Enabling Privacy-Preserving Model Personalization via On-Device Incremental Training

by

Jiacheng Yang

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science

Graduate Department of Electrical & Computer Engineering
University of Toronto

Enabling Privacy-Preserving Model Personalization
via On-Device Incremental Training

Jiacheng Yang
Master of Applied Science

Graduate Department of Electrical & Computer Engineering
University of Toronto
2022

Abstract

Inference on edge devices (e.g., smartphones) is becoming increasingly common, benefiting from low latency inference and user privacy by keeping data on-device. However, on-device *training* is still considered impractical due to mobile computational and memory capacity constraints. Nevertheless, on-device training is an important direction for enabling privacy-preserving model personalization, whereby a model is customized to a user's preferences and behaviors. We propose MoIL which enables privacy-preserving, efficient, and accurate model personalization by performing compute-intensive training on a curated global dataset in the cloud, and continuing training *incrementally* on-device using a user-specific local dataset. MoIL proposes three key optimizations — global dataset mixing, layer freezing, and feature map caching — to ensure high accuracy predictions and to dramatically reduce on-device training time. MoIL reduces the training time of an image classification model from 3.1 years to 48 minutes and an audio tagging model from 5.2 months to 61 minutes. MoIL achieves local/global on-device accuracy comparable to training on centralized local/global datasets in the cloud, deviating at most 7.3% for global accuracy and 0.9% for local accuracy.

Acknowledgments

First and foremost, I would like to thank my supervisor Gennady Pekhimenko for the opportunity to conduct research at the University of Toronto. During the journey of doing research, I have encountered many challenges. It is through your continuous guidance and feedback that I was able to develop my research skills, which has empowered me to continue my research as a Ph.D. student.

I would like to thank my collaborators James Gleeson and Mostafa Elhoushi who have been there to provide helpful and thoughtful research suggestions to broaden my horizons. I would also like to thank Serina Tan, Shang Wang, and Hongyu Zhu. It is your experience and insight that make my graduate student life enjoyable.

I would like to thank all my lab mates for creating such a great academic atmosphere where we exchange creative ideas freely. Despite everyone having a different cultural background, being able to work and think together has always been my most memorable time.

Last but not least, I would like to sincerely thank my parents. Doing research has never been easy and is often frustrating. It is your constant and considerate support that rescues me from endless self-doubt during my hardest time.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter, we explore the potential benefits and challenges of moving training of deep neural networks (DNN) from the cloud directly onto the user's mobile smartphone device. Besides the obvious privacy benefits of keeping user's data on-device, we show that on-device training allows us to tailor DNNs to the user's behaviors and preferences, achieving better accuracy than just using the pre-trained models downloaded from the cloud. While training from scratch on-device suffers from intolerable training time due to hardware limitations, we propose using incremental training, where we perform pre-training DNNs on expansive user-general datasets in the cloud, and minimal fine-tuning with user-specific data on-device. Unfortunately, incremental training with high accuracy and efficiency is non-trivial to achieve on today's smartphone devices. In particular, a notorious phenomenon called incremental training will naturally occur. To address the challenges of on-device training, we introduce our framework of Mobile Incremental Training (MoIL) in which we propose three key optimizations, global dataset mixing, layer freezing, and feature map caching.

1.1 Overview

The recent success of deep learning (DL) in areas such as computer vision [19, 2], natural language processing [61, 6], and speech recognition [17] has resulted in an increasing interest in deploying DNN inference directly on edge devices such as smartphones and home appliances [7, 36, 34]. On-device inference offers several benefits over performing inference in the cloud. First, since sensitive user data (e.g., images and voices) are never exposed to cloud servers, the user's privacy is thus preserved. Second, on-device inference avoids network requests to the cloud, thereby reducing the inference latency. To achieve effective and efficient on-device inference, smartphones are now equipped with powerful dedicated accelerators (e.g. GPUs and NPUs). These accel-

erators are capable of on-device DNN inference with DNNs designed to run within mobile hardware constraints [22, 42, 30]. Meanwhile, many DL frameworks [4, 51, 38] can further optimize DNNs (e.g. pruning and quantization [16]) for these accelerators to meet the strict hardware constraints of edge devices. The ecosystem of both hardware and software can achieve real-time DNN inference in practice, which eliminates the dependency of cloud servers and network connections for DNN inference and hence ameliorates privacy concerns.

Just as on-device inference enables privacy-preserving low-latency applications, on-device *training* can enable privacy-preserving adaptation of DNNs to specific users through model personalization. Model personalization tailors trained DNNs to better match the user's preferences and behaviors, which is critical for achieving high accuracy on samples that are underrepresented or even not present in the cloud dataset that the DNNs are originally trained on. For example, a user can introduce new food types for nutrition applications [34] or add new words for keyboard prediction applications [18]. One approach to model personalization is to collect and upload the user's data to cloud servers, re-train the DNNs using powerful cloud hardware, and then download the updated DNNs back to the user's device [15]. However, uploading the user's data to the cloud obviously raises privacy concerns and poses security risks. Furthermore, tethering DNN training to cloud servers requires reliable high-bandwidth network connections, which thus makes real-time model updates impossible when such network conditions are poor. If on-device training were available, the user's data would never leave the device, and the resulting privacy, security, and network issues could be averted.

Unfortunately, on-device training is still generally considered intractable due to mobile hardware limitations and a lack of DL framework and mobile System-on-Chip (SoC) vendor library support. For example, as shown in Table 1.1, the theoretical FP32 FLOPS of a commonly used cloud GPU NVIDIA A100 (19.5 TFLOPS [48]) is approximately 27× as much as that of an Apple A13 Bionic chip [35, 9]. On the other hand, the lack of DL framework support further prevents DNN researchers from exploring on-device training application scenarios that would motivate specialized hardware support for backward operators in the first place, resulting in a Chicken-or-the-Egg dilemma. As a result, naïvely training DNNs on edge devices today suffers from intolerably long training times that overwhelm memory, battery, and can even cause thermal throttling issues. Consequently, DNN training is considered prohibitive on-device and is offloaded to cloud servers.

To enable on-device training, we propose **M**obile **o**n-device **I**ncremental **L**earning (MoIL). Since the greatest barrier to on-device training today is the disparity between

	NVIDIA A100 [49]	Apple A13 Bionic [35, 9]
TDP	400W	$\approx$ 6W
Memory	40GB HBM2	4GB LPDDR4X
Memory bandwidth	1,555 GB/s	42.7 GB/s
Number of transistors	5.4×10^{10}	8.5×10^{9}
FP32	19.5 TFLOPs	736 GFLOPs

Table 1.1: Comparison between a typical cloud GPU and a smartphone SoC.

edge and cloud hardware, we adopt an incremental training approach, whereby the model is pre-trained on a large *global dataset* in the cloud, and then incrementally personalized on-device on a user-specific *local dataset*. This approach avoids a large portion of on-device training time while still preserving user privacy. For simplicity, unless otherwise noted, we refer to the pre-trained model as the *global model* (i.e., model trained on the global dataset) and the model after incremental training as the *local model*. We delineate the general process of model personalization in Figure 1.1.

To make on-device incremental training both efficient and capable of producing models with high accuracy comparable to training on centralized global/local datasets in the cloud, MoIL must address two key challenges. First, we must avoid any significant sacrifice of global accuracy in favor of local accuracy during incremental training, since both accuracies are important for realistic application scenarios. Unfortunately, naïvely continuing training on-device using the local dataset leads to catastrophic forgetting [32] whereby the model overfits the local dataset and labels originally recognized in the global dataset are quickly forgotten early in training. We reproduced this phenomenon using ImageNet as a global dataset and a bird species dataset as a local dataset (Section 5.1), where we observed a 69% accuracy drop in global accuracy after 20 epochs of training, from which the global accuracy never recovered. Second, the on-device training process must satisfy strict on-device compute and memory requirements. Prior works that perform model personalization through incremental learning [29, 58, 58] do not fit within mobile hardware constraints. In particular, these methods require full model training that can take days to complete a single epoch on smartphones, and that will exceed on-device memory capacity.

To address the challenges of deploying incremental training to edge devices, MoIL contains three key optimizations that enable on-device training of off-the-shelf pre-trained models (e.g., from Hugging Face [57] or torchvision [14]) by allowing a configurable trade-off between training time, local accuracy, and global accuracy. First, *global dataset mixing* (GDM) mixes a small portion of the global dataset with the device's local dataset to ensure previously learned knowledge is not forgotten as new knowledge is acquired. The size of mixed global data is carefully chosen so as not to

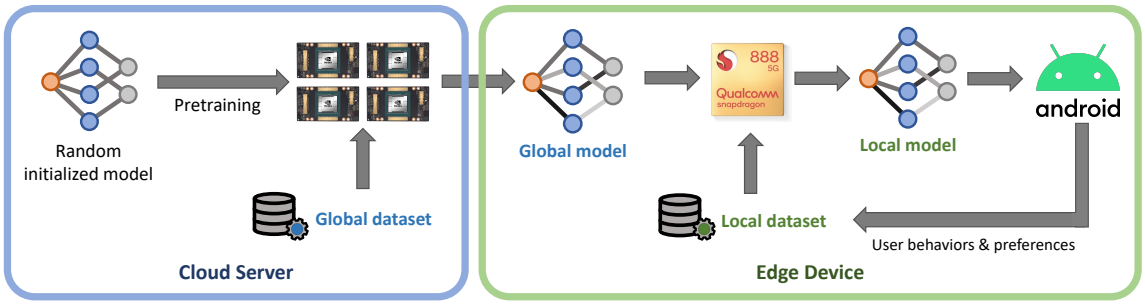


Figure 1.1: Illustration of global/local dataset and global/local model.

overwhelm mobile compute resources while balancing the optimization of both local and global accuracy. Second, *layer freezing* (LF) reduces the compute and memory usage of backpropagation by freezing the weights of the layers at the start of the network thereby limiting gradient flow to only the remaining layers of the model, which serves to naturally preserve features learned during pre-training thus preserving global accuracy. Finally, *feature map caching* (FMC) further reduces the compute time of backpropagation by bypassing the forward pass of the frozen layers. Surprisingly, for models used in image classification tasks, the size of cached feature maps can be even smaller than the original input images; hence, this optimization can help enable efficient storage needed for the GDM optimization.

Our contributions are summarized as follows:

- We demonstrate that existing on-device accelerators can provide at most a 6.2× speedup for on-device training time and that this is insufficient, providing at most a reduction from 3.2 years to 6.4 months. This motivates our approach of incremental learning to reduce on-device compute/memory requirements.
- We propose MoIL that solves two key challenges with on-device incremental learning: (1) catastrophic forgetting, and (2) mobile hardware constraints. Prior works either do not preserve global accuracy [6, 61], do not preserve user privacy [33, 18], do not fit within on-device compute/memory [29, 58, 58], or are limited to inference and cannot be applied directly to training [22, 42, 23, 19, 5].
- We demonstrate how to tune MoIL's hyperparameters to enable training of different DNN models by choosing the correct trade-off between local accuracy, global accuracy, and training time specific to a given application scenario (i.e., DNN model and dataset).
- We show that MoIL makes on-device training on today's mobile smartphones possible by training models well within an 8-hour overnight period during which the phone is typically idle and charging. MoIL reduces training time of an

image classification model, MobileNetV2 [42], from 3.1 years to 48 minutes and an audio tagging model, YAMNet [45], from 5.2 months to 61 minutes. MoIL achieves local/global on-device accuracy comparable to training on centralized local/global datasets in the cloud, deviating at most 7.3% for global accuracy and 0.9% for local accuracy.

The remainder of this thesis is structured as follows:

- In the remainder of Chapter 1, we provide basic background knowledge on stochastic gradient descent with a focus on two mobile application scenarios that form the focus of our study, image recognition and audio tagging.

- Chapter 2 compares and contrasts MoIL with previous works on incremental learning and on-device training. We show that existing prior works do not consider mobile hardware constraints.

- In Chapter 3, we identify key challenges that prevent on-device training from being possible on today's smartphones. In particular, mobile accelerators (e.g. GPUs and NPUs) provide insufficient speed-ups for reducing total training time, and to make matters worse, there is no DL framework support for running backward operators on mobile accelerators, causing huge overhead that overrides the benefit of using these accelerators.

- Chapter 4 introduces MoIL with the three proposed key optimizations. Global dataset mixing alleviates catastrophic forgetting [32] and preserves global accuracy. Layer freezing reduces time spent backpropagation time by freezing part of the DNNs and only training the remaining layers. Feature map caching further reduces backpropagation time by caching feature maps after the frozen layers.

- Chapter 5 evaluates the effectiveness of the proposed framework on two tasks, image classification and audio tagging. We show by experiments that we can achieve training both image classification (48 minutes) and audio tagging (61 minutes) models overnight.

- Chapter 6 concludes this thesis with a discussion of future directions required for making end-to-end deployment of on-device training practical in today's mobile applications through techniques such as automatic local/global dataset reconciliation, and device-assisted labeling of local datasets.

1.2 Background

In this section, we will introduce the basics and the background of this thesis including the formulation and basic algorithms of deep learning and incremental training and commonly used network architectures for various tasks.

1.2.1 Basics of Deep Learning

A deep neural network (DNN), also named as deep model, is a parameterized function $\mathbf{o} = \mathcal{M}(\mathbf{x}; \boldsymbol{\theta})$ where $\boldsymbol{\theta}$ is the model parameters, $\mathbf{x}$ stands for the input to the deep model, and $\mathbf{o}$ denotes the output of the deep model. For different tasks, the input and the output of the deep model may vary. For example, the input can be an RGB pixel image represented as a matrix, the mel-spectrogram of an audio recording, or a sequence of words encoded as a one-hot vector. The output can be a vector of probabilities (how likely the image or the audio is labeled with a class), or a sequence of vectors standing for the likelihood of the next word in machine translation.

Though different from each other, deep models are often organized in a layer-by-layer manner and hence can be formulated as follows (see the forward pass in Figure 1.2),

$$\begin{aligned} \mathbf{F}_1 &= L_1(\mathbf{x}; \boldsymbol{\theta}_1) \\ \mathbf{F}_2 &= L_2(\mathbf{F}_1; \boldsymbol{\theta}_2) \\ &\vdots \\ \mathbf{F}_{n-1} &= L_{n-1}(\mathbf{F}_{n-2}; \boldsymbol{\theta}_{n-1}) \\ \mathcal{M}(\mathbf{x}; \boldsymbol{\theta}) &= L_n(\mathbf{F}_{n-1}; \boldsymbol{\theta}_n) \end{aligned} \tag{1.1}$$

in which $\mathbf{F}_k$ is called the *feature map* of the k-th layer. Note that some convolution networks such as Inception [46, 47] can have branches, but those architectures are still feedforward when viewed in the building block perspective and hence the formulation is still applicable.

To measure how well the deep model performs, we have an objective function $\mathcal{L}(\boldsymbol{\theta}; \mathbf{x}, y)$ to measure the "distance" between the network outputs $\mathcal{M}(\mathbf{x}; \boldsymbol{\theta})$ and the ground truth y and by minimizing this objective function we can find the optimal set of model parameters. For classification tasks, we often use softmax cross-entropy as our objective function. In this case, $\mathcal{L}(\boldsymbol{\theta}; \mathbf{x}, y)$ is defined as,

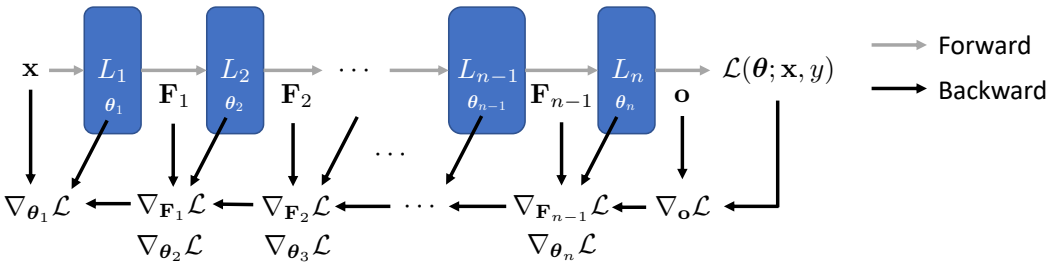


Figure 1.2: Illustration of forward and backward pass of DNN training.

$$\begin{aligned}\mathcal{L}(\boldsymbol{\theta}; \mathbf{x}, y) &= -\mathbf{1}_y \cdot \log\left(\text{Softmax}(\mathbf{o})\right) \\ &= -\mathbf{o}[y] + \log\left(\sum_{k \in \mathcal{C}} e^{\mathbf{o}[k]}\right)\end{aligned} \tag{1.2}$$

where $\mathbf{o}$ represents the outputs of the deep model $\mathcal{M}(\mathbf{x}; \boldsymbol{\theta})$, $\mathcal{C}$ stands for all possible classes, and $\mathbf{1}_c$ denotes the one-hot indicator vector, namely,

$$\mathbf{1}_c[i] = \begin{cases} 1 & i = c \\ 0 & i \neq c \end{cases} \tag{1.3}$$

For tagging problems where a dataset item can have multiple labels, we often use sigmoid cross-entropy loss, namely,

$$\mathcal{L}(\boldsymbol{\theta}; \mathbf{x}, y) = -\left[\sum_{k \in y} \log \sigma(\mathbf{o}[\mathbf{k}]) + \sum_{k \notin y} \log(1 - \sigma(\mathbf{o}[\mathbf{k}]))\right] \tag{1.4}$$

where $\sigma(\cdot)$ denotes the Sigmoid function,

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{1.5}$$

To train the model on a dataset $\mathcal{D}$ which contains pairs of input and output, we use gradient descent (GD) to iteratively optimize the objective,

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \gamma \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}; \mathcal{D}) \tag{1.6}$$

The gradient with respect to the parameters of each layer can be solved by back-propagation (see backward pass in Figure 1.2), namely,

$$\nabla_{\mathbf{F}_k} \mathcal{L} = \mathbf{J}^{\mathsf{T}}_{\mathbf{F}_{k+1}}(\mathbf{F}_k) \nabla_{\mathbf{F}_{k+1}} \mathcal{L} \tag{1.7}$$

$$\nabla_{\boldsymbol{\theta}_k}\mathcal{L} = \mathbf{J}^{\mathsf{T}}_{\mathbf{F}_{k+1}}(\boldsymbol{\theta}_k)\nabla_{\mathbf{F}_{k+1}}\mathcal{L} \tag{1.8}$$

where $\mathbf{J}^{\mathsf{T}}_{\mathbf{f}}(\mathbf{x})$ is transpose of the Jacobian matrix [52] of a vector function $\mathbf{f}$ with respect to the variable $\mathbf{x}$.

Note that directly solving the gradient on the full dataset is slow and often prohibitive due to the large memory footprint of storing intermediate activation maps. In practice, we often use batch gradient descent (BGD) as an alternative instead. The loss function of BGD is defined as follows,

$$\mathcal{L}_{\text{BGD}}(\boldsymbol{\theta}; \mathcal{D}) = \mathbb{E}_{\mathcal{B}\sim\mathcal{U}(\mathcal{D})}\left[\frac{1}{|\mathcal{B}|}\sum_{(\mathbf{x}_i, y_i)\in\mathcal{B}}\mathcal{L}(\boldsymbol{\theta}; \mathbf{x}_i, y_i)\right] \tag{1.9}$$

where $\mathcal{B}$ denotes the batch uniformly sampled from the dataset $\mathcal{D}$.

Recently, researchers have found momentum-based optimizers (e.g. RMSProp [21] and Adam [27]) can speed-up the convergence of training. These optimizers introduce optimizer's state which we represent as $\boldsymbol{\phi}$ and we can generalize each optimization step as,

$$\boldsymbol{\theta}_{t+1}, \boldsymbol{\phi}_{t+1} = \text{OptimizerStep}(\mathbf{g}, \boldsymbol{\theta}_t, \boldsymbol{\phi}_t, \gamma) \tag{1.10}$$

where $\mathbf{g}$ is the gradient of the current batch.

For evaluation or inference of deep models, we can deduce the predictions from the deep model's outputs. For classification tasks, we typically choose the index of the largest outputs as the predicted label. The whole training and evaluation procedure is illustrated as Algorithm 1.

Algorithm 1 Training and evaluation of a deep model

```
function TRAIN(N, θ_Init, φ_Init, D, γ)
    θ ← θ_Init
    φ ← φ_Init
    for i ∈ [0, N) do
        for B ∈ Batchify(RandomShuffled(D)) do
            g ← ∇_θ (1/|B|) Σ_{(x_i,y_i)∈B} L(θ; x_i, y_i)
            θ, φ ← OptimizerStep(g, θ, φ, γ)
        end for
    end for
    return θ
end function
function EVALUATE(θ, x)
    o ← M(x; θ)
    return arg max_{c∈C} o[c]
end function
```

Note that in evaluation we often also have a metric function to measure the per-

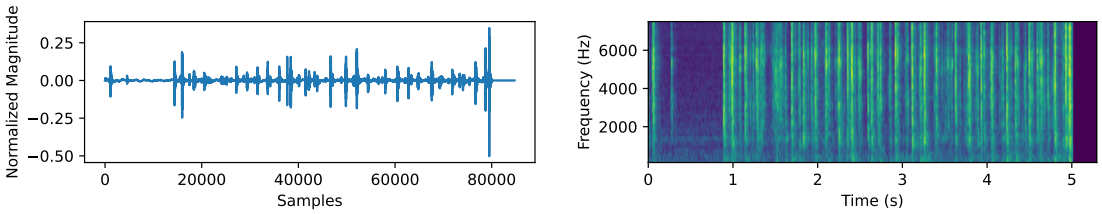
formance of deep models. The main difference between the metric function and the objective function is that the objective function can be used in gradient descent and hence is required to be differentiable. But oftentimes, the objective function is not a direct benchmark tool to show the deep model's performance. On the other hand, the metric function does not need to be differentiable and often it can benchmark the deep model directly. In classification tasks, for example, the metric function we often used is top-k accuracy which is how many correct predictions among the top k-th outputs from the deep model. However, top-k accuracy is not differentiable and hence cannot be used as the objective function. Instead, softmax cross-entropy is often used to optimize the deep model.

1.2.2 Image Classification with Convolutional Neural Networks

Convolution neural network (CNN) is the backbone of DNNs for numerous computer vision tasks. For CNN, autonomous driving uses image recognition to detect obstacles and pedestrians with CNN as its backbone [8]. FaceNet [43], a widely applied face detection method, also relies on CNN and triplet loss to extract feature vectors of given facial images. Those feature vectors can then be used to decide whether two images are in the same cluster.

Among all computer vision tasks, perhaps the most straightforward one to use CNN is the image classification task. Once trained, the deep model can classify given images into predefined classes that are present in the training dataset. The finding of CNN greatly improves the accuracy of image classifier and also many other computer vision tasks compared to manual feature engineering methods such as SIFT [31]. On ImageNet [40], a large labelled dataset for object detection and image classification, convolution-based deep neural networks (e.g. ResNet152) can even outperform the median of human annotators [19].

In this thesis, our settings and requirements of training CNNs on edge devices mimic doing incremental updates in the image recognition system of autonomous cars. The CNN is first pre-trained on the global dataset containing a wide variety of classes of images in an effort to recognize objects as general as possible. Then the model is incrementally trained on the user's local dataset which represents the images in the user's frequently-visited places. After incremental training, the deep model should perform well on the user's local dataset and simultaneously preserve the accuracy on the global dataset.



(a) The waveform of a piece of audio.

(b) The log mel-spectrogram of the same audio.

Figure 1.3: An example of log mel-spectrogram transformation.

1.2.3 Audio Tagging with Convolutional Neural Networks

Audio tagging is a widely used technique in many applications. For example, in a musical application users can ask the application to classify their favorite music into different musical genres. In this scenario, a deep model will be asked to tag a piece of audio with a pre-defined set of labels. The audio can be seen as a sequence of numbers each of which represents a sample on the audio waveform. Multiple preprocessing techniques exist to transform the 1-D audio waveform to a 2-D spectrogram. As an example, the spectrogram in the mel scale, also known as the mel spectrogram, is a compact 2-D representation of the original audio (illustrated in Figure 1.3).

One common approach to feed audio to deep models is to treat the spectrogram as images and then use convolutional networks for tagging. Note that different from classification problem where an item only falls into one class, in audio tagging a piece of audio can be tagged with multiple labels, and the sigmoid cross-entropy is used as the objective function instead for training audio tagging models, namely

$$\mathcal{L}(\boldsymbol{\theta}; \mathbf{x}, y) = -\sum_{k \in \mathcal{C}} [k \in y] \log \sigma(\mathbf{o}_k) + (1 - [k \in y]) \log(1 - \sigma(\mathbf{o}_k)) \tag{1.11}$$

where $\mathbf{o}$ is the outputs of the deep model.

In this thesis, we follow the same routine to preprocess the audio data and feed them to a lightweight convolutional network called YAMNet [45]. We mimic the potential incremental updates to an audio tagging application where the model is first pre-trained with a large global dataset and then incrementally updated with the user's local dataset. After incremental training, the fine-tuned deep model should achieve better accuracy than the pre-trained model on the local dataset while still preserving the accuracy on the global dataset.

Chapter 2

Related Works

In this chapter, we provide a high-level comparison of MoIL against different categories of techniques for enabling on-device training and model personalization; detailed comparisons are provided below.

- *Efficient On-Device Inference* approaches reduce the compute requirements of on-device inference, but still rely on the cloud for training. In contrast, MoIL conducts training on-device without cloud involvement thereby preserving user data and model privacy.
- *Incremental Learning* approaches rely on knowledge distillation from a larger model which is too compute/memory heavy to be applied on-device.In contrast, MoIL tailors the SGD algorithm to run within on-device compute/memory requirements.
- *Transfer Learning* only optimizes local accuracy at the expense of global accuracy, whereas MoIL optimizes both during model personalization.
- *Federated Learning* distributes SGD across devices, but sacrifices privacy by aggregating weights on centralized machines, whereas MoIL preserves user data and model privacy by keeping weights and user data on-device.

2.1 Efficient On-Device Inference

Since on-device training suffers from obvious hardware limitations, training workloads are offloaded to the cloud and only inference is conducted on-device. To improve the efficiency of on-device inference, researchers have invented multiple techniques which can be classified into two categories, efficient DNN architectures and deep compression.

Different from MoIL, prior works mainly focused on improving the efficiency of on-device inference while training workloads are offloaded to the cloud due to large compute requirements. To enable real-time inference under strict hardware limitations, prior works either invent efficient DNNs [22, 42, 23] with fewer compute FLOPS or DNN compression techniques [19, 5] to further speedup DNN inference. However, training with these lightweight DNNs incurs even more memory footprint [13] than the naïve version (e.g. ResNet-34 [19]) and hence still requires the presence of cloud hardware. Moreover, the DNN compression techniques incur non-negligible accuracy loss and training with compressed networks is still an open research question [28]. Besides, they also do not resolve the catastrophic forgetting issue. Hence, these techniques still cannot enable on-device training.

2.2 Incremental Learning

One of the assumptions of deep learning is that the training datasets are prepared beforehand and evenly distributed. However, unfortunately in real-world scenarios, the dataset is collected in an incremental manner and the distribution of the collected dataset can evolve over time. To tackle this challenge, it is natural to also incrementally fine-tuning the existing DNN to the newly collected data. Unfortunately, naïvely fine-tuning the DNN suffers notoriously from the catastrophic forgetting problem [32].

While MoIL solves catastrophic forgetting by global dataset mixing and layer freezing, most incremental learning literature tackle this challenge by knowledge distillation [29, 58, 58]. However, these techniques only focus on preserving the global accuracy and still requires to backpropagate the full model. Hence, these techniques alone do not solve compute and memory restrictions in on-device training. But they are orthogonal and can be integrated into MoIL to better preserve the global accuracy.

2.3 Transfer Learning

Transfer learning also tackles the accuracy and the computational challenges of DNN training by starting from an existing pre-trained DNN. However, different from MoIL, transfer learning targets a local task that is different from the pre-training task and only local accuracy is considered. Transfer learning can be used to improve training accuracy when the data are not sufficient to train a DNN from scratch. For example, BERT [6] is a pre-trained language model which is trained on a large unsupervised corpus[1], and the pre-trained BERT model can be fine-tuned using transfer learning to

[1]BERT was trained on BooksCorpus (800M words) and English Wikipedia (2,500M words) [6].

achieve better accuracy than a model trained from scratch [61]. In computer vision, pre-training on ImageNet is a de facto standard starting point for object detection [12]. However, since transfer learning only targets the local task, the accuracy on the global task still suffers from catastrophic forgetting and hence cannot be used in our on-device training scenario.

2.4 Federated Learning

Federated learning (FL) alleviates privacy issues associated with collecting user data when training deep models from scratch in a centralized cloud server. Instead, FL performs a distributed stochastic gradient descent (SGD) algorithm and only collects the gradients from the edge devices every few steps. However, most FL approaches [33, 18] still require a centralized weight server to aggregate gradients for training. This cloud dependency incurs potential privacy and security concerns [10, 62] whereas in MoIL the local dataset never leaves the user's device and hence privacy is guaranteed.

Chapter 3

Challenges of On-device Training

On-device incremental training suffers from two main challenges: (1) alleviating catastrophic forgetting and preserving the accuracy on the global dataset, and (2) making the whole training process fit within the edge device hardware constraints. Unfortunately, current DL frameworks have poor support for on-device training, which causes a significant overhead when using on-device DL accelerators such as GPUs and NPUs [53] for training. Hence, we conclude that simply using DL accelerators does not resolve the challenges.

3.1 Hardware Limitations of Edge Devices

Comparison with Cloud Hardware

Table 3.1 shows the major differences of hardware specifications between cloud and edge devices. Note that edge devices can range from self-driving cars to IoT devices. Some of them, e.g. self-driving cars, can be equipped with powerful graphics cards. However, in this work, we focus on smartphones since they are the most widely used edge devices today.

There are two major differences in hardware constraints between cloud and edge devices. First, edge devices have significantly lower memory capacity and speed.

Scenarios	Product	Memory	TDP	FP32 (TFLOPs)
Cloud Server	RTX 2080 Ti	11GB GDDR6	250W	13.45
	Tesla V100	16GB HBM2	450W	31.33
	TPU v3	32GB HBM2	≈200W	14
Smartphones	Apple A13	4GB LPDDR4X	6W	0.73
	Kirin 990	8GB LPDDR4X	8W	0.89
IoT Devices	Arduino Uno	2KB SRAM	<1W	

Table 3.1: Comparison between hardware on cloud servers and on edge devices.

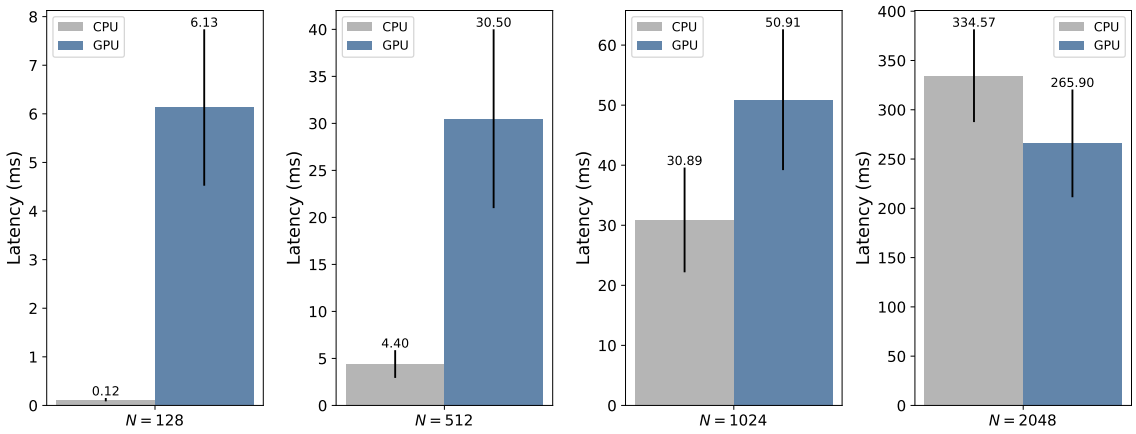


Figure 3.1: Comparison between mobile CPU and GPU on matrix multiplication kernels.

Though the table shows smartphones can have comparable memory size to cloud hardware, the memory is unified and shared by the operating system, making the actual available memory less than that in the specs. In our experiments, we observe that an application using more than 75% of the total memory can trigger swapping and can forcefully reboot the smartphone. Moreover, compared to cloud hardware which is equipped with dedicated memory, the memory throughput is much less on edge devices. For example, NVIDIA 2080Ti is equipped with GDDR6 memory whose maximum data transfer rate is 18 Gbits/s per pin [55] whereas LPDDR4X memory only delivers 4.2 Gbits/s per pin [56] at maximum. Second, the power budget for edge devices is much more constrained. A typical cloud server is often plugged and cooled down with air conditioners. In contrast, edge devices are typically running on batteries in small form factors, and heat produced by their chips can, in turn, constrain the device and even force it to run at a lower frequency to prevent the device from overheating.

Limitations on Mobile GPUs

DL accelerators such as GPUs and TPUs [25] have been critical for enabling training on cloud servers. A natural question to ask is whether using mobile GPUs can achieve similar benefits when conducting training on-device. To answer this question, we investigated the best case speedup from using mobile GPUs for on-device training based on both empirical measurements of performance-critical GPU optimized DNN operators, and theoretical FLOPS improvements of mobile GPUs over mobile CPUs.

For empirical measurements, we select two widely used kernels in DNNs: 2-dimensional convolution and matrix multiplication as representative operators. We measure the latency of these kernels with a variety of hyperparameter configurations

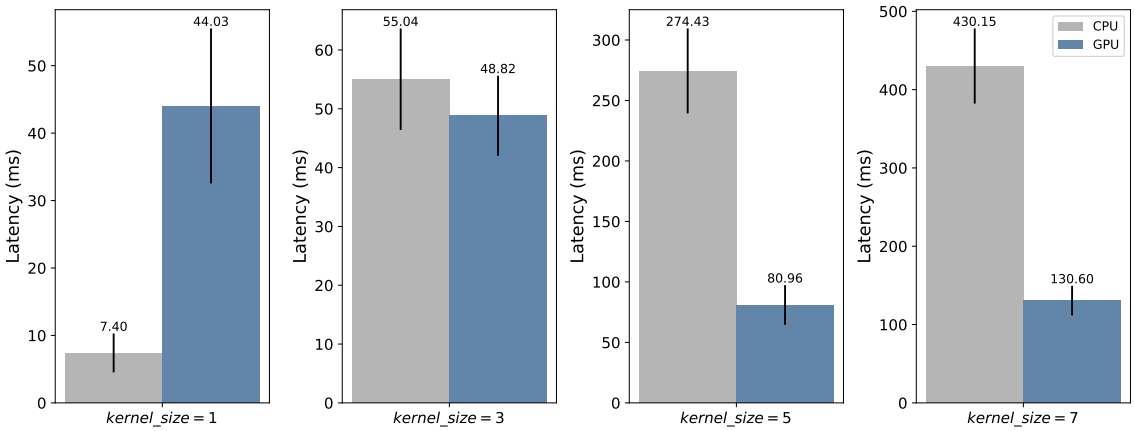


Figure 3.2: Comparison between mobile CPU and GPU on convolutional kernels.

on both the mobile CPU and GPU. Figure 3.1 shows the comparisons on the matrix multiplication kernels of two $N \times N$ matrices. We observe that using GPUs is only superior to CPUs when the size of the matrix is large. A similar phenomenon can also be seen on convolutional kernels (Figure 3.2) where the GPU has clear benefits over the CPU only when the kernel size is large enough. In particular, we observe at most a 3.3× speedup from the GPU when kernel size is 5 or 7, which are rare kernel sizes in CNNs. However, for the common kernel sizes of 1 or 3, the GPU is either 6× slower than the CPU in the former, or only 14% faster in the latter. These results contradict the common knowledge that GPUs can run much faster than CPUs on deep learning based tasks [3]. On mobile phones, GPUs are only useful when the kernel is sufficiently compute-intensive.

For our theoretical best case speedup, we optimistically considered the manufacturer's stated theoretical FLOPS of mobile GPU (1,720 FLOPS) and mobile CPU (276 FLOPS[1]). Hence, the best case speedup for using the mobile GPU is 6.2× and can shrink the on-device training time of MobileNetV2 for an image classification task (see Section 5.1.1) from 3.2 years to 6.2 months, which still cannot make on-device training practical in this scenario. Besides, in practice, the advantages of the GPU can be undermined by memory, API overheads, and temperature issues.

As a result, we conclude that naïvely replicating the cloud DNN training process in the mobile context will incur extremely long training times and is hence infeasible on today's mobile devices.

[1]The Samsung S21 Ultra 5G has 1 core at 2.84 GHz, 3 cores at 2.42 GHz, and 4 cores at 1.80 GHz and hence the theoretical throughput can be calculated as $(2.84\ \text{GHz} + 3 \times 2.4\ \text{GHz} + 4 \times 1.8\ \text{GHz}) \times 16\ \text{FLOP/cycle} = 276\ \text{FLOPS}$ [54].

3.2 Catastrophic Forgetting Phenomenon

The catastrophic forgetting phenomenon [32] naturally arises when performing incremental training on a user's local dataset, causing accuracy on the global dataset to diminish as we train the DNN. Our evaluation (Section 5.3.1) reproduces this phenomenon by incrementally training without any global dataset leading to close-to-zero global accuracy, indicating that the knowledge the DNN learned from the pre-training process is completely forgotten. In many scenarios of incremental training, retaining the accuracy of the original dataset is a must-have feature. As an example, a smart photo application can automatically extract and classify a person's face from photos. Though the goal of incremental training is to make the image classifier inside the application recognize new objects or the faces of the user's families more accurately, the classifier should still be able to classify other common objects as usual. A usable on-device incremental training framework should therefore also aim to alleviate the forgetting issue and incur affordable accuracy loss on the global dataset.

3.3 Lack of DL Framework Support

To support on-device inference, many hardware vendors design their own dedicated DL frameworks [24, 60, 38, 51, 50] to optimize DNN workloads on the device. Among these frameworks, only MNN [24] and TensorFlow Lite [51] support on-device training, but this support is still experimental. The major issues for current DL frameworks to support on-device training come from the absence of the native backward operator support, which leads to suboptimal performance of the backpropagation algorithm. For example, Android provides NNAPI [1] which can accelerate mobile inference using NPUs. On the other hand, TFLite supports GPUs for on-device DNN inference. However, they have no GPU support for backward operators. Attempting to use these accelerators for training on-device can cause partitioning of the computational graph into subgraphs running on CPUs and subgraphs running on accelerators, which results in undesirable data copies/movements and large overheads that typically negate the benefit of using these accelerators in the first place.

Chapter 4

Enabling On-device Incremental Training

In this chapter, we introduce our three key optimizations in MoIL to solve the challenges of on-device incremental training. First, global dataset mixing (GDM) allows us to alleviate the catastrophic forgetting issue [32] and to preserve the global dataset accuracy. Second, layer freezing allows us to further preserve global accuracy and reduce computational overheads of backpropagation by reducing the number of layers being trained. Finally, feature map caching can prune out unnecessary forward passes on the frozen layers during layer freezing by caching the feature maps. By combining all these three techniques, we are able to alleviate the catastrophic forgetting issue [32] and achieve global and local accuracy similar to training on centralized local/global datasets in the cloud with much lower computational requirements.

4.1 Global Dataset Mixing

Illustrated in Figure 4.1, we propose global dataset mixing (GDM) to mitigate the issue of catastrophic forgetting section 3.2 by incrementally training the DNN on a mixture of the local dataset $\mathcal{D}_L$ and a small fraction (α) of the global dataset $\mathcal{D}_G$. The hyperparameter $\alpha \in [0, 1]$ determines how much data we borrow from the pre-training dataset. Specifically, instead of just training on $\mathcal{D}_L$, we propose to train on a mixture of datasets $\mathcal{D}_M$ which is defined as follows,

$$\mathcal{D}_M = \mathcal{D}_L \cup \text{SamplePerClass}(\mathcal{D}_G, \alpha) \tag{4.1}$$

$$\text{SamplePerClass}(\mathcal{D}, \alpha) = \bigcup_{k \in \mathcal{C}} \text{Sample}(\{(x, y) \in \mathcal{D} | y = k\}, \alpha) \tag{4.2}$$

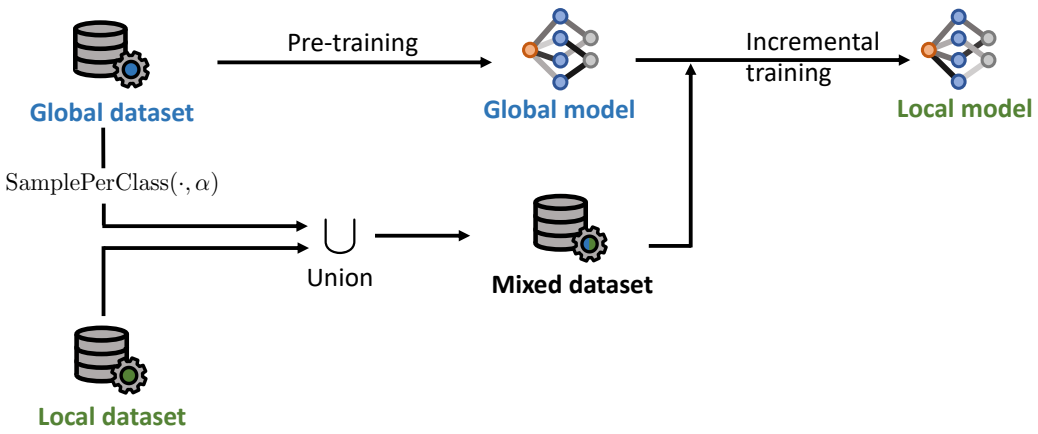


Figure 4.1: Illustration of global dataset mixing.

where Sample($\mathcal{D}, \alpha$) denotes uniformly randomly sampling α percent entries from the set $\mathcal{D}$. Note that $\alpha = 0$ means we borrow nothing from the pre-training dataset and solely train the deep model on the local dataset ($\mathcal{D}_M = \mathcal{D}_L$). When $\alpha = 1$, we train the deep model on both global and local dataset ($\mathcal{D}_M = \mathcal{D}_L \cup \mathcal{D}_G$). An N-epochs incremental training procedure on the mixed dataset $\mathcal{D}_M$ can then be formulated as $\textsc{Train}(N, \boldsymbol{\theta}_{\text{RandomInit}}, \boldsymbol{\phi}_{\text{RandomInit}}, \mathcal{D}_M, \gamma)$ (see Algorithm 1 for details).

To satisfy the restrictive compute and storage requirements of edge devices, we want α to be as small as possible. In our experiments, we found that only borrowing 1% of the pre-training data (i.e., $\alpha = 0.01$) is sufficient to preserve accuracy on the pre-training dataset (see Section 5.3.1).

4.2 Layer Freezing

Practical on-device training requires the computation in training loops to be affordable for edge devices. However, naïvely training on an edge device is two orders of magnitude slower than that on a model server. As an example, one iteration of back-propagation on MobileNetV2 on the Samsung Galaxy S21 mobile phone is 173× (6055 ms) slower than on NVIDIA RTX 2080 Ti GPU (35 ms). This gap is mainly due to the huge difference in available computational resources, as shown in Section 3.1.

We propose to freeze parts of the models and only train the remaining layers to aggressively reduce the compute requirements, which is illustrated in Figure 4.2. Specifically, we split the model $\mathcal{M}(\cdot; \boldsymbol{\theta})$ into two parts, $\mathcal{M}_F(\cdot; \boldsymbol{\theta_F})$ denoting the frozen part and $\mathcal{M}_R(\cdot; \boldsymbol{\theta}_R)$ denoting the remaining trainable part of the model, such that,

$$\mathcal{M}(\cdot; \boldsymbol{\theta}) = \mathcal{M}_R(\mathcal{M}_F(\cdot; \boldsymbol{\theta}_F); \boldsymbol{\theta}_R) \tag{4.3}$$

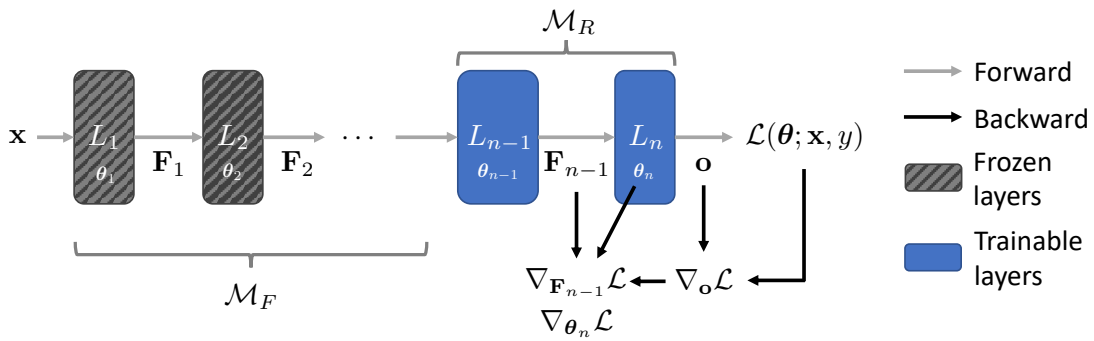


Figure 4.2: Illustration of layer freezing.

During training, we only update the trainable layers' weights $\boldsymbol{\theta}_R$ and leave the frozen layers' weights $\boldsymbol{\theta}_F$ fixed. Consequently, during backpropagation the gradients from the frozen layers do not need to be computed, thereby drastically reducing the training time. Furthermore, during the forward pass, we only need to retain the feature maps for layers being trained for backpropagation, and hence with layer freezing we have lower memory footprint, allowing us to use the same batch sizes as in cloud-based training.

In terms of accuracy, layer freezing further mitigates the forgetting issue since there are fewer trainable layers, allowing some knowledge from the global dataset to be preserved. However, it is important to note that the local accuracy can be affected by the number of trainable layers. Having fewer trainable layers results in less plasticity in the DNN. In the most extreme case, we can make the whole network not trainable, which naturally preserves all the pre-trained knowledge but also prevents personalizing the network to the user's local dataset. Therefore, we need to carefully select the number of layers to train. Fortunately, in our evaluation, we observe that we only need to make a few layers trainable to achieve acceptable local accuracy (see Section 5.3.2 for details).

4.3 Feature Map Caching

As demonstrated in Figure 4.3, when layer freezing is applied, we observe an important side effect: across different gradient update steps, if we pass the exact same batch of data through the frozen layers, we will compute the same feature maps for the frozen layers. Hence, across different gradient update steps, rather than recomputing the forward pass of frozen layers for each batch, we can instead cache the feature maps of the last frozen layer for each training sample and feed the feature map of the final frozen layer into the first trainable layer. This optimization has several

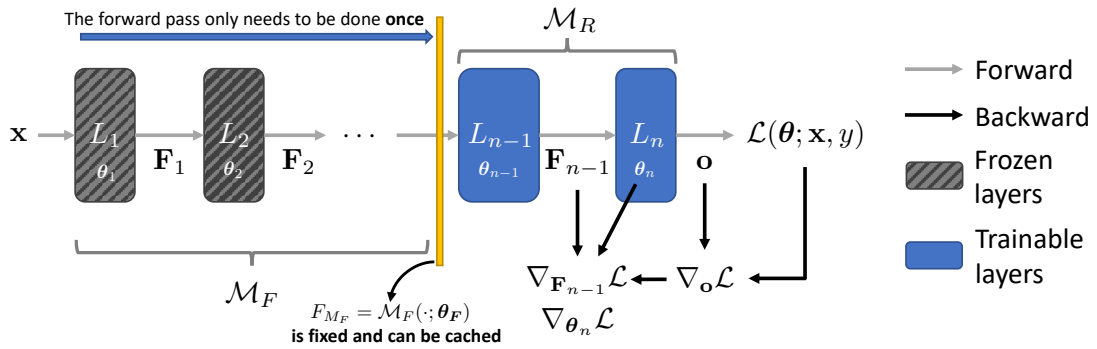


Figure 4.3: Illustration of feature map caching.

important characteristics. First, this optimization does not affect the local/global accuracy, since we are simply caching computation. Second, we only need to cache the feature maps of the *last* n-th frozen layer, since gradients will not flow to preceding layers. Third, as the model is updated, we can still re-use the cached feature map in subsequent epochs (i.e., it does not become stale) since the preceding layer weights are frozen, effectively freezing the feature maps for a given input sample. Note that besides samples from the training set, we also cache feature maps for samples from the validation set to speed up evaluation.

Algorithm 2 MoIL: Mobile Incremental Training Framework

function TRAIN(N, $\boldsymbol{\theta}_{\text{Init}}$, $\boldsymbol{\phi}_{\text{Pre-Trained}}$, $\mathcal{D}_G$, $\mathcal{D}_L$, α, γ)
  $\mathcal{D}_M \leftarrow \mathcal{D}_L \cup \text{SamplePerClass}(\mathcal{D}_G, \alpha)$ ▷ Global dataset mixing
  $[\boldsymbol{\theta}_F; \boldsymbol{\theta}_R] \leftarrow \boldsymbol{\theta}_{\text{Init}}$
  $\boldsymbol{\phi} \leftarrow \boldsymbol{\phi}_{\text{Init}}$
  for $(\mathbf{x}_i, y_i) \in \mathcal{D}_M$ **do** ▷ Cache feature maps to the disk
    $\mathbf{h}_i \leftarrow \mathcal{M}_F(\mathbf{x}_i; \boldsymbol{\theta}_F)$
    Store $\mathbf{h}_i$ to the disk.
  end for
  for $i \in [0, N)$ **do**
    for $\mathcal{B} \in \text{Batchify}(\text{RandomShuffled}(\mathcal{D}_M))$ **do**
      For each item $(\mathbf{x}_i, y_i) \in \mathbf{B}$, read corresponding $\mathbf{h}_i$ from the disk.
      $\mathbf{g} \leftarrow \nabla_{\boldsymbol{\theta}_R} \frac{1}{|\mathcal{B}|} \sum_{(\mathbf{x}_i, y_i) \in \mathcal{B}} \mathcal{L}(\mathcal{M}_R(\mathbf{h}_i; \boldsymbol{\theta}_R), y_i)$ ▷ Layer freezing
      $\boldsymbol{\theta}_R, \boldsymbol{\phi} \leftarrow \text{OptimizerStep}(\mathbf{g}, \boldsymbol{\theta}_R, \boldsymbol{\phi}, \gamma)$
    end for
  end for
  return $[\boldsymbol{\theta}_F; \boldsymbol{\theta}_R]$
end function

In Algorithm 2, we show how to augment the core SGD training algorithm to incorporate MoIL's three techniques (global dataset mixing (GDM), layer freezing (LF), feature map caching (FMC)) for enabling on-device incremental training.

Chapter 5

Evaluation

MoIL allows us to reduce total training time to within a realistic time budget of less than an hour so it can easily be performed overnight or when charging. We reduce image classification training time from 3.2 years to 0.8 hours and audio tagging training time from 0.4 years to 0.5 hours, thereby demonstrating that MoIL makes model personalization possible on today's mobile smartphones.

Currently, we mostly focused on image classification and audio tagging models, since these are common mobile use cases for which realistic scenarios for on-device training exist. Supporting additional models requires intensive engineering effort and is part of our future work. In particular, there is no major framework support for on-device training, and, as result, deploying the training stage of a new task/model requires re-implementing the entire training loop, including data pre-processing, on-device. While generalizing MoIL to other tasks may have different trade-offs, e.g., on global dataset mixing and layer freezing (as demonstrated in our ablation study in Section 5.3), we believe that it is conceptually straightforward to apply MoIL to new tasks/models, as it does not impose any major restrictions on the training datasets, loss functions, and optimizers.

5.1 Experimental Settings

The experimental platform we used is Samsung Galaxy S21 Ultra 5G [41] equipped with Snapdragon 888 SoC. Figure 5.1 shows one of our real training workloads on the device. Although production builds of TensorFlow Lite (TFLite) [51] only ship with support for inference operators, we can enable portable CPU-based implementations of backward operations by recompiling TensorFlow Lite with FlexDelegate [44] enabled[1]. Unless otherwise noted, our mobile implementation strictly uses the

[1]The TFLite we use is built from GitHub source with commit `316726a03e6`.

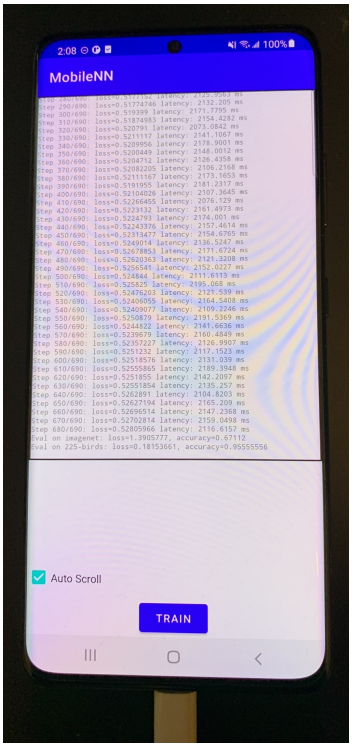


Figure 5.1: A demo of a real training workload on a smartphone.

mobile CPU, since currently there is no support from mobile SoC vendor libraries or DL frameworks for backward operators on mobile GPUs nor neural processing units (NPUs).

5.1.1 Incremental Classification with MobileNetV2

In this scenario, we consider a hypothetical user that is an avid bird watcher and has compiled a local dataset of photos of various birds (inputs) and their corresponding species (labels). The user wants to tune the image classifier on the device to better classify those specific bird species while preserving the ability to recognize common objects.

To emulate this use case, we mimic the user's routine by using ImageNet [39] as the global dataset, and the "225 bird species" dataset [11] from Kaggle as the local dataset, as illustrated in Figure 5.2. The ImageNet dataset is composed of 1,000 classes of images which denote common objects that an image classifier could be asked to classify. The 225 bird species dataset contains 225 classes of birds species, representing the dataset that the user collects. We use MobileNetV2 as our model architecture, which is widely used on smartphones for object detection and image classification [42, 59]. The global model is constructed in the cloud by training on

(a) Global dataset D_G (ImageNet) (b) Local dataset D_L (225 bird species dataset)

Figure 5.2: Examples in the global dataset D_G and the local dataset D_L in image classification.

Local	Global	Local	Global
Dog	Dog, Bark, Domestic animals, Animal	Rain	Rain, Natural sounds
Rooster	Crowing, Rooster, Livestock, Animal	Sea waves	Waves, Ocean, Natural sounds
Pig	Pig, Oink, Livestock, Animal	Laughing	Human voice, Laughter
Cow	Cattle, Moo, Livestock, Animal	Keyboard typing	Typing, Computer keyboard

Table 5.1: Examples of manual mappings from labels in the local dataset to labels in the global dataset.

ImageNet using 4 RTX 2080 Ti GPUs[2]. We use *top-1 accuracy* as a key metric to measure the model accuracy on both global and local datasets.

5.1.2 Audio Tagging with YAMNet

In this scenario, we consider a user who has downloaded a general-purpose audio tagging application and wants to customize the DNN to better recognize nearby environmental sounds (e.g., clock ticks, can opening, and fireworks). For example, this type of audio tagging technique could be useful for home surveillance camera deployments that provide users with a summary of important detected activity over the course of a day.

Different global and local audio datasets often have semantically overlapping but non-identical label names that must be reconciled before training. This scenario is more commonly encountered in audio tagging than image classification since unlike image classification tasks, audio tagging is often framed as a multi-label classification problem. Hence, it is common for a single audio sample to have multiple valid labels. For example, an audio sample of a barking dog may be labeled only with "Dog" in the local dataset, where the local label "Dog" semantically maps to the global labels "Animal" and "Bark" from the global dataset (more examples are shown in Table 5.1). Hence, in order to train on combined local and global datasets, labels of

[2]All hyperparameters to pre-train the deep model identical to the original paper [42].

the local dataset must be mapped to labels of the global dataset so that local training samples can be re-labeled accordingly[3]. This reconciliation is important to perform since otherwise the model may be arbitrarily punished for correctly predicting missing labels and may not learn effectively. We leave techniques for automatically reconciling local and global labels to future work, and instead leave the task of label reconciliation to the user.

The model we use is YAMNet [45], which is a commonly used DNN model for audio tagging tasks on smartphones. To measure the model accuracy, we use the same metric as prior works [20], Area Under the Curve (AUC) using the trapezoidal rule, as our metric function. In the pre-training process, we trained 10 epochs on AudioSet using Adam optimizer with a learning rate 0.0005. The training set contains 19,070 items and the validation set contains 17,477 items[4]. The pre-trained model can achieve nearly the same AUC score (0.956) as the official model (0.950). The local dataset we use is the ESC-50 dataset [37] containing 2,000 audios of 50 classes. We use 1,600 audios as the training set and the remaining 400 audios as the validation set.

5.2 Overall Training Time and Accuracy

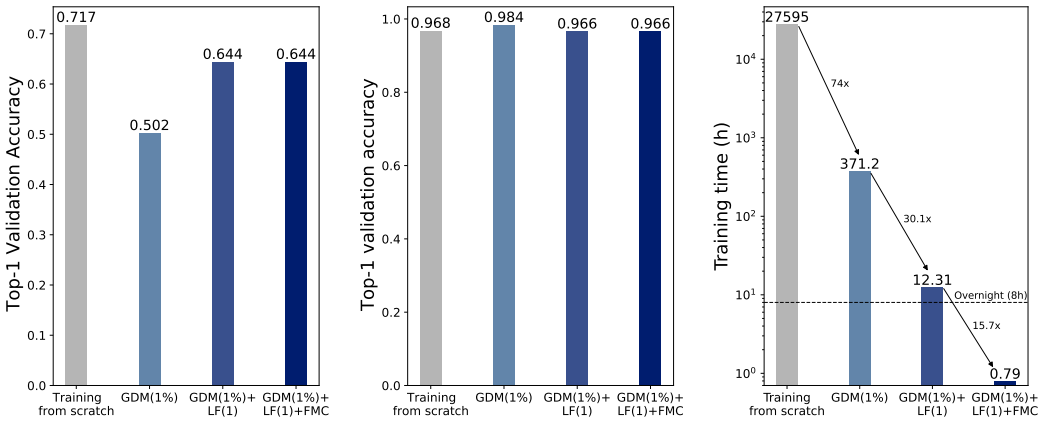
In this section, we present the training time and accuracy results for each of the two application scenarios. Given that naïvely training entire models from scratch on the phone can take years, we found it beneficial to conduct experiments exploring the local and global accuracy trade-offs of our optimizations on desktop GPUs first. Nevertheless, unless otherwise noted, all of our cited training times are based on real mobile hardware (Samsung S21 Ultra 5G), which we obtain by extrapolating from the steady-state throughput of running for 30 training iterations.

Image Classification

Figure 5.3 compares the overall total training time, global accuracy, and local accuracy of three proposed techniques on personalizing an image classifier. GDM(α) denotes global dataset mixing with α as the percent of the global dataset we mixed in for on-device training, LF(n) denotes freezing all but except the last n layers, and FMC indicates feature map caching.

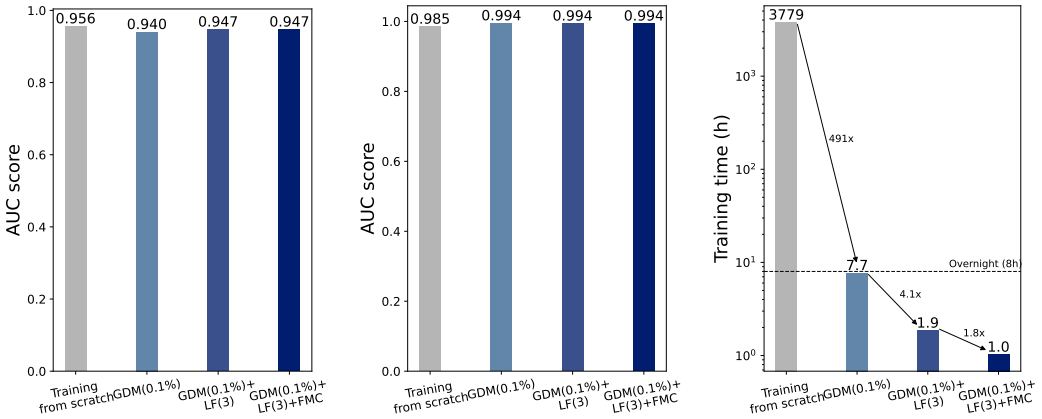
As shown in Figure 5.3a and Figure 5.3b, the naïve approach of training on both the full global and the local dataset from scratch until convergence delivers the best

[3] We choose to map local to global labels, since the global labels are more diverse and subsume the local labels.
[4] Note that some audio samples in AudioSet are not available due to copyright issues.



(a) Accuracy on global validation set D_G. (b) Accuracy on local validation set D_L. (c) Time for training till convergence.

Figure 5.3: Overall results of personalizing MobileNetV2 on image classification.



(a) Accuracy on global validation set D_G. (b) Accuracy on local validation set D_L. (c) Time for training till convergence.

Figure 5.4: Overall results of personalizing YAMNet on audio tagging.

global accuracy and close to the best local accuracy. In comparison, mixing only 1% of the global dataset (GDM(1%)) suffers from the catastrophic forgetting issue [32] and loses 21.5% in global accuracy. However, when we only train the last layer and freeze the remaining layers (LF(1)), the forgetting issue is alleviated by a large margin and consequently, we only observe 6.6% accuracy loss on the global dataset. As discussed in section 4.3, feature map caching does not affect the accuracy on both global and local datasets (the accuracy stays the same).

Though accuracy loss of 6.6% is non-negligible, we trade off this accuracy loss with the reduced training time, which is compared in Figure 5.3c. Though training from scratch can achieve better global accuracy, every training epoch consumes 184 hours, and training to convergence requires 150 epochs, which adds up to 3.2 years,

making it infeasible to conduct naïve training on such devices. In contrast, by training incrementally from a pre-trained model using GDM(1%), we only need 60 epochs and the training set (i.e. $1\%D_G \cup D_L$) is much smaller than that of the baseline (i.e. $D_G \cup D_L$). This can drastically reduce the whole training time to 371.2 hours (74× speed-up). Applying LF(1) together with GDM(1%) can reduce the training time to 12.3 hours (30.1× speed-up) since we only need to train the last layer that reduces overall backpropagation time. Feature map caching[5] can further reduce the training time to 48 minutes (15.7× speed-up) since we eliminate forward pass time by skipping computing feature maps for all but the last layer. In combination, these optimizations reduce total training time from 3.2 years down to 48 minutes which is well within an 8-hour overnight period when the device is idle for training. Hence, MoIL makes model personalization possible on today's smartphone devices.

Audio Tagging

In the scenario of audio tagging, we observe similar conclusions in Figure 5.4. The only difference is that the forgetting issue is less severe. This is mainly due to the reconciliation process we did for local dataset labels and hence the distribution on the local dataset is similar to the global dataset. As shown in Figure 5.4a, when just mixing a portion of global dataset (i.e., GDM(0.1%)), we only observe 1.6% AUC score decrease. Layer freezing can preserve more global accuracy (0.9% accuracy loss). On the local dataset (Figure 5.4b), all proposed techniques achieve almost identical results.

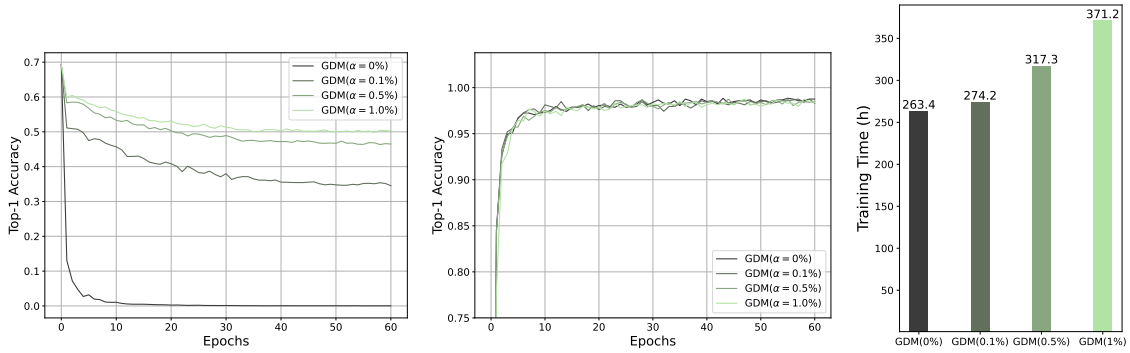
In terms of the training time (Figure 5.4c), the baseline, namely naïvely training from scratch for 10 epochs on both datasets, takes 5.2 months, which is impractical for edge devices. By leveraging 0.1% of the global dataset we can reduce the training time to 3.8 hours (491× speed-up). With layer freezing, we can further shrink the training time to 0.9 hours (4.1× speed-up). By applying feature map caching [6], we can achieve the training time of 61 minutes (1.8× speed-up), well within a limit of overnight charging scenario.

5.3 Ablation Studies

In this section, we discuss the trade-off between accuracy and training time by choosing different hyperparameters for global dataset mixing and layer freezing.

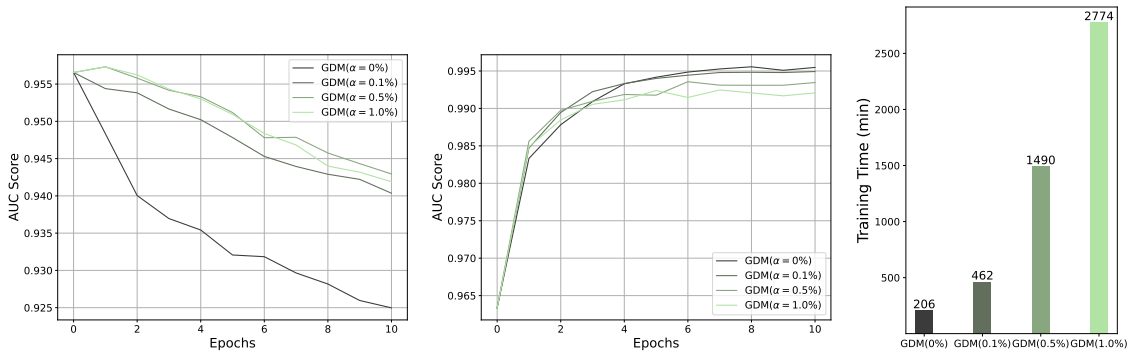
[5]Note that the cached feature maps only take $1,280$ FP32 values ($5,120$ bytes), whereas $3 \times 224^2 \approx 1.5 \times 10^5$ integers ($602,112$ bytes) to store the original image (117.6× storage reduction).

[6]The raw 10-seconds audio waveform takes 625 KB each while the feature map of each audio waveform takes 960 KB (50% more storage).



(a) Accuracy curve on the global dataset D_G. (b) Accuracy curve on the local dataset D_L. (c) Total training time.

Figure 5.5: Comparison of different sharing ratio α of global dataset mixing in image classification.



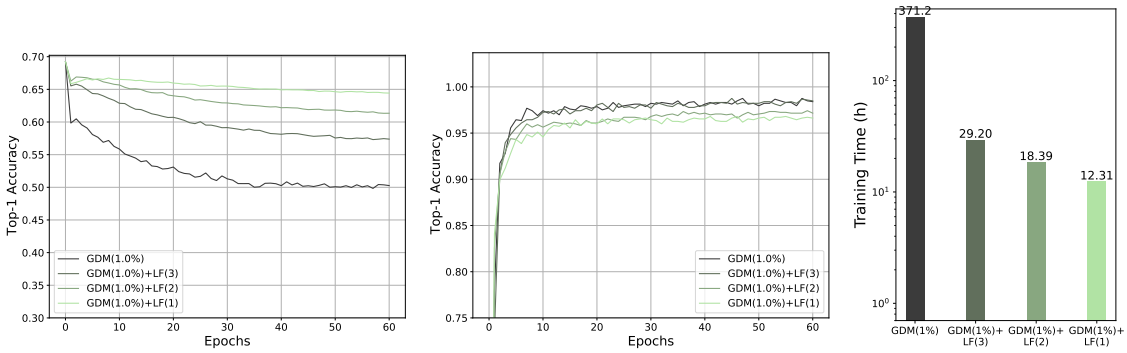
(a) Accuracy on the global dataset D_G. (b) Accuracy on the local dataset D_L. (c) Total training time.

Figure 5.6: Comparison of different sharing ratio α of global dataset mixing in audio tagging.

5.3.1 Global Dataset Mixing

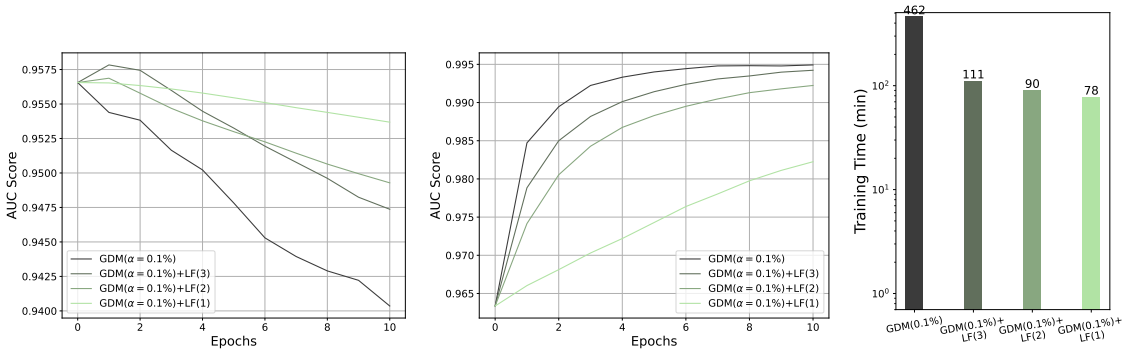
In global dataset mixing, we must select the hyperparameter α representing what percentage of the global dataset to keep on-device to mix with our local dataset when training. Large α ($\leq 100\%$) will help retain global accuracy and will increase total training time whereas small α ($\geq 0\%$) will suffer from catastrophic forgetting [32] that reduces global accuracy but decreases total training time.

Figure 5.5 explores global dataset mixing with different sharing ratios α on the image classification task. If we just do incremental training on the local dataset (i.e., GDM(0)) the accuracy on the global dataset will wane very quickly (Figure 5.5a), which is known as catastrophic forgetting [32]. In contrast, as we mix more and more global data for incremental training, the forgetting issues become less and less severe. On the local dataset, we observe the accuracy curves are almost overlapped (Figure 5.5b), demonstrating that global dataset mixing itself does not bring any accuracy penalty on the local dataset. However, even though training on more global



(a) Accuracy on the global dataset D_G. (b) Accuracy on the local dataset D_L. (c) Total training time.

Figure 5.7: Comparison among different numbers of trainable layers n for layer freezing LF(n) in image classification.



(a) AUC on the global dataset D_G. (b) AUC on the local dataset D_L. (c) Total training time.

Figure 5.8: Comparison of accuracies of different number of trainable layers n for layer freezing LF(n) in audio tagging.

data is better to preserve the global accuracy, we also increase the number of batches in every epoch, which consequently increases the total training time (Figure 5.5c). As a trade-off between the training time and the global accuracy, for image classification we choose the sharing ratio $\alpha = 1\%$.

Figure 5.6 compares different sharing ratios on the audio tagging task. On the global dataset, the forgetting issue still exists but the level of forgetting is much less severe than that of the image classification (Figure 5.6a). With at most 3% global accuracy loss on all explored α values, we chose to train YAMNet for only 10 epochs since we found the model converged on the local dataset. On the local dataset, mixing more global dataset incurs slight but negligible local accuracy loss ($< 1\%$) Figure 5.6b. In this scenario, we choose to mix $\alpha = 0.1\%$ of the global dataset.

5.3.2 Layer Freezing

Figure 5.7 compares different numbers of trainable layers for image classification. On the global dataset, we found that the more layers being frozen, the better we can preserve accuracy on the global dataset (Figure 5.7a). However, we observe slight local accuracy loss (Figure 5.7b). This is mainly attributed to less plasticity of the network due to layer freezing. However, we trade off this slight local accuracy loss with huge training time benefits. As shown in Figure 5.7c, training the last 3 layers still requires training time of 29.2 hours while only training the last layer requires only 12.3 hours. In this scenario, we chose to only train the last layer since we can already achieve high accuracy on the local dataset (96.6%).

Figure 5.8 shows the comparison in the scenario of audio tagging. On the global dataset, though the catastrophic forgetting issue [32] is much less severe, we still found that training the last layer results in the least accuracy loss on the global dataset (Figure 5.8a). However on the local dataset, training just the last layer converges much slower than training more layers (Figure 5.8b). As shown in Figure 5.8c, training more layers does not significantly increase the training time and hence we chose to train the last 3 layers (i.e., LF(3)).

Chapter 6

Conclusion

6.1 Summary

In this thesis, we discuss the limitations of current DNN deployment where model personalization can only be done in the cloud, which suffers from privacy and networking issues. Naïvely porting training directly from the cloud to the device results in training times on the order of months and even years due to mobile hardware limitations. To tackle these challenges, we propose MoIL, where we tailor the training process to mobile hardware constraints. MoIL can enable on-device incremental training of models typically only trained from scratch in the cloud, reducing on-device training time from years to merely a few hours. Further, local accuracy can be optimized *without* suffering global accuracy loss from catastrophic forgetting [32]. While on-device training is generally constrained by hardware limitations, MoIL proves that on-device training is possible and can alleviate privacy and networking issues during incremental training.

6.2 Limitations and Future Works

Though in this thesis we have shown on-device incremental training is practical by our proposed techniques, it has the following known limitations. First, for more complicated tasks such as language modeling, we need to train more layers to achieve useable accuracy. However, training more layers results in increasing training time. In this case, training DNNs for those tasks is still considered intractable and requires further considerations. Second, as mentioned in the discussion for audio tagging, we assume the local dataset is well-labeled. However, in practice, the user's dataset is often unlabeled, insufficiently labeled, or has noisy labels. For example, in a photo app, the user may be unwilling to label a large number of photos. In this scenario,

the on-device model personalization framework should be responsible to provide a user-friendly interface for labeling. Third, in this work, we assume we know the best hyperparameters for global dataset mixing and layer freezing beforehand by measuring accuracy on cloud servers. However, in practice, the hyperparameters need to be tuned automatically on the user's device, which can result in multiple training runs.

Hence, we propose several future directions. First, it has been shown that the training set can be further eliminated by importance sampling [26]. This technique is of great importance in the context of on-device training. By training on less data, we can reduce the batches needed, and hence the network can converge faster. Second, in this work, we only use CPUs for training. However, modern smartphones are equipped with dedicated deep learning accelerators such as GPUs and NPUs. By leveraging these accelerators, on-device incremental training can achieve better compute efficiency. Training DNNs on-device with a combination of these accelerators is still an open research question. Finally, we could explore other efficient training methods and DNN architectures. For example, we could combine MoIL with orthogonal techniques like quantize training that use lower precision floating numbers to train a DNN. Moreover, we could also use neural architecture search [63] to find DNN architectures that converge faster than the others. In the future, we could also explore coordination methods for decentralized distributed training among multiple nearby edge devices.

Bibliography

[1] Android NDK. *Neural Networks API | Android NDK*. URL: https://developer.android.com/ndk/guides/neuralnetworks (visited on 11/24/2021).

[2] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. "YOLOv4: Optimal Speed and Accuracy of Object Detection". In: *CoRR* abs/2004.10934 (2020). arXiv: 2004.10934. URL: https://arxiv.org/abs/2004.10934.

[3] Ebubekir BUBER and Banu DIRI. "Performance Analysis and CPU vs GPU Comparison for Deep Learning". In: *2018 6th International Conference on Control Engineering Information Technology (CEIT)*. 2018, pp. 1–6. DOI: 10.1109/CEIT.2018.8751930.

[4] Computer Vision Machine Learning Team. *An On-device Deep Neural Network for Face Detection*. Nov. 2017. URL: https://machinelearning.apple.com/research/face-detection.

[5] Matthieu Courbariaux and Yoshua Bengio. "BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1". In: *CoRR* abs/1602.02830 (2016). arXiv: 1602.02830. URL: http://arxiv.org/abs/1602.02830.

[6] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *CoRR* abs/1810.04805 (2018). arXiv: 1810.04805. URL: http://arxiv.org/abs/1810.04805.

[7] Chi Nhan Duong et al. "MobiFace: A Lightweight Deep Learning Face Recognition on Mobile Devices". In: *CoRR* abs/1811.11080 (2018). arXiv: 1811.11080. URL: http://arxiv.org/abs/1811.11080.

[8] Hironobu Fujiyoshi, Tsubasa Hirakawa, and Takayoshi Yamashita. "Deep learning-based image recognition for autonomous driving". In: *IATSS Research* 43.4 (2019), pp. 244–252. ISSN: 0386-1112. DOI: https://doi.org/10.1016/j.iatssr.2019.11.008. URL: https://www.sciencedirect.com/science/article/pii/S0386111219301566.

[9] GadgetVersus. *Apple A13 Bionic Specs*. URL: https://gadgetversus.com/processor/apple-a13-bionic-specs/.

[10] Jonas Geiping et al. "Inverting Gradients - How easy is it to break privacy in federated learning?" In: *CoRR* abs/2003.14053 (2020). arXiv: 2003.14053. URL: https://arxiv.org/abs/2003.14053.

[11] Gerry. *225 Bird Species*. Sept. 2020. URL: https://web.archive.org/web/20200920135615/https://www.kaggle.com/gpiosenka/100-bird-species.

[12] Ross B. Girshick et al. "Rich feature hierarchies for accurate object detection and semantic segmentation". In: *CoRR* abs/1311.2524 (2013). arXiv: 1311.2524. URL: http://arxiv.org/abs/1311.2524.

[13] *GitHub - albanie/convnet-burden: Memory consumption and FLOP count estimates for convnets*. URL: https://github.com/albanie/convnet-burden (visited on 12/22/2021).

[14] *GitHub - pytorch/vision: Datasets, Transforms and Models specific to Computer Vision*. https://github.com/pytorch/vision.

[15] Yu Gong et al. "EdgeRec: Recommender System on Edge in Mobile Taobao". In: *CIKM '20: The 29th ACM International Conference on Information and Knowledge Management, Virtual Event, Ireland, October 19-23, 2020*. Ed. by Mathieu d'Aquin et al. ACM, 2020, pp. 2477–2484. DOI: 10.1145/3340531.3412700. URL: https://doi.org/10.1145/3340531.3412700.

[16] Song Han, Huizi Mao, and William J. Dally. "Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding". In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2016. URL: http://arxiv.org/abs/1510.00149.

[17] Awni Y. Hannun et al. "Deep Speech: Scaling up end-to-end speech recognition". In: *CoRR* abs/1412.5567 (2014). arXiv: 1412.5567. URL: http://arxiv.org/abs/1412.5567.

[18] Andrew Hard et al. "Federated Learning for Mobile Keyboard Prediction". In: *CoRR* abs/1811.03604 (2018). arXiv: 1811.03604. URL: http://arxiv.org/abs/1811.03604.

[19] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016.

[20] Shawn Hershey et al. "CNN Architectures for Large-Scale Audio Classification". In: *CoRR* abs/1609.09430 (2016). arXiv: 1609.09430. URL: http://arxiv.org/abs/1609.09430.

[21] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. *Lecture 6a - Overview of mini-batch gradient descent*. URL: https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf. Page 15.

[22] Andrew G. Howard et al. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". In: *CoRR* abs/1704.04861 (2017). arXiv: 1704.04861. URL: http://arxiv.org/abs/1704.04861.

[23] Forrest N. Iandola et al. "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size". In: *CoRR* abs/1602.07360 (2016). arXiv: 1602.07360. URL: http://arxiv.org/abs/1602.07360.

[24] Xiaotang Jiang et al. "MNN: A Universal and Efficient Inference Engine". In: *MLSys*. 2020.

[25] Norman P. Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *CoRR* abs/1704.04760 (2017). arXiv: 1704.04760. URL: http://arxiv.org/abs/1704.04760.

[26] Angelos Katharopoulos and Francois Fleuret. "Not All Samples Are Created Equal: Deep Learning with Importance Sampling". In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, Oct. 2018, pp. 2525–2534. URL: https://proceedings.mlr.press/v80/katharopoulos18a.html.

[27] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015. URL: http://arxiv.org/abs/1412.6980.

[28] Hao Li et al. "Training Quantized Nets: A Deeper Understanding". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper/2017/file/1c303b0eed3133200cf715285011b4e4-Paper.pdf.

[29] Zhizhong Li and Derek Hoiem. "Learning without Forgetting". In: *CoRR* abs/1606.09282 (2016). arXiv: 1606.09282. URL: http://arxiv.org/abs/1606.09282.

[30] Ji Lin, Chuang Gan, and Song Han. "TSM: Temporal Shift Module for Efficient Video Understanding". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. Oct. 2019.

[31] David G Lowe. "Distinctive image features from scale-invariant keypoints". In: *International journal of computer vision* 60.2 (2004), pp. 91–110.

[32] Michael McCloskey and Neal J. Cohen. "Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem". In: ed. by Gordon H. Bower. Vol. 24. Psychology of Learning and Motivation. Academic Press, 1989, pp. 109–165. DOI: https://doi.org/10.1016/S0079-7421(08)60536-8. URL: https://www.sciencedirect.com/science/article/pii/S0079742108605368.

[33] H. Brendan McMahan et al. "Federated Learning of Deep Networks using Model Averaging". In: *CoRR* abs/1602.05629 (2016). arXiv: 1602.05629. URL: http://arxiv.org/abs/1602.05629.

[34] Austin Myers et al. "Im2Calories: Towards an Automated Mobile Vision Food Diary". In: *2015 IEEE International Conference on Computer Vision (ICCV)*. 2015, pp. 1233–1241. DOI: 10.1109/ICCV.2015.146.

[35] NanoReview. *Apple A13 Bionic: specs and benchmarks*. URL: https://nanoreview.net/en/soc/apple-a13-bionic.

[36] Jinhwan Park et al. "Fully Neural Network Based Speech Recognition on Mobile and Embedded Devices". In: *Advances in Neural Information Processing Systems*. Ed. by S. Bengio et al. Vol. 31. Curran Associates, Inc., 2018. URL: https://proceedings.neurips.cc/paper/2018/file/42299f06ee419aa5d9d07798b56779e2-Paper.pdf.

[37] Karol J. Piczak. "ESC: Dataset for Environmental Sound Classification". In: *Proceedings of the 23rd ACM International Conference on Multimedia*. MM '15. Brisbane, Australia: Association for Computing Machinery, 2015, pp. 1015–1018. ISBN: 9781450334594. DOI: 10.1145/2733373.2806390. URL: https://doi.org/10.1145/2733373.2806390.

[38] *PyTorch Mobile*. URL: https://pytorch.org/mobile/home/ (visited on 10/28/2021).

[39] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *CoRR* abs/1409.0575 (2014). arXiv: 1409.0575. URL: http://arxiv.org/abs/1409.0575.

[40] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: 10.1007/s11263-015-0816-y.

[41] *Samsung Galaxy S21 Ultra 5G | Samsung CA*. https://www.samsung.com/ca/smartphones/galaxy-s21-ultra-5g/. Jan. 2022.

[42] Mark Sandler et al. "MobileNetV2: Inverted Residuals and Linear Bottlenecks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2018.

[43] Florian Schroff, Dmitry Kalenichenko, and James Philbin. "FaceNet: A Unified Embedding for Face Recognition and Clustering". In: *CoRR* abs/1503.03832 (2015). arXiv: 1503.03832. URL: http://arxiv.org/abs/1503.03832.

[44] *Select TensorFlow operators | TensorFlow Lite*. TensorFlow. URL: https://www.tensorflow.org/lite/guide/ops_select (visited on 10/30/2021).

[45] *Sound classification with YAMNet | TensorFlow Hub*. URL: https://www.tensorflow.org/hub/tutorials/yamnet (visited on 10/25/2021).

[46] Christian Szegedy et al. "Going Deeper with Convolutions". In: *CoRR* abs/1409.4842 (2014). arXiv: 1409.4842. URL: http://arxiv.org/abs/1409.4842.

[47] Christian Szegedy et al. "Rethinking the Inception Architecture for Computer Vision". In: *CoRR* abs/1512.00567 (2015). arXiv: 1512.00567. URL: http://arxiv.org/abs/1512.00567.

[48] TechPowerUp. *NVIDIA A100 PCIe Specs*. https://www.techpowerup.com/gpu-specs/a100-pcie.c3623. Dec. 2021.

[49] TechPowerUp GPU Database. *NVIDIA A100 PCIe Specs*. URL: https://www.techpowerup.com/gpu-specs/a100-pcie.c3623.

[50] Tencent. *GitHub - Tencent/ncnn: ncnn is a high-performance neural network inference framework optimized for the mobile platform*. URL: https://github.com/Tencent/ncnn (visited on 11/24/2021).

[51] *TensorFlow Lite | ML for Mobile and Edge Devices*. URL: https://www.tensorflow.org/lite.

[52] Wikipedia. *Jacobian matrix and determinant — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Jacobian%20matrix%20and%20determinant&oldid=1049645408. [Online; accessed 29-October-2021]. 2021.

[53] Wikipedia contributors. *AI accelerator — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=AI_accelerator&oldid=1064529755. [Online; accessed 13-January-2022]. 2022.

[54] Wikipedia contributors. *FLOPS — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=FLOPS&oldid=1056204741. [Online; accessed 24-November-2021]. 2021.

[55] Wikipedia contributors. *GDDR6 SDRAM — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=GDDR6_SDRAM&oldid=1053266282. [Online; accessed 20-November-2021]. 2021.

[56] Wikipedia contributors. *LPDDR — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=LPDDR&oldid=1055855606. [Online; accessed 20-November-2021]. 2021.

[57] Thomas Wolf et al. "HuggingFace's Transformers: State-of-the-art Natural Language Processing". In: *CoRR* abs/1910.03771 (2019). arXiv: 1910.03771. URL: http://arxiv.org/abs/1910.03771.

[58] Yue Wu et al. "Large Scale Incremental Learning". In: *CoRR* abs/1905.13260 (2019). arXiv: 1905.13260. URL: http://arxiv.org/abs/1905.13260.

[59] Qian Xiang et al. "Fruit Image Classification Based on MobileNetV2 with Transfer Learning Technique". In: *Proceedings of the 3rd International Conference on Computer Science and Application Engineering*. CSAE 2019. Sanya, China: Association for Computing Machinery, 2019. ISBN: 9781450362948. DOI: 10.1145/3331453.3361658. URL: https://doi.org/10.1145/3331453.3361658.

[60] Xiaomi. *Mobile AI Compute Engine Documentation — MACE documentation*. URL: https://mace.readthedocs.io/en/latest/ (visited on 11/24/2021).

[61] Jiacheng Yang et al. "Towards Making the Most of BERT in Neural Machine Translation". In: *CoRR* abs/1908.05672 (2019). arXiv: 1908.05672. URL: http://arxiv.org/abs/1908.05672.

[62] Hongxu Yin et al. "See through Gradients: Image Batch Recovery via GradInversion". In: *CoRR* abs/2104.07586 (2021). arXiv: 2104.07586. URL: https://arxiv.org/abs/2104.07586.

[63] Barret Zoph and Quoc V. Le. "Neural Architecture Search with Reinforcement Learning". In: *CoRR* abs/1611.01578 (2016). arXiv: 1611.01578. URL: http://arxiv.org/abs/1611.01578.