On-Device Training Under 256KB Memory

Ji Lin[1*] **Ligeng Zhu**[1*] **Wei-Ming Chen**[1] **Wei-Chen Wang**[1] **Chuang Gan**[2] **Song Han**[1]
[1]MIT [2]MIT-IBM Watson AI Lab
https://tinyml.mit.edu

Abstract

On-device training enables the model to adapt to new data collected from the sensors by fine-tuning a pre-trained model. Users can benefit from customized AI models without having to transfer the data to the cloud, protecting the privacy. However, the training memory consumption is prohibitive for IoT devices that have tiny memory resources. We propose an algorithm-system co-design framework to make on-device training possible with only *256KB* of memory. On-device training faces two unique challenges: (1) the quantized graphs of neural networks are hard to optimize due to low bit-precision and the lack of normalization; (2) the limited hardware resource does not allow full back-propagation. To cope with the optimization difficulty, we propose ***Quantization-Aware Scaling*** to calibrate the gradient scales and stabilize 8-bit quantized training. To reduce the memory footprint, we propose ***Sparse Update*** to skip the gradient computation of less important layers and sub-tensors. The algorithm innovation is implemented by a lightweight training system, ***Tiny Training Engine***, which prunes the backward computation graph to support sparse updates and offload the runtime auto-differentiation to compile time. Our framework is the *first* solution to enable tiny on-device training of convolutional neural networks under 256KB SRAM and 1MB Flash without auxiliary memory, using less than 1/1000 of the memory of PyTorch and TensorFlow while matching the accuracy on tinyML application VWW [20]. Our study enables IoT devices not only to perform inference but also to continuously adapt to new data for on-device lifelong learning. A video demo can be found here.

1 Introduction

On-device training 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 improve its predictions and perform lifelong learning and user customization. For example, fine-tuning a language model enables continual learning from users' typing and writing; adapting a vision model enables recognizing new objects from a mobile camera. 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 [47, 46, 7, 11, 43, 24, 44, 60], let alone the *training*, which requires extra computation for the backward and extra memory for intermediate activation [18]. On the other hand, modern deep training frameworks (*e.g.*, PyTorch [56], TensorFlow [4]) are usually designed for cloud servers and require a large memory footprint (>300MB) even when training a small model (*e.g.*, MobileNetV2-w0.35 [61]) with batch size 1 (Figure. 1).

The huge gap (>1000×) makes it impossible to run on tiny IoT devices with current frameworks and algorithms. Current deep learning training systems like PyTorch [56], TensorFlow [4], JAX [10],

∗ indicates equal contributions.

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

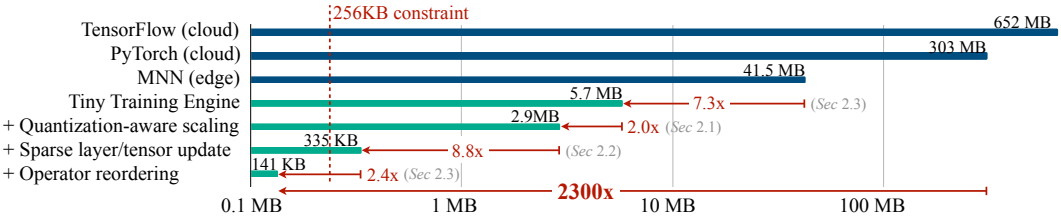


Figure 1. Algorithm and system co-design reduces the training memory from 303MB (PyTorch) to 141KB with the same transfer learning accuracy, leading to 2300× reduction. The numbers are measured with MobilenetV2-w0.35 [61], batch size 1 and resolution 128×128. It can be deployed to a microcontroller with 256KB SRAM.

MXNet [16], *etc*. do not consider the tight resources on edge devices. Edge deep learning inference frameworks like TVM [17], TF-Lite [3], NCNN [2], *etc*. provide a slim runtime, but lack the support for back-propagation. Though there are low-cost efficient transfer learning algorithms like training only the final classifier layer, bias-only update [12], *etc*., the accuracy drop is significant (Figure 9), and existing training system can not realize the theoretical saving into measured saving. 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.

In this paper, we aim to bridge the gap and enable tiny on-device training with algorithm-system co-design. We investigate tiny on-device training and find two unique challenges: (1) the model is quantized on edge devices. A *real* quantized graph is difficult to optimize due to low-precision tensors and the lack of Batch Normalization layers [33]; (2) the limited hardware resource (memory and computation) of tiny hardware does not allow full back-propagation, whose memory usage can easily exceed the SRAM of microcontrollers by more than an order of magnitude. Only updating the last layer leads to poor accuracy (Figure 9). To cope with the optimization difficulty, we propose ***Quantization-Aware Scaling (QAS)*** to automatically scale the gradient of tensors with different bit-precisions, which effectively stabilizes the training and matches the accuracy of the floating-point counterpart (Section 2.1). QAS is hyper-parameter free and no tuning is required. To reduce the memory footprint of the full backward computation, we propose ***Sparse Update*** to skip the gradient computation of less important layers and sub-tensors. We developed an automated method based on contribution analysis to find the best update scheme under different memory budgets (Section 2.2). Finally, we propose a lightweight training system, ***Tiny Training Engine (TTE)*** , to implement the algorithm innovation (Section 2.3). TTE is based on code generation; it offloads the auto-differentiation to the compile-time to greatly cut down the runtime overhead. It also supports advanced graph optimization like graph pruning and reordering to support sparse updates, achieving measured memory saving and speedup.

Our framework is the first solution to enable tiny on-device training of convolutional neural networks under 256KB SRAM and 1MB Flash without auxiliary memory. **(1)** Our solution enables weight update not only for the *classifier* but also for the *backbone*, which provides a *high transfer learning accuracy* (Figure 9). For tinyML application VWW [20], our on-device finetuned model matches the accuracy of cloud training+edge deployment, and surpasses the common requirement of tinyML (MLPerf Tiny [8]) by 9%. **(2)** Our system-algorithm co-design scheme effectively *reduces the memory footprint*. As shown in Figure 1, the proposed techniques greatly reduce the memory usage by more than 1000× compared to PyTorch and Tensorflow. **(3)** Our framework also greatly *accelerates training*, reducing the per-iteration time by more than 20× compared to dense update and vanilla system design (Figure 10). **(4)** We deployed our training system to a Cortex M7 microcontroller STM32F746 to demonstrate the feasibility, suggesting that tiny IoT devices can not only perform inference but also training to adapt to new data. Our study paves the way for lifelong on-device learning and opens up new possibilities for privacy-preserving device personalization.

2 Approach

Preliminaries. Neural networks usually need to be quantized to fit the limited memory of edge devices for inference [47, 34]. For a `fp32` linear layer $\mathbf{y}_{\texttt{fp32}} = \mathbf{W}_{\texttt{fp32}}\mathbf{x}_{\texttt{fp32}} + \mathbf{b}_{\texttt{fp32}}$, the `int8`

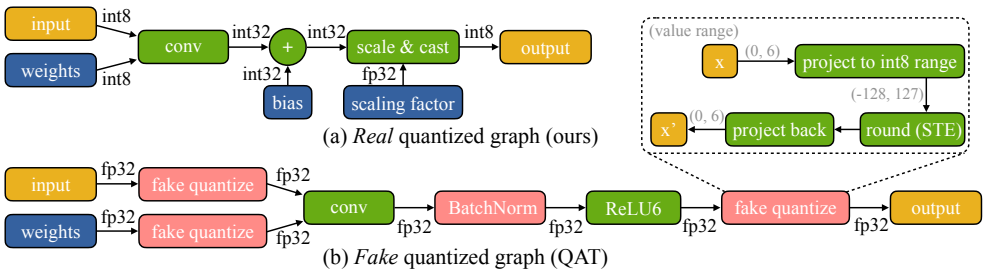


Figure 2. *Real* quantized graphs (our optimized graph, designed for *efficiency*) *vs. fake* quantized graphs (for QAT, designed for *simulation*). The fake quantize graphs cannot provide memory saving due to floating-point operations. We need to use real quantized graph to fit the tight memory constraint.

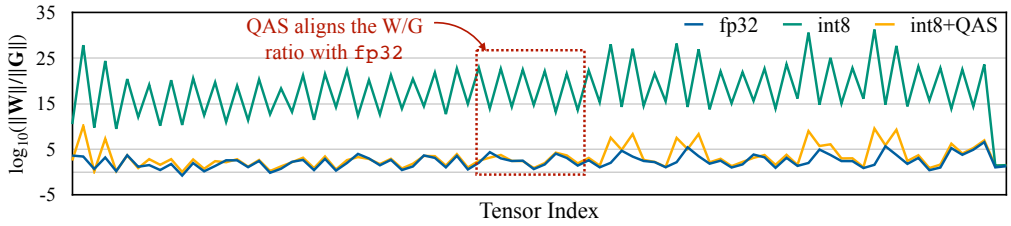


Figure 3. The quantized model has a very different weight/gradient norm ratio (*i.e.*, $\|\mathbf{W}\|/\|\mathbf{G}\|$) compared to the floating-point model at training time. QAS stabilizes the $\|\mathbf{W}\|/\|\mathbf{G}\|$ ratio and helps optimization. For example, in the highlighted area, the ratios of the quantized model fluctuate dramatically in a zigzag pattern (weight, bias, weight, bias, ...); after applying QAS, the pattern stabilizes and matches the `fp32` counterpart.

quantized counterpart is:

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

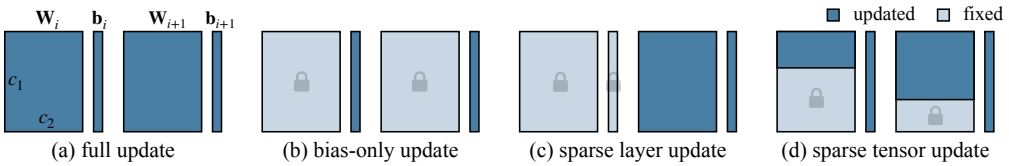
where $\bar{\cdot}$ denotes the tensor being quantized to fixed-point numbers, and s is a floating-point scaling factor to project the results back into `int8` range. We call it *real* quantized graphs (Figure 2(a)) since tensors are in `int8` format. To keep the memory efficiency, we deploy and update the *real* quantized graph on microcontrollers, and keep the updated weights as `int8`. The update formula is: $\bar{\mathbf{W}}'_{\texttt{int8}} = \texttt{cast2int8}(\bar{\mathbf{W}}_{\texttt{int8}} - \alpha \cdot \mathbf{G}_{\bar{\mathbf{W}}})$, where α is the learning rate, and $\mathbf{G}_{\bar{\mathbf{W}}}$ is the gradient of the weights. The gradient computation is also performed in `int8` for better computation efficiency.

We update the real quantized graph for training, which is fundamentally different to quantization-aware training (QAT), where a *fake* quantized graph (Figure 2(b)) is trained on the cloud, and converted to a real one for deployment. As shown in Figure 2(b), the fake quantization graph uses `fp32`, leading to no memory or computation savings. *Real* quantized graphs are for *efficiency*, while *fake* quantized graphs are for *simulation*.

2.1 Optimizing Real Quantized Graphs

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 [33] layers (fused), leading to unstable gradient update.

Gradient scale mismatch. When optimizing a quantized graph, the accuracy is lower compared to the floating-point counterpart. We hypothesize that the quantization process distorts the gradient update. To verify the idea, we 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 [40] in Figure 3. 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 we use a fixed learning rate for all the tensors, then the update speed of each tensor would be very different compared to the floating-point case, leading to inferior accuracy. We empirically find that adaptive-learning rate optimizers like Adam [36] cannot fully address the issue (Section 3.2).



(a) full update (b) bias-only update (c) sparse layer update (d) sparse tensor update

Figure 4. Different update paradigms of two linear layers in a deep neural network.

Quantization-aware scaling (QAS). To address the problem, we propose a hyper-parameter-free learning rate scaling rule, QAS. 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*, we compute a scaling rate $s_{\mathbf{W}} \in \mathbb{R}$, such that $\bar{\mathbf{W}}$'s largest magnitude is $2^7 - 1 = 127$:

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

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:

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

We find that the weight and gradient ratios are off by $s_{\mathbf{W}}^{-2}$, leading to the distorted pattern in Figure 3: (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, we propose Quantization-Aware Scaling (QAS) by compensating the gradient of the quantized graph according to Equation 3:

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

where $s_{\mathbf{X}}^{-2}$ is the scaling factor for quantizing input $\mathbf{x}$ (a scalar following [34], note that $s = s_{\mathbf{W}} \cdot s_{\mathbf{x}}$ in Equation 1). We plot the $\|\mathbf{W}\|/\|\mathbf{G}\|$ curve with QAS in Figure 3 (int8+scale). After scaling, the gradient ratios match the floating-point counterpart. QAS enables fully quantized training (`int8` for both forward and backward) while matching the accuracy of the floating-point training (Table 1).

2.2 Memory-Efficient Sparse Update

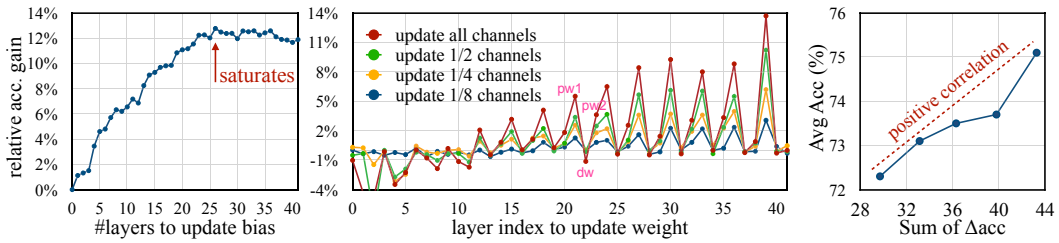
Though QAS makes optimizing a quantized model possible, updating the whole model (or even the last several blocks) requires a large amount of memory, which is not affordable for the tinyML setting. We propose to sparsely update the layers and the tensors.

Sparse layer/tensor update. Pruning techniques prove to be quite successful for achieving sparsity and reducing model size [29, 30, 48, 31, 50, 49]. Instead of pruning *weights* for inference, we "prune" the *gradient* during backpropagation, and update the model sparsely. Given a tight memory budget, we skip the update of the *less important* parameters to reduce memory usage and computation cost. We consider updating a linear layer $\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$ (similar analysis applies to convolutions). Given the output gradient $\mathbf{G}_{\mathbf{y}}$ from the later layer, we can compute the gradient update by $\mathbf{G}_{\mathbf{W}} = f_1(\mathbf{G}_{\mathbf{y}}, \mathbf{x})$ and $\mathbf{G}_{\mathbf{b}} = f_2(\mathbf{G}_{\mathbf{y}})$. Notice that updating the biases does not require saving the intermediate activation $\mathbf{x}$, leading to a lighter memory footprint [12]†; while updating the weights is more memory-intensive but also more expressive. For hardware like microcontrollers, we also need an extra copy for the updated parameters since the original ones are stored in read-only FLASH [47]. Given the different natures of updating rules, we consider the sparse update rule in three aspects (Figure 4): (1) *Bias update*: how many layers should we backpropagate to and update the biases (bias update is cheap, we always update the biases if we have backpropagated to a layer). (2) *Sparse layer update*: select a subset of layers to update the corresponding weights. (3) *Sparse tensor update*: we 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 [47] 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.

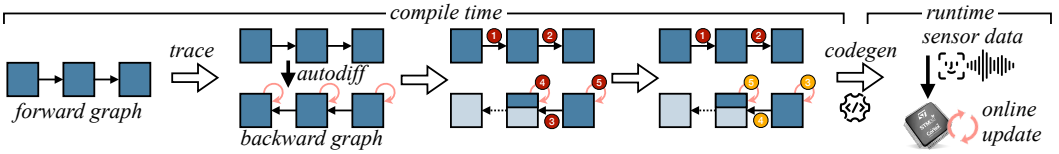
*For simplicity. We actually used per-channel quantization [34] and the scaling factor is a vector of size c_2.

†If we update many layers, the intermediate activation could consume a large memory [18].



(a) Contribution of last k biases $\Delta\text{acc}_{\mathbf{b}[:k]}$ (b) Contribution of a certain weight $\Delta\text{acc}_{\mathbf{W}_{i,r}}$ (c) Effectiveness of search

Figure 5. Contribution analysis of updating biases and weights. (a) For bias update, the accuracy generally goes higher as more layers are updated, but plateaus soon. (b) 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 [61] appears to be more important; and the gains are bigger with more channels updated. (c) The automated selection based on contribution analysis is effective: the actual downstream accuracy shows a positive correlation with $\sum \Delta\text{acc}$.



(a) input model (b) compile-time autodiff (c) graph pruning (d) op reordering (e) on-device training

Figure 6. 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).

Automated selection with contribution analysis. We propose to automatically derive the sparse update scheme by *contribution analysis*. We find the contribution of each parameter (weight/bias) to the downstream accuracy. Given a convolutional neural network with l layers, we measure the accuracy improvement from (1) biases: the improvement of updating *last* k biases $\mathbf{b}_l, \mathbf{b}_{l-1}, ..., \mathbf{b}_{l-k+1}$ (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 5 (MCUNet on Cars [39] dataset; please find more results in appendix Section F). After we find $\Delta\text{acc}_{\mathbf{b}[:k]}$ and $\Delta\text{acc}_{\mathbf{W}i}$ $(1 \leq k, i \leq l)$, we solve an optimization problem to find:

$$k^*, \mathbf{i}^*, \mathbf{r}^* = \max_{k,\mathbf{i},\mathbf{r}} (\Delta\text{acc}_{\mathbf{b}[:k]} + \sum_{i \in \mathbf{i}, r \in \mathbf{r}} \Delta\text{acc}_{\mathbf{W}i,r}) \quad \text{s.t. Memory}(k, \mathbf{i}, \mathbf{r}) \leq \text{constraint}, \tag{5}$$

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, we find the combination of (#layers for bias update, the subset of weights to update), such that the total contribution are maximized while the memory overhead does not exceed the constraint. The problem can be efficiently solved with evolutionary search (see Section D). Here we assume that the accuracy contribution of each tensor (Δacc) can be summed up. Such approximation is quite effective (Figure 5(c)).

2.3 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. We co-designed an efficient training system, Tiny Training Engine (TTE), to transform the above algorithms into slim binary codes (Figure 6).

Compile-time differentiation and code generation. TTE offloads the auto-differentiation from the runtime to the compile-time, generating a static backward graph which can be pruned and optimized (see below) to reduce the memory and computation. TTE is based on code generation: it compiles the

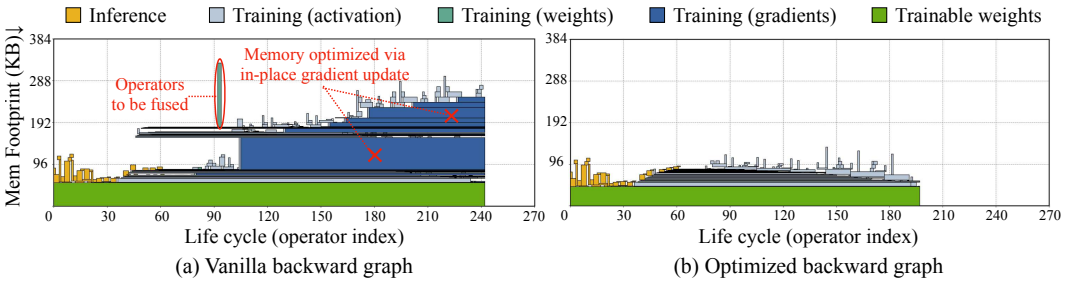


Figure 7. Memory footprint reduction by operator reordering. With operator reordering, TTE can apply in-place gradient update and perform operator fusion to avoid large intermediate tensors to reduce memory footprint. We profiled MobileNetV2-w0.35 in this figure (same as Figure 1).

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. We prune the redundant nodes in the backward graph before compiling it to binary codes. For sparse layer update, we prune away the gradient nodes of the frozen weights, only keeping the nodes for bias update. Afterwards, we traverse 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, we introduce 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. Our compiler translates the sparse update algorithm into measured memory saving, reducing the training memory 7-9× without losing accuracy (Figure 10(a), blue v.s. yellow).

Operator reordering and in-place update. The execution order of different operations affects the life cycle of tensors and the overall memory footprint. This has been well-studied for inference [6, 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, we can immediately apply the gradient update to a specific tensor (in-place update) before back-propagating to earlier layers, so that the gradient can be released. As such, we 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×, Figure 10(a), yellow v.s. red). The memory life cycle analysis in Figure 7 reflects the memory saving from in-place gradient update and operator fusion.

3 Experiments

3.1 Setups

Training. We used three popular tinyML models in our experiments: MobileNetV2 [61] (width multiplier 0.35, backbone 17M MACs, 0.25M Param), ProxylessNAS [13] (width multiplier 0.3, backbone 19M MACs, 0.33M Param), MCUNet [47] (the 5FPS ImageNet model, backbone 23M MACs, 0.48M Param). We pre-trained the models on ImageNet [22] and perform post-training quantization [34]. The quantized models are fine-tuned on downstream datasets to evaluate the transfer learning capacity. We perform the training and memory/latency measurement on a microcontroller STM32F746 (320KB SRAM, 1MB Flash) using a single batch size. To faster obtain the accuracy statistics on multiple downstream datasets, we simulate the training results on GPUs, and we verified that the simulation obtains the same level of accuracy compared to training on microcontrollers. Please refer to the the appendix (Section C) for detailed training hyper-parameters. We also provide a *video demo* of deploying our training system on microcontroller in the appendix (Section A).

Datasets. We measure the transfer learning accuracy on multiple downstream datasets and report the average accuracy [37]. We follow [12] to use a set of vision datasets including Cars [39], CIFAR-10 [40], CIFAR-100 [40], CUB [68], Flowers [54], Food [9], and Pets [55][‡]. We fine-tuned the models on all these datasets for 50 epochs following [12]. We also include VWW dataset [20], a

[‡]Pets uses CC BY-SA 4.0 license; Cars and ImageNet use the ImageNet license; others are not listed.

Table 1. Updating real quantized graphs (`int8`) for the fine-tuning is difficult: the accuracy falls behind the floating-point counterpart (`fp32`), even with adaptive learning rate optimizers like Adam [36] and LARS [69]. QAS helps to bridge the accuracy gap without memory overhead (slightly higher due to randomness). The numbers are for updating the last two blocks of MCUNet-5FPS [47] 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 [36]	54.0	84.5	61.0	58.5	87.2	62.6	80.1	86.5	71.8
	LARS [69]	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**

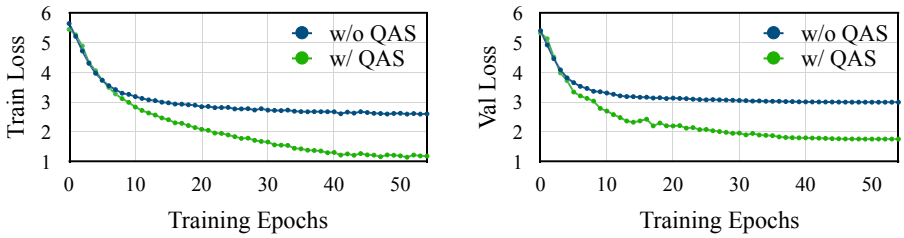


Figure 8. Training and validation loss curves w/ and w/o QAS. QAS effectively helps convergence, leading to better accuracy. The results are from updating the last two blocks of the MCUNet model on the Cars dataset.

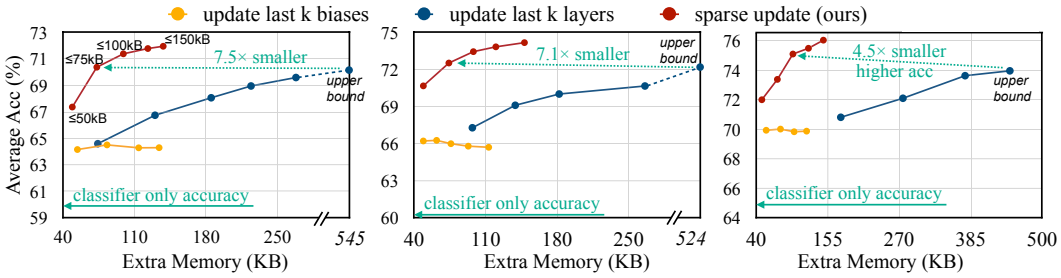
widely used benchmark for tinyML applications. We train on VWW for 10 epochs following [47]. We used resolution 128 for all datasets and models for a fair comparison.

Memory estimation. The memory usage of a computation graph is related to its implementation [6, 44, 47, 46]. We provide two settings for memory measurement: (1) **analytic profiling**: we count the size of *extra* tensors required for backward computation, including the saved intermediate activation, binary truncation task, and the updated weights. The size is implementation-agnostic. It is used for a fast profiling; (2) **on-device profiling**: we measure the actual memory usage when running model training on an STM32F746 MCU (320KB SRAM, 1MB Flash). We used TinyEngineV2 [46] as the backend and 2×2 patch-based inference [46] for the initial stage to reduce the forward peak memory. The *measured* memory determines whether a solution can be deployed on the hardware.

3.2 Experimental Results

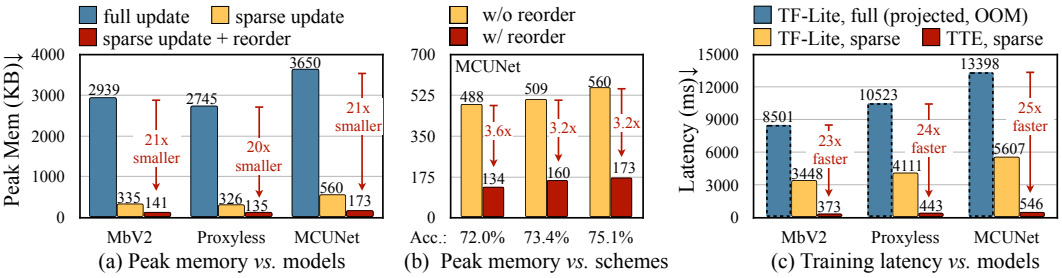
Quantization-aware scaling (QAS) addresses the optimization difficulty. We fine-tuned the last two blocks (simulate low-cost fine-tuning) of MCUNet to various downstream datasets (Table 1). With momentum SGD, the training accuracy of the quantized model (`int8`) falls behind the floating-point counterpart due to the optimization difficulty. Adaptive learning rate optimizers like Adam [36] can improve the accuracy but are still lower than the `fp32` fine-tuning results; it also costs **3× memory** consumption due to second-order momentum, which is not desired for tinyML settings. LARS [69] cannot converge well on most datasets despite extensive hyper-parameter tuning (over both learning rate and the "trust coefficient"). We hypothesize that the aggressive gradient scaling rule of LARS makes the training unstable. The accuracy gap is closed when we apply QAS, matching the accuracy of floating-point training at no extra memory cost. The learning curves (fine-tuning) of MCUNet on the Cars dataset w/ and w/o QAS are also provided in Figure 8. Therefore, QAS effectively helps optimization.

Sparse update obtains better accuracy at lower memory. We compare the performance of our searched sparse update schemes with two baseline methods: fine-tuning only biases of the last k layers; fine-tuning weights and biases of the last k layers (including fine-tuning the full model, when k equals to the total #layers). For each configuration, we measure the average accuracy on the 8 downstream datasets and the *analytic* extra memory usage. We also compare with a simple baseline by only fine-tuning the classifier. As shown in Figure 9, the accuracy of classifier-only update is low



(a) MbV2-w0.35 (b) Proxyless-w0.3 (c) MCUNet-5FPS

Figure 9. Sparse update can achieve higher transfer learning accuracy using 4.5-7.5× 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.



(a) Peak memory *vs.* models (b) Peak memory *vs.* schemes (c) Training latency *vs.* models

Figure 10. *Measured* peak memory and latency: **(a)** Sparse update with TTE graph optimization can reduce the measured peak memory by 20-21× 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× 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.

due to the limited learning capacity. *Updating the classifier alone is not enough; we also need to update the backbone.* Bias-only update outperforms classifier-only update but the accuracy quickly plateaus and does not improve even more biases are tuned. For updating last k layers, the accuracy generally goes higher as more layers are tuned; however, it has a very large memory footprint. Take MCUNet as an example, updating the last two blocks leads to an extra memory surpassing 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 last k layers, sparse update can achieve higher downstream accuracy with smaller memory footprint. We also measure the highest accuracy achievable by updating the last k layers (including fine-tuning the full model[§]) as the baseline upper bound (denoted as "upper bound"). Interestingly, our sparse update achieves a better downstream accuracy compared to the baseline best statistics. We hypothesize that the sparse update scheme alleviates over-fitting or makes momentum-free optimization easier.

Matching cloud training accuracy for tinyML. Remarkably, the downstream accuracy of our on-device training has *matched or even surpassed* the accuracy of cloud-trained results on tinyML application VWW [20]. Our framework uses 206KB *measured* SRAM while achieving 89.1% top-1 accuracy for on-device training (we used gradient accumulation for the VWW dataset; see the appendix Section C for details). The result is higher than the accuracy of the same model reported by the state-of-the-art solution MCUNet (88.7%, trained on cloud and deployed to MCU). Both settings transfer the ImageNet pre-trained model to VWW. The on-device accuracy is far above the common requirement for tinyML (>80% by MLPerf Tiny [8]) and surpassed the results of industry solution TF-Lite Micro+MobileNetV2 (86.2% [47] under 256KB, *inference-only, no training support*).

Tiny Training Engine: memory saving. We measure the training memory of three models on STM32F746 MCU to compare the memory saving from TTE. We measure the peak SRAM usage

[§]Note that fine-tuning the entire model does not always lead to the best accuracy. We grid search for the best k on Cars dataset: k =36 for MobileNetV2, 39 for ProxylessNAS, 12 for MCUNet, and apply it to all datasets.

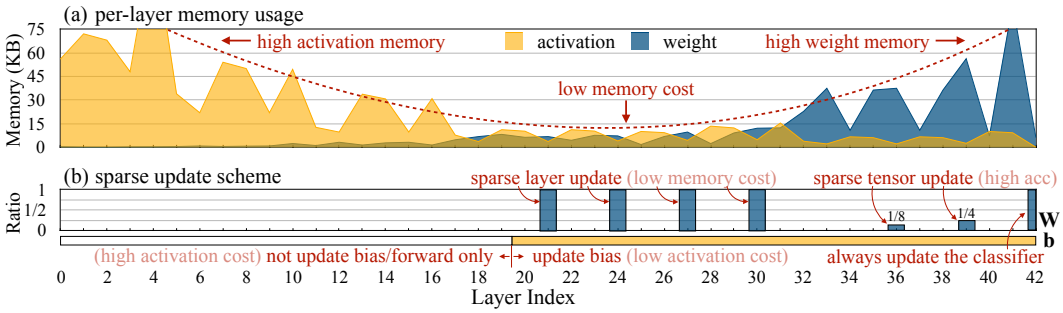


Figure 11. (a) The weight and activation memory cost of updating *each* layer of MCUNet (analytic). We find that the activation cost is high for the starting layers; the weight cost is high for the later layers; the overall memory cost is low for the middle layers. **(b)** Dissecting the sparse update scheme: we update the biases of the last 22 layers due to its low activation cost. For weight update, we update some middle layers due to its low memory cost, and update partial channels of the two later layers since they are important for accuracy (Figure 5).

under three settings: general full update, sparse update, and sparse update with TTE graph reordering (Figure 10(a)). The sparse update effectively reduces peak memory by 7-9× 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 9). The memory is further reduced with operator reordering, leading to 20-21× total memory saving. With both techniques, the training of all 3 models fits 256KB SRAM. We also compare the memory saving of reordering under different update schemes on MCUNet (Figure 9(b), indicated by different accuracy levels). Reordering consistently reduces the peak memory for different sparse update schemes of varying learning capacities.

Tiny Training Engine: faster training. We further measure the training latency per image on the STM32F746 MCU with three settings: full update with TF-Lite Micro kernels, sparse update with TF-Lite Micro kernels, and sparse update with TTE kernels (Figure 10(c)). Notice that TF-Lite *does not* support training; we just used the kernel implementation to measure latency. By graph optimization and 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× compared to the full update + TF-Lite Micro kernels, leading to energy saving and making training practical. Note that TF-Lite with full update leads to OOM, so we report the projected latency according to the average speed of each op type (marked in dashed columns).

3.3 Ablation Studies and Analysis

Dissecting update schedules. We visualize the update schedule of the MCUNet [47] model searched under 100KB extra memory (analytic) in Figure 11 (lower subfigure (b), with 10 classes). It updates the biases of the last 22 layers, and sparsely updates the weights of 6 layers (some are sub-tensor update). The initial 20 layers are frozen and run forward only. To understand why this scheme makes sense, we also plot the memory cost from activation and weight when updating *each* layer in the upper subfigure (a). We see a clear pattern: the activation cost is high for the initial layers; the weight cost is high for the ending layers; while the total memory cost is low when we update the middle layers (layer index 18-30). The update scheme matches the memory pattern: to skip the initial stage of high activation memory, we only update biases of the later stage of the network; we update the weights of 4 intermediate layers due to low overall memory cost; we also update the partial weights of two later layers (1/8 and 1/4 weights) due to their high contribution to the downstream accuracy (Figure 5). Interestingly, all the updated weights are from the first point-wise convolution in each inverted residual block [61] as they generally have a higher contribution to accuracy (the peak points on the zigzag curve in Figure 5(b)).

Effectiveness of contribution analysis. We verify if the update scheme search based on contribution analysis is effective. We collect several data points during the search process (the update scheme and the search criteria, *i.e.*, the sum of Δacc). We train the model with each update scheme to get the average accuracy on the downstream datasets (the real optimization target) and plot the comparison in Figure 5(c). We observe a positive correlation, indicating the effectiveness of the search.

Sub-channel selection. Similar to weight pruning, we need to select the subset of channels for sub-tensor update. We update the last two blocks of the MCUNet [47] model and only 1/4 of the weights for each layer to compare the accuracy of different channel selection methods (larger magnitude, smaller magnitude, and random). The results are quite similar (within 0.2% accuracy difference). Channel selection is not very important for transfer learning (unlike pruning). We choose to update the channels with a larger weight magnitude since it has slightly higher accuracy.

4 Related Work

Efficient transfer learning. There are several ways to reduce the transfer learning cost compared to fine-tuning the full model [38, 21, 37]. The most straightforward way is to only update the classifier layer [15, 23, 26, 62], but the accuracy is low when the domain shift is large [12]. Later studies investigate other tuning methods including updating biases [12, 71], updating normalization layer parameters [53, 25], updating small parallel branches [12, 32], *etc*. These methods only reduce the trainable parameter number but lack the study on system co-design to achieve real memory savings. Most of them do not fit tinyML settings (cannot handle quantized graph and lack of BatchNorm [33]).

Systems for deep learning. The success of deep learning is built on top of popular training frameworks such as PyTorch [56], TensorFlow [5], MXNet [16], JAX [10], *etc*. These systems usually depend on a host language (*e.g*. Python) and various runtime, which brings significant overhead (>300MB) and does not fit tiny edge devices. Inference libraries like TVM [17], TF-Lite [3], NCNN [1], TensorRT [2], and OpenVino [66] provide lightweight runtime environments but do not support training; MNN [35] has a preliminary support for full model training but cannot fit tiny IoT devices. Recently, POET [57] utilizes rematerialization and paging to train on microcontrollers, but it relies on a large external memory.

Tiny deep learning on microcontrollers. Tiny deep learning on microcontrollers is challenging. Existing work explores model compression (pruning [29, 30, 48, 31, 50, 70, 45], quantization [29, 58, 67, 19, 60, 42, 47, 34]) and neural architecture search [72, 73, 65, 47, 7, 43, 24, 51, 47, 46] to reduce the required resource of deep learning models. There are several deep learning systems for tinyML (TF-Micro [5], CMSIS-NN [41], TinyEngine [47], MicroTVM [17], CMix-NN [14], *etc*.). However, the above algorithms and systems are only for inference but not training. There are several preliminary attempts to explore training on microcontrollers [59, 28, 64, 63]. However, due to the lack of efficient algorithm and system support, they are only able to tune one layer or a very small model, while our work supports the tuning of modern CNNs for real-life applications.

5 Conclusion

In this paper, we propose the first solution to enable tiny on-device training on microcontrollers under a tight memory budget of 256KB and 1MB Flash without auxiliary memory. Our algorithm system co-design solution significantly reduces the training memory (more than 1000× compared with PyTorch and TensorFlow) and per-iteration latency (more than 20× speedup over TensorFlow-Lite Micro), allowing us to obtain higher downstream accuracy. Our study suggests that tiny IoT devices can not only perform inference but also continuously adapt to new data for lifelong learning.

Limitations and societal impacts. Our work achieves the first practical solution for transfer learning on tiny microcontrollers. However, our current study is limited to vision recognition with CNNs. In the future, we would like to extend to more modalities (*e.g*., audio) and more models (*e.g*., RNNs, Transformers). Our study improves tiny on-device learning, which helps to protect the privacy on sensitive data (*e.g*., healthcare). However, to design and benchmark our method, we experimented on many downstream datasets, leading to a fair amount of electricity consumption.

Acknowledgments

We thank National Science Foundation (NSF), MIT-IBM Watson AI Lab, MIT AI Hardware Program, Amazon, Intel, Qualcomm, Ford, Google for supporting this research.

References

[1] Ncnn : A high-performance neural network inference computing framework optimized for mobile platforms. https://github.com/Tencent/ncnn.

[2] Nvidia tensorrt, an sdk for high-performance deep learning inference. https://developer.nvidia.com/tensorrt.

[3] Tensorflow lite. https://www.tensorflow.org/lite.

[4] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

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

[6] Byung Hoon Ahn, Jinwon Lee, Jamie Menjay Lin, Hsin-Pai Cheng, Jilei Hou, and Hadi Esmaeilzadeh. Ordering chaos: Memory-aware scheduling of irregularly wired neural networks for edge devices. *arXiv preprint arXiv:2003.02369*, 2020.

[7] Colby Banbury, Chuteng Zhou, Igor Fedorov, Ramon Matas, Urmish Thakker, Dibakar Gope, Vijay Janapa Reddi, Matthew Mattina, and Paul Whatmough. Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers. *Proceedings of Machine Learning and Systems*, 3, 2021.

[8] Colby R Banbury, Vijay Janapa Reddi, Max Lam, William Fu, Amin Fazel, Jeremy Holleman, Xinyuan Huang, Robert Hurtado, David Kanter, Anton Lokhmotov, et al. Benchmarking tinyml systems: Challenges and direction. *arXiv preprint arXiv:2003.04821*, 2020.

[9] Lukas Bossard, Matthieu Guillaumin, and Luc Van Gool. Food-101–mining discriminative components with random forests. In *European conference on computer vision*, pages 446–461. Springer, 2014.

[10] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[11] Alessio Burrello, Angelo Garofalo, Nazareno Bruschi, Giuseppe Tagliavini, Davide Rossi, and Francesco Conti. Dory: Automatic end-to-end deployment of real-world dnns on low-cost iot mcus. *IEEE Transactions on Computers*, 70(8):1253–1268, 2021.

[12] Han Cai, Chuang Gan, Ligeng Zhu, and Song Han. Tinytl: Reduce activations, not trainable parameters for efficient on-device learning. *arXiv preprint arXiv:2007.11622*, 2020.

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

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

[15] Ken Chatfield, Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Return of the devil in the details: Delving deep into convolutional nets. In *BMVC*, 2014.

[16] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

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

[18] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

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

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

[21] Yin Cui, Yang Song, Chen Sun, Andrew Howard, and Serge Belongie. Large scale fine-grained categorization and domain-specific transfer learning. In *CVPR*, 2018.

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

[23] Jeff Donahue, Yangqing Jia, Oriol Vinyals, Judy Hoffman, Ning Zhang, Eric Tzeng, and Trevor Darrell. Decaf: A deep convolutional activation feature for generic visual recognition. In *ICML*, 2014.

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

[25] Jonathan Frankle, David J Schwab, and Ari S Morcos. Training batchnorm and only batchnorm: On the expressive power of random features in cnns. *arXiv preprint arXiv:2003.00152*, 2020.

[26] Chuang Gan, Naiyan Wang, Yi Yang, Dit-Yan Yeung, and Alex G Hauptmann. Devnet: A deep event network for multimedia event detection and evidence recounting. In *CVPR*, pages 2568–2577, 2015.

[27] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[28] Marc Monfort Grau, Roger Pueyo Centelles, and Felix Freitag. On-device training of machine learning models on microcontrollers with a look at federated learning. In *Proceedings of the Conference on Information Technology for Social Good*, pages 198–203, 2021.

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

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

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

[32] Edward Hu, Yelong Shen, Phil Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021.

[33] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *ICML*, 2015.

[34] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2704–2713, 2018.

[35] Xiaotang Jiang, Huan Wang, Yiliu Chen, Ziqi Wu, Lichuan Wang, Bin Zou, Yafeng Yang, Zongyang Cui, Yu Cai, Tianhang Yu, et al. Mnn: A universal and efficient inference engine. *arXiv preprint arXiv:2002.12418*, 2020.

[36] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[37] Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Joan Puigcerver, Jessica Yung, Sylvain Gelly, and Neil Houlsby. Big transfer (bit): General visual representation learning. In *European conference on computer vision*, pages 491–507. Springer, 2020.

[38] Simon Kornblith, Jonathon Shlens, and Quoc V Le. Do better imagenet models transfer better? In *CVPR*, 2019.

[39] Jonathan Krause, Michael Stark, Jia Deng, and Li Fei-Fei. 3d object representations for fine-grained categorization. In *Proceedings of the IEEE international conference on computer vision workshops*, pages 554–561, 2013.

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

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

[42] Hamed F Langroudi, Vedant Karia, Tej Pandit, and Dhireesha Kudithipudi. Tent: Efficient quantization of neural networks on the tiny edge with tapered fixed point. *arXiv preprint arXiv:2104.02233*, 2021.

[43] Edgar Liberis, Łukasz Dudziak, and Nicholas D Lane. μnas: Constrained neural architecture search for microcontrollers. *arXiv preprint arXiv:2010.14246*, 2020.

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

[45] Edgar Liberis and Nicholas D Lane. Differentiable network pruning for microcontrollers. *arXiv preprint arXiv:2110.08350*, 2021.

[46] Ji Lin, Wei-Ming Chen, Han Cai, Chuang Gan, and Song Han. Mcunetv2: Memory-efficient patch-based inference for tiny deep learning. *arXiv preprint arXiv:2110.15352*, 2021.

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

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

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

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

[51] Bo Lyu, Hang Yuan, Longfei Lu, and Yunye Zhang. Resource-constrained neural architecture search on edge devices. *IEEE Transactions on Network Science and Engineering*, 2021.

[52] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, 2018.

[53] Pramod Kaushik Mudrakarta, Mark Sandler, Andrey Zhmoginov, and Andrew Howard. K for the price of 1: Parameter-efficient multi-task and transfer learning. In *ICLR*, 2019.

[54] Maria-Elena Nilsback and Andrew Zisserman. Automated flower classification over a large number of classes. In *2008 Sixth Indian Conference on Computer Vision, Graphics & Image Processing*, pages 722–729. IEEE, 2008.

[55] Omkar M Parkhi, Andrea Vedaldi, Andrew Zisserman, and CV Jawahar. Cats and dogs. In *2012 IEEE conference on computer vision and pattern recognition*, pages 3498–3505. IEEE, 2012.

[56] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[57] Shishir G Patil, Paras Jain, Prabal Dutta, Ion Stoica, and Joseph Gonzalez. Poet: Training neural networks on tiny devices with integrated rematerialization and paging. In *International Conference on Machine Learning*, pages 17573–17583. PMLR, 2022.

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

[59] Haoyu Ren, Darko Anicic, and Thomas A Runkler. Tinyol: Tinyml with online-learning on microcontrollers. In *2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2021.

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

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

[62] Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. Cnn features off-the-shelf: an astounding baseline for recognition. In *CVPR Workshops*, 2014.

[63] Bharath Sudharsan, John G Breslin, and Muhammad Intizar Ali. Globe2train: A framework for distributed ml model training using iot devices across the globe. In *2021 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/IOP/SCI)*, pages 107–114. IEEE, 2021.

[64] Bharath Sudharsan, Piyush Yadav, John G Breslin, and Muhammad Intizar Ali. Train++: An incremental ml model training algorithm to create self-learning iot devices. In *2021 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/IOP/SCI)*, pages 97–106. IEEE, 2021.

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

[66] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

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

[68] Peter Welinder, Steve Branson, Takeshi Mita, Catherine Wah, Florian Schroff, Serge Belongie, and Pietro Perona. Caltech-ucsd birds 200. Technical Report CNS-TR-201, Caltech, 2010.

[69] Yang You, Igor Gitman, and Boris Ginsburg. Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888*, 2017.

[70] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. Scalpel: Customizing dnn pruning to the underlying hardware parallelism. *ACM SIGARCH Computer Architecture News*, 45(2):548–560, 2017.

[71] Elad Ben Zaken, Shauli Ravfogel, and Yoav Goldberg. Bitfit: Simple parameter-efficient fine-tuning for transformer-based masked language-models. *CoRR*, abs/2106.10199, 2021.

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

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

A Video Demo

We prepared a video demo showing that we can deploy our framework to a microcontroller (STM32F746, 320KB SRAM, 1MB Flash) to enable on-device learning. We adapt the MCUNet model (pre-trained on ImageNet) to classify whether there is a person in front of the camera or not. The training leads to decent accuracy within the tight memory budget. Please find the demo here: https://youtu.be/XaDCO8YtmBw.

The training is performed with 100 sample images from the VWW dataset [20] fed through the camera (50 positive and 50 negative). The total (pure) training throughput for the pipeline (including overheads like camera IO) is shown in the Figure. 12. The total training time would be around minutes. This is quite affordable for tiny on-device learning applications.

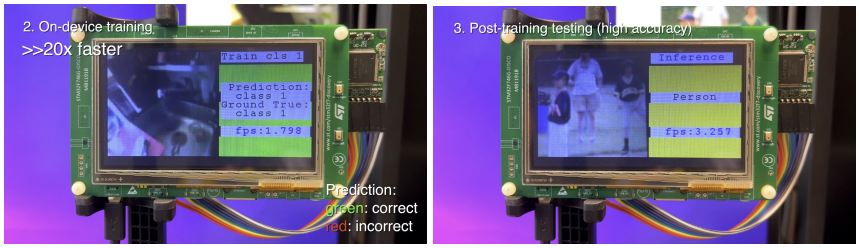


Figure 12. A screenshot of our video demo.

B Variance of Different Runs

We notice that the variance of different runs is quite small in our experiments. Here we provide detailed information about the variance.

Firstly, if we use the same random seed for the data loader, we will get *exactly the same* results for multiple runs. The weight quantization process after each iteration (almost) eliminates the non-determinism from GPU training[¶]. Therefore, we study the randomness from different random seeds in *data shuffling*. Here we provide the results of 3 runs in Table 2 to show the variance. We train the MobileNetV2-w0.35 model with the sparse update scheme (searched under 100KB analytic memory usage) 3 times independently. We find the variance is very small, especially when we report the average accuracy (for most of our results): the standard derivation is only $\pm 0.07\%$.

Table 2. The variance between different runs is small, especially when we report the average accuracy (only $\pm 0.07\%$). Results obtained by training MobileNetV2-w0.35 for three times using the sparse update scheme searched under 100KB analytic memory constraint.

Runs	Accuracy (%)								Avg Acc.
	Cars	CF10	CF100	CUB	Flowers	Food	Pets	VWW	
run1	51.59	87.03	63.89	54.14	85.95	62.28	77.84	88.34	71.38
run2	52.87	86.8	63.81	54.87	85.30	62.45	77.30	88.65	71.50
run3	52.49	87.13	63.80	55.16	85.35	61.99	77.08	88.21	71.40
mean	52.32	86.99	63.83	54.72	85.53	62.24	77.41	88.40	**71.43**
±std	±0.66	±0.17	±0.05	±0.52	±0.36	±0.23	±0.39	±0.22	**±0.07**

C Training Setups & Discussions

In this section, we introduce detailed training setups and discuss the reasons that lead to several design choices.

We used SGD optimizer+QAS for training. We set weight decay as 0 since we observed no over-fitting during experiments. This is also a common choice in transfer learning [37]. We find the initial learning rate significantly affects the accuracy, so we extensively tuned the learning rate for each run to report the best accuracy. We used cosine learning rate decay and performed warm-up [27] for 1 epoch on VWW and 5 epochs on other datasets. We used Ray [52] for experiment launching and hyper-parameter tuning.

[¶] https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9911-determinism-in-deep-learning.pdf

Data type of the classifier. During transfer learning, we usually need to randomly initialize the classifiers (or add some classes) for novel categories. Although the backbone is fully quantized for efficiency, we find that using a floating-point classifier is essential for transfer learning performance. Using a floating-point classifier is also cost-economical since the classifier consists of a very small part of the model size (0.3% for 10 classes).

We compare the results of the quantized classifier and floating-point classifier in Table 3. We update the last two blocks of the MCUNet model with SGD-M optimizer and QAS to measure the downstream accuracy. We find that keeping the classifier as floating-point significantly improves the downstream accuracy by 2.3% (on average) at a marginal overhead. *Therefore, we use floating-point for the classifier by default.*

Table 3. Keeping the classifier as floating-point greatly improves the downstream accuracy.

`fp32` classifier	Accuracy (%)								Avg Acc.
	Cars	CF10	CF100	CUB	Flowers	Food	Pets	VWW	
✗	50.8	86.1	62.7	56.8	82.5	61.7	80.8	87.8	71.2
✓	55.2	86.9	64.6	57.8	89.1	64.4	80.9	89.3	**73.5**

Single-batch training & momentum. For on-device training on microcontrollers, we can only fit batch size 1 due to the tight memory constraint. However, single-batch training has very low efficiency when simulated on GPUs since it cannot leverage the hardware parallelism, making experiments slow. We study the performance gap between single-batch training and normal-batch training (batch size 128) to see if we can use the latter as an approximation.

We compare the results of different batch sizes in Table 4, with and without momentum. Due to the extremely low efficiency of single-batch training, we only report results on datasets of a smaller size. We used SGD+QAS as the optimizer and updated the last two blocks of the MCUNet [47] model. We extensively tuned the initial learning rate to report the best results.

Table 4. Momentum helps transfer learning with batch size 128, but not with batch size 1; without momentum, we can use the normal-batch training results as an approximation for single-batch training. Results obtained by updating the last two blocks of MCUNet [47] with SGD+QARS.

Batch size	Momentum	Mem Cost	Accuracy (%)					Avg Acc.
			Cars	CUB	Flowers	Pets	VWW	
128 (GPU simulate)	0.9	2×	55.2	57.8	89.1	80.9	89.3	74.4
	0	1×	47.8	57.2	87.3	80.8	88.8	72.4
1 (tinyML)	0.9	2×	47.8	54.8	88.5	80.5	86.2	71.5
	0	1×	51.1	56.2	88.7	79.3	86.0	72.3

We can make two observations:

1. Firstly, momentum helps optimization for normal-batch training as expected (average accuracy 74.4% *vs*. 72.4%). However, it actually makes the accuracy slightly worse for the single-batch setting (71.5% *vs*. 72.3%). Since using momentum will double the memory requirement for updating parameters (assume we can safely quantize momentum buffer; otherwise the memory usage will be 5× larger), we will not use momentum for tinyML on-device learning.
2. Without momentum, normal-batch training, and single-batch training lead to a similar average accuracy (72.4% *vs*. 72.3%), allowing us to use batched training results for evaluation.

Given the above observation, *we report the results of batched training without momentum by default*, unless otherwise stated.

Gradient accumulation. With the above training setting, we can get a similar average accuracy compared to actual on-device training on microcontrollers. The reported accuracy on each dataset is quite close to the real on-device accuracy, with *only one exception*: the VWW dataset, where the accuracy is 2.5% lower. This is because VWW only has two categories (binary classification), so the information from each label is small, leading to unstable gradients. For the cases where the number of categories is small, we can add gradient accumulation to make the update more stable. We show the comparison of adapting the pre-trained MCUNet model in Table 5. The practice closes the accuracy gap at a small extra memory cost (11%), allowing us to get 89.1% top-1 accuracy within 256KB memory usage.

To provide a clear comparison, we *do not* apply gradient accumulation in our experiments except for this comparison.

Table 5. Gradient accumulation helps the optimization on datasets with a small category number. Numbers obtained by training with batch size 1, the same setting as on microcontrollers.

model	accumulate grad	SRAM	VWW accuracy
MCUNet-5FPS	✗	160KB	86.6%
	✓	188KB	89.1%

D Evolutionary Search *vs*. Random Search

We find that evolutionary search can efficiently explore the search space to find a good sparse update scheme given a memory constraint. Here we provide the comparison between evolutionary search and random search in Figure 13. We collect the curves when searching for an update scheme of the MCUNet-5FPS [47] model under 100KB memory constraint (analytic). We find that evolutionary search has a much better sample efficiency and can find a better final solution (higher sum of Δacc) compared to random search. The search process is quite efficient: we can search for a sparse update scheme within 10 minutes based on the contribution information. Note that we use the *same* update scheme for all downstream datasets.

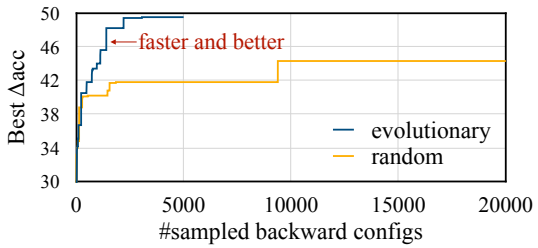


Figure 13. Evolutionary search has a better sample efficiency and leads to a better final result compared with random search when optimizing sparse update schemes.

E Amount of Compute

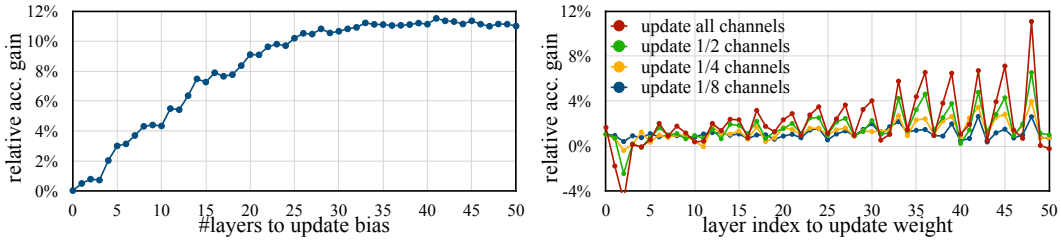
To evaluate the performance of different training schemes, we simulate the training on GPUs to measure the average accuracy on 8 downstream datasets. Thanks to the small model size (for the tinyML setting) and the small dataset size, the training cost for each scheme is quite modest: it only takes **3.2 GPU hours** for training on all 8 downstream datasets (cost for one run; do not consider hyper-parameter tuning).

For the pre-training on ImageNet [22], it takes about **31.5 GPU hours** (300 epochs). Note that we only need to pre-train each model *once*.

We performed training with NVIDIA GeForce RTX 3090 GPUs.

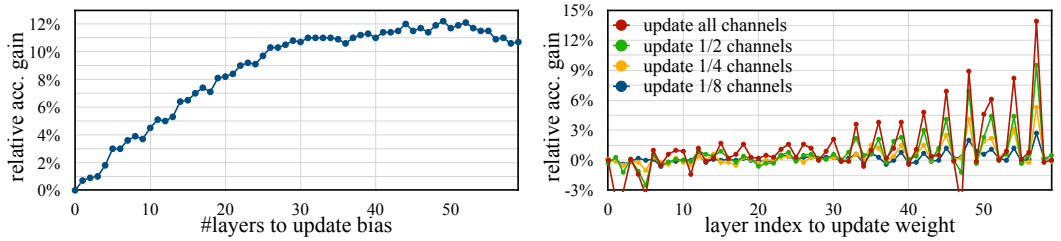
F More Contribution Analysis Results

Here we provide the contribution analysis results of the MobileNetV2-w0.35 [61] and ProxylessNAS-w0.3 [13] on the Cars dataset [39] (Figure 14 and 15). The pattern is similar to the one from the MCUNet model: the later layers contribute to the accuracy improvement more; within each block, the first point-wise convolutional layer contributes to the accuracy improvement the most.



(a) Contribution of last k biases $\Delta\text{acc}_{\mathbf{b}_{[:k]}}$ (b) Contribution of a certain weight $\Delta\text{acc}_{\mathbf{W}_{i,r}}$

Figure 14. Contribution analysis of updating biases and weights for MobileNetV2-w0.35 [61].



(a) Contribution of last k biases $\Delta\text{acc}_{\mathbf{b}_{[:k]}}$ (b) Contribution of a certain weight $\Delta\text{acc}_{\mathbf{W}_{i,r}}$

Figure 15. Contribution analysis of updating biases and weights for ProxylessNAS-w0.3 [13].

G Other Partial Update Methods That Did Not Work

During our experiments, we also considered other efficient partial update methods (apart from sparse layer/tensor update) but they did not work well. Here are a few methods we tried but failed:

1. Low-rank update. LoRA [32] aims to adapt a model by adding a low-rank decomposed weight to each of the original weight matrix. It is designed for adapting large language models, but could potentially be applied here. Specifically, LoRA freezes the original weight $\mathbf{W} \in \mathbb{R}^{c \times c}$ but trains a small $\Delta\mathbf{W} = \mathbf{MN}$, where $\mathbf{M} \in \mathbb{R}^{c \times c'}, \mathbf{N} \in \mathbb{R}^{c' \times c}, c' << c$. The low-rank decomposed $\Delta\mathbf{W}$ has much fewer parameters compared to $\mathbf{W}$. After training, we can merge $\Delta\mathbf{W}$ so that no extra computation is incurred: $\mathbf{y} = (\mathbf{W} + \Delta\mathbf{W})\mathbf{x}$. However, such method does not work in our case:

1. The weights are quantized in our models. If we merge $\Delta\mathbf{W}$ and $\mathbf{W}$, we will produce a new weight $\mathbf{W}' = \Delta\mathbf{W} + \mathbf{W}$ that has the same size as $\mathbf{W}$, taking up a large space on the SRAM (that is why we need the sparse tensor update).
2. Even if we can tolerate the extra memory overhead by running $\mathbf{y} = \mathbf{Wx} + \Delta\mathbf{Wx}$, the $\Delta\mathbf{W}$ is randomly initialized and we empirically find that it is difficult to update a quantized weight from scratch, leading to worse performance.

2. Replacing convolutions with lighter alternatives. As shown in the contribution curves (Figure 4 in the main paper, Figure 14, and Figure 15), the first point-wise convolutional layer in each block has the highest contribution to accuracy. We tried replacing the first point-wise convolutional layer with a lighter alternative, like grouped convolutions. However, although such replacement greatly reduces the cost to update the layers, it also hinders transfer learning accuracy significantly. Therefore, we did not choose to use such modification. It also involves extra complexity by changing model architectures, which is not desired.

H Changelog

v1 Initial preprint release.

v2 Fix a typo in Equation 4.

v3 Camera-ready version.

v4 Update project and demo links.