Supplementary Material
TinyTrain: Deep Neural Network Training at the Extreme Edge

Table of Contents

A Detailed Experimental Setup **16**
A.1 Datasets . . . 16
A.2 Model Architectures . . . 16
A.3 Training Details . . . 16
A.4 Details for Evaluation Setup . . . 17
A.5 Baselines . . . 18

B Details of Sampling Algorithm during Meta-Testing **18**
B.1 Sampling Algorithm during Meta-Testing . . . 18
B.2 Sample Statistics during Meta-Testing . . . 19

C Fine-tuning Procedure during Meta-Testing **19**

D System Implementation **20**

E Additional Results **21**
E.1 Memory- and Compute-aware Analysis . . . 21
E.2 Pairwise Comparison among Different Channel Selection Schemes . . . 21
E.3 End-to-End Latency Breakdown of *TinyTrain* and *SparseUpdate* . . . 22
E.4 Impact of Meta-Training . . . 22
E.5 Robustness of Dynamic Channel Selection . . . 23

F Further Analysis and Discussion **30**
F.1 Further Analysis of Calculating Fisher Information on Activations . . . 30
F.2 Cost Analysis of Meta-Training . . . 30

G Extended Related Work **31**
G.1 On-Device Training . . . 31
G.2 Few-Shot Learning . . . 31

A DETAILED EXPERIMENTAL SETUP

This section provides additional information on the experimental setup.

A.1 DATASETS

Following the conventional setup for evaluating cross-domain FSL performances on MetaDataset in prior arts (Hu et al., 2022; Triantafillou et al., 2020; Guo et al., 2020), we use *MiniImageNet* (Vinyals et al., 2016) for **Meta-Train** and the non-ILSVRC datasets in MetaDataset (Triantafillou et al., 2020) for **Meta-Test**. Specifically, MiniImageNet contains 100 classes from ImageNet-1k, split into 64 training, 16 validation, and 20 testing classes. The resolution of the images is downsampled to 84×84. The MetaDataset used as **Meta-Test datasets** consists of nine public image datasets from a variety of domains, namely *Traffic Sign* (Houben et al., 2013), *Omniglot* (Lake et al., 2015), *Aircraft* (Maji et al., 2013), *Flowers* (Nilsback and Zisserman, 2008), CUB (Welinder et al., 2011), DTD (Cimpoi et al., 2014), QDraw (Jongejan et al., 2016), Fungi (Schroeder and Cui, 2018), and COCO (Lin et al., 2014). Note that the ImageNet dataset is excluded as it is already used for pre-training the models during the meta-training phase, which makes it an in-domain dataset. We showcase the robustness and generality of our approach to the challenging cross-domain few-shot learning (CDFSL) problem via extensive evaluation of these datasets. The details of each target dataset employed in our study are described below.

The **Traffic Sign** (Houben et al., 2013) dataset consists of 50,000 images out of 43 classes regarding German road signs.

The **Omniglot** (Lake et al., 2015) dataset has 1,623 handwritten characters (*i.e.* classes) from 50 different alphabets. Each class contains 20 examples.

The **Aircraft** (Maji et al., 2013) dataset contains images of 102 model variants with 100 images per class.

The **VGG Flowers (Flower)** (Nilsback and Zisserman, 2008) dataset is comprised of natural images of 102 flower categories. The number of images in each class ranges from 40 to 258.

The **CUB-200-2011 (CUB)** (Welinder et al., 2011) dataset is based on the fine-grained classification of 200 different bird species.

The **Describable Textures (DTD)** (Cimpoi et al., 2014) dataset comprises 5,640 images organised according to a list of 47 texture categories (classes) inspired by human perception.

The **Quick Draw (QDraw)** (Jongejan et al., 2016) is a dataset consisting of 50 million black-and-white drawings of 345 categories (classes), contributed by players of the game Quick, Draw!

The **Fungi** (Schroeder and Cui, 2018) dataset is comprised of around 100K images of 1,394 wild mushroom species, each forming a class.

The **MSCOCO (COCO)** (Lin et al., 2014) dataset is the train2017 split of the COCO dataset. COCO contains images from Flickr with 1.5 million object instances of 80 classes.

A.2 MODEL ARCHITECTURES

Following (Lin et al., 2022), we employ optimised DNN architectures designed to be used in resource-limited IoT devices, including **MCUNet** (Lin et al., 2020), **MobileNetV2** (Sandler et al., 2018), and **ProxylessNASNet** (Cai et al., 2019). The DNN models are pre-trained using ImageNet (Deng et al., 2009). Specifically, the backbones of MCUNet (using the 5FPS ImageNet model), MobileNetV2 (with the 0.35 width multiplier), and ProxylessNAS (with a width multiplier of 0.3) have 23M, 17M, 19M MACs and 0.48M, 0.25M, 0.33M parameters, respectively. Note that MACs are calculated based on an input resolution of 128×128 with an input channel dimension of 3. The basic statistics of the three DNN architectures are summarised in Table 3.

A.3 TRAINING DETAILS

We adopt a common training strategy to meta-train the pre-trained DNN backbones, which helps us avoid over-engineering the training process for each dataset and architecture (Hu et al., 2022).

Table 3: The statistics of our employed DNN architectures.

Model	**Param**	**MAC**	**# Layers**	**# Blocks**
MCUNet	0.46M	22.5M	42	13
MobileNetV2	0.29M	17.4M	52	17
ProxylessNASNet	0.36M	19.2M	61	20

Specifically, we meta-train the backbone for 100 epochs. Each epoch has 2000 episodes/tasks. A warm-up and learning rate scheduling with cosine annealing are used. The learning rate increases from 10^{-6} to 5×10^{-5} in 5 epochs. Then, it decreases to 10^{-6}. We use SGD with momentum as an optimiser.

A.4 Details for Evaluation Setup

To evaluate the cross-domain few-shot classification performance, we sample 200 different tasks from the test split for each dataset. Then, as key performance metrics, we first use testing accuracy on unseen samples of a new domain as the target dataset. Note that the number of classes and support/query sets are sampled uniformly at random based on the dataset specifications. In addition, we analytically calculate the computation cost and memory footprint required for the forward pass and backward pass (*i.e.* model parameters, optimisers, activations). For the memory footprint of the backward pass, we include (1) model memory for the weights to be updated, (2) optimiser memory for gradients, and (3) activations memory for intermediate outputs for weights update. For the computational cost, as in (Xu et al., 2022), we report the number of MAC operations of the backward pass, which incurs 2× more MAC operations than the forward pass (inference). Also, we measure latency and energy consumption to perform end-to-end training of a deployed DNN on the edge device. We deploy *TinyTrain* and the baselines on a tiny edge device, Pi Zero. To measure the end-to-end training time and energy consumption, we include the time and energy used to: (1) load a pre-trained model, (2) perform training using all the samples (*e.g.* 25) for a certain number of iterations (*e.g.* 40). For *TinyTrain*, we also include the time and energy to conduct a dynamic layer/ channel selection based on our proposed importance metric, by computing the Fisher information on top of those to load a model and fine-tune it. Regarding energy, we measure the total amount of energy consumed by a device during the end-to-end training process. This is performed by measuring the power consumption on Pi Zero using a YOTINO USB power meter and deriving the energy consumption following the equation: Energy = Power × Time.

Further Details on Memory Usage. There are several components that account for the memory footprint for the backward pass of training. Specifically, (F1) model weights and (F2) buffer space containing input and output tensors of a layer comprise the memory usage during the forward pass (*i.e.* inference). On top of that, during the backward pass (*i.e.* training), we also need to consider (B1) the model weights to be updated or accumulated gradients (*i.e.* a buffer space that contains newly updated weights or accumulated gradients from back-propagation), (B2) other optimiser parameters such as momentum values, (B3) values used to compute the derivatives of non-linear functions like ReLU from the last layer L to a layer i up to which we perform back-propagation, and (B4) inputs x_i of the layers selected to be updated from the last layer L to a layer i up to which we back-propagate.

Regarding (B3), ReLU-type activation functions only need to store a binary mask indicating whether the value is smaller than zero or not. Hence, the memory cost of each non-linearity activation function based on ReLU is $|x_i|$ bits (32× smaller than storing the whole x_i), which is negligible. In our work, the employed network architectures (*e.g.* MCUNet, MobileNetV2, and ProxylessNASNet) rely on the ReLU non-linearity function. Regarding (B4), it is worth mentioning that when computing the gradient $g(W_i)$ given the inputs (x_i) and the gradients ($g(x_{i+1})$) to a (i-th) layer, we perform $g(W_i) = g(x_{i+1}).T * x_i$ to get gradient w.r.t the weights and $g(x_i) = g(x_{i+1}) * W_i$. Note that the intermediate inputs (x_i) are only required to get the gradient of the weights ($g(W_i)$), meaning that the backward memory can be substantially reduced *if we do not update* the model weights (W_i). This property is applicable to linear layers, convolutional layers, and normalisation layers as studied by Cai et al. (2020).

In our evaluation (§3.2), we conducted memory analysis to present the memory usage by taking into account both inference and backward-pass memory. We adopt the memory cost profiler used in prior work (Cai et al., 2020), which reuses the inference memory space during the backward pass wherever possible. Specifically, the memory space of (F2) can be overlapped with (B3) and (B4) as the buffer space for input and output tensors can be reused for intermediate variables of (B3) and (B4). On the other hand, the memory space for (B1) and (B2) cannot be overlapped with (F2) when the gradient accumulation is used as the system needs to retain the updated model weights and optimiser parameters throughout the training process. In addition, we would like to add that, depending on the hardware and deployment libraries, the model weights (F1) reside in the storage instead of being loaded on the main memory space. For example, on MCUs, model weights are stored on Flash (storage) and do not consume space on SRAM. Thus, we only include the model weights to be updated when calculating the memory usage for the backward pass.

A.5 Baselines

We include the following baselines in our experiments to evaluate the effectiveness of ***TinyTrain***.

None. This baseline does not perform any on-device training during deployment. Hence, it shows the accuracy drops of the DNNs when the model encounters a new task of a cross-domain dataset.

FullTrain. This method trains the entire backbone, serving as the strong baseline in terms of accuracy performance, as it utilises all the required resources without system constraints. However, this method intrinsically consumes the largest amount of system resources in terms of memory and computation among all baselines.

LastLayer. This refers to adapting only the head (*i.e.* the last layer or classifier), which requires relatively small memory footprint and computation. However, its accuracy typically is too low to be practical. Prior works (Ren et al., 2021; Lee and Nirjon, 2020) adopt this method to update the last layer only for on-device training.

TinyTL (Cai et al., 2020). This method proposes to add a small convolutional block, named the lite-residual module, to each convolutional block of the backbone network. During training, TinyTL updates the lite-residual modules while freezing the original backbone, requiring less memory and fewer computations than training the entire backbone. As shown in our results, TinyTrain requires the second largest amount of memory and compute resources among all baselines.

SparseUpdate (Lin et al., 2022). This method reduces the memory footprint and computation in performing on-device training. Memory reduction comes from updating selected layers in the network, followed by another selection of channels within the selected layers. However, SparseUpdate adopts a static channel and layer selection policy that relies on evolutionary search (ES). This ES-based selection scheme requires compute and memory resources that the extreme-edge devices can not afford. Even in the offline compute setting, it takes around 10 mins to complete the search.

B Details of Sampling Algorithm during Meta-Testing

B.1 Sampling Algorithm during Meta-Testing

We now describe the sampling algorithm during meta-testing that produces realistically imbalanced episodes of various ways and shots (*i.e.* K-way-N-shot), following Triantafillou et al. (2020). The sampling algorithm is designed to accommodate realistic deployment scenarios by supporting the various-way-various-shot setting. Given a data split (*e.g.* train, validation, or test split) of the dataset, the overall procedure of the sampling algorithm is as follows: (1) sample of a set of classes $\mathcal{C}$ and (2) sample support and query examples from $\mathcal{C}$.

Sampling a set of classes. First of all, we sample a certain number of classes from the given split of a dataset. The 'way' is sampled uniformly from the pre-defined range [5, MAX], where MAX indicates either the maximum number of classes or 50. Then, 'way' many classes are sampled uniformly at random from the given split of the dataset. For datasets with a known class organisation, such as ImageNet and Omniglot, the class sampling algorithm differs as described in (Triantafillou et al., 2020).

Sampling support and query examples. Having selected a set of classes, we sample support and query examples by following the principle that aims to simulate realistic scenarios with limited (*i.e.* few-shot) and imbalanced (*i.e.* realistic) support set sizes as well as to achieve a fair evaluation of our system via query set.

- **Support Set Size.** Based on the selected set of classes from the first step (*i.e.* sampling a set of classes), the support set is at most 100 (excluding the query set described below). The support set size is at least one so that every class has at least one image. The sum of support set sizes across all the classes is capped at 500 examples as we want to consider few-shot learning (FSL) in the problem formulation.
- **Shot of each class.** After having determined the support set size, we now obtain the 'shot' of each class.
- **Query Set Size.** We sample a class-balanced query set as we aim to perform well on all classes of samples. The number of minimum query sets is capped at 10 images per class.

B.2 Sample Statistics during Meta-Testing

In this subsection, we present summary statistics regarding the support and query sets based on the sampling algorithm described above in our experiments. In our evaluation, we conducted 200 trials of experiments (200 sets of support and query samples) for each target dataset. Table 4 shows the average (Avg.) number of ways, samples, and shots of each dataset as well as their standard deviations (SD), demonstrating that the sampled target data are designed to be the challenging and realistic various-way-various-shot CDFSL problem. Also, as our system performs well on such challenging problems, we demonstrate the effectiveness of our system.

C Fine-tuning Procedure during Meta-Testing

As we tackle realistic and challenging scenarios of the cross-domain few-shot learning (CDFSL) problem, the pre-trained DNNs can encounter a target dataset drawn from an unseen domain, where the pre-trained DNNs could fail to generalise due to a considerable shift in the data distribution.

Hence, to adjust to the target data distribution, we perform fine-tuning (on-device training) on the pre-trained DNNs by a few gradient steps while leveraging the data augmentation (as explained below). Specifically, the feature backbone as the DNNs is fine-tuned as our employed models are based on ProtoNet.

Our fine-tuning procedure during the meta-testing phase is similar to that of (Guo et al., 2020; Hu et al., 2022). First of all, as the support set is the only labelled data during meta-testing, prior work (Guo et al., 2020) fine-tunes the models using only the support set. For (Hu et al., 2022), it first uses data augmentation with the given support set to create a pseudo query set. After that, it uses the support set to generate prototypes and the pseudo query set to perform backpropagation using Eq. 1. Differently from (Guo et al., 2020),the fine-tuning procedure of (Hu et al., 2022) does not need to compute prototypes and gradients using the same support set using Eq. 1. However, Hu

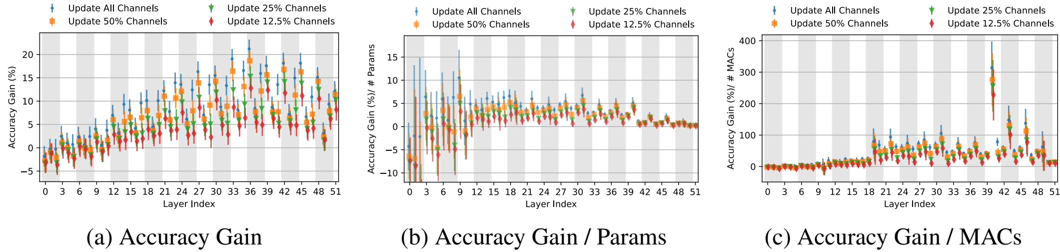
Table 4: The summary statistics of the support and query sets sampled from nine cross-domain datasets.

	Traffic	Omniglot	Aircraft	Flower	CUB	DTD	QDraw	Fungi	COCO
Avg. Num of Ways	22.5	19.3	9.96	9.5	15.6	6.2	27.3	27.2	21.8
Avg. Num of Samples (Support Set)	445.9	93.7	369.4	287.8	296.3	324.0	460.0	354.7	424.1
Avg. Num of Samples (Query Set)	224.8	193.4	99.6	95.0	156.4	61.8	273.0	105.5	217.8
Avg. Num of Shots (Support Set)	29.0	4.6	38.8	30.7	20.7	53.3	23.6	15.6	27.9
Avg. Num of Shots (Query Set)	10	10	10	10	10	10	10	10	10
SD of Num of Ways	11.8	10.8	3.4	3.1	6.6	0.8	13.2	14.4	11.5
SD of Num of Samples (Support Set)	90.6	81.2	135.9	159.3	152.4	148.7	94.8	158.7	104.9
SD of Num of Samples (Query Set)	117.7	108.1	34.4	30.7	65.9	8.2	132.4	51.8	114.8
SD of Num of Shots (Support Set)	21.9	2.4	14.9	14.9	10.5	24.5	17.0	8.9	20.7
SD of Num of Shots (Query Set)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Num of Trials	200	200	200	200	200	200	200	200	200

et al. (2022) simply fine-tune the entire DNNs without memory-and compute-efficient on-device training techniques, which becomes one of our baselines, FullTrain requiring prohibitively large memory footprint and computation costs to be done on-device during deployment. In our work, for all the on-device training methods including *TinyTrain*, we adopt the fine-tuning procedure introduced in (Hu et al., 2022). However, we extend the vanilla fine-tuning procedure with existing on-device training methods (*i.e.* LastLayer, TinyTL, SparseUpdate, which serve as the baselines of on-device training in our work) so as to improve the efficiency of on-device training on the extremely resource-constrained devices. Furthermore, our system, *TinyTrain*, not only extends the fine-tuning procedure with memory-and compute-efficient on-device training but also proposes to leverage data-efficient FSL pretraining to enable the first data-, memory-, and compute-efficient on-device training framework on edge devices.

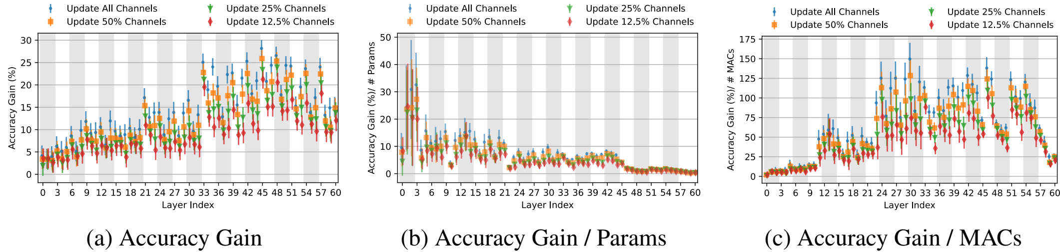
D System Implementation

The *offline* component of our system is built on top of PyTorch (version 1.10) and runs on a Linux server equipped with an Intel Xeon Gold 5218 CPU and NVIDIA Quadro RTX 8000 GPU. This component is used to obtain the pre-trained model weights, *i.e.* pre-training and meta-training. Then, the *online* component of our system is implemented and evaluated on Raspberry Pi Zero 2 and NVIDIA Jetson Nano, which constitute widely used and representative embedded platforms. Pi Zero 2 is equipped with a quad-core 64-bit ARM Cortex-A53 and limited 512 MB RAM. Jetson Nano has a quad-core ARM Cortex-A57 processor with 4 GB of RAM. Also, we do not use sophisticated memory optimisation methods or compiler directives between the inference layer and the hardware to decrease the peak memory footprint; such mechanisms are orthogonal to our algorithmic innovation and may provide further memory reduction on top of our task-adaptive sparse update.



(a) Accuracy Gain (b) Accuracy Gain / Params (c) Accuracy Gain / MACs

Figure 7: Memory- and compute-aware analysis of **MobileNetV2** by updating four different channel ratios on each layer.



(a) Accuracy Gain (b) Accuracy Gain / Params (c) Accuracy Gain / MACs

Figure 8: Memory- and compute-aware analysis of **ProxylessNASNet** by updating four different channel ratios on each layer.

E Additional Results

In this section, we present additional results that are not included in the main content of the paper due to the page limit.

E.1 Memory- and Compute-aware Analysis

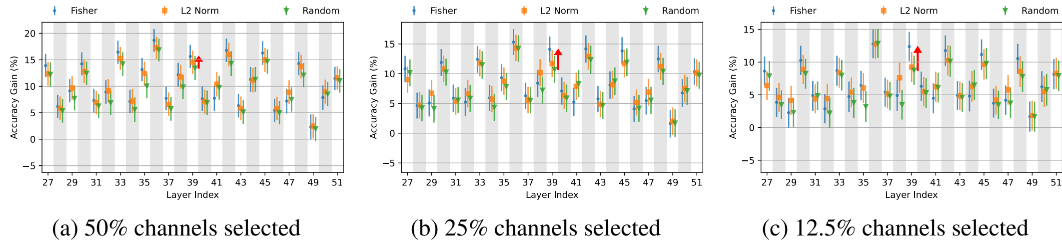
In §2.2, to investigate the trade-offs among accuracy gain, compute and memory cost, we analysed each layer's contribution (*i.e.* accuracy gain) on the target dataset by updating a single layer at a time, together with cost-normalised metrics, including accuracy gain *per parameter* and *per MAC operation* of each layer. MCUNet is used as a case study. Hence, here we provide the results of memory- and compute-aware analysis on the remaining architectures (MobileNetV2 and ProxylessNASNet) based on the Traffic Sign dataset as shown in Figure 7 and 8.

The observations on MobileNetV2 and ProxylessNASNet are similar to those of MCUNet. Specifically: (a) accuracy gain per layer is generally highest on the first layer of each block for both MobileNetV2 and ProxylessNASNet; (b) accuracy gain per parameter of each layer is higher on the second layer of each block for both MobileNetV3 and ProxylessNASNet, but it is not a clear pattern; and (c) accuracy gain per MACs of each layer has peaked on the second layer of each block for MobileNetV2, whereas it does not have clear patterns for ProxylessNASNet. These observations indicate a non-trivial trade-off between accuracy, memory, and computation for all the employed architectures in our work.

E.2 Pairwise Comparison among Different Channel Selection Schemes

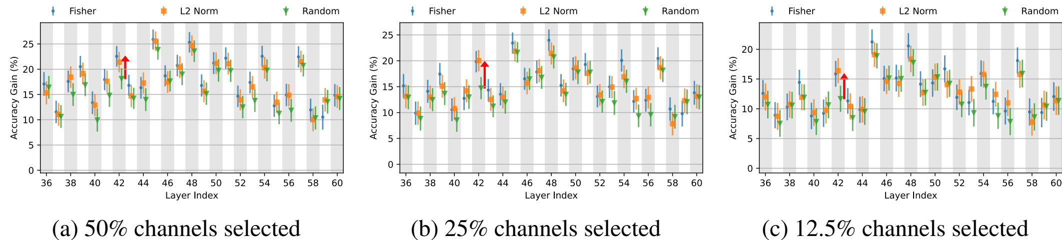
Here, we present additional results regarding the pairwise comparison between our dynamic channel selection and static channel selections (*i.e.* Random and L2-Norm). Figure 9 and 10 show that the results of MobileNetV2 and ProxylessNASNet on the Traffic Sign dataset, respectively.

Similar to the results of MCUNet, the dynamic channel selection on MobileNetV2 and ProxylessNASNet consistently outperforms static channel selections as the accuracy gain per layer differs by



(a) 50% channels selected (b) 25% channels selected (c) 12.5% channels selected

Figure 9: The pairwise comparison between our dynamic channel selection and static channel selections (*i.e.* Random and L2-Norm) on **MobileNetV2**.



(a) 50% channels selected (b) 25% channels selected (c) 12.5% channels selected

Figure 10: The pairwise comparison between our dynamic channel selection and static channel selections (*i.e.* Random and L2-Norm) on **ProxylessNASNet**.

up to 5.1%. Also, the gap between dynamic and static channel selection increases as fewer channels are selected for updates.

E.3 End-to-End Latency Breakdown of *TinyTrain* and *SparseUpdate*

In this subsection, we present the end-to-end latency breakdown to highlight the efficiency of our task-adaptive sparse update (*i.e.* the dynamic layer/channel selection process during deployment) by comparing our work (*TinyTrain*) with previous SOTA (*SparseUpdate*). We present the time to identify important layers/channels by calculating Fisher Potential (*i.e.* Fisher Calculation in Table 5 and 6) and the time to perform on-device training by loading a pre-trained model and performing backpropagation (*i.e.* Run Time in Tables 5 and 6).

In addition to the main results of on-device measurement on Pi Zero 2 presented in §3.2, we selected Jetson Nano as an additional device and performed experiments in order to ensure that our results regarding system efficiency are robust and generalisable across diverse and realistic devices. We used the same experimental setup (as detailed in §3.1 and §A.4) as the one used for Pi Zero 2.

As shown in Table 5 and 6, our experiments show that *TinyTrain* enables efficient on-device training, outperforming *SparseUpdate* by 1.3-1.7× on Jetson Nano and by 1.08-1.12× on Pi Zero 2 with respect to end-to-end latency. Moreover, Our dynamic layer/channel selection process takes around 18.7-35.0 seconds on our employed edge devices (*i.e.* Jetson Nano and Pi Zero 2), accounting for only 3.4-3.8% of the total training time of *TinyTrain*.

E.4 Impact of Meta-Training

In this subsection, we present the complete results of the impact of meta-training. As discussed in §3.3, Figure 6a shows the average Top-1 accuracy with and without meta-training using MCUNet over nine cross-domain datasets. This analysis shows the impact of meta-training compared to conventional transfer learning, demonstrating the effectiveness of our FSL-based pre-training. However, it does not reveal the accuracy results of individual datasets and models. Hence, in this subsection, we present figures that compare Top-1 accuracy with and without meta-training for each architecture and dataset with all the on-device training methods to present the complete results of the impact of meta-training.

Table 5: The end-to-end latency breakdown of *TinyTrain* and SOTA on **Pi Zero 2**. The end-to-end latency includes time (1) to load a pre-trained model, (2) to perform training using given samples (*e.g.* 25) over 40 iterations, and (3) to calculate fisher information on activation (For *TinyTrain*).

Model	Method	Fisher Calculation (s)	Run Time (s)	Total (s)	Ratio
MCUNet	SparseUpdate	0.0	607	607	1.12×
	TinyTrain (Ours)	18.7	526	**544**	1×
MobileNetV2	SparseUpdate	0.0	611	611	1.10×
	TinyTrain (Ours)	20.1	536	**556**	1×
ProxylessNASNet	SparseUpdate	0.0	645	645	1.08×
	TinyTrain (Ours)	22.6	575	**598**	1×

Table 6: The end-to-end latency breakdown of *TinyTrain* and SOTA on **Jetson Nano**. The end-to-end latency includes time (1) to load a pre-trained model, (2) to perform training using given samples (e.g., 25) over 40 iterations, and (3) to calculate fisher information on activation (For *TinyTrain*).

Model	Method	Fisher Calculation (s)	Run Time (s)	Total (s)	Ratio
MCUNet	SparseUpdate	0.0	1,189	1,189	1.3×
	TinyTrain (Ours)	35.0	892	**927**	1×
MobileNetV2	SparseUpdate	0.0	1,282	1,282	1.5×
	TinyTrain (Ours)	32.2	815	**847**	1×
ProxylessNASNet	SparseUpdate	0.0	1,517	1,517	1.7×
	TinyTrain (Ours)	26.8	869	**896**	1×

Figures 11, 12, and 13 demonstrate the effect of meta-training based on MCUNet, MobileNetV2, and ProxylessNASNet, respectively, across all the on-device training methods and nine cross-domain datasets.

E.5 Robustness of Dynamic Channel Selection

As described in §3.3, to show how much improvement is derived from dynamically selecting important channels based on our method at deployment time, Figure 6b compares the accuracy of *TinyTrain* with and without dynamic channel selection, with the same set of layers to be updated within strict memory constraints using MCUNet. In this subsection, we present the full results regarding the robustness of our dynamic channel selection scheme using all the employed architectures and cross-domain datasets. Figures 14, 15, and 16 demonstrate the robustness of dynamic channel selection using MCUNet, MobileNetV2, and ProxylessNASNet, respectively, based on nine cross-domain datasets. Note that the reported results are averaged over 200 trials, and 95% confidence intervals are depicted.

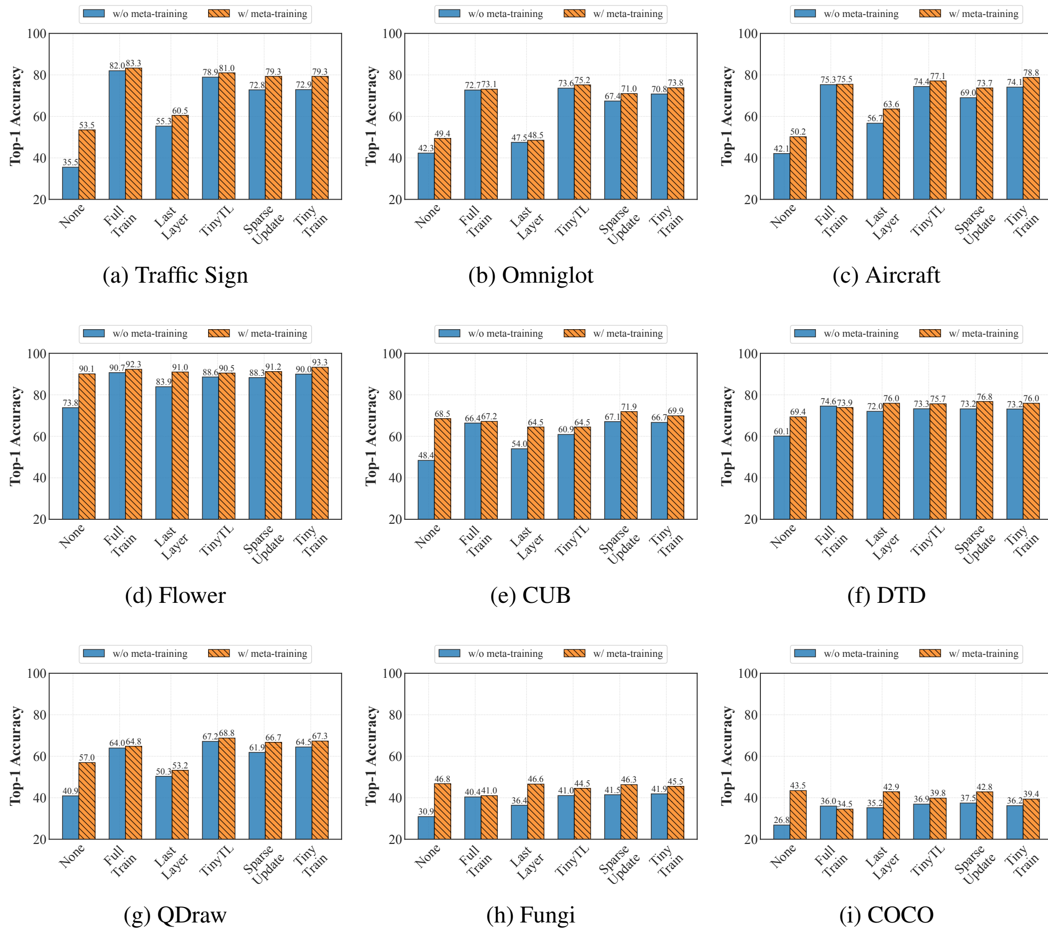


Figure 11: The effect of meta-training on **MCUNet** across all the on-device training methods and nine cross-domain datasets.

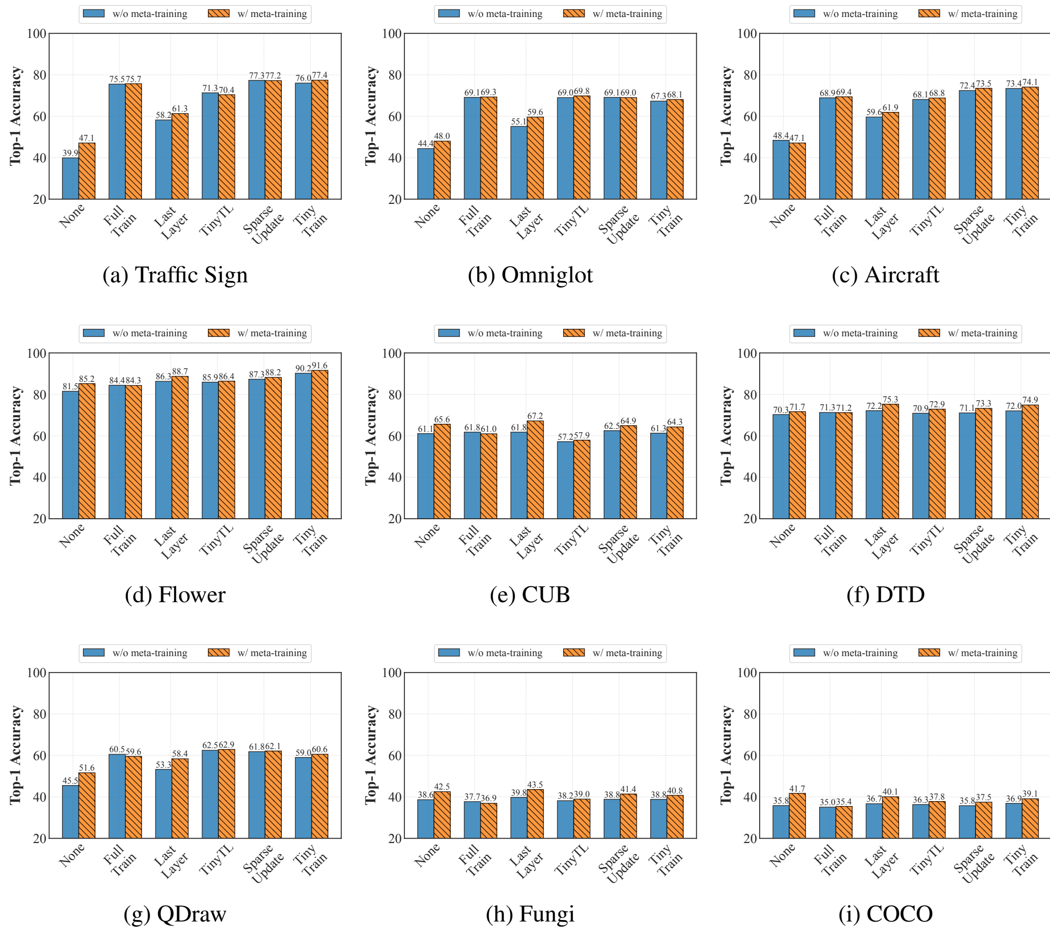
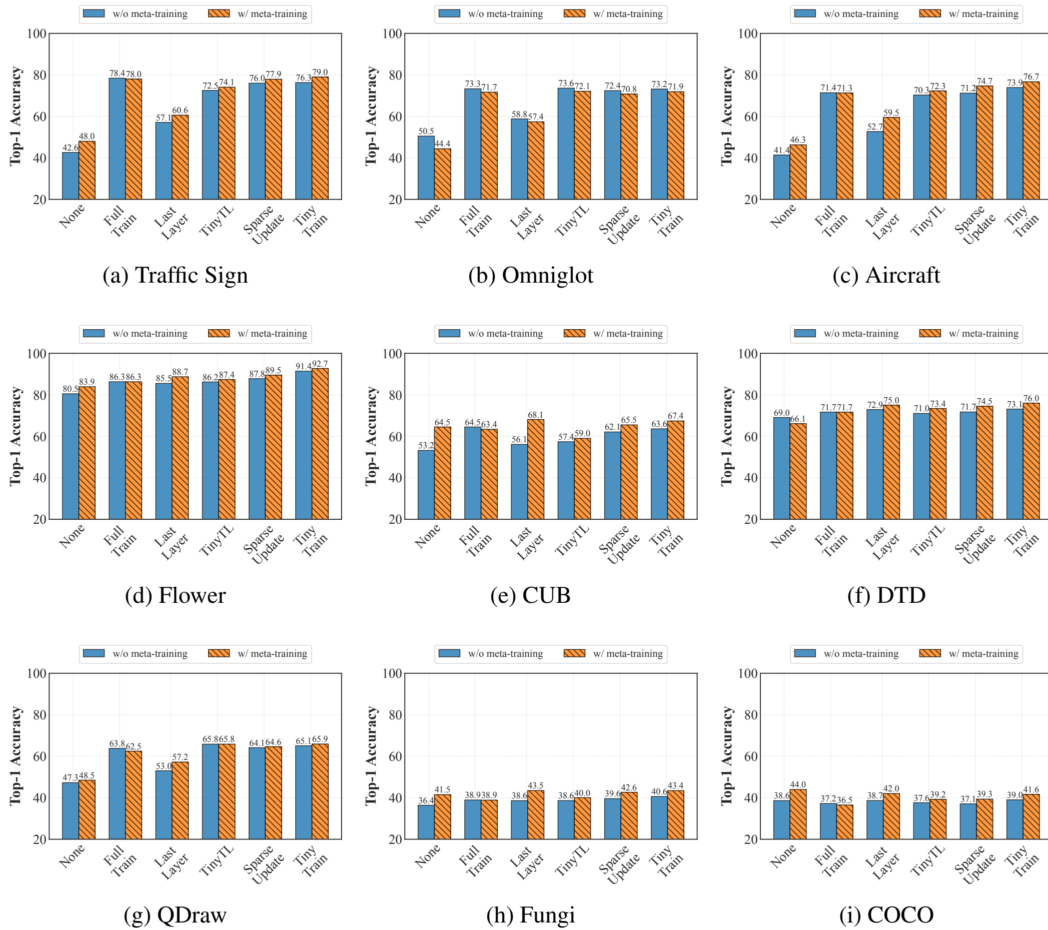


Figure 12: The effect of meta-training on **MobileNetV2** across all the on-device training methods and nine cross-domain datasets.



(a) Traffic Sign (b) Omniglot (c) Aircraft

(d) Flower (e) CUB (f) DTD

(g) QDraw (h) Fungi (i) COCO

Figure 13: The effect of meta-training on **ProxylessNASNet** across all the on-device training methods and nine cross-domain datasets.

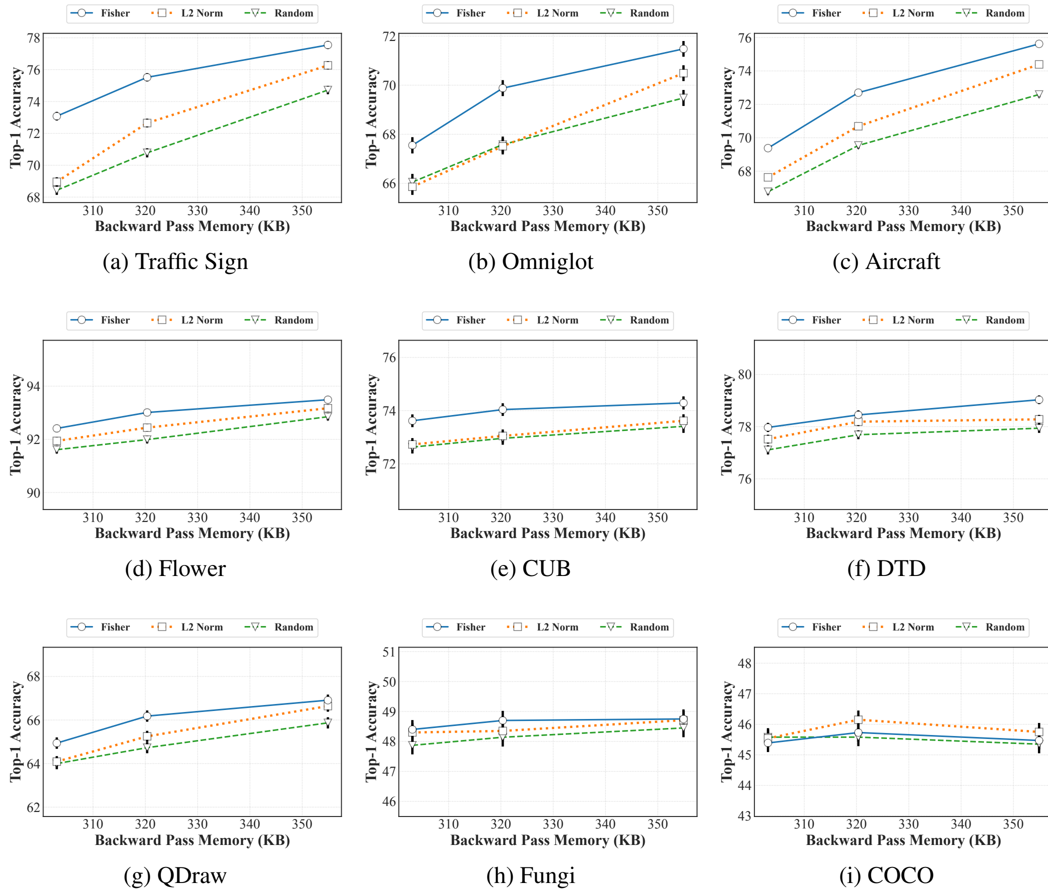


Figure 14: The effect of dynamic channel selection using **MCUNet** on nine cross-domain datasets.

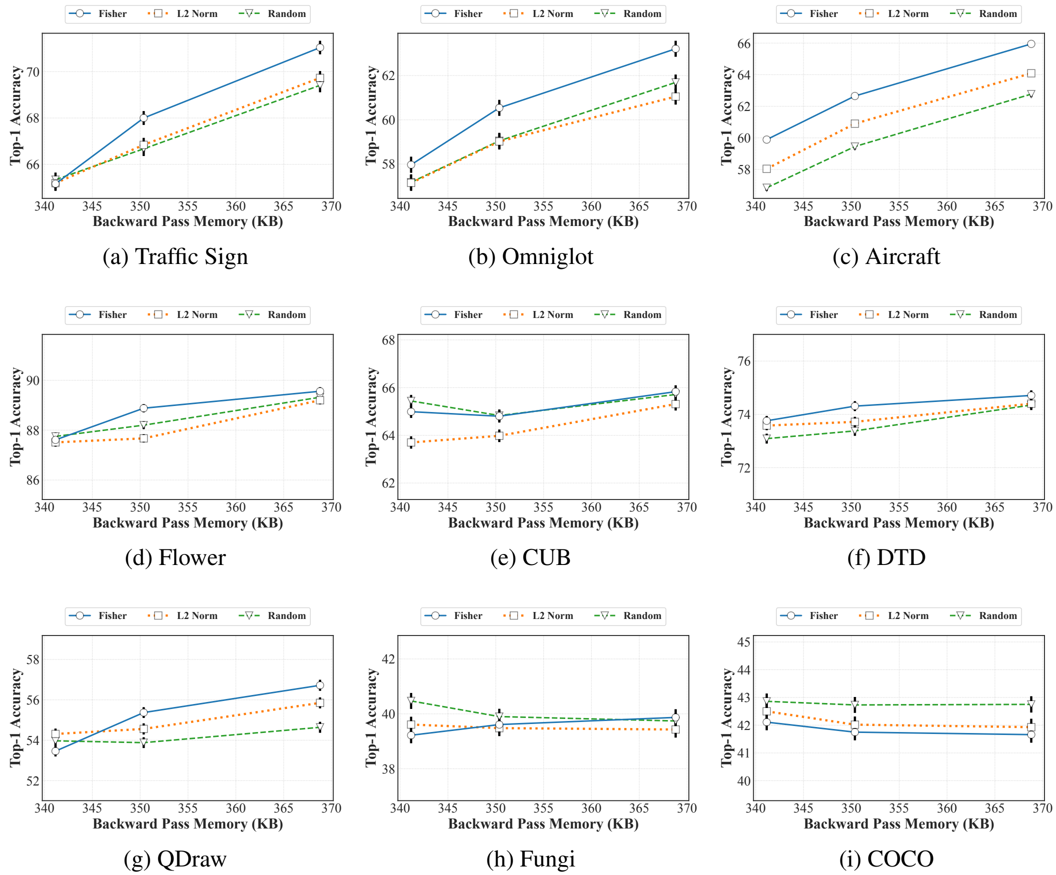
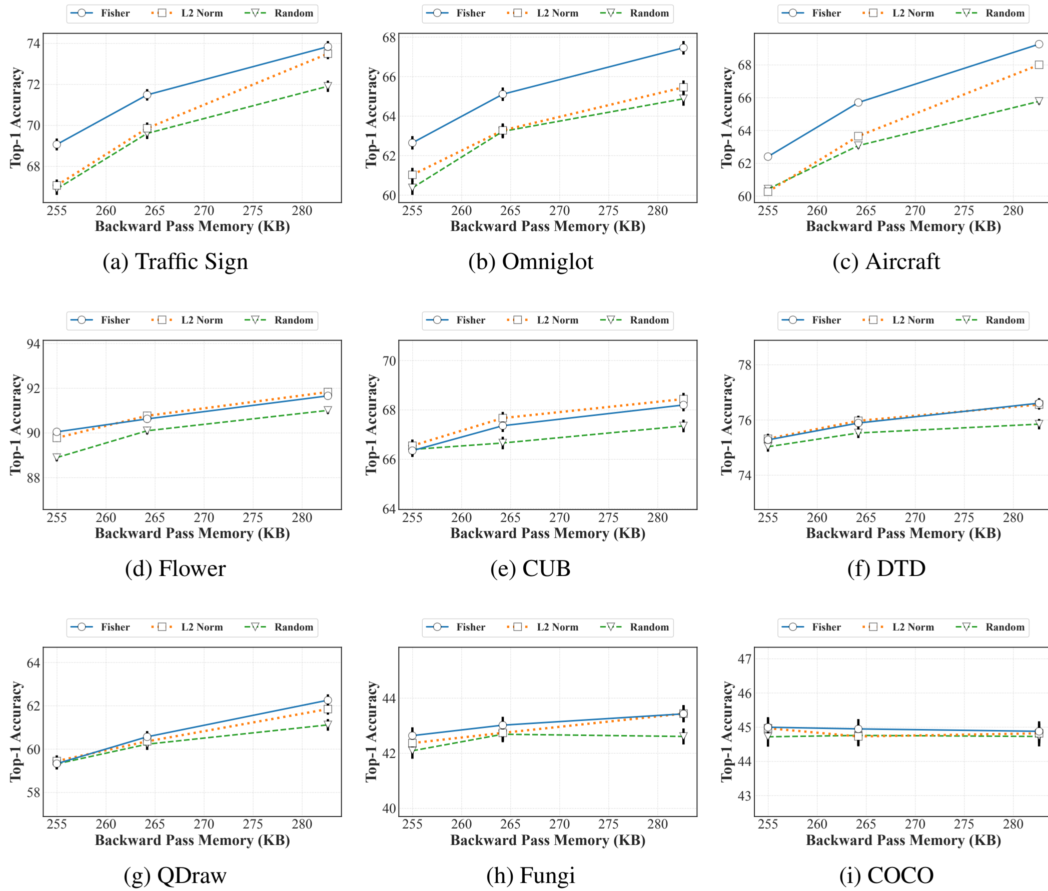


Figure 15: The effect of dynamic channel selection with **MobileNetV2** on nine cross-domain datasets.



(a) Traffic Sign (b) Omniglot (c) Aircraft
(d) Flower (e) CUB (f) DTD
(g) QDraw (h) Fungi (i) COCO

Figure 16: The effect of dynamic channel selection with **ProxylessNASNet** on all the datasets.

F Further Analysis and Discussion

F.1 Further Analysis of Calculating Fisher Information on Activations

In this section, we describe how *TinyTrain* calculates the Fisher information on activations (the primary variable for our proposed multi-objective criterion) without incurring excessive memory overheads. Specifically, computing Fisher information on activations is designed to be within the memory and computation budget for a backward pass determined by hardware and users (e.g., in our evaluation, we use roughly 1 MB as a memory budget). Also, as described in Appendix A.4, the memory space used for saving intermediate variables can be overlapped with that of input/output tensors. As observed in prior works (Lin et al., 2022; 2021), the size of the activation is large for the first few layers and small for the remaining ones. Table 7 shows the saved activations' size to compute the backward pass up to the last k blocks/layers. The sizes of saved activations are well within the peak memory footprint of input/output tensors (*i.e.* 640 KB for MCUNet, 896 KB for MobileNetV2, and 512 KB for ProxylessNASNet). Thus, the memory space of input/output tensors can be reused to store the intermediate variables required to calculate Fisher information on activations.

Also, we empirically demonstrated that important layers for a CDFSL task are located among those last few layers (as shown in Figure 3 for MCUNet, Figure 7 for MobileNetV2, and Figure 8 for ProxylessNASNet). A prior work (Lin et al., 2022) also observed the same trend. In our experiments, TinyTrain demonstrates that inspecting 30-44% of layers is enough to achieve SOTA accuracy, as shown in Table 1 in Section 3.2. Also, note that this process on edge devices is very swift as analysed in Section 3.3.

In addition, it is possible to further reduce the memory usage by optimising the execution scheduling during the forward pass (*e.g.* patch-based inference (Lin et al., 2021) or partial execution (Liberis and Lane, 2023)). This process trade-offs more computation for lower memory usage, consuming more time. However, this can reduce the peak memory to meet the constraints of the target platform. We leave this optimisation as future work.

Table 7: The total size of the saved activations in KB to compute the backward pass up to the last k blocks/layers across three architectures used in our work.

Last k Blocks	Last k Layers	MCUNet	MobileNetV2	ProxylessNASNet
6	18	479.0	432.9	299.3
5	15	392.3	325.7	241.5
4	12	281.0	218.4	171.6
3	9	191.6	148.5	118.0
2	6	135.9	101.6	89.1
1	3	80.3	54.7	60.3

F.2 Cost Analysis of Meta-Training

In this subsection, we analyse the cost of meta-training, one of the major components of our FSL-based pre-training, in terms of the overall latency to perform meta-training. *TinyTrain*'s meta-training stage takes place offline (as illustrated in Figure 2) on a server equipped with sufficient computing power and memory (refer to §D for more details regarding hardware specifications used in our work) prior to deployment on-device. In our experiments, the offline meta-training on MiniImageNet takes around 5-6 hours across three architectures. However, note that this cost is small as meta-training needs to be performed *only once* per architecture. Furthermore, this cost is amortised by being able to reuse the same resulting meta-trained model across multiple downstream tasks (different target datasets) and devices, *e.g.* Raspberry Pi Zero 2 and Jetson Nano, while achieving significant accuracy improvements (refer to Table 1 and Figure 6a in the main manuscript and Figures 11, 12, and 13 in the appendix).

G EXTENDED RELATED WORK

G.1 ON-DEVICE TRAINING

Scarce memory and compute resources are major bottlenecks in deploying DNNs on tiny edge devices. In this context, researchers have largely focused on optimising *the inference stage* (*i.e.* forward pass) by proposing lightweight DNN architectures (Gholami et al., 2018; Sandler et al., 2018; Ma et al., 2018), pruning (Han et al., 2016; Liu et al., 2020), and quantisation methods (Jacob et al., 2018; Krishnamoorthi, 2018; Rastegari et al., 2016), leveraging the inherent redundancy in weights and activations of DNNs. Also, researchers investigated on how to efficiently leverage heterogeneous processors (Jeong et al., 2022; Ling et al., 2021b;a), and offload computation (Yao et al., 2020). Driven by the increasing privacy concerns and the need for post-deployment adaptability to new tasks or users, the research community has recently turned its attention to enabling DNN *training* (*i.e.,* backpropagation having both forward and backward passes, and weights update) at the edge.

Researchers proposed memory-saving techniques to resolve the memory constraints of training (Sohoni et al., 2019; Chen et al., 2021; Pan et al., 2021; Evans and Aamodt, 2021; Liu et al., 2022). For example, gradient checkpointing (Chen et al., 2016; Jain et al., 2020; Kirisame et al., 2021) discards activations of some layers in the forward pass and recomputes those activations in the backward pass. Microbatching (Huang et al., 2019) splits a minibatch into smaller subsets that are processed iteratively, to reduce the peak memory needs. Swapping (Huang et al., 2020; Wang et al., 2018; Wolf et al., 2020) offloads activations or weights to an external memory/storage (*e.g.* from GPU to CPU or from an MCU to an SD card). Some works (Patil et al., 2022; Wang et al., 2022; Gim and Ko, 2022) proposed a hybrid approach by combining two or three memory-saving techniques. Although these methods reduce the memory footprint, they incur additional computation overhead on top of the already prohibitively expensive on-device training time at the edge. Instead, *TinyTrain* drastically minimises not only memory but also the amount of computation through its dynamic sparse update that identifies and trains only the most important layers/channels on-the-fly.

A few existing works (Lin et al., 2022; Cai et al., 2020; Profentzas et al., 2022; Qu et al., 2022) have also attempted to optimise both memory and computations, with prominent examples being *TinyTL* (Cai et al., 2020), *p-Meta* (Qu et al., 2022), and *SparseUpdate* (Lin et al., 2022). By selectively updating only a subset of layers and channels during on-device training, these methods effectively reduce both the memory and computation load. Nonetheless, as shown in §3.2, the performance of this approach drops dramatically (up to 7.7% for *SparseUpdate*) when applied at the extreme edge where data availability is low. This occurs because the approach requires access to the entire target dataset (*e.g. SparseUpdate* (Lin et al., 2022) uses the entire CIFAR-100 dataset (Krizhevsky et al., 2009)), which is unrealistic for such devices in the wild. More importantly, it requires a large number of epochs (*e.g. SparseUpdate* requires 50 epochs) to reach high accuracy, which results in an excessive training time of up to 10 days when deployed on extreme edge devices, such as STM32F746 MCUs. Also, these methods require running *a few thousands of* computationally heavy search (Lin et al., 2022) or pruning (Profentzas et al., 2022) processes on powerful GPUs to identify important layers/channels for each target dataset; as such, the current *static* layer/channel selection scheme cannot be adapted on-device to match the properties of the user data and hence remains fixed after deployment, leading to an accuracy drop. *p-Meta* enables pre-selected layer-wise updates learned during offline meta-training and dynamic channel-wise updates during online on-device training. However, as *p-Meta* requires additional learned parameters such as a meta-attention module identifying important channels for every layer, its computation and memory saving are relatively low. For example, *p-Meta* still incurs up to 4.7× higher memory usage than updating the last layer only. In addition, *TinyTL* still demands excessive memory and computation (see §3.2). In contrast, *TinyTrain* enables data-, compute-, and memory-efficient training on tiny edge devices by adopting few-shot learning pre-training and dynamic layer/channel selection.

G.2 FEW-SHOT LEARNING

Due to the scarcity of labelled user data on the device, developing Few-Shot Learning (FSL) techniques is a natural fit for on-device training (Hospedales et al., 2022). FSL methods aim to learn a target task given a few examples (*e.g.* 5-30 samples per class) by transferring the knowledge from large source data (*i.e.* meta-training) to scarcely annotated target data (*i.e.* meta-testing). Until now, several FSL schemes have been proposed, ranging from gradient-based (Finn et al., 2017; Antoniou

et al., 2018; Li et al., 2017), and metric-based (Snell et al., 2017; Sung et al., 2018; Satorras and Estrach, 2018) to Bayesian-based (Zhang et al., 2021). Recently, a growing body of work has been focusing on cross-domain (out-of-domain) FSL (CDFSL) (Guo et al., 2020). The CDFSL setting dictates that the source (meta-train) dataset drastically differs from the target (meta-test) dataset. As such, although CDFSL is more challenging than the standard in-domain (*i.e.* within-domain) FSL (Hu et al., 2022), it tackles more realistic scenarios, which are similar to the real-world deployment scenarios targeted by our work. In our work, we focus on realistic use-cases where the available source data (*e.g.* MiniImageNet (Vinyals et al., 2016)) are significantly different from target data (*e.g.* meta-dataset (Triantafillou et al., 2020)) with a few samples (5-30 samples per class), and hence incorporate CDFSL techniques into *TinyTrain*.

FSL-based methods only consider data efficiency and neglect the memory and computation bottlenecks of on-device training. Therefore, we explore joint optimisation of three major pillars of on-device training such as data, memory, and computation.

In addition, Un-/Self-Supervised Learning could be a potential solution to data scarcity issues. However, as investigated in (Liu et al., 2021), self-supervised learning in the presence of significant distribution shifts, as in the cross-domain tasks, could result in severe overfitting and insufficiency to capture the complex distribution of high-dimensional features in low-order statistics, leading to deteriorated accuracy. Further investigation could potentially reveal the feasibility of applying these techniques in cross-domain on-device training.