRAM.

The training latency per image on the STM32F746 MCU is measured in Figure 21(c). By graph optimization and

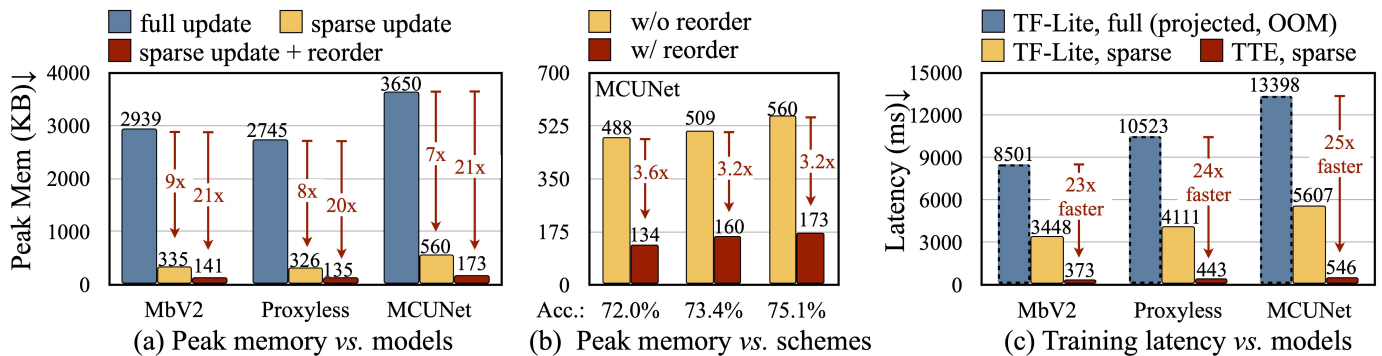


Fig. 21. Measured peak memory and latency: (a) Sparse update with TTE graph optimization can reduce the measured peak memory by 20-21 \times for different models, making training feasible on tiny edge devices. (b) Graph optimization consistently reduces the peak memory for different sparse update schemes (denoted by different average transfer learning accuracies). (c) Sparse update with TTE operators achieves 23-25 \times faster training speed compared to the full update with TF-Lite Micro operators, leading to less energy usage. *Note:* for sparse update, we choose the config that achieves the same accuracy as full update.

exploiting multiple compiler optimization approaches (such as loop unrolling and tiling), our sparse update + TTE kernels can significantly enhance the training speed by 23-25 \times compared to the full update + TF-Lite Micro kernels, leading to energy saving and making training practical.

V. CONCLUSION AND OUTLOOK

In conclusion, TinyML is a rapidly evolving field that enables deep learning on resource-constrained devices. It fosters a wide range of customized and private AI applications on edge devices, which can process the data collected from the sensors right at the source. We point out several unique challenges of TinyML. First, we need to redesign the model design space since deep models designed for mobile and other platforms do not work well for TinyML. Second, we need to redesign backpropagation schemes and investigate new learning algorithms since directly adapting models for inference does not work for tiny training. Third, co-design is necessary for TinyML. We summarize the related works aiming to overcome the challenges from the algorithm and the system perspectives. Furthermore, we introduce the TinyML techniques that not only enable practical AI applications on a wide range of IoT platforms for inference, but also allow AI to be continuously trained over time, adapting to a world that is changing fast. Looking to the future, TinyML will continue to be an active and rapidly growing area, which requires continued efforts to improve the performance and energy efficiency. We discuss several possible directions for the future development of TinyML.

More applications and modalities. This review mainly focuses on convolutional neural networks (CNNs) as computer vision is widely adapted to tiny devices. However, TinyML has a broad range of applications beyond computer vision, including but not limited to audio processing, language processing, anomaly detection, *etc.*, with sensor inputs from temperature/humidity sensors, accelerometers, current/voltage sensors, among others. TinyML enables local devices to process multiple-sensor inputs to handle multi-task workloads, opening up future avenues for numerous potential applications. We will leave further exploration of these possibilities for future work.

Self-supervised learning. Obtaining accurately labeled data

for on-device learning on the edge can be challenging. In some cases, like keyboard typing, we can use the next input word as the prediction target for the model. However, this is not always practical for most applications, such as domain adaptation for vision tasks (e.g., segmentation, detection), obtaining supervision can be expensive and difficult. One potential solution is to design self-supervised learning tasks for on-device training, as has been proposed in recent research [151].

Relationship between TinyML and LargeML. TinyML and LargeML both aim to develop efficient models under resource constraints such as memory, computation, engineering effort, and data. While TinyML is primarily focusing on making models run efficiently on small devices, many of its techniques can also be applied in cloud environments for large-scale machine learning scenarios. For example, quantization techniques have been effective in both TinyML [8, 9] and LargeML settings [152, 153], and the concept of sparse learning has been used in both scenarios to run models efficiently with limited resources [154, 56]. These efficient techniques are generally applicable and should not be limited to TinyML settings.

The concept of TinyML is constantly evolving and expanding. When ResNet-50 [10] was first introduced in 2016, it was considered as a large model with 25M parameters and 4G MACs. However, 6 years later, with the rapid advances in hardware, it can now achieve sub-millisecond inference on a smartphone DSP (Qualcomm Snapdragon 8Gen1). As hardware continues to improve, what was once considered a “large” model may be considered “tiny” in the future. The scope of TinyML should evolve and adapt over time.

VI. ACKNOWLEDGEMENT

We thank MIT AI Hardware Program, National Science Foundation, NVIDIA Academic Partnership Award, MIT-IBM Watson AI Lab, Amazon and MIT Science Hub, Qualcomm Innovation Fellowship, Microsoft Turing Academic Program for supporting this research.

REFERENCES

- [1] Z. Liu, Z. Wu, C. Gan, L. Zhu, and S. Han, “Datamix: Efficient privacy-preserving edge-cloud inference,” in

- European Conference on Computer Vision*. Springer, 2020, pp. 578–595.
- [2] A. Singh, P. Vepakomma, O. Gupta, and R. Raskar, “Detailed comparison of communication efficiency of split learning and federated learning,” *arXiv preprint arXiv:1909.09145*, 2019.
 - [3] J. Konečný, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon, “Federated learning: Strategies for improving communication efficiency,” *arXiv preprint arXiv:1610.05492*, 2016.
 - [4] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, “Mobilenetv2: Inverted residuals and linear bottlenecks,” in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.
 - [5] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, “Mobilenets: Efficient convolutional neural networks for mobile vision applications,” *arXiv preprint arXiv:1704.04861*, 2017.
 - [6] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun, “Shufflenet v2: Practical guidelines for efficient cnn architecture design,” in *ECCV*, 2018.
 - [7] X. Zhang, X. Zhou, M. Lin, and J. Sun, “Shufflenet: An extremely efficient convolutional neural network for mobile devices,” in *CVPR*, 2018.
 - [8] J. Lin, W.-M. Chen, Y. Lin, J. Cohn, C. Gan, and S. Han, “Mcnunet: Tiny deep learning on iot devices,” in *NeurIPS*, 2020.
 - [9] J. Lin, W.-M. Chen, H. Cai, C. Gan, and S. Han, “Mcnunetv2: Memory-efficient patch-based inference for tiny deep learning,” *arXiv preprint arXiv:2110.15352*, 2021.
 - [10] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.
 - [11] V. Tsoukas, E. Boumpa, G. Giannakas, and A. Kakarountas, “A review of machine learning and tinyml in healthcare,” in *25th Pan-Hellenic Conference on Informatics*, 2021, pp. 69–73.
 - [12] A. Rana, Y. Dhiman, and R. Anand, “Cough detection system using tinyml,” in *2022 International Conference on Computing, Communication and Power Technology (IC3P)*. IEEE, 2022, pp. 119–122.
 - [13] O. D’Souza, S. C. Mukhopadhyay, and M. Sheng, “Health, security and fire safety process optimisation using intelligence at the edge,” *Sensors*, vol. 22, no. 21, 2022. [Online]. Available: <https://www.mdpi.com/1424-8220/22/21/8143>
 - [14] A.-T. Shumba, T. Montanaro, I. Sergi, L. Fachechi, M. De Vittorio, and L. Patrono, “Leveraging iot-aware technologies and ai techniques for real-time critical healthcare applications,” *Sensors*, vol. 22, no. 19, 2022. [Online]. Available: <https://www.mdpi.com/1424-8220/22/19/7675>
 - [15] M. Vuletic, V. Mujagic, N. Milojevic, and D. Biswas, “Edge ai framework for healthcare applications.”
 - [16] A. Wong, M. Famouri, M. Pavlova, and S. Surana, “Tinyspeech: Attention condensers for deep speech recognition neural networks on edge devices,” *arXiv preprint arXiv:2008.04245*, 2020.
 - [17] M. Mazumder, C. Banbury, J. Meyer, P. Warden, and V. J. Reddi, “Few-shot keyword spotting in any language,” *arXiv preprint arXiv:2104.01454*, 2021.
 - [18] E. Hardy and F. Badets, “An ultra-low power rnn classifier for always-on voice wake-up detection robust to real-world scenarios,” *arXiv preprint arXiv:2103.04792*, 2021.
 - [19] C.-H. Lu and X.-Z. Lin, “Toward direct edge-to-edge transfer learning for iot-enabled edge cameras,” *IEEE Internet of Things Journal*, vol. 8, no. 6, pp. 4931–4943, 2020.
 - [20] M. Giordano, P. Mayer, and M. Magno, “A battery-free long-range wireless smart camera for face detection,” in *Proceedings of the 8th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems*, 2020, pp. 29–35.
 - [21] T. Luukkonen, A. Colley, T. Seppänen, and J. Häkkinä, “Cough activated dynamic face visor,” in *Augmented Humans Conference 2021*, 2021, pp. 295–297.
 - [22] P. Mohan, A. J. Paul, and A. Chirania, “A tiny cnn architecture for medical face mask detection for resource-constrained endpoints,” in *Innovations in Electrical and Electronic Engineering*. Springer, 2021, pp. 657–670.
 - [23] A. Wong, M. Famouri, and M. J. Shafiee, “Attendnets: tiny deep image recognition neural networks for the edge via visual attention condensers,” *arXiv preprint arXiv:2009.14385*, 2020.
 - [24] S. Benatti, F. Montagna, V. Kartsch, A. Rahimi, D. Rossi, and L. Benini, “Online learning and classification of emg-based gestures on a parallel ultra-low power platform using hyperdimensional computing,” *IEEE transactions on biomedical circuits and systems*, vol. 13, no. 3, pp. 516–528, 2019.
 - [25] A. Moin, A. Zhou, A. Rahimi, A. Menon, S. Benatti, G. Alexandrov, S. Tamakloe, J. Ting, N. Yamamoto, Y. Khan *et al.*, “A wearable biosensing system with in-sensor adaptive machine learning for hand gesture recognition,” *Nature Electronics*, vol. 4, no. 1, pp. 54–63, 2021.
 - [26] A. Zhou, R. Muller, and J. Rabaey, “Memory-efficient, limb position-aware hand gesture recognition using hyperdimensional computing,” *arXiv preprint arXiv:2103.05267*, 2021.
 - [27] S. Bian and P. Lukowicz, “Capacitive sensing based on-board hand gesture recognition with tinyml,” in *Adjunct Proceedings of the 2021 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2021 ACM International Symposium on Wearable Computers*, 2021, pp. 4–5.
 - [28] A. J. Paul, P. Mohan, and S. Sehgal, “Rethinking generalization in american sign language prediction for edge devices with extremely low memory footprint,” in *2020 IEEE Recent Advances in Intelligent Computational Systems (RAICS)*. IEEE, 2020, pp. 147–152.
 - [29] M. de Prado, M. Rusci, A. Capotondi, R. Donze, L. Benini, and N. Pazos, “Robustifying the deployment

- of tinyml models for autonomous mini-vehicles,” *Sensors*, vol. 21, no. 4, p. 1339, 2021.
- [30] A. N. Roshan, B. Gokulapriyan, C. Siddarth, and P. Kokil, “Adaptive traffic control with tinyml,” in *2021 Sixth International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)*. IEEE, 2021, pp. 451–455.
 - [31] W. Bao, C. Wu, S. Guleng, J. Zhang, K.-L. A. Yau, and Y. Ji, “Edge computing-based joint client selection and networking scheme for federated learning in vehicular iot,” *China Communications*, vol. 18, no. 6, pp. 39–52, 2021.
 - [32] J. Ying, J. Hsieh, D. Hou, J. Hou, T. Liu, X. Zhang, Y. Wang, and Y.-T. Pan, “Edge-enabled cloud computing management platform for smart manufacturing,” in *2021 IEEE International Workshop on Metrology for Industry 4.0 & IoT (MetroInd4.0&IoT)*, 2021, pp. 682–686.
 - [33] Y. Y. Siang, M. R. Ahamd, and M. S. Z. Abidin, “Anomaly detection based on tiny machine learning: A review,” *Open International Journal of Informatics*, vol. 9, no. Special Issue 2, pp. 67–78, 2021.
 - [34] K. Roth, L. Pemula, J. Zepeda, B. Schölkopf, T. Brox, and P. Gehler, “Towards total recall in industrial anomaly detection,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 14 318–14 328.
 - [35] F. Alongi, N. Ghielmetti, D. Pau, F. Terraneo, and W. Fornaciari, “Tiny neural networks for environmental predictions: an integrated approach with miosix,” in *2020 IEEE International Conference on Smart Computing (SMARTCOMP)*. IEEE, 2020, pp. 350–355.
 - [36] C. Vuppapapati, A. Ilapakurti, K. Chillara, S. Kedari, and V. Mamidi, “Automating tiny ml intelligent sensors devops using microsoft azure,” in *2020 IEEE International Conference on Big Data (Big Data)*. IEEE, 2020, pp. 2375–2384.
 - [37] C. Vuppapapati, A. Ilapakurti, S. Kedari, J. Vuppapapati, S. Kedari, and R. Vuppapapati, “Democratization of ai, albeit constrained iot devices & tiny ml, for creating a sustainable food future,” in *2020 3rd International Conference on Information and Computer Technologies (ICICT)*. IEEE, 2020, pp. 525–530.
 - [38] F. Nakhle and A. L. Harfouche, “Ready, steady, go ai: A practical tutorial on fundamentals of artificial intelligence and its applications in phenomics image analysis,” *Patterns*, vol. 2, no. 9, p. 100323, 2021.
 - [39] D. J. Curnick, A. J. Davies, C. Duncan, R. Freeman, D. M. Jacoby, H. T. Shelley, C. Rossi, O. R. Wearn, M. J. Williamson, and N. Pettoelli, “Smallsats: a new technological frontier in ecology and conservation?” *Remote Sensing in Ecology and Conservation*, vol. 8, no. 2, pp. 139–150, 2022.
 - [40] C. Nicolas, B. Naila, and R.-C. Amar, “Tinyml smart sensor for energy saving in internet of things precision agriculture platform,” in *2022 Thirteenth International Conference on Ubiquitous and Future Networks (ICUFN)*. IEEE, 2022, pp. 256–259.
 - [41] L. Lai, N. Suda, and V. Chandra, “Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus,” *arXiv preprint arXiv:1801.06601*, 2018.
 - [42] STMicroelectronics, “X-cube-ai: Ai expansion pack for stm32cubemx,” <https://www.st.com/en/embedded-software/x-cube-ai.html>.
 - [43] “microtvm: Tvm on bare-metal,” <https://tvm.apache.org/docs/topic/microtvm/index.html>.
 - [44] E. Liberis and N. D. Lane, “Neural networks on microcontrollers: saving memory at inference via operator reordering,” *arXiv preprint arXiv:1910.05110*, 2019.
 - [45] M. Rusci, A. Capotondi, and L. Benini, “Memory-driven mixed low precision quantization for enabling deep network inference on microcontrollers,” in *MLSys*, 2020.
 - [46] A. Capotondi, M. Rusci, M. Fariselli, and L. Benini, “Cmix-nn: Mixed low-precision cnn library for memory-constrained edge devices,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 5, pp. 871–875, 2020.
 - [47] R. David, J. Duke, A. Jain, V. Janapa Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, T. Wang, P. Warden, and R. Rhodes, “Tensorflow lite micro: Embedded machine learning for tinyml systems,” in *Proceedings of Machine Learning and Systems*, vol. 3, 2021, pp. 800–811.
 - [48] C. Banbury, C. Zhou, I. Fedorov, R. Matas, U. Thakker, D. Gope, V. Janapa Reddi, M. Mattina, and P. Whatmough, “Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers,” *Proceedings of Machine Learning and Systems*, vol. 3, 2021.
 - [49] S. Sadiq, J. Hare, P. Maji, S. Craske, and G. V. Merrett, “Tinyops: Imagenet scale deep learning on microcontrollers,” in *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2022, pp. 2701–2705.
 - [50] “Tinymaix,” <https://github.com/sipeed/TinyMaix>.
 - [51] I. Fedorov, R. Matas, H. Tann, C. Zhou, M. Mattina, and P. Whatmough, “UDC: Unified DNAs for compressible tinyML models for neural processing units,” in *Advances in Neural Information Processing Systems*, 2022.
 - [52] H. Cai, C. Gan, L. Zhu, and S. Han, “Tinytl: Reduce activations, not trainable parameters for efficient on-device learning,” *arXiv preprint arXiv:2007.11622*, 2020.
 - [53] H. Ren, D. Anicic, and T. A. Runkler, “Tinyol: Tinyml with online-learning on microcontrollers,” in *2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2021, pp. 1–8.
 - [54] S. G. Patil, P. Jain, P. Dutta, I. Stoica, and J. Gonzalez, “Poet: Training neural networks on tiny devices with integrated rematerialization and paging,” in *International Conference on Machine Learning*. PMLR, 2022, pp. 17 573–17 583.
 - [55] C. Profentzas, M. Almgren, and O. Landsiedel, “Minilearn: On-device learning for low-power iot devices,” in *Proceedings of the 2022 International Conference on Embedded Wireless Systems and Networks (Linz, Austria)(EWSN’22)*. Junction Publishing, USA, 2022.
 - [56] J. Lin, L. Zhu, W.-M. Chen, W.-C. Wang, C. Gan, and

- S. Han, "On-device training under 256kb memory," 2022.
- [57] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *NeurIPS*, 2015.
- [58] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *ICCV*, 2017.
- [59] J. Lin, Y. Rao, J. Lu, and J. Zhou, "Runtime neural pruning," in *NeurIPS*, 2017.
- [60] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, "Learning efficient convolutional networks through network slimming," in *ICCV*, 2017.
- [61] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han, "Amc: Automl for model compression and acceleration on mobile devices," in *ECCV*, 2018.
- [62] Z. Liu, H. Mu, X. Zhang, Z. Guo, X. Yang, K.-T. Cheng, and J. Sun, "MetaPruning: Meta Learning for Automatic Neural Network Channel Pruning," in *ICCV*, 2019.
- [63] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," in *ICLR*, 2016.
- [64] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," in *ICLR*, 2017.
- [65] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *ECCV*, 2016.
- [66] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *arXiv preprint arXiv:1606.06160*, 2016.
- [67] M. Courbariaux and Y. Bengio, "Binarynet: Training deep neural networks with weights and activations constrained to+ 1 or-1," *arXiv preprint arXiv:1602.02830*, 2016.
- [68] J. Choi, Z. Wang, S. Venkataramani, P. I.-J. Chuang, V. Srinivasan, and K. Gopalakrishnan, "Pact: Parameterized clipping activation for quantized neural networks," *arXiv preprint arXiv:1805.06085*, 2018.
- [69] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "HAQ: Hardware-Aware Automated Quantization with Mixed Precision," in *CVPR*, 2019.
- [70] H. F. Langroudi, V. Karia, T. Pandit, and D. Kudithipudi, "Tent: Efficient quantization of neural networks on the tiny edge with tapered fixed point," *arXiv preprint arXiv:2104.02233*, 2021.
- [71] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky, "Speeding-up convolutional neural networks using fine-tuned cp-decomposition," *arXiv preprint arXiv:1412.6553*, 2014.
- [72] Y. Gong, L. Liu, M. Yang, and L. Bourdev, "Compressing deep convolutional networks using vector quantization," *arXiv preprint arXiv:1412.6115*, 2014.
- [73] Y.-D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin, "Compression of deep convolutional neural networks for fast and low power mobile applications," *arXiv preprint arXiv:1511.06530*, 2015.
- [74] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.
- [75] W. Park, D. Kim, Y. Lu, and M. Cho, "Relational knowledge distillation," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 3967–3976.
- [76] F. Tung and G. Mori, "Similarity-preserving knowledge distillation," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019, pp. 1365–1374.
- [77] S. I. Mirzadeh, M. Farajtabar, A. Li, N. Levine, A. Matsukawa, and H. Ghasemzadeh, "Improved knowledge distillation via teacher assistant," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 34, no. 04, 2020, pp. 5191–5198.
- [78] L. Wang and K.-J. Yoon, "Knowledge distillation and student-teacher learning for visual intelligence: A review and new outlooks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.
- [79] Z. Yang, Z. Li, X. Jiang, Y. Gong, Z. Yuan, D. Zhao, and C. Yuan, "Focal and global knowledge distillation for detectors," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 4643–4652.
- [80] B. Zhao, Q. Cui, R. Song, Y. Qiu, and J. Liang, "Decoupled knowledge distillation," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 11 953–11 962.
- [81] L. Beyer, X. Zhai, A. Royer, L. Markeeva, R. Anil, and A. Kolesnikov, "Knowledge distillation: A good teacher is patient and consistent," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 10 925–10 934.
- [82] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," in *ICLR*, 2017.
- [83] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *CVPR*, 2018.
- [84] H. Liu, K. Simonyan, and Y. Yang, "Darts: Differentiable architecture search," in *ICLR*, 2019.
- [85] H. Cai, L. Zhu, and S. Han, "ProxylessNAS: Direct neural architecture search on target task and hardware," in *ICLR*, 2019. [Online]. Available: <https://arxiv.org/pdf/1812.00332.pdf>
- [86] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, "Mnasnet: Platform-aware neural architecture search for mobile," in *CVPR*, 2019.
- [87] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer, "Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search," in *CVPR*, 2019.
- [88] I. Radosavovic, R. P. Kosaraju, R. Girshick, K. He, and P. Dollár, "Designing network design spaces," *arXiv preprint arXiv:2003.13678*, 2020.
- [89] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga et al., "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.
- [90] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis,

- J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, “Tensorflow: A system for large-scale machine learning,” in *OSDI*, 2016.
- [91] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, “Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems,” *arXiv preprint arXiv:1512.01274*, 2015.
- [92] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Neca, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, “JAX: composable transformations of Python+NumPy programs,” 2018. [Online]. Available: <http://github.com/google/jax>
- [93] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, “{TVM}: An automated end-to-end optimizing compiler for deep learning,” in *OSDI*, 2018.
- [94] “Tensorflow lite,” <https://www.tensorflow.org/lite>.
- [95] X. Jiang, H. Wang, Y. Chen, Z. Wu, L. Wang, B. Zou, Y. Yang, Z. Cui, Y. Cai, T. Yu, C. Lv, and Z. Wu, “Mnn: A universal and efficient inference engine,” in *MLSys*, 2020.
- [96] “Ncnn : A high-performance neural network inference computing framework optimized for mobile platforms,” <https://github.com/Tencent/ncnn>.
- [97] “Nvidia tensorrt, an sdk for high-performance deep learning inference,” <https://developer.nvidia.com/tensorrt>.
- [98] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” in *NeurIPS*, 2017.
- [99] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, “Learning to optimize tensor programs,” in *NeurIPS*, 2018.
- [100] A. Stoutchinin, F. Conti, and L. Benini, “Optimally scheduling cnn convolutions for efficient memory access,” *arXiv preprint arXiv:1902.01492*, 2019.
- [101] B. H. Ahn, J. Lee, J. M. Lin, H.-P. Cheng, J. Hou, and H. Esmaeilzadeh, “Ordering chaos: Memory-aware scheduling of irregularly wired neural networks for edge devices,” *arXiv preprint arXiv:2003.02369*, 2020.
- [102] H. Miao and F. X. Lin, “Enabling large neural networks on tiny microcontrollers with swapping,” *arXiv preprint arXiv:2101.08744*, 2021.
- [103] M. Alwani, H. Chen, M. Ferdman, and P. Milder, “Fused-layer cnn accelerators,” in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.
- [104] K. Goetschalckx and M. Verhelst, “Breaking high-resolution cnn bandwidth barriers with enhanced depth-first execution,” *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 323–331, 2019.
- [105] O. Saha, A. Kusupati, H. V. Simhadri, M. Varma, and P. Jain, “Rnnpool: Efficient non-linear pooling for ram constrained inference,” *arXiv preprint arXiv:2002.11921*, 2020.
- [106] M. Tan and Q. Le, “Efficientnet: Rethinking model scaling for convolutional neural networks,” in *International Conferences on Machine Learning (ICML)*. PMLR, 2019, pp. 6105–6114.
- [107] M. Tan and Q. V. Le, “Efficientnetv2: Smaller models and faster training,” *CoRR*, vol. abs/2104.00298, 2021. [Online]. Available: <https://arxiv.org/abs/2104.00298>
- [108] A. Gruslys, R. Munos, I. Danihelka, M. Lanctot, and A. Graves, “Memory-efficient backpropagation through time,” in *NeurIPS*, 2016, p. 4132–4140.
- [109] T. Chen, B. Xu, C. Zhang, and C. Guestrin, “Training deep nets with sublinear memory cost,” *arXiv preprint arXiv:1604.06174*, 2016.
- [110] K. Greff, R. K. Srivastava, and J. Schmidhuber, “Highway and residual networks learn unrolled iterative estimation,” in *ICLR*, 2017. [Online]. Available: <https://arxiv.org/pdf/1604.06174.pdf>
- [111] L. Liu, L. Deng, X. Hu, M. Zhu, G. Li, Y. Ding, and Y. Xie, “Dynamic sparse graph for efficient deep learning,” in *ICLR*, 2019.
- [112] Y. Wang, Z. Jiang, X. Chen, P. Xu, Y. Zhao, Y. Lin, and Z. Wang, “E2-train: Training state-of-the-art cnns with over 80% energy savings,” 2019. [Online]. Available: <https://arxiv.org/abs/1910.13349>
- [113] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan, “Training deep neural networks with 8-bit floating point numbers,” in *NeurIPS*, 2018.
- [114] X. Sun, J. Choi, C.-Y. Chen, N. Wang, S. Venkataramani, V. V. Srinivasan, X. Cui, W. Zhang, and K. Gopalakrishnan, “Hybrid 8-bit floating point (hfp8) training and inference for deep neural networks,” in *NeurIPS*, 2019.
- [115] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in Neural Information Processing Systems (NIPS)*, 2012.
- [116] Y. Cui, Y. Song, C. Sun, A. Howard, and S. Belongie, “Large scale fine-grained categorization and domain-specific transfer learning,” in *CVPR*, 2018.
- [117] S. Kornblith, J. Shlens, and Q. V. Le, “Do better imagenet models transfer better?” in *CVPR*, 2019.
- [118] A. Kolesnikov, L. Beyer, X. Zhai, J. Puigcerver, J. Yung, S. Gelly, and N. Houlsby, “Big transfer (bit): General visual representation learning,” in *European conference on computer vision*. Springer, 2020, pp. 491–507.
- [119] K. Chatfield, K. Simonyan, A. Vedaldi, and A. Zisserman, “Return of the devil in the details: Delving deep into convolutional nets,” in *BMVC*, 2014.
- [120] J. Donahue, Y. Jia, O. Vinyals, J. Hoffman, N. Zhang, E. Tzeng, and T. Darrell, “Decaf: A deep convolutional activation feature for generic visual recognition,” in *International Conferences on Machine Learning (ICML)*, 2014.
- [121] C. Gan, N. Wang, Y. Yang, D.-Y. Yeung, and A. G. Hauptmann, “DevNet: a deep event network for multimedia event detection and evidence recounting,” in *CVPR*, 2015, pp. 2568–2577.
- [122] A. Sharif Razavian, H. Azizpour, J. Sullivan, and S. Carlsson, “Cnn features off-the-shelf: an astounding baseline for recognition,” in *CVPR Workshops*, 2014.

- [123] Q. Wang, M. Xu, C. Jin, X. Dong, J. Yuan, X. Jin, G. Huang, Y. Liu, and X. Liu, "Melon: Breaking the memory wall for resource-efficient on-device machine learning," in *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, ser. MobiSys '22, 2022, p. 450–463.
- [124] D. Xu, M. Xu, Q. Wang, S. Wang, Y. Ma, K. Huang, G. Huang, X. Jin, and X. Liu, "Mandheling: Mixed-precision on-device dnn training with dsp offloading," in *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, ser. MobiCom '22, 2022, p. 214–227.
- [125] I. Gim and J. Ko, "Memory-efficient dnn training on mobile devices," in *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, ser. MobiSys '22, 2022, p. 464–476.
- [126] J. Frankle, D. J. Schwab, and A. S. Morcos, "Training batchnorm and only batchnorm: On the expressive power of random features in cnns," *arXiv preprint arXiv:2003.00152*, 2020.
- [127] P. K. Mudrakarta, M. Sandler, A. Zhmoginov, and A. Howard, "K for the price of 1: Parameter efficient multi-task and transfer learning," in *ICLR*, 2019.
- [128] A. Canziani, A. Paszke, and E. Culurciello, "An analysis of deep neural network models for practical applications," *arXiv preprint arXiv:1605.07678*, 2016.
- [129] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han, "Once for all: Train one network and specialize it for efficient deployment," in *ICLR*, 2020. [Online]. Available: <https://arxiv.org/pdf/1908.09791.pdf>
- [130] G. Bender, P.-J. Kindermans, B. Zoph, V. Vasudevan, and Q. Le, "Understanding and simplifying one-shot architecture search," in *International Conferences on Machine Learning (ICML)*, 2018.
- [131] Z. Guo, X. Zhang, H. Mu, W. Heng, Z. Liu, Y. Wei, and J. Sun, "Single path one-shot neural architecture search with uniform sampling," *arXiv preprint arXiv:1904.00420*, 2019.
- [132] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations (ICLR)*, 2015.
- [133] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan *et al.*, "Searching for mobilenetv3," in *ICCV*, 2019.
- [134] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes (voc) challenge," *International journal of computer vision*, vol. 88, no. 2, pp. 303–338, 2010.
- [135] J. Redmon and A. Farhadi, "YOLOv3: An Incremental Improvement," *arXiv*, 2018.
- [136] S. Yang, P. Luo, C. C. Loy, and X. Tang, "Wider face: A face detection benchmark," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.
- [137] Y. Yoo, D. Han, and S. Yun, "Extd: Extremely tiny face detector via iterative filter reuse," *arXiv preprint arXiv:1906.06579*, 2019.
- [138] Y. He, D. Xu, L. Wu, M. Jian, S. Xiang, and C. Pan, "Lffd: A light and fast face detector for edge devices," *arXiv preprint arXiv:1904.10633*, 2019.
- [139] X. Zhao, X. Liang, C. Zhao, M. Tang, and J. Wang, "Real-time multi-scale face detector on embedded devices," *Sensors*, vol. 19, no. 9, p. 2158, 2019.
- [140] S. Zhang, X. Zhu, Z. Lei, H. Shi, X. Wang, and S. Z. Li, "S3fd: Single shot scale-invariant face detector," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 192–201.
- [141] A. Burrello, A. Garofalo, N. Bruschi, G. Tagliavini, D. Rossi, and F. Conti, "Dory: Automatic end-to-end deployment of real-world dnns on low-cost iot mcus," *IEEE Transactions on Computers*, vol. 70, no. 8, pp. 1253–1268, 2021.
- [142] E. Liberis, Ł. Dudziak, and N. D. Lane, "μnas: Constrained neural architecture search for microcontrollers," *arXiv preprint arXiv:2010.14246*, 2020.
- [143] I. Fedorov, R. P. Adams, M. Mattina, and P. Whatmough, "Sparse: Sparse architecture search for cnns on resource-constrained microcontrollers," in *NeurIPS*, 2019.
- [144] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: <http://tensorflow.org/>
- [145] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International Conferences on Machine Learning (ICML)*, 2015.
- [146] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *CVPR*, 2018, pp. 2704–2713.
- [147] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," *Master's thesis, Department of Computer Science, University of Toronto*, 2009.
- [148] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.
- [149] Y. You, I. Gitman, and B. Ginsburg, "Large batch training of convolutional networks," *arXiv preprint arXiv:1708.03888*, 2017.
- [150] J. Krause, M. Stark, J. Deng, and L. Fei-Fei, "3d object representations for fine-grained categorization," in *Proceedings of the IEEE International Conference on Computer Vision Workshops*, 2013.
- [151] Y. Sun, X. Wang, Z. Liu, J. Miller, A. Efros, and M. Hardt, "Test-time training with self-supervision for generalization under distribution shifts," in *International conference on machine learning*. PMLR, 2020, pp.

9229–9248.

- [152] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer, “Llm.int8(): 8-bit matrix multiplication for transformers at scale,” *arXiv preprint arXiv:2208.07339*, 2022.
- [153] G. Xiao, J. Lin, M. Seznec, J. Demouth, and S. Han, “Smoothquant: Accurate and efficient post-training quantization for large language models,” *arXiv preprint arXiv:2211.10438*, 2022.
- [154] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, Y. Tay, N. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. Garcia, V. Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei, K. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, and N. Fiedel, “Palm: Scaling language modeling with pathways,” in *Machine Learning and Systems (MLSys)*, 2022.



Wei-Chen Wang is a Postdoctoral Associate at MIT EECS. Dr. Wang received his Ph.D. degree in Computer Science from the Department of Computer Science and Information Engineering at National Taiwan University in 2021. His current research interests include efficient deep learning, model compression, TinyML, and embedded systems.



NVIDIA, Samsung and SONY.

Song Han is an associate professor at MIT EECS. Dr. Han received the Ph.D. degree from Stanford University. Dr. Han’s research focuses on efficient deep learning computing at the intersection between machine learning and computer architecture. He proposed “Deep Compression” and the “Efficient Inference Engine” that widely impacted the industry. He is a recipient of NSF CAREER Award, Sloan Research Fellowship, MIT Technology Review Innovators Under 35, best paper awards at the ICLR and FPGA, and faculty awards from Amazon, Facebook,



Ji Lin is a PhD student at MIT EECS. Previously, he graduated from Department of Electronic Engineering, Tsinghua University. His research interests lie in efficient and hardware-friendly machine learning, model compression and acceleration.



Ligeng Zhu is a PhD student at MIT EECS, supervised by Professor Song Han. His study focuses on efficient and accelerated deep learning systems and algorithms.



Wei-Ming Chen is a Postdoctoral Associate at MIT EECS. Dr. Chen received his Master’s and Doctorate degrees in computer science and information engineering from National Taiwan University in 2015 and 2020, respectively. His research interests include TinyML and embedded systems with a focus on enabling efficient deep learning on edge devices.