FedCaSe: Enhancing Federated Learning with Heterogeneity-aware Caching and Scheduling

Redwan Ibne Seraj Khan
Virginia Tech
Virginia, USA
redwan@vt.edu

Arnab K. Paul
BITS Pilani, KK Birla Goa Campus
Goa, India
arnabp@goa.bits-pilani.ac.in

Yue Cheng
University of Virginia
Virginia, USA
mrz7dp@virginia.edu

Xun (Steve) Jian
Virgina Tech
Virginia, USA
xunj@vt.edu

Ali R. Butt
Virgina Tech
Virginia, USA
butta@cs.vt.edu

Abstract

Federated learning (FL) has emerged as a new paradigm of machine learning (ML) with the goal of collaborative learning on the vast pool of private data available across distributed edge devices. The focus of most existing works in FL systems has been on addressing the challenges of computation and communication heterogeneity inherent in training with edge devices. However, the crucial impact of I/O and the role of limited on-device storage has not been explored fully in FL context. Without policies to exploit the on-device storage for placement of client data samples, and schedule clients based on I/O benefits, FL training can lead to inefficiencies, such as increased training time and impacted accuracy convergence.

In this paper, we propose FedCaSe, a framework for efficiently caching client samples in-situ on limited on-device storage and scheduling client participation. FedCaSe boosts the I/O performance by exploiting a unique characteristic—the *experience*, i.e., relative impact on overall performance, of data samples and clients. FedCaSe utilizes this information in adaptive caching policies for sample placement inside the limited memory of edge clients. The framework also exploits the experience information to orchestrate the future selection of clients. Our experiments with representative workloads and policies show that compared to the state of the art, FedCaSe improves the training time by 2.06× for accuracy convergence at the scale of thousands of clients.

SoCC '24, November 20–22, 2024, Redmond, WA, USA

ACM ISBN 979-8-4007-1286-9/24/11
https://doi.org/10.1145/3698038.3698559

CCS Concepts

• **Computing methodologies** → *Artificial intelligence*; **Planning and scheduling**; **Mobile agents**; *Machine learning*; • **Information systems** → **Distributed storage**; **Cloud based storage**.

Keywords

Federated Learning, Caching, Sampling, Scheduling

ACM Reference Format:
Redwan Ibne Seraj Khan, Arnab K. Paul, Yue Cheng, Xun (Steve) Jian, and Ali R. Butt. 2024. FedCaSe: Enhancing Federated Learning with Heterogeneity-aware Caching and Scheduling. In *ACM Symposium on Cloud Computing (SoCC '24), November 20–22, 2024, Redmond, WA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3698038.3698559

1 Introduction

Modern machine learning (ML) is no longer constrained to running on well-endowed data center clusters alone, rather is being increasingly done closer to the sources of data such as edge devices [20]. As mobile devices become more prominent, the potential to train powerful models with end-user data has garnered a lot of interest [6]. This new paradigm of ML in a distributed collaborative nature, known as federated learning (FL), had a reported global market value of $110.8 million in 2021, and is expected to grow at 10.7% annually [2]. While promising, FL poses numerous challenges, which make it different from traditional ML. First, many unreliable devices that can drop out any moment can end up participating in training. Second, the end-user devices are extremely heterogeneous in terms of training data, computing power, communication capabilities, etc., making selection and efficient cooperation between them challenging.

To address the challenges associated with data heterogeneity and system heterogeneity, researchers and practitioners have put forward a number of techniques [8, 10, 11, 28, 34,

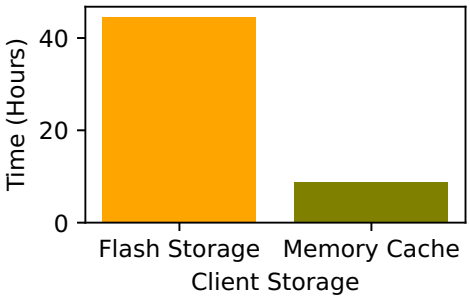


Figure 1: Training time comparison between clients using memory cache and flash storage.

38, 41–43] for client scheduling and client data sampling to improve the accuracy of the global model and reduce the training time. However, one crucial aspect of system heterogeneity–the memory capacity of client devices, has remained mostly unexplored. While a recent paper [14] examined the impact of limited on-device storage in FL and proposed policies for on-device data selection, it does not fully consider the case of efficiently handling heterogeneous memory cache and flash storage available across devices when I/O is a bottleneck.

To understand the impact that memory and flash device can have on FL training time, we perform two sets of experiments using ResNet-18 [16] model on the FEMNIST [16] dataset. In one, we allow multiple clients to train by fetching all samples from the memory cache. In the other, we allow the clients to fetch samples only from the flash device. Figure 1 shows that clients fetching from the flash storage took 5.1× longer to complete the training consisting of 150 rounds. This observation underscores that efficiently exploiting the limited memory cache on client devices can improve performance when training on I/O-intensive workloads.

One way to exploit the limited device memory is through caching client data samples. However, adapting the caching policies across a large number of distributed, heterogeneous client devices is non-trivial as it requires rethinking existing FL client scheduling and data sampling methods, especially with I/O performance as the focus. In particular, there is a need to orchestrate three connected components—(1) client scheduling, (2) client data sampling, and (3) client data sample caching—in such a way that can improve the training time and accuracy convergence of FL workloads.

In this paper, we design FedCaSe—a novel solution aimed at improving FL accuracy and training time in scenarios involving a large number of diverse client devices with heterogeneous memory and data. At the core, FedCaSe builds atop three strategies. First, FedCaSe uses a new sample selection strategy for samples that have demonstrated the most significant potential to improve model accuracy, i.e., experienced samples, and caching them within client memory for efficient, repeated training. Second, FedCaSe identifies experienced clients based on their record of improving model accuracy and reducing wall clock training time, which is gathered through training on the selected influential samples in the previous step. Third, FedCaSe schedules experienced clients with experienced samples in their cache for multiple rounds to ensure robust model training, which reduces the overall training time.

Specifically, this paper makes the following contributions.

- We characterize the potential of clients (i.e., client experience) having heterogeneous memory to improve accuracy and reduce wall clock time for the FL model.
- We introduce an experience-based score (i.e., sample experience) for data sampling of clients having heterogeneous memory to identify the sample's potential in increasing accuracy and improving read hit ratio (RHR) in limited heterogeneous memory cache of clients.
- We design a novel reverse-optimization technique (RO) to adaptively drive the scheduling of experienced clients in future rounds to improve FL performance.
- We present the design and implementation of FedCaSe–a new FL framework that leverages the experience of millions of clients and their samples to improve overall performance atop heterogeneous memory devices.
- We incorporate FedCaSe in a widely-used FL framework [27] and compare against a series of baselines and advanced scheduling, sampling, and caching methods. Results on a testbed of up to 2800 clients show that compared to the state of the art, FedCaSe improves the experienced client participation up to 29.1×, improves the global read hit ratio (RHR) by up to 81.7× (locally up to 318.58×) given the heterogeneous limited memory cache of clients, and thus ensures accuracy improvement rate up to 2.06× faster based on wall clock time, up to 1.4× faster based on number of rounds while keeping the round duration up to 2.4× less. FedCaSe is open-source and publicly available at

  https://github.com/R-I-S-Khan/FedCaSe/.

2 Background & Motivation

This section explores the key concepts of Distributed Deep Learning and Federated Learning, providing essential context for the entire paper. Furthermore, we perform an exploratory analysis on the potential of harnessing both client and sample experiences to improve the training performance by effectively mitigating I/O bottlenecks.

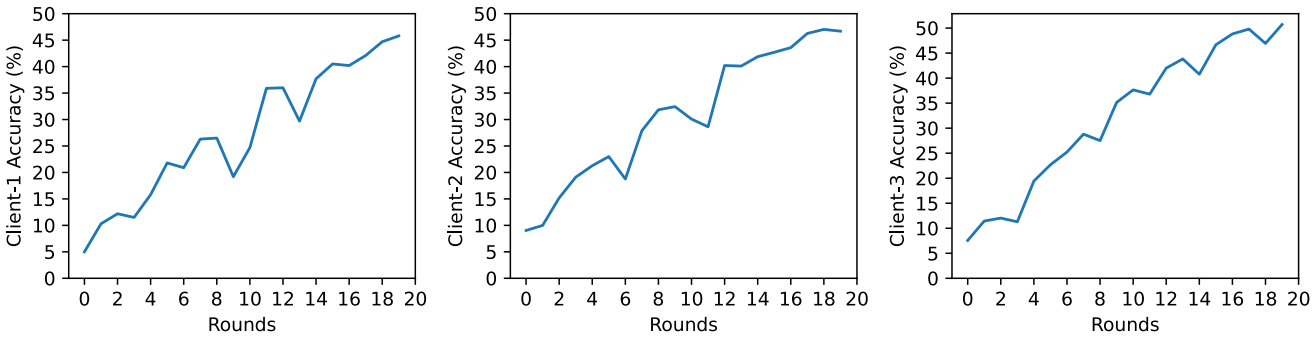


Figure 2: Improvement of test accuracy for three different clients using different samples for training from the EMNIST dataset.

2.1 Distributed Machine Learning

With the rise in the adoption of ML, it is no longer limited to single-node or single-process settings. At present, even though it might run in a single node for research purposes, it is often, if not always, run with multiple accelerators. This procedure of running ML workloads using multiple accelerators or nodes to satisfy the computational requirements is known as distributed ML. The two most common forms of distributed machine learning are data-parallel and model-parallel distributed training.

The entire dataset is partitioned across all participating nodes in equal chunks in data-parallel settings. In contrast, the model is partitioned across multiple accelerators in model-parallel settings, and different accelerators compute different parts of the entire model. These kinds of distributed training are often conducted in large in-house clusters of homogeneous computational nodes. Moreover, the dataset is partitioned into equal chunks across the participating nodes. Models are trained on these datasets in batches through iterative forward and backward passes over the model to minimize a loss function—a mathematical function for quantifying the difference between the predicted values generated by a model and the actual observed target values in a dataset. Minimizing the loss function means finding the model parameters that best fit the available data. The notable concern to us is that there often is no heterogeneity in data quantity and quality along with system resources in distributed ML.

2.2 Federated Learning

Federated Learning (FL) [31] follows the same paradigm of distributed ML in that the computation is not limited to one client. In this case, the agenda is for multiple clients to train a global model with their respective local datasets multiple times (i.e, rounds) and then pass on the trained local models back to a global server, which aggregates the local model parameters of the different clients to produce a more robust global model which all of the clients will later use. This collaboration benefits clients by allowing them to retain their data while accessing a powerful model trained on otherwise unavailable data. FL deployments have a constraint on selecting clients to participate in each round from the entire global set of clients in order to reduce communication and computational overhead: updates are collected from N (configurable) participants that completed training the earliest each round [6]. In FL, client local models are trained while the master model only aggregates the model weights from all clients. Aggregation (tensor calculation) and selection are high-speed compared to the local client training, causing no bottlenecks at the aggregator and scheduler level. The main difference between FL and traditional distributed ML is in data heterogeneity, system heterogeneity, and network heterogeneity among the participating clients, making FL significantly more challenging than conventional ML.

To tackle the heterogeneity, efforts have been made towards guided client selection [28], clustering [8], or improving upon existing ML algorithms [29, 40]. Although these works focus on improving client communication, computation, or client selection, none fully consider the I/O impact of client samples in a scaled-up federated setting. Some works [24, 26] look at client data sampling, but these are limited to in-house homogeneous cluster settings. While limited storage in client devices has been brought to attention [14], proper methods of exploiting limited memory cache due to I/O bottleneck has not been fully investigated.

In this work, we examine the I/O bottleneck from thousands of diverse client devices with limited memory cache during large-scale FL. We prototype our solution for image-based workloads, i.e., computer vision, as these workloads are known to be I/O-intensive [5, 24, 32, 33, 37] in scaled

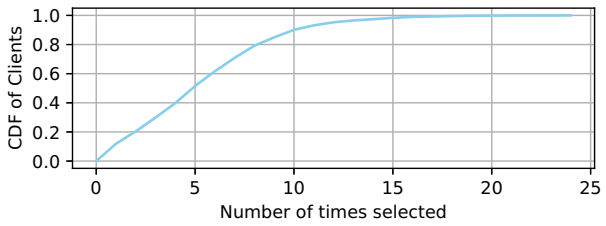


Figure 3: CDF of the number of clients selected during random scheduling on the FEMNIST dataset.

cross-device FL [20, 21]. However, our approach is general and works with other FL modes and I/O-intensive workloads.

2.3 Exploiting Client Experience

In typical cross-device FL, clients are randomly scheduled for a particular round. The random nature of client selection is suboptimal, creating scenarios where clients are scheduled only once throughout training. Selecting a client once means that a particular client has been trained for only one round. To better understand the distribution of the number of times clients get trained during random selection and the impact of rounds, we analyze random-scheduling-based client training with benchmarking datasets.

Impact of Rounds on Training. First, we look at the impact of rounds on improving the accuracy of an ML model. We track an increase in accuracy improvement while training a ResNet-18 [16] model using three clients on the EMNIST [12] dataset. We divide the entire EMNIST dataset into 1000 random chunks and assign three random chunks to three clients, respectively. Each client's data is heterogeneous and each client has a different set of samples from the EMNIST dataset. Figure 2 looks at accuracy improvement rate (not the global accuracy) across rounds for each client (x-axis) when clients train on their individual datasets. We observe that after round one, all clients achieve a local accuracy less than 10%. The accuracy increases as the number of rounds increases. This phenomenon indicates that selecting a client only once for training in FL is not ideal and that training a client for multiple rounds is more impactful.

Impact of Random Client Scheduling. Second, we analyze the number of times clients get selected when client scheduling occurs randomly. We train a ResNet-18 model on the FEMNIST [7] dataset, a federated version of the EMNIST dataset. In each round, 100 out of 2800 total clients are randomly selected each round for training over 150 rounds. Figure 3 shows that 11.9% of the total clients are scheduled for training only once and around 20.4% of the clients have been scheduled less than three times, indicating that a significant portion of clients are receiving insufficient training opportunities, which may hinder their contribution to the overall model accuracy.

As shown in Figure 2, clients need to be trained more than once or twice to reach a reasonable accuracy. However, if more time is taken for a round to complete due to I/O overhead from the flash storage, it can negatively impact the training performance. Hence, characterizing clients' experience based on their impact on decreasing round time and increasing accuracy and using that as a metric for increasing client participation in future client scheduling decisions can likely improve the performance and quality of FL training.

2.4 Exploiting Sample Experience

As previously noted, significant heterogeneity exists in the available data samples among clients. In this section, we demonstrate how leveraging this sample heterogeneity across multiple clients can effectively reduce I/O bottlenecks.

Impact of Client Data Sampling. It is known from prior research [18, 22, 30, 44] that some samples in a dataset are more important than others as they contribute more to improving the accuracy of an ML model. We verify this claim through our accuracy improvement experiment in §2.3 on the three clients shown in Figure 2. Table 1 shows the starting and end accuracy of the three clients on a test dataset. We observe that different clients having different samples have different starting and end accuracies, indicating that certain samples can indeed influence the accuracy of the model in different ways. Moreover, the accuracies of different clients vary across the training. For example, although client 2 has the highest starting accuracy, client 3 ends training with the highest accuracy. In this case, the experience of the clients as well as the samples belonging to it towards improving model accuracy varies throughout the training. We plan to exploit this variance to our advantage.

In FL, client models are aggregated to improve the global model. Hence, we can prioritize client participation along with their samples at certain intervals for improving the global model. For example, Figure 2 shows that during round 10, both clients 2 and 3 has a local accuracy over 30% but client 1 has around 20%. In this case, we can improve the global model's accuracy by prioritizing clients 2 and 3 during round 10 and training more on the samples belonging to them. Hence, to improve global model accuracy by merging client contributions, it's essential to carefully consider the impact of each client when selecting them and their data samples for future training rounds. By monitoring how client performance and sample characteristics vary, prioritizing clients with higher-quality or more experienced samples can accelerate model improvements. Training more intensively on these experienced clients with valuable data will allow

Table 1: Contribution towards accuracy improvement is different in each round for different client samples.

Client ID	Round-1 Acc (%)	Round-20 Acc (%)
1	5.05	45.81
2	9.04	46.69
3	7.56	50.71

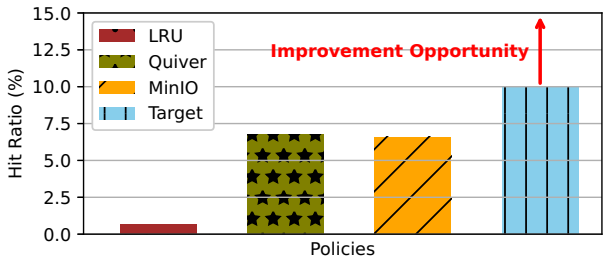


Figure 4: Global read hit ratio (RHR) of different caching policies in a heterogenous FL setup of 1000 clients.

the model to converge faster and achieve better accuracy in fewer rounds, enhancing overall system efficiency.

I/O Bottleneck in Training. As I/O bottleneck is a known problem [13, 37] in training models, recent works [24, 26] have tried to introduce caching solutions to tackle it. While promising, these solutions are specially catered for homogeneous node settings found in traditional, centralized, distributed training and do not fundamentally solve the severe heterogeneity posed by numerous client devices in FL. In FL, client devices are equipped with fast-processing accelerators [1, 3], which are more data-hungry compared with devices equipped with only CPUs, thus I/O might become a bottleneck in some FL clients. We can measure the I/O performance through looking at the global RHR and local RHR of the clients. Global RHR is the ratio of the total number of hits of all clients over the sum of their total hits and misses. Local RHR is the ratio of the total hits of a particular client over the sum of its total hits and misses.

This I/O bottleneck is exacerbated by the fact that client devices can have heterogeneity in memory cache capacity and the number and size of the data samples. Figure 4 shows that a traditional policy, LRU (least recently used), has a poor global RHR of less than 1% in an FL setup of 1000 clients having 10% working set size (WSS). Although employing new optimizations like no evictions (MinIO [32]) and sample substitution (Quiver [26]) can increase the RHR, it is still less than 10%. A lack of a client selection technique that looks at previous patterns means that a client that warmed up its cache in the current round might not be used in the future,

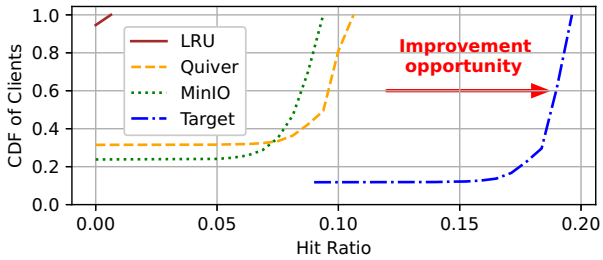


Figure 5: Heterogeneous memory creates problems in efficient utilization of memory cache of FL clients.

resulting in low global RHR. Our target policy should achieve at least a 10% hit ratio when clients cache 10% WSS, with potential to improve beyond 10% (not bounded).

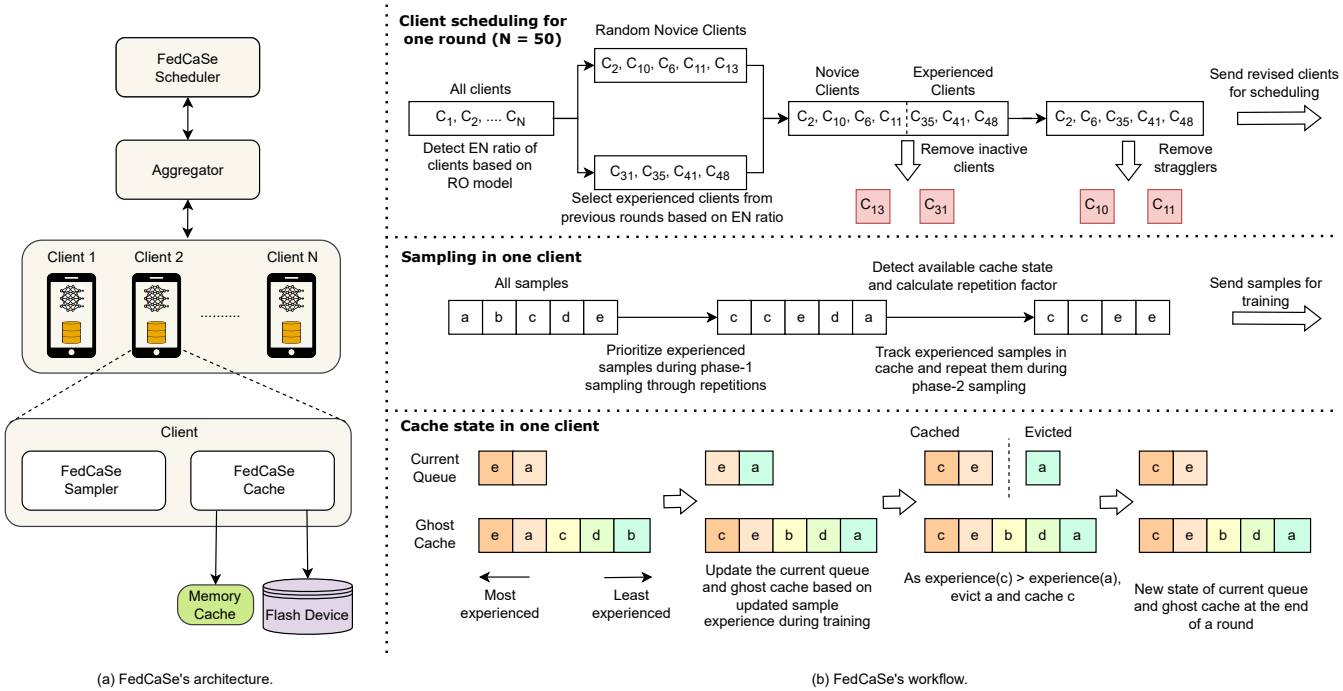
We further investigate the clients' local RHR. Figure 5 shows no client can reach an RHR of 0.1 (i.e., 10%) for LRU and MinIO. Only 19% of the clients have an RHR over 10% for Quiver. As a portion of the samples of each client are randomly selected for each round to reduce time [27], LRU evicts samples from the cache every time a new sample arrives. Moreover, a random selection of samples entails that samples in the cache might not even be used later. Hence, even after employing policies like Quiver or MinIO, only a small portion of the clients get to use the samples inside the cache. If clients cache 10% WSS, then an ideal policy (our target) should be able to exploit previous patterns in training to utilize all of the samples from their limited cache so that the local RHR becomes at least equal (but can be more) to the WSS. At the same time the ideal policy should prioritize selecting experienced and warmed-up clients so that the global RHR also increases above 10%. In this case, intelligently driving the client and sample selection can provide more opportunities for improvement.

3 FedCaSe Design

Our study in §2 sheds light on the potential to address the I/O bottleneck posed by heterogeneous client devices in an FL setting and motivates a new cooperative mechanism between client scheduling, sampling, and caching co-designed with the FL framework. This section presents the challenges and design principles of FedCaSe, followed by the design detail.

3.1 Challenges

We aim to utilize the limited memory available in client devices to enhance the training performance at client granularity and use the obtained performance benefit at the client-level training to drive future clients' scheduling decisions to improve the FL performance. This is challenging as there exists heterogeneity in terms of available memory space, the



(a) FedCaSe's architecture.

(b) FedCaSe's workflow.

Figure 6: (a) FedCaSe architecture overview. (b) An illustration of how FedCaSe's components interact in a single FL round.

number and size of client data samples, and the computational and communication abilities of the clients. Although existing policies propose certain techniques, they are limited to addressing only the computational and communication heterogeneity across clients. They overlook one key aspect, i.e., storage heterogeneity across clients. Our main task is realizing a framework that takes clients' caching and scheduling decisions in FL while considering the heterogeneity in all areas surrounding clients to provide enhanced performance.

The key insight of FedCaSe is steering I/Os, and more specifically, manipulating a cache-oriented sampling selection policy through a lens of FL scheduling, has a (positive) impact on FL training. The reason behind this is that different clients, along with their associated data samples, vary in their ability to perform in a heterogeneous setting, therefore revealing interesting exploitation opportunities to enhance the training performance by intelligently prioritizing clients and samples having a higher experience in improving FL performance.

Leveraging this insight, FedCaSe models a large FL deployment as a geo-distributed, "virtual" object cache; the objective of this newly identified cache optimization problem is to minimize training time and maximize training quality through caching the most experienced samples. To achieve the objective, we present a codesign of FL scheduling algorithm and per-client sample caching policies. However, translating potential exploit opportunities into FL training improvements poses non-trivial challenges.

First, different clients have heterogeneity regarding available memory storage, computation, and communication abilities. Hence, it becomes challenging to factor in these variables and prioritize clients for training. An ideal priority scheme would characterize clients' experience to navigate client heterogeneity and enhance the FL performance.

Second, in each round of FL, numerous clients need to be scheduled. Scheduling only experienced clients might be detrimental to the overall training as there might be other clients which have not been trained. Hence, properly fixing the *experienced* : *novice* (EN) ratio becomes non-trivial.

Third, clients have heterogeneity in the size and quantity of samples residing within them. Again, some samples might carry more utility toward improving the model accuracy. Since memory cache is limited within the clients, designing caching policies for each client individually for the samples residing within them becomes challenging.

3.2 FedCaSe Overview

FedCaSe consists of three main components—the client scheduler, the client data sampler, and the client data cache. The scheduler works in tandem with the aggregator. It calculates the experience of clients based on their ability to increase

the model accuracy (model utility) and their ability to decrease the wall clock time (I/O utility). Then, it uses the client experience as a metric to decide the appropriate experienced:novice (EN) ratio of scheduling for the next round using an optimization model. Once the clients are scheduled for training, the FedCaSe client data sampler works at the client level to calculate sample experience for all of the client's samples based on their ability to improve the model's accuracy (model utility) and prospect towards further training (train utility). Based on the sample experience, it decides the best combination of samples for training. During training, the FedCaSe cache keeps the client memory cache updated with the most experienced samples for faster retrieval to minimize training time and maximize training quality. Figure 6 shows the architecture overview of FedCaSe.

3.2.1 Client Scheduler. The client scheduler performs two main functions: (1) calculating the client experience from the client model and I/O utility; and (2) fixing the experienced:novice (EN) client ratio for scheduling clients.

To calculate the client experience from various heterogeneous variables, FedCaSe uses three techniques. The first technique captures a client's relative accuracy improvement performance (model utility) compared to others. The second technique captures the relative improvement in decreasing wall clock time (I/O utility) by a client compared to others. The third technique calculates a composite experience score using the model and I/O utility that takes into account the impact of both utilities. FedCaSe client scheduler then uses the computed experience score as a metric to determine how many and which experienced clients will take part in training in the following rounds. To determine the ratio of experienced:novice (EN) clients, FedCaSe uses a novel reverse optimization (RO) technique grounded in ML.

All these techniques combined help in conducting training with the clients, which will help increase the accuracy of the global model and decrease the round training time. Algorithm 1 shows the steps for client scheduling. At the global server level, scheduling involves tracking client experience scores and running RO policy, with no additional computations beyond standard FL procedures.

Client Model Utility (CM_u). The model utility of a client denotes the utility it carries towards improving the FL model accuracy. To understand model utility, first, we observe the loss (l_c) of each round for a particular client. Loss quantifies how far off the predicted values of a ML model are from the actual values in the training dataset. Using loss for determining importance has been explored in literature [18, 22, 30, 44]. Instead of using raw losses, FedCaSe client scheduler collects the loss of every client at the end of each round as soon as the client finishes training and uses min-max scaling to assign a relative model utility for each client in real time (Alg. 1, lines (7-13)). This standardizes the losses to a uniform scale and makes it amenable to being used with other utilities.

Algorithm 1: FedCaSe Client Scheduling

```
1  Input: C: set of all clients
2  F: {} # sorted dict(client:experience), curr_clients: {}
3  Δ_l: {}, Δ_d: {} # set of loss and round diff. of clients
4  P: {} # set of EN ratios of rounds
5  l_min, l_max: ∞, −∞, d_min, d_max: ∞, −∞
6  Function collect_client_exp(clients):
7      D: {}, L: {}# duration and loss sets of clients
8      for c in clients do
9          d_c, l_c = train(client) # duration and loss
10         D = D ∪ {d_c}, L = L ∪ {l_c}
11         l_min, l_max = min(l_min, l_c), max(l_max, l_c)
12         d_min, d_max = min(d_min, d_c), max(d_max, d_c)
13         CM_u = (l_c - l_min) / (l_max - l_min)
14         CI_u = (d_max - d_c) / (d_max - d_min)
15         client_experience = α*CI_u + β*CM_u
16         F[c] = client_experience # record experience
17     cur_t, cur_l = find_cur_dur_loss(D,L)
18     return cur_t, cur_l
19 Function get_round_diff(prev_l,prev_t):
20     cur_t, cur_l = collect_client_exp(curr_clients)
21     δ_l = cur_l - prev_l, Δ_l = Δ_l ∪ {δ_l}
22     δ_d = cur_t - prev_t, Δ_d = Δ_d ∪ {δ_d}
23     return cur_l, cur_t
24 Function find_EN(Δ_l, Δ_d):
25     Tδ_l = max(Δ_l) + std(Δ_l), Tδ_d = min(Δ_d) -std(Δ_d)
26     ρ_r = regression(Δ_l, Δ_d, P, Tδ_l, Tδ_d)
27     return ρ_r
28 Function schedule_clients(T_clients,len_rc):
29     prev_l, prev_t = 0, 0
30     for r in rounds do
31         l_r, t_r = get_round_diff (prev_l, prev_t)
32         ρ_r =find_EN(Δ_l, Δ_d)
33         P = P ∪ {ρ_r} # append ρ_r for next round
34         E_clients = ρ_r * F[:len_rc] # experienced clients
35         len_nc = len_rc - len(E_clients)
36         # select novice & new clients randomly
37         N_clients = random_permutation(C)[:len_nc]
38         selected_clients = E_clients + N_clients
39         prev_l, prev_t = l_r, t_r
40         yield selected_clients
```

Client I/O Utility (CI_u). The I/O utility of a client denotes the utility that it carries towards decreasing the round duration. To measure the I/O utility, FedCaSe client scheduler observes the time required to complete a round (d_c) by a client. Each client takes different times to complete a round due to the heterogeneity in computational, communications, and sample size differences. To understand the I/O impact of each client on a standardized scale, it uses min-max scaling to calculate each client's relative I/O utility (Alg. 1, line 14).

Client Experience. The client experience is a composite score of the model and the I/O utility for each client. Since the model utility and I/O utility are normalized to a standardized scale, the FedCaSe scheduler manages to convert the sensitivity of both metrics to a common scale. For example, the loss can be minimal (in fractions), while the time required might be very large. As a result, it becomes challenging to unify both utilities, as a minor change in one might drastically impact a client's overall utility. A relative scale for both metrics helps FedCaSe capture each metric's impact appropriately compared to others. Both I/O utility and the model utility have associated weight factors α and β respectively for placing emphasis on the particular utility required for a job. A sensitivity analysis on the impact of these weight factors has been provided in §5.3. The scheduler maintains a sorted dictionary of every client's experience and periodically updates (Alg. 1, lines 15-16) the experience in real-time as soon as each client finishes a new round.

Reverse Optimization (*RO*) Model. In FL, the global model gets impacted by heterogeneous datasets of different clients. Hence, new or novice clients need to take part in training to improve the global model accuracy. A major dilemma arises when we try to balance the experienced:novice (EN) ratio, ρ_r of clients. To address this challenge, FedCaSe scheduler uses the *RO* model to adaptively change ρ_r to reduce the round duration and improve the accuracy.

The improvement in accuracy can be understood through a stable decrease in the model loss. After every round, FedCaSe proactively checks the decrease in loss of the global model (δ_l) and the round duration (δ_d) using the current round clients (*curr_client*) due to the ρ_r of the previous round (Alg. 1, lines 19-23). It keeps a list, Δ_d for keeping (δ_d)s and another list, Δ_l for keeping (δ_l)s of consecutive rounds. To improve the wall clock time and accuracy, we need to decrease the δ_d and increase the δ_l between rounds. Deciding the increase or decrease of EN ratio to get a decrease in δ_d and an increase in the δ_l requires solving many complicated auxiliary problems that are often computationally expensive [19]. Hence, FedCaSe looks at the problem in reverse and introduces a new technique called reverse optimization (RO).

Since we aim to reduce δ_d and increase δ_l, assume that we have already decreased the δ_d and increased the δ_l. The current value of δ_d is δ_d - ∂t and the current value of δ_l is δ_l + ∂l. Now, the problem that FedCaSe aims to solve is—given that it knows the target round duration ($T\delta_d$), i.e., δ_d - ∂t and target round loss ($T\delta_l$), i.e., δ_l + ∂l, what would be the ρ_r? Hence, the complex problem now gets mapped to a classical regression analysis problem. This regression problem is solved using a regression model by the FedCaSe scheduler. δ_d and δ_l act as the features (i.e., attributes used to train) of the regression model. In the initial warm-up rounds (~10), EN ratios are decided randomly to observe the relationship between changing (ρ_r)s with δ_d and δ_l. Since the features are normalized, it improves the stability of the regression model and prevents overfitting.

As the goal is to decrease the δ_d and increase the δ_l, FedCaSe sets the desired base value of δ_d and δ_l as the min and the max of Δ_d and Δ_l respectively. The ∂t and the ∂l are obtained from the standard deviation of the Δ_d and Δ_l (Alg. 1 lines 24-27) to maintain a steady increase in δ_l and decrease in δ_d. In this way, it continuously learns to select the best EN ratio. FedCaSe always chooses the most experienced clients based on this EN ratio, which when determined intelligently, can be anything between 0-1. In each round, if the EN ratio is minimum, then less experienced clients and more new clients are selected, which tackles the overfitting and bias. Additionally, choosing the same client more times reduces the model utility, and hence, biased clients are automatically avoided by RO policy. Once the EN ratio has been predicted, it selects the most experienced clients from the dictionary, $\mathcal{F}$ ⟨K:client_id, V:client_experience⟩ sorted based on experience that it keeps (Alg. 1 lines 28-40). The impact of RO in FL has been evaluated in §5.3. The *RO* model can be trained online or offline periodically for taking scheduling decisions with negligible overhead (~0.02% of entire round).

Tackling Bias in Client Selection. Client utility is measured by both model and I/O utility. The I/O utility considers whether a device is high-end by checking how fast it can complete a round for the client. While high-end devices might be favored, this occurs only if high-end devices consist of valuable data samples too. However, this is temporary as their utility decreases through repetitive training. Moreover, to prevent bias the RO policy ensures random client selection according to EN ratio.

3.2.2 Client Data Sampler. The client data sampler works with each client to perform two main functions: (1) calculating the experience score of each data sample belonging to each client; and (2) determining the memory:flash (MF) device ratio, i.e., how many and which samples would be received from memory and flash for training in each round.

To calculate the client sample experience, it uses three techniques. The first technique captures the prospect of a client sample towards improving a model (model utility, M_u)

Algorithm 2: FedCaSe Client Data Sampling

1 **Input and Initialization:**
2 S_L: a set containing loss of each sample of a client
3 S_F: a set containing the frequency of sample usage
4 S_E: a set containing the experience of all samples
5 **for** *s* ***in*** *client_samples* **do**
6 # find model utility, M_u and train utility, T_u
7 M_u = (s_l - min(S_L)) / (max(S_L) - min(S_L))
8 T_u = (s_f - min(S_F))/ (max(S_F) - min(S_F))
9 s_e = M_u + T_u # client sample experience
10 S_E = $S_E \cup s_e$ # add updated sample experience
11 phase1_samples = prob_dist(S_E)
12 # find current samples in memory and flash cache
13 M_S, F_S = *find_experienced_cache*(phase1_samples)
14 # R_S: required samples for training
15 r = α * (R_S/ len(M_S)) # α = 0.5 by default
16 rep_samples = r * M_S # repeat samples in memory
17 len_{flash} = R_S - len(rep_samples) # quantity from flash
18 phase2_samples = rep_samples + F_S[:len_{flash}]
19 return phase2_samples

through relative loss. The second technique captures a sample's potential and usefulness towards future participation in training (train utility, T_u) through relative frequency. The third technique calculates a composite sample experience score using the model and train utility that takes into account the impact of both utilities.

FedCaSe client data sampler then uses the sample experience as a metric to determine the number and the identity of the samples that will be used for training from the memory cache and the flash storage in a particular round. Algorithm 2 shows the steps for data sampling inside each client.

Sample Model Utility (M_u). The sample model utility denotes the ability of a sample to contribute towards improving the training model accuracy. It is calculated using training loss like the client model utility. However, to quantify the impact of each sample, FedCaSe uses loss of individual data samples of a client rather than the combined batch-based training loss used in the client model utility.

Sample Train Utility (T_u). The train utility is calculated through the number of times a sample has been trained, i.e., the access frequency of the samples. It captures a sample's potential and usefulness towards future participation in training. Each client keeps a set of the access frequency of each sample and updates that set as training moves on.

Sample Experience. Once the sample model and train utility has been calculated and their relative impact determined, the sample experience, s_e, is obtained from the summation of both utilities (Alg. 2, line 7-9). Considering both model and train utility while determining experience allows FedCaSe to adapt dynamically and helps prevent convergence on a data subset.

In contrast to importance-sampling-based techniques [18, 22, 24] that focus on loss only, this experience-based approach takes into account the impact of both utilities when performing sampling from a probability distribution. As a result, it manages to prioritize samples that have been trained on more and are likely to be in the cache during training, therefore decreasing the wall clock time of the client.

At the same time, when similar samples are selected too often, their model utility decreases. Hence, experience scores give them less priority while choosing for the next round, thus avoiding sample bias. In this way, FedCaSe prioritizes important frequently used samples, but excludes them once their utility decreases. Such a bias-tackling mechanism, combined with insights from important sampling techniques, enables FedCaSe to perform repetitions in sample selection without making the model biased and sacrificing accuracy.

Sampling Procedure. FedCaSe performs sampling in two phases. In the first phase, the sampler collects a list of repetitive samples with the most experience out of all the samples belonging to the client. A probability distribution determines how the experienced and novice samples would be merged for training that will likely produce the best accuracy and round duration improvement (Alg. 2, line 11).

In the second phase, FedCaSe checks the cache for experienced samples, M_s already in the memory cache (Alg. 2, line 13). Then, according to the number of samples required for training (R_s) and the number of samples inside the client's memory cache (M_s), we find a repetition factor (r) that denotes the number of times the experienced samples would be trained more (Alg. 2, line 15). r is reduced by a depreciation factor, α, to maintain a good balance of variation in the samples used to be used for training. Experienced samples undergo further repetitions, ensuring a higher read hit ratio (RHR) for the client's memory cache (Alg. 2, lines 16-19).

3.2.3 Client Data Sample Cache. The client data cache works inside each client and performs two main functions: (1) tracking the experience of each sample in the memory cache and flash cache while dynamically updating the experience scores; and (2) making caching and eviction decisions from the memory cache during training.

Since our goal is to train more on experienced samples from a client's limited memory cache to reduce I/O time, we must always keep the memory cache occupied with the most experienced samples so that repetitions of those samples during sampling (Alg. 2) can be beneficial for training. For tracking the experience of each sample belonging to a client, FedCaSe maintains two sorted metadata queues - the current queue (CPQ) and the ghost cache (GPQ) based on the loss,

Algorithm 3: FedCaSe Client Data Sample Caching

```
1  Input and Initialization:
2  # Initialize priority queue of (loss, frequency) of
     samples in memory and trained samples,
     respectively
3  CPQ: {}, GPQ: {}
4  for s in client_samples do
5      if s in memory_cache then
6          fetch_from_memory(s)
7          l_s = get_loss_from_training() , f_s = CPQ[s][1]
8          CPQ[s] = (l_s, f_s+1)
9      else if s in GPQ then
10         l_s, f_s = GPQ[s]
11         # Calculate using Alg. 2 (line 6-9)
12         e_s = get_sample_experience(l_s, f_s)
13         s_min, (l_min, f_min) = arg min(CPQ), min(CPQ)
14         e_min = get_sample_experience(l_min, f_min)
15         if e_s > e_min then
16             evict(s_min) &CPQ.remove(s_min)
17             l_s = get_loss_from_training()
18             cache(s) & CPQ.insert(s, (l_s, f_s + 1))
19         else
20             fetch_from_flash(s)
21     else
22         fetch_from_flash(s)
23     l_s = get_loss_from_training(), f_s = GPQ[s][1]
24     GPQ[s] = (l_s, f_s + 1)
```

l_s, and the number of times a sample has been trained, f_s. These queues store l_s and f_s as a metadata tuple $\langle l_s, f_s \rangle$. The sample experience, e_s, is calculated from these queues during taking caching and eviction decisions (Alg. 3, lines 10-14). The CPQ tracks the experience associated with every client sample inside the cache, and the GPQ tracks every client sample that has already participated in the training. During the training process, if a certain sample's experience is more than the sample having the lowest experience inside the current memory cache, then the sample having the lowest experience is evicted from the cache, and the new sample having the higher experience score is cached (Alg. 3, lines 15-22). This way, the FedCaSe client cache dynamically updates the memory cache.

Memory Overhead. Each client maintains two priority queues having ⟨float, float⟩ tuples. Assuming that each client has enough space to store 100B samples (each 1MB), the total space available to the client would be $97.7 * 10^6$GB. Assuming storing each tuple will need 8 bytes, the memory overhead for maintaining the queues is very negligible (~ 0.001% of the entire space required to keep the samples).

4 Implementation

FedCaSe is implemented as three separate Python libraries for client scheduling, client data sampling, and client data caching built to work with a popular FL engine, FedScale [27]. The client scheduler determines the EN ratio by using the RandomForestRegressor model from scikit-learn [36]. The client data sampling library and the caching library work with the clients of an FL engine. FedCaSe uses PyTorch [35] multinomial and numpy [15] library to build the logic behind client data sampling. Each client builds its dataset by inheriting PyTorch's Dataset class. FedCaSe includes APIs for communicating with the memory cache to conduct caching and eviction decisions inside the Dataset class.

5 Evaluation

5.1 Experimental Setup

FedCaSe has been designed to work in extensive deployments with millions of edge devices. Nevertheless, such deployment is both cost-prohibitive and challenging to guarantee the reproducibility of experiments. Hence, we resort to training with 2800 emulated clients with the selection of up to 100 clients per round using NVIDIA P100 GPUs on Chameleon Cloud testbed [23]. We simulate real-world FL training using two representative computer vision datasets—FEMNIST [7] and CIFAR-10 [25] and two models, ResNet-18 [16] and MobileNetV2 [39] that are widely used in FL evaluations. Train and test datasets are partitioned following LEAF [7] benchmark.

Clients in our study have sample sizes ranging from 1 MB to 100 MB and capture the inherent data sample size heterogeneity in cross-device FL. Figure 7 shows the CDF of the average sample sizes of the different clients used for running our experiments. Moreover, the clients vary in the amount of available memory cache, i.e., different clients can cache different numbers of samples. Figure 8 shows the heterogeneity in memory cache space across the clients normalized based on amount of samples that can be fit in memory, i.e., data sample quantity heterogeneity. The heterogeneity related to computing and communication is simulated using data provided by FedScale [27] which uses data from AI Benchmark [17] and MobiPerf [4] to replicate real-world FL deployments.

To show the effectiveness of the FedCaSe's caching policy, we need to allow clients to use a particular portion of their memory for caching samples. Although memory cache for samples can be fixed at any percentage as necessary by the client, to ensure consistency and reproducibility, in our evaluations, we allow each client to cache 10% of its WSS in its memory cache. Note that since each client has a different

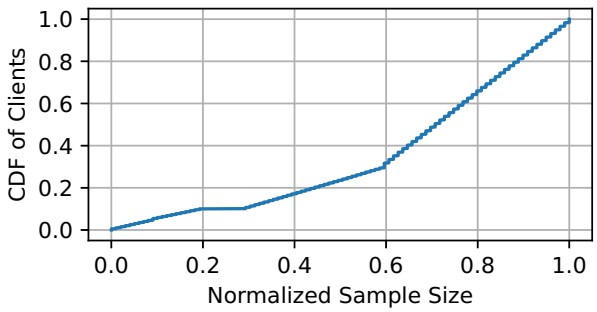


Figure 7: Sample sizes of clients differ greatly. Sample sizes normalized with respect to their sizes across x-axis.

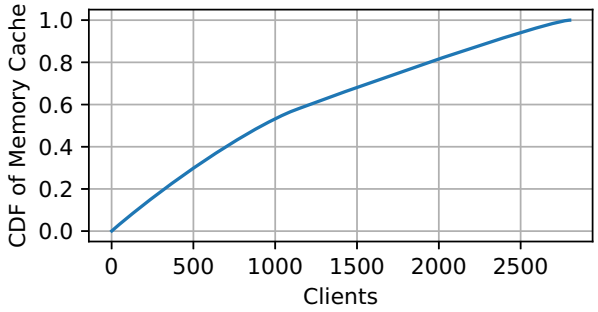


Figure 8: Memory cache space of the clients differ greatly. X-axis denotes the Client IDs.

number of samples as their WSS, the cache sizes become heterogeneous for the clients. Assume three clients have WSS sizes ⟨50,100,150⟩; then caching 10% of WSS would mean the number of samples they can cache would be 5, 10, and 15, respectively. We report the simulated wall clock time and rounds in our evaluations based on the capabilities of the devices. While the operation time for individual devices is estimated, the actual training is performed on real data samples using real GPUs to enhance model accuracy.

Our evaluation aims to address the following questions:

- How much can FedCaSe increase the global and local RHR in large-scale heterogeneous FL? What is the impact of each of its component on RHR? (§5.2)
- How can the improved RHR translate into improved model accuracy during FL training through exploiting client and sample experience? How much adaptive is the Reverse Optimization (RO) policy throughout the training process? (§5.3)
- How much can FedCaSe decrease the round duration in FL? How much does each of its utility functions contribute to reducing the number of rounds? (§5.4)

5.2 Impact on Read Hit Ratio (RHR)

In this section, we evaluate FedCaSe's client data sample caching policy against other state-of-the-art caching policies used in traditional homogeneous settings. We evaluate the RHR of FedCaSe against three state-of-the-art policies—SHADE [24], Quiver [26], and MinIO [32] along with traditional policies like LRU and LFU (least frequently used) to show how much FedCaSe can improve the global and local RHR of thousands of clients having heterogeneous memory cache. We also perform an ablation study on FedCaSe's different policies (C: Caching, S: Sampling, Sched: Scheduling) to understand how much each policy impacts the RHR.

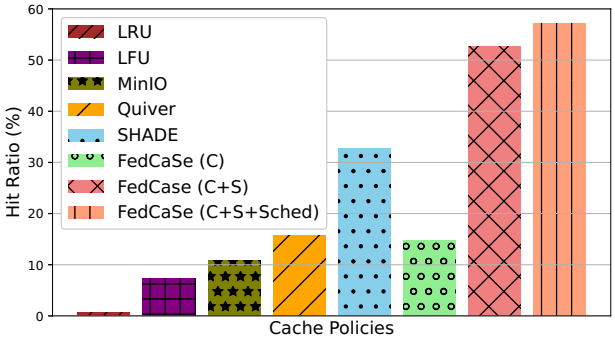


Figure 9: Global read hit ratio (RHR) of different caching policies in a heterogenous FL setup of 2800 clients. Ablation study on three components of FedCaSe (C: Caching, S: Sampling, and Sched: Scheduling).

SHADE clients (i.e., clients using SHADE) use a sample priority-aware caching technique for caching samples in their memory. Quiver clients use a substitutability technique that prioritizes samples already in the client's cache for training. MinIO clients do not evict samples once they are cached in memory. FedCaSe differs from these policies in two aspects—it samples based on clients' heterogeneous memory cache space and caches based on sample experience, which improves its RHR and accuracy. The clients are scheduled using both the default scheduling technique (i.e., random) and a state-of-the-art scheduling technique, Oort [28].

Ablation Study. Figure 9 shows that FedCaSe (C) performs similarly to Quiver. However, when we consider memory heterogeneity in the sampling procedure and adjust the experienced samples based on each client's limited memory cache size, FedCaSe (C+S) improves performance by 3.59×. Including experience-based scheduling, FedCaSe (C+S+Sched) further enhances the performance, ensuring a 1.75×, 3.62×, 5.29×, 7.73×, and 81.72× higher RHR compared to SHADE,

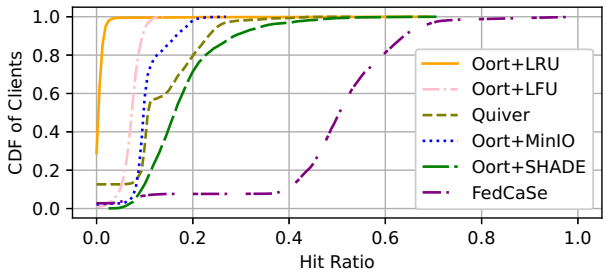


Figure 10: FedCaSe outperforms other state-of-the-art policies in local RHR at the scale of thousands of heterogeneous clients.

Quiver, MinIO, LFU, and LRU cache policies respectively when trained for 150 rounds. LRU and LFU policies perform the worst in RHR as these policies cannot exploit random data sampling patterns of thousands of clients in the limited heterogeneous cache. Since SHADE uses loss-based sampling to enhance RHR, it performs better than the other policies. However, like other policies discussed in §2, SHADE cannot adjust the sampling and scheduling to match the heterogeneity of the client devices. FedCaSe's experience-based scheduling policy prioritizes the usage of warmed-up clients in the future, thus outperforming SHADE in global RHR.

Next, we examine the individual contributions of each client, i.e., local RHR, compared to the collective gain in global RHR across the entire set of 2800 clients. Figure 10 illustrates that in FedCaSe, 92.39% clients have a RHR greater than 0.2, which is 318.58×, 48.37×, 3.65×, and 3.24× more compared to LRU, MinIO, Quiver, and SHADE respectively. Lack of a sample retention policy like FedCaSe means new samples displace existing ones in the cache, resulting in a high eviction rate. Since FedCaSe learns from previous patterns in training through sample experience, it can properly guide the sampling procedure of clients with limited heterogeneous memory to increase the local RHR.

5.3 Impact on Accuracy Improvement

To investigate whether the gains in RHR shown in the previous section translate well into accuracy improvement, in this section, we evaluate the accuracy improvement of FedCaSe against random and advanced sampling and scheduling techniques. We use top-1 accuracy for evaluation, which denotes the percentage of predictions where the model's top prediction matches the true label of the input data. We train up to the point when the accuracy curve starts to plateau, indicating convergence.

In the first set of experiments, we evaluate FedCaSe against three baselines based on random client scheduling, data sampling, and traditional caching policies:

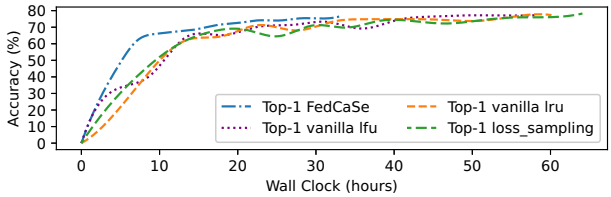


Figure 11: Accuracy improvement of FedCaSe vs. `vanilla` configurations using ResNet-18 on FEMNIST.

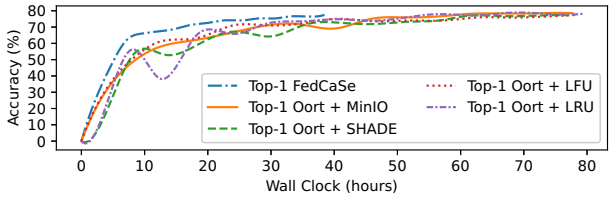


Figure 12: Accuracy improvement rate of FedCaSe vs. Oort. Training ResNet-18 on FEMNIST for 150 rounds.

(1) `vanilla lfu`, which performs client scheduling and client data sampling randomly. However, clients can put 10% WSS in the cache and evict the data samples based on the LFU policy when updating the cache.
(2) `vanilla lru`, which performs client scheduling and client data sampling randomly. However, clients can put 10% WSS in the cache and evict the data samples based on the LRU policy when updating the cache.
(3) `loss_sampling`, which uses a loss-based advanced sampling policy like SHADE [24] and Mercury [43], and it is equipped with a caching policy that caches samples based on loss importance and evicts samples randomly when looking to cache important samples.

We observe the accuracy improvement rate of FedCaSe and the baselines during training. Figure 11 shows that FedCase, caching 10% WSS, is up to 1.85× faster than `vanilla` and `loss_sampling`. These baselines are unable to choose samples and adapt the sampling conveniently to improve accuracy when clients have a limited heterogeneous memory cache. Although `loss_sampling` chooses samples based on importance, it cannot fully exploit the opportunities that important samples present at client granularity. While FedCaSe decides the experience based on model utility and train utility of a sample, `loss_sampling` decides the samples for the next round based only on loss and disregards a crucial metric, i.e., train utility to filter samples. As train utility is not considered, `loss_sampling` cannot reward experienced samples based on I/O benefits and hence suffers from increased round durations.

Table 2: The improvement of FedCaSe in number of rounds clients get scheduled in different ranges over vanilla and Oort. ≥ X means a single client gets called ≥ X times throughout training.

Client Calls	Improvement (vanilla)	Improvement (Oort)
≥ 10	1.51×	3.01×
≥ 15	3.28×	6.66×
≥ 20	19.61×	29.1×

Next, we compare FedCaSe's client scheduling technique against four additional baselines using Oort [28], a state-of-the-art client scheduling technique. Oort uses loss generated by clients to detect client importance, thereby using that as a driving metric for selecting clients in the next rounds. Oort penalizes clients which become stragglers but does not reward clients which complete rounds faster, like FedCaSe. Hence, Oort does not fully capture a client's ability to reduce round duration. Additionally, Oort prioritizes exploring more clients rather than training more on already experienced clients like FedCaSe. This approach makes Oort leave out on gaining the most benefits from a single client due to lack of training as previously discussed in §2.3.

For evaluation, we equip Oort with four caching policies to show the impact FedCaSe's experienced clients have on accuracy improvement against advanced policies:

(1) `Oort + MinIO`, which uses a caching policy similar to MinIO [32] where cached samples are never evicted.
(2) `Oort + SHADE`, which leverages SHADE [24], a policy using importance of samples to take eviction decisions.
(3) `Oort + LFU`, which evicts the data samples based on the LFU policy when updating the cache.
(4) `Oort + LRU`, which evicts the data samples based on the LRU policy when updating the cache.

Figure 12 shows that FedCaSe takes 2.02×, 2×, 2.04×, and 2.06× less time than `Oort + MinIO`, `Oort + SHADE`, `Oort + LFU`, and `Oort + LRU` respectively to reach accuracy convergence.

To understand why accuracy improved faster, we further analyze client participation compared to `vanilla` and Oort configurations across different ranges. Table 2 shows that FedCaSe can call experienced clients over 20 times (i.e., the same client takes part in entire training for over 20 rounds) throughout training 19.6× and 29.1× more compared to `vanilla` and Oort respectively. As Oort prioritizes clients with a higher impact on accuracy improvement, it explores more clients which are inexperienced. In contrast, FedCaSe uses RO policy to find experienced clients across rounds. Figure 13 shows that the RO policy adapts itself (40% to 90% of total client selection can be experienced) to minimize the

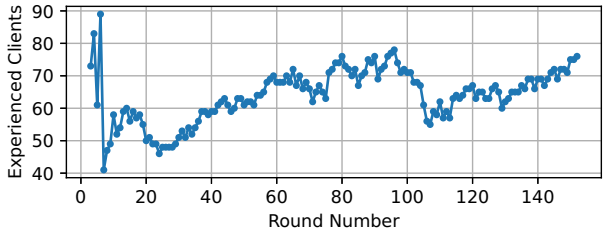


Figure 13: Adaptivity of RO policy in scheduling experienced clients.

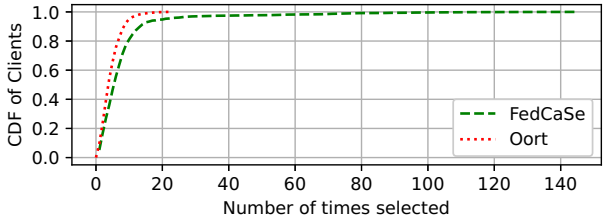


Figure 14: Experienced clients get scheduled for participation in future training more frequently in FedCaSe.

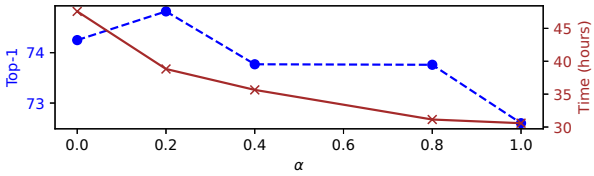


Figure 15: Sensitivity analysis on the impact (Top-1 accuracy and training time) of placing weight, α on I/O utility when determining client experience.

training time and maximize the accuracy improvement while exploiting the limited memory cache of clients.

Figure 14 shows that the same clients are scheduled more frequently (some over 100 times) in FedCaSe compared to Oort. As Oort places less emphasis on the I/O utility of clients and more on exploration, it ends up with a suboptimal list of clients for decreasing the I/O time in each round. On the other hand, as FedCaSe's policy is to train more on experienced clients which already have experienced samples stored in the cache, it ensures faster accuracy improvement.

Sensitivity Analysis. Two parameters that affect the client experience, and thus the client scheduling, are the tunable weight parameters α and β of the client's I/O utility and model utility, respectively. We change the values of α between [0,1], where β = 1 - α, and train for 150 rounds on the FEMNIST dataset using ResNet-18. Figure 15 shows that

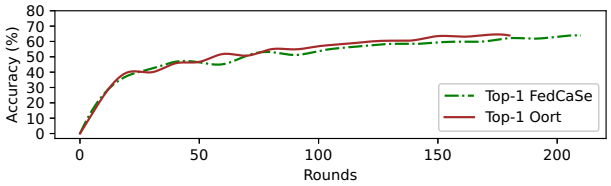


Figure 16: Accuracy improvement rate of FedCaSe vs. Oort with respect to rounds. Training ResNet-18 on CIFAR-10.

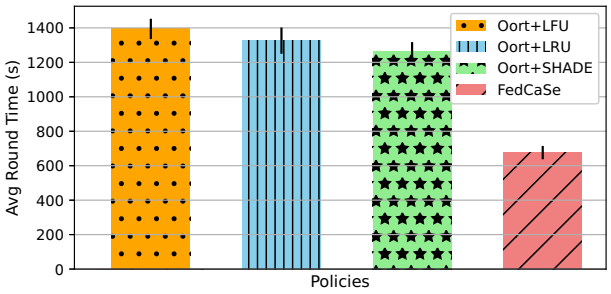


Figure 17: FedCaSe vs. Oort round time improvement using different cache policies during training MobileNetV2 on FEMNIST.

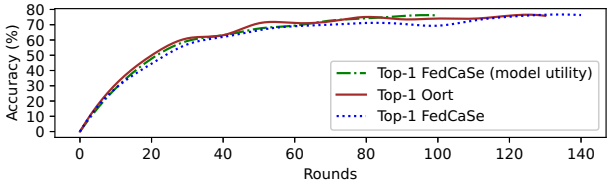


Figure 18: Accuracy improvement rate of FedCaSe vs. Oort with respect to rounds. Ablation study on FedCaSe's model and I/O utility fucntions. Training ResNet-18 on FEMNIST.

the training time and top-1 accuracy decrease as α increases, thus prioritizing clients that reduce the training time.

5.4 Impact on Rounds

In this section, we will evaluate how FedCaSe's policies impact the number and duration of rounds in FL. Given that Oort prioritizes exploring clients, which can increase accuracy, it is expected to reach accuracy convergence in fewer rounds. However, each round takes significantly longer as I/O utility is only partially considered in client scheduling.

When training ResNet-18 on FEMNIST shown in Figure 12, we observe that although FedCaSe requires 1.25× more rounds to reach Oort's accuracy, FedCaSe's round duration is 2.04× less. Figure 16 shows that although FedCaSe takes 1.16× more rounds than Oort when training ResNet-18 on CIFAR-10, its round duration is 2.4× less as it strikes a good balance between model and I/O utility of clients. Hence, FedCaSe can quickly increase its accuracy improvement rate (1.6×) compared to Oort based on wall clock time.

We perform another round of experiments after equipping Oort with three cache policies: LRU, LFU, and SHADE, to visualize the average round time taken by these baselines using MobileNetV2 model on FEMNIST. Figure 17 shows that FedCaSe round time is 2.06×, 1.96×, and 1.87× less compared to `Oort + LFU`, `Oort + LRU`, and `Oort + SHADE` respectively. As discussed in §5.2, FedCaSe's ability to increase the global RHR significantly reduces the round duration.

Ablation Study. By default, FedCaSe prioritizes both the model and I/O utility during data sampling and client scheduling. Hence, although it can take a few more rounds to reach accuracy convergence compared to Oort, the wall clock time taken decreases significantly. However, users can also empirically prioritize the model utility during data sampling and client selection if accuracy convergence needs to be achieved in fewer rounds. Moreover, if users do not want to use a cache, or does not have a sufficient one, they can prioritize model utility to get the maximum performance. We try to understand the impact of model and I/O utility in decreasing the number of rounds during training through an ablation study where FedCaSe uses both model and I/O utility and FedCaSe (model utility) uses only model utility.

Figure 18 shows that prioritizing model utility can enable FedCaSe to reach accuracy convergence up to 1.4× faster compared to Oort with respect to rounds. This result also shows that even without a cache (i.e., 0% WSS in cache), FedCaSe's client utility measuring mechanisms can boost the performance of training. While Oort prioritizes selecting clients with a higher potential to increase accuracy, it performs random data sampling within clients. Instead of random sampling, FedCaSe emphasizes model utility during both data sampling and client scheduling, which results in faster convergence with respect to rounds.

6 Assumptions and Limitations

In this section we discuss the assumptions and limitations that may influence the scope and applicability of our findings to provide a comprehensive understanding of our approach.

Client Incentivization. Client incentivization is a separate branch of FL research that is orthogonal to our work, as we assume voluntary client participation, similar to related studies [8, 9, 28]. In future work, we plan to extend FedCaSe with incentive mechanisms to encourage client contributions based on observed utilities.

Broader Applicability of the Proposed Solution. Our approach is applicable to NLP tasks, as NLP models can use loss during training and large text documents can cause I/O bottlenecks. However, for NLP workloads, we need to make trade-offs between the importance of tokens, sentences, or sequences during calculating model utility. In tasks like translation, summarization, or question-answering, sentence pairs may influence caching decisions. We plan to explore these challenges in future work.

7 Related Work

Client Scheduling. A line of research [8, 10, 11, 28, 34, 41, 42] is aimed towards optimizing client scheduling in FL. Nonetheless, all of the works try to improve client scheduling based on the communication or computing abilities of the clients and do not fully consider the I/O utility of heterogeneous clients. Hence, these policies are prone to sub-optimal performance when running I/O intensive workloads in FL. As FedCaSe intelligently drives the client scheduling based on model and I/O utility, it performs significantly better when memory and data heterogeneity exists across clients in FL.

Client Data Sampling. Recent research works [24, 38, 43] use some form of importance metric for sampling data of clients. However, these sampling techniques do not address the system and memory heterogeneity that occurs in FL. Since FedCaSe performs data sampling based on the samples' experience focused on improving I/O time and accuracy, it performs better in FL settings with heterogeneous system and memory specifications.

Client Data Caching. Several works [24, 26, 32] have proposed policies for caching data samples. Nevertheless, these policies excel in homogeneous settings commonly encountered in traditional clusters, overlooking the heterogeneity arising from data samples and memory cache management of millions of devices in FL. FedCaSe's caching policy is adaptive according to millions of clients' heterogeneous limited memory capacity, and it performs eviction based on the importance of their associated data samples, thereby keeping the most important samples for each client at any moment during the training for faster retrieval.

8 Conclusion

System resource heterogeneity across the network, memory, computing power, etc., poses a significant challenge in improving performance across millions of FL client devices. Furthermore, heterogeneity in data sample size and quantity exacerbates the issue. While several policies address specific aspects of heterogeneity, none address how to navigate a crucial heterogeneous resource—memory across millions of clients. FedCaSe is the first to realize a unified intelligent client scheduling, data sampling, and caching solution for millions of client devices having heterogeneous limited memory sizes. FedCaSe leverages client experience and sample experience, calculated based on the ability to increase model accuracy and decrease round time to intelligently and adaptively drive the client scheduling, data sampling, and caching decisions. Thus, it improves the accuracy improvement rate by up to 2.06× and increases the read-hit ratio by up to 81.72× globally compared to state-of-the-art client scheduling and caching policies.

Acknowledgments

We thank our shepherd, Matthias Boehm, and our anonymous reviewers for their detailed feedback and valuable suggestions. This work is sponsored in part by the NSF under the grants: CSR-2312785, CSR-2106634, CCF-1919113, CCF 2318628, CCF 1919075, CMMI 2134689, OAC 2106446, CNS 2322860, by SERB, Government of India under grant SRG/2023/002445, by BITS BioCyTiH Foundation under grant BBF/BITS(G)/FY2022-23/BCPS-123/24-25/R1, and by BITS Pilani under grants GOA/ACG/2022-2023/Oct/11, and BITS CDRF - C1/23/173. Results presented in this paper were obtained using Chameleon Cloud supported by the NSF.

References

[1] 2021. Mobile GPU rankings 2021. https://www.techcenturion.com/mobile-gpu-rankings.

[2] 2021. POLARIS Market Research. https://tinyurl.com/polarisfedmarket.

[3] 2021. Smartphone GPU ranking. https://www.phoneworld.com.pk/best-gpu-ranking-list/.

[4] 2024. The M-Lab MobiPerf Data Set. https://www.measurementlab.net/tests/mobiperf/.

[5] Roman Böhringer, Nikoli Dryden, Tal Ben-Nun, and Torsten Hoefler. 2021. Clairvoyant Prefetching for Distributed Machine Learning I/O. *arXiv preprint arXiv:2101.08734* (2021).

[6] 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.

[7] Sebastian Caldas, Sai Meher Karthik Duddu, Peter Wu, Tian Li, Jakub Konečnỳ, H Brendan McMahan, Virginia Smith, and Ameet Talwalkar. 2018. Leaf: A benchmark for federated settings. *arXiv preprint arXiv:1812.01097* (2018).

[8] Zheng Chai, Ahsan Ali, Syed Zawad, Stacey Truex, Ali Anwar, Nathalie Baracaldo, Yi Zhou, Heiko Ludwig, Feng Yan, and Yue Cheng. 2020. Tifl: A tier-based federated learning system. In *Proceedings of the 29th international symposium on high-performance parallel and distributed computing*. 125–136.

[9] Zheng Chai, Yujing Chen, Ali Anwar, Liang Zhao, Yue Cheng, and Huzefa Rangwala. 2021. FedAT: A high-performance and communication-efficient federated learning system with asynchronous tiers. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 1–16.

[10] Wenlin Chen, Samuel Horvath, and Peter Richtarik. 2020. Optimal client sampling for federated learning. *arXiv preprint arXiv:2010.13723* (2020).

[11] Yae Jee Cho, Jianyu Wang, and Gauri Joshi. 2022. Towards understanding biased client selection in federated learning. In *International Conference on Artificial Intelligence and Statistics*. PMLR, 10351–10375.

[12] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and Andre Van Schaik. 2017. EMNIST: Extending MNIST to handwritten letters. In *2017 international joint conference on neural networks (IJCNN)*. IEEE, 2921–2926.

[13] Nikoli Dryden, Roman Böhringer, Tal Ben-Nun, and Torsten Hoefler. 2021. Clairvoyant prefetching for distributed machine learning I/O. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[14] Chen Gong, Zhenzhe Zheng, Fan Wu, Yunfeng Shao, Bingshuai Li, and Guihai Chen. 2023. To Store or Not? Online Data Selection for Federated Learning with Limited Storage. In *Proceedings of the ACM Web Conference 2023*. 3044–3055.

[15] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

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

[17] Andrey Ignatov, Radu Timofte, Andrei Kulik, Seungsoo Yang, Ke Wang, Felix Baum, Max Wu, Lirong Xu, and Luc Van Gool. 2019. Ai benchmark: All about deep learning on smartphones in 2019. In *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*. IEEE, 3617–3635.

[18] Tyler B Johnson and Carlos Guestrin. 2018. Training deep models faster with robust, approximate importance sampling. *Advances in Neural Information Processing Systems* 31 (2018), 7265–7275.

[19] Donald R Jones, Matthias Schonlau, and William J Welch. 1998. Efficient global optimization of expensive black-box functions. *Journal of Global optimization* 13 (1998), 455–492.

[20] 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.

[21] Sai Praneeth Karimireddy, Martin Jaggi, Satyen Kale, Mehryar Mohri, Sashank Reddi, Sebastian U Stich, and Ananda Theertha Suresh. 2021. Breaking the centralized barrier for cross-device federated learning. *Advances in Neural Information Processing Systems* 34 (2021), 28663–28676.

[22] Angelos Katharopoulos and Francois Fleuret. 2018. Not All Samples Are Created Equal: Deep Learning with Importance Sampling. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer Dy and Andreas Krause (Eds.). PMLR, 2525–2534. http://proceedings.mlr.press/v80/katharopoulos18a.html

[23] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. 2020. Lessons Learned from the Chameleon Testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association.

[24] Redwan Ibne Seraj Khan, Ahmad Hossein Yazdani, Yuqi Fu, Arnab K. Paul, Bo Ji, Xun Jian, Yue Cheng, and Ali R. Butt. 2023. SHADE: Enable Fundamental Cacheability for Distributed Deep Learning Training. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. USENIX Association, Santa Clara, CA, 135–152. https://www.usenix.org/conference/fast23/presentation/khan

[25] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

[26] Abhishek Vijaya Kumar and Muthian Sivathanu. 2020. Quiver: An Informed Storage Cache for Deep Learning. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*. 283–296.

[27] 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 *International Conference on Machine Learning*. PMLR, 11814–11827.

[28] 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.

[29] Tian Li, Maziar Sanjabi, Ahmad Beirami, and Virginia Smith. 2019. Fair resource allocation in federated learning. *arXiv preprint arXiv:1905.10497* (2019).

[30] Ilya Loshchilov and Frank Hutter. 2015. Online batch selection for faster training of neural networks. *arXiv preprint arXiv:1511.06343* (2015).

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

[32] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. 2020. Analyzing and mitigating data stalls in DNN training. *arXiv preprint arXiv:2007.06775* (2020).

[33] Derek G Murray, Jiri Simsa, Ana Klimovic, and Ihor Indyk. 2021. tf.data: A machine learning data processing framework. *arXiv preprint arXiv:2101.12127* (2021).

[34] Takayuki Nishio and Ryo Yonetani. 2019. Client selection for federated learning with heterogeneous resources in mobile edge. In *ICC 2019-2019 IEEE international conference on communications (ICC)*. IEEE, 1–7.

[35] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*. 8026–8037.

[36] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[37] Sarunya Pumma, Min Si, Wu-Chun Feng, and Pavan Balaji. 2019. Scalable deep learning via I/O analysis and optimization. *ACM Transactions on Parallel Computing (TOPC)* 6, 2 (2019), 1–34.

[38] Elsa Rizk, Stefan Vlaski, and Ali H Sayed. 2021. Optimal importance sampling for federated learning. In *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 3095–3099.

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

[40] Jianyu Wang and Gauri Joshi. 2019. Adaptive communication strategies to achieve the best error-runtime trade-off in local-update SGD. *Proceedings of Machine Learning and Systems* 1 (2019), 212–229.

[41] Shibo Wang, Shusen Yang, Hailiang Li, Xiaodan Zhang, Chen Zhou, Chenren Xu, Feng Qian, Nanbin Wang, and Zongben Xu. 2022. PyramidFL: A fine-grained client selection framework for effiicient federated learning. In *28th ACM Annual International Conference on Mobile Computing and Networking, MobiCom 2022*. Association for Computing Machinery, 542–555.

[42] Hongda Wu and Ping Wang. 2022. Node selection toward faster convergence for federated learning on non-iid data. *IEEE Transactions on Network Science and Engineering* 9, 5 (2022), 3099–3111.

[43] Xiao Zeng, Ming Yan, and Mi Zhang. 2021. Mercury: Efficient On-Device Distributed DNN Training via Stochastic Importance Sampling. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*. 29–41.

[44] Peilin Zhao and Tong Zhang. 2015. Stochastic optimization with importance sampling for regularized loss minimization. In *international conference on machine learning*. PMLR, 1–9.