

p-Meta: Towards On-device Deep Model Adaptation

Zhongnan Qu
ETH Zurich
Zurich, Switzerland
quz@ethz.ch

Yongxin Tong
Beihang University
Beijing, China
yxtong@buaa.edu.cn

Zimu Zhou
Singapore Management University
Singapore, Singapore
zimuzhou@smu.edu.sg

Lothar Thiele
ETH Zurich
Zurich, Switzerland
thiele@ethz.ch

ABSTRACT

Data collected by IoT devices are often private and have a large diversity across users. Therefore, learning requires pre-training a model with available representative data samples, deploying the pre-trained model on IoT devices, and adapting the deployed model on the device with local data. Such an on-device adaption for deep learning empowered applications demands data and memory efficiency. However, existing gradient-based meta learning schemes fail to support memory-efficient adaptation. To this end, we propose p-Meta, a new meta learning method that enforces structure-wise partial parameter updates while ensuring fast generalization to unseen tasks. Evaluations on few-shot image classification and reinforcement learning tasks show that p-Meta not only improves the accuracy but also substantially reduces the peak dynamic memory by a factor of 2.5 on average compared to state-of-the-art few-shot adaptation methods.

CCS CONCEPTS

• **Computing methodologies** → *Neural networks*.

KEYWORDS

deep neural networks; meta learning; memory-efficient training

ACM Reference Format:

Zhongnan Qu, Zimu Zhou, Yongxin Tong, and Lothar Thiele. 2022. p-Meta: Towards On-device Deep Model Adaptation. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '22)*, August 14–18, 2022, Washington, DC, USA. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3534678.3539293>

1 INTRODUCTION

Adaption to *unseen* environments, users, and tasks is crucial for deep learning empowered IoT applications to deliver consistent performance and customized services. Data collected by IoT devices

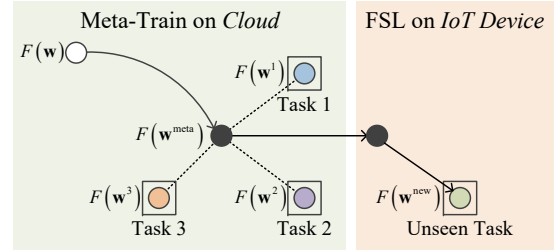


Figure 1: Meta learning and few-shot learning (FSL) in the context of on-device adaptation. The backbone $F(w)$ is meta-trained into $F(w^{meta})$ on the cloud and is deployed to IoT devices to adapt to unseen tasks as $F(w^{new})$ via FSL.

are often private and have a large diversity across users. For instance, activity recognition with smartphone sensors should adapt to countless walking patterns and sensor orientation [13]. Human motion prediction with home robots needs fast learning of unseen poses for seamless human-robot interaction [19]. In these applications, the *new* data collected for model adaptation tend to relate to personal habits and lifestyle. Hence, *on-device* model adaptation is preferred over uploading the data to cloud servers for retraining.

Yet on-device adaption of a deep neural network (DNN) demands *data efficiency* and *memory efficiency*. The excellent accuracy of contemporary DNNs is attributed to training with high-performance computers on large-scale datasets [14]. For example, it takes 29 hours to complete a 90-epoch ResNet50 [21] training on ImageNet (1.2 million training images) [31] with 8 NVIDIA Tesla P100 GPUs [16]. For on-device adaptation, however, neither abundant *data* nor *resources* are available. A personal voice assistant, for example, may learn to adapt to users' accent and dialect within a few sentences, while a home robot should learn to recognize new object categories with few labelled images to navigate in new environments. Furthermore, such adaptation is expected to be conducted on low-resource platforms such as smart portable devices, home hubs, and other IoT devices, with only several *KB* to *MB* memory.

For *data-efficient* DNN adaptation, we resort to *meta learning*, a paradigm that learns to fast generalize to unseen tasks [22]. Of our particular interest is *gradient-based* meta learning [1, 10, 26, 28] for its wide applicability in classification, regression and reinforcement learning, as well as the availability of gradient-based training frameworks for low-resource devices, *e.g.*, TensorFlow Lite [36]. Fig. 1 explains major terminologies in the context of on-device adaptation. Given a backbone, its weights are *meta-trained* on *many* tasks, to

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

KDD '22, August 14–18, 2022, Washington, DC, USA

© 2022 Association for Computing Machinery.

ACM ISBN 978-1-4503-9385-0/22/08...\$15.00

<https://doi.org/10.1145/3534678.3539293>

output a *model* that is expected to fast adapt to new *unseen* tasks. The process of adaptation is also known as *few-shot learning*, where the meta-trained model is further retrained by standard stochastic gradient descent (SGD) on *few new* samples only.

However, existing gradient-based meta learning schemes [1, 10, 26, 28] fail to support *memory-efficient* adaptation. Although *meta training* is conducted in the cloud, *few-shot learning* (adaptation) of the meta-trained model is performed on IoT devices. Consider to retrain a common backbone ResNet12 in a 5-way (5 new classes) 5-shot (5 samples per class) scenario. One round of SGD consumes 370.44MB peak dynamic memory, since the inputs of all layers must be stored to compute the gradients of these layers' weights in the backward path. In comparison, inference only needs 3.61MB. The necessary dynamic memory is a key bottleneck for on-device adaptation due to cost and power constraints, even though the meta-trained model only needs to be retrained with a few data.

Prior efficient DNN training solutions mainly focus on parallel and distributed training on data centers [3, 4, 17, 18, 29]. On-device training has been explored for *vanilla supervised training* [15, 24, 25], where training and testing are performed on the *same* task. A pioneer study [2] investigated on-device adaptation to new tasks via memory-efficient *transfer learning*. Yet transfer learning is prone to overfitting when only a few samples are available [10].

In this paper, we propose p-Meta, a new meta learning method for data- and memory-efficient DNN adaptation. The key idea is to enforce *structured partial parameter updates* while ensuring *fast generalization to unseen tasks*. The idea is inspired by recent advances in understanding gradient-based meta learning [26, 28]. Empirical evidence shows that only the *head* (the last output layer) of a DNN needs to be updated to achieve reasonable few-shot classification accuracy [28] whereas the *body* (the layers closed to the input) needs to be updated for cross-domain few-shot classification [26]. These studies imply that certain weights are more important than others when generalizing to unseen tasks. Hence, we propose to automatically identify these *adaptation-critical weights* to minimize the memory demand in few-shot learning.

Particularly, the critical weights are determined in two structured dimensionalities as, (i) layer-wise: we meta-train a layer-by-layer learning rate that enables a *static* selection of critical layers for updating; (ii) channel-wise: we introduce meta attention modules in each layer to select critical channels *dynamically*, i.e., depending on samples from new tasks. Partial updating of weights means that (structurally) sparse gradients are generated, reducing memory requirements to those for computing nonzero gradients. In addition, the computation demand for calculating zero gradients can be also saved. To further reduce the memory, we utilize *gradient accumulation* in few-shot learning and *group normalization* in the backbone. Although weight importance metrics and SGD with sparse gradients have been explored in vanilla training [8, 15, 20, 29], it is unknown (i) how to identify adaptation-critical weights and (ii) whether meta learning is robust to sparse gradients, where the objective is fast adaptation to *unseen* tasks.

Our main contributions are summarized as follows.

- We design p-Meta, a new meta learning method for data- and memory-efficient DNN adaptation to unseen tasks. p-Meta automatically identifies adaptation-critical weights

both layer-wise and channel-wise for low-memory adaptation. The hierarchical approach combines static identification of layers and dynamic identification of channels whose weights are critical for few-shot adaptation. To the best of our knowledge, p-Meta is the first meta learning method designed for on-device few-shot learning.

- Evaluations on few-shot image classification and reinforcement learning show that, p-Meta not only improves the accuracy but also reduces the peak dynamic memory by a factor of 2.5 on average over the state-of-the-art few-shot adaptation methods. p-Meta can also simultaneously reduce the computation by a factor of 1.7 on average.

In the rest of this paper, we introduce the preliminaries and challenges in Sec. 2, elaborate on the design of p-Meta in Sec. 3, present its evaluations in Sec. 4, review related work in Sec. 5, and conclude in Sec. 6.

2 PRELIMINARIES AND CHALLENGES

In this section, we provide the basics on meta learning for fast adaptation and highlight the challenges to enable on-device adaptation.

Meta Learning for Fast Adaptation. Meta learning is a prevailing solution to adapt a DNN to unseen tasks with limited training samples, i.e., few-shot learning [22]. We ground our work on model-agnostic meta learning (MAML) [10], a generic meta learning framework which supports classification, regression and reinforcement learning. Given the dataset $\mathcal{D} = \{\mathcal{S}, \mathcal{Q}\}$ of an unseen few-shot task, where \mathcal{S} (support set) and \mathcal{Q} (query set) are for training and testing, MAML trains a model $F(\mathbf{w})$ with weights \mathbf{w} such that it yields high accuracy on \mathcal{Q} even when \mathcal{S} only contains a few samples. This is enabled by simulating the few-shot learning experiences over abundant few-shot tasks sampled from a task distribution $p(\mathcal{T})$. Specifically, it meta-trains a backbone F over few-shot tasks $\mathcal{T}^i \sim p(\mathcal{T})$, where each \mathcal{T}^i has dataset $\mathcal{D}^i = \{\mathcal{S}^i, \mathcal{Q}^i\}$, and then generates $F(\mathbf{w}^{\text{meta}})$, an initialization for the unseen few-shot task \mathcal{T}^{new} with dataset $\mathcal{D}^{\text{new}} = \{\mathcal{S}^{\text{new}}, \mathcal{Q}^{\text{new}}\}$. Training from $F(\mathbf{w}^{\text{meta}})$ over \mathcal{S}^{new} is expected to achieve a high test accuracy on \mathcal{Q}^{new} .

MAML achieves fast adaptation via two-tier optimization. In the *inner loop*, a task \mathcal{T}^i and its dataset \mathcal{D}^i are sampled. The weights \mathbf{w} are updated to \mathbf{w}^i on support dataset \mathcal{S}^i via K gradient descent steps, where K is usually small, compared to vanilla training:

$$\mathbf{w}^{i,k} = \mathbf{w}^{i,k-1} - \alpha \nabla_{\mathbf{w}} \ell(\mathbf{w}^{i,k-1}; \mathcal{S}^i) \quad \text{for } k = 1 \cdots K \quad (1)$$

where $\mathbf{w}^{i,k}$ are the weights at step k in the inner loop, and α is the inner step size. Note that $\mathbf{w}^{i,0} = \mathbf{w}$ and $\mathbf{w}^i = \mathbf{w}^{i,K}$. $\ell(\mathbf{w}; \mathcal{D})$ is the loss function on dataset \mathcal{D} . In the *outer loop*, the weights are optimized to minimize the sum of loss at \mathbf{w}^i on query dataset \mathcal{Q}^i across tasks. The gradients to update weights in the outer loop are calculated w.r.t. the starting point \mathbf{w} of the inner loop.

$$\mathbf{w} \leftarrow \mathbf{w} - \beta \nabla_{\mathbf{w}} \sum_i \ell(\mathbf{w}^i; \mathcal{Q}^i) \quad (2)$$

where β is the outer step size.

The meta-trained weights \mathbf{w}^{meta} are then used as initialization for few-shot learning into \mathbf{w}^{new} by K gradient descent steps over \mathcal{S}^{new} . Finally we assess the accuracy of $F(\mathbf{w}^{\text{new}})$ on \mathcal{Q}^{new} .

Table 1: The static memory, the peak dynamic memory, and the total computation (GMACs = 10^9 MACs) of inference and adaptation for sample applications. For image classification we use batch size = 25. For robot locomotion we use rollouts = 20, horizon = 200; each sample corresponds to a rollout episode, and the case for an observation is reported in brackets. The calculation is based on Appendix A.

Application	Model / Benchmark	Static Memory (MB)		Peak Dynamic Memory (MB)		GMACs	
		Model	Sample	Inference	Adaptation	Inference	Adaptation
Image Classification	4Conv [10] / MiniImageNet [39]	0.13	0.53	0.90	48.33	0.72	1.96
Image Classification	ResNet12 [27] / MiniImageNet [39]	32.0	0.53	3.61	370.44	62.08	185.42
Robot Locomotion	MLP [10] / MuJoCo [37]	0.05	0.016(0.00008)	0.08(0.0004)	3.72	0.05	0.15

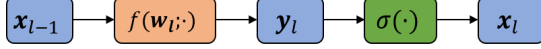


Figure 2: A typical layer l in DNNs. x_{l-1} is the input tensor; x_l is the output tensor, also the input tensor of layer $l+1$; y_l is the intermediate tensor; w_l is the weight tensor.

Memory Bottleneck of On-device Adaptation. As mentioned above, the meta-trained model $F(\mathbf{w}^{\text{meta}})$ can adapt to unseen tasks via K gradient descent steps. Each step is the same as the inner loop of meta-training Eq.(1), but on dataset \mathcal{S}^{new} .

$$\mathbf{w}^{\text{new},k} = \mathbf{w}^{\text{new},k-1} - \alpha \nabla_{\mathbf{w}^{\text{new}}} \ell(\mathbf{w}^{\text{new},k-1}; \mathcal{S}^{\text{new}}) \quad (3)$$

where $\mathbf{w}^{\text{new},0} = \mathbf{w}^{\text{meta}}$. For brevity, we omit the superscripts of model adaption in Eq.(3) and use $\mathbf{g}(\cdot)$ as the loss gradients w.r.t. the given tensor. Hence, without ambiguity, we simplify the notations of Eq.(3) as follows:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \mathbf{g}(\mathbf{w}) \quad (4)$$

Let us now understand where the main memory cost for iterating Eq.(4) comes from. For the sake of clarity, we focus on a feed forward DNNs that consist of L convolutional (conv) layers or fully-connected (fc) layers. A typical layer (see Fig. 2) consists of two operations: (i) a linear operation with trainable parameters, e.g., convolution or affine; (ii) a parameter-free non-linear operation (may not exist in certain layers), where we consider max-pooling or ReLU-styled (ReLU, LeakyReLU) activation functions in this paper.

Take a network consisting of conv layers only as an example. The memory requirements for storing the activations $\mathbf{x}_l \in \mathbb{R}^{C_l \times H_l \times W_l}$ as well as the convolution weights $\mathbf{w}_l \in \mathbb{R}^{C_l \times C_{l-1} \times S_l \times S_l}$ of layer l in words can be determined as

$$m(\mathbf{x}_l) = C_l H_l W_l, \quad m(\mathbf{w}_l) = C_l C_{l-1} S_l^2$$

where C_{l-1} , C_l , H_l , and W_l stand for input channel number, output channel number, height and width of layer l , respectively; S_l stands for the kernel size. The detailed memory and computation demand analysis as provided in Appendix A reveals that the by far largest memory requirement is neither attributed to determining the activations \mathbf{x}_l in the forward path nor to determining the gradients of the activations $\mathbf{g}(\mathbf{x}_l)$ in the backward path. Instead, the memory bottleneck lies in the computation of the weight gradients $\mathbf{g}(\mathbf{w}_l)$, which requires the availability of the activations \mathbf{x}_{l-1} from the forward path. Following Eq.(17) in Appendix A, the necessary memory in words is

$$\sum_{1 \leq l \leq L} m(\mathbf{x}_{l-1}) \quad (5)$$

Tab. 1 summarizes the memory consumption and the total computation of the commonly used few-shot learning backbone models [10, 27]. The requirements are based on the detailed analysis in Appendix A. We can draw two intermediate conclusions.

- The total computation of adaptation (training) is approximately $2.7\times$ to $3\times$ larger compared to inference. Yet the peak dynamic memory of training is far larger, $47\times$ to $103\times$ over inference. The peak dynamic memory consumption of training is also significantly higher than the static memory consumption from the model and the training samples in few-shot learning.
- To enable adaptation for memory-constrained IoT devices, we need to find some way of getting rid of the major dynamic memory contribution in Eq.(5).

3 METHOD

This section presents p-Meta, a new meta learning scheme that enables memory-efficient few-shot learning on unseen tasks.

3.1 p-Meta Overview

We first provide an overview of p-Meta and introduce its main concepts, namely selecting critical gradients, using a hierarchical approach to determine adaption-critical layers and channels, and using a mixture of static and dynamic selection mechanisms.

Principles. We impose *structured sparsity* on the gradients $\mathbf{g}(\mathbf{w}_l)$ such that the corresponding tensor dimensions of \mathbf{x}_l do not need to be saved. There are other options to reduce the dominant memory demand in Eq.(5). They are inapplicable for the reasons below.

- One may trade-off computation and memory by recomputing activations \mathbf{x}_{l-1} when needed for determining \mathbf{w}_l , see for example [4, 18]. Due to the limited processing abilities of IoT devices, we exclude this option.
- It is also possible to prune activations \mathbf{x}_{l-1} . Yet based on our experiments in Appendix B.3, imposing sparsity on \mathbf{x}_{l-1} hugely degrades few-shot learning accuracy as this causes error accumulation along the propagation, see also [29].
- Note that unstructured sparsity, as proposed in [11, 40], does not in general lead to memory savings, since there is a very small probability that all weight gradients for which an element of \mathbf{x}_{l-1} is necessary have been pruned.

We impose sparsity on the gradients in a hierarchical manner.

- **Selecting adaption-critical layers.** We first impose layer-by-layer sparsity on $\mathbf{g}(\mathbf{w}_l)$. It is motivated by previous results showing that manual freezing of certain layers does no harm

to few-shot learning accuracy [26, 28]. Layer-wise sparsity reduces the number of layers whose weights need to be updated. We determine the adaptation-critical layers from the meta-trained *layer-wise sparse learning rates*.

- **Selecting adaption-critical channels.** We further reduce the memory demand by imposing sparsity on $\mathbf{g}(\mathbf{w}_l)$ within each layer. Noting that calculating $\mathbf{g}(\mathbf{w}_l)$ needs both the input channels \mathbf{x}_{l-1} and the output channels $\mathbf{g}(\mathbf{y}_l)$, we enforce sparsity on both of them. Input channel sparsity decreases memory and computation overhead, whereas output channel sparsity improves few-shot learning accuracy and reduces computation. We design a novel *meta attention mechanism* to *dynamically* determine adaptation-critical channels. They take as inputs \mathbf{x}_{l-1} and $\mathbf{g}(\mathbf{y}_l)$ and determine adaptation-critical channels during few-shot learning, based on the given few data samples from new unseen tasks. Dynamic channel-wise learning rates as determined by meta attention yield a significant higher accuracy than static channel-wise learning rate (see Sec. 4.4).

Memory Reduction. The reduced memory demand due to our hierarchical approach can be seen in Eq.(17) in Appendix A:

$$\sum_{1 \leq l \leq L} \hat{\alpha}_l \mu_l^{\text{fw}} m(\mathbf{x}_{l-1})$$

where $\hat{\alpha}_l \in \{0, 1\}$ is the mask from the static selection of critical layers and $0 \leq \mu_l^{\text{fw}} \leq 1$ denotes the relative amount of dynamically chosen input channels.

Next, we explain how p-Meta selects adaptation-critical layers (Sec. 3.2) and channels within layers (Sec. 3.3) as well as the deployment optimizations (Sec. 3.5) for memory-efficient adaptation.

3.2 Selecting Adaption-Critical Layers by Learning Sparse Inner Step Sizes

This subsection introduces how p-Meta meta-learns adaptation-critical layers to reduce the number of updated layers during few-shot learning. Particularly, instead of manual configuration as in [26, 28], we propose to automate the layer selection process. During meta training, we identify adaptation-critical layers by learning layer-wise sparse inner step sizes (Sec. 3.2.1). Only these critical layers with nonzero step sizes will be updated during on-device adaptation to new tasks (Sec. 3.2.2).

3.2.1 Learning Sparse Inner Step Sizes in Meta Training. Prior work [1] suggests that instead of a global fixed inner step size α , learning the inner step sizes α for each layer and each gradient descent step improves the generalization of meta learning, where $\alpha = \alpha_{1:L}^{1:K} \geq \mathbf{0}$. We utilize such learned inner step sizes to infer layer importance for adaptation. We learn the inner step sizes α in the outer loop of meta-training while fixing them in the inner loop.

Learning Layer-wise Inner Step Sizes. We change the inner loop of Eq.(1) to incorporate the per-layer inner step sizes:

$$\mathbf{w}_l^{i,k} = \mathbf{w}_l^{i,k-1} - \alpha_l^k \nabla_{\mathbf{w}_l} \ell(\mathbf{w}_{1:L}^{i,k-1}; \mathcal{S}^i) \quad (6)$$

where $\mathbf{w}_l^{i,k}$ is the weights of layer l at step k optimized on task i (dataset \mathcal{S}^i). In the outer loop, weights \mathbf{w} are still optimized as

$$\mathbf{w} \leftarrow \mathbf{w} - \beta \nabla_{\mathbf{w}} \sum_i \ell(\mathbf{w}^i; \mathcal{Q}^i) \quad (7)$$

where $\mathbf{w}^i = \mathbf{w}^{i,K} = \mathbf{w}_{1:L}^{i,K}$, which is a function of α . The inner step sizes α are then optimized as

$$\alpha \leftarrow \alpha - \beta \nabla_{\alpha} \sum_i \ell(\mathbf{w}^i; \mathcal{Q}^i) \quad (8)$$

Imposing Sparsity on Inner Step Sizes. To facilitate layer selection, we enforce sparsity in α , i.e., encouraging a subset of layers to be selected for updating. Specifically, we add a Lasso regularization term in the loss function of Eq.(8) when optimizing α . Hence, the final optimization of α in the outer loop is formulated as

$$\alpha \leftarrow \alpha - \beta \nabla_{\alpha} \left(\sum_i \ell(\mathbf{w}^i; \mathcal{Q}^i) + \lambda \sum_{l,k} m(\mathbf{x}_{l-1}) \cdot |\alpha_l^k| \right) \quad (9)$$

where λ is a positive scalar to control the ratio between two terms in the loss function. We empirically set $\lambda = 0.001$. $|\alpha_l^k|$ is re-weighted by $m(\mathbf{x}_{l-1})$, which denotes the necessary memory in Eq.(5) if only updating the weights in layer l .

3.2.2 Exploiting Sparse Inner Step Sizes for On-device Adaptation.

We now explain how to apply the learned α to save memory during on-device adaptation. After deploying the meta-trained model to IoT devices for adaptation, at updating step k , for layers with $\alpha_l^k = 0$, the activations (i.e., their inputs) \mathbf{x}_{l-1} need not be stored, see Eq.(16) and Eq.(17) in Appendix A. In addition, we do not need to calculate the corresponding weight gradients $\mathbf{g}(\mathbf{w}_l)$, which saves computation, see Eq.(18) in Appendix A.

3.3 Selecting Adaption-Critical Channels within Layers via Sparse Meta Attention

This subsection explains how p-Meta learns a novel meta attention mechanism in each layer to dynamically select adaptation-critical channels for further memory saving in few-shot learning. Despite the widespread adoption of channel-wise attention for inference [5, 23], we make the first attempt to use attention for memory-efficient training (few-shot learning in our case). For each layer, its meta attention outputs a dynamic channel-wise sparse attention score based on the samples from new tasks. The sparse attention score is used to re-weight (also sparsify) the weight gradients. Therefore, by calculating only the nonzero gradients of critical weights within a layer, we can save both memory and computation. We first present our meta attention mechanism during meta training (Sec. 3.3.1) and then show its usage for on-device model adaptation (Sec. 3.3.2).

3.3.1 Learning Sparse Meta Attention in Meta Training. Since mainstream backbones in meta learning use small kernel sizes (1 or 3), we design the meta attention mechanism channel-wise. Fig. 3 illustrates the attention design during meta-training.

Learning Meta Attention. The attention mechanism is as follows.

- We assign an attention score to the weight gradients of layer l in the inner loop of meta training. The attention scores are expected to indicate which weights/channels are important and thus should be updated in layer l .
- The attention score is obtained from two attention modules: one taking \mathbf{x}_{l-1} as input in the forward pass, and the other taking $\mathbf{g}(\mathbf{y}_l)$ as input during the backward pass. We use \mathbf{x}_{l-1} and $\mathbf{g}(\mathbf{y}_l)$ to calculate the attention scores because they are used to compute the weight gradients $\mathbf{g}(\mathbf{w}_l)$.

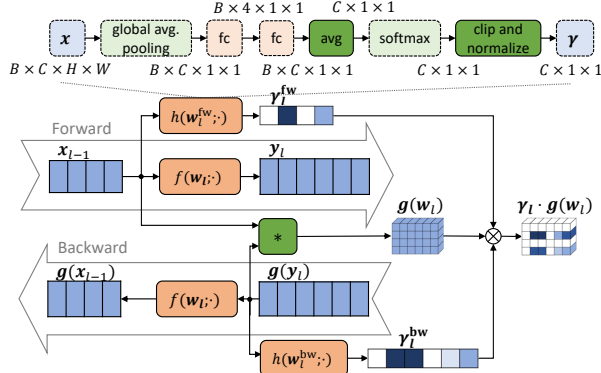


Figure 3: Meta attention of layer l during meta-training. The blue blocks correspond to tensors; the orange blocks correspond to computation units with parameters, and the green ones without. Each column of a tensor corresponds to one channel. The input tensor x_{l-1} has 4 channels; the output tensor y_l has 6 channels. The other dimensions (e.g., height, width and batch) are omitted here. The green block with $*$ stands for the operations involved to compute $g(w_l)$. In order to compute the gradients of the parameters in meta attention, i.e., w_l^{fw} and w_l^{bw} , the full dense gradients $g(w_l)$ are computed during meta-training, and then are masked by γ_l . An example meta attention module for a conv layer is shown in the upper part. B denotes the batch size. The newly added blocks related to the inference attention in [5] are marked with solid lines.

Concretely, we define the forward and backward attention scores for a conv layer as,

$$\gamma_l^{fw} = h(w_l^{fw}; x_{l-1}) \in \mathbb{R}^{C_{l-1} \times 1 \times 1} \quad (10)$$

$$\gamma_l^{bw} = h(w_l^{bw}; g(y_l)) \in \mathbb{R}^{C_l \times 1 \times 1} \quad (11)$$

where $h(\cdot; \cdot)$ stands for the meta attention module, and w_l^{fw} and w_l^{bw} are the parameters of the meta attention modules. The overall (sparse) attention scores $\gamma_l \in \mathbb{R}^{C_l \times C_{l-1} \times 1 \times 1}$ and is computed as,

$$\gamma_{l,ba11} = \gamma_{l,a11}^{fw} \cdot \gamma_{l,b11}^{bw} \quad (12)$$

In the inner loop, for layer l , step k and task i , γ_l is (broadcasting) multiplied with the dense weight gradients to get the sparse ones,

$$\gamma_l^{i,k} \odot \nabla_{w_l} \ell(w_{1:L}^{i,k-1}; S^i) \quad (13)$$

The weights are then updated by,

$$w_l^{i,k} = w_l^{i,k-1} - \alpha_l^k (\gamma_l^{i,k} \odot \nabla_{w_l} \ell(w_{1:L}^{i,k-1}; S^i)) \quad (14)$$

Let all attention parameters be $w^{\text{atten}} = \{w_l^{fw}, w_l^{bw}\}_{l=1}^L$. The attention parameters w^{atten} are optimized in the outer loop as,

$$w^{\text{atten}} \leftarrow w^{\text{atten}} - \beta \nabla_{w^{\text{atten}}} \sum_i \ell(w^i; Q^i) \quad (15)$$

Note that we use a dense forward path and a dense backward path in both meta-training and on-device adaptation, as shown in Fig. 3.

Algorithm 1: Clip and normalization

Input: softmax output (normalized) $\pi \in \mathbb{R}^C$, clip ratio ρ

Output: sparse γ

- 1 Sort π in ascending order and get sorted indices $d_{1:C}$;
 - 2 Find the smallest c such that $\sum_{i=1}^c \pi_{d_i} \geq \rho$;
 - 3 Set $\pi_{d_{1:c}}$ as 0; // if $\rho = 0$, do nothing
 - 4 Normalize $\gamma = \pi / \sum \pi$;
 - 5 Re-scale $\gamma = \gamma \cdot C$; // keeping step sizes' magnitude
-

Algorithm 2: p-Meta

Input: meta-training task distribution $p(T)$, backbone F with initial weights w , meta attention parameters w^{atten} , inner step sizes α , outer step sizes β

Output: meta-trained weights w , meta-trained meta attention parameters w^{atten} , meta-trained sparse inner step sizes α

- 1 **while not done do**
 - 2 Sample a batch of I tasks $T^i \sim p(T)$;
 - 3 **for** $i \leftarrow 1$ **to** I **do**
 - 4 Update w^i in K gradient descent steps with (14);
 - 5 Update w with (7);
 - 6 Update inner step sizes α with (9);
 - 7 Update attention parameters w^{atten} with Eq.(15);
-

That is, the attention scores γ_l^{fw} and γ_l^{bw} are only calculated locally and will not affect y_l during forward and $g(x_{l-1})$ during backward.

Meta Attention Module Design. Fig. 3 (upper part) shows an example meta attention module. We adapt the inference attention modules used in [5, 23], yet with the following modifications.

- Unlike inference attention that applies to a single sample, training may calculate the averaged loss gradients based on a batch of samples. Since $g(w_l)$ does not have a batch dimension, the input to softmax function is first averaged over the batch data, see in Fig. 3.
- We enforce sparsity on the meta attention scores such that they can be utilized to save memory and computation in few-shot learning. The original attention in [5, 23] outputs normalized scales in $[0, 1]$ from softmax. We clip the output with a clip ratio $\rho \in [0, 1]$ to create zeros in γ . This way, our meta attention modules yield batch-averaged sparse attention scores γ_l^{fw} and γ_l^{bw} . Alg. 1 shows this clipping and re-normalization process. Note that Alg. 1 is not differentiable. Hence we use the straight-through-estimator for its backward propagation in meta training.

3.3.2 Exploiting Meta Attention for On-device Adaptation. We now explain how to apply the meta attention to save memory during on-device few-shot learning. Note that the parameters in the meta attention modules are fixed during few-shot learning. Assume that at step k , layer l has a nonzero step size α_l^k . In the forward pass, we only store a sparse tensor $\gamma_l^{fw} \cdot x_{l-1}$, i.e., its channels are stored only if they correspond to nonzero entries in γ_l^{fw} . This reduces memory

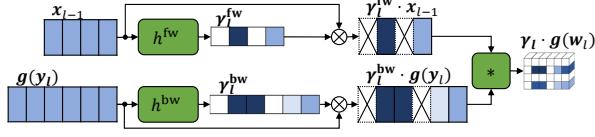


Figure 4: Meta attention of layer l during on-device few-shot learning. Note that “Forward” part and “Backward” part are the same as Fig. 3, which are omitted for simplicity. Meta attention modules are not optimized during few-shot learning, thus are expressed as parameter-free functions h_l^{fw} and h_l^{bw} . The input x_{l-1} stored during forward path is a sparse re-weighted tensor.

consumption as shown in Eq.(17) in Appendix A. Similarly, in the backward pass, we get a channel-wise sparse tensor $\gamma_l^{bw} \cdot g(y_l)$. Since both sparse tensors are used to calculate the corresponding nonzero gradients in $g(w_l)$, the computation cost is also reduced, see Eq.(18) in Appendix A. We plot the meta attention during on-device adaptation in Fig. 4.

3.4 Summary of p-Meta

Alg. 2 shows the overall process of p-Meta during meta-training. The final meta-trained weights \mathbf{w} from Alg. 2 are assigned to \mathbf{w}^{meta} , see Sec. 2. The meta-trained backbone model $F(\mathbf{w}^{\text{meta}})$, the sparse inner step sizes α , and the meta attention modules will be then deployed on edge devices and used to conduct a memory-efficient few-shot learning.

3.5 Deployment Optimization

To further reduce the memory during few-shot learning, we propose gradient accumulation during backpropagation and replace batch normalization in the backbone with group normalization.

3.5.1 Gradient Accumulation. In standard few-shot learning, all the new samples (e.g., 25 for 5-way 5-shot) are fed into the model as one batch. To reduce the peak memory due to large batch sizes, we conduct few-shot learning with gradient accumulation (GA).

GA is a technique that (i) breaks a large batch into smaller partial batches; (ii) sequentially forward/backward propagates each partial batches through the model; (iii) accumulates the loss gradients of each partial batch and get the final averaged gradients of the full batch. Note that GA does not increase computation, which is desired for low-resource platforms. We evaluate the impact of different sample batch sizes in GA in Appendix B.2.

3.5.2 Group Normalization. Mainstream backbones in meta learning typically adopt batch normalization layers. Batch normalization layers compute the statistical information in each batch, which is dependent on the sample batch size. When using GA with different sample batch sizes, the inaccurate batch statistics can degrade the training performance (see Appendix B.1). As a remedy, we use group normalization [43], which does not rely on batch statistics (i.e., independent of the sample batch size). We also apply meta attention on group normalization layers when updating their weights.

The only difference w.r.t. conv and fc layers is that the stored input tensor (also the one used for the meta attention) is not x_{l-1} , but its normalized version.

4 EVALUATION

This section presents the evaluations of p-Meta on standard few-shot image classification and reinforcement learning benchmarks.

4.1 General Experimental Settings

Compared Methods. We test the meta learning algorithms below.

- MAML [10]: the original model-agnostic meta learning.
- ANIL [28]: update the last layer only in few-shot learning.
- BOIL [26]: update the body except the last layer.
- MAML++ [1]: learn a per-step per-layer step sizes α .
- p-Meta (3.2): can be regarded as a sparse version of MAML++, since it learns a sparse α with our methods in Sec. 3.2.
- p-Meta (3.2+3.3): the full version of our methods which include the meta attention modules in Sec. 3.3.

For fair comparison, all the algorithms are re-implemented with the deployment optimization in Sec. 3.5.

Implementation. The experiments are conducted with tools provided by TorchMeta [6, 7]. Particularly, the backbone is meta-trained with full sample batch size (e.g., 25 for 5-way 5-shot) on meta training dataset. After each meta training epoch, the model is tested (i.e., few-shot learned) on meta validation dataset. The model with the highest validation performance is used to report the final few-shot learning results on meta test dataset. We follow the same process as TorchMeta [6, 7] to build the dataset. During few-shot learning, we adopt a sample batch size of 1 to verify the model performance under the most strict memory constraints.

In p-Meta, meta attention is applied to all conv, fc, and group normalization layers, except the last output layer, because (i) we find modifying the last layer’s gradients may decrease accuracy; (ii) the final output is often rather small in size, resulting in little memory saving even if imposing sparsity on the last layer. Without further notations, we set $\rho = 0.3$ in forward attention, and $\rho = 0$ in backward attention across all layers, as the sparsity of γ_l^{bw} almost has no effect on the memory saving.

Metrics. We compare the peak memory and MACs of different algorithms. Note that the reported peak memory and MACs for p-Meta also include the consumption from meta attention, although they are rather small related to the backward propagation.

4.2 Performance on Image Classification

Settings. We test on standard few-shot image classification tasks (both in-domain and cross-domain). We adopt two common backbones, “4Conv” [10] which has 4 conv blocks with 32 channels in each block, and “ResNet12” [27] which contains 4 residual blocks with {64, 128, 256, 512} channels in each block respectively. We replace the batch normalization layers with group normalization layers, as discussed in Sec. 3.5.2. We experiment in both 5-way 1-shot and 5-way 5-shot settings. We train the model on MiniImageNet [39] (both meta training and meta validation dataset) with 100 meta epochs. In each meta epoch, 1000 random tasks are drawn from the task distribution. The task batch size is set to 4 in general,

Table 2: Few-shot image classification results on 4Conv and ResNet12. All methods are meta-trained on MiniImageNet, and are few-shot learned on the reported datasets: MiniImageNet, TieredImageNet, and CUB (denoted by Mini, Tiered, and CUB in the table). The total computation (# GMACs) and the peak memory (MB) during few-shot learning are reported based on the theoretical analysis in Appendix A.

Benchmarks		5-way 1-shot					5-way 5-shot				
		Mini	Tiered Accuracy	CUB	Mini GMAC	Mini Memory	Mini	Tiered Accuracy	CUB	Mini GMAC	Mini Memory
4Conv	MAML [10]	46.2%	51.4%	39.7%	0.39	2.06	61.4%	66.5%	55.6%	1.96	2.06
	ANIL [28]	46.4%	51.5%	39.2%	0.14	0.92	60.6%	64.5%	54.2%	0.72	0.92
	BOIL [26]	44.7%	51.3%	42.3%	0.39	2.05	60.5%	65.3%	58.3%	1.96	2.05
	MAML++ [1]	48.2%	53.2%	43.2%	0.39	2.06	63.7%	68.5%	59.1%	1.96	2.06
	p-Meta (3.2)	47.1%	52.3%	41.8%	0.16	1.00	62.9%	68.3%	59.3%	1.34	1.09
	p-Meta (3.2+3.3)	48.8%	53.9%	42.6%	0.15	0.99	65.0%	68.5%	60.2%	1.11	1.04
ResNet12	MAML [10]	51.7%	57.4%	41.3%	37.08	54.69	64.7%	69.6%	53.8%	185.42	54.69
	ANIL [28]	50.3%	56.7%	40.6%	12.42	3.62	62.3%	68.7%	54.0%	62.08	3.62
	BOIL [26]	42.7%	47.7%	44.2%	37.08	54.69	53.6%	59.8%	53.7%	185.42	54.69
	MAML++ [1]	53.1%	58.6%	45.1%	37.08	54.69	68.6%	73.4%	63.9%	185.42	54.69
	p-Meta (3.2)	51.8%	58.3%	40.6%	25.84	17.66	68.8%	72.6%	65.9%	124.15	18.95
	p-Meta (3.2+3.3)	53.6%	59.4%	45.4%	24.02	16.01	69.7%	73.3%	66.6%	116.79	17.17

Table 3: Few-shot reinforcement learning results on 2D navigation and robot locomotion (larger return means better). A MLP with two hidden layers of size 100 is used as the policy model. The total computation (# GMACs) and the peak memory (MB) during few-shot learning are reported based on the theoretical analysis in Appendix A.

Benchmarks	20 Rollouts			20 Rollouts		
	Half-Cheetah Velocity			2D Navigation		
	Return	GMAC	Memory	Return	GMAC	Memory
MAML [10]	-82.2	0.15	0.24	-13.3	0.12	0.21
ANIL [28]	-78.8	0.06	0.09	-13.8	0.04	0.08
BOIL [26]	-76.4	0.15	0.23	-12.4	0.12	0.21
MAML++ [1]	-69.6	0.15	0.24	-17.6	0.12	0.21
p-Meta (3.2)	-65.5	0.11	0.12	-11.2	0.09	0.09
p-Meta (3.2+3.3)	-64.0	0.11	0.11	-11.8	0.09	0.09

except for ResNet12 under 5-way 5-shot settings where we use 2. The model is updated with 5 gradient steps (*i.e.*, $K = 5$) in both inner loop of meta-training and few-shot learning. We use Adam optimizer with cosine learning rate scheduling as [1] for all outer loop updating. The (initial) inner step size α is set to 0.01. The meta-trained model is then tested on three datasets MiniImageNet [39], TieredImageNet [30], and CUB [41] to verify both *in-domain* and *cross-domain* performance.

Results. Tab. 2 shows the accuracy averaged over 5000 new unseen tasks randomly drawn from the meta test dataset. We also report the average number of GMACs and the average peak memory per task according to Appendix A. Clearly, p-Meta almost always yields the highest accuracy in all settings. Note that the comparison between “p-Meta (3.2)” and “MAML++” can be considered as the ablation studies on learning sparse layer-wise inner step sizes proposed in Sec. 3.2. Thanks to the imposed sparsity on α , “p-Meta (3.2)” significantly reduces the peak memory (2.5 \times saving on average and up to 3.1 \times) and the computation burden (1.7 \times saving on average and up to 2.4 \times) over “MAML++”. Note that the imposed sparsity also cause a moderate accuracy drop. However, with the meta attention, “p-Meta (3.2+3.3)” not only notably improves the accuracy but also

further reduces the peak memory (2.7 \times saving on average and up to 3.4 \times) and computation (1.9 \times saving on average and up to 2.6 \times) over “MAML++”. “ANIL” only updates the last layer, and therefore consumes less memory but also yields a substantially lower accuracy.

4.3 Performance on Reinforcement Learning

Settings. To show the versatility of p-Meta, we experiment with two few-shot reinforcement learning problems: 2D navigation and Half-Cheetah robot locomotion simulated with MuJoCo library [37]. For all experiments, we mainly adopt the experimental setup in [10]. We use a neural network policy with two hidden fc layers of size 100 and ReLU activation function. We adopt vanilla policy gradient [42] for the inner loop and trust-region policy optimization [32] for the outer loop. During the inner loop as well as few-shot learning, the agents rollout 20 episodes with a horizon size of 200 and are updated for one gradient step. The policy model is trained for 500 meta epochs, and the model with the best average return during training is used for evaluation. The task batch size is set to 20 for 2D navigation, and 40 for robot locomotion. The (initial) inner step size α is set to 0.1. Each episode is considered as a data sample, and thus the gradients are accumulated 20 times for a gradient step.

Results. Tab. 3 lists the average return averaged over 400 new unseen tasks randomly drawn from simulated environments. We also report the average number of GMACs and the average peak memory per task according to Appendix A. Note that the reported computation and peak memory do not include the estimations of the advantage [9], as they are relatively small and could be done during the rollout. p-Meta consumes a rather small amount of memory and computation, while often obtains the highest return in comparison to others. Therefore, p-Meta can fast adapt its policy to reach the new goal in the environment with less on-device resource demand.

4.4 Ablation Studies on Meta Attention

We study the effectiveness of our meta attention via the following two ablation studies. The experiments are conducted on “4Conv” in both 5-way 1-shot and 5-way 5-shot as Sec. 4.2.

Table 4: Ablation results of meta attention on 4Conv.

ρ		5-way 1-shot			5-way 5-shot		
fw	bw	Mini	Tiered	CUB	Mini	Tiered	CUB
x	x	47.1%	52.3%	41.8%	62.9%	68.3%	59.3%
0	x	48.1%	53.2%	41.7%	64.1%	68.4%	59.0%
x	0	47.8%	53.1%	40.9%	63.9%	68.5%	60.0%
0	0	49.0%	54.2%	43.1%	64.5%	69.2%	60.2%
0	0.3	48.5%	53.4%	42.2%	64.7%	68.2%	59.3%
0.3	0	48.8%	53.9%	42.6%	65.0%	68.5%	60.2%
0.3	0.3	48.7%	53.7%	42.3%	64.5%	68.3%	59.5%
0.5	0.5	48.2%	53.4%	42.7%	64.8%	68.1%	59.1%
α^{Ch}		47.8%	52.8%	41.0%	63.6%	68.1%	58.1%

x: no forward/backward meta attention, i.e., $\gamma_l^{\text{fw}} = 1$ or $\gamma_l^{\text{bw}} = 1$.

α^{Ch} : introducing an input- and output-channel-wise inner step sizes α^{Ch} per layer. We use $\alpha \cdot \alpha^{\text{Ch}}$ as the overall inner step sizes. α^{Ch} is meta-trained as α without imposing sparsity.

Sparsity in Meta Attention. Tab. 4 shows the few-shot classification accuracy with different sparsity settings in the meta attention.

We first do not impose sparsity on γ_l^{fw} and γ_l^{bw} (i.e., set both ρ 's as 0), and adopt forward attention and backward attention separately. In comparison to no meta attention at all, enabling either forward or backward attention improves accuracy. With both attention enabled, the model achieves the best performance.

We then test the effects when imposing sparsity on γ_l^{fw} or γ_l^{bw} (i.e., set $\rho > 0$). We use the same ρ for all layers. We observe a sparse γ_l^{bw} often cause a larger accuracy drop than a sparse γ_l^{fw} . Since a sparse γ_l^{bw} does not bring substantial memory or computation saving (see Appendix A), we use $\rho = 0$ for backward attention and $\rho = 0.3$ for forward attention.

Attention scores γ_l introduce a dynamic channel-wise learning rate according to the new data samples. We further compare meta attention with a static channel-wise learning rate, where the channel-wise learning rate α^{Ch} is meta-trained as the layer-wise inner step sizes in Sec. 3.2 while without imposing sparsity. By comparing “ α^{Ch} ” with “0, 0” in Tab. 4, we conclude that the dynamic channel-wise learning rate yields a significantly higher accuracy.

Layer-wise Updating Ratios. To study the resulted updating ratios across layers, i.e., the layer-wise sparsity of weight gradients, we randomly select 100 new tasks and plot the layer-wise updating ratios, see Fig. 5 Left (1:5). The “4Conv” backbone has 9 layers ($L = 9$), i.e., 8 alternates of conv and group normalization layers, and an fc output layer. As mentioned in Sec. 4.1, we do not apply meta attention to the output layer, i.e., $\gamma_9 = 1$. The used backbone is updated with 5 gradient steps ($K = 5$). We use $\rho = 0.3$ for forward attention, and $\rho = 0$ for backward. Note that Alg. 1 adaptively determines the sparsity of γ_l , which also means different samples may result in different updating ratios even with the same ρ (see Fig. 5). The size of x_{l-1} often decreases along the layers in current DNNs. As expected, the latter layers are preferred to be updated more, since they need a smaller amount of memory for updating. Interestingly, even if with a small ρ ($= 0.3$), the ratio of updated weights is rather small, e.g., smaller than 0.2 in step 3 of 5-way 5-shot. It implies that the outputs of softmax have a large discrepancy, i.e., only a few channels are adaptation-critical for each sample, which in turn verifies the effectiveness of our meta attention mechanism.

We also randomly pair data samples and compute the cosine similarity between their attention scores γ_l . We plot the cosine similarity of step 1 in Fig. 5 Right. The results show that there may exist a considerable variation on the adaptation-critical weights selected by different samples, which is consistent with our observation in Tab. 4, i.e., dynamic learning rate outperforms the static one.

5 RELATED WORK

Meta Learning for Few-Shot Learning. Meta learning is a prevailing solution to few-shot learning [22], where the meta-trained model can learn an unseen task from a few training samples, i.e., data-efficient adaptation. The majority of meta learning methods can be divided into two categories, (i) metric-based methods [34, 35, 39] that learn an embedded metric for classification tasks to map the query samples onto the classes of labeled support samples, (ii) gradient-based methods [1, 10, 26, 28, 38, 40] that learn an initial model (and/or optimizer parameters) such that it can be adapted with gradient information calculated on the new few samples. In comparison to metric-based methods, we focus on gradient-based meta learning methods for their wide applicability in various learning tasks (e.g., regression, classification, reinforcement learning) and the availability of gradient-based training frameworks for low-resource devices [36].

Particularly, we aim at meta training a DNN that allows effective adaptation on memory-constrained devices. Most meta learning algorithms [1, 10, 40] optimize the backbone network for better generalization yet ignore the workload if the meta-trained backbone is deployed to low-resource platforms for model adaptation. Manually fixing certain layers during on-device few-shot learning [26, 28, 33] may also reduce memory and computation, but to a much lesser extent as shown in our evaluations.

Efficient DNN Training. Existing efficient training schemes are mainly designed for high-throughput GPU training on large-scale datasets. A general strategy is to trade memory with computation [4, 18], which is unfit for IoT device with a limited computation capability. An alternative is to sparsify the computational graphs in backpropagation [29]. Yet it relies on massive training iterations on large-scale datasets. Other techniques include layer-wise local training [17] and reversible residual module [12], but they often incur notable accuracy drops.

There are a few studies on DNN training on low-resource platforms, such as updating the last several layers only [25], reducing batch sizes [24], and gradient approximation [15]. However, they are designed for vanilla supervised training, i.e., train and test on the same task. One recent study proposes to update the bias parameters only for memory-efficient transfer learning [2], yet transfer learning is prone to overfitting when trained with limited data [10].

6 CONCLUSION

In this paper, we present p-Meta, a new meta learning scheme for data- and memory-efficient on-device DNN adaptation. It enables structured partial parameter updates for memory-efficient few-shot learning by automatically identifying adaptation-critical weights both layer-wise and channel-wise. Evaluations show a reduction in peak dynamic memory by 2.5 \times on average over the state-of-the-art

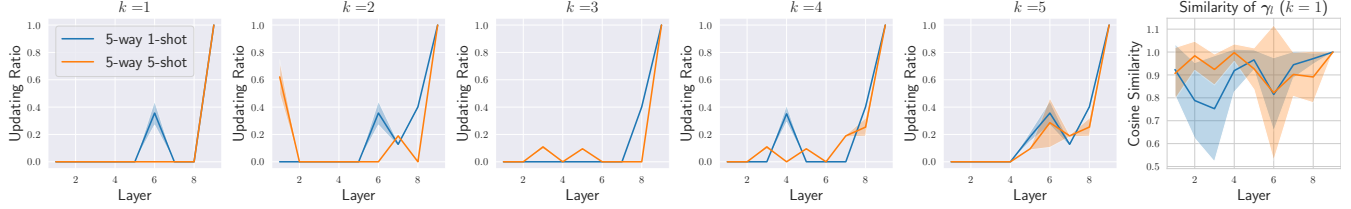


Figure 5: Left (1:5): Layer-wise updating ratios (mean \pm standard deviation) in each updating step. Note that the ratio of updated weights is determined by both static layer-wise inner step sizes $\alpha_{1:L}^{1:K}$ and the dynamic meta attention scores $\gamma_{1:L}$. The layer with an updating ratio of 0 means its $\alpha = 0$. Right: Cosine similarity (mean \pm standard deviation) of $\gamma_{1:L}$ between random pair of data samples. The results are reported in step 1, because all samples are fed into the same initial model in step 1.

few-shot adaptation methods. We envision p-Meta as an early step towards adaptive and autonomous edge intelligence applications.

ACKNOWLEDGEMENT

Part of Zhongnan Qu and Lothar Thiele’s work was supported by the Swiss National Science Foundation in the context of the NCCR Automation. This research was supported by the Lee Kong Chian Fellowship awarded to Zimu Zhou by Singapore Management University. Zimu Zhou is the corresponding author.

REFERENCES

- [1] Antreas Antoniou, Harrison Edwards, and Amos Storkey. 2019. How to train your MAML. In *ICLR*.
- [2] Han Cai, Chuang Gan, Ligeng Zhu, and Song Han. 2020. TinyTL: Reduce Memory, Not Parameters for Efficient On-Device Learning. In *NeurIPS*.
- [3] Beidi Chen, Zichang Liu, Binghui Peng, Zhaozhuo Xu, Jonathan Lingjie Li, Tri Dao, Zhao Song, Anshumali Shrivastava, and Christopher Re. 2021. MONGOOSE: A Learnable LSH Framework for Efficient Neural Network Training. In *ICLR*.
- [4] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. arXiv:1604.06174
- [5] Yinpeng Chen, Xiyang Dai, Mengchen Liu, Dongdong Chen, Lu Yuan, and Zicheng Liu. 2020. Dynamic Convolution: Attention Over Convolution Kernels. In *CVPR*.
- [6] Tristan Deleu. 2018. Model-Agnostic Meta-Learning for Reinforcement Learning in PyTorch. Available at: <https://github.com/tristandeleu/pytorch-maml-rl>.
- [7] Tristan Deleu, Tobias Würfl, Mandana Samiei, Joseph Paul Cohen, and Yoshua Bengio. 2019. Torchmeta: A Meta-Learning library for PyTorch. <https://arxiv.org/abs/1909.06576> Available at: <https://github.com/tristandeleu/pytorch-meta>.
- [8] Lei Deng, Guoqi Li, Song Han, Luping Shi, and Yuan Xie. 2020. Model compression and hardware acceleration for neural networks: a comprehensive survey. *Proc. IEEE* 108, 4 (2020), 485–532.
- [9] Yan Duan, Xi Chen, Rein Houthoofd, John Schulman, and Pieter Abbeel. 2016. Benchmarking Deep Reinforcement Learning for Continuous Control. In *ICML*.
- [10] Chelsea Finn, Pieter Abbeel, and Sergey Levine. 2017. Model-agnostic meta-learning for fast adaptation of deep networks. In *ICML*.
- [11] Dawei Gao, Xiaoxi He, Zimu Zhou, Yongxin Tong, and Lothar Thiele. 2021. Pruning meta-trained networks for on-device adaptation. In *CIKM*.
- [12] Aidan N Gomez, Mengye Ren, Raquel Urtasun, and Roger B Grosse. 2017. The Reversible Residual Network: Backpropagation Without Storing Activations. In *NeurIPS*.
- [13] Taesik Gong, Yeonsu Kim, Jinwoo Shin, and Sung-Ju Lee. 2019. Metasense: few-shot adaptation to untrained conditions in deep mobile sensing. In *SenSys*.
- [14] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. 2016. *Deep learning*. Vol. 1. MIT press Cambridge.
- [15] Mary Gooneratne, Khe Chai Sim, Petr Zadrzail, Andreas Kabel, Françoise Beaufays, and Giovanni Motta. 2020. Low-rank Gradient Approximation For Memory-Efficient On-device Training of Deep Neural Network. In *ICASSP*.
- [16] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. arXiv:1706.02677
- [17] Klaus Greff, Rupesh K. Srivastava, and Jürgen Schmidhuber. 2017. Highway and Residual Networks learn Unrolled Iterative Estimation. In *NeurIPS*.
- [18] Audrunas Gruslys, Remi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. 2016. Memory-Efficient Backpropagation Through Time. In *NeurIPS*.
- [19] Liang-Yan Gui, Yu-Xiong Wang, Deva Ramanan, and José MF Moura. 2018. Few-shot human motion prediction via meta-learning. In *ECCV*.
- [20] Song Han, Huizi Mao, and William J Dally. 2016. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. In *ICLR*.
- [21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *CVPR*.
- [22] Timothy Hospedales, Antreas Antoniou, Paul Micaelli, and Amos Storkey. 2020. Meta-learning in neural networks: a survey. arXiv:2004.05439
- [23] Jie Hu, Li Shen, and Gang Sun. 2018. Squeeze-and-Excitation Networks. In *CVPR*.
- [24] Seulki Lee and Shahriar Nirjon. 2019. Neuro.ZERO: a zero-energy neural network accelerator for embedded sensing and inference systems. In *SenSys*.
- [25] Akhil Mathur, Daniel J. Beutel, Pedro Porto Buarque de Gusmão, Javier Fernandez-Marques, Taner Topal, Xinchu Qiu, Titouan Parcollet, Yan Gao, and Nicholas D. Lane. 2021. On-device Federated Learning with Flower. In *MLSys*.
- [26] Jaehoon Oh, Hyungjun Yoo, ChangHwan Kim, and Se-Young Yun. 2021. BOIL: Towards Representation Change for Few-shot Learning. In *ICLR*.
- [27] Boris N. Oreshkin, Pau Rodriguez, and Alexandre Lacoste. 2018. TADAM: Task dependent adaptive metric for improved few-shot learning. In *NeurIPS*.
- [28] Aniruddh Raghu, Maithra Raghu, Samy Bengio, and Oriol Vinyals. 2020. Rapid learning or feature reuse? Towards understanding the effectiveness of MAML. In *ICLR*.
- [29] Md Aamir Raihan and Tor M. Aamodt. 2020. Sparse Weight Activation Training. In *NeurIPS*.
- [30] Mengye Ren, Eleni Triantafillou, Sachin Ravi, Jake Snell, Kevin Swersky, Joshua B. Tenenbaum, Hugo Larochelle, and Richard S. Zemel. 2018. Meta-Learning for Semi-Supervised Few-Shot Classification. In *ICLR*.
- [31] 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. 2015. ImageNet large scale visual recognition challenge. *International Journal of Computer Vision* 115, 3 (2015), 211–252.
- [32] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. 2015. Trust Region Policy Optimization. In *ICML*.
- [33] Zhiqiang Shen, Zechun Liu, Jie Qin, Marios Savvides, and Kwang-Ting Cheng. 2021. Partial Is Better Than All: Revisiting Fine-tuning Strategy for Few-shot Learning. In *AAAI*.
- [34] Jake Snell, Kevin Swersky, and Richard S. Zemel. 2017. Prototypical Networks for Few-shot Learning. In *NeurIPS*.
- [35] Flood Sung, Yongxin Yang, Li Zhang, Tao Xiang, Philip H. S. Torr, and Timothy M. Hospedales. 2018. Learning to Compare: Relation Network for Few-Shot Learning. In *CVPR*.
- [36] TensorFlow. [n.d.]. On-Device Training with TensorFlow Lite. https://www.tensorflow.org/lite/examples/on_device_training/overview. Accessed: 2022-01-15.
- [37] Emanuel Todorov, Tom Erez, and Yuval Tassa. 2012. Mujoco: A physics engine for model-based control. In *IROS*.
- [38] Eleni Triantafillou, Tyler Zhu, Vincent Dumoulin, Pascal Lamblin, Utku Evci, Kelvin Xu, Ross Goroshin, Carles Gelada, Kevin Swersky, Pierre-Antoine Manzagol, and Hugo Larochelle. 2020. Meta-Dataset: A Dataset of Datasets for Learning to Learn from Few Examples. In *ICLR*.
- [39] Oriol Vinyals, Charles Blundell, Timothy Lillicrap, Koray Kavukcuoglu, and Daan Wierstra. 2016. Matching Networks for One Shot Learning. In *NeurIPS*.
- [40] Johannes Von Oswald, Dominic Zhao, Seijin Kobayashi, Simon Schug, Massimo Caccia, Nicolas Zucchet, and João Sacramento. 2021. Learning where to learn: Gradient sparsity in meta and continual learning. In *NeurIPS*.
- [41] P. Welinder, S. Branson, T. Mita, C. Wah, F. Schroff, S. Belongie, and P. Perona. 2010. *Caltech-UCSD Birds 200*. Technical Report CNS-TR-2010-001. California Institute of Technology.
- [42] Ronald J. Williams. 1992. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Mach. Learn.* 8, 3–4 (may 1992), <https://doi.org/10.1007/BF00992696>
- [43] Yuxin Wu and Kaiming He. 2018. Group Normalization. In *ECCV*.

A MEMORY AND COMPUTATION

In the following, we derive the memory requirement and computation workload for inference and adaptation. We restrict ourselves to a feed-forward network of fully-connected (fc) or convolutional (conv) layers. Note that our analysis focuses on 2D conv layers but can apply to other conv layer types as well. We assume the rectified linear activation function (ReLU) for all layers, denoted as σ . For simplicity, we omit the bias, normalization layers, pooling or strides. We use the notation $m(\mathbf{x})$ to denote the memory demand in words to store tensor \mathbf{x} . The wordlength is denoted as T .

For representing indexed summations we use the Einstein notation. If index variables appear in a term on the right hand side of an equation and are not otherwise defined (free indices), it implies summation of that term over the range of the free indices. If indices of involved tensor elements are out of range, the values of these elements are assumed to be 0.

A.1 Single Layer

We start with a single layer and accumulate the memory and computation for networks with several layers afterwards. Assume the input tensor of a layer is \mathbf{x} , the weight tensor is \mathbf{w} , the result after the linear transformation is \mathbf{y} , and the layer output after the non-linear operator is \mathbf{z} which is also the input to the next layer.

For convolutional layers, we have $\mathbf{x} \in \mathbb{R}^{C_I \times H_I \times W_I}$ and elements x_{cij} , where C_I , H_I , and W_I denote the number of input channels, height and width, respectively. In a similar way, we have $\mathbf{z} \in \mathbb{R}^{C_O \times H_O \times W_O}$ with elements z_{fij} where C_O , H_O , and W_O denote the number of output channels, height and width, respectively. Moreover, $\mathbf{w} \in \mathbb{R}^{C_O \times C_I \times S \times S}$ with elements w_{fcmn} . Therefore,

$$m(\mathbf{x}) = C_I H_I W_I, \quad m(\mathbf{y}) = m(\mathbf{z}) = C_O H_O W_O, \quad m(\mathbf{w}) = C_O C_I S^2$$

For fully connected layers we have $\mathbf{x} \in \mathbb{R}^{C_I}$, $\mathbf{y}, \mathbf{z} \in \mathbb{R}^{C_O}$, and $\mathbf{w} \in \mathbb{R}^{C_O \times C_I}$ with memory demand

$$m(\mathbf{x}) = C_I, \quad m(\mathbf{y}) = m(\mathbf{z}) = C_O, \quad m(\mathbf{w}) = C_O C_I$$

A.1.1 Fully Connected Layer. For inference we derive the relations $y_f = w_{fc} x_c$ and $z_f = \sigma(y_f)$ for all admissible indices $f \in [1, C_O]$. The necessary dynamic memory has a size of about $m(\mathbf{x}) + m(\mathbf{y})$ words and we need about $m(\mathbf{w})$ MAC operations.

For adaptation, we suppose that $\frac{\partial \ell}{\partial z_i}$ is already provided from the next layer. We find $\frac{\partial \ell}{\partial y_i} = \sigma'(y_i) \cdot \frac{\partial \ell}{\partial z_i}$ with $\sigma'(y_i) = \begin{cases} 1 & \text{if } y_i > 0 \\ 0 & \text{if } y_i < 0 \end{cases}$ which leads to $\frac{\partial \ell}{\partial x_i} = w_{ji} \cdot \frac{\partial \ell}{\partial y_j}$. The necessary dynamic memory is about $m(\mathbf{x}) + m(\mathbf{y}) \cdot (1 + \frac{1}{T})$ words, where the last term comes from storing $\sigma'(y_i)$ single bits from the forward path. We need about $m(\mathbf{w})$ MAC operations.

According to the approach described in the paper we are only interested in the partial derivatives $\frac{\partial \ell}{\partial w_{fc}}$ if $\alpha > 0$ for this layer, and if scales $\gamma_f^{\text{bw}} > 0$ and $\gamma_c^{\text{fw}} > 0$ for indices f, c . To simplify the notation, let us define the critical ratios

$$\mu^{\text{fw}} = \frac{\text{number of nonzero elements of } \gamma_c^{\text{fw}}}{C_I}$$

$$\mu^{\text{bw}} = \frac{\text{number of nonzero elements of } \gamma_f^{\text{bw}}}{C_O}$$

which are 1 if all channels are determined to be critical for weight adaptation, and 0 if none of them.

We find $\gamma_f^{\text{bw}} \frac{\partial \ell}{\partial w_{fc}} \gamma_c^{\text{fw}} = (\gamma_f^{\text{bw}} \frac{\partial \ell}{\partial y_f}) \cdot (\gamma_c^{\text{fw}} x_c)$. Therefore, we need $\mu^{\text{fw}} \mu^{\text{bw}} m(\mathbf{w}) + \mu^{\text{fw}} m(\mathbf{x})$ words dynamic memory if $\alpha > 0$ where the latter term considers the information needed from the forward path. We require about $\mu^{\text{fw}} \mu^{\text{bw}} m(\mathbf{w})$ MAC operations if $\alpha > 0$.

A.1.2 Convolutional Layer. The memory analysis for a convolutional layer is very similar, just replacing matrix multiplication by convolution. For inference we find $y_{fij} = w_{fcmn} x_{c,i+m-1,j+n-1}$ and $z_{fij} = \sigma(y_{fij})$ for all admissible indices f, i, j . The necessary dynamic memory has a size of about $\max\{m(\mathbf{x}), m(\mathbf{y})\}$ words when using memory sharing between input and output tensors. We need about $H_O W_O \cdot m(\mathbf{w})$ MAC operations.

For adaptation, we again suppose that $\frac{\partial \ell}{\partial z_{fij}}$ is provided from the next layer. We find $\frac{\partial \ell}{\partial y_{fij}} = \sigma'(y_{fij}) \cdot \frac{\partial \ell}{\partial z_{fij}}$ and get $\frac{\partial \ell}{\partial x_{cij}} = w_{fcmn} \cdot \frac{\partial \ell}{\partial y_{f,i+m-1,j+n-1}}$. The necessary memory is about $\max\{m(\mathbf{x}), m(\mathbf{y})\} + \frac{m(\mathbf{y})}{T}$ words, where the last term comes from storing $\sigma'(y_{fij})$ single bits from the forward path. We need about $H_I W_I \cdot m(\mathbf{w})$ multiply and accumulate operations.

For determining the weight gradients we find $\frac{\partial \ell}{\partial w_{fcmn}} = \frac{\partial \ell}{\partial y_{fij}} \cdot x_{c,i+m-1,j+n-1}$. When considering the scales for filtering, we yield $\gamma_f^{\text{bw}} \frac{\partial \ell}{\partial w_{fcmn}} \gamma_c^{\text{fw}} = (\gamma_f^{\text{bw}} \frac{\partial \ell}{\partial y_{fij}}) \cdot (\gamma_c^{\text{fw}} x_{c,i+m-1,j+n-1})$. As a result, we need $\mu^{\text{fw}} \mu^{\text{bw}} m(\mathbf{w}) + \mu^{\text{fw}} m(\mathbf{x})$ words of dynamic memory if $\alpha > 0$ where the latter term considers the information needed from the forward path. We require about $\mu^{\text{fw}} \mu^{\text{bw}} H_O W_O m(\mathbf{w})$ MAC operations if $\alpha > 0$.

Finally, let us determine the required memory and computation to determine the scales γ_c^{fw} and γ_f^{bw} . According to Fig. 3, we find as an upper bound for the memory $B \cdot (C_I + C_O)$ and $(C_I H_I W_I + 2C_I^2 + C_O H_O W_O + 2C_O^2)$ MAC operations.

A.2 All Layers

The above relations are valid for a single layer. The following relations hold for the overall network. In order to simplify the notation, we consider a network that consists of convolution layers only. Extensions to mixed layers can simply be done.

We suppose L layers with sizes C_l , H_l , W_l and S_l for the number of output channels, output width, output height and kernel size, respectively. We assume that the step-sizes α_l for some iteration of the adaption are given. The memory requirement in words is

$$m(\mathbf{x}_l) = C_l H_l W_l, \quad m(\mathbf{w}_l) = C_l C_{l-1} S_l^2$$

and the word-length is again denoted as T . We define as $\hat{\alpha}_l = \begin{cases} 1 & \text{if } \alpha_l > 0 \\ 0 & \text{if } \alpha_l = 0 \end{cases}$ the mask that determines whether the weight adaptation for this layer is necessary or not.

Let us first look at the forward path. The necessary dynamic memory is about $\max_{0 \leq l \leq L} \{m(\mathbf{x}_l)\}$ words. The amount of MAC operations is $\sum_{1 \leq l \leq L} H_l W_l m(\mathbf{w}_l)$.

The backward path needs only to be evaluated until we reach the first layer where we require the computation of the gradients. We define $l_{\min} = \min\{l \mid \hat{\alpha}_l = 1\}$. For the calculation of the partial derivatives of the activations we need dynamic memory

of $\max_{l_{\min} \leq l \leq L} \{m(\mathbf{x}_l)\} + \frac{1}{T} \sum_{l_{\min} \leq l \leq L} m(\mathbf{x}_l)$ words where the last term is due to storing the derivatives of the ReLU operations. We need about $\sum_{l_{\min}+1 \leq l \leq L} H_{l-1} W_{l-1} m(\mathbf{w}_l)$ MAC operations.

The second contribution of the backward path is for computing the weight gradients. The memory and computation demand of the scales will be neglected as they are much smaller than other contributions. We can determine the necessary dynamic memory as $\max_{1 \leq l \leq L} \{\hat{\alpha}_l \mu_l^{\text{fw}} \mu_l^{\text{bw}} m(\mathbf{w}_l)\} + \sum_{1 \leq l \leq L} \hat{\alpha}_l \mu_l^{\text{fw}} m(\mathbf{x}_{l-1})$, and we need $\sum_{1 \leq l \leq L} \hat{\alpha}_l \mu_l^{\text{fw}} \mu_l^{\text{bw}} H_l W_l m(\mathbf{w}_l)$ MAC operations.

Considering all necessary dynamic memory with memory reuse for an adaptation step, we get an estimation of memory in words

$$\begin{aligned} & \max_{0 \leq l \leq L} \{m(\mathbf{x}_l)\} + \sum_{1 \leq l \leq L} \hat{\alpha}_l m(\mathbf{w}_l) + \\ & + \sum_{1 \leq l \leq L} \hat{\alpha}_l \mu_l^{\text{fw}} m(\mathbf{x}_{l-1}) + \frac{1}{T} \sum_{l_{\min} \leq l \leq L} m(\mathbf{x}_l) \end{aligned} \quad (16)$$

if we accumulate the weight gradients before doing an SGD step and re-use some memory during back-propagation. More elaborate memory re-use can be used to slightly sharpen the bounds without a major improvement. For conventional training, each parameter is in 32-bit floating point format, *i.e.*, one word corresponds to 32-bit. As discussed in Sec. 2, we only consider max-pooling and ReLU-styled activation as the σ function. The wordlength T in Eq.(16) is set as 16 for max-pooling, and 32 for ReLU-styled activation. One can see that under the typical assumptions for network parameters, the above memory requirement in words is dominated by

$$\sum_{1 \leq l \leq L} \hat{\alpha}_l \mu_l^{\text{fw}} m(\mathbf{x}_{l-1}) \quad (17)$$

The necessary storage between the forward and backward path is reduced proportionally to μ_l^{fw} with factor $m(\mathbf{x}_{l-1})$.

Finally, the amount of MAC computations can be estimated as

$$\sum_{1 \leq l \leq L} H_l W_l m(\mathbf{w}_l) (1 + \hat{\alpha}_l \mu_l^{\text{fw}} \mu_l^{\text{bw}}) + \sum_{l_{\min} \leq l \leq L} H_{l-1} W_{l-1} m(\mathbf{w}_l) \quad (18)$$

while neglecting lower order terms. Here it is important to note that all terms are of similar order. The approach used in the paper does not determine a trade-off between computation and memory, but reduces the amount of MAC operations. This reduction is less than the reduction in required dynamic memory.

B OTHER EXPERIMENTS

B.1 Pooling & Normalization

In this section, we test the backbone network with different types of pooling and normalization. Without further notations in the following experiments, we meta-train our “4Conv” backbone on MiniImageNet with full batch sizes, and conduct few-shot learning with gradient accumulation with a batch size of 1, as in Sec. 4. Here, we report the results with the original “MAML” method [10] in Tab. 5. Clearly, the discrepancy of batch statistics between meta-training phase and few-shot learning phase causes a large accuracy loss in batch normalization layers. Batch normalization works only if few-shot learning uses full batch sizes, *i.e.*, without gradient accumulation, which however does not fit in our memory-constrained scenarios (see Sec. 3.5.1). In addition, max-pooling performs better than average-pooling. We thus use group normalization and max-pooling in our backbone model, see Sec. 4.

Table 5: Comparison between different pooling and normalization layers.

4Conv		5-way 1-shot		
Pooling	Norm.	Mini	Tiered	CUB
Average	Batch	25.3%	27.2%	26.1%
Average	Group	45.8%	50.3%	40.2%
Max	Batch	27.6%	28.9%	26.5%
Max	Group	46.2%	51.4%	39.9%

Table 6: Ablation results of sample batch sizes.

Batch Size	5-way 1-shot			5-way 5-shot		
	1	2	5	1	5	25
Mini	48.8%	48.7%	48.3%	65.0%	65.1%	64.7%
Tiered	53.9%	53.6%	54.3%	68.5%	68.9%	68.1%
CUB	42.6%	42.1%	42.4%	60.2%	59.5%	60.6%

Table 7: Ablation results of sparse \mathbf{x}_{l-1} and sparse $\mathbf{g}(\mathbf{y}_l)$.

$\rho = 0.3$		5-way 1-shot			5-way 5-shot		
fw	bw	Mini	Tiered	CUB	Mini	Tiered	CUB
x	x	47.1%	52.3%	41.8%	62.9%	68.3%	59.3%
$\mathbf{g}(\mathbf{w}_l)$	x	48.2%	53.6%	41.2%	63.6%	69.0%	59.0%
\mathbf{x}_{l-1}	x	37.4%	37.9%	35.4%	47.9%	49.3%	42.5%
x	$\mathbf{g}(\mathbf{w}_l)$	48.0%	53.0%	42.6%	64.0%	67.8%	59.9%
x	$\mathbf{g}(\mathbf{y}_l)$	22.8%	21.1%	20.6%	20.7%	21.0%	20.4%

x: no forward/backward (sparse) meta attention, *i.e.*, $\gamma_l^{\text{fw}} = 1$ or $\gamma_l^{\text{bw}} = 1$.

B.2 Sample Batch Size

In this section, we show the effects brought from different sample batch sizes. During few-shot learning phase, gradient accumulation is applied to fit in different on-device memory constraints. We report the accuracy when adopting different sample batch sizes in gradient accumulation. Although group normalization eliminates the variance of batch statistics, adopting different batch sizes may still result in diverse performance due to the batch-averaged scores in meta attention. The results in Tab. 6 show that different batch sizes yield a similar accuracy level, which indicates that our meta attention module is relatively robust to batch sizes.

B.3 Sparse \mathbf{x} and Sparse $\mathbf{g}(\mathbf{y})$

Our meta attention modules take \mathbf{x}_{l-1} and $\mathbf{g}(\mathbf{y}_l)$ as inputs, and output attention scores which are used to create sparse $\mathbf{g}(\mathbf{w}_l)$. However, applying the resulted sparse attention scores on \mathbf{x}_{l-1} and $\mathbf{g}(\mathbf{y}_l)$ can also bring memory and computation benefits, as discussed in Sec. 3.1. We conduct the ablations when multiplying attention scores γ_l^{fw} and γ_l^{bw} on $\mathbf{g}(\mathbf{w}_l)$ (also the one used in the main text), or on \mathbf{x}_{l-1} and $\mathbf{g}(\mathbf{y}_l)$ respectively. The results in Tab. 7 show that a channel-wise sparse \mathbf{x}_{l-1} hugely degrades the performance, in comparison to only imposing sparsity on $\mathbf{g}(\mathbf{w}_l)$ while using a dense \mathbf{x}_{l-1} in the forward pass. In addition, directly adopting a sparse $\mathbf{g}(\mathbf{y}_l)$ in backpropagation may even cause non-convergence in few-shot learning. We think this is due to the fact that the error accumulates along the propagation when imposing sparsity on \mathbf{x}_{l-1} or $\mathbf{g}(\mathbf{y}_l)$.