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

Qipeng Wang[1*], Mengwei Xu[2*#], Chao Jin[1], Xinran Dong[1], Jinliang Yuan[2], Xin Jin[1], Gang Huang[1], Yunxin Liu[3], Xuanzhe Liu[1#]
[1]Key Lab of High Confidence Software Technologies (Peking University), Beijing, China
[2]State Key Laboratory of Networking and Switching Technology (BUPT), Beijing, China
[3]Institute for AI Industry Research (AIR), Tsinghua University, Beijing, China
wangqipeng@stu.pku.edu.cn,{chaojin,dongxinran0805,xinjinpku,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

ACM Reference Format:
Qipeng Wang[1*], Mengwei Xu[2*#], Chao Jin[1], Xinran Dong[1], Jinliang Yuan[2], Xin Jin[1],, Gang Huang[1], Yunxin Liu[3], Xuanzhe Liu[1#]. 2022. Melon: Breaking the Memory Wall for Resource-Efficient On-Device Machine Learning. In *The 20th Annual International Conference on Mobile Systems, Applications and Services (MobiSys '22), June 25–July 1, 2022, Portland, OR, USA.* ACM, New York, NY, USA, 14 pages. 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, 62, 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]

*Equal contributions; #Corresponding authors.

MobiSys '22, June 25–July 1, 2022, Portland, OR, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9185-6/22/06…$15.00
https://doi.org/10.1145/3498361.3538928

and energy consumption) as the ideal case when the memory budget is unlimited.

Melon is judiciously built based on our insight of leveraging two potential techniques that are not well explored on the cloud: *micro-batch* [24] and *recomputation* [11]. Micro-batch is originally proposed for cross-GPU parallelism in distributed learning, yet rarely used to reduce memory usage in datacenters because it compromises hardware parallelism. This drawback can be well mitigated due to the limited hardware capacity of mobile devices. Another drawback is that micro-batch cannot guarantee the mathematical equivalence if the model includes BatchNormalization (BN) layers that introduce cross-sample dependency in a batch. BN layer has become a de-facto in modern DNN, e.g., ResNet [19] and Transformer [59]. Therefore, for models with BN layers, Melon leverages recomputation [11, 17, 47] as complement. Recomputation can help save memory by discarding and recomputing the intermediate tensors, but is generally regarded as less-efficient on the cloud due to the computation overhead, compared to memory swapping [23, 52]. As demonstrated later in this paper, through our novel design and techniques, the cost of recomputation is practically acceptable for on-device training.

Challenges and techniques. With a given DNN, Melon automatically generates execution plans that guide the training behavior under different memory budgets. Each execution plan elaborates the in-memory tensor allocation, micro-batch size, and recomputation scheduling policy that achieves the optimal performance under a specified memory budget. It should be noted that Melon is not simply built upon the combination of micro-batch and recomputation. Instead, we encounter the following unique challenges and solve them through our novel techniques.

• ***Lifetime-aware memory pool***. First, we observe heavy memory fragmentation during model training. On-device training frameworks often maintain a large memory pool to manage the weights and intermediate activation. However, because of different allocation policies and various memory access patterns [6, 26, 45], memory space of the pool becomes incontinuous, broken into small pieces. According to our measurements, the wasted memory of those existing memory pools can reach up to 42% during DNN training. To deal with the memory fragmentation, Melon uses a model-specific user-space memory pool that incorporates the knowledge of static memory access patterns (i.e., when a tensor is needed and how much memory it takes, or called *lifetime*) during model training. It is based on a simple yet critical observation that the tens of thousands of tensors generated during the training have diversified lifetime. The longer a tensor remains in memory, the more "interference" it is likely to cause with other tensors. Therefore, Melon uses a greedy algorithm to place long-lifetime tensors at low memory addresses to better consolidate the memory pool.

• ***Memory-calibrated progressive recomputation***. To incorporate the proposed lifetime-aware memory pool with recomputation technique, Melon faces a "chicken or the egg" dilemma. The memory pool takes the lifetime of all tensors as input and generates a tensor allocation plan as well as the total size of memory pool required. However, recomputation takes the pool size as input to make decision, which can affect the pool's strategy. Separately optimizing for each of them and simply applying one atop the other leads to suboptimal performance. Therefore, 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.

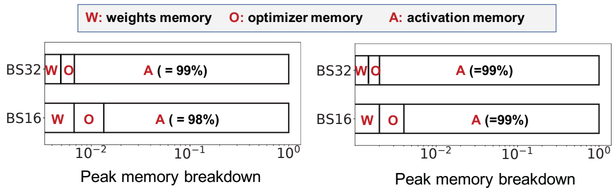
• ***On-the-fly memory budget adapting***. The preceding two techniques are adequate to only static memory budget. However, mobile devices are multi-app or multi-task environments. Hence, Melon should support dynamic memory budgets. Simply aborting the current batch training leads to a substantial waste 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-sourced[1].
- We evaluate Melon with representative DNN models and commodity mobile devices. The results demonstrate its effectiveness.

[1] https://github.com/qipengwang/Melon



(a) MobileNetV2 (b) SqueezeNet

Figure 1: The peak memory usage breakdown during DNN training using library MNN [26].

Settings	convergence accuracy and round			
	Original (BS=32)		Optimal (BS=128)	
	Accuracy	Round	Accuracy	Round
M-Net-centralized	67.58%	171	69.56%	123
M-Net-federated	58.22%	239	62.16%	164
S-Net-centralized	66.24%	211	68.28%	155
S-Net-federated	59.18%	191	62.96%	168

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

2 MOTIVATION AND PRELIMINARIES

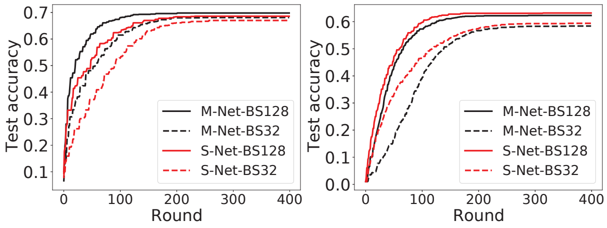
In this section, we briefly introduce on-device training and conduct preliminary experiments to motivate the memory wall.

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 outputs generated during forward pass to be kept until they have 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.



(a) Centralized learning (b) Federated learning

Figure 2: The convergence process with different batch sizes in both centralized and federated settings. Model: MobileNetV2/SqueezeNet; Dataset: CIFAR-100.

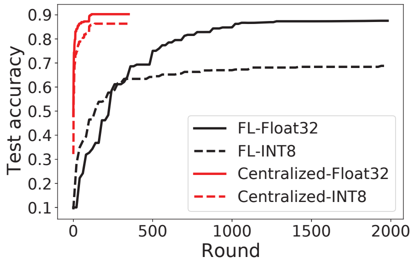


Figure 3: The accuracy loss of training-time compression is amplified in federated learning compared to centralized setting, with MobileNetV2 and CIFAR-10.

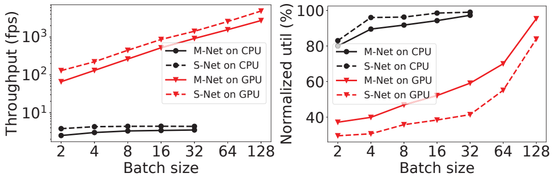
2.2 The Memory Wall

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.



(a) Training throughput **(b) Hardware utilization**

Figure 4: A relatively small batch size is enough to fully exploit mobile CPU capacity. "M/S-Net": MobileNetV2/SqueezeNet-50; "CPU": Samsung Note10 CPU; "GPU": Nvidia P100.

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) layer[2], 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.

[2] Note that there are many variants of BN such as BatchRenormalization, AdaBN, etc. In this work, we treat them equally in memory optimization.

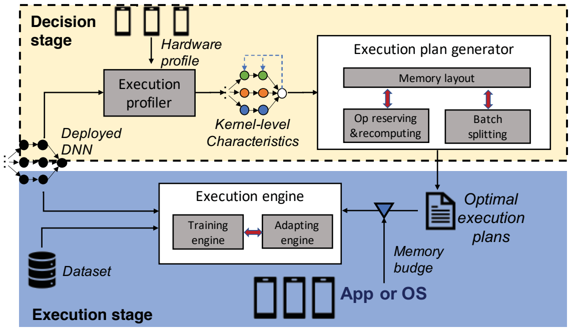


Figure 5: An overview of Melon.

4 THE DESIGN

In this section, we will first give an overview of Melon, then elaborate its each novel technique.

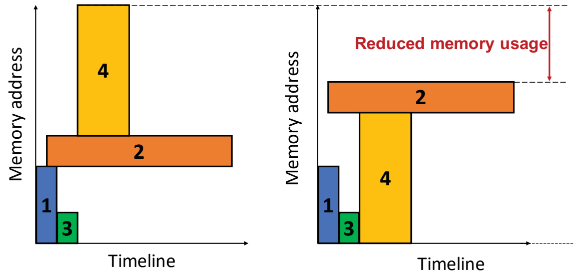
4.1 Overview

Design goal. Melon aims to maximize the model training performance under given batch size and memory budget. Within a training 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 training (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 tensor 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.



(a) On-demand memory pool (b) Improved memory pool

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

Hence, such a design does not introduce any additional programming 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, therefore 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.

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 T_2 is assigned to an address aside T_1. After T_1 is released, T_4 cannot fit into the memory space below T_2, therefore it should be located in the address above T_2. Consequently, the total memory footprint is the sum of T_1, T_2 and T_4. In the optimized allocation strategy shown in Figure 6(b), the memory footprint size can be reduced to the sum of T_2 and T_4.

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.

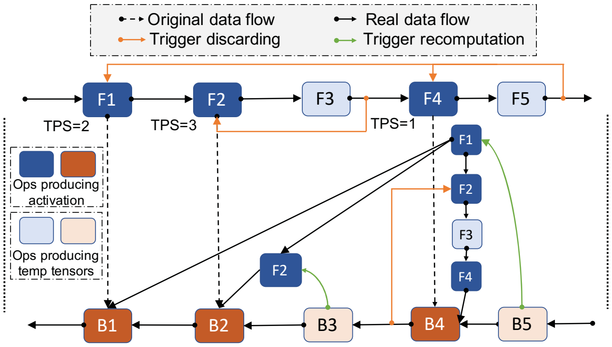


Figure 7: The recomputation workflow of Melon.

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 F_3 and B_3 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_bydget
   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($op_{ith}$, skipRelease):
5      allocated.add($op_{ith}$.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 $op_i$ in comp_seq do
13     if $op_i$.outputs not in allocated then
14         rp_ops ← getRecomputeOpsRecursively($op_i$, allocated);
15         pool.Add(rp_ops.outputs, timestamp);
16         foreach $op_j$ in recomp_ops do
17             compute($op_j$, rp_ops.inputs);
18         end foreach
19     end if
20     compute($op_i$, $op_{i+1}$.inputs);
21 end foreach
```

$$TPS = \frac{TensorSize * FreedLifetime}{RecomputationTime} \tag{1}$$

Melon's recomputation mechanism is carried out in a progressive manner. It first initializes the memory pool via original the 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 $F_1 \longrightarrow F_5$ and $B_5 \longrightarrow B_1$ 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_4, while F_3 and F_5 are operators producing temporary tensors. Assume that the memory budget is exhausted after F_3, the output of F_2 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 budge 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 $(A - C) \cap B$ in memory and dumps others. Note that the dumping action does not need any memory operation physically but marks only the corresponding

Batch size		MobileNetV2		SqueezeNet	
		TFLite	MNN	TFLite	MNN
4	Peak Mem. (MB)	2,257	1,112	1,028	849
	Latency (ms)	1,757	1,474	1,156	1,108
8	Peak Mem. (MB)	2,257	1,112	1,028	849
	Latency (ms)	3,377	2,675	2,239	1,981
12	Peak Mem. (MB)	3,242	1,629	1,430	1,303
	Latency (ms)	4,572	3,826	3,342	2,998

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

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.

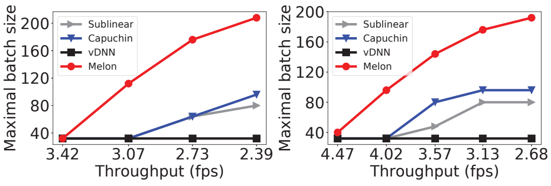
While MNN is conceptually compatible with both Android and iOS devices, our prototype currently targets Android devices as there are many OS-specific memory operations. Currently, the prototype mainly supports training on mobile CPU, because MNN has very limited supports for training-related operators 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 [11]: 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.

Device	SoC	Memory	Model	Params
SN10	SD 855	8 GB	MobileNetV1 [22]	3.3M
VIN3	SD 865	6 GB	MobileNetV2 [53]	2.4M
RN9P	SD 720	6 GB	SqueezeNet [25]	0.8M
RN8	SD 655	4 GB	ResNet50 [19]	23.8M

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.



(a) MobileNetV2 **(b) SqueezeNet**

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.

6 EVALUATION

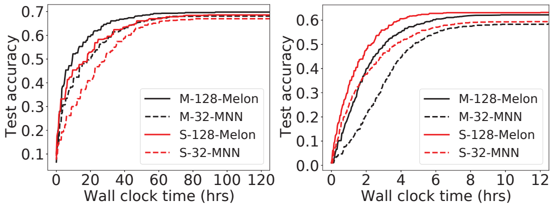
In this section, we evaluate Melon and baselines in various aspects to demonstrate the efficiency of Melon.

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 224×224×3 [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 ($throughput = \frac{BatchSize}{PerBatchLatency}$).



(a) Centralized learning **(b) Federated learning**

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

6.2 Overall Performance

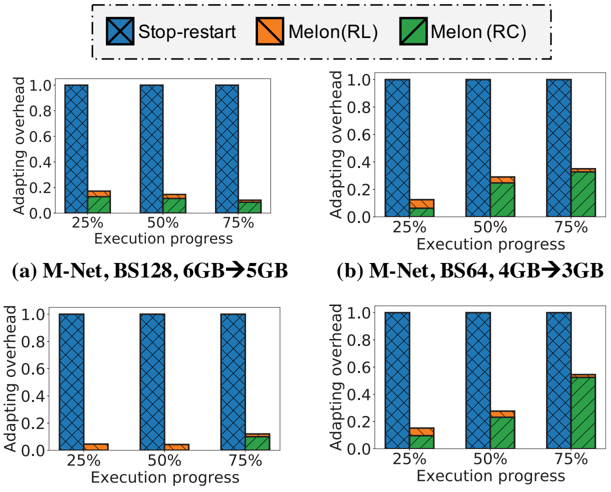
We first measure Melon's overall performance in 3 main aspects, i.e., maximal batch size supported when achieving the same throughput, training convergence performance, and training throughput with larger batch.

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 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 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 MNN, 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 MNN, 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.



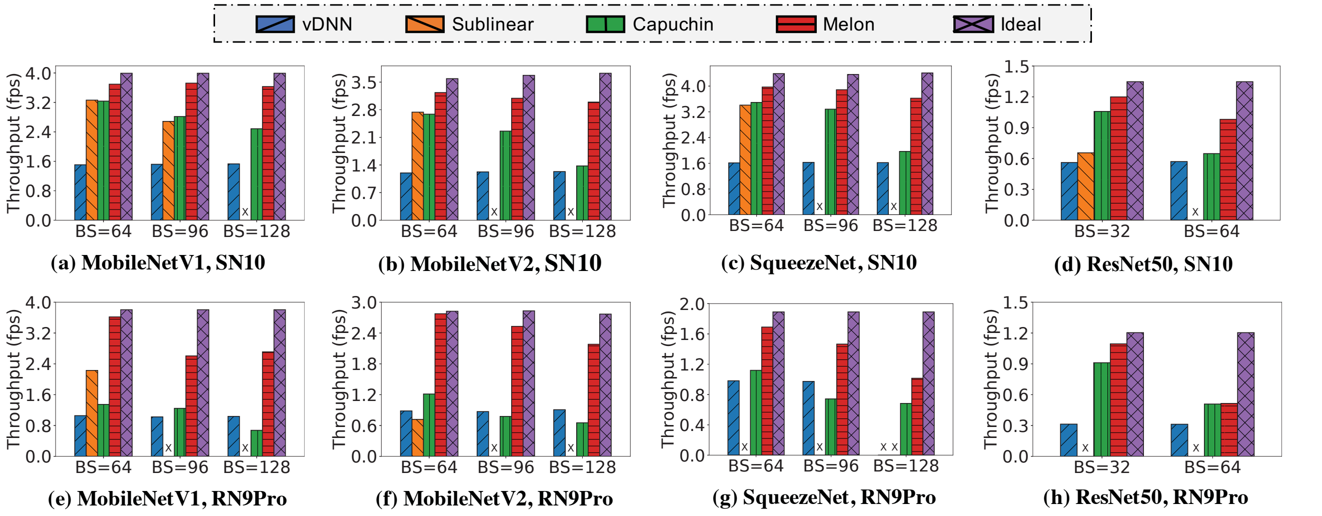
(a) M-Net, BS128, 6GB→5GB **(b) M-Net, BS64, 4GB→3GB**

(c) S-Net, BS128, 6GB→5GB **(d) S-Net, BS64, 4GB→3GB**

Figure 10: The memory budge adaptation overhead. "M-Net": MobileNetV2; "S-Net": SqueezeNet. "RL": relayout; "RC": recomputation. 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 swapping speed 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,



(a) MobileNetV1, SN10

(b) MobileNetV2, SN10

(c) SqueezeNet, SN10

(d) ResNet50, SN10

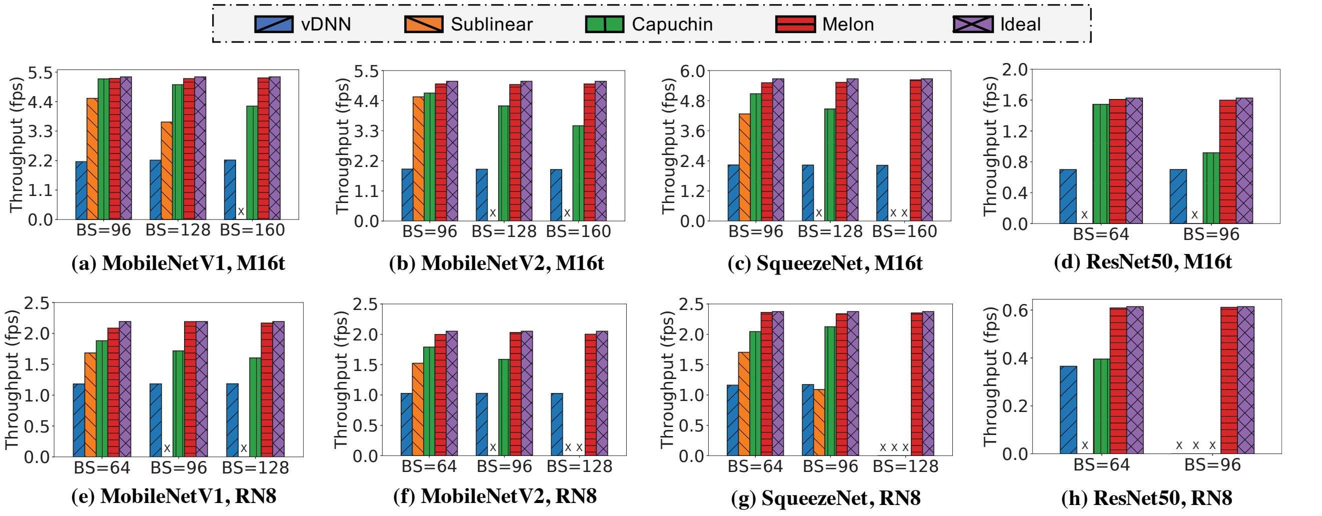
(e) MobileNetV1, RN9Pro

(f) MobileNetV2, RN9Pro

(g) SqueezeNet, RN9Pro

(h) ResNet50, RN9Pro

Figure 11: Training throughput (Y-axis) under various batch sizes (X-axis) for different models/devices with batch normalization. "X" means the approach cannot support the training of that batch size. "SN10": Samsung Note10; "RN9Pro": Redmi Note9 Pro.



(a) MobileNetV1, M16t

(b) MobileNetV2, M16t

(c) SqueezeNet, M16t

(d) ResNet50, M16t

(e) MobileNetV1, RN8

(f) MobileNetV2, RN8

(g) SqueezeNet, RN8

(h) ResNet50, RN8

Figure 12: Training throughput (Y-axis) under various batch sizes (X-axis) for different models/devices without batch normalization. "X" means the approach cannot support the training of that batch size. "M16t": Meizu 16t; "RN8": Redmi Note8.

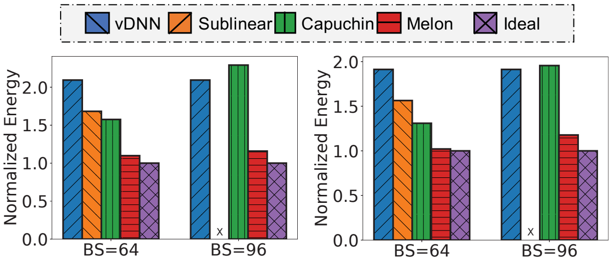
there will be less activation and less computation, leading to high performance of baselines. However, unlike Melon, the performance of those baselines decays with a large batch size.

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.

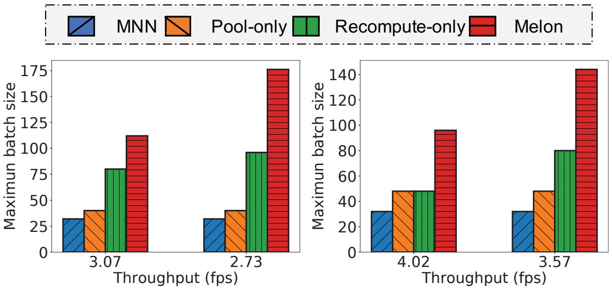
The results are shown in Figure 10. The adapting overhead is the time cost in new plan to reach the same operator as old plan when adapting happens, normalized to the *stop-restart* approach that simply discards all tensors when adapting. In comparison, Melon incurs much less adapting overhead, i.e., 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 relayout 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



(a) MobileNetV2 (b) SqueezeNet

Figure 13: The energy consumption of Melon.



(a) MobileNetV2 (b) SqueezeNet

Figure 14: Ablation study of Melon.

at posterior execution point. This is because the memory movement speed is much faster than computation on mobile devices.

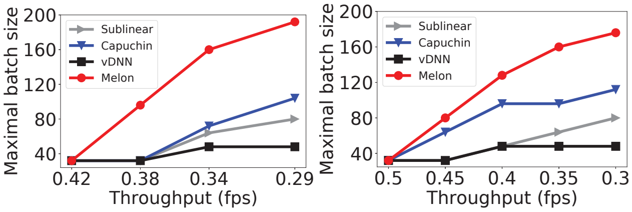
6.4 Energy Consumption

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 throughput. We perform the experiments with MobileNetV2 and SqueezeNet on Samsung Note10. The results are illustrated in Figure 14.



(a) MobileNetV2 (b) SqueezeNet

Figure 15: Maximal batch size with GPU support.

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 prepaging 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 BJJWZYJH012019 10001004, 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

[1] Federated learning: Collaborative machine learning without centralized training data. https://ai.googleblog.com/2017/04/federated-learning-collaborative.html, 2017.

[2] How apple personalizes siri without hoovering up your data. https://www.technologyreview.com/2019/12/11/131629/apple-ai-personalizes-siri-federated-learning/, 2019.

[3] Deep learning for java. https://deeplearning4j.org/, 2021.

[4] General data protection regulation. https://gdpr-info.eu/, 2021. Accessed Dec 5, 2021.

[5] On-device training with tensorflow lite. https://www.tensorflow.org/lite/examples/on_device_training/overview, 2021.

[6] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In OSDI, pages 265–283, 2016.

[7] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. Fused-layer cnn accelerators. In 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 1–12. IEEE, 2016.

[8] Andreas Bortfeldt. A genetic algorithm for the two-dimensional strip packing problem with rectangular pieces. European Journal of Operational Research, 172(3):814–837, 2006.

[9] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In Proceedings of COMPSTAT'2010, pages 177–186. Springer, 2010.

[10] Dongqi Cai, Qipeng Wang, Yuanqiang Liu, Yunxin Liu, Shangguang Wang, and Mengwei Xu. Towards ubiquitous learning: A first measurement of on-device training performance. In Proceedings of the 5th International Workshop on Embedded and Mobile Deep Learning, pages 31–36, 2021.

[11] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. arXiv preprint arXiv:1604.06174, 2016.

[12] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In Advances in neural information processing systems, pages 3123–3131, 2015.

[13] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. arXiv preprint arXiv:1602.02830, 2016.

[14] Long Duong, Trevor Cohn, Steven Bird, and Paul Cook. Low resource dependency parsing: Cross-lingual parameter sharing in a neural network parser. In Proceedings of the 53rd annual meeting of the Association for Computational Linguistics and the 7th international joint conference on natural language processing (volume 2: short papers), pages 845–850, 2015.

[15] Biyi Fang, Xiao Zeng, and Mi Zhang. Nestdnn: Resource-aware multi-tenant on-device deep learning for continuous mobile vision. In Proceedings of the 24th Annual International Conference on Mobile Computing and Networking, pages 115–127, 2018.

[16] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. arXiv preprint arXiv:1706.02677, 2017.

[17] Audrunas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. Memory-efficient backpropagation through time. Advances in Neural Information Processing Systems, 29:4125–4133, 2016.

[18] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In International conference on machine learning, pages 1737–1746. PMLR, 2015.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 770–778, 2016.

[20] Xiaoxi HE, Zimu ZHOU, and Lothar THIELE. Multi-task zipping via layer-wise neuron sharing.(2018). In Proceedings of the 32nd Annual Conference on Advances in Neural Information Processing Systems, Montréal, Canada, pages 2–8, 2018.

[21] Elad Hoffer, Itay Hubara, and Daniel Soudry. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. arXiv preprint arXiv:1705.08741, 2017.

[22] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv:1704.04861, 2017.

[23] Chien-Chin Huang, Gu Jin, and Jinyang Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In ASPLOS, pages 1341–1355, 2020.

[24] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. Advances in neural information processing systems, 32:103–112, 2019.

[25] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. arXiv preprint arXiv:1602.07360, 2016.

[26] Xiaotang Jiang, Huan Wang, Yiliu Chen, Ziqi Wu, Lichuan Wang, Bin Zou, Yafeng Yang, Zongyang Cui, Yu Cai, Tianhang Yu, et al. Mnn: A universal and efficient inference engine. arXiv preprint arXiv:2002.12418, 2020.

[27] Hai Jin, Bo Liu, Wenbin Jiang, Yang Ma, Xuanhua Shi, Bingsheng He, and Shaofeng Zhao. Layer-centric memory reuse and data migration for extreme-scale deep learning on many-core architectures. ACM Transactions on Architecture and Code Optimization (TACO), 15(3):1–26, 2018.

[28] Tian Jin and Seokin Hong. Split-cnn: Splitting window-based operations in convolutional neural networks for memory system optimization. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 835–847, 2019.

[29] Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Keith Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. Advances and open problems in federated learning. arXiv preprint arXiv:1912.04977, 2019.

[30] Sang-Hoon Kim, Jinkyu Jeong, Jin-Soo Kim, and Seungryoul Maeng. Smartlmk: A memory reclamation scheme for improving user-perceived app launch time. ACM Transactions on Embedded Computing Systems (TECS), 15(3):1–25, 2016.

[31] Youngsok Kim, Joonsung Kim, Dongju Chae, Daehyun Kim, and Jangwoo Kim. μlayer: Low latency on-device inference using cooperative single-layer acceleration and processor-friendly quantization. In Proceedings of the Fourteenth EuroSys Conference 2019, pages 1–15, 2019.

[32] Stefanos Laskaridis, Stylianos I Venieris, Mario Almeida, Ilias Leontiadis, and Nicholas D Lane. Spinn: synergistic progressive inference of neural networks over device and cloud. In Proceedings of the 26th Annual International Conference on Mobile Computing and Networking, pages 1–15, 2020.

[33] Tung D Le, Haruki Imai, Yasushi Negishi, and Kiyokuni Kawachiya. Tflms: Large model support in tensorflow by graph rewriting. arXiv preprint arXiv:1807.02037, 2018.

[34] Niel Lebeck, Arvind Krishnamurthy, Henry M Levy, and Irene Zhang. End the senseless killing: Improving memory management for mobile operating systems. In 2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20), pages 873–887, 2020.

[35] Joohyun Lee, Kyunghan Lee, Euijin Jeong, Jaemin Jo, and Ness B Shroff. Context-aware application scheduling in mobile systems: What will users do and not do next? In Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing, pages 1235–1246, 2016.

[36] Seulki Lee and Shahriar Nirjon. Fast and scalable in-memory deep multitask learning via neural weight virtualization. In Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services, pages 175–190, 2020.

[37] Taegyeong Lee, Zhiqi Lin, Saumay Pushp, Caihua Li, Yunxin Liu, Youngki Lee, Fengyuan Xu, Chenren Xu, Lintao Zhang, and Junehwa Song. Occlumency: Privacy-preserving remote deep-learning inference using sgx. In The 25th Annual International Conference on Mobile Computing and Networking, pages 1–17, 2019.

[38] Sijin Li, Zhi-Qiang Liu, and Antoni B Chan. Heterogeneous multi-task learning for human pose estimation with deep convolutional neural network. In Proceedings of the IEEE conference on computer vision and pattern recognition workshops, pages 482–489, 2014.

[39] Jiaqi Ma, Zhe Zhao, Xinyang Yi, Jilin Chen, Lichan Hong, and Ed H Chi. Modeling task relationships in multi-task learning with multi-gate mixture-of-experts. In Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pages 1930–1939, 2018.

[40] Arun Mallya and Svetlana Lazebnik. Packnet: Adding multiple tasks to a single network by iterative pruning. In Proceedings of the IEEE conference on Computer Vision and Pattern Recognition, pages 7765–7773, 2018.

[41] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In Artificial Intelligence and Statistics, pages 1273–1282. PMLR, 2017.

[42] Chen Meng, Minmin Sun, Jun Yang, Minghui Qiu, and Yang Gu. Training deeper models by gpu memory optimization on tensorflow. In Proc. of ML Systems Workshop in NIPS, 2017.

[43] Fan Mo, Hamed Haddadi, Kleomenis Katevas, Eduard Marin, Diego Perino, and Nicolas Kourtellis. Ppfl: privacy-preserving federated learning with trusted execution environments. The 19th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'21), 2021.

[44] Chaoyue Niu, Fan Wu, Shaojie Tang, Lifeng Hua, Rongfei Jia, Chengfei Lv, Zhihua Wu, and Guihai Chen. Billion-scale federated learning on mobile clients: A submodel design with tunable privacy. In Proceedings of the 26th Annual International Conference on Mobile Computing and Networking, pages 1–14, 2020.

[45] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. arXiv preprint arXiv:1912.01703, 2019.

[46] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based gpu memory management for deep learning. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 891–905, 2020.

[47] Geoff Pleiss, Danlu Chen, Gao Huang, Tongcheng Li, Laurens van der Maaten, and Kilian Q Weinberger. Memory-efficient implementation of densenets. arXiv preprint arXiv:1707.06990, 2017.

[48] Antonio Polino, Razvan Pascanu, and Dan Alistarh. Model compression via distillation and quantization. arXiv preprint arXiv:1802.05668, 2018.

[49] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21), pages 1–18. USENIX Association, July 2021.

[50] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. arXiv preprint arXiv:2104.07857, 2021.

[51] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In European conference on computer vision, pages 525–542. Springer, 2016.

[52] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 1–13. IEEE, 2016.

[53] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 4510–4520, 2018.

[54] Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V Le. Don't decay the learning rate, increase the batch size. arXiv preprint arXiv:1711.00489, 2017.

[55] Nimit Sharad Sohoni, Christopher Richard Aberger, Megan Leszczynski, Jian Zhang, and Christopher Ré. Low-memory neural network training: A technical report. arXiv preprint arXiv:1904.10631, 2019.

[56] Sam Son, Seung Yul Lee, Yunho Jin, Jonghyun Bae, Jinkyu Jeong, Tae Jun Ham, Jae W Lee, and Hongil Yoon. {ASAP}: Fast mobile application switch via adaptive prepaging. In 2021 {USENIX} Annual Technical Conference ({USENIX}{ATC} 21), pages 365–380, 2021.

[57] Dasaratha V Sridhar, Eric B Bartlett, and Richard C Seagrave. An information theoretic approach for combining neural network process models. Neural Networks, 12(6):915–926, 1999.

[58] Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. Improving the speed of neural networks on cpus. In Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011, 2011.

[59] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Advances in neural information processing systems, pages 5998–6008, 2017.

[60] Praneeth Vepakomma, Otkrist Gupta, Tristan Swedish, and Ramesh Raskar. Split learning for health: Distributed deep learning without sharing raw patient data. arXiv preprint arXiv:1812.00564, 2018.

[61] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: Dynamic gpu memory management for training deep neural networks. In Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming, pages 41–53, 2018.

[62] Manni Wang, Shaohua Ding, Ting Cao, Yunxin Liu, and Fengyuan Xu. Asymo: scalable and efficient deep-learning inference on asymmetric mobile cpus. In MobiCom, pages 215–228, 2021.

[63] Maolin Wang, Seyedramin Rasoulinezhad, Philip HW Leong, and Hayden KH So. Niti: Training integer neural networks using integer-only arithmetic. arXiv preprint arXiv:2009.13108, 2020.

[64] Bernard Widrow, Istvan Kollar, and Ming-Chang Liu. Statistical theory of quantization. IEEE Transactions on instrumentation and measurement, 45(2):353–361, 1996.

[65] Mengwei Xu, Zhe Fu, Xiao Ma, Li Zhang, Yanan Li, Feng Qian, Shangguang Wang, Ke Li, Jingyu Yang, and Xuanzhe Liu. From cloud to edge: a first look at public edge platforms. In Proceedings of the 21st ACM Internet Measurement Conference, pages 37–53, 2021.

[66] Mengwei Xu, Jiawei Liu, Yuanqiang Liu, Felix Xiaozhu Lin, Yunxin Liu, and Xuanzhe Liu. A first look at deep learning apps on smartphones. In The World Wide Web Conference, pages 2125–2136, 2019.

[67] Mengwei Xu, Feng Qian, Qiaozhu Mei, Kang Huang, and Xuanzhe Liu. Deeptype: On-device deep learning for input personalization service with minimal privacy concern. IMWUT, 2(4):1–26, 2018.

[68] Mengwei Xu, Feng Qian, Mengze Zhu, Feifan Huang, Saumay Pushp, and Xuanzhe Liu. Deepwear: Adaptive local offloading for on-wearable deep learning. IEEE Transactions on Mobile Computing, 19(2):314–330, 2019.

[69] Mengwei Xu, Xiwen Zhang, Yunxin Liu, Gang Huang, Xuanzhe Liu, and Felix Xiaozhu Lin. Approximate query service on autonomous iot cameras. In Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services, pages 191–205, 2020.

[70] Mengwei Xu, Mengze Zhu, Yunxin Liu, Felix Xiaozhu Lin, and Xuanzhe Liu. Deepcache: Principled cache for mobile deep vision. In Proceedings of the 24th Annual International Conference on Mobile Computing and Networking, pages 129–144, 2018.

[71] Tingxin Yan, David Chu, Deepak Ganesan, Aman Kansal, and Jie Liu. Fast app launching for mobile devices using predictive user context. In Proceedings of the 10th international conference on Mobile systems, applications, and services, pages 113–126, 2012.

[72] Chengxu Yang, QiPeng Wang, Mengwei Xu, Shangguang Wang, Kaigui Bian, and Xuanzhe Liu. Heterogeneity-aware federated learning. arXiv preprint arXiv:2006.06983, 2020.

[73] Hyunho Yeo, Chan Ju Chong, Youngmok Jung, Juncheol Ye, and Dongsu Han. Nemo: enabling neural-enhanced video streaming on commodity mobile devices. In Proceedings of the 26th Annual International Conference on Mobile Computing and Networking, pages 1–14, 2020.

[74] Yuan Yu, Martín Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, et al. Dynamic control flow in large-scale machine learning. In Proceedings of the Thirteenth EuroSys Conference, pages 1–15, 2018.

[75] Junzhe Zhang, Sai Ho Yeung, Yao Shu, Bingsheng He, and Wei Wang. Efficient memory management for gpu-based deep learning systems. arXiv preprint arXiv:1903.06631, 2019.

[76] Qiyang Zhang, Xiang Li, Xiangying Che, Xiao Ma, Ao Zhou, Mengwei Xu, Shangguang Wang, Yun Ma, and Xuanzhe Liu. A comprehensive benchmark of deep learning libraries on mobile devices. In Proceedings of the ACM Web Conference 2022, pages 3298–3307, 2022.

[77] Qin Zhao, Rodric Rabbah, and Weng-Fai Wong. Dynamic memory optimization using pool allocation and prefetching. ACM SIGARCH Computer Architecture News, 33(5):27–32, 2005.

[78] Qihua Zhou, Song Guo, Zhihao Qu, Jingcai Guo, Zhenda Xu, Jiewei Zhang, Tao Guo, Boyuan Luo, and Jingren Zhou. Octo: Int8 training with loss-aware compensation and backward quantization for tiny on-device learning. In 2021 {USENIX} Annual Technical Conference ({USENIX} {ATC} 21), pages 177–191, 2021.