Cache and Reuse: Rethinking the Efficiency of On-device Transfer Learning

Yuedong Yang Hung-Yueh Chiang Guihong Li Diana Marculescu Radu Marculescu
Chandra Family Department of Electrical and Computer Engineering
The University of Texas at Austin
{albertyoung, hungyueh.chiang, lgh, dianam, radum}@utexas.edu

Abstract

Training only the last few layers in deep neural networks has been considered an effective strategy for enhancing the efficiency of on-device training. Prior work has adopted this approach and focused on accelerating backpropagation. However, by conducting a thorough system-wide analysis, we discover that the primary bottleneck is actually the forward propagation through the frozen layers, rather than backpropagation, if only the last few layers are trained. To address this issue, we introduce the "cache and reuse" idea for on-device transfer learning and propose a two-stage training method, which consists of a cache initialization stage, where we store the output from the frozen layers, followed by a training stage. To make our approach practical, we also propose augmented feature caching and cache compression to address the challenges of non-cacheable feature maps and cache size explosion. We carry out extensive experiments on various models (e.g., convolutional neural network and vision transformers) using real edge devices to demonstrate the effectiveness of our method. As an example, on NVIDIA Jetson Orin NX with MobileNet-V2, our approach boosts the training speed by $6.6\times$*, and improves the accuracy by 2.1%. For EfficientNet-b0, our method increases the training speed by* $2.2\times$ *and improves its accuracy by 1.3%. Therefore, our approach represents a significant improvement in enabling practical on-device transfer learning for edge devices with limited resources.*

1. Introduction

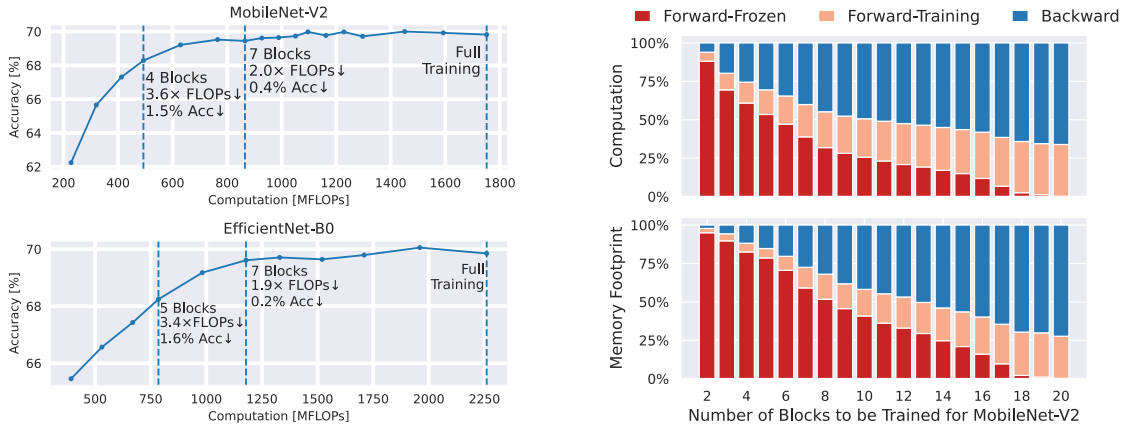
On-device transfer learning serves as the foundation for numerous resource-limited machine learning applications [1, 7, 9, 21, 24, 29, 37, 41]. For instance, in federated learning [7, 29, 41], to safeguard users' privacy, training is restricted to an individual's personal device, which typically has limited computational and energy resources [20, 33, 38]. In such situations, training efficiency is of paramount importance. Indeed, users may be willing to spend 10 minutes and use 5% of their battery energy to achieve a 70% accuracy improvement on their smartphones, but may decline to invest 10 hours and consume 90% of their battery power for a model with 75% accuracy, despite the marginally higher accuracy.

In fact, as shown in Figure 1a, we observe a diminishing rate of return in transfer learning-based neural network adaptation: the marginal accuracy gain obtained by training one additional parameter, decreases as more parameters are trained. This finding implies that training only the last few layers (closer to the output) and freezing the earlier layers is a more beneficial design choice for on-device learning. Some previous works have implicitly incorporated this approach in their designs [12, 25, 27, 39, 40]. For example, [25] trains only a few selected convolutional filters among the last layers, while keeping the other parameters frozen; similarly, [39] trains the last 2 or 4 layers in ResNets [13]. We note that these methods primarily concentrate on the efficiency of the backpropagation (BP) process during training, while overlooking the end-to-end efficiency; we argue that they should actually consider both forward and backward propagation instead. Indeed, to further improve the efficiency of on-device training, we ask two research questions:

(i) ***What*** *causes the key bottleneck for on-device training if only the last few layers are trained?*

(ii) ***How*** *can we improve the end-to-end efficiency if only the last few layers are trained?*

To tackle the first research question, we perform an end-to-end analysis to identify the unnecessary computations in the traditional "single-stage" training loop, which encompasses data processing, forward propagation, and backward propagation. We discover that, when training only the last few layers, the forward propagation becomes the primary bottleneck instead of BP (which is the constraint when training the entire model). To address the second research question, we introduce a new two-stage training method to further improve end-to-end efficiency. Drawing inspiration from modern memory hierarchy design used in computer systems, we propose to alleviate the bottleneck by caching and reusing the output from the frozen sub-network. How-



(a) (b)

Figure 1. (a) Accuracy on VTAB-1K when finetuning varying numbers of layers in MobileNet-V2 and EfficientNet-B0. (b) Analysis of computation and memory footprint for Steps (2-4) in Algorithm 1 when training different numbers of layers in MobileNet-V2 using single-stage transfer learning. For many cases, forward propagation through the frozen sub-network is the key bottleneck instead of BP.

ever, due to the inherent randomness and large data size, simply storing the output is not feasible. Therefore, we also propose new augmented feature caching and cache compression techniques to make the feature caching idea truly practical. In summary, our contributions are as follows:

- We identify the key computational bottleneck for on-device training when only the last few layers are trained. Contrary to popular belief that BP is the main bottleneck, we discover that the true bottleneck lies in the forward propagation through the frozen sub-network.
- We propose a new two-stage training method that substantially improves the end-to-end on-device training efficiency when only the last few layers are trained.
- We propose new augmented feature caching and cache compression techniques to solve the issue of non-cacheable feature maps and cache size explosion.
- We conduct extensive experiments with various neural networks (*i.e.*, convolutional neural network and vision transformer) on a real edge device, and demonstrate the effectiveness of our proposed method. For example, our approach accelerates training for MobileNet-V2 by $6.6\times$ and improves the testing accuracy by 2.1%.

The paper is organized as follows. Section 2 introduces the motivation for efficient on-device transfer learning. Section 3 formulates the on-device training problem. Section 4 presents our method in detail. Experimental results are provided in Section 5. Section 6 discusses related prior works. Finally, Section 7 summarizes our main contributions.

2. Background: Why training only the last few layers?

The idea of training only layers close to the network output has been commonly used in previous on-device transfer learning studies [12, 25, 27, 39, 40]. There are two main reasons for this:

Diminishing Returns in Transfer Learning: Figure 1a shows the accuracy when finetuning a variable number of layers for ImageNet [32]-pretrained neural networks [33, 36]. For both models, a generally concave type of accuracy-computation curve is observed, indicating strong diminishing returns. For instance, by spending around 500 MFLOPs to train 4 blocks in MobileNet-V2, we can achieve an 68.3% accuracy or 0.14% accuracy gain per MFLOP. However, training the full model with 1258 MFLOPs yields only a 1.5% increase in accuracy and a very low efficiency of only 0.04% accuracy gain per MFLOP. This suggests that training only the last few layers often results in better efficiency.

Limited Computation for Training: For gradient-based optimizers, such as stochastic gradient optimizers [2], the BP algorithm is employed to compute the gradient for each weight in the neural network. BP calculates gradients layer-by-layer, starting from the model's output towards its inputs. With a fixed budget, we may opt to halt BP early, thus resulting in training only the last few layers.

Although the strategy of training just the last few layers can reduce computation for BP and enhance the efficiency of on-device training, as shown in Figure 1b and later explained in Section 4.1, we find that under the conventional single-stage training framework, numerous redundant calculations lead to both computation and memory waste. We address this issue next by proposing a new two-stage on-device transfer learning framework.

3. Problem Formulation

We assume that a deep neural network $f(x;\Theta)$, which has inputs x and a set of weights Θ, can be divided

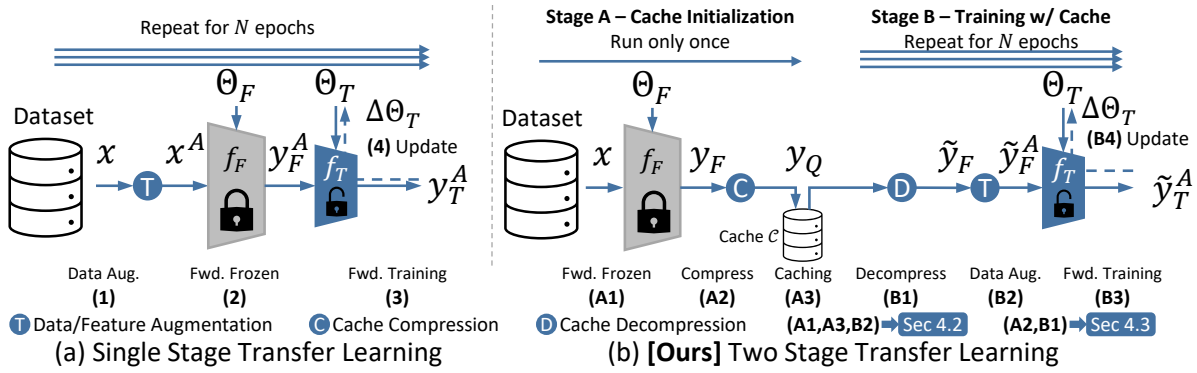


Figure 2. Workflow of (a) conventional single-stage transfer learning and (b) our proposed two-stage transfer learning. Symbols and steps correspond to Algorithms 1 and 2. Essentially, we speed up the training by caching and reusing the output from the frozen sub-network. To make two-stage traing viable and practical, we also introduce augmented feature caching (Step A1, A3, B2, Section 4.2) and cache compression techniques (Step A2, B1, Section 4.3).

into two nested sub-networks: f_F with weights Θ_F, and f_T with weights Θ_T. In other words, $f(x;\Theta) = f_T(f_F(x;\Theta_F);\Theta_T)$.

Since most state-of-the-art neural networks are built by stacking basic building blocks (*e.g.*, the residual block in ResNet), this assumption is valid in general [13, 26, 33, 36]. Under this assumption, $f_T(\cdot;\Theta_T)$ and $f_F(\cdot;\Theta_F)$ represent the sub-networks with blocks closer to the output and input, respectively. Then, with data augmentation method $T(\cdot)$, dataset $(\mathcal{X},\mathcal{Y})$ and optimizer optim$(\cdot)$, the vanilla single-stage on-device transfer learning for N epochs can be written as in Algorithm 1.

Algorithm 1 Training Process for Vanilla Single Stage On-device Transfer Learning

for epoch in $1...N$ **do** ▷ **Training for N epochs**
  for $(x,\hat{y}) \in (\mathcal{X},\mathcal{Y})$ **do**
    ***Step (1)**: Data augmentation*
    $x^A \leftarrow T(x)$
    ***Step (2)**: Forward for frozen sub-network*
    $y_F^A \leftarrow f_F(x^A;\Theta_F)$
    ***Step (3)**: Forward for training sub-network*
    $y_T^A \leftarrow f_T(y_F^A;\Theta_T)$
    ***Step (4)**: Back propagation and weight update*
    $\Delta\Theta_T \leftarrow \text{optim}(\Theta_T, y_T^A, \hat{y}, y_F^A)$
    $\Theta_T \leftarrow \Theta_T + \Delta\Theta_T$
  end for
end for

For every sample in dataset, we first perform data augmentation. Next, in Steps (2-3), we pass the input through both frozen and training sub-networks. Finally, in Step (4), we perform BP and update the weights in the training sub-network. This process is repeated for N epochs.

4. A New Two-Stage On-device Transfer Learning Approach

In this section, we present our two-stage on-device transfer learning method. First, we identify performance bottlenecks in the standard single-stage training method when only the last few layers are trained, as shown in Figure 2a. To address these bottlenecks, we then suggest a two-stage on-device transfer learning approach, as shown in Figure 2b. The main idea involves *caching and reusing* the output y_F^A created by the frozen sub-network f_F in Figure 2a, which would eliminate the primary bottleneck in both computation and memory usage. Consequently, our method comprises two stages, as illustrated in Figure 2b: A *cache initialization* stage (Stage A), where the frozen sub-network is run for building the cache only once, and a *training* stage (Stage B), where the training sub-network is fine-tuned using the cached feature map for multiple epochs.

Although our idea appears simple, there are two main challenges. First, the randomness of y_F^A renders it non-reusable and, therefore, non-cacheable. Second, if we were to make the intermediate feature map y_F^A cacheable, the resulting cache would be significantly larger than the original dataset. This would be undesirable since it would place an excessive strain on the memory system. To overcome these challenges, we introduce the augmented feature caching (Section 4.2) and cache compression (Section 4.3) techniques, as demonstrated in Figure 2b, and outline our method in Algorithm 2.

4.1. Motivation: Forward propagation through the frozen sub-network is the New Bottleneck

Figure 3 shows the accuracy for training different numbers of blocks. Figure 1b presents the breakdown of computation and memory footprint, *i.e.*, FLOPs and I/O between processor and main memory, needed for steps (2-4) in Algorithm 1 when various layers are trained. From these figures,

we have the following findings:

- When training the entire model, back propagation consumes most resources.
- For both EfficientNet-B0 and MobileNet-V2, training fewer than 7 blocks delivers almost the same accuracy as training the entire model, but with significantly reduced computational demand. For instance, training 7 blocks in MobileNet-V2 results in an accuracy that is only 0.4 lower than a full training, while demanding just half the computational resources.
- When less than 7 blocks are trained, forward propagation through the frozen part of the model is the major consumer of computation and memory I/O. For example, when training 5 blocks, forward propagation through the frozen part of the model consumes 53% computation and 79% memory I/O, which is higher than the sum of others.

Our first observation confirms that when training the whole model, back propagation is the primary resource drain. However, our second observation highlights that in the context of transfer learning, it might not be resource-efficient to train the entire model. This insight steers our attention to scenarios where only the latter layers are trained. In such situations, as indicated by our third observation, the main resource drain is the forward propagation through the non-trained subsection, denoted as sub-network f_F.

4.2. Augmented Feature Caching (Steps A1, A3 and B2 in Figure 2b)

When training the last few layers, it is essential to address the system-wide efficiency, specifically the efficiency of Step (2) in Algorithm 1, which is the forward propagation through the frozen part. To tackle this issue, we propose an augmented feature caching method based on the following properties:

1. For a given input, the inference through the frozen part f_F produces a deterministic output;
2. With a fixed dataset, the only source of randomness for the input is data augmentation $T(\cdot)$;
3. Vision models can exhibit equivariance to commonly used data augmentation transformations [23].

The first two properties are straightforward. Indeed, since all parameters in the frozen part f_F are fixed, the network is static and will always return the same output for the same input. Moreover, given the fixed data source, data augmentation is the only source of randomness. However, we cannot directly cache and reuse the feature map y_F^A produced by the frozen neural network f_F due to the unpredictable nature of input. To make it cacheable, we need the third property.

The third property can be either embedded in the neural network architecture (*e.g.*, translation equivariance in CNN) or learned during model pretraining before on-device transfer learning (*e.g.*, translation equivariance and scaling equivariance for ViT). More importantly, the third property suggests that the transformation on the model input can be approximately shifted to the model output. For instance, steps (1) and (2) in Algorithm 1 can be written as $y_F^A = f_F(T(x); \Theta_F)$, where $x \in \mathcal{X}$. With property 3, this can be approximated as:

$$\tilde{y}_F^A = T(y_F) = T(f_F(x; \Theta_F)) \overset{prop.3}{\approx} f_F(T(x); \Theta_F) = y_F^A \tag{1}$$

For simplicity, in Equation (1), we skipped the compression $C(\cdot)$ and decompression $D(\cdot)$ shown in Figure 2b. By doing this, randomness is eliminated from the input of the frozen part of the neural network, and y_F, the output of f_F, becomes deterministic and cacheable.

CIFAR100 - 224×224	664MB
FP32 Cache	3.6GB
INT8 Cache	898MB
INT4 Cache	460MB
INT2 Cache	226MB
INT1 Cache	113MB

Table 1. Comparision between size of CIFAR100 dataset, cache without compression (FP32), and cache with compression (INT).

4.3. Cache Compression (Steps A2 and B1, Figure 2b)

Although augmented feature caching can alleviate the computational bottleneck, it puts stress on the memory system. As shown in Table 1, the CIFAR100 [19] dataset occupies 664MB of memory space, which can fit in the DRAM of most devices. However, when training the last five blocks of the MobileNet-V2 neural network and caching the intermediate feature maps from previous layers, the cache size balloons to 3.6GB. This size, which is about 5.6× larger than the original dataset, may not fit in the DRAM of most edge devices (*e.g.*, Raspberry Pi [11]). Storing cached feature maps on a hard drive and retrieving them back into DRAM when needed is not an ideal solution since the throughput and latency for accessing a hard drive are much worse than accessing DRAM, thus leading to significant system delays [14].

The primary issue is that feature maps typically have higher numerical precision than input images (INT8 *vs.* single-precision floating-point), which means that pixels in feature maps require more storage space than those in images. To address this problem, we draw inspiration from model compression techniques and propose a feature quantization method. Specifically, we apply a channel-wise quantile-based quantization method. For $y_F[i]$, the i-th channel data generated by the frozen sub-network f_F, we

quantize the feature map to n-bit y_q with $C(\cdot)$ as follows:

$$y_q[i] = C(y_F[i]) = \text{round}\left(s * (y_F[i] - b)\right)$$
$$s = \frac{2^n - 1}{Q_{1-k}(y_F[i]) - Q_k(y_F[i])}, b = Q_k(y_F[i]) \quad (2)$$

where $Q_k(y_F[i])$ is the value k-th quantile for $y_F[i]$. As illustrated in Table 1, by quantizing the feature map to 2-bit, the cache size reduces to 226MB, which is smaller than both the original CIFAR100 dataset and the uncompressed cache. More importantly, we demonstrate that, in most cases, quantizing the feature map to 2-bit has minimal impact on accuracy.

Finally, to minimize the effect of cache compression on model training, we introduce a decompression step $D(\cdot)$ to recover the feature map to its original data format (*e.g.*, dequantize the 2-bit data to floating-point data):

$$\tilde{y}_F[i] = D(y_q[i]) = y_q[i]/s + b \quad (3)$$

To this end, we have tackled both problems related to non-cacheable feature maps and excessive cache size, resulting in our method being concisely represented as Algorithm 2. In the first stage, we process each sample in the dataset using the frozen sub-network. The output feature map is then compressed and stored in the cache. This stage is carried out only once. In the second stage, we first load and decompress the feature map from the cache. After augmenting these features, we feed them into the training sub-network. Finally, we perform BP and update the trainable weights. This stage is repeated for a total of N epochs.

Algorithm 2 Training Process for Two Stage On-device Transfer Learning

***Stage A**: Cache initialization for only once*
for $x \in \mathcal{X}$ **do**
  ***Step (A1)**: Forward for frozen sub-network*
  $y_F \leftarrow f_F(x; \Theta_F)$
  ***Step (A2)**: Cache compression*
  $y_Q \leftarrow C(y_F)$
  ***Step (A3)**: Caching*
  $\mathcal{C} \leftarrow \mathcal{C} \cup y_Q$
end for
***Stage B**: Training with cache for N epochs*
for epoch in $1...N$ **do**
  for $(y_Q, \hat{y}) \in (\mathcal{C}, \mathcal{Y})$ **do**
    ***Step (B1)**: Cache decompression*
    $\tilde{y}_F \leftarrow D(y_Q)$
    ***Step (B2)**: Feature augmentation*
    $\tilde{y}_F^A \leftarrow T(y_F)$
    ***Step (B3)**: Forward for training sub-network*
    $\tilde{y}_T^A \leftarrow f_T(\tilde{y}_F^A; \Theta_{T,i})$
    ***Step (B4)**: Back propagation and update weights*
    $\Delta\Theta_T \leftarrow \text{optim}(\Theta_T, y_T^A, \hat{y}, y_F^A)$
    $\Theta_T \leftarrow \Theta_T + \Delta\Theta_T$
  end for
end for

5. Experimental Results

In this section, we first present our experimental results for running the two-stage on-device training approach on real edge devices. Next, in order to support our assertion that our paper provides an orthogonal approach to previous studies, we showcase an example that combines our methodology with [25]. Finally, we evaluate the performance of the main elements of our system - specifically, the augmented feature caching (see Section 4.2) and cache compression components (see Section 4.3).

5.1. Experimental Setup

Evaluation of Training Accuracy: To assess the accuracy of our method, we utilize a GPU-workstation equipped with the NVIDIA-A6000 GPU. The environment is set up using PyTorch 1.13 and MMClassification v0.25 on the Ubuntu 20.04 operating system, along with CUDA 11.7. We finetune ImageNet-pretrained models on six different datasets: CIFAR100 [19] (CF100), CIFAR10 [19] (CF10), Stanford Cars [18] (Cars), Flowers [30], Food-101 [4] (Food), and Oxford-IIIT Pet [31] (Pet). All images across the datasets are standardized to a resolution of 224×224. We finetune each model for 50 epochs, utilizing the AdamW optimizer [28] and a batch size of 64. The learning rate is adjusted for each dataset according to the performance of ResNet-18 with vanilla BP. For evaluation on large scale dataset, since our model are pretrained on ImageNet, we choose the finetuning target as Places365 dataset [43], which contains more images (1.8M *vs.* 1.2M in ImageNet) and is more challenging.

On-Device Training Environment: We measure the efficiency of our method on the NVIDIA Jetson Orin NX [10], equipped with an 8-core Arm Cortex-A78A CPU and a 1024-core NVIDIA Ampere architecture GPU. During this evaluation, we set the power mode to MAXN, with the maximum frequencies for the CPU and GPU being 1984 MHz and 918 MHz, respectively. Latency is calculated by averaging the time taken to train 200 batches, with additional 10 batches for warm-up. The data size is determined by measuring the size of the dataset or the generated cache.

Baselines: Being the first work for two-stage on-device training, our method offers a unique perspective compared to previous work on on-device training, making direct comparisons impractical. However, our method can be integrated with previous methods to further enhance efficiency. As such, in Section 5.2, we primarily compare our method to the traditional single-stage training method, while in Sec-

Model	Type	Latency [ms]	Cache [MB]	Accuracy [%]					
				CF10	CF100	Cars	Flowers	Food	Pets
MobileNet-V2[33] Last 1 Block	Baseline	3.30	664	91.07	70.53	**65.59**	91.45	74.27	**87.79**
	Ours-4b	**0.50**	**188**	**91.67**	**72.60**	64.59	**91.82**	74.42	87.54
	Ours-2b	**0.50**	**95**	90.71	70.01	59.76	91.32	**75.12**	87.71
MobileNet-V2[33] Last 4 Blocks	Baseline	3.87	664	94.21	76.28	75.79	93.02	77.52	87.52
	Ours-4b	**1.33**	**450**	**94.60**	**76.60**	**77.63**	**93.49**	**78.86**	87.60
	Ours-2b	**1.34**	**226**	94.43	76.33	77.15	93.40	79.14	**87.98**
MobileNet-V2[33] Last 7 Blocks	Baseline	4.70	664	**95.10**	**77.74**	80.13	92.41	79.43	**87.52**
	Ours-4b	**2.49**	**300**	94.83	77.18	**80.67**	**92.42**	80.10	87.38
	Ours-2b	**2.49**	**151**	94.84	76.99	80.49	91.72	**80.17**	87.30
EfficientNet-B0[36] Last 3 Blocks	Baseline	7.41	664	94.19	76.49	74.58	**93.46**	**79.62**	91.50
	Ours-4b	**2.32**	**226**	**94.45**	**77.09**	**75.20**	92.78	79.15	91.44
	Ours-2b	**2.34**	**113**	94.38	76.76	74.17	92.39	79.24	**91.55**
EfficientNet-B0[36] Last 5 Blocks	Baseline	8.66	664	95.10	78.65	79.21	**94.57**	**81.59**	**91.09**
	Ours-4b	**3.94**	**525**	95.54	**79.12**	80.05	92.88	81.14	90.71
	Ours-2b	**3.94**	**263**	**95.82**	78.83	**80.55**	93.71	81.04	90.60
ResNet-18[13] Last 2 Blocks	Baseline	2.87	664	93.63	75.47	**74.98**	**92.03**	75.84	**88.85**
	Ours-4b	**1.15**	1200	93.82	**76.38**	74.85	91.93	**75.84**	88.50
	Ours-2b	**1.14**	**599**	**93.83**	76.24	74.18	92.01	75.27	88.72
EfficientFormerV2-S0[22] Last 5 Blocks (ViT)	Baseline	6.62	664	94.20	**74.97**	**77.58**	**91.51**	**81.98**	**90.57**
	Ours-4b	**2.61**	**450**	94.10	73.95	75.35	90.34	80.24	89.07
	Ours-2b	**2.61**	**226**	**94.21**	74.22	75.60	90.83	80.72	89.15

Table 2. Results for on-device transfer learning with vanilla single-stage training (Baseline, Algorithm 1) and proposed two-stage training (Algorithm 2). Ours-2/4b refers to the proposed method with 2/4-bit quantization for cache compression. For the baseline, metric "Data" represent the size of the dataset, and for our two-stage training, metric "Data" represents the size of the cache, as shown in Figure 2. Latency and data size are measured by training with CIFAR100. For latency and data size, we highlight results that outperform the baseline; for accuracy, we highlight the highest accuracy for each dataset.

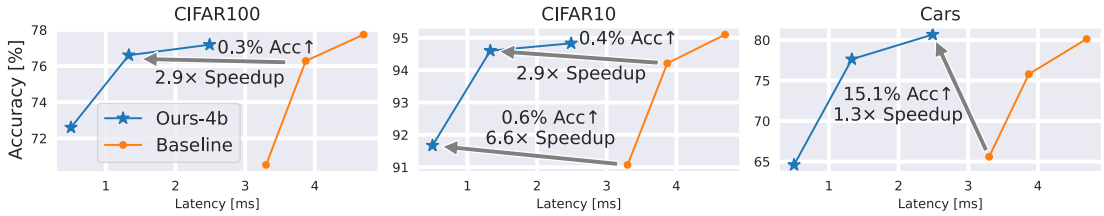


Figure 3. Accuracy and latency for finetuning the last 1/4/7 blocks in MobileNet-V2 on three different datasets. Latencies are measured on NVIDIA Jetson Orin NX for the CIFAR100 dataset.

tion 5.5, we provide an example of how our method can be combined with [25].

5.2. Evaluation: Accuracy and Speed

Figure 3 and Table 2 provide a comparison between our two-stage training approach and the vanilla single-stage method, in terms of accuracy, latency, and dataset size. In most cases, our method outperforms the single-stage training approach by offering lower latency, higher accuracy, and reduced memory usage for dataset storage. For instance, as shown in Figure 3, with a similar accuracy (0.4% higher than the baseline), our method achieves a 2.9× speedup for training the MobileNet-V2 on CIFAR10 dataset, and with a similar latency (1.3× faster than the baseline), our method achieves a 15.1% higher accuracy for training the MobileNet-V2 on the Cars dataset. These results clearly illustrate that our method significantly enhances the efficiency of the on-device training system.

5.3. Scalability on Large-scale Datasets

While our technique is primarily designed for efficient transfer learning on edge devices, it also performs impressively on larger datasets. To test its scalability, we evaluate the accuracy and training time on a large-scale dataset

Model	Method	Speedup	Training Time [minutes]	Accuracy [%]
MBV2-1Blk	Baseline	-	460	50.98
	Ours-4b	6.9×	67	50.50
MBV2-4Blk	Baseline	-	429	53.21
	Ours-4b	3.0×	141	52.71
MBV2-7Blk	Baseline	-	603	53.70
	Ours-4b	1.9×	310	53.36

Table 3. Experimental results for training MobileNet-V2 on Places365 dataset with NVIDIA A6000 GPU. "MBV2-1Blk" represents training the last block in MobileNet-V2. "Training Time" represents total training time for 50 epochs on Places365 dataset. "Ours-4b" refers our method with 4-bit cache compression.

Places365 which contains over 1.8M data samples. Of note, since our model is pretrained on ImageNet, we cannot test our transfer learning method on ImageNet.

The results in Table 3 underscore our method's efficiency. We achieved up to a 7× increase in speed with only a minor drop in accuracy. For instance, when training the last block of the MobileNet-V2 model using our technique, it took merely 67 minutes to hit an accuracy of 50.50%. In contrast, the baseline approach demanded 7× longer (460 minutes) to achieve a marginally better accuracy of 50.98%. This clearly demonstrates our method's capability for scaling on larger datasets.

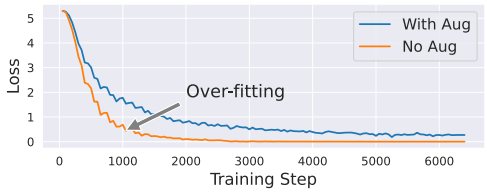


Figure 4. Loss curve for training 4 blocks in MobileNet-V2 on Cars dataset with/without feature augmentation (Aug).

5.4. Ablation Study

In this section, we delve into the significance of augmented feature caching (Section 4.2) and cache compression (Section 4.3). Table 4 presents the results of our experiments. More specifically:

Augmented Feature Caching: Augmented feature caching is crucial for the two-stage on-device training as it enables the caching of outputs from the frozen sub-network. Thus, we cannot entirely disable this feature. Instead, we assess the effectiveness of the feature augmentation step (Step B2 in Algorithm 2) within the augmented feature caching method. For this experiment, we set the cache compression quantization to 8-bit and deactivate the feature augmentation. As shown in Table 4, the absence of augmented feature caching results in a consistent decline in accuracy without any latency benefits. For instance, when training the Cars dataset without feature augmentation, the accuracy drops by 2.97% compared to our method with feature augmentation and 8-bit cache compression. As Figure 4 illustrates, training without feature augmentation causes the loss value to rapidly converge to zero, indicating model overfitting. This finding underscores the importance of feature augmentation in preventing overfitting.

Cache Compression: Table 4 shows that, in the absence of cache compression, the cache size explodes to 3.5 GB, which is too large for many edge devices with limited memory capacity. For example, Raspberry Pi devices may have the DRAM size as small as just 1 GB. Indeed, our cache compression method can substantially reduce memory usage, making on-device training feasible for a broad range of hardware platforms.

5.5. Combining our Method with Prior Work

Table 5 presents the experimental results combining our method with the "quantization-aware scaling" method proposed in [25]. With quantized inference and BP, our method can significantly reduce the training latency while achieving a higher accuracy. For instance, we observe a 1.58% increase in accuracy and a 5.31× speedup when training the last two blocks in MCUNet-5fps on the CIFAR100 dataset. These findings highlight the broad applicability of our method. In fact, our two-stage on-device training can be effectively combined with several other techniques [3, 6, 15, 16, 34, 42] to further enhance efficiency.

5.6. Cache Compression Overhead

The key overhead of our method is cache compression and decompression (Step A2 and B1 in Figure 2). Table 6 details the time required to compress and decompress one feature map when training the last four blocks of MobileNet-V2. Meanwhile, the average time taken in both steps across 50 epochs is also provided.

As shown in Table 6, compression takes around 465us. The majority of this time is spent on collecting the quantile of the feature map (Q_k and Q_{1-k} in Equation 2) for quantization. In contrast, the decompression is significantly faster, taking only about 30us. This time slightly increases as more bits are used for quantization. This is because decompression (Equation 3) spent most time accessing the memory. Consequently, as we use more bits for quantization, more memory is accessed, leading to a slight increase in time.

It is important to note that compression is a one-time process to generate the cache, while decompression is repeated for every epoch. Therefore, compression does **not** pose a major issue as long as decompression remains efficient. The final row of the table shows the average time taken by both operations over 50 epochs. This duration is almost equivalent to the time needed for decompression and is considerably less than the overall training time.

Method	Cache Compression	Feature Aug.	Latency [ms]	Cache [MB]	Accuracy [%]					
					CF10	CF100	Cars	Flowers	Food	Pets
Baseline	-	-	3.87	664	94.21	76.28	75.79	93.02	77.52	87.52
No Feature Aug.	8 bit	✗	1.34	898	93.94	75.15	73.90	92.47	76.44	86.75
No Compression	✗	✓	1.34	3589	94.64	76.35	77.66	93.71	78.58	87.71
Compression 8 bit	8 bit	✓	1.34	898	94.49	76.35	76.87	93.49	78.50	87.76
Compression 4 bit	4 bit	✓	1.33	450	94.60	76.60	77.63	93.49	78.86	87.60
Compression 2 bit	2 bit	✓	1.34	226	94.43	76.33	77.15	93.40	79.14	87.98
Compression 1 bit	1 bit	✓	1.35	113	92.34	69.85	72.78	90.13	75.81	86.59

Table 4. Experimental results for disabling or changing hyper-parameters in our two-stage on-device transfer learning method. "Feature Aug." is short for "Feature Augmentation". Baseline refers to the vanilla single-stage training algorithm (Algorithm 1).

Method	Latency [ms]	Accuracy	
		CF10	CF100
[25]	5.16	87.55	64.04
+Ours-8b	**0.97**	**87.93**	**65.62**

Table 5. Accuracy and latency for training the last two blocks in MCUNet-5fps by combining our method with "quantization-aware scaling" method in [25]. Best results are highlighted.

Quantization bit	8-bit	4-bit	2-bit
Compression [μs]	466.36	471.99	465.78
Decompression [μs]	34.83	30.23	28.40
50 Epoch Average [μs]	44.16	39.67	37.72

Table 6. Overhead for cache compression. Average run time is calculated for 50 epochs, *i.e.*, compression runs once and decompression is repeated 50 times.

6. Related Work

On-device transfer learning with layer freezing: Training only the last few layers has been used in prior work. For example, [25] selectively train several channels in the convolutional neural networks and accelerate the BP process (Step 4 in Algorithm 1) with gradient quantization. [39] train the last 2 or 4 layers in ResNets and other neural networks and speed up BP with gradient approximation. Indeed, by training only the last few layers, the computational cost for BP can be greatly reduced. However, as we analysed in Section 4.1, the key bottleneck is actually the forward propagation through the frozen sub-network (Step 2 in Algorithm 1) instead of BP. So although these methods can accelerate the BP significantly, the end-to-end speedup, including both forward propagation and BP, is quite limited. Our method, instead, targets directly the true bottleneck, forward propagation through the frozen sub-network, and achieves significant end-to-end speedup.

Orthogonal On-device Training methods: Existing works can be classified into two categories. The first category [3, 6, 8, 15–17, 25, 34, 39, 42] targets efficiency in BP and does not modify the neural network. For example, [39] propose to accelerate BP by approximating the gradients with their average value, which can simplify the computation; [3, 6, 15, 16, 25, 34, 42] reduce the cost for arithmetic operations (add and multiply) by quantizing the gradient during BP; [17] reduce the memory consumption by pruning the context (x^A in Step (2) Algorithm 1). The second category of approaches proposes to accelerate on-device training by attaching and training only a small residual connections to the neural network [5, 35]. For example, [5] attach small inverted residual blocks to the MobileNet-like models. Though [5] can reduce the memory consumption, it incurs larger computational cost. Since our method is the first one introducing feature caching, it can be combined with prior works for a better efficiency. Section 5.5 shows an example combining our method with [25], which achieves 5.3× higher speedup with 1.6% more accuracy.

7. Conclusion

In this paper, we have addressed the on-device transfer learning for resource-constrained edge devices. Through comprehensive profiling, we have identified that the primary bottleneck for on-device training lies in the forward propagation through the frozen sub-network. As a result, we have proposed a novel two-stage on-device training approach to overcome this bottleneck and enhance the efficiency of on-device transfer learning. In Section 4, we have demonstrated our method, which eases the bottleneck by storing and reusing the output from the frozen sub-network. We have also addressed the challenges of non-cachable feature maps and cache size explosion. To this end, we have developed augmented feature caching and cache compression techniques, making our two-stage on-device transfer learning method truly practical. Our extensive experiments presented in Section 5 have demonstrated our method's efficiency and broad applicability on a real edge device. Indeed, our approach achieves a significant speedup while maintaining accuracy comparable to baseline methods.

Acknowledgment

This work was supported in part by the US National Science Foundation (NSF) grant CNS-2007284.

References

[1] Rahaf Aljundi, Min Lin, Baptiste Goujaud, and Yoshua Bengio. Gradient based Sample Selection for Online Continual Learning. In *Advances in Neural Information Processing Systems*, 2019. 1

[2] Shun-ichi Amari. Backpropagation and Stochastic Gradient Descent Method. *Neurocomputing*, 1993. 2

[3] Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. Scalable Methods for 8-Bit Training of Neural Networks. In *Advances in Neural Information Processing Systems*, 2018. 7, 8

[4] Lukas Bossard, Matthieu Guillaumin, and Luc Van Gool. Food-101 – Mining Discriminative Components with Random Forests. In *European Conference on Computer Vision*, 2014. 5

[5] Han Cai, Chuang Gan, Ligeng Zhu, and Song Han. Tinytl: Reduce Memory, not Parameters for Efficient On-device Learning. In *Advances in Neural Information Processing Systems*, 2020. 8

[6] Jianfei Chen, Yu Gai, Zhewei Yao, Michael W Mahoney, and Joseph E Gonzalez. A Statistical Framework for Low-bitwidth Training of Deep Neural Networks. In *Advances in Neural Information Processing Systems*, 2020. 7, 8

[7] Wei Chen, Kartikeya Bhardwaj, and Radu Marculescu. Fedmax: Mitigating Activation Divergence for Accurate and Communication-Efficient Federated Learning. In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2020*, 2021. 1

[8] Hung-Yueh Chiang, Natalia Frumkin, Feng Liang, and Diana Marculescu. Mobiletl: On-device transfer learning with inverted residual blocks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2023. 8

[9] Gabriela Csurka. A comprehensive survey on domain adaptation for visual applications. *Domain adaptation in computer vision applications*, 2017. 1

[10] Michael Ditty. NVIDIA ORIN System-On-Chip. In *IEEE Hot Chips 34 Symposium*, 2022. 5

[11] Warren Gay. *Raspberry Pi Hardware Reference*. Apress, 2014. 4

[12] Yunhui Guo, Honghui Shi, Abhishek Kumar, Kristen Grauman, Tajana Rosing, and Rogerio Feris. Spottune: Transfer Learning through Adaptive Fine-tuning. In *IEEE conference on computer vision and pattern recognition*, 2019. 1, 2

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *IEEE conference on computer vision and pattern recognition*, 2016. 1, 3, 6

[14] John L Hennessy and David A Patterson. *Computer architecture: A quantitative approach*. Elsevier, 2011. 4

[15] Ziyang Hong and C Patrick Yue. Efficient-grad: Efficient training deep convolutional neural networks on edge devices with grad ient optimizations. *ACM Transactions on Embedded Computing Systems*, 2022. 7, 8

[16] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 2017. 7, 8

[17] Ziyu Jiang, Xuxi Chen, Xueqin Huang, Xianzhi Du, Denny Zhou, and Zhangyang Wang. Back razor: Memory-efficient transfer learning by self-sparsified backpropagation. In *Advances in Neural Information Processing Systems*, 2022. 8

[18] 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. 5

[19] Alex Krizhevsky. Learning multiple layers of features from tiny images. 2009. 4, 5

[20] Guihong Li, Sumit K Mandal, Umit Y Ogras, and Radu Marculescu. FLASH: Fast Neural Architecture Search with Hardware Optimization. *ACM Transactions on Embedded Computing Systems*, 2021. 1

[21] Rui Li, Qianfen Jiao, Wenming Cao, Hau-San Wong, and Si Wu. Model Adaptation: Unsupervised Domain Adaptation without Source Data. In *IEEE conference on computer vision and pattern recognition*, 2020. 1

[22] Yanyu Li, Ju Hu, Yang Wen, Georgios Evangelidis, Kamyar Salahi, Yanzhi Wang, Sergey Tulyakov, and Jian Ren. Rethinking Vision Transformers for MobileNet Size and Speed. *arXiv preprint arXiv:2212.08059*, 2022. 6

[23] Yanghao Li, Hanzi Mao, Ross Girshick, and Kaiming He. Exploring Plain Vision Transformer Backbones for Object Detection. In *European Conference on Computer Vision*, 2022. 4

[24] Ching-Yi Lin and Radu Marculescu. Model Personalization for Human Activity Recognition. In *2020 IEEE International Conference on Pervasive Computing and Communications Workshop*, 2020. 1

[25] Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. On-device Training under 256kb Memory. In *Advances in Neural Information Processing Systems*, 2022. 1, 2, 5, 6, 7, 8

[26] Ze Liu, Han Hu, Yutong Lin, Zhuliang Yao, Zhenda Xie, Yixuan Wei, Jia Ning, Yue Cao, Zheng Zhang, Li Dong, et al. Swin Transformer v2: Scaling up Capacity and Resolution. In *IEEE conference on computer vision and pattern recognition*, 2022. 3

[27] Mingsheng Long, Yue Cao, Jianmin Wang, and Michael Jordan. Learning Transferable Features with Deep Adaptation Networks. In *International conference on machine learning*, 2015. 1, 2

[28] Ilya Loshchilov and Frank Hutter. Decoupled Weight Decay Regularization. *arXiv preprint arXiv:1711.05101*, 2017. 5

[29] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *Artificial intelligence and statistics*, 2017. 1

[30] Maria-Elena Nilsback and Andrew Zisserman. A Visual Vocabulary for Flower Classification. In *IEEE conference on computer vision and pattern recognition*, 2006. 5

[31] Omkar M. Parkhi, Andrea Vedaldi, Andrew Zisserman, and C. V. Jawahar. Cats and Dogs. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2012. 5

[32] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 2015. 2

[33] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted Residuals and Linear Bottlenecks. In *IEEE conference on computer vision and pattern recognition*, 2018. 1, 2, 3, 6

[34] Xiao Sun, Naigang Wang, Chia-Yu Chen, Jiamin Ni, Ankur Agrawal, Xiaodong Cui, Swagath Venkataramani, Kaoutar El Maghraoui, Vijayalakshmi (Viji) Srinivasan, and Kailash Gopalakrishnan. Ultra-Low Precision 4-bit Training of Deep Neural Networks. In *Advances in Neural Information Processing Systems*, 2020. 7, 8

[35] Yi-Lin Sung, Jaemin Cho, and Mohit Bansal. Lst: Ladder Side-tuning for Parameter and Memory Efficient Transfer Learning. In *Advances in Neural Information Processing Systems*, 2022. 8

[36] Mingxing Tan and Quoc Le. Efficientnet: Rethinking Model Scaling for Convolutional Neural Networks. In *International conference on machine learning*, 2019. 2, 3, 6

[37] Anastasia Tkach, Andrea Tagliasacchi, Edoardo Remelli, Mark Pauly, and Andrew Fitzgibbon. Online Generative Model Personalization for Hand Tracking. *ACM Transactions on Graphics*, 2017. 1

[38] Yuedong Yang, Zihui Xue, and Radu Marculescu. Anytime Depth Estimation with Limited Sensing and Computation Capabilities on Mobile Devices. In *Conference on Robot Learning*, 2022. 1

[39] Yuedong Yang, Guihong Li, and Radu Marculescu. Efficient On-device Training via Gradient Filtering. In *IEEE conference on computer vision and pattern recognition*, 2023. 1, 2, 8

[40] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How Transferable are Features in Deep Neural Networks? In *Advances in neural information processing systems*, 2014. 1, 2

[41] Chen Zhang, Yu Xie, Hang Bai, Bin Yu, Weihong Li, and Yuan Gao. A Survey on Federated Learning. *Knowledge-Based Systems*, 2021. 1

[42] Kang Zhao, Sida Huang, Pan Pan, Yinghan Li, Yingya Zhang, Zhenyu Gu, and Yinghui Xu. Distribution Adaptive INT8 Quantization for Training CNNs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2021. 7, 8

[43] Bolei Zhou, Agata Lapedriza, Aditya Khosla, Aude Oliva, and Antonio Torralba. Places: A 10 million image database for scene recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2017. 5