Melon: Breaking the Memory Wall for Resource-Efficient On-Device Machine Learning

Qipeng Wang1*, Mengwei Xu2*, Chao Jin1, Xinran Dong1, Jinliang Yuan2, Xin Jin1, Gang Huang1, Yunxin Liu3, Xuanzhe Liu1#

1Key Lab of High Confidence Software Technologies (Peking University), Beijing, China
2State Key Laboratory of Networking and Switching Technology (BUPT), Beijing, China
3Institute for AI Industry Research (AIR), Tsinghua University, Beijing, China

wangqipeng@stu.pku.edu.cn, chaojin,dongxinran@0805,xinjinpuk,hg,xzl}@pku.edu.cn
{mwx,yuanjinliang}@bupt.edu.cn
liuyunxin@air.tsinghua.edu.cn

ABSTRACT
On-device learning is a promising technique for emerging privacy-preserving machine learning paradigms. However, through quantitative experiments, we find that commodity mobile devices cannot well support state-of-the-art DNN training with a large enough batch size, due to the limited local memory capacity. To fill the gap, we propose Melon, a memory-friendly on-device learning framework that enables the training tasks with large batch size beyond the physical memory capacity. Melon judiciously retrofits existing memory saving techniques to fit into resource-constrained mobile devices, i.e., recomputation and micro-batch. Melon further incorporates novel techniques to deal with the high memory fragmentation and memory adaptation. We implement and evaluate Melon with various typical DNN models on commodity mobile devices. The results show that Melon can achieve up to 4.33× larger batch size under the same memory budget. Given the same batch size, Melon achieves 1.89× on average (up to 4.01×) higher training throughput, and saves up to 49.43% energy compared to competitive alternatives. Furthermore, Melon reduces 78.59% computation on average in terms of memory budget adaptation.

CCS CONCEPTS
• Human-centered computing → Ubiquitous and mobile computing.
• Software and its engineering → Memory management.

KEYWORDS
Mobile device, deep learning, memory optimization

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

ACM Reference Format:
Qipeng Wang1*, Mengwei Xu2*, Chao Jin1, Xinran Dong1, Jinliang Yuan2, Xin Jin1, Gang Huang1, Yunxin Liu3, Xuanzhe Liu1#
1Key Lab of High Confidence Software Technologies (Peking University), Beijing, China
2State Key Laboratory of Networking and Switching Technology (BUPT), Beijing, China
3Institute for AI Industry Research (AIR), Tsinghua University, Beijing, China

© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9185-6/22/06...
https://doi.org/10.1145/3498361.3538928

1 INTRODUCTION
Deep Neural Networks (DNNs) have been the key component for today’s mobile apps, e.g., voice assistant, augmented reality, etc. Extensive work explores how to bring the inference stage of DNN to mobile devices by leveraging powerful hardware and various optimizations [15, 31, 32, 66, 68–70, 73, 76]. As a step forward, on-device learning is emerging as a new paradigm to directly perform model training on mobile devices, especially achieving strong privacy preservation and personalization. It has become the basis of advanced learning techniques (e.g., federated learning, split learning, etc [41, 60, 67]) and applications (e.g., input method, virtual assistant, etc [1, 2, 44]). However, due to the constrained local hardware resources, it is intuitive to ask whether the training of modern DNN is affordable on mobile devices.

Unfortunately, as we will quantitatively show in §2, even a high-end mobile device with 8GB memory cannot support DNN training with a large enough batch size, which is critical to achieve high accuracy and stable convergence [16, 54]. In other words, the memory wall hinders the training performance. In federated learning, such memory deficiency will be amplified as the low-end devices will be the bottleneck of end-to-end convergence. To this end, we aim to break the memory wall through memory optimization techniques.

Prior wisdom. We note that memory optimization for model training has been extensively studied in cloud computing for years, but seldom discussed in mobile devices. As a result, our first intuition is to investigate whether the most established cloud-side memory optimization techniques can be leveraged to mobile devices. To our surprise, our quantitative experiments in §3 reveal that cloud-side techniques can hardly apply to mobile devices: (1) Swapping [33, 42, 52, 74] introduces severe synchronization overheads because mobile SoCs lack high-speed I/O links like server GPUs do (e.g., PCIe). (2) Compression at the training time substantially compromises model accuracy, especially in the federated setting [63, 78].

Our design. We propose Melon, the first-of-its-kind memory-optimized DNN training framework that can be practically deployed on mobile devices. Melon caps the peak memory usage under a memory budget, the size of available memory for the training process specified by app or OS. Melon does not incur any accuracy drop and achieves comparable performance (e.g., training throughput [49])
works often maintain a large memory pool to manage the weights. As demonstrated later in this paper, through our novel design and techniques that are not well explored on the cloud: Melon proposes a progressive recomputation algorithm with calibrating the memory pool. Following the execution order, when the memory used is larger than the budget, Melon discards an allocated tensor and calibrates the locations of the tensors whose lifetime has "interference" with the discarded tensor. When a tensor needed by current operator is not present in memory, Melon searches all of source tensors to be recomputed and allocates memory for them. The allocating performs by extending the "time-axis", adding the tensors to the pool according to their lifetime. Then Melon calibrates the pool in the same way as aforementioned.

- **On-the-fly memory budget adapting.** The preceding two techniques are adequate to only static memory budget. However, mobile devices are multi-app or multi-task environments. Hence, Melon should support dynamic memory budgets. Simply aborting the current batch training leads to a substantial waste of computational resources, e.g., tens of seconds. To quickly respond to a new memory budget with low overhead, Melon uses an on-the-fly memory adapting mechanism. Once a new budget comes, Melon first loads the new execution plan and expands/shrinks the memory pool to meet the memory budget. It then recomputes the tensors that shall be kept in memory according to the new plan yet are not presented (due to the difference of new/old plans or the discarded memory space). Melon then adjusts the tensor locations to fit the new plan and resumes the training. In such a way, Melon reduces the switching overhead by reusing parts of previous computation results, instead of re-executes the DNN from the very beginning.

**Implementation and evaluation.** We have fully implemented Melon and 4 baselines atop MNN [26], the state-of-the-art on-device training library as we will demonstrate in §5. The decision stage runs on clients for one shot, e.g., when the app is installed, therefore incurs almost zero programming efforts to developers. We then conducted extensive experiments on four typical DNN models and four commodity Android devices. Experimental results demonstrate that Melon is adequate to support on-device training with much larger batch size (4.33×) compared to the vanilla MNN, which is much more significant than all baselines. Such a larger batch size enables Melon to accelerate the convergence progress of training job by up to 3.48× and increase the convergence accuracy by 2.2% in an end-to-end learning task. To support the same large batch size, Melon reduces up to 49.43% energy consumption compared to baselines. Furthermore, Melon saves up to 95.73% memory budget switching overhead compared to a reboot mechanism.

**Contributions** are summarized as following.

- We thoroughly measure and explore the insightful implications of promising memory optimizations for on-device training.
- We design and implement the first memory-optimized on-device training framework, Melon, with three novel techniques, i.e., lifetime-aware memory pool, memory-calibrated progressive recomputation, and on-the-fly memory adapting. The prototype of Melon have been fully open-sourced1.
- We evaluate Melon with representative DNN models and commodity mobile devices. The results demonstrate its effectiveness.

---

1[https://github.com/qipengwang/Melon](https://github.com/qipengwang/Melon)
2 MOTIVATION AND PRELIMINARIES

In this section, we briefly introduce on-device training and conduct preliminary experiments to motivate the memory wall.

2.1 On-Device Training

The ability of on-device training is the foundation of many advanced learning scenarios like federated learning [41] and on-device transfer learning [67] under the edge settings [65]. Such a need is ever-growing with the increasing public concerns over data privacy and the promulgation of related laws like GDPR [4].

On-device training typically employs the Stochastic Gradient Descent (SGD) [9], where an epoch of training can be divided into some mini-batches. The training of every single batch should experience a complete data flow: forward pass to calculate loss, backward pass to obtain the gradients, and parameter update based on gradients. Unlike the model inference (i.e., prediction) stage where the intermediate tensors can be released once they have already been used by the following layer, the training stage requires the intermediate tensors to be kept until they have already been used during the backward pass. Consequently, training is far more memory-hungry than inference.

**Breakdown.** We conduct a breakdown analysis of the peak memory footprint during the DNN training with state-of-the-art on-device training library MNN [26]. The results are demonstrated in Figure 1. We classify the memory usage into 3 categories, i.e., weight memory (storing parameters), activation memory (storing intermediate outputs), and optimizer memory (storing gradients) [55]. It shows that the activation memory often dominates the overall memory consumption and linearly scales with the batch size. It implies us to optimize this part of memory during the on-device learning process.

Table 1: The convergence result that can be achieved on devices with different memory capacities. “M-Net”: MobileNetV2; “S-Net”: SqueezeNet.

<table>
<thead>
<tr>
<th>Settings</th>
<th>convergence accuracy and round</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Accuracy</td>
</tr>
<tr>
<td>M-Net-centralized</td>
<td>67.58%</td>
</tr>
<tr>
<td>M-Net-federated</td>
<td>58.22%</td>
</tr>
<tr>
<td>S-Net-centralized</td>
<td>66.24%</td>
</tr>
<tr>
<td>S-Net-federated</td>
<td>59.18%</td>
</tr>
</tbody>
</table>

Table 3: The accuracy loss of training-time compression is amplified in federated learning compared to centralized setting, with MobileNetV2 and CIFAR-10.

2.2 The Memory Wall

Here, an intuitive yet unexplored question is: can commodity mobile devices support the training of typical DNN models towards good accuracy? In practice, the machine learning community has reached a consensus that a large batch size can help stabilize the convergence direction [16, 54]. We also conduct a measurement study on how the batch size affects the model convergence in both centralized (i.e., data in a single GPU machine) and federated settings (i.e., data is assumed to be distributed on many clients in a non-IID manner). The experiment results are shown in Figure 2. We are confirmed that the large size is a need to ensure good accuracy and convergence speed. Specifically, for MobileNetV2 with the batch size of 128 in federated settings, the training process converges at round 164, which is 45.73% faster than that with the batch size of 32. Additionally, the testing accuracy is 3.94% higher. The same observation can be found in centralized settings, where using larger batch size leads to 2% higher accuracy or 39.02% faster convergence time.

However, it is not surprising that training models with larger batch size requires much more memory capacity. In practice, commodity mobile devices cannot adequately support large-batch training, i.e., memory wall. Table 1 summarizes how the memory wall affects the on-device training. Even with a flagship high-end commodity device (Samsung Note 10, 8GB RAM), only the batch size of 32 can be supported on the MNN library, while resulting in lower accuracy and more training rounds in both centralized and federated settings.

3 EXPLORING EXISTING TECHNIQUES

In this section, we first examine the existing memory saving techniques that are originally designed for the cloud, and quantitatively analyze why these techniques are not sufficient for mobile devices.
Then we explore new design space that can probably contribute to saving training memory consumption.

- **Model & gradients compression.** Quantization [48, 64] is widely adopted to compress DNNs by reducing the number of bits required to represent each weight. For example, 8-bit and 16-bit quantization are the most common solutions to compress DNNs with negligible accuracy loss [18, 58]. In the extreme case, using 1-bit representation has been demonstrated to be effective [12, 13, 51]. However, to reduce the memory footprint, training model in low-precision representation is more challenging than the inference, and often decays the model accuracy unacceptably.

We realize that such an accuracy gap between FP32-based and INT8-based training can not be closed through advanced learning algorithms. A recent effort [78] proposes a loss-aware compensation for backward quantization, yet experienced up to 7.9% accuracy drop on CIFAR-10 dataset. Another state-of-the-art integer-based training algorithm, NITI [63], which uses a discrete parameter update scheme also drops DNN accuracy significantly. What’s even worse, such an accuracy gap could be amplified in emerging learning paradigms like FL. This phenomenon is observed in our preliminary experiments shown in Figure 3, where we compare the convergence process of NITI in both centralized and federated settings. It shows that the accuracy degradation of NITI as compared to FP32-based training is much more evident in federated settings.

- **Host-device memory swapping.** Cloud GPUs are usually equipped with dedicated memory cards and the data movement between those memory cards is very fast, e.g., 128GB/s for PCIe 5.0. Given that the main memory is typically more abundant than GPU memory, prior efforts [42, 50, 52, 61] have explored using the main memory as the external data backup during DNN training. A smart swapping mechanism can reduce the on-GPU memory footprint with marginal throughput loss, because the I/O between CPU/GPU memory can overlap with the training and the overhead can be totally covered.

However, compared to the cloud, swapping does not apply to mobile devices, which commonly use the integrated memory chip for all processors. Consequently, swapping can be performed only between the main memory and the disk on devices, where the bandwidth is very limited, e.g., 100~300MB/s as tested on the devices listed in Table 3 for the write operation. We will also experimentally show that swapping-based mechanisms exhibit inferior performance on devices in §6.

- **Activation recomputation.** The activation generated during forward pass dominates the memory usage as aforementioned. Therefore, a little literature has explored discarding the intermediate activation during forward pass and recomputing them when needed at the backward stage. Instead of discarding all of the activation, Chen et al. [11] proposed to store a subset of them, a.k.a checkpoints, and the recomputation can start from the corresponding checkpoint instead of the very beginning of a model training. A key theme of recomputation literature is to select checkpoints, atop which many algorithms have been proposed [46, 47].

We argue that recomputation is potentially useful for on-device learning, as it does not decay model accuracy and does not rely on weak characteristics between device hardware. However, current algorithms are based on a naive assumption that the sum of reserved tensor size is equal to the total memory footprint, which will be inaccurate when a user-space memory pool is used. To our best knowledge, none of them considers the effect of memory pool.

- **Splitting mini-batch to micro-batch.** With the mini-batch SGD algorithm, the weight gradients are averaged across all samples in a batch. Therefore, a mini-batch can be further split into various smaller batches, i.e. micro-batch [24, 55], and its gradients are the average of all micro-batch gradients. Our measurement in Figure 1 shows that the activation memory size is proportional to the batch size, and thus splitting mini-batch into micro-batch can significantly reduce required memory.

Micro-batch is originally designed for pipeline parallelism to achieve efficiently distributed machine learning. This technique is rarely used in the cloud to reduce the memory footprint, mainly because a small micro-batch size cannot fully utilize the high parallelism of cloud GPUs as demonstrated in Figure 4. On mobile devices, however, a relatively small batch size is sufficient to reach the maximal hardware resource utilization.

In addition, the computation correctness of micro-batch cannot be guaranteed for DNN models with BatchNormalization (BN) layer2, which involves the inter-sample data dependency. Even though algorithms like GhostBN [21] are proposed to solve this problem, the statistic change is still inevitable. Therefore, we treat micro-batch as an opportunity for on-device memory saving techniques only for the models without BN layer.

**Summarized Implications.** Through preceding measurement of existing techniques, we find that there exists a gap between mobile and cloud scenarios. On the one hand, swapping and compression, which are extensively studied in the cloud, do not suit mobile devices well. On the other hand, micro-batch brings a new opportunity whose drawback is mitigated due to limited hardware capacity of mobile device, and recomputation technique is generic enough to support various hardware and models. These findings indicate that on-device memory optimization is quite different from the cloud, leading us to build Melon as a mobile-specific framework. Especially, Melon needs to retrofit the proper techniques (micro-batch and recomputation) and, for the first time, integrate them to get the most benefit in memory saving.

---

2Note that there are many variants of BN such as BatchRenormalization, AdaBN, etc. In this work, we treat them equally in memory optimization.
we propose a novel recomputation mechanism as to be shown in
while the memory budget can be dynamically adjusted by the app
without BN layers that introduce the cross-sample dependency,
At the decision stage,
decision stage is automatically triggered before the execution stage.
in existing effort [46]. Note that both stages run on devices, and the
sor access patterns during DNN training, which has been adopted
such a two-stage design is based on the opportunity of regular ten-
execution stage,
the best performance under diverse memory budgets; (2) At the
§
§
Workflow. As shown in Figure 5, MeLon works in two stages: (1)
At the decision stage, MeLon generates execution plans that achieve
the best performance under diverse memory budgets; (2) At the
execution stage, MeLon performs DNN training based on the plans.
Such a two-stage design is based on the opportunity of regular ten-
sor access patterns during DNN training, which has been adopted
in existing effort [46]. Note that both stages run on devices, and the
decision stage is automatically triggered before the execution stage.

Figure 5: An overview of MeLon.

4 THE DESIGN
In this section, we will first give an overview of MeLon, then elabo-
rate its each novel technique.

4.1 Overview
Design goal. MeLon aims to maximize the model training perfor-
mance under given batch size and memory budget. Within a train-
ing task, the batch size is usually fixed by the algorithm developer,
while the memory budget can be dynamically adjusted by the app
or OS at runtime.

MeLon retrofits the micro-batch and recomputation techniques
for memory saving, incorporated with a novel memory pool to
reduce the memory fragmentation (§4.2). When training models
without BN layers that introduce the cross-sample dependency,
MeLon adopts the micro-batch technique. MeLon uses the largest
micro-batch size possible that satisfies the memory budget. The
overhead of micro-batch comes from two parts. First, aggregating
the buffered gradients from each micro-batch takes time, but the
overhead is trivial compared to the training time (≤1%). The second
overhead is that small batch size reduces the parallelism of intra-
op execution. This overhead is also negligible due to the limited
hardware capacity of mobile devices as discussed in §3. Therefore,
we deem that the memory wall issue of certain DNNs is well solved
by MeLon with micro-batch technique.

However, BN layer becomes the de-facto standard in DNN train-
ing (e.g., ResNet [19] and Transformers [59]). Thus, MeLon takes
a step further and focuses on supporting generic DNN models
that include BN layers through recomputation. The key design of MeLon
is to minimize the recomputation overhead by determining when
and what tensors should be discarded or recomputed. However,
directly applying pool and recomputation will face a dilemma that
both need global knowledge of each other. To tackle this problem,
we propose a novel recomputation mechanism as to be shown in
§4.3.

Workflow. As shown in Figure 5, MeLon works in two stages: (1)
At the decision stage, MeLon generates execution plans that achieve
the best performance under diverse memory budgets; (2) At the
execution stage, MeLon performs DNN training based on the plans.
Such a two-stage design is based on the opportunity of regular ten-
sor access patterns during DNN training, which has been adopted
in existing effort [46]. Note that both stages run on devices, and the
decision stage is automatically triggered before the execution stage.

Hence, such a design does not introduce any additional program-
ing efforts to the developers (e.g., only one line of shell command
in our implementation).

• Decision stage. Before training a DNN model, MeLon first
runs a profiling iteration to obtain the runtime information via
Execution Profiler. The profiled information contains NN operators
and tensors being generated during training process, including the
data flow dependency, the size of each tensor, the computation
time of each operator, the lifetime of every single tensor, etc. The
profiled information is then fed to the Execution plan generator,
which generates execution plans to elaborate the memory saving
details like: (1) where each tensor is placed in a large memory
pool; (2) which operators need to be recomputed. Additionally, the
execution plan also contains the batch splitting strategy, which
specifies the split batch size for the models without BN layers.
Because this technique has no impact on the statistic characteristics
of training process, we simply use the largest micro-batch size that
the device supports to minimize the additional overhead introduced
by aggregating the buffered gradients. The following subsections
describe how MeLon searches for the optimal execution plan.

Each execution plan corresponds to one memory budget, there-
fore MeLon pre-defines a set of memory budgets and generates
an optimal execution plan for each of them. These plans will be
stored locally with the model for execution stage. Adapting to a
set of pre-defined budgets rather than arbitrary budgets simplifies
MeLon’s design of memory optimizations. The cost is trivial as each
execution plan takes only a few KBs in our implementation.

• Execution stage. Once a training task starts, the training
engine of MeLon loads a proper execution plan according to the
current memory budget and performs the training guided by the
plan. When the memory budget changes, MeLon checks if a new
plan needs to be loaded. If needed, MeLon quickly switches to a new
plan based on the technique discussed in §4.4.

To minimize the manual efforts from developers, the decision
stage of MeLon runs on devices to automatically generate the
execution plans. The plans can be stored on local storage so they
need to be generated for only once, e.g., when the app is installed or a
new model is fetched from servers.

Figure 6: An example allocation strategy via using on-
demand strategy and our improved strategy. Each rectan-
gle in the figure represents a tensor generated, of which the
width/height indicate its lifetime/size, and its y-axis coordi-
nate is the allocated memory address.

Melon: Breaking the Memory Wall for Resource-Efficient On-Device Machine Learning MobiSys ’22, June 25–July 1, 2022, Portland, OR, USA
4.2 Lifetime-Aware Memory Pool

User-space memory pool [77] is a common approach used by training frameworks [6, 26, 45] to manage memory. It avoids the high overhead for frequently interacting with the OS to allocate/release memory blocks. Nowadays memory pools used by those frameworks allocate memory for tensors sequentially, and update the pool information after each allocation. However, such designs ignore the unique characteristic that DNN training repeats iteratively and can lead to severe memory fragmentation, e.g., up to 42% memory space is wasted in the same setting as §6.1 using MNN.

4.2.1 Opportunity and Heuristics. An opportunity to improve memory layout is the consistent memory operations across the training at batch granularity. Based on the profiled memory operating information, it is possible to architect an optimal layout with minimal memory size. Figure 6 shows an example of how the memory can be saved through a better layout. With the on-demand strategy shown in Figure 6(a), the tensor T2 is assigned to an address aside T1. After T1 is released, T4 cannot fit into the memory space below T2, therefore it should be located in the address above T2. Consequently, the total memory footprint is the sum of T1, T2 and T4. In the optimized allocation strategy shown in Figure 6(b), the memory footprint size can be reduced to the sum of T2 and T4.

However, solving the preceding memory saving problem is similar to 2DSP problem [8] — a classical NP-Hard problem. The input of this problem consists of thousands of tensors, making it impossible to exhaust the optimal solution. To obtain a near-optimal solution, very few efforts have been invested [27, 75]. These approaches usually perform the memory allocation in a greedy way of “large tensor first”. Instead, we find that tensors’ lifetime (longitude) can impose a huge influence on the layout effectiveness. Intuitively, the longer a tensor remains in the memory pool, the more “interference” it can introduce with other tensors as it splits the memory pool into two disjunct segments given by a timestamp. Indeed, such lifetime diversification is prevalent in DNN training, and can be classified into two main categories. (1) The activation spans a long lifetime, i.e., produced at the forward pass and released at the backward pass. Similar to the stack data structure, it follows a “First Produce Last Release (FPLR)” order, i.e., the earlier an activation is produced, the later it is released. (2) The lifetime of other temporary tensors is much shorter than activation, spanning only a few or even one operator. Such observations guide us to allocate each tensor according to their lifetime in a greedy way to approximate the optimal solution to this 2DSP-like problem.

4.2.2 Our Approach. Based on the preceding observations, Melon employs a tensor-lifetime-aware algorithm for memory layout optimization. The key idea is to place those long-lifetime tensors beneath short-lifetime ones to consolidate the overall memory layout. Melon iteratively places the tensor with the longest lifetime over the lowest memory address possible. The memory pool expands when a tensor’s tail exceeds the current pool size. This process is performed in a greedy fashion. With the profiled information of each tensor, Melon abstracts the memory pool and tensors into a 2D axis and rectangles as shown in Figure 6. The memory address can be represented as relative offset to the bottom of pool. During the execution stage, Melon requests all memory space at one time through the malloc function. When allocating the memory for each tensor, Melon just assigns each tensor with the certain address in the pool according to the execution plan.

4.3 Memory-Calibrated Progressive Recomputation

4.3.1 Problems of Existing Techniques. First, prior recomputation strategies [11, 46] consider only the activation generated in forward propagation. However, we observe that a lot of fragmented and temporary tensors are generated during both forward and backward stages, e.g., block F3 and B3 in Figure 7. Second, previous work makes the recomputation policy only based on the naive peak memory, i.e., the sum of all activation tensors. Indeed, among over 1,800 tensors generated during the MobileNetV2’s training forward pass with MNN, only about 200 of them are activation that needs to be persisted for long-term use. While other tensors exist for only a short lifespan, they can occupy non-trivial memory space and cause the overflow of memory usage. To our best knowledge, none of them considers the influence of memory pool, and the recomputation policy can lead to inaccurate results.

4.3.2 Our Approach. To this end, Melon introduces a different recomputation mechanism that comprehensively considers the influence of the memory pool. However, as is mentioned in §4.2, the pool needs the global knowledge of all tensors to make the allocation decision, i.e., lifetime of all tensors which can be affected by recomputation. The recomputation strategy can be made only when the information of pool is accessible, i.e., whether the tail of current tensors exceed the memory budget. In other words, both the memory pool and recomputation need the complete knowledge from each other to make a good decision. To tackle this dilemma, we introduce our memory-calibrated progressive recomputation, as shown in Algorithm 1.

Melon takes the whole operator graph as input and treats each tensor equally for recomputation. When determining which tensor to be discarded for recomputation, Melon introduces the metric Triangle Per Second (TPS) (Eq 1) to estimate the benefit of recomputing each tensor, i.e., the tensor with the larger size, longer freed lifetime, and less recomputation time has a higher priority to be discarded and recomputed later. The freed lifetime is defined as the lifetime span between discarding and recomputing. Larger size and longer freed lifetime indicate that discarding the tensor can bring more available space in the memory pool as shown in Figure 6.
Algorithm 1: Recomputation mechanism

Input: profiling, memory_budget
Output: execution plan
1 Initialize comp_seq and pool w/o recomputation;
2 allocated ← Set(); // in-memory tensors set
3 timestamp ← 0; // logical timestamp
4 Function compute(op2th, skipRelease):
5     allocated.add(op2th.output);
6     while pool.size(timestamp+1) ≥ memory_budget do
7         T ← MaxTPS(skip=skipRelease);
8         pool.Evict(T, timestamp+1);
9         allocated.add(T);
10     end while
11     timestamp ← timestamp + 1;
12     foreach op in comp_seq do
13         if op.outputs not in allocated then
14             rp_ops ← getRecomputeOpsRecursively(op1, allocated);
15             pool.Add(rp_ops.outputs, timestamp);
16             foreach op1 in recom_op do
17                 compute(op1, rp_ops.inputs);
18             end foreach
19         end if
20         compute(op1, op1.inputs);
21 end foreach

$$TPS = \frac{\text{TensorSize} \times \text{FreedLifetime}}{\text{RecomputationTime}}$$ (1)

Melon’s recomputation mechanism is carried out in a progressive manner. It first initializes the memory pool via the original execution flow (line 1), then it simulates executing operators one by one following the original execution flow (line 12). During the simulation execution, each tensor is assigned with the address exactly where it is in the pool.

When a tensor’s tail exceeds the memory budget, the recomputation mechanism is triggered (line 6–10). The recomputation mechanism continuously discards the tensor with maximal TPS value and calibrates the memory pool until the pool size is not larger than the budget. The input tensors of next operator are considered to be not discarded (line 7). During the discarding process, the pool releases tensor at current step (removes part of the rectangle as visualization in Figure 6). Once a tensor is discarded, Melon calibrates the memory address of all the tensors generated afterwards whose lifetime has “interference” with it (line 8). Here the interference is defined as the overlap of two tensors’ lifetime. As a visualization in Figure 6, the tensors on the right at this point will “sink” to lower address. Such discarding process is repeated on the in-memory tensors until the pool size does not overflow the memory budget.

When a tensor needed by the current operator is not presented in memory, the algorithm allocates memory and recomputes it along with its source tensors (line 13–19). The source tensors are collected recursively until the input tensors are presented in memory (line 14). Through such mechanism (line 7 and 14), the input dependency between operator is guaranteed. Recomputing the tensors causes nontrivial time overhead because it can produce some tensors that should be added to the pool. Therefore, Melon needs to expand the lifetime of already-in-memory tensors (time-axis in Figure 6). First, the pool extends lifetime from current step by exact the length of these tensors’ lifetime, and all of the “rectangles” right to current time will move rightward, indicating that they will be generated later. Then the tensors are added to the pool, and the pool calibrates tensors whose lifetime has interference with them in the same way.

Figure 7 illustrates an example of how Melon’s recomputation works. Assume that the operator graph in topology order is $F_1 \rightarrow F_3$ and $B_3 \rightarrow B_1$ where $F$ denotes the forward pass, and $B$ denotes the backward one. The black arrows in the figure denote the dataflow in operator graph. The activation is produced by $F_1$, $F_2$ and $F_3$, while $F_4$ and $F_5$ are operators producing temporary tensors. Assume that the memory budget is exhausted after $F_3$, the output of $F_4$ with maximal TPS will be discarded, even though the output of $F_3$ is not an activation. In this case the in-place memory allocation cannot work because the outputs of $F_1$ and $F_2$ should be kept until they are not used in the backward pass. During the backward pass, $B_3$ acts as an intermediate operator to support the computation of $B_2$. Even in the backward pass, the memory footprint size can exceed the budget, and then the algorithm chooses a tensor to be evicted in the same way as forward pass. The TPS of each tensor will be updated when a tensor is evicted.

4.4 Memory Budget Adaptation

Mobile devices typically support multi-app execution environments, where the hardware resources allocated to each app/service can be highly dynamic. Such signal of memory adaptation may come from OS or the app itself. In adapting to the new memory budget, Melon needs to (i) quickly respond to the change, e.g., releasing memory if needed, and (ii) minimize the overhead of switching the execution plan.

For the case of expanded memory budget, Melon simply employs a lazy switch strategy, i.e., waiting till the training end of the current batch and switching to the new execution plan. However, for the case of shrunk memory budget, such a lazy strategy is not feasible as the memory needs to be immediately released to the app or OS. Another intuitive method is stop-restart, which means the whole memory pool will be reallocated and all the intermediate results at current batch will be discarded. While it can release memory instantly, it also causes very high overhead to re-execute the operators.

To this end, we propose an on-the-fly memory adapting mechanism that can quickly respond to the memory budget change and resume the execution based on the preserved (partial) results. Once a new budget comes, Melon first shrinks the pool size to meet the memory budget. Melon preserves the size of new memory budget from the beginning of the current memory pool and dumps the rest through realloc function. It then loads the new execution plan and jumps to the execution point of the current operator.

The next key step is to recover the memory layout for the new plan. We use $A$, $B$, and $C$ to denote the in-memory tensor set of the old execution plan, the tensor set that ought to be presented in memory in the new execution plan, and the discarded tensor set, respectively. Melon keeps only the tensors in $(A \cap B)$ in memory and dumps others. Note that the dumping action does not need any memory operation physically but marks only the corresponding
memory blocks as free, Melon then adjusts the memory address of the remaining tensors from the old execution plan to the new one. Finally, Melon recomputes the tensor in \( C = (A - B) \) based on the execution order of the model. With the preceding steps done, Melon can successfully recover the memory layout and resume the training with the new execution plan, instead of re-executing the previous operators from scratch.

### 5 IMPLEMENTATION

We have fully implemented a prototype of Melon atop MNN (v1.1.0) [26]. To the best of our knowledge, MNN, TFLite [5], and DL4J [3] are the only three libraries that support training modern DNNs on Android devices. We use MNN because it outperforms the other two in consideration of speed and memory usage, as demonstrated in our measurement (Table 2) and prior work [10]. Note that the design of Melon is general enough to be incorporated into other libraries as well.

Our prototype mainly includes two modules (6.4k LoC in C++ in total): (1) the execution engine for offline profiling and online memory-optimized execution; (2) the execution plan generator generates the optimal execution plans under different memory budgets. Note that both of them run on devices in an automated manner, imposing no additional efforts for developers.

While MNN is conceptually compatible with both Android and iOS devices, our prototype currently targets Android devices as there are many OS-specific memory operations. Currently, the prototype mainly supports training on mobile CPU, because MNN has very limited supports for training-related operators on GPUs, and even the supported models exhibit poor performance compared to CPU [10]. It is worth mentioning that the design of Melon is mostly compatible with mobile GPU yet unique challenges need to be addressed such as the memory copy overhead during the memory adaptation. To our best knowledge, MNN is currently the only on-device training library that supports mobile GPU as of the publication of this work. The evaluation in §6.7 will demonstrate Melon’s compatibility and generality.

**Baselines.** We also implemented four baselines by learning lessons from prior literature. Note that the source code of some prior work is not available, so we try our best to reproduce them according to corresponding papers. For fair comparison, we re-implement each of them atop MNN.

- **Ideal:** the ideal case where we assume the devices are equipped with infinite memory capacity, implemented by directly reusing the device memory (thus compromising computation correctness). This baseline provides the strict upper-bound performance achievable by Melon or other baselines.
- **vDNN** [52]: a runtime memory management solution that virtualizes the memory usage of DNNs based on swapping. Here we swap data between memory and disk (internal storage on devices).
- **Sublinear** [41]: a layer-wise recomputation algorithm that evicts a tensor when the current memory usage exceeds a threshold determined by heuristics.
- **Capuchin** [46]: an efficient tensor-based optimization algorithm that combines swapping and recomputation.

### 6 EVALUATION

In this section, we evaluate Melon and baselines in various aspects to demonstrate the efficiency of Melon.

#### 6.1 Experiment Settings

**Models and datasets.** We evaluate Melon with 4 typical CNN models listed in Table 3, which are widely used on mobile devices, including MobileNetV1 [22], MobileNetV2 [53], SqueezeNet [25], and ResNet-50 [19]. For each model, we implement two versions: with and without BN layer (added after each convolution layer). We did not include language models because MNN lacks such support. We use CIFAR-100 dataset with input resized to \( 224 \times 224 \times 3 \) [25, 53].

**Hardware setup.** We conduct the experiments on 4 Android devices with diverse SoCs and memory capacity (Table 3). We always run Melon and other baselines on the big cores to achieve fair comparison.

**Metrics.** We measure the memory usage, energy consumption, and throughput during training. The memory usage is monitored by `procrank`. The energy consumption is calculated through Android’s vfs `(/sys/class/power_supply)` [10]. The training throughput is defined as the number of data samples trained per second (`throughput = \frac{\text{Batch Size}}{\text{Per Batch Latency}}`).

### Table 2: Experiments show MNN is the state-of-the-art library that supports on-device learning.

<table>
<thead>
<tr>
<th>Batch size</th>
<th>MobileNetV2</th>
<th>SqueezeNet</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TFLite MNN</td>
<td>TFLite MNN</td>
</tr>
<tr>
<td>4</td>
<td>Peak Mem. (MB)</td>
<td>2.257 1.112 1.028 849</td>
</tr>
<tr>
<td></td>
<td>Latency (ms)</td>
<td>1.757 1.474 1.156 1.016</td>
</tr>
<tr>
<td>8</td>
<td>Peak Mem. (MB)</td>
<td>2.257 1.112 1.028 849</td>
</tr>
<tr>
<td></td>
<td>Latency (ms)</td>
<td>3.777 2.675 2.239 1.981</td>
</tr>
<tr>
<td>12</td>
<td>Peak Mem. (MB)</td>
<td>3.342 1.629 1.430 1.303</td>
</tr>
<tr>
<td></td>
<td>Latency (ms)</td>
<td>4.572 3.826 3.342 2.998</td>
</tr>
</tbody>
</table>

We have fully implemented a prototype of Melon atop MNN (v1.1.0) [26]. To the best of our knowledge, MNN, TFLite [5], and DL4J [3] are the only three libraries that support training modern DNNs on Android devices. We use MNN because it outperforms the other two in consideration of speed and memory usage, as demonstrated in our measurement (Table 2) and prior work [10]. Note that the design of Melon is general enough to be incorporated into other libraries as well.

Our prototype mainly includes two modules (6.4k LoC in C++ in total): (1) the execution engine for offline profiling and online memory-optimized execution; (2) the execution plan generator generates the optimal execution plans under different memory budgets. Note that both of them run on devices in an automated manner, imposing no additional efforts for developers.

While MNN is conceptually compatible with both Android and iOS devices, our prototype currently targets Android devices as there are many OS-specific memory operations. Currently, the prototype mainly supports training on mobile CPU, because MNN has very limited supports for training-related operators on GPUs, and even the supported models exhibit poor performance compared to CPU [10]. It is worth mentioning that the design of Melon is mostly compatible with mobile GPU yet unique challenges need to be addressed such as the memory copy overhead during the memory adaptation. To our best knowledge, MNN is currently the only on-device training library that supports mobile GPU as of the publication of this work. The evaluation in §6.7 will demonstrate Melon’s compatibility and generality.

**Baselines.** We also implemented four baselines by learning lessons from prior literature. Note that the source code of some prior work is not available, so we try our best to reproduce them according to corresponding papers. For fair comparison, we re-implement each of them atop MNN.

- **Ideal:** the ideal case where we assume the devices are equipped with infinite memory capacity, implemented by directly reusing the device memory (thus compromising computation correctness). This baseline provides the strict upper-bound performance achievable by Melon or other baselines.
- **vDNN** [52]: a runtime memory management solution that virtualizes the memory usage of DNNs based on swapping. Here we swap data between memory and disk (internal storage on devices).
- **Sublinear** [41]: a layer-wise recomputation algorithm that evicts a tensor when the current memory usage exceeds a threshold determined by heuristics.
- **Capuchin** [46]: an efficient tensor-based optimization algorithm that combines swapping and recomputation.

### Table 3: Mobile devices and models used in experiments. "SD": Qualcomm snapdragon. “SN10”: Samsung Note10; “VIN3”: Vivo IQOO Neo3. “RN9P”: Redmi Note9 Pro. "RN8": Redmi Note8.

<table>
<thead>
<tr>
<th>Device</th>
<th>SoC</th>
<th>Memory</th>
<th>Model</th>
<th>Params</th>
</tr>
</thead>
<tbody>
<tr>
<td>SN10</td>
<td>SD 855</td>
<td>8 GB</td>
<td>MobileNetV1 [22]</td>
<td>3.3M</td>
</tr>
<tr>
<td>VIN3</td>
<td>SD 865</td>
<td>6 GB</td>
<td>MobileNetV2 [53]</td>
<td>2.4M</td>
</tr>
<tr>
<td>RN9P</td>
<td>SD 720</td>
<td>6 GB</td>
<td>SqueezeNet [25]</td>
<td>0.8M</td>
</tr>
<tr>
<td>RN8</td>
<td>SD 655</td>
<td>4 GB</td>
<td>ResNet50 [19]</td>
<td>23.8M</td>
</tr>
</tbody>
</table>

**Figure 8:** Maximal batch size that can be trained with different throughput. The x-axis shows decreased throughput, as trade-off for large batch size. The leftmost throughput is the training throughput of Ideal approach.

We conduct the experiments on 4 Android devices. We use CIFAR-100 dataset with input resized to \( 224 \times 224 \times 3 \) [25, 53].
We first measure the maximal batch size that can be achieved with different throughputs. We perform the experiments with MobileNetV2 and SqueezeNet (both with BN layers) on Samsung Note 10. The results are illustrated in Figure 8. It shows that Melon’s memory optimization scales quite well with different throughputs, and always outperforms the alternative approaches significantly. For example, when the throughput is 2.39fps, Melon can train MobileNetV2 with batch size 208, while other baselines achieve batch size smaller than 96.

End-to-end convergence performance. We also evaluate how Melon performs in an end-to-end learning task in both centralized and federated settings. The dataset we used is CIFAR-100. For federated settings, we initialize the training process with 10 devices, and federated settings. The dataset we used is CIFAR-100. For federated settings, we initialize the training process with 10 devices, and the distribution of data across all devices is non-IID [29]. The data on each device covers only a subset of classes. Since the experiment is to illustrate Melon’s effectiveness in trading batch size and training speed, we do not consider the device heterogeneity [72] in federated learning, but take into account only the training speed on Samsung Note10. Other settings are the same for both federated and centralized scenarios.

As demonstrated in Figure 9, by supporting a larger batch size, Melon achieves higher convergence accuracy than original vDNN, i.e., 3.94% and 3.20% with MobileNet-V2 and SqueezeNet respectively in federated settings. The convergence accuracy of Melon is 1.98% and 2.04% higher in centralized settings, respectively. On the other side, Melon significantly reduces the training time towards the same accuracy. For example, it takes 2.80× and 3.48× less time for Melon to the convergence accuracy (58.22% and 59.18%) for MobileNetV2 and Squeeze-Net compared to original vDNN, respectively.

Throughput with the same batch size. We then comprehensively investigate the training performance of Melon by varying different (upscaled) batch sizes that cannot be trained without using memory saving techniques. The experiments are performed on 4 devices, 2 for models with BN layers and 2 for models without BN layers. For each combination, we select 2–3 batch sizes, e.g., if the original maximal batch size is 32, we use 64, 96, and 128 as the testing batch sizes. The results are illustrated in Figure 11 and Figure 12, respectively.

6.2 Overall Performance

We first measure Melon’s overall performance in 3 main aspects, i.e., maximal batch size supported when achieving the same throughput, training convergence performance, and training throughput with larger batch.

Maximal batch size supported. We first measure the maximal batch size that can be achieved with different throughputs. We perform the experiments with MobileNetV2 and SqueezeNet (both with BN layers) on Samsung Note 10. The results are illustrated in Figure 11 and Figure 12. Respectively.

Throughput with the same batch size. We then comprehensively investigate the training performance of Melon by varying different (upscaled) batch sizes that cannot be trained without using memory saving techniques. The experiments are performed on 4 devices, 2 for models with BN layers and 2 for models without BN layers. For each combination, we select 2–3 batch sizes, e.g., if the original maximal batch size is 32, we use 64, 96, and 128 as the testing batch sizes. The results are illustrated in Figure 11 and Figure 12, respectively.

Figure 9: The end-to-end convergence performance with different batch size in centralized and federated settings. Model: MobileNetV2/SqueezeNet; Dataset: CIFAR-100.

Figure 10: The memory budge adaptation overhead. "M-Net": MobileNetV2; "S-Net": SqueezeNet. "RL": relayout; "RC": re-computation. The adapting overhead is defined as the computation of current batch being wasted (to be recomputed) and memory relayout due to the execution plan switching.

Our key observation is that Melon consistently and remarkably outperforms other alternative optimizing baselines, and often achieves similar performance compared to the Ideal baseline. For instance, on models with BN layers (Figure 11), Melon achieves 1.51× – 3.49× higher throughput than vDNN, 1.13× – 3.86× higher throughput than Sublinear, and 1.01× – 4.01× higher throughput than Capuchin. Melon demonstrates that its advantage is more significant on larger batch sizes, e.g., 3.25× and 3.34× improvement over Capuchin when training MobileNetV2 on Redmi Note9 Pro with batch size 64 and 128, respectively. Nevertheless, the performance gap between Melon and the Ideal baseline always increases with a larger batch size (e.g., from 10.67% to 21.21% for training MobileNetV2 on Redmi Note9 Pro), because Melon needs to more aggressively discard and recompute tensors that incur computation overhead. Among the baselines, vDNN exhibits the worst performance in most cases because of the limited data swappingspeed on mobile devices as aforementioned in §3. Note that Capuchin’s improvement almost benefits from recomputation because swapping introduces severe synchronization overhead. It also ignores the impact of memory pool, which means that it will waste more space and recompute more tensors to support larger batch size, leading to the throughput loss.

For models without BN layer (Figure 12), Melon can almost catch up with the performance Ideal baseline in arbitrary batch sizes (only less than 1% loss). Accordingly, the performance improvement over other optimizing baselines is profound as well, e.g., 1.77× on average (up to 2.66×) on Meizu 16t and 1.57× on average (up to 2.15×) on Redmi Note8. This is because, for models without BN layer, Melon leverages the micro-batch technique, which introduces little performance drop as discussed in §3. Note that when the batch size is relatively small, other baselines can also achieve relatively high performance. This is because all of the BN layers are removed, the number of which is close to the convolutional layer. In such cases,
there will be less activation and less computation, leading to high performance of baselines. However, unlike Melon, the performance of those baselines decays with a large batch size.

### 6.3 Memory Budget Adaptation

We then evaluate the memory budget adapting design of Melon presented in §4.4. We focus on the case of decreased memory budget as it is more challenging in practice. The experiments are performed with 2 models (MobileNetV2 and SqueezeNet) on Samsung Note 10. We select 2 different adaptation scenarios: switching from 6GB to 5GB for the batch size of 128 and from 4GB to 3GB for the batch size of 64. Note that in each case the memory budget is not enough to train the batch size without memory optimization. We also select 3 adaptive points, i.e. when the execution progress has reached 25%, 50% and 75% of the total. The baseline compared in this experiment is stop-restart as previously discussed in §4.4.

The results are shown in Figure 10. The adapting overhead is the time cost in new plan to reach the same operator as old plan when adapting happens, normalized to the stop-restart approach that simply discards all tensors when adapting. In comparison, Melon incurs much less adapting overhead, i.e., 4.27%–54.50%. The overhead of Melon increases at the posterior execution point, mainly because the number of tensors to be recomputed for recovering the memory layout of the new execution plan increases.

To further understand the adapting performance, we also breakdown the overhead into 2 main categories: in-memory tensor relay-out and recomputation of the missed tensors according to the new execution plan. Figure 10 shows that the recomputation overhead dominates the overall adapting overhead in most cases, especially
at posterior execution point. This is because the memory movement speed is much faster than computation on mobile devices.

6.4 Energy Consumption
Energy consumption is another key metric to be optimized due to the constrained battery capacity on mobile devices. Although Melon is mainly optimized for high training throughput instead of reducing energy consumption, we still evaluate it along with other baselines in this aspect. Here we test two models (MobileNetV2 and SqueezeNet with BN layer) on Meizu 16t device. The results are illustrated in Figure 13 with numbers normalized to the Ideal baseline.

It shows that Melon significantly reduces the energy consumption against the baselines, i.e., 22.00% - 49.43%. Compared to the Ideal baseline, the increased energy consumption of Melon is only 11.4% on average and as low as 2.1% for the best case. Melon’s improvement mainly comes from the reduced training time. The performance of vDNN is much improved compared to the training throughput, because the read/write operation is less energy-intensive than computation (about 2.5× gap). Yet it still consumes much more energy than Melon because of its lengthened training time.

6.5 Ablation Study
We further conduct a breakdown analysis of the benefit brought by each technique, i.e., lifetime-aware memory pool or memory-calibrated progressive recomputation, respectively. We evaluate the maximal batch size that each method can achieve with different throughputs. We perform the experiments with MobileNetV2 and SqueezeNet on Samsung Note10. The results are illustrated in Figure 14.

We observe that both techniques have non-trivial contribution to the improvement. For example, when the throughput is 3.07fps for MobileNetV2, the maximal batch size that our lifetime-aware memory pool and recomputation techniques can reach is 40 and 80, respectively. Combining them, the batch size can be boosted to 112, which is almost linearly proportional. Because the memory access pattern keeps the same for one model and one batch size, the improvement brought by the pool keeps the same across different throughputs. We also find that for both models with lower throughput, the improvement brought by Melon is larger than the sum of improvement brought by pool and recomputation. The reason is that as the batch size increasing, there are fewer tensors with long lifetime, which can introduce more opportunities to perform lifetime-aware allocation as illustrated in Figure 6.

6.6 Complexity Analysis
We measure the cost of our algorithm, i.e., the time to generate execution plans. First, the time of profiling is equal to that of training a batch, which can be almost negligible compared with the whole training process illustrated in Figure 9. Note that we overlap all tensors during this process, i.e., all tensors share the same piece of memory, because the statistic value has no impact on profiling. For example, it takes about 10.3s and 147MB for profiling training MobileNetV2 with the batch size of 32 using Samsung Note10. The additional time to log the per-operator latency is also negligible.

The major source of Melon’s offline time comes from the generation of execution plans. We measure this overhead in training SqueezeNet on Samsung Note10 as an example to analyze the algorithm complexity. Our used batch size spans from 64 to 172 with the step of 16. The experiment result shows that it takes 10.9s on average to generate a plan. Such latency incurs for only one shot because the generated plan can be stored permanently, so the cost of our algorithm is also acceptable in practice.

6.7 GPU support
We conduct an experiment to explore Melon’s performance on mobile GPU. We measure the maximal batch size supported. The setting is the same as §6.2. The results are illustrated in Figure 15. It shows that Melon can also achieve the largest batch size with different throughputs. The result is less impressive than on CPU, though, because Melon is mainly optimized for CPUs.

7 RELATED WORK
Reducing memory footprint of on-device learning. In addition to the four typical memory saving techniques discussed in §3, Split-CNN [28] proposed to split the weights of a single layer into
multiple sub-windows, on which memory offloading and prefetching are applied to reduce not only activation memory but also the weight memory. However, according to our measurement (Figure 1), activation often dominates the total memory footprint for mobile-oriented DNNs, therefore the benefit to apply such window-based offloading can be quite limited. A few memory saving techniques [7] are also proposed for hardware accelerators, but they are not compatible with commodity mobile devices.

Cross-model memory management. In the vision of co-running multiple DNNs simultaneously on devices, prior literature has explored the multi-task learning [20, 38], DNN packing [40, 57], weight sharing [14, 39], and weight virtualization [36] to better fit those models into the limited memory. MeLon is designed to reduce the memory footprint of single model training, which is orthogonal and compatible with those methods.

Fitting DNNs into TEE memory. A few pieces of existing work [37, 43] explored using Trusted Execution Environment (TEE), a hardware-level security mechanism, to guarantee the integrity and safety of DNN execution on devices. Given the very limited memory capacity available on TEE (typically tens of MBs for ARM TrustZone), those work ports only part of a DNN (critical layers) to TEE while leaving the rest on main memory (assumed to be enough). Instead, MeLon is designed for the case even main memory cannot support learning DNN and proposes effective techniques that trade very little performance and no accuracy drop.

General memory management of mobile OSes. Given that memory is a crucial and scarce resource of mobile devices, memory saving has long been a concerned research direction of mobile community. Existing studies mostly focus on app-level memory management [30, 34, 35, 71]. For instance, ASAP [56] used the prepping technique to achieve fast context switch of multiple apps on mobile devices. In comparison, MeLon targets on-device DNN training, and is seamlessly compatible with OS-level memory management mechanisms.

8 DISCUSSION

Extending to more model types. MeLon is evaluated on CNNs because MNN lacks the support for other model types like RNN and Transformer. Though, we believe MeLon can be easily applied to other types of models as long as they can be represented as a series of operators and tensors, and the training process can be represented as dataflow among operators. Indeed, the design of MeLon is model-independent.

Extending to other backends. MeLon is built atop MNN for its superior performance on mobile processors. Yet, MeLon’s key techniques are not bound to MNN and are compatible with other backends like TensorFlow. This is because the underlying principle is similar to those backends that perform computation on tensors via various operators and maintain a self-defined memory pool.

Other memory saving techniques. There may be other memory saving techniques like operator fusion, NEON and TVM. NEON and operator fusion are integrated with MNN, and to our best knowledge, TVM is designed for inference, making a mismatch with our design goal. MeLon presents optimization on operator- and tensor-level, which are the underlying representation of computation graph, so MeLon is compatible with other graph-level optimizing methods.

9 CONCLUSION

In this paper, we have designed and implemented MeLon, a memory-optimized DNN training framework for on-device learning. MeLon retrofits existing memory saving techniques and integrates them harmoniously to enable mobile devices to train larger batch with minimal performance loss. Our experiments have demonstrated that MeLon can adequately train the same large batch with the highest throughput compared to baselines, and achieve significant convergence speedup in end-to-end federated learning tasks.

10 ACKNOWLEDGEMENT

This work was supported by the National Key Research and Development Program of China under the grant number 2020YFB2104100, the National Natural Science Foundation of China under the grant number 61725201, 62172008, and 62102045, the Beijing Outstanding Young Scientist Program under the grant number BJWZYJH012019 1001004, and PKU-Baidu Fund Project under the grant number 2020BD007. Mengwei Xu was partly supported by Beijing Nova Program under the grant number Z211100002121118. The authors would also like to appreciate the China Cloud Credits for Research Program of Amazon in supports of cloud resources.

REFERENCES
