Tiny Transfer Learning: Towards Memory-Efficient On-Device Learning

Han Cai[1], **Chuang Gan**[2], **Ligeng Zhu**[1], **Song Han**[1]
[1]Massachusetts Institute of Technology, [2]MIT-IBM Watson AI Lab

Abstract

We present *Tiny-Transfer-Learning* (TinyTL), an efficient on-device learning method to adapt pre-trained models to newly collected data on edge devices. Different from conventional transfer learning methods that fine-tune the full network or the last layer, TinyTL freezes the weights of the feature extractor while only learning the biases, thus doesn't require storing the intermediate activations, which is the major memory bottleneck for on-device learning. To maintain the adaptation capacity without updating the weights, TinyTL introduces memory-efficient lite residual modules to refine the feature extractor by learning small residual feature maps in the middle. Besides, instead of using the same feature extractor, TinyTL adapts the architecture of the feature extractor to fit different target datasets while fixing the weights: TinyTL pre-trains a large super-net that contains many weight-shared sub-nets that can individually operate; different target dataset selects the sub-net that best match the dataset. This backpropagation-free discrete sub-net selection incurs no memory overhead. Extensive experiments show that TinyTL can reduce the training memory cost by order of magnitude (up to **13.3**×) without sacrificing accuracy compared to fine-tuning the full network.

1 Introduction

Intelligent edge devices with rich sensors (e.g., billions of mobile phones and IoT devices)[1] have been ubiquitous in our daily lives. These devices keep collecting *new* and *sensitive* data through the sensor every day while being expected to provide high-quality and customized services without sacrificing privacy[2]. This requires the AI systems to have the ability to continually adapt pre-trained models to these newly collected data without leaking them to the cloud (i.e., on-device learning).

While there is plenty of efficient inference techniques that have significantly reduced the parameter size and the computation FLOPs [2, 3, 18, 19, 21, 22, 40, 42, 47, 51], the size of intermediate activations, required by back-propagation, causes a huge training memory footprint (Figure 1 left), making it difficult to train on edge devices.

First, edge devices are memory-constrained. For example, a Raspberry Pi 1 Model A only has 256MB of memory, sufficient for the inference. However, as shown in Figure 1 (left, red line), the memory footprint of the training phase can easily exceed this limit, even using a lightweight neural network architecture (MobileNetV2 [40]). Furthermore, the memory is shared by various on-device applications (e.g., other deep learning models) and the operating system. A single application may only be allocated a small fraction of the total memory, which makes this challenge more critical. Second, edge devices are energy-constrained. Under the 45nm CMOS technology [19], a 32bit off-chip DRAM access consumes 640 pJ, which is two orders of magnitude larger than a 32bit on-chip SRAM access (5 pJ) or a 32bit float multiplication (3.7 pJ). The large memory footprint

[1] https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/
[2] https://ec.europa.eu/info/law/law-topic/data-protection_en

Preprint. Under review.

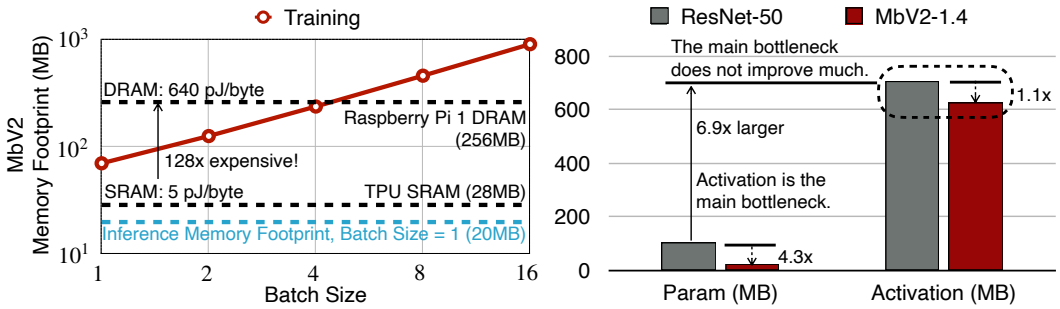

Figure 1: *Left*: The memory footprint required by training grows linearly w.r.t. the batch size and soon exceeds the limit of edge devices. *Right*: Memory cost comparison between ResNet-50 and MobileNetV2-1.4 under batch size 8. Recent advances in efficient model design only reduce the size of parameters, but activation size, the main bottleneck for training, does not improve much.

required by training cannot fit into the limited on-chip SRAM. For instance, TPU [27] only has 28MB of SRAM that is far smaller than the training memory footprint of MobileNetV2, even using a batch size of 1 (Figure 1 left). It results in many costly DRAM accesses and thereby consumes lots of energy, draining the battery of edge devices.

In this work, we propose *Tiny-Transfer-Learning* (TinyTL) to address these challenges. By analyzing the memory footprint during the backward pass, we notice that the intermediate activations (the main bottleneck) are only involved in updating the weights; updating the biases does not require them (Eq. 2). Inspired by this finding, we propose to freeze the weights of the pre-trained feature extractor to reduce the memory footprint (Figure 2b). To compensate for the capacity loss due to freezing weights while keeping the memory overhead small, we introduce *lite residual learning* that improves the model capacity by learning lite residual modules to refine the intermediate feature maps of the pre-trained feature extractor (Figure 2c). Meanwhile, it aggressively shrinks the resolution dimension and width dimension of the lite residual modules to have a small memory overhead. We also empirically find that different transfer datasets require very different feature extractors, especially when the weights are frozen (Figure 3). Therefore, we introduce *feature extractor adaptation* to update the architecture of the feature extractor while fixing the weights to fit different target datasets (Figure 2d). Concretely, we select different sub-nets from a large pre-trained super-net. Different from conventional approaches that fix the architecture and update the weights in the continuous optimization space, our approach optimizes the feature extractor in the discrete space, which does not require any back-propagation and thus do not incur additional memory overhead. Extensive experiments on transfer learning datasets demonstrate that TinyTL achieves the same level (or even higher) accuracy then fine-tuning the full network while reducing the training memory footprint by up to **13.3**×. Our contributions can summarized as follows:

- We propose TinyTL, a novel transfer learning method for memory-efficient on-device learning. To the best of our knowledge, this is the first work that tackles this challenging but critical problem.
- We systematically analyze the memory bottleneck of training and find the heavy memory cost comes from updating the weights, not biases (assume ReLU activation).
- We propose two novel techniques (*lite residual learning* and *feature extractor adaptation*) to improve the model capacity while freezing the weights with little memory overhead.
- Extensive experiments on transfer learning tasks show that our method is highly memory-efficient and effective. It reduces the training memory footprint by up to 13.3×, making it possible to learn on memory-constrained edge devices (e.g., Raspberry Pi) without sacrificing accuracy.

2 Related Work

Efficient Inference Techniques. Improving the inference efficiency of deep neural networks on resource-constrained edge devices has recently drawn extensive attention. Starting from [10, 14, 18, 19, 44], one line of research focuses on compressing pre-trained neural networks, including i) network pruning that removes less-important units [12, 19] or channels [20, 33]; ii) network quantization that reduces the bitwidth of parameters [7, 18] or activations [26, 45]. However, these techniques cannot handle the training phase, as they rely on a well-trained model on the target task as the starting point.

Another line of research focuses on lightweight neural architectures by either manual design [22, 23, 24, 40, 51] or neural architecture search [1, 4, 42, 47]. These lightweight neural networks provide highly competitive accuracy [2, 43] while significantly improving inference efficiency. However, concerning the training memory efficiency, key bottlenecks are not solved: the training memory is dominated by activations, not parameters. For example, Figure 1 (right) shows the cost comparison between ResNet-50 and MobileNetV2-1.4. In terms of parameter size, MobileNetV2-1.4 is 4.3× smaller than ResNet-50. However, in terms of the training activation size, MobileNetV2-1.4 is almost the same as ResNet-50 (only 1.1× smaller), leading to little memory footprint reduction.

Training Memory Cost Reduction. Researchers have been seeking ways to reduce the training memory footprint. One typical approach is to re-compute discarded activations during backward [6, 16]. This approach reduces memory usage at the cost of a large computation overhead. Thus it is not preferred for edge devices. Layer-wise training [15] can also reduce the memory footprint compared to end-to-end training. However, it cannot achieve the same level of accuracy as end-to-end training. Another representative approach is through activation pruning [32], which builds a dynamic sparse computation graph to prune activations during training. Similarly, [46] proposes to reduce the bitwidth of training activations by introducing new reduced-precision floating-point formats. Different from these techniques that prune or quantize existing networks with a given architecture, we can adapt the architecture to different datasets, and our method is orthogonal to these techniques.

Transfer Learning. Neural networks pre-trained on large-scale datasets (e.g., ImageNet [9]) are widely used as a fixed feature extractor for transfer learning, then only the last layer needs to be fine-tuned [5, 11, 13, 41]. This approach does not require to store the intermediate activations of the feature extractor, and thus is memory-efficient. However, the capacity of this approach is limited, resulting in poor accuracy, especially on datasets [30, 35] whose distribution is far from ImageNet (e.g., only 45.9% Aircraft top1 accuracy achieved by Inception-V3 [36]). Alternatively, fine-tuning the full network can achieve better accuracy [8, 29]. But it requires a vast memory footprint and hence is not friendly for training on edge devices. Recently, [37] proposes to reduce the number of trainable parameters by only updating parameters of the batch normalization (BN) [25] layers. Unfortunately, parameter-efficiency doesn't translate to memory-efficiency. It still requires a large amount of memory (e.g., 326MB under batch size 8) to store the input activations of the BN layers (Table 1). Additionally, the accuracy of this approach is still much worse than fine-tuning the full network (70.7% v.s. 85.5%; Table 1). People can also partially fine-tune some layers, but how many layers to select is still ad hoc. This paper provides a systematic approach to adapt the feature extractor to different datasets and use lite residual learning to save memory.

3 Method

3.1 Understanding the Memory Footprint of Back-propagation

Without loss of generality, we consider a neural network $\mathcal{M}$ that consists of a sequence of layers:

$$\mathcal{M}(\cdot) = \mathcal{F}_{\mathbf{w}_n}(\mathcal{F}_{\mathbf{w}_{n-1}}(\cdots \mathcal{F}_{\mathbf{w}_2}(\mathcal{F}_{\mathbf{w}_1}(\cdot))\cdots)), \tag{1}$$

where $\mathbf{w}_i$ denotes the parameters of the i^{th} layer. Let $\mathbf{a}_i$ and $\mathbf{a}_{i+1}$ be the input and output activations of the i^{th} layer, respectively, and $\mathcal{L}$ be the loss. In the backward pass, given $\frac{\partial \mathcal{L}}{\partial \mathbf{a}_{i+1}}$, there are two goals for the i^{th} layer: computing $\frac{\partial \mathcal{L}}{\partial \mathbf{a}_i}$ and $\frac{\partial \mathcal{L}}{\partial \mathbf{w}_i}$.

Assuming the i^{th} layer is a linear layer whose forward process is given as: $\mathbf{a}_{i+1} = \mathbf{a}_i\mathbf{W} + \mathbf{b}$, then its backward process under batch size 1 is

$$\frac{\partial \mathcal{L}}{\partial \mathbf{a}_i} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}_{i+1}}\frac{\partial \mathbf{a}_{i+1}}{\partial \mathbf{a}_i} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}_{i+1}}\mathbf{W}^T, \qquad \frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \mathbf{a}_i^T\frac{\partial \mathcal{L}}{\partial \mathbf{a}_{i+1}}, \qquad \frac{\partial \mathcal{L}}{\partial \mathbf{b}} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}_{i+1}}. \tag{2}$$

According to Eq. (2), the intermediate activations (i.e., $\{\mathbf{a}_i\}$) that dominate the memory footprint are only required to compute the gradient of the weights (i.e., $\frac{\partial \mathcal{L}}{\partial \mathbf{W}}$), not the bias. If we only update the bias, training memory can be greatly saved. This property is also applicable to convolution layers and normalization layers (e.g., batch normalization [25], group normalization [48], etc) since they can be considered as special types of linear layers.

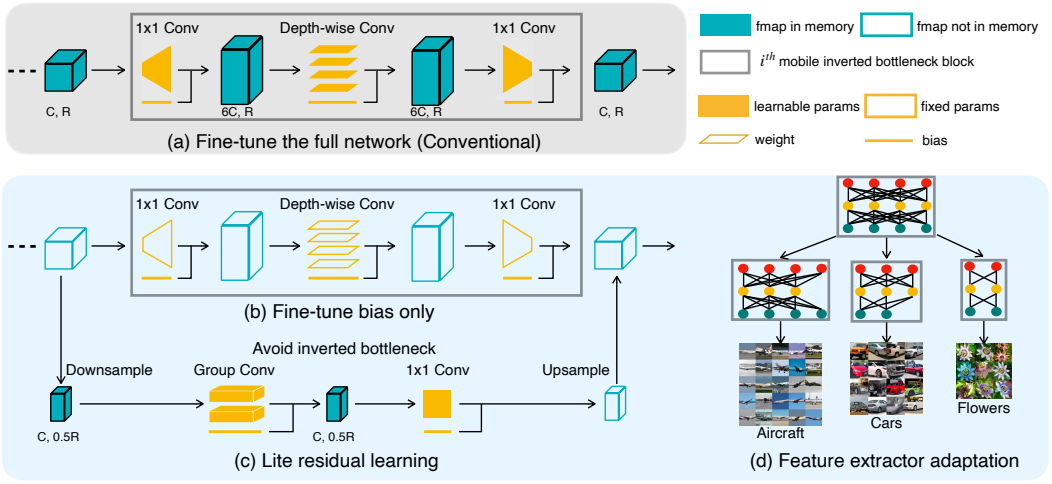

Figure 2: TinyTL overview ("C" denotes the width and "R" denote the resolution). Conventional transfer learning fixes the architecture of the feature extractor and relies on fine-tuning the weights to adapt the model (Fig.a), which requires a large amount of activation memory (in blue) for back-propagation. TinyTL reduces the memory usage by fixing the weights while: (Fig.b) only fine-tuning the bias. (Fig.c) exploit *lite residual learning* to compensate for the capacity loss, using group convolution and avoiding inverted bottleneck to achieve high arithmetic intensity and small memory footprint. (Fig.d) adapting the feature extractor architecture to different downstream tasks, which can specialize a small feature extractor for an easy dataset (Flowers), and a large feature extractor for a difficult dataset (Aircraft). Their weights are shared from the same super-net, which is also parameter-efficient.

Regarding non-linear activation layers (e.g., ReLU, sigmoid, h-swish)[3], sigmoid and h-swish require to store $\mathbf{a}_i$ to compute $\frac{\partial \mathcal{L}}{\partial \mathbf{a}_i}$, hence they are not memory-efficient. Activation layers that build upon them are also not memory-efficient consequently, such as tanh, swish [39], etc. In contrast, ReLU and other ReLU-styled activation layers (e.g., LeakyReLU [49]) only requires to store a binary mask representing whether the value is smaller than 0, which is 32× smaller than storing $\mathbf{a}_i$.

3.2 Tiny Transfer Learning

Based on the memory footprint analysis, one possible solution of reducing the memory cost is to freeze the weights of the pre-trained feature extractor while only update the biases (Figure 2b). However, only updating biases has limited adaptation capacity. In this section, we explore two optimization techniques to improve the model capacity without updating weights: i) lite residual modules to refine the intermediate feature maps (Figure 2c); ii) feature extractor adaptation to enable specialized feature extractors that best match different transfer datasets (Figure 2d).

3.2.1 Lite Residual Learning

Formally, a layer with frozen weights and learnable biases can be represented as:

$$\mathbf{a}_{i+1} = \mathcal{F}_{\mathbf{W}}(\mathbf{a}_i) + \mathbf{b}. \tag{3}$$

To improve the model capacity while keeping a small memory footprint, we propose to add a lite residual module that generates a residual feature map to refine the output:

$$\mathbf{a}_{i+1} = \mathcal{F}_{\mathbf{W}}(\mathbf{a}_i) + \mathbf{b} + \mathcal{F}_{\mathbf{w}_r}(\mathbf{a}'_i = \text{reduce}(\mathbf{a}_i)), \tag{4}$$

where $\mathbf{a}'_i = \text{reduce}(\mathbf{a}_i)$ is the reduced activation. According to Eq. (2), learning these lite residual modules only requires to store the reduced activations $\{\mathbf{a}'_i\}$ rather than the full activations $\{\mathbf{a}_i\}$.

Implementation (Figure 2c). We apply Eq. (4) to mobile inverted bottleneck blocks (MB-block) [40]. The key principle is to keep the activation small. Following this principle, we explore two design dimensions to reduce the activation size:

[3]Detailed forward and backward processes of the activation layers are provided in Appendix D.

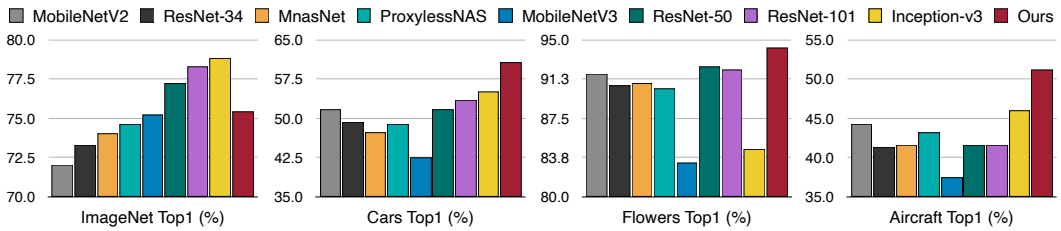

Figure 3: Transfer learning performances of various ImageNet pre-trained models with the last linear layer trained. The relative accuracy order between different pre-trained models changes significantly among ImageNet and the transfer learning datasets. For example, our specialized feature extractors (red) consistently achieve the best results on transfer datasets, though having weaker ImageNet accuracy. It suggests that we need to *adapt* the feature extractor to fit different transfer datasets instead of using the *same* one for all datasets.

- **Width.** The widely-used inverted bottleneck requires a huge number of channels ($6\times$) to compensate for the small capacity of a depthwise convolution, which is parameter-efficient but highly activation-inefficient. Even worse, converting $1\times$ channels to $6\times$ channels back and forth requires two 1×1 projection layers, which doubles the total activation to $12\times$. Depthwise convolution also has a very low arithmetic intensity (its OPs/Byte is less than 4% of 1×1 convolution's OPs/Byte if with 256 channels), thus highly memory in-efficient with little reuse. To solve these limitations, our lite residual module employs the group convolution (g=2) that has $300\times$ higher arithmetic intensity than depthwise convolution, providing a good trade-off between FLOPs and memory. That also removes the 1×1 projection layer, reducing the total channel number by $6 \times 2 = 12\times$.
- **Resolution.** The activation size grows quadratically with the resolution. Therefore, we aggressively shrink the resolution in the lite residual module by employing a 2×2 average pooling to downsample the input feature map. The output of the lite residual module is then upsampled to match the size of the main branch's output feature map via bilinear upsampling. Combining resolution and width optmizations, the activation of our lite residual module is $2 \times 2 \times 12 = 48\times$ smaller than the inverted bottleneck.

3.2.2 Feature Extractor Adaptation

Motivation. Conventional transfer learning chooses the feature extractor according to higher pre-training accuracy (e.g., ImageNet accuracy) and uses the same one for all transfer tasks [8, 37]. However, we find this approach sub-optimal, since different target tasks may need very different feature extractors and high pre-training accuracy does not guarantee good transferability of the pre-trained weights. This is especially critical in our case where the weights are frozen.

Figure 3 shows the top1 accuracy of various widely used ImageNet pre-trained models on three transfer datasets by only learning the last layer, which reflects the transferability of their pre-trained weights. The relative order between different pre-trained models is not consistent with their ImageNet accuracy on all three datasets. This result indicates that the ImageNet accuracy is not a good proxy for transferability. Besides, we also find that the same pre-trained model can have very different rankings on different tasks. For instance, Inception-V3 gives poor accuracy on Flowers but provides top results on the other two datasets. Therefore, we need to specialize the feature extractor to best match the target dataset.

Implementation (Figure 2d). Motivated by these observations, we propose to adapt the feature extractor for different transfer tasks. This is achieved by allowing a set of candidate weight operations instead of using a fixed weight operation:

$$\{\mathcal{M}(\cdot)\} = \mathcal{F}_{\{\mathbf{w}_n^1,\cdots,\mathbf{w}_n^m\}}(\cdots\mathcal{F}_{\{\mathbf{w}_2^1,\cdots,\mathbf{w}_2^m\}}(\mathcal{F}_{\{\mathbf{w}_1^1,\cdots,\mathbf{w}_1^m\}}(\cdot))\cdots). \tag{5}$$

It forms a discrete optimization space, allowing us to adapt the feature extractor for different target datasets without updating the weights. The detailed training flow is described as follows:

- **Pre-training**. The size of all possible weight operation combinations is exponentially large w.r.t. the depth, making it computationally impossible to pre-train all of them independently. Therefore, we employ the weight sharing technique [4, 31, 50] to reduce the pre-training cost, where a single super-net is jointly optimized on the pre-training dataset (e.g., ImageNet) to support all

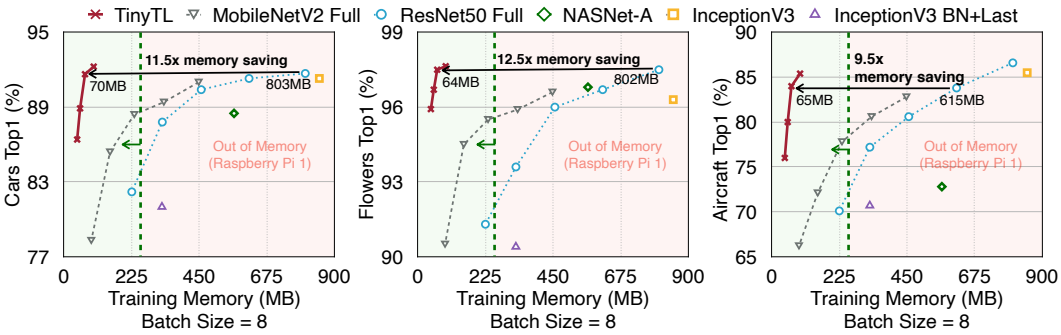

Figure 4: Results under different resolutions. With the same level of accuracy, TinyTL provide an order of magnitude training memory saving compared to fine-tuning the full ResNet-50, making it possible to learning on-device for Raspberry Pi 1.

possible sub-nets (i.e., different combinations of weight operations). Different sub-nets can operate independently by selecting different parts from the super-net. For example, centered weights in the full convolution kernels are taken to form smaller convolution kernels; blocks are skipped to form a sub-net with a lower depth; channels are skipped to reduce the width of a convolution operation. In our experiments, we use ImageNet as the pre-training dataset. We employ progressive shrinking [2, 50] for training the super-net, using the same training setting suggested by [2].

- **Fine-tuning the super-net.** We fine-tune the pre-trained super-net on the target transfer dataset with the weights of the main branches (i.e., MB-blocks) frozen and the other parameters (i.e., biases, lite residual modules, classification head) updated via gradient descent. In this phase, we randomly sample one sub-net in each training step.
- **Discrete operation search.** Based on the fine-tuned super-net, we collect 450 [sub-net, accuracy] pairs on the validation set (20% randomly sampled training data) and train an accuracy predictor[4] using the collected data [2]. We employ evolutionary search [17] based on the accuracy predictor to find the sub-net (i.e., the combination of weight operations) that best matches the target transfer dataset. No back-propagation on the super-net is required in this step, thus incurs no additional memory overhead.
- **Final fine-tuning.** Finally, we fine-tune the searched model with the weights of the main branches frozen and the other parameters updated (i.e., biases, lite residual modules, classification head), using the full training set to get the final results.

4 Experiments

Following the common practice [8, 29, 37], we evaluate our TinyTL on three benchmark datasets including Cars [30], Flowers [38], and Aircraft [35], using ImageNet [9] as the pre-training dataset.

Model Architecture. We build the super-net using the MobileNetV2 design space [4, 42] that contains five stages with a gradually decreased resolution, and each stage consists of a sequence of MB-blocks. In the stage-level, it supports different depths (i.e., 2, 3, 4). In the block-level, it supports different kernel sizes (i.e., 3, 5, 7) and different width expansion ratios (i.e., 3, 4, 6)[5]. For each MB-block, we insert a lite residual module as described in Section 3.2.1 and Figure 2 (c). The group number = 2, and the kernel size = 5. We use the ReLU activation since it is more memory-efficient according to Section 3.1.

Training Details. We freeze the weights of the feature extractors while allowing biases to be updated during transfer learning. Both the lite residual learning (LiteResidual; Section 3.2.1) and feature extractor adaptation (FeatureAdapt; Section 3.2.2) are applied in our experiments. For fine-tuning the pre-trained super-net, we use the Adam optimizer [28] with an initial learning rate of 4e-3 following the cosine learning rate decay [34]. The model is trained on 80% randomly sampled training data for 50 epochs. For fine-tuning the searched model, we use the same training setting

[4]Details of the accuracy predictor is provided in Appendix E.

[5]The detailed architecture of the super-net is provided in Appendix C.

Table 1: Comparison with conventional transfer learning methods. * indicates our re-implemented results. "R" denotes the input image size. TinyTL reduces the training memory by **13.3** × without sacrificing accuracy compared to fine-tuning the full Inception-V3.

Method		Flowers (Batch Size = 8) Train. Mem.	Flowers (Batch Size = 8) Reduce Rate	Cars Top1 (%)	Flowers Top1 (%)	Aircraft Top1 (%)
Last	Inception-V3 [37]	94MB	1.0×	55.0	84.5	45.9
	ResNet-50*	76MB	1.2×	51.6	92.4	41.5
TinyTL	FeatureAdapt (FA)	**41MB**	**2.3**×	**60.7**	**94.2**	**51.5**
BN+ Last	ResNet-50*	391MB	1.0×	80.1	95.4	72.2
	Inception-V3 [37]	326MB	1.2×	81.0	90.4	70.7
	MobileNetV2*	224MB	1.7×	77.5	95.0	65.5
TinyTL	LiteResidual (R=320)	70MB	5.6×	**89.4**	**96.9**	**81.5**
	LiteResidual (R=224)	**40MB**	**9.8**×	85.5	96.2	75.6
Full	Inception-V3 [8]	850MB	1.0×	91.3	96.3	85.5
	ResNet-50 [29]	802MB	1.1×	91.7	97.5	86.6
	MobileNetV2-1.4 [29]	644MB	1.3×	91.8	97.5	86.8
	NASNet-A [29]	566MB	1.5×	88.5	96.8	72.8
TinyTL	FA + LiteResidual (R=320)	92MB	9.2×	**92.3**	**97.6**	85.4
	FA + LiteResidual (R=256)	**64MB**	**13.3**×	91.6	97.5	84.0

but on the full training data. Additionally, we apply 8bits weight quantization [18] on the frozen weights to reduce the parameter size, which causes a negligible accuracy drop in our experiments. For all compared methods, we also assume the 8bits weight quantization is applied if eligible when calculating their training memory footprint.

Main Results. Table 1 reports the comparison between TinyTL and previous transfer learning methods that are divided into three groups, including: i) fine-tuning the last linear layer [5, 11, 41] (referred as *Last*); ii) fine-tuning the BN layers and the last linear layer [36] (referred as *BN+Last*) ; iii) fine-tuning the full network [8, 29] (referred as *Full*).

In the first group, we only apply FeatureAdapt to adapt the feature extractor while only training the parameters of the last linear layer, similar to *Last*. Compared to *Last*+Inception-V3, our model reduces the training memory cost by **2.3**× while improving the top1 accuracy by **5.7%** on Cars, **9.7%** on Flowers, and **5.6%** on Aircraft. It shows our specialized feature extractors can better fit different transfer datasets than these fixed feature extractors. In the second group, we only apply LiteResidual to refine the intermediate feature maps using Proxyless-Mobile [4] as the feature extractor. Compared to *BN+Last* with ResNet-50, our model improves the training memory efficiency by **9.8**× while providing consistently better accuracy (**5.4%** higher on Cars, **0.8%** higher on Flowers, and **3.4%** higher on Aircraft). By increasing the input image size from 224 to 320, we can further increase the accuracy improvement from **5.4% to 9.3%** on Cars, from **0.8% to 1.5%** on Flowers, from **3.4% to 9.3%** on Aircraft, which shows that learning lite residual modules and biases is not only more memory-efficient but also more effective than *BN+Last*. In the third group, we apply both FeatureAdapt and LiteResidual. Compared to *Full*+Inception-V3, TinyTL can achieve the same level of accuracy while providing **13.3**× training memory saving, reducing the training memory from 850MB to 64MB (the same level as only learning the last linear layer).

Figure 4 demonstrates the results under different input resolution. With similar accuracy, TinyTL provides an order of magnitude memory reduction (**11.5**× on Cars, **12.5**× on Flowers, and **9.5**× on Aircraft) compared to fine-tuning the full ResNet-50. **Remarkably, it moves the training memory cost from the out-of-memory region (red) to the feasible region (green) on Raspberry Pi 1, making it possible to learn on-device without sacrificing accuracy.**

4.1 Ablation Studies and Discussions

Comparison with Dynamic Activation Pruning. The comparison between TinyTL and dynamic activation pruning [32] is summarized in Figure 5. TinyTL is more effective because it re-designed the transfer learning architecture (lite residual module, feature extractor adaptation) rather than prune an existing architecture. The transfer accuracy drops quickly when the pruning ratio increases beyond

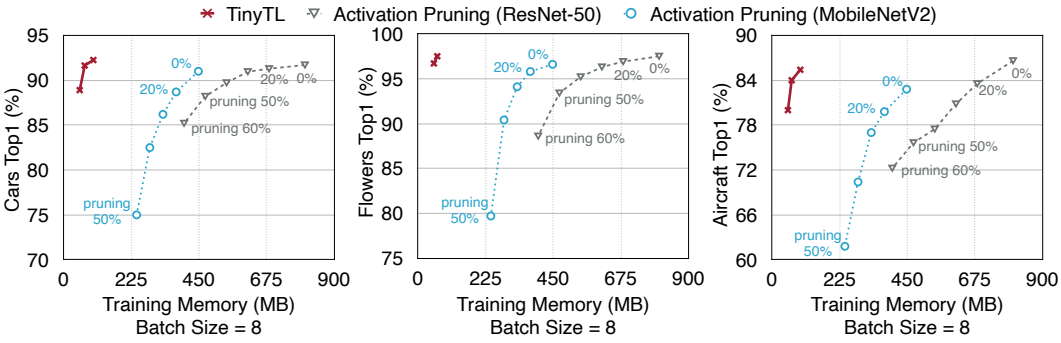

Figure 5: Compared with dynamic activation pruning [32], TinyTL saves the memory more effectively.

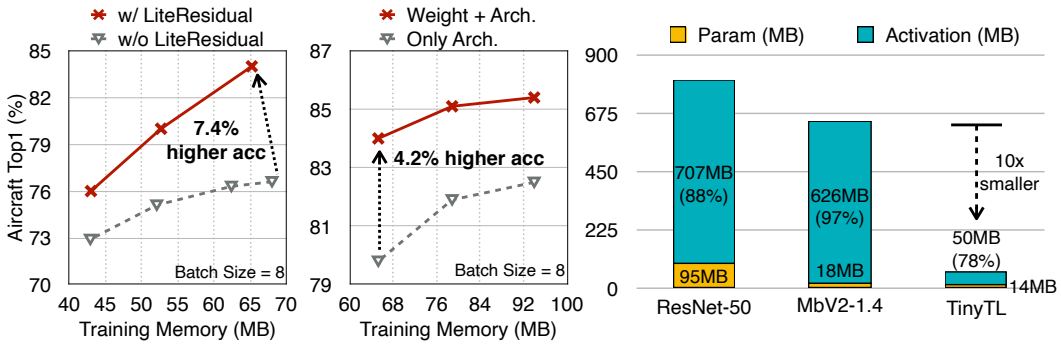

Figure 6: *Left & Middle:* Ablation Studies of TinyTL on Aircraft. *Right:* TinyTL reduces both the parameter size and the activation size, providing a more balanced cost composition than previous efficient inference techniques that focus on reducing the parameter size.

50% (only 2× memory saving). In contrast, TinyTL can achieve much higher memory reduction without loss of accuracy.

Effectiveness of LiteResidual. Figure 6 (left) shows the results of TinyTL with and without LiteResidual (only bias) on Aircraft, where we can observe significant accuracy drops (up to 7.4%) if disabling the lite residual modules.

Pre-trained Weight Matters, Not Only Architecture. Figure 6 (middle) reports the performance of TinyTL if retraining the searched feature extractor on ImageNet (only arch). The retrained feature extractor cannot reach the same accuracy compared to keeping both the pre-trained weight and the architecture. It suggests that not only the architecture of the feature extractor matters, the pre-trained weight also contributes a lot to the final performance.

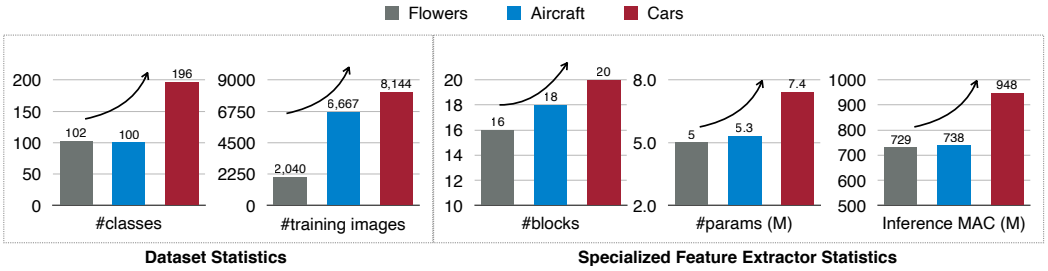

Figure 7: TinyTL can adapt the feature extractor's architecture to different transfer datasets.

Adapt the Feature Extractor to Different Transfer Datasets. Figure 7 reports the details of the transfer learning datasets and the corresponding feature extractors specialized for these datasets in TinyTL. For an easier dataset such as Flowers (fewer #classes, fewer #training images), TinyTL

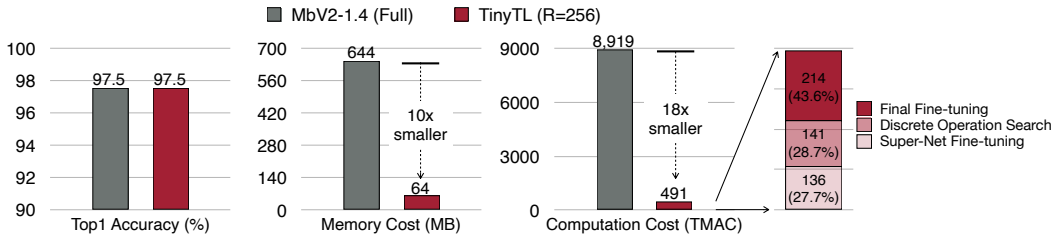

Figure 8: On-device training cost on Flowers. Achieving the same accuracy, TinyTL requires 10× smaller memory cost and 18× smaller computation cost compared to fine-tuning the full MobileNetV2-1.4 [29].

chooses a smaller feature extractor (fewer blocks, fewer parameters, less computation). For a more difficult dataset like Cars (more #classes, more #training images), TinyTL chooses a larger feature extractor (more blocks, more parameters, more computation).

Cost Details. As shown in Figure 6 (right), TinyTL reduces both the parameter size and the activation size instead of only reducing the parameter size as previous efficient inference methods did, hence provides a more balanced cost composition. This activation size is the peak activation size during the three on-device phases (Section 3.2.2), including fine-tuning the super-net, discrete operation search, and final fine-tuning. Concretely, for each layer, we compute the size of already stored activations (required by back-propagation), the size of already stored binary masks (required by ReLU layers), and the size of buffers (required by the forward process). The peak value of their sum across all layers is taken as the peak activation size.

The on-device training cost is summarized in Figure 8. TinyTL reduces the training memory by 10×, and reduces the training computation by 18×, achieving the same accuracy as fine-tuning the full MobileNetV2-1.4. The peak memory cost of TinyTL under resolution 256 is 64MB while the total MAC is 491T. In contrast, fine-tuning the full network requires 644MB and the total MAC is 8,919T (20,000 steps with batch size 256 [29])[6]. TinyTL is not only much more memory-efficient but also much more computation-efficient.

5 Conclusion

We proposed Tiny-Transfer-Learning (TinyTL) for memory-efficient on-device learning that aims to adapt pre-trained models to newly collected data on edge devices. Unlike previous transfer learning methods that fix the architecture and fine-tune the weights to fit different target datasets, TinyTL fixes the weights while adapting the architecture of the feature extractor and learning memory-efficient lite residual modules and biases to fit different target datasets. Extensive experiments on benchmark transfer learning datasets consistently show the effectiveness and memory-efficiency of TinyTL, paving the way for efficient on-device machine learning.

References

[1] Han Cai, Tianyao Chen, Weinan Zhang, Yong Yu, and Jun Wang. Efficient architecture search by network transformation. In *AAAI*, 2018. 3

[2] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once for all: Train one network and specialize it for efficient deployment. In *ICLR*, 2020. 1, 3, 6

[3] Han Cai, Ji Lin, Yujun Lin, Zhijian Liu, Kuan Wang, Tianzhe Wang, Ligeng Zhu, and Song Han. Automl for architecting efficient and specialized neural networks. *IEEE Micro*, 2019. 1

[4] Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct neural architecture search on target task and hardware. In *ICLR*, 2019. 3, 5, 6, 7

[5] Ken Chatfield, Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Return of the devil in the details: Delving deep into convolutional nets. In *BMVC*, 2014. 3, 7

[6] We report the memory cost under batch size 8 for consistency, which does not change the reduction ratio.

[6] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016. 3

[7] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *NeurIPS*, 2015. 2

[8] Yin Cui, Yang Song, Chen Sun, Andrew Howard, and Serge Belongie. Large scale fine-grained categorization and domain-specific transfer learning. In *CVPR*, 2018. 3, 5, 6, 7

[9] 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. 3, 6

[10] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *NeurIPS*, 2014. 2

[11] 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. 3, 7

[12] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *ICLR*, 2019. 2

[13] 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. 3

[14] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014. 2

[15] Klaus Greff, Rupesh K Srivastava, and Jürgen Schmidhuber. Highway and residual networks learn unrolled iterative estimation. *arXiv preprint arXiv:1612.07771*, 2016. 3

[16] Audrunas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. Memory-efficient backpropagation through time. In *NeurIPS*, 2016. 3

[17] Zichao Guo, Xiangyu Zhang, Haoyuan Mu, Wen Heng, Zechun Liu, Yichen Wei, and Jian Sun. Single path one-shot neural architecture search with uniform sampling. *arXiv preprint arXiv:1904.00420*, 2019. 6

[18] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *ICLR*, 2016. 1, 2, 7

[19] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *NeurIPS*, 2015. 1, 2

[20] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *ICCV*, 2017. 2

[21] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for mobilenetv3. In *ICCV*, 2019. 1, 13

[22] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017. 1, 3, 13

[23] Gao Huang, Shichen Liu, Laurens Van der Maaten, and Kilian Q Weinberger. Condensenet: An efficient densenet using learned group convolutions. In *CVPR*, 2018. 3

[24] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016. 3

[25] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, 2015. 3

[26] 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 *CVPR*, 2018. 2

[27] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *ISCA*, 2017. 2

[28] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 6

[29] Simon Kornblith, Jonathon Shlens, and Quoc V Le. Do better imagenet models transfer better? In *CVPR*, 2019. 3, 6, 7, 9

[30] 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*, 2013. 3, 6

[31] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. In *ICLR*, 2019. 5

[32] Liu Liu, Lei Deng, Xing Hu, Maohua Zhu, Guoqi Li, Yufei Ding, and Yuan Xie. Dynamic sparse graph for efficient deep learning. In *ICLR*, 2019. 3, 7, 8

[33] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *ICCV*, 2017. 2

[34] Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016. 6

[35] Subhransu Maji, Esa Rahtu, Juho Kannala, Matthew Blaschko, and Andrea Vedaldi. Fine-grained visual classification of aircraft. *arXiv preprint arXiv:1306.5151*, 2013. 3, 6

[36] 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. 3, 7

[37] 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. 3, 5, 6, 7

[38] Maria-Elena Nilsback and Andrew Zisserman. Automated flower classification over a large number of classes. In *Sixth Indian Conference on Computer Vision, Graphics & Image Processing*, 2008. 6

[39] Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. In *ICLR Workshop*, 2018. 4

[40] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR*, 2018. 1, 3, 4, 12, 13

[41] Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. Cnn features off-the-shelf: an astounding baseline for recognition. In *CVPR Workshops*, 2014. 3, 7

[42] 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. 1, 3, 6

[43] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *ICML*, 2019. 3

[44] Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. Improving the speed of neural networks on cpus. In *NeurIPS Deep Learning and Unsupervised Feature Learning Workshop*, 2011. 2

[45] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. Haq: Hardware-aware automated quantization. In *CVPR*, 2019. 2

[46] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. In *NeurIPS*, 2018. 3

[47] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *CVPR*, 2019. 1, 3

[48] Yuxin Wu and Kaiming He. Group normalization. In *ECCV*, 2018. 3

[49] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*, 2015. 4

[50] Jiahui Yu, Pengchong Jin, Hanxiao Liu, Gabriel Bender, Pieter-Jan Kindermans, Mingxing Tan, Thomas Huang, Xiaodan Song, Ruoming Pang, and Quoc Le. Bignas: Scaling up neural architecture search with big single-stage models. *arXiv preprint arXiv:2003.11142*, 2020. 5, 6

[51] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *CVPR*, 2018. 1, 3

A Detailed Architectures of Specialized Feature Extractors

(a) Feature extractor on Flowers

(b) Feature extractor on Aircraft

(c) Feature extractor on Cars

Figure 9: Detailed architectures of the feature extractors on different transfer datasets. "LR" denotes the lite residual module (Section 3.2.1) while "MB4 7 × 7" denotes the mobile inverted bottleneck block [40] with expansion ratio 4 and kernel size 7. TinyTL adapts a higher-capacity feature extractor for a harder task (Cars).

B Details of the On-device Training Cost

The detailed training cost of the on-device learning phases is described as follows:

- **Fine-tuning the super-net.** We fine-tune the pre-trained super-net under resolution 224. The peak memory cost of this phase is 64MB, which is reached when the largest sub-net is sampled. Regarding the computation cost, the average MAC (forward & backward)[7] of sampled sub-nets is (802M + 2535M) / 2 = 1668.5M per sample, where 802M is the training MAC of the smallest sub-net and 2535M is the training MAC of the largest sub-net. Therefore, the total MAC of this phase is 1668.5M × 2040 × 0.8 × 50 = 136T (27.7% of 491T) on Flowers, where 2040 is the number of total training samples, 0.8 means the super-net is fine-tuned on 80% of the training samples (the remaining 20% is reserved for search), and 50 is the number of training epochs.
- **Discrete operation search.** As discussed in Appendix E, the memory overhead and computation overhead of the accuracy predictor are negligible. The primary memory cost and computation cost of this phase come from collecting 450 [sub-net, accuracy] pairs required to train the accuracy predictor. It only involves the forward processes of sampled sub-nets, and no back-propagation is required. Therefore, the memory overhead of this phase is negligible compared to the super-net fine-tuning phase. The average MAC (only forward) of sampled sub-nets is (352M + 1179M) / 2 = 765.5M per sample, where 352M is the inference MAC of the smallest sub-net and 1179M is the inference MAC of the largest sub-net. Therefore, the total MAC of this phase is 765.5M × 2040 × 0.2 × 450 = 141T (28.7% of 491T) on Flowers, where 2040 is the number of total training samples,

[7] In the super-net fine-tuning phase, the training MAC of a sampled sub-net is roughly 2× larger than its inference MAC, rather than 3×, since we do not need to update the weights of the main branches.

0.2 means the validation set consists of 20% of the training samples, and 450 is the number of measured sub-nets.

- **Final fine-tuning.** To achieve the same accuracy as fine-tuning the full MobileNetV2-1.4, we use a resolution of 256. The memory cost of this phase is 63.9MB and the total MAC is 2100M $\times$ 2040 $\times$ 1.0 $\times$ 50 = 214T (43.6% of 491T), on Flowers, where 2100M is the training MAC, 2040 is the number of total training samples, 1.0 means the full training set is used, and 50 is the number of training epochs.

C Detailed Architecture of the Super-Net

Table 2: Detailed architecture of the super-net using the MobileNetV2 design space with lite residual modules (Section 3.2.1). "SepConv" denotes the separable convolution block [22] that consists of a depthwise-separable convolution layer and a 1×1 convolution layer. "MB-LiteResidual" denotes the mobile inverted bottleneck block [40] with a lite residual module (described in Section 3.2.1).

Input	Operator	#Out	Stride	LiteResidual Stride	Kernel Size	Expand Ratio	Repeat
$224^2 \times 3$	Conv2d	40	2	4	3	-	1
$112^2 \times 40$	SepConv	24	1	2	3	1	1
$112^2 \times 24$	MB-LiteResidual	32	2	4	$3,5,7$	$3,4,6$	1
$56^2 \times 32$	MB-LiteResidual	32	1	2	$3,5,7$	$3,4,6$	$1,2,3$
$56^2 \times 32$	MB-LiteResidual	56	2	4	$3,5,7$	$3,4,6$	1
$28^2 \times 56$	MB-LiteResidual	56	1	2	$3,5,7$	$3,4,6$	$1,2,3$
$28^2 \times 56$	MB-LiteResidual	104	2	4	$3,5,7$	$3,4,6$	1
$14^2 \times 104$	MB-LiteResidual	104	1	2	$3,5,7$	$3,4,6$	$1,2,3$
$14^2 \times 104$	MB-LiteResidual	128	1	2	$3,5,7$	$3,4,6$	1
$14^2 \times 128$	MB-LiteResidual	128	1	2	$3,5,7$	$3,4,6$	$1,2,3$
$14^2 \times 128$	MB-LiteResidual	248	2	4	$3,5,7$	$3,4,6$	1
$7^2 \times 248$	MB-LiteResidual	248	1	2	$3,5,7$	$3,4,6$	$1,2,3$
$7^2 \times 248$	MB-LiteResidual	416	1	2	$3,5,7$	$3,4,6$	1
$7^2 \times 416$	Conv2d	1664	1	-	1	-	1
$7^2 \times 1664$	Avg-pool	1664	7	-	7	-	1
$1^2 \times 1664$	Linear	1000	-	-	-	-	1

D Memory Footprint of Non-Linear Activation Layers

Table 3: Detailed forward and backward processes of non-linear activation layers. $|\mathbf{a}_i|$ denotes the number of elements of $\mathbf{a}_i$. "$\circ$" denotes the element-wise product. $(\mathbf{1}_{\mathbf{a}_i \geq 0})_j = 0$ if $(\mathbf{a}_i)_j < 0$ and $(\mathbf{1}_{\mathbf{a}_i \geq 0})_j = 1$ otherwise. $\text{ReLU6}(\mathbf{a}_i) = \min(6, \max(0, \mathbf{a}_i))$.

Layer Type	Forward	Backward	Memory Cost
ReLU	$\mathbf{a}_{i+1} = \max(0, \mathbf{a}_i)$	$\frac{\partial \mathcal{L}}{\partial \mathbf{a}_i} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}_{i+1}} \circ \mathbf{1}_{\mathbf{a}_i \geq 0}$	$\lvert\mathbf{a}_i\rvert$ bits
sigmoid	$\mathbf{a}_{i+1} = \sigma(\mathbf{a}_i) = \frac{1}{1+\exp(-\mathbf{a}_i)}$	$\frac{\partial \mathcal{L}}{\partial \mathbf{a}_i} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}_{i+1}} \circ \sigma(\mathbf{a}_i) \circ (1 - \sigma(\mathbf{a}_i))$	32 $\lvert\mathbf{a}_i\rvert$ bits
h-swish [21]	$\mathbf{a}_{i+1} = \mathbf{a}_i \circ \frac{\text{ReLU6}(\mathbf{a}_i+3)}{6}$	$\frac{\partial \mathcal{L}}{\partial \mathbf{a}_i} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}_{i+1}} \circ (\frac{\text{ReLU6}(\mathbf{a}_i+3)}{6} + \mathbf{a}_i \circ \frac{\mathbf{1}_{-3 \leq \mathbf{a}_i \leq 3}}{6})$	32 $\lvert\mathbf{a}_i\rvert$ bits

E Details of the Accuracy Predictor

The accuracy predictor is a three-layer feed-forward neural network with a hidden dimension of 400 and ReLU as the activation function for each layer. It takes the one-hot encoding of the sub-net's architecture as the input and outputs the predicted accuracy of the given sub-net. The inference MAC of this accuracy predictor is only 0.37M, which is 3-4 orders of magnitude smaller than the inference MAC of the CNN classification models. The memory footprint of this accuracy predictor is only 5KB. Therefore, both the computation overhead and the memory overhead of the accuracy predictor are negligible.