FedHybrid: Breaking the Memory Wall of Federated Learning via Hybrid Tensor Management

Kahou Tam
State Key Laboratory of IoTSC
University of Macau
Macau SAR, China
tamkahou.2023@connect.um.edu.mo

Chunlin Tian
State Key Laboratory of IoTSC
University of Macau
Macau SAR, China
yc27402@um.edu.mo

Li Li*
State Key Laboratory of IoTSC
University of Macau
Macau SAR, China
llili@um.edu.mo

Haikai Zhao
Simon Fraser University
Canada
hza214@sfu.ca

ChengZhong Xu
State Key Laboratory of IoTSC
University of Macau
Macau SAR, China
czxu@um.edu.mo

ABSTRACT

Federated Learning (FL) emerges as a new learning paradigm that enables multiple devices to collaboratively train a shared model while preserving data privacy. However, one fundamental and prevailing challenge that hinders the deployment of FL on mobile devices is the memory limitation. This paper proposes *FedHybrid*, a novel framework that effectively reduces the memory footprint during the training process while guaranteeing the model accuracy and the overall training progress. Specifically, *FedHybrid* first selects the participating devices for each training round by jointly evaluating their memory budget, computing capability, and data diversity. After that, it judiciously analyzes the computational graph and generates an execution plan for each selected client in order to meet the corresponding memory budget while minimizing the training delay through employing a hybrid of recomputation and compression techniques according to the characteristic of each tensor. During the local training process, *FedHybrid* carries out the execution plan with a well-designed activation compression technique to effectively achieve memory reduction with minimum accuracy loss. We conduct extensive experiments to evaluate *FedHybrid* on both simulation and off-the-shelf mobile devices. The experiment results demonstrate that *FedHybrid* achieves up to a 39.1% increase in model accuracy and a 15.5× reduction in wall clock time under various memory budgets compared with the baselines.

CCS CONCEPTS

• **Computing methodologies** → **Distributed computing methodologies**; • **Human-centered computing** → **Ubiquitous and mobile computing**.

*Corresponding author

SenSys '24, November 4–7, 2024, Hangzhou, China

ACM ISBN 979-8-4007-0697-4/24/11
https://doi.org/10.1145/3666025.3699346

KEYWORDS

Federated learning, Mobile computing, Memory optimization.

1 INTRODUCTION

Federated Learning (FL) [44] coordinates multiple mobile devices to collaboratively train a shared model while preserving data privacy [29]. Most existing FL approaches [33, 37, 48, 51] simply assume that all the participating clients have sufficient resources to update the local model with their own private data. However, in real-world cases, a fundamental challenge that impedes the deployment of FL on mobile devices is memory limitation. During the local training process, the model weight, intermediate activation, and optimizer states are mandated to be stored in the memory. For instance, training MobileNetV2 [50], specifically designed for on-device vision tasks, necessitates over 8GB of memory. However, commercial mobile devices typically only have 4GB to 12GB of RAM [13]. More importantly, in order to retrieve high analysis quality, the model architecture is becoming deeper and wider and the memory requirement keeps increasing.

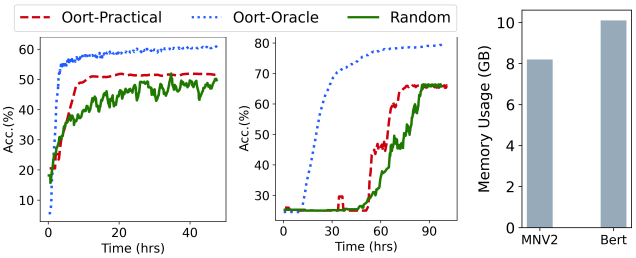
The Memory Wall in FL. Unfortunately, the memory limitation deteriorates the performance of FL from multiple perspectives. First, the low-end devices cannot even afford to conduct local training. Directly dropping them leads to inferior model accuracy as the unique data on them are underrepresented in the global model, especially when the amount of low-end devices is relatively large. On the other side, although other devices can afford to conduct local training, the learning process can be severely slowed down. This is for the reason that when the memory footprint exceeds a predefined threshold [41], it triggers the operating system to reclaim pages by moving some memory pages occupied by processes to the swap area on the memory or storage devices which leads to high training latency. Worse still, memory contention caused by the currently running background apps can exacerbate page reclaiming and further slow down the local training process. This deceleration can be amplified in FL as the slowest devices bottleneck the overall convergence process. Thus, the memory wall can simultaneously hurt the model performance and training efficiency in FL.

Prior Art. Memory optimization has been widely studied in on-server training. Gradient checkpointing [4, 27, 30], activation

compression [3, 43], and swapping [25, 49, 53] have been widely employed. However, they rely on architectures that are invalid on mobile devices, such as host-side memory for swapping. Though achieving memory reduction, directly applying them either leads to inferior model accuracy or results in prolonged training progress in FL. Recently, in order to surmount the resource constraints of the participating devices, partial training is proposed [1, 9, 24]. It first trains a lower-complexity submodel on the devices and then integrates the trained submodels into the full global model. However, these approaches adversely affect model performance as numerous filters/parameters must be discarded and the architecture of the model is severely compromised. *Thus, a new framework that can effectively guide the training process on memory-constrained devices while jointly guaranteeing the model accuracy and training efficiency is crucial for the deployment of FL in real-world cases.*

Challenges. Designing such a framework is not straightforward and faces the following critical challenges. First, memory constraints of the mobile devices simultaneously impact the model accuracy and training performance. Thus, how to jointly trade off the heterogeneity of memory capacity, computing capability and data distribution across different devices is the first critical challenge. Second, simply dropping the memory-constrained devices can severely hurt the model accuracy. Though directly applying the existing on-server memory-saving techniques, such as swapping and compression, can reduce the memory footprint, they can slow down the training progress or deteriorate the model performance at the same time. Thus, how to achieve memory-efficient training without compromising training efficiency is the second critical challenge. In addition, due to the resource contention caused by the background apps, the memory budget for the training process varies across different training rounds. Thus, how to conduct efficient training in a highly dynamic training environment is the third critical challenge.

Our Contribution. In this work, we propose *FedHybrid*, a new framework that efficiently conducts FL on memory-limited mobile devices through hybrid tensor management. It aims to strike a balance among the 1) memory reduction of the local training process, 2) accuracy of the global model and 3) training efficiency of the overall system. Specifically, *FedHybrid* consists of the following three core components. In each training round, the Memory-aware Client Selector first selects the participating devices by jointly evaluating their memory budget, and computing capability along with data diversity. After that, for the selected devices, the Heterogeneity-aware Graph Optimizer delicately analyzes the computational graph and generates an execution plan tailored to the features of the mobile platform, which employs a judiciously designed memory optimization strategy through hybrid tensor management in order to meet the memory constraint with minimum training latency. After receiving the execution plan, the Local Training Engine directs the local training process with a well-designed activation compression strategy and the recomputation technique to further boost runtime efficiency with minimal accuracy loss. Additionally, this engine incorporates a novel memory budget predictor in order to retrieve the safe memory budget in a dynamic training environment. To the best of our knowledge, *FedHybrid* is the ***first*** work that conducts memory-efficient FL on a highly heterogeneous and dynamic training environment.



(a) MobileNet-OpenImage (b) Bert-AGNEWS (c) Memory Usage

Figure 1: Performance impact of memory constraints on Oort and memory utilization for different models. (a) Accuracy of MobileNetV2 trained on OpenImage with a batch size of 32. (b) Accuracy of Bert trained on AGNEWS with a batch size of 8. (c) Memory utilization on mobile devices during training, measured with MNN[28]; MNV2 denotes MobileNetV2.

Specifically, we make the following key contributions:

• We propose *FedHybrid*, a memory-efficient federated learning framework, in which a hierarchical structure is designed to well balance memory reduction, model accuracy, and the overall training process.

• We design and implement the three core components, the Memory-aware Client Selector, the Heterogeneity-Aware Graph Optimizer, and the Local Training Engine to interact with each other and guide the whole training process.

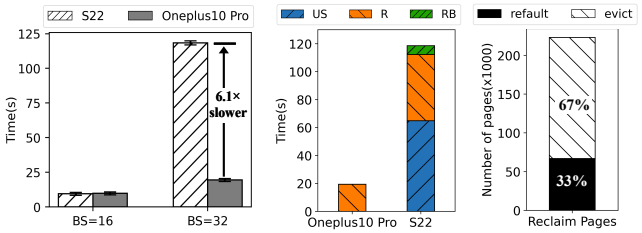
• To evaluate the effectiveness of *FedHybrid*, we conduct extensive experiments based on representative DNN models, datasets, and commodity mobile devices.

2 BACKGROUND AND MOTIVATIONS

2.1 The Memory Wall in FL

One pivotal question required to be explored first is: *how does the memory limitation impact a heterogeneous FL system?* In order to investigate the impact, we conduct the following experiments. Specifically, we establish a hybrid FL platform with both simulation and hardware testbed to conduct the following two training tasks: MobileNetV2 [50] on OpenImage [31] for image classification and Bert [34] on AGNEWS [58] for text classification. Leveraging the FedScale [32] benchmark to simulate the real-world environment, the system contains 6,582 clients for image classification and 2,040 clients for text classification, respectively. The hardware configuration including the computing capability and the memory capacity are emulated with the data from AI Benchmark [26], which provides statistical runtime information of the training process across different types of mobile devices. In specific, the memory distributions are as follows: 4GB (15%), 6GB (25%), 8GB (30%), 12GB (25%), and 16GB (5%). Moreover, off-the-shelf mobile devices, including S22 and Oneplus 10-Pro, are adopted to delve into the impact of memory limitation on the runtime.

Observation 1: Memory wall deteriorates the model performance. Figure 1 presents the training performance in three different scenarios including: 1) Oort- Practical, 2) Oort-Oracle, and 3) Random. Specifically, Oort [33] is a client selection methodology that jointly considers data and system heterogeneity. In this case, Oort-Practical directly applies Oort on the memory-constrained devices, whereas Oort-Oracle represents a theoretical baseline, which



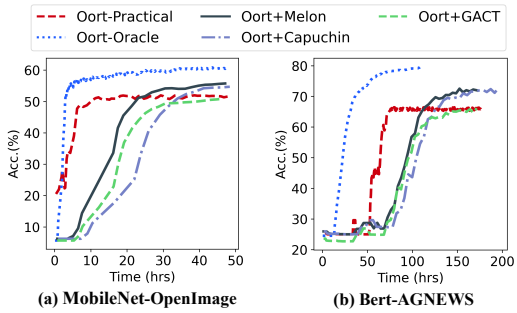
(a) Runtime comparasion (b) Runtime breakdown (c) Ratio of Pages

Figure 2: The analysis of local training runtime on mobile devices. We use MNN to conduct the local training without any background application. (a) Compare the runtimes of different devices under 16 and 32 batch sizes. (b) Breakdown of the training process's runtime status and average system memory usage during training with 32 batch sizes. The US represents the Uninterruptible Sleep status, R represents the Running status, and RB represents the Runnable status. (c) Distribution of evicted pages during the training process with 32 batch sizes in S22.

assumes that all the devices have sufficient memory to complete local training. For Random, it just randomly selects a certain number of clients to participate in each training round. It is worth noting that 200 clients are selected in each training round. Figure 1 shows that, compared with Oort Ideal, Oort-Practical shows a prominent accuracy decline, 10.8% for image classification and 14.2% for text classification. This is for the reason that, as shown in Figure 1(c), over 8GB of memory is required for MobileNetV2, and 10GB is required for Bert. In this case, Oort-Practical excludes over 40% and 70% of clients for MobileNet and Bert training. This exclusion not only reduces data diversity but also compromises the comprehensiveness and representativeness of the training data.

Observation 2: Memory wall impedes the training efficiency. Memory constraints not only affect training performance, but also significantly degrade overall training efficiency. We observe that the average duration for each training round with Oort-Practical extends 3.8× longer for MobileNetV2 and 2.4× longer for Bert compared with Oort-Oracle to attain the respective target accuracy. To investigate the root cause, we employ the MNN [28] framework to perform local training on commercial mobile devices, including Galaxy S22 (8GB) and OnePlus 10 Pro (12GB).

Figure 2(a) presents the local training completion time of one training round on the two different smartphones, utilizing different batch sizes. With a batch size of 16, the runtime for these devices is nearly identical, reflecting their comparable computation capability. However, a significant discrepancy is observed when the batch size is increased to 32. The training completion time of S22 significantly exceeds that of the OP10P, being over 6.1 times slower. Figure 2(b) shows the duration the CPU spends in different statuses within the local training procedure including: 1) uninterruptible sleep (UR), which refers to a state in which the process awaits a hardware resource and cannot be interrupted 2) active running (R) denotes that the training process is currently executing on the CPU, and 3) runnable (RB) which signifies readiness for execution pending CPU availability. We can find that, for the S22, 56% of CPU time is dedicated to uninterruptible sleep, 40% to active running, and 4% to a runnable state. The substantial time spent in uninterruptible sleep status, over 50% for the S22, is primarily attributed to



(a) MobileNet-OpenImage (b) Bert-AGNEWS

Figure 3: The performance of existing memory-saving techniques applied in FL with memory constraints.

I/O operations related to memory management. Concurrently, our assessment of average system memory usage indicates that it is at 74% for the OP10P and increases to 91% for the S22 as shown in Figure 2(b). Under conditions of high memory pressure, the system triggers page reclaim which identifies and reclaims inactive pages. The identified evicted pages are relocated to secondary storage, and if needed again ('page refault'), they would be reintroduced into main memory, causing substantial delays due to I/O requests. Notably, Figure 2(c) shows, S22 experiences 96,666 page refaults[1], with 33% necessitating re-access which prominently slows down the local training process. In addition to the substantial memory footprint leading to heightened memory pressure, memory contention from background applications can further exacerbate the runtime overhead.

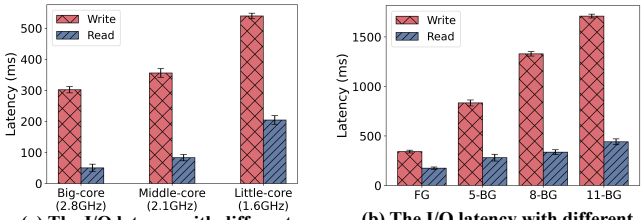
Summarized Takeaway: Memory constraints of the participating devices severely deteriorate the model performance and the training efficiency at the same time. Moreover, the apps concurrently running with the local training process can further exacerbate this situation. *Thus, a new FL system that effectively coordinates the training on memory-limited devices with dynamic resource contention is crucial for real-world deployment.*

3 EXPLORING EXISTING MEMORY SAVING TECHNIQUES IN FL

Another important question yet to be explored is: *whether the existing memory saving techniques are sufficient to break the memory wall in FL?* In this section, we investigate the following techniques that are widely adopted in on-server training.

•**Host-device memory swapping.** Swapping reduces memory usage during training by offloading activations or model parameters from GPU memory to external memory (e.g., CPU memory) in the cloud server. However, mobile devices predominantly utilize integrated memory for all processors, necessitating data swapping between main memory and storage disks. By applying Capuchin [49], a widely utilized swapping mechanism designed in on-server training, we observe a notable improvement in accuracy by 4.7% and 7.2% for image and text classification tasks, respectively, as depicted in Figure 3. This is for the reason that as the memory footprint is reduced, more low-end devices can be involved in the training process and the corresponding private local data can well benefit

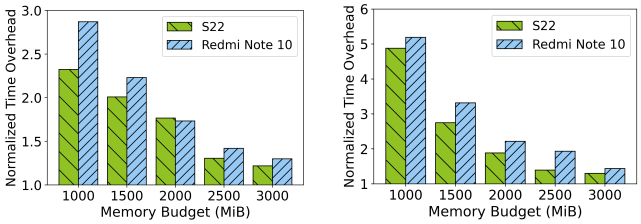
[1]We instrument the Android kernel source code (Linux kernel version 5.10.136) and use the adb to obtain information on memory allocations and the reclaim process of our evaluated smartphone.



(a) The I/O latency with different CPU core setups.

(b) The I/O latency with different background Apps usages.

Figure 4: The I/O latency of write and read for a 128 MB File in UFS 3.1: simulation of swapping In and Out a 128 MB tensor. (a) A higher frequency of the CPU core correlates with increased I/O performance. (b) Multi-Apps deteriorate the I/O performance due to contention in the UFS command queue.



(a) MobileNetV2

(b) Bert

Figure 5: Efficiency analysis of activation recomputation in FL with heterogeneous devices. All the experiments are conducted on Melon [54]. Comparison of different memory budgets versus training latency overhead in MobileNetV2 with a batch size of 32, and BERT with a batch size of 8, on the S22 and Note 10, respectively.

the global model. However, in the meantime, it extends the training duration required to achieve target accuracy by factors of 4.7× and 1.7×, significantly degrading training efficiency. This inefficiency is primarily due to the instability of device I/O for data swapping in mobile environments. As shown in Figure 4(a), the read/write latency is influenced by the CPU issuing the I/O command, with higher CPU core frequencies correlating with decreased I/O latency. This is because the CPU core initiating the read/write needs to run the UFS driver, which involves several tasks such as handling interrupts and managing queues. A higher frequency allows faster processing of these UFS-related I/O operations. Additionally, unlike NVMe, UFS storage in mobile devices has a single command queue, lacking internal concurrency capabilities. Current mobile devices frequently engage in background I/O activities, primarily due to the frequent reading and writing of cache files, which are first stored in main memory and then written back to flash storage [40]. As shown in Figure 4(b), such frequent I/O activities result in resource competition between swapping processes and background applications, pushing the hardware to its throughput limit and leading to high latency issues [22, 35, 56]. Moreover, in FL, the variance in I/O speeds across different devices [29, 39] can exacerbate these performance bottlenecks.

•**Activation compression.** Activation compression compresses the training activation during the forward pass and then decompresses the saved activation during backpropagation to calculate the gradient. However, this approach requires detailed, model architecture-dependent analysis to minimize compression error, restricting their applicability. In this case, we incorporate GACT [43], a model-universal framework for activation compression that adaptively modulates the compression ratio for each neural network layer. As illustrated in Figure 3, integrating GACT does not lead to enhanced model accuracy, and it simultaneously degrades the training efficiency. This accuracy decline results from the error introduced during the compression process. While, the extra training latency lies in its uniform compression of all activations for gradient computations during backpropagation as some activations are too large to be processed quickly on mobile devices

•**Activation recomputation.** Activation recomputation evicts activations during the forward pass and recomputes them as needed during the backward stage. Recent studies [14, 54] have used advanced recomputation methods to reduce the memory footprint for on-device training. Melon [54] is a representative recomputation-based approach, which combines recomputation and micro-batch while eliminating fragments to reduce the memory footprint. As illustrated in Figure 3, integrating Melon results in a notable increase in accuracy 5% and 8% higher compared to Oort-Practical for training MobileNet and Bert, as more mobile devices can be involved in the training process. However, such a method leads to significantly prolongs the convergence process by 3.5× and 1.5×, respectively. The primary reason for this slowdown is that through achieving memory reduction, the existing recomputation strategy in Melon involves computation-intensive operators that require repeated calculation and significantly extend the training time. This inefficiency is particularly pronounced when memory budgets are extremely limited. As illustrated in Figure 5 (a) and (b), empirical evidence demonstrates substantial time overheads: the S22 device experiences 2.3× and 4.8× increases for MobileNetV2 and BERT respectively, with a memory budget of 1000. These overheads are even more significant for lower-end devices such as the Redmi Note 10, reaching 2.9× and 5.2× respectively. Thus, while local training can be completed within the memory budget, the prolonged duration can severely extend each training round.

Summarized Takeaway: Though directly applying existing memory reduction techniques can achieve memory saving, it either degrades the model performance or deteriorates the training efficiency due to challenges posed by the unique architecture of mobile devices and an FL system. *Thus, a memory-efficient training framework specifically designed for a mobile-based FL system is urgently required.*

4 SYSTEM DESIGN

FedHybrid is designed to effectively deploy FL on memory limited mobile devices while jointly guaranteeing the model performance and training efficiency. In this section, we first give an overview of *FedHybrid* and then discuss each key component in detail.

4.1 Overview

Figure 6 depicts the architecture of *FedHybrid*. Following the standard schema of FL, *FedHybrid* employs a server/client architecture and consists of the following three core components: 1) Memory-aware Client Selector, 2) Heterogeneity-aware Graph Optimizer

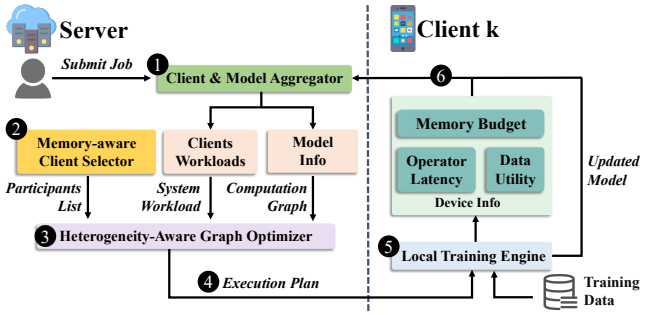


Figure 6: The workflow of *FedHybrid*.

hosted by the central server and the 3) Local Training Engine deployed on each participating device. These three core components coordinate with each other to direct the overall training procedure in order to strike a balance between memory reduction, model performance, and training efficiency.

Workflow. The overall workflow can be divided into the following main stages: ① *Initialization.* At the initialization stage, the central server first initializes the global model to be trained and oversees the status (e.g., hardware configuration) of the participating devices. ② *Participant Selection*: the Memory-aware Client Selector selects the participating clients for the current training round through jointly evaluating their computing capability, memory budget and data utility in order to well balance the model accuracy and training efficiency. ③ *Execution Graph Optimization*: For each selected client, Heterogeneity-aware Graph Optimizer judiciously analyzes the computation graph and configures the right memory saving strategy for each tensor in order to minimize the latency of the local training process given a specific memory budget. ④ *Model&Plan Transmission*: The server then broadcasts the model parameters together with the execution plan to the corresponding client. ⑤ *On-device Local Training*: After receiving the execution plan, the client employs the Memory-Aware Efficient Training Engine, crafted for executing detailed training plans. It optimizes runtime efficiency by employing a carefully orchestrated channel-wise mixed compression alongside activation recomputation. At the same time, it monitors the system status and predicts the memory budget for the next training round. ⑥ *Model aggregation & Client status uploading*: After completing local training, the client sends the updated model, memory budget, operators latency, and data utility back to the server. The central server then aggregates the local models and updates the global model. This process iterates until the model converges.

4.2 Memory-aware Client Selection

Existing client selection schemes simply assume that all the devices have sufficient memory to well support the local learning process and fail to consider the impact on both model accuracy and training efficiency according to the discussion in Section 2.2. In this section, we first introduce the utility function and then discuss how *FedHybrid* selects the clients through jointly taking into account the memory budget, computing capability and data utility.

Memory Utility. The memory utility of a device is defined as follows:

$$\text{MemStat(i)} = \left(\frac{M_i}{M_G}\right)^{\mathbf{1}(M_G > M_i)} \tag{1}$$

where M_i is the memory budget of client i, representing the available memory for the training process as specified by the application or os and determined by the client's memory budget predictor (see Section 4.4.2). This predictor estimates the client's memory budget over time, through taking into account the resource contention caused by the concurrently running applications in a dynamic training environment. M_G is the memory requirement to train the model in sufficient memory conditions. $\mathbf{1}(x)$ is an indicator function that takes value 1 if x is true and 0 otherwise. This metric evaluates whether a device has sufficient memory ($M_i \geq M_G$) to train the model without any system overhead, yielding a memory utility of 1, indicating full capability. In contrast, a memory utility less than 1 ($M_i < M_G$) represents that a device lacks sufficient memory, highlighting potential limitations in its contribution to the training process due to memory constraints. The lower the value is, the higher the probability the system will trigger page reclaim during the local training process and thus lead to high training latency.

Data Utility. To tackle the data heterogeneity of the clients, we measure the statistical utility of client i by:

$$\text{Stat(i)} = \sqrt{\frac{1}{|B_i|}\sum_{k\in B_i}\text{Loss}(k)^2} \tag{2}$$

where the $|B_i|$ is the local training batch size of clients and the $\text{Loss}(k)$ indicates the training loss of data sample. A higher statistical utility indicates greater data importance due to the inclusion of challenging or underrepresented samples, which are essential for improving model robustness and generalization [29, 33, 37, 51].

Computing Utility. The computing utility of client i is defined as:

$$\text{CompStat(i)} = \frac{1}{\sum_{o\in \text{CM}_i}^{M} t_o} \tag{3}$$

where CM_i represents the set of unique operators, M in total , employed within the training model, and t_o is the average kernel computation time of operator o on device i collected from the local training engine. A higher CompStat(i) suggests client i has a strong computation capability, leading to shorter computation times for NN operators. Conversely, a lower CompStat(i) indicates a weaker computational capability, resulting in longer computation times for executing the same NN operations.

We unify the above utility models and generate the comprehensive utility function of client i as follows:

$$\text{Util}(i) = \sqrt{\frac{1}{|B_i|}\sum_{k\in B_i}\text{Loss}(k)^2} \times \left(\frac{M_i}{M_G}\right)^{\mathbf{1}(M_i > M_G)} \times \left(\frac{1}{\sum_{o\in \text{CM}_i}^{M} t_o}\right) \tag{4}$$

Through this fine-grained design of utility, even if a client lacks sufficient memory for local training, our memory-aware selection criterion might still choose that client if its statistical utility outweighs its system utility. This approach involves a larger, more diverse set of clients, significantly enhancing data diversity during training. Our selection strategy is grounded in theoretical insights: sampling high-loss samples reduces variance similarly to gradient norm-based sampling, resulting in faster convergence compared to naive random sampling [6]. To maintain system efficiency and fairness while enhancing the data diversity of the participant pool, we design an exploration-exploitation strategy based on the Multi-Armed Bandit with Upper Confidence Bound (MAB-UCB) model

Algorithm 1: Plan Generation Mechanism.

Input: *origin_comp_seqs*, *memory_budget*, *memory_pool*
Output: optimized plan

```
Function compute(comp_op):
    for t in comp_op.input_tensors do
        if t.state == COMPRESSED then
            t.address ← alloc(t.size);
        else if t.status == EVICTED then
            recover(t);
    for t in comp_op.output_tensors do
        if t is not allocated then
            t.address ← alloc(t.size);
Function alloc(comp_op):
    while pool_size + size ≥ memory_budget do
        tensor ← MaxMPS();
        if tensor.MPS_compute > tensor.MPS_compress then
            evict(tensor);
        else
            compress(tensor);
Main:
    for op_i in origin_comp_seqs do
        compute(op_i);
```

[5] for client selection. In this model, each device is treated as an "arm" of the bandit, and the set of selected arms is referred to as the super arm N. We first calculate the utilities of all devices based on Equation 4 and sort them from high to low. Then, we select ϵ fraction of total selection clients from the high-utility participants. Additionally, we sample an $1-\epsilon$ fraction of participants to explore potential participants that have not been selected before. These unexplored participants lack data utility, but we can prioritize those with rich system resources, as clients regularly check in and send memory and compute information as a "heartbeat" in existing FL systems [2, 33, 48]. This check-in only occurs when the phone is charging and connected to an unmetered network such as WiFi. All information is encrypted [10, 57] to ensure user privacy and anonymized.

4.3 Heterogeneity-aware Graph Optimizer

For a selected client, the Heterogeneity-aware Graph Optimizer is designed to conduct efficient training on devices with a given memory budget. According to the discussion in Section 3, directly applying the existing memory-saving techniques either deteriorates the model accuracy or slows down the overall training process. Such observations lead us to a pivotal inquiry: *Is there an approach that can effectively reduce the memory footprint without compromising the model accuracy and training efficiency for FL?*

4.3.1 Opportunity. To further investigate this issue, we propose a new metric, MPS (Memory reduced Per Second) to quantify the reward (RW) of the application of memory-saving techniques—namely recomputation, compression, and swapping-on targeted tensor d

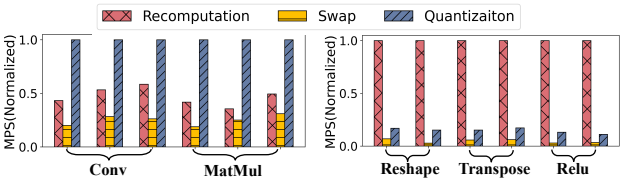


Figure 7: Comparison of MPS with different tensors from MobileNetV2.

which is defined as follows:

$$\mathrm{RW}(d) = SM(d) \times FLT(d),\ \mathrm{MPS}(d) = \frac{\mathrm{RW}(d)}{PC(d) + RC(d)} \quad (5)$$

where SM (Saved Memory) represents the amount of memory conserved by using a particular technique, while FLT (Freed Lifetime) indicates the duration for which the tensor remains unused in memory, thus contributing to overall memory efficiency. The effectiveness of MPS lies in its ability to balance memory savings with computational overhead. PC (Purge Cost) and RC (Regain Cost) are critical in this metric, as they account for the time required to purge tensors and regenerate them when needed. By incorporating these factors, MPS provides a comprehensive tensor measurement that reflects not just the memory conserved but also the efficiency of the process in terms of computational time. A high MPS value signifies efficient memory savings with minimal purge or regeneration time costs, indicating a well-balanced trade-off between memory conservation and computational efficiency in mobile devices. Conversely, a low MPS value points to insufficient memory savings or high time costs, suggesting a need for improved strategies or computational optimizations tailored for mobile hardware constraints.

Figure 7 represents the MPS of different memory-saving techniques on various tensors. Specifically, we select the 20 tensors in MobileNetV2 with the highest reward as case study, grouping them by their operator type and using different shapes to reduce duplication, resulting in 12 unique tensors. These high-reward tensors are selected because they have larger sizes and longer freed lifetimes, meaning discarding these tensors can free up more space during training, thus improving overall memory efficiency.

The following key findings direct us to the design of a graph optimizer. 1) Low MPS for swapping. The swapping technique consistently demonstrates the lowest MPS values across all tensors compared to recomputation and compression. This is for the reason that, although the embedded Universal Flash Storage (eUFS) 3.1 offers throughput of 2,100 MB/s read and 1,200 MB/s write, the main problem with swapping in a mobile environment is the unstealable I/O contention from background apps, which is due to the restriction of a single command queue and the lack of internal concurrency. 2) Various MPS in Recomputation and Compression. The utility of recomputation and compression highly depends on the specific tensor, with compression showing superior performance for complex operations like convolution and matrix multiplication. This superiority is attributed to the inefficiencies of mobile convolution operators, suggesting that compression is more adept at managing computational demands. Thus, *it is essential to select the optimal technique (either recomputation or compression) for specific tensors within the computation graph.*

Impact of Layout Transformation. In addition to the regeneration efficiency issues related to operator types, we found that data

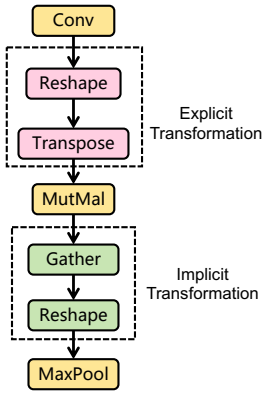


Figure 8: Examples of layout transformation in DNNs.

Table 1: Latency and transformation breakdown across various models. 'Layout Tran.' refers to the latency associated with explicitly transforming the tensor's layout. 'Other Comp' indicates the latency attributed to the remaining operators. FT refers to finetine.

Model	Lat. breakdown (%)	
	Layout Tran.	Other Comp.
ResNet34	19.7	81.3
MobileNetV2	14.2	85.8
ResNet50	22.8	77.2
Bert	57.8	42.2
Swin (FT*)	69.6	30.4

layout transformations can introduce significant overhead during tensor recomputation in mobile training. Figure 8 illustrates a computation graph where the model explicitly includes Reshape and Transpose operations between the Conv and MutMal layers, and implicitly adds Gather and Reshape operations between the MutMal and MaxPool layers. These transformations ensure compatibility between different operators within a DNN's computation graph, as certain operators, especially those optimized for high execution performance on CPUs or GPUs, require specific data memory layouts such as NCHW or NC4HW4. Layout transformations consume substantial memory bandwidth and disrupt the locality of subsequent operations. While individual transformations are relatively fast, their cumulative effect leads to significant performance overhead. The tensors generated by these transformations are not stored and have very short lifetimes. Therefore, each time a recomputed operator requires a layout transformation, it must be recalculated, further adding to the recomputation overhead. As shown in Table 1, which details the latency and transformation breakdown across various models using recomputation in MNN, a significant portion of time is spent on layout transformations, particularly in Transformer-based models. Previous recomputation strategies [49, 54] primarily attribute the cost of tensor recomputation to the computational effort of the tensor's operator alone. However, they overlook the additional costs associated with layout transformations. This oversight can lead to considerable performance overheads in various DNNs. To the best of our knowledge, no existing strategies account for the influence of layout transformations during tensor recomputation. This gap can result in suboptimal graph optimization policies and inefficiencies in the recomputation process. Thus, *it is crucial to consider the layout transformation cost when using recomputation to balance memory reduction and training efficiency.*

Guided by the above design principles, the Heterogeneity-aware graph optimizer aims to minimize the overall execution cost while ensuring that peak memory usage remains below the specified budget by incorporating a **hybrid** tensor-saving strategy. This approach is detailed in Algorithm 1. When determining which tensor needs to be discarded, we exploit the MPS (Equation 5) to estimate the regeneration cost. To accurately measure the recomputation cost for tensor d generated by operator j, we propose new metric $RC_{compute}(d)$ for computing $MPS_{compute}$:

$$RC_{compute}(d) = OpTime(j) + \psi \sum_{i=0}^{Dep(d)} OpTime(i) \tag{6}$$

where $OpTime(j)$ is the time cost of recomputing operator j, $\psi = 2$ when including CPU/GPU transformation, otherwise $\psi = 1$. $Dep(d)$ is the Depth-first search (DFS) depth from the output of operator j to the final operator z that takes tensor d as input, including any intermediate layout transformation operators. When generating the computation graph, we attribute a property to tensors requiring layout transformations that record the original operator index. This is achieved through layout check functions in MNN, which inspect the tensor's layout. In addition, we compare the original computation graph with the framework-generated graph to identify these layout transformation operators. The final transformed tensors retain the original operator's index, enabling the DFS to trace back to the original tensor's operator. This ensures all layout transformation operators are included in the recomputation cost calculation.

The graph optimizer starts by taking the entire computation graph as input (lines 18-19). When memory is exhausted, it initiates a memory reclamation procedure (lines 11-16). During this process, the optimizer evaluates each tensor's Memory Pressure Score (MPS) to decide whether to evict or compress the tensor with the highest MPS (line 12). Each tensor is assigned a recomputation MPS ($MPS_{compute}$) and a compression MPS ($MPS_{compress}$). If $MPS_{compute}$ exceeds $MPS_{compress}$, the tensor is evicted; otherwise, it is compressed (lines 13-16). This process repeats until memory usage is within the budget (lines 1-8). When a compressed tensor required by the current operator is needed, it is decompressed, and memory is allocated accordingly (lines 3-4). If a tensor is missing from memory, the optimizer recomputes it from its original tensors, potentially recomputing source tensors if they are also unavailable (lines 5-6). This mechanism ensures that logical dependencies are maintained during operations while adhering to memory constraints.

Plan generation for large scale of clients. As discussed in Section 3, generating an execution plan frequently is time-consuming for mobile devices, even when using heuristic-based algorithms. To address this issue, *FedHybrid* offloads this procedure to a central server, thereby reducing computational burden for the participating devices. In FL, generating an execution plan for each participant is challenging and time-consuming due to varying available memory and computing capabilities. To efficiently manage large-scale clients, we capitalize on the heterogeneous nature of the client pool. We employ clustering techniques, such as k-means, based on clients' memory and computing characteristics. Within each cluster, we select the client with minimum capability to generate an execution plan, ensuring feasibility for all cluster members. This approach is optimal as the resulting plan is universally applicable within the cluster without compromising performance on more capable devices. To further enhance efficiency, we implement a cache table for rapid plan generation and retrieval. This table stores previously generated plans based on different budgets and client device information, enabling quick adaptation to various client scenarios. This

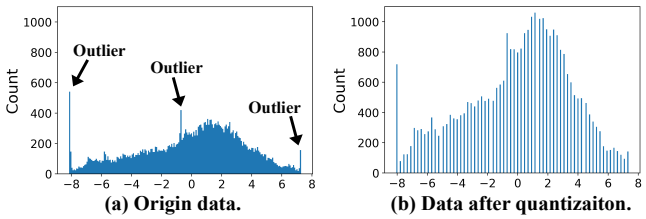


Figure 9: The data distribution of before and after quantization with outlier.

strategy ensures efficient use of resources while maintaining the quality of the FL process across a wide range of client capabilities.

4.4 Local Training Engine

The Local Training Engine is designed to conduct the local training according to the received execution plan generated by the Heterogeneity-aware Graph Optimizer. The Local Training Engine then sends the memory budget, as predicted by the memory budget predictor, along with the operator's cost time, which is the average time collected from the last batch of all training rounds. All information is encrypted to ensure user privacy and anonymized. However, efficiently performing the execution plan faces the following critical challenges: 1) According to the execution plan, compression or recomputation is selected to apply on a specific tensor in order to achieve memory reduction while guaranteeing training efficiency. However, we find that existing compression techniques can introduce compression errors at the same time as they overlook the outlier value in tensor, making quantization ineffective. *Thus, how to conduct effective compression without minimum accuracy loss is the first critical challenge.* 2) The resource contention caused by concurrently running apps can dynamically change the available system memory. *Thus, how to precisely determine the safe memory budget is another critical challenge.* In order to conduct efficient local training, we propose a new compression technique called Channel-wise Mix Compression and memory budget predictor.

4.4.1 Channel-wise Mix Compression. Current activation compression methods [3, 11, 43] use lossy compression to reduce memory via quantization, but they often neglect the impact of tensor outliers, leading to significant errors. Outliers, play a crucial role in DNNs, with their values in CNNs and Transformers [20] exceeding standard values by over 81× and 347×. This discrepancy causes substantial deviation in post-quantization data from the original, as seen in Figure 9, resulting in a loss of information. This problem is exacerbated in FL, where data compression across multiple clients amplifies these errors. Additionally, the significance of outliers in determining layer outputs and gradient updates [23, 42] highlights the necessity for improved compression techniques that consider outlier preservation to maintain data integrity and model accuracy.

To balance accuracy and compression without compromising quantization effectiveness, we design adaptive channel-wise mix compression, as shown in Figure 10. This approach exploits the observation that activation values are mostly concentrated [20], allowing precise quantization of normal values. Outliers are identified using the empirical 3σ rule [55], acknowledging their sparsity and channel-wise clustering rather than treating them individually. This insight guides us to classify channels into outlier-rich (salient) and normal, applying a mix-quantization strategy that significantly

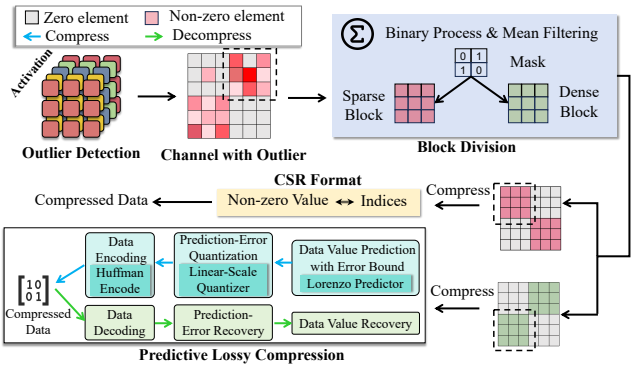


Figure 10: Workflow of Channel-wise Mix Compression.

compresses data while retaining crucial model performance information. Normal channels, characterized by concentrated values, undergo quantization to achieve compression, ensuring an efficient balance between memory usage and data integrity.

For outlier-rich channels, which typically have many zero values, we cannot directly use existing lossy compression methods [3, 12] since we observe that zero values among the channels can cause significant errors during compression. Specifically, these errors arise because zero values are not preserved accurately after decompression. This inaccuracy can result in high variance in the gradients, preventing the training process from converging. To address this problem, we design a nuanced block-wise compression method that preserves outliers while achieving high accuracy. We first categorize the non-zero values of a channel feature map ($h \times w$) into dense and sparse blocks by starting with a binary mask matrix to distinguish non-zero values. This step addresses the local sparsity of DNN activations. To improve operation speed, we use average pooling to process the feature map instead of element-wise counting. Mean filtering through average pooling (window size $n \times n$) helps identify sparse blocks by comparing them against a sparsity threshold τ. Sparse blocks are then efficiently compressed using the Compressed Sparse Row (CSR) format, balancing data integrity with compression effectiveness.

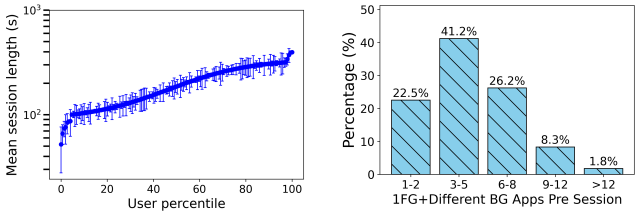
In dense blocks, we implement predictive compression, leveraging the local data relationship for more efficient compression within an error-bound, distinct from the conventional linear scale approach. This process encompasses three stages to effectively compress the data. Initially, for each element x in block B_n, the Lorenzo predictor is employed to estimate its value $\hat{x}$ using neighboring elements, formalized as:

$$P_n = \{p^* | p^* = f_l(x_k^{SR}), x_k \in B_n\}, \tag{7}$$

where P_n represents the predicted value for block B_n, x_k^{SR} denotes the surrounding values of x_k, and f_l is the Lorenzo predictor, which is defined by:

$$\sum_{\substack{0 \le k_1,\ldots,k_m \le n \\ k_1,\ldots,k_d \neq 0}} \left(\prod_{j=1}^{m} (-1)^{k_j+1} \binom{n}{k_j} \right) \cdot x_{c_1-k_1,\ldots,c_d-k_d}, \tag{8}$$

ensuring that the sum of coefficients equals one. The predictive model ranges from simple 1-dimensional first-order predictions to more complex multidimensional constructs. Subsequently, we



(a) Session Length (b) Apps Usage Per Session

Figure 11: Analysis of Users' Long-Term App Usage in the Carat dataset [47]. (a) The mean and standard deviation of the duration of individual interaction sessions across all users, presented on a logarithmic scale. (b) The number of applications accessed during each interaction session.

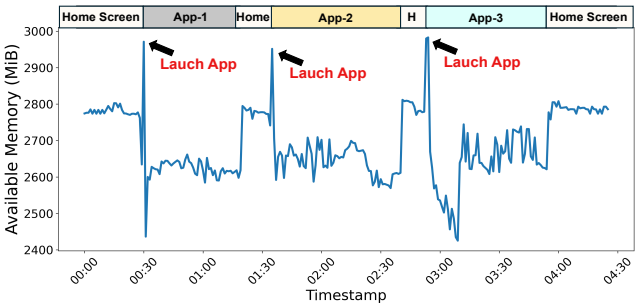


Figure 12: Analysis of Memory Usage Patterns. We select a sample user usage trace from the Carat dataset [47] and emulate it on the S22 mobile device using the Monkey testing tool [17]. Caching background app re-launch, with the top bar representing the trace process.

assess the prediction error φ_n between the original and predicted values, confined within the error bound ϵ, to determine:

$$\varphi_n = \{\delta | \delta = p^* - x_k, \delta < \epsilon, p^* \in P_n, x_k \in B_n\}, \quad (9)$$

where δ is the prediction error. Values within ϵ are quantized to conserve memory, while outliers are preserved intact, ensuring compression accuracy as predictions utilize decompressed rather than original values. To enhance the compression ratio, especially for outliers requiring additional bits, a Huffman tree is constructed from the quantization array for encoding, coupled with lossless compression for the Lorenzo predictor's coefficients, facilitating decompression.

Decompression involves Huffman decoding, followed by the reconstruction of floating-point values from quantization codes and their prediction. This predictive compression not only optimizes memory use but also captures and leverages data relationships beyond simple scaling, offering a nuanced approach to data compression in dense NN blocks.

4.4.2 Memory Budget Predictor. Memory budget of the training process can be impacted by the resource contention caused by the concurrently running apps during user interaction. User interaction with mobile apps often follows discernible patterns over time [38, 52]. To make the investigation, we analyze long-term app usage data from 150 users over 30 days using the Carat dataset [38, 47]. Our analysis reveals consistent patterns in both session length and app usage. As illustrated in Figure 11(a), over 70% of users have an average session duration of 153 seconds, with 73% falling between

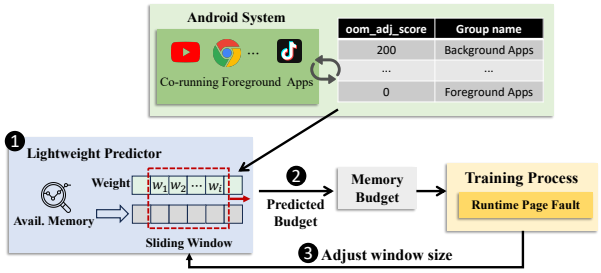


Figure 13: The workflow of Memory Budget Predictor.

126 to 412 seconds, suggesting relatively fixed and brief interaction times. Furthermore, Figure 11(b) demonstrates that users typically operate with one foreground app and 3-5 background apps during sessions. These observations reveal a consistent memory usage pattern where a significant portion of device memory is typically occupied by a fixed number of active apps, corresponding to certain usage patterns. This consistency in app usage behavior provides a foundation for predicting memory budget in dynamic mobile environments.

To investigate how these patterns translate to actual memory usage, we emulated user behavior on the S22 using the Monkey tool [17]. Figure 12 illustrates the resulting memory usage during app launches and transitions. In Android's ecosystem, apps don't fully exit when moved to the background; instead, they remain cached for quick reactivation. This design leads to distinctive memory usage patterns. App launches cause brief memory spikes followed by stabilization, while switching between cached apps results in sharp spikes with quick recovery, as shown in Figure 12 (a). These spikes indicate that memory demand is highly dynamic during app switching and launches, but once a foreground app is active, memory consumption stabilizes. This insight reveals that short-term memory spikes during app launches and switches significantly increase the variance of available memory. However, these spikes are transient and do not reflect the stable memory usage during active app use. Therefore, these fluctuations should be filtered out when determining real-time memory budgets.

Based on the above-observed memory usage pattern, we propose the lightweight memory budget predictor specifically designed to account for the fluctuating workloads of the participating devices in FL. Figure 13 shows the architecture and workflow of the memory budget predictor. It not only estimates available memory under dynamic usage conditions but also coordinates with the graph optimizer to determine the optimal moments for plan regeneration, balancing overall efficiency with regeneration overhead. To effectively predict the system's available memory, we first introduce an adjusted available memory metric M_{safe}, which considers the impact of different swap types on page reclamation. In practice, we observe that swap type significantly influences the timing of page reclamation. For example, compression-based swaps like ZRAM [15] use a portion of DRAM as swap space, reducing available DRAM and triggering earlier memory reclamation. To address this, we define adjusted available DRAM memory: $M_{safe} = M_{Avail} - \alpha \cdot \text{Watermark}_{high}$. M_{Avail} is the current available DRAM memory from */proc/meminfo*, and Watermark_{high} is a threshold for stopping background page reclamation, typically a fixed value proportional to total DRAM pages. The parameter α

Table 2: Android devices for the testbed experiments.

	Year	Mem	SoC	OS
Samsung Galaxy S23 Ultra	2023	8G	Snapdragon 8 Gen 2	Android 13
Samsung Galaxy S22	2022	8G	Snapdragon 8 Gen 1	Android 12
OnePlus 10 Pro	2022	12G	Snapdragon 8 Gen 1	Android 12
Google Pixel 6	2021	8G	Google Tensor	Android 12
Xiaomi Redmi Note 10	2021	6G	Snapdragon 678	Android 11
Honor V30 Pro	2019	8G	Kirin 990	HarmonyOS 3
Realme GT Neo 2	2021	8G	Snapdragon 870	Android 11

represents swap activity type: $\alpha = 1$ for disk-based swap and $\alpha = 2$ for ZRAM.

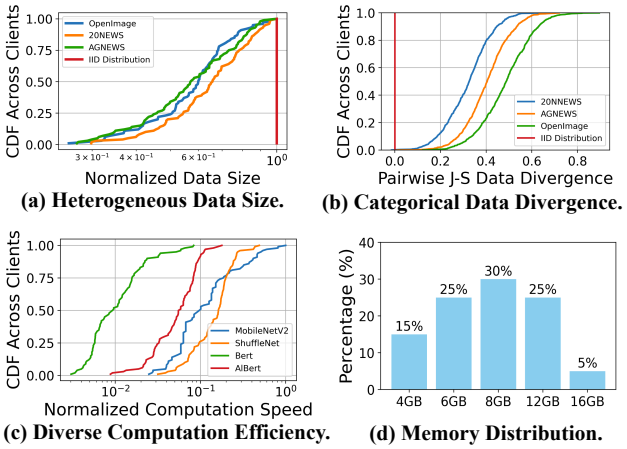
To mitigate the short-term fluctuations caused by app switching and launching, we employ a weighted moving average method to capture the most recent and important memory usage patterns. For each sampling period T_{sample}, we calculate the average adjusted available DRAM memory M_{safe}. The window size, W, is initiated as the duration of one training epoch, ensuring that memory predictions align with the training process. The sliding size is T_{slide}. The weights in the moving average are calculated based on the importance of the applications running during each window. We use the *oom_adj_score* [19], which prioritizes processes for termination in the Android system, with higher scores indicating lower importance. We calculate the weights w_i at time i by: $w_i = \sum_{j=1}^{B} \frac{\text{Max\_Score}}{\text{oom\_adj\_socre}_j}$. B is the total application's activities [2] collected by *ActivityManager* [18]. Foreground applications and critical background applications (such as music playback, downloads, and synchronization tasks) are given higher weights, while less critical background applications receive normal weights. For example, when an app is running in the foreground, its *oom_adj_score* is 0 (indicating highest priority), but when the user switches to the home screen or another app, the *oom_adj_score* for the previous app increases, reducing its importance in memory calculations. Additionally, we found that processes associated with the home launcher [21] (e.g., the home screen) have significantly higher *oom_adj_score* than foreground apps. As a result, the transition from the home screen to an app launch is assigned lower weights, making the impact of this transition less significant in the memory prediction. This weighting scheme inherently assigns lower weights to short-term fluctuations, as memory spikes are often caused by temporary background activities with higher *oom_adj_score*. This approach helps monitor short-term trends in DRAM memory usage and allows us to adjust memory budgets effectively. The prediction process can be described as follows:

$$M^t_{Pred} = \frac{\sum_{i=t-W+1}^{t} w_i \cdot M^i_{safe}}{\sum_{i=t-W+1}^{t} w_i} \tag{10}$$

where M^t_{Pred} represents the predicted weighted moving average of the adjusted available DRAM memory M_{safe} at time t, M^i_{safe} represents the adjusted available DRAM memory at the time i, and w_i are the importance weights of the task applications at time i.

Plan Regeneration Rule. To balance runtime efficiency with the dynamics of available memory, we set the following criteria for clients to regenerate the plan from the server: 1). The current round's training process total page faults count exceeds the previous round's average by more than TP_1. 2). The training process is terminated by the Low Memory Killer (LMK) more than TP_2 times. If plan regeneration is needed, the predictor first adjusts the sliding window size with factor WS_{adj}as: $W_{new} = W_{current} \times WS_{adj}$. Then, a new budget is generated.

[2]We only calculate processes with oom_adj_score ≥ 0. The maximum score is 1,000. Foreground applications have an oom_adj_score of 0, which is set to 1 when calculating the weight.



(a) Heterogeneous Data Size. (b) Categorical Data Divergence. (c) Diverse Computation Efficiency. (d) Memory Distribution.

Figure 14: Experiment setup. (a) per-client quantity of samples, (b) deviation of per-client categorical distributions using Jensen-Shannon divergence, (c) distribution of the computation efficiency across clients, (d) distribution of memory distribution across clients.

5 EVALUATION

5.1 Implementation

We implemented *FedHybrid* using a combination of C++ and Python. For the local training engine, which includes channel-wise mix compression and the memory budget predictor, we build on top of MNN (version 2.3.0) [28], adding 5.8K lines of code (LOC). For the memory-aware client selection and heterogeneity-aware graph optimizer, we extend FedScale [32] with an additional 1.5K LOC. Additionally, we implement three baselines (Capuchin, GACT, and HeteroFL) based on prior literature. Since some of these models were designed for servers rather than mobile devices, we adapte and re-implemented them on top of MNN for a fair comparison.

5.2 Experiment Setup

Setup. To accurately simulate memory impact on training across diverse devices and emulate system heterogeneity, we combine simulation with physical testing. While we use AI-Benchmark [26] to model device heterogeneity, similar to [33, 37, 51], we enhance accuracy by conducting real on-device training using MNN on a testbed of devices with 7 hardware configurations (Table 2). We then extrapolate these results to a wider range of SoCs by scaling measured training times based on AI-Benchmark's theoretical computation time ratios, applying this only to devices with matching memory configurations. This hybrid approach provides a more realistic assessment of training performance across varied mobile devices. The computation distribution across clients is shown in Figure 14(c). To simulate realistic dynamic memory conditions, we vary background applications based on real-world distributions [39] and

Table 3: Summary of improvements on time to accuracy. We take the highest accuracy that random selection can achieve as the target, which is moderate due to the high task complexity and lightweight models.

Dataset	Model	Random+FedProx		Oort+FedProx		HeteroFL+FedProx		*FedHybrid*+FedProx		Random+Yogi		Oort+Yogi		HeteroFL+Yogi		*FedHybrid*+Yogi	
		Speedup	Acc	Speedup	Acc	Speedup	Acc	Speedup	Acc	Speedup	Acc	Speedup	Acc	Speedup	Acc	Speedup	Acc
OpenImg	MobileNetV2	1×	51.56	2.89×	52.75	-	38.65	**15.55×**	**68.09**	1×	53.42	2.02×	54.25	-	38.76	**11.34×**	**70.12**
	ShuffleNet	1×	51.03	2.19×	52.09	-	38.25	**9.86×**	**66.03**	1×	52.01	1.86×	53.95	-	37.35	**8.43×**	**68.83**
20NEWS	Bert	1×	31.71	1.14×	36.72	0.80×	31.92	**9.05×**	**70.42**	1×	33.52	1.08×	35.22	1.12×	33.89	**8.57×**	**72.68**
	Albert	1×	26.53	1.05 ×	31.22	-	15.36	**7.85×**	**63.39**	1×	28.71	1.03×	32.56	-	20.21	**7.03×**	**65.12**
AGNEWS	Bert	1×	66.47	1.21×	67.21	-	54.75	**3.70×**	**86.97**	1×	67.89	1.22×	69.21	-	53.34	**4.22×**	**88.34**
	Albert	1×	65.63	1.14 ×	66.84	-	50.23	**4.52×**	**82.69**	1×	66.74	1.08×	67.96	-	51.25	**4.13×**	**84.24**

Table 4: Summary of improvements. Take the highest accuracy that random+prox can achieve as the target.

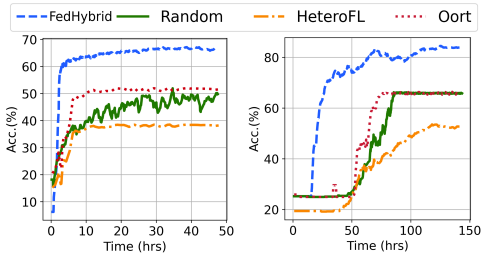
Dataset	Model	***FedHybrid*+FedProx**		Oort+Melon		Oort+Capuchin		Oort+GACT	
		Speedup	**Acc.**	Speedup	Acc.	Speedup	Acc.	Speedup	Acc
OpenImg	MobileNetV2	**15.55×**	**68.09**	0.83×	57.62	0.53×	57.40	0.62×	51.84
	ShuffleNet	**9.86×**	**66.03**	0.94×	54.59	0.57×	53.95	0.61×	51.63
20NEWS	Bert	**9.05×**	**70.42**	0.68×	44.53	0.48×	42.74	0.51×	32.68
	Albert	**7.85×**	**63.39**	0.64×	42.72	0.52×	40.45	0.56×	30.62
AGNEWS	Bert	**3.70×**	**86.97**	0.77×	76.55	0.67×	76.23	0.59×	67.04
	Albert	**4.52×**	**82.69**	0.86×	72.68	0.76×	71.89	0.58×	66.39

simulate foreground app usage using the Carat dataset [47]. The devices are configured with standardized ZRAM [15], using 1 GB of memory for swap space with lz4 compression [15]. Additionally, a 3 GB swap partition is allocated on NAND storage. The memory capacity distribution is shown in Figure 14(d). *To emulate the data heterogeneity*, we categorize training datasets into different levels of non-identical distribution (Non-IID) with Dirichlet distribution [29] $\alpha = 0.1$, and employ Jensen-Shannon divergence [29, 37] to measuring sample counts and categorical distribution variations, as illustrated in Figure 14(a)(b). We also build a simulator based on the Server/Client architecture utilizing a GPU server equipped with 8× NVIDIA H800 GPUs as the established FL practices [33, 36, 37, 51].

Datasets and Models. We evaluate several well-known DNN and Transformer models in computer vision (CV) and natural language processing (NLP). For CV tasks, we train the OpenImg dataset [31], which includes 1.5 million images across 600 categories, on MobileNetV2 [50] and ShuffleNet [59], optimized for resource-limited mobile devices. Each round selects 200 out of 6,582 clients. For NLP tasks, we utilize the AGNEWS [58] and 20NEWS [7] datasets, with over 127K and 18K items respectively, training on Albert [34] and Bert [8] for text classification. The number of client selection varies by dataset: for 20NEWS, 20 out of 100 clients are chosen each round, while for AGNEWS, 200 out of 2,040 clients are selected.

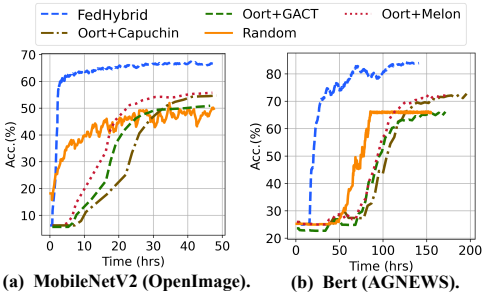
Hyper-parameters. We set $\epsilon = 0.9, TP_1 = 2, TP_2 = 3$ and $WS_{adj} = 0.9$. For CV tasks, the batch size is 64, with 500 rounds and a local training epoch of 10 at a learning rate of 0.045. For NLP tasks, the batch size is 8, with 400 rounds and a local training iteration of 1 at a learning rate of 0.1. The max sequence len gth is 128 for both 20NEWS and AGNEWS.

Baselines. We compare *FedHybrid* with the following baselines. Addressing FL heterogeneity: (1) Random [45] selects clients randomly each round. (2) Oort [33] optimizes device selection by combining statistical utility with device training time. (3) HeteroFL [9] adjusts convolutional layer channels to fit diverse memory capacities. Enhancing on-device memory efficiency: (4) Melon [54] uses



(a) MobileNetV2 (OpenImage). (b) Bert (AGNEWS).

Figure 15: Time-to-accuracy comparison with FL baselines.



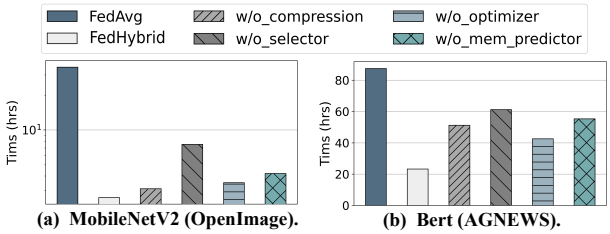
(a) MobileNetV2 (OpenImage). (b) Bert (AGNEWS).

Figure 16: Time-to accuracy-comparison of FL integrated memory saving techniques.

micro-batching and recomputation to minimize memory usage. (5) Capuchin [49] employs swap and recomputation strategies for memory reduction. (6) GACT [43] dynamically compresses activations, adjusting compression ratios per network layer.

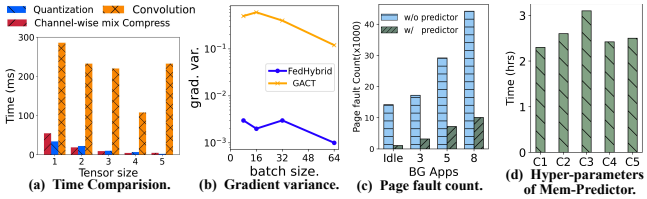
5.3 End-to-end Performance

***FedHybrid* significantly outperforms existing FL baselines across all domains and tasks.** Table 3 presents the end-to-end performance of *FedHybrid* in comparison to several FL baselines. Figure 15 reports the accuracy achieved by *FedHybrid* and FL baselines across different models. We observe that: (a) Superior Accuracy: *FedHybrid*+FedProx achieves 15.3% and 33.7% higher accuracy than Oort+FedProx for training MobileNetV2 and Bert, respectively. This improvement is primarily due to *FedHybrid*'s ability to enable participation from more memory-constrained clients, which other methods fail to incorporate effectively. Both Oort and Random selection methods exclude memory-constrained clients, leading to reduced sample diversity. HeteroFL, while attempting to accommodate memory limitations, excessively prunes submodel parameters to fit constrained devices, resulting in performance degradation. This underscores HeteroFL's reliance on a certain number of full-model capable clients for optimal performance. (b) Faster



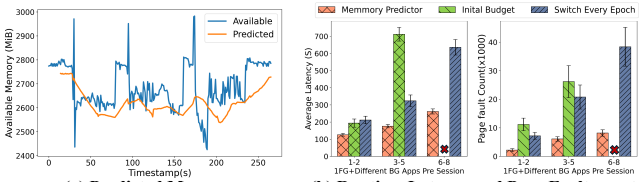
(a) MobileNetV2 (OpenImage). (b) Bert (AGNEWS).

Figure 17: Significance of Key Designs of *FedHybrid*.



(a) Time Comparision. (b) Gradient variance. (c) Page fault count. (d) Hyper-parameters of Mem-Predictor.

Figure 18: Ablation study of Local training engine. (a) The tensor sizes from id = 1 to 5 are: 1) 64,112,112 2) 64,56,56 3) 128,28,28 4) 256,14,14 5) 512,7,7. (d) Hyper-parameter of (TP_1, TP_2): C1(2,3), C2(1.5,3), C3(3,3), C4(2,4), C5(2,5).



(a) Predicted Memory (b) Runtime Latency and Page Fault count

Figure 19: Evaluation of the Memory Budget Predictor on real user traces from the Carat [47] dataset. We conducted 5 training iterations on 30 example user app usage traces on the S22. (a) The predicted memory budget for an example trace from the predictor. (b) The mean training runtime latency and page fault count among the 30 example traces.

Convergence: When training MobileNetV2, *FedHybrid* achieves the target accuracy 5.38× and 15.55× faster than Oort+FedProx and Random+FedProx, respectively. This acceleration is driven by two main factors: *FedHybrid* optimizes the computation graph based on the system workload of each client, and the local engine minimizes the overhead associated with memory optimization operations. Oort and Random selection methods are slowed down by memory-constrained clients in each round, extending overall training time. The speedup is even more pronounced when training Bert, as its large model size necessitates significant client data, which hinders the overall training progress in alternative methods.

***FedHybrid* Significantly Improves Time-to-Accuracy Compared to SOTA FL Frameworks with Memory Saving Optimization.** Figure 16 reports the accuracy achieved by *FedHybrid* and FL baselines combined with memory saving methods across different models. As shown in Table 4, we compare *FedHybrid*'s performance against Random and Oort (each integrated with Melon, Capuchin, and GACT). All methods surpass Random, highlighting the "memory wall" that limits participant involvement and impairs model performance. However, *FedHybrid* outperforms Oort integrated with Melon, Capuchin, and GACT in terms of accuracy. Specifically, when training Bert on the 20NEWS dataset, *FedHybrid* achieves 25.89%, 27.68%, and 37.74% higher accuracy, respectively. This is due to the simplistic application of memory reduction strategies in FL training, which overlooks client heterogeneity. Additionally, Oort's selection bias towards clients with shorter training completion times, while sidelining those with less memory but valuable data, further differentiates *FedHybrid*. Therefore, Oort's time-prioritized selection may suboptimally choose clients, as it might exclude those whose completion times are prolonged due to memory limitations. These factors, combined with the extra computational demands of these strategies, hinder their convergence speed. Consequently, they require 13.3×, 18.8×, and 17.7× longer training times for Bert on the 20NEWS dataset compared to *FedHybrid*.

5.4 Effectiveness of Each Component

Impact of Memory-aware Client Selector. Replacing the memory-aware client selector with a random selector significantly slows convergence speed, as shown in Figure 17. In both tasks depicted, *FedHybrid* without the selector converges 3× and 2.6× slower than *FedHybrid*. This demonstrates that leveraging more abundant client data enhances overall performance, indicating that considering both the statistical and system utility of clients is crucial for achieving optimal accuracy.

Impact of Heterogeneity-aware Graph Optimizer. Removing the global optimizer and requiring each client to generate its own execution plan using Melon's plan generation results in a 1.3× and 1.8× slower convergence speed in two tasks, as shown in Figure 17. This slowdown is primarily due to the time-consuming nature of execution plan generation (as shown in Figure 20(c), 9× slower than server's generation), which hinders some clients from completing their training tasks and uploading the model on time. Moreover, existing plan generation methods often overlook the cost of layout transformation, resulting in less effective plans.

Impact of Channel-wise Mix Compression. Disabling compression of the optimizer significantly slows convergence speed, as shown in Figure 17, with speeds 1.1× and 2.2× slower across both tasks due to the recomputation overhead for each client. A comparison of representative tensors reveals that the time required for quantization and *FedHybrid*'s compression is comparable and much lower than that for convolution, highlighting the efficiency gains from compression, as shown in Figure 18(a). Figure 18(b) shows a decrease in gradient variance of restored tensors after decompression, leading to minimal accuracy degradation.

Impact of Memory Budget Predictor. Replacing the Memory Budget Predictor with direct memory measurement significantly slows convergence speed, as shown in Figure 17, with speeds 1.6× and 2.4× slower in both tasks due to inaccurate memory measurements affected by dynamic background applications on mobile devices. This inaccuracy impacts client selection and execution plan generation. Figure 18(c) demonstrates that using the Memory Budget Predictor decreases page faults, reducing runtime overhead. Additionally, Figure 18(d) shows the total training time for MobileNetV2 with different hyperparameters TP_1 and TP_2. Increasing TP_1 reduces the frequency of plan regeneration triggers, leading to more time spent on disk swapping due to memory contention from inaccurate local memory measurements. To further validate

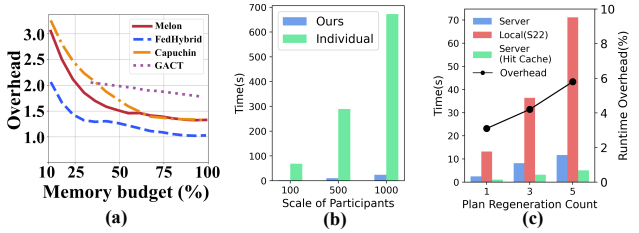


Figure 20: Overhead Analysis for *FedHybrid*. (a) Execution time under various budget ratio for MobileNetV2 in S22. (b) Overhead of graph optimizer with different scale of clients. (c) The runtime cost of plan regeneration with various counts in S22 (connected 100Mbps WIFI).

the predictor's effectiveness, we use 30 real-world user traces from the Carat [47] dataset with S22 for 5 rounds of on-device training on MobileNetV2. Figure 19(a) shows the predicted memory according to an example trace. The predictor provides a stable memory budget range for dynamic memory usage, with available memory variance lower than 172.2MB compared to the 576.8MB in practice. This demonstrates the predictor's ability to mitigate fluctuations caused by dynamic usage, applying a lower window weight to fluctuations. Figure 19(b) reports the average runtime overhead among 30 traces with different training budgets. We compare the memory budget predictor with two baselines: (1) using initial available memory as a fixed budget, and (2) changing the budget every training round. The results show that without the memory predictor, training latency can increase up to 6.9× when using initial available memory, and up to 2.2× overhead when changing the budget every round, particularly when there are many background apps. Thus, the memory predictor can effectively handles the dynamicity of diverse client environments and provides more stable and efficient training performance

5.5 Overhead Analysis

Client's Overhead. The client's overhead consists of two main parts. First, the Memory Budget Predictor runs as a background service in the Android system, monitoring available memory asynchronously to prevent page access suspension. On the S22 device, it averages 3-5% CPU usage, as measured by Perfetto[16]. Second, dynamic changes in the memory budget may require plan regeneration, incurring additional communication costs. Figure 20(c) shows that even with five regeneration requests per ten local training rounds, the total communication time remains below 6% of the overall training time, indicating minimal additional overhead.

Energy Consumption. We measure energy usage using the Monsoon Power Monitor [46], testing MobileNetV2 and SqueezeNet with a batch size of 16 on an OnePlus10 Pro device under varying memory budgets (Figure 21). This Ideal scenario represents local training with sufficient memory capacity to complete normal training without any page fault overhead. *FedHybrid* reduces energy consumption by 1.1×-1.8× compared to baselines. Compared to an ideal baseline, *FedHybrid* only increases energy use by an average of 14.2% in different memory budgets. This moderate increase in energy consumption is attributed to our method's ability to significantly reduce the time overhead associated with a hybrid-tensor saving strategy. By substantially shortening training completion

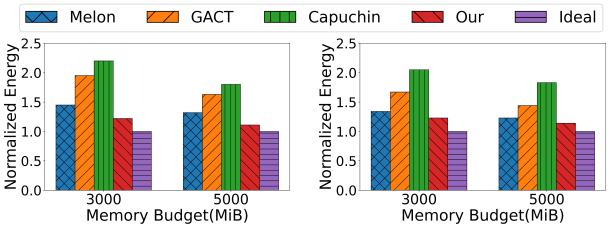


Figure 21: The energy consumption of *FedHybrid*. This Ideal scenario represents local training with sufficient memory capacity to complete normal training without any page fault overhead. All results are normalized to this Ideal baseline.

times, *FedHybrid* effectively minimizes overall energy consumption. In contrast, other methods like Capuchin, while having lower power consumption for I/O operations during swapping compared to computation, ultimately consume more energy due to prolonged training time.

Server's Overhead. The server's overhead primarily arises from generating plans for a large number of clients. Figure 20(b) illustrates the server's plan generation time across different participant scales. The results highlight the importance of our clustering and cache-based optimization, demonstrating that it can be up to 647× faster than generating plans individually.

6 RELATED WORKS

Memory-Efficient Training on Mobile Devices: Several approaches [14, 54] have recently been proposed to reduce the memory footprint for on-device training. For instance, Melon [14] reduces the memory footprint of on-device training through recomputation and a lifetime-aware memory pool. Sage [14] adopts multiple memory management and saving techniques, including micro batch, operator fusion, and gradient checkpointing, to surmount memory constraints on mobile devices. However, directly employing these memory-saving techniques designed for a single device not only degrades the model performance but also deteriorates the training efficiency due to the following characteristics of the FL system. Micro-batch techniques can degrade model performance in FL due to limited local data and non-IID distributions across clients, leading to noisy gradient estimates and poor generalization. Recomputation strategies substantially increase training latency in each FL round, particularly problematic for heterogeneous client devices with varying computational capabilities. This prolongs overall training time and limits FL's scalability in large-scale deployments. Furthermore, the memory management approaches of Melon and Sage are not well-suited for the dynamic memory environments typical in FL. Clients have varying and fluctuating memory availability due to concurrent applications, making static or arbitrary memory budgets ineffective. While these methods have mechanisms to adapt to changing memory budgets, frequent adjustments introduce additional computational overhead, increasing energy consumption and training duration—particularly problematic for battery-powered devices. To address these challenges, our approach combines recomputation with a novel Channel-wise Mix Compression technique and incorporates a dynamic memory budget predictor, effectively balancing memory efficiency and training speed across diverse devices in FL environments.

System Optimization for FL: Mobile devices often face resource constraints that lead to performance bottlenecks in FL from a system perspective. Several studies [33, 37, 51] have aimed to tackle system heterogeneity in FL, primarily focusing on optimizing device compute speeds and communication bandwidths by client selection and coordination. Oort [33] designs a guide-based client selection metric to choose participants for FL based on devices' resource conditions or training data distributions. However, this type of approach tends to tend to overlook the critical issue of available memory, which is essential for a device to participate in training. Without sufficient memory, a device is excluded from the training process, thereby limiting the diversity and effectiveness of the FL system. Recently, Partial training [1, 9, 24] has been introduced to mitigate computational resource limitations by training lower-complexity submodels on devices. For example, HeteroFL [9] trains a lower complexity submodel according to device capability and integrates it into the full global model. However, this approach reduces memory usage at the expense of model performance, as it requires discarding numerous filters and parameters, compromising the model architecture. In contrast, our work directly addresses memory constraints while preserving model integrity and performance. By dynamically selecting memory-constrained clients and optimizing their memory usage, our approach enables more devices to participate effectively in FL.

7 CONCLUSION

In this paper, we present *FedHybrid*, a memory-efficient federated learning framework tailored for devices with limited memory. By incorporating a Memory-aware Client Selector, a Heterogeneity-aware Graph Optimizer, and a Local Training Engine, *FedHybrid* significantly enhances training efficiency. Our experiments demonstrate that *FedHybrid* boosts model accuracy by up to 39.1% and cuts training time by up to 15.5× compared to conventional methods.

ACKNOWLEDGMENTS

We sincerely appreciate the anonymous shepherd and reviewers for their valuable comments. This research was supported in part by the MYRG-GRG2023-00211-IOTSC-UMDF and SRG2022-00010-IOTSC grants from the University of Macau.

REFERENCES

[1] Samiul Alam, Luyang Liu, Ming Yan, and Mi Zhang. 2022. Fedrolex: Model-heterogeneous federated learning with rolling sub-model extraction. *Advances in Neural Information Processing Systems* 35 (2022), 29677–29690.

[2] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloe Kiddon, Jakub Konečnỳ, Stefano Mazzocchi, Brendan McMahan, et al. 2019. Towards federated learning at scale: System design. *Proceedings of machine learning and systems* 1 (2019), 374–388.

[3] Jianfei Chen, Lianmin Zheng, Zhewei Yao, Dequan Wang, Ion Stoica, Michael Mahoney, and Joseph Gonzalez. 2021. Actnn: Reducing training memory footprint via 2-bit activation compressed training. In *Proceedings of ICML*. PMLR, 1803–1813.

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

[5] Wei Chen, Yajun Wang, and Yang Yuan. 2013. Combinatorial multi-armed bandit: General framework and applications. In *Proceedings of ICML*. PMLR, 151–159.

[6] Yae Jee Cho, Jianyu Wang, and Gauri Joshi. 2020. Client selection in federated learning: Convergence analysis and power-of-choice selection strategies. *arXiv preprint arXiv:2010.01243* (2020).

[7] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. 2016. Convolutional neural networks on graphs with fast localized spectral filtering. *Advances in neural information processing systems* 29 (2016).

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[9] Enmao Diao, Jie Ding, and Vahid Tarokh. 2020. Heterofl: Computation and communication efficient federated learning for heterogeneous clients. *arXiv preprint arXiv:2010.01264* (2020).

[10] Cynthia Dwork. 2008. Differential privacy: A survey of results. In *International conference on theory and applications of models of computation*. Springer, 1–19.

[11] R David Evans and Tor Aamodt. 2021. Ac-gc: Lossy activation compression with guaranteed convergence. *Advances in Neural Information Processing Systems* 34 (2021), 27434–27448.

[12] R David Evans, Lufei Liu, and Tor M Aamodt. 2020. Jpeg-act: accelerating deep learning via transform-based lossy compression. In *Proceedings of ACM/IEEE ISCA*. IEEE, 860–873.

[13] Gary Sims. 2023. How much RAM does your Android phone really need in 2023? https://www.androidauthority.com/how-much-ram-do-i-need-phone-3086661/. Accessed: 2023.12.

[14] In Gim and JeongGil Ko. 2022. Memory-efficient DNN training on mobile devices. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*. 464–476.

[15] Google. 2023. samsung-galaxy-s21-ram-plus-update. https://developer.android.com/topic/performance/memory-management. Accessed: 2023.12.

[16] Google. 2023. System profiling, app tracing and trace analysis. https://perfetto.dev/. Accessed: 2023.12.

[17] Google. 2023. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/other-testing-tools/monkey. Accessed: 2023.12.

[18] Google. 2024. ActivityManager. https://developer.android.com/reference/android/app/ActivityManager.AppTask. Accessed: 2023.12.

[19] Google. 2024. oom-adj-score. https://developer.android.com/topic/performance/memory-management#low-memory_killer. Accessed: 2023.12.

[20] Cong Guo, Jiaming Tang, Weiming Hu, Jingwen Leng, Chen Zhang, Fan Yang, Yunxin Liu, Minyi Guo, and Yuhao Zhu. 2023. Olive: Accelerating large language models via hardware-friendly outlier-victim pair quantization. In *Proceedings of ACM/IEEE ISCA*. 1–15.

[21] Dianne Hackborn. 2013. ProcessList.java. https://android.googlesource.com/platform/frameworks/base/+/6285a32/services/java/com/android/server/am/ProcessList.java. Accessed: 2023.12.

[22] Kyuhwa Han and Dongkun Shin. 2020. Command queue-aware host I/O stack for mobile flash storage. *Journal of systems architecture* 109 (2020), 101758.

[23] Jung Hwan Heo, Jeonghoon Kim, Beomseok Kwon, Byeongwook Kim, Se Jung Kwon, and Dongsoo Lee. 2023. Rethinking channel dimensions to isolate outliers for low-bit weight quantization of large language models. *arXiv preprint arXiv:2309.15531* (2023).

[24] Samuel Horvath, Stefanos Laskaridis, Mario Almeida, Ilias Leontiadis, Stylianos Venieris, and Nicholas Lane. 2021. Fjord: Fair and accurate federated learning under heterogeneous targets with ordered dropout. *Advances in Neural Information Processing Systems* 34 (2021), 12876–12889.

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

[26] Andrey Ignatov, Radu Timofte, Andrei Kulik, Seungsoo Yang, Ke Wang, Felix Baum, Max Wu, Lirong Xu, and Luc Van Gool. 2023. AI Benchmark: All About Deep Learning on Smartphones. https://ai-benchmark.com/ranking.html. Accessed: 2023.12.

[27] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. 2020. Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization. In *Proceedings of Machine Learning and Systems 2020*. 497–511.

[28] Xiaotang Jiang, Huan Wang, Yiliu Chen, Ziqi Wu, Lichuan Wang, Bin Zou, Yafeng Yang, Zongyang Cui, Yu Cai, Tianhang Yu, Chengfei Lv, and Zhihua Wu. 2020. MNN: A Universal and Efficient Inference Engine. In *Proceedings of MLSys*.

[29] Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. 2021. Advances and open problems in federated learning. *Foundations and Trends® in Machine Learning* 14, 1–2 (2021), 1–210.

[30] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. 2020. Dynamic tensor rematerialization. *arXiv preprint arXiv:2006.09616* (2020).

[31] Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Alexander Kolesnikov, et al. 2020. The open images dataset v4: Unified image classification, object detection, and visual relationship detection at scale. *International Journal of Computer Vision* 128, 7 (2020), 1956–1981.

[32] Fan Lai, Yinwei Dai, Sanjay Singapuram, Jiachen Liu, Xiangfeng Zhu, Harsha Madhyastha, and Mosharaf Chowdhury. 2022. Fedscale: Benchmarking model and system performance of federated learning at scale. In *Proceedings of ACM ICML*. PMLR, 11814–11827.

[33] Fan Lai, Xiangfeng Zhu, Harsha V Madhyastha, and Mosharaf Chowdhury. 2021. Oort: Efficient federated learning via guided participant selection. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*. 19–35.

[34] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2019. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942* (2019).

[35] Gyeongyong Lee, Jaewook Kwak, Joonyong Jeong, Daeyong Lee, Moonseok Jang, Jungwook Choi, and Yong Ho Song. 2021. Internal Task-Aware Command Scheduling to Improve Read Performance of Embedded Flash Storage Systems. *IEEE Access* 9 (2021), 71638–71650.

[36] Ang Li, Jingwei Sun, Pengcheng Li, Yu Pu, Hai Li, and Yiran Chen. 2021. Hermes: an efficient federated learning framework for heterogeneous mobile clients. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*. 420–437.

[37] Chenning Li, Xiao Zeng, Mi Zhang, and Zhichao Cao. 2022. PyramidFL: A fine-grained client selection framework for efficient federated learning. In *Proceedings of ACM MobiCom*. 158–171.

[38] Tong Li, Yali Fan, Yong Li, Sasu Tarkoma, and Pan Hui. 2021. Understanding the long-term evolution of mobile app usage. *IEEE Transactions on Mobile Computing* 22, 2 (2021), 1213–1230.

[39] Yu Liang, Jinheng Li, Rachata Ausavarungnirun, Riwei Pan, Liang Shi, Tei-Wei Kuo, and Chun Jason Xue. 2020. Acclaim: Adaptive memory reclaim to improve user experience in android systems. In *Proceedings of USENIX ATC*. 897–910.

[40] Yu Liang, Riwei Pan, Tianyu Ren, Yufei Cui, Rachata Ausavarungnirun, Xianzhang Chen, Changlong Li, Tei-Wei Kuo, and Chun Jason Xue. 2022. {CacheSifter}: Sifting Cache Files for Boosted Mobile Performance and Lifetime. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. 445–459.

[41] Geunsik Lim, Donghyun Kang, MyungJoo Ham, and Young Ik Eom. 2023. SWAM: Revisiting Swap and OOMK for Improving Application Responsiveness on Mobile Devices. *arXiv preprint arXiv:2306.08345* (2023).

[42] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Xingyu Dang, and Song Han. 2023. Awq: Activation-aware weight quantization for llm compression and acceleration. *arXiv preprint arXiv:2306.00978* (2023).

[43] Xiaoxuan Liu, Lianmin Zheng, Dequan Wang, Yukuo Cen, Weize Chen, Xu Han, Jianfei Chen, Zhiyuan Liu, Jie Tang, Joey Gonzalez, et al. 2022. GACT: Activation compressed training for generic network architectures. In *Proceedings of ACM ICML*. PMLR, 14139–14152.

[44] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. 2017. Communication-efficient learning of deep networks from decentralized data. In *Proceedings of AISTATS*. PMLR, 1273–1282.

[45] Brendan McMahan and Daniel Ramage. 2017. Federated learning: Collaborative machine learning without centralized training data. *Google Research Blog* 3 (2017).

[46] Inc Monsoon Solutions. 2023. High voltage power moniter. https://www.msoon.com/. Accessed: 2023.12.

[47] Adam J Oliner, Anand P Iyer, Ion Stoica, Eemil Lagerspetz, and Sasu Tarkoma. 2013. Carat: Collaborative energy diagnosis for mobile devices. In *Proceedings of the 11th ACM conference on embedded networked sensor systems*. 1–14.

[48] Matthias Paulik, Matt Seigel, Henry Mason, Dominic Telaar, Joris Kluivers, Rogier van Dalen, Chi Wai Lau, Luke Carlson, Filip Granqvist, Chris Vandevelde, et al. 2021. Federated evaluation and tuning for on-device personalization: System design & applications. *arXiv preprint arXiv:2102.08503* (2021).

[49] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. 2020. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of ACM ASPLOS*. 891–905.

[50] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of IEEE CVPR*. 4510–4520.

[51] Jaemin Shin, Yuanchun Li, Yunxin Liu, and Sung-Ju Lee. 2022. FedBalancer: data and pace control for efficient federated learning on heterogeneous clients. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*. 436–449.

[52] Zhen Tu, Runtong Li, Yong Li, Gang Wang, Di Wu, Pan Hui, Li Su, and Depeng Jin. 2018. Your apps give you away: distinguishing mobile users by their app usage fingerprints. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 2, 3 (2018), 1–23.

[53] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: Dynamic GPU memory management for training deep neural networks. In *Proceedings of ACM PPoPP*. 41–53.

[54] Qipeng Wang, Mengwei Xu, Chao Jin, Xinran Dong, Jinliang Yuan, Xin Jin, Gang Huang, Yunxin Liu, and Xuanzhe Liu. 2022. Melon: Breaking the memory wall for resource-efficient on-device machine learning. In *Proceedings of ACM MobiSys*. 450–463.

[55] Wikipedia contributors. Year the page was last edited. *68–95–99.7 rule*. https://en.wikipedia.org/wiki/68%E2%80%9395%E2%80%9399.7_rule Accessed: Access date.

[56] Zhenliang Xue, Yixin Song, Zeyu Mi, Le Chen, Yubin Xia, and Haibo Chen. 2024. PowerInfer-2: Fast Large Language Model Inference on a Smartphone. *arXiv preprint arXiv:2406.06282* (2024).

[57] Chengliang Zhang, Suyi Li, Junzhe Xia, Wei Wang, Feng Yan, and Yang Liu. 2020. {BatchCrypt}: Efficient homomorphic encryption for {Cross-Silo} federated learning. In *2020 USENIX annual technical conference (USENIX ATC 20)*. 493–506.

[58] Xiang Zhang, Junbo Zhao, and Yann LeCun. 2015. Character-level convolutional networks for text classification. *Advances in neural information processing systems* 28 (2015).

[59] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2018. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of IEEE CVPR*. 6848–6856.