Oort: Informed Participant Selection for Scalable Federated Learning

Fan Lai, Xiangfeng Zhu, Harsha V. Madhyastha, Mosharaf Chowdhury
University of Michigan

Abstract

Federated Learning (FL) is an emerging direction in distributed machine learning (ML) that enables in-situ model training and testing on edge data. Despite having the same end goals as traditional ML, FL executions differ significantly in scale, spanning thousands to millions of participating devices. As a result, data characteristics and device capabilities vary widely across clients. Yet, existing efforts randomly select FL participants, which leads to poor model and system efficiency.

In this paper, we propose Oort to improve the performance of federated training and testing with guided participant selection. With an aim to improve time-to-accuracy performance in model training, Oort prioritizes the use of those clients who have both data that offers the greatest utility in improving model accuracy and the capability to run training quickly. To enable FL developers to interpret their results in model testing, Oort enforces their requirements on the distribution of participant data while improving the duration of federated testing by cherry-picking clients. Our evaluation shows that, compared to existing participant selection mechanisms, Oort improves time-to-accuracy performance by 1.2×-14.1× and final model accuracy by 1.3%-9.8%, while efficiently enforcing developer-specified model testing criteria at the scale of millions of clients.

1 Introduction

Machine learning (ML) today is experiencing a paradigm shift from cloud datacenters toward the edge [20, 45]. Edge devices, ranging from smartphones and laptops to enterprise surveillance cameras and edge clusters, routinely store application data and provide the foundation for machine learning beyond datacenters. With the goal of not exposing raw data, large companies such as Google and Apple deploy *federated learning (FL)* for computer vision (CV) and natural language processing (NLP) tasks across user devices [2, 26, 35, 76]; NVIDIA applies FL to create medical imaging AI [51]; smart cities perform in-situ image training and testing on AI cameras to avoid expensive data migration [37, 43, 53]; and video streaming and networking communities use FL to interpret and react to network conditions [10, 74].

Although the life cycle of an FL model is similar to that in traditional ML, the underlying execution in FL is spread across thousands to millions of devices in the wild. Similar to traditional ML, the FL developer often first prototypes model architectures and hyperparameters with a proxy dataset. After selecting a suitable configuration, she can use federated training to improve model performance by training across a crowd of participants [20, 45]. The wall clock time for training a model to reach an accuracy target (i.e., time-to-accuracy) is still a key performance objective even though it may take significantly longer than centralized training [45]. To circumvent biased or stale proxy data in hyperparameter tuning [59], to inspect these models being trained, or to validate deployed models after training [73,74], developers may want to perform federated testing on the real-life client data, wherein enforcing their requirements on the testing set (e.g., N samples for each category or following the representative categorical distribution[1]) is crucial for them to reason about model performance under different data characteristics [22, 59].

Unfortunately, clients may not all be simultaneously available for FL training or testing [45]; they may have heterogeneous data distributions and system capabilities [20, 39]; and including too many may lead to wasted work and suboptimal performance [20] (§2). Consequently, a fundamental problem in practical FL is the *selection of a "good" subset of clients as participants*, where each participant locally processes its own data, and only their results are collected and aggregated at a (logically) centralized coordinator.

Existing works optimize for *statistical model efficiency* (i.e., better training accuracy with fewer training rounds) [24, 49, 61, 70] or *system efficiency* (i.e., shorter rounds) [56, 69], while randomly selecting participants. Although random participant selection is easy to deploy, unfortunately, it results in poor performance of federated training because of large heterogeneity in device speed and/or data characteristics. Worse, random participant selection can lead to biased testing sets and loss of confidence in results. As a result, developers often resort to more participants than perhaps needed [59, 71].

We present Oort for FL developers to enable guided participant selection throughout the life cycle of an FL model (§3). Specifically, Oort cherry-picks participants to improve time-to-accuracy performance for federated training, and it enables developers to specify testing criteria for federated model testing. It makes informed participant selection by relying on the information already available in existing FL solutions [45] with little modification.

Selecting participants for federated training is challenging because of the trade-off between heterogeneous system and

[1] A categorical distribution is a discrete probability distribution showing how a random variable can take the result from one of K possible categories.

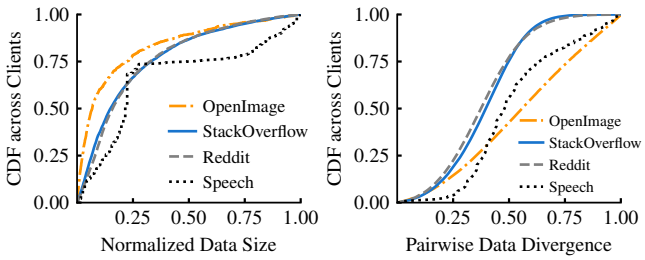
statistical model utilities both across clients and of any specific client over time (as the trained model changes). First, simply picking clients with high statistical utility can lead to longer training rounds due to the coupled nature of client data and system performance. The challenge is further exacerbated by the large population, as capturing the latest utility of all clients is impractical. As such, we identify clients with high statistical utility, which is measured in terms of their most recent aggregate training loss, adjusted for spatiotemporal variations, and penalize the utility of a client if her system speed is likely to elongate the duration necessary to complete global aggregation. To navigate the sweet point of jointly maximizing statistical and system efficiency, we adaptively allow for longer training rounds to admit clients with higher statistical utility. We then employ an online exploration-exploitation strategy to probabilistically select participants among high-utility clients for robustness to outliers. Our design can accommodate diverse selection criteria (e.g., fairness), and deliver improvements while respecting privacy (§4).

Although FL developers often have well-defined requirements on their testing data, satisfying these requirements is not straightforward. Similar to traditional ML, developers may request a testing dataset that follows the global distribution to avoid testing on all clients [40, 59]. However, clients' data characteristics in some private FL scenarios may not be available [31, 76]. To preserve the deviation target of participant data from the global, Oort performs participant selection by bounding the number of participants needed. Second, for cases where clients' data characteristics are provided [53], developers can specify specific distribution of the testing set to debug model efficiency (e.g., using balanced distribution) [16, 77]. At scale, satisfying this requirement in FL suffers large overhead. Therefore, we propose a scalable heuristic to efficiently enforce developer requirements, while optimizing the duration of testing (§5).

We have integrated Oort with PySyft (§6) and evaluated it across various FL tasks with real-world workloads (§7). Compared to the state-of-the-art selection techniques used in today's FL deployments [23, 71, 76], Oort improves time-to-accuracy performance by 1.2×-14.1× and final model accuracy by 1.3%-9.8% for federated training, while achieving close-to-optimal statistical performance. For federated testing, Oort can efficiently respond to developer-specified data distribution across millions of clients, and improves testing duration by 4.7× on average over state-of-the-art solutions.

Overall, we make the following contributions in this paper:

1. We highlight the tension between statistical and systems efficiency when selecting FL participants and present Oort to effectively navigate the tradeoff.
2. We propose participant selection algorithms to improve the time-to-accuracy performance of training and to scalably enforce developers' FL testing criteria.
3. We implement and evaluate these algorithms at scale in Oort, showing both statistical and systems performance improvements over the state-of-the-art.



(a) Unbalanced data size. (b) Heterogeneous data distribution.

Figure 1: ***Client data differs in size and distribution greatly.***

2 Background and Motivation

We start with a quick primer on federated learning (§2.1), followed by the challenges it faces based on our analysis of real-world datasets (§2.2). Next, we highlight the key shortcomings of the state-of-the-art that motivate our work (§2.3).

2.1 Federated Learning

Training and testing play crucial roles in the life cycle of an FL model, whereas they have different criteria.

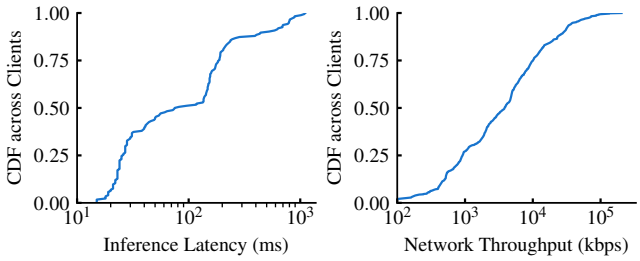
Federated model training aims to learn an accurate model across thousands to potentially millions of clients. Because of the large population size and diversity of user data and their devices in FL, training runs on a subset of clients (hundreds of participants) in each round, and often takes hundreds of rounds (each round lasts a few minutes) and several days to complete. For example, in Gboard keyboard, Google runs federated training of NLP models over weeks across 1.5 million end devices [4, 76]. For a given model, achieving a target model accuracy with less wall clock time (i.e., time-to-accuracy) is still the primary target [49, 64].

To inspect a model's accuracy during training (e.g., to detect cut-off accuracy), to validate the trained model before deployment [23, 71, 76], or to circumvent biased proxy data in hyperparameter tuning [16, 63], FL developers sometimes test model's performance on real-life datasets. Similar to traditional ML, developers often request the representativeness of the testing set with requirements like *"50k representative samples"* [16], or *"x samples of class y"* to investigate model performance on specific categories [77]. When the data characteristics of participants are not available, coarse-grained yet non-trivial requests, such as *"a subset with less than X% data deviation from the global"* are still informative [55, 59].

2.2 Challenges in Federated Learning

Apart from the challenges faced in traditional ML, FL introduces new challenges in terms of data, systems, and privacy.

Heterogeneous statistical data. Data in each FL participant is typically generated in a distributed manner under different contexts and stored independently. For example, images collected by cameras will reflect the demographics of each



(a) Heterogeneous compute capacity. (b) Heterogeneous network capacity.

Figure 2: ***Client system performance differs significantly.***

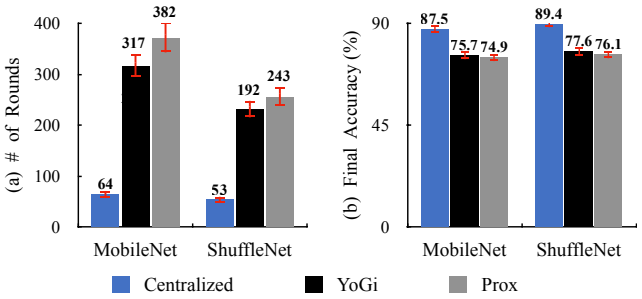


Figure 3: ***Existing works are suboptimal in: (a) round-to-accuracy performance and (b) final model accuracy. (a) reports number of rounds required to reach the highest accuracy of Prox on MobileNet (i.e., 74.9%). Error bars show standard deviation.***

camera's location. This breaks down the widely-accepted assumption in traditional ML that samples are independent and identically distributed (i.i.d.) from a data distribution.

We analyze four real-world datasets for CV (OpenImage [3]) and NLP (StackOverflow [9], Reddit [8] and Google Speech [72]) tasks. Each consists of thousands or up to millions of clients and millions of data points (details in Appendix A). In each individual dataset, we see a high statistical deviation across clients not only in the quantity of samples (Figure 1(a)) but also in the data distribution (Figure 1(b)).[2]

Heterogeneous system performance. As individual data samples are tightly coupled with the participant device, in-situ computation on this data experiences significant heterogeneity in system performance. We analyze the inference latency of MobileNet [66] across hundreds of mobile phones used in a real-world FL deployment [76], and their available bandwidth (details in Appendix A). Unlike the homogeneous setting in datacenter ML, system performance across clients exhibits an order-of-magnitude difference in both computational capabilities (Figure 2(a)) and network bandwidth (Figure 2(b)).

Enormous population and pervasive uncertainty. While traditional ML runs in a well-managed cluster with a number of machines, federated learning often involves up to millions of clients, making it challenging for the coordinator to efficiently identify and manage valuable participants. During execution, devices often vary in system performance [20, 45] – they may slow down or drop out – and the model performance varies in FL training as the model updates over rounds.

Privacy concerns. Inquiring the privacy-sensitive information of clients (e.g., raw data or even data distribution) can alienate participants in contributing to FL [26, 67, 68]. Hence, realistic FL solutions have to seek efficiency improvements but with limited information available in practical FL, and their deployments must be non-intrusive to clients.

2.3 Limitations of Existing FL Solutions

While existing FL solutions have made considerable progress in tackling some of the above challenges (§8), they mostly rely on hindsight – given a pool of participants, they optimize model performance [50, 61] or system efficiency [56] to tackle data and system heterogeneity. However, the potential for curbing these disadvantages by cherry-picking participants before execution has largely been overlooked. For example, FL training and testing today still rely on randomly picking participants [20], which leaves large room for improvements.

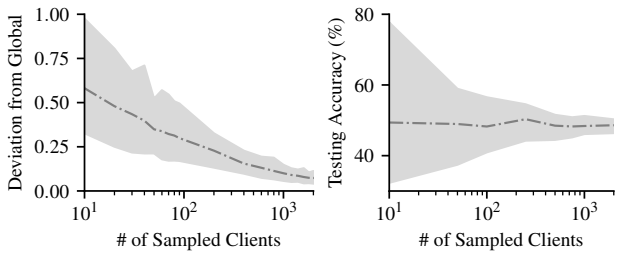
Suboptimality in maximizing efficiency. We first show that today's participant selection underperforms for FL solutions. Here, we train two popular image classification models tailored for mobile devices (i.e., MobileNet [66] and ShuffleNet [78]) with 1.4 million images of the OpenImage dataset, and randomly pick 100 participants out of more than 14k clients in each training round. We consider a performance *upper bound* by creating a hypothetical centralized case where images are evenly distributed across only 100 clients, and train on all 100 clients in each round. As shown in Figure 3, even with state-of-the-art optimizations, such as YoGi [64] and Prox [49],[3] the round-to-accuracy and final model accuracy are both far from the upper-bound. Moreover, overlooking the system heterogeneity can elongate each round, further exacerbating the suboptimality of time-to-accuracy performance.

Inability to enforce data selection criteria. While an FL developer often fine-tunes her model by understanding the input dataset, existing solutions do not provide any systems support for her to express and reason about what data her FL model was trained or tested on. Even worse, existing participant selection not only inflates the execution, but can lead to bias and loss of confidence in results [22, 39].

To better understand how existing works fall short, we take the global categorical distribution as an example requirement, and experiment with the above pre-trained ShuffleNet model. Figure 4(a) shows that: (i) even for the same number of participants, random selection can result in noticeable data deviations from the target distribution; (ii) while this deviation decreases as more participants are involved, it is non-trivial to quantify how it varies with different number of participants, even if we ignore the cost of enlarging the participant set. One

[2] We report the pairwise deviation of categorical distributions between two clients, using the popular L1-divergence metric [57].

[3] These two adapt traditional stochastic gradient descent algorithms to tackle the heterogeneity of the client datasets.



(a) Data deviation vs. participant size. (b) Accuracy vs. participant size.

Figure 4: ***Participant selection today leads to (a) deviations from developer requirements, and thus (b) affects testing result. Shadow indicates the [min, max] range of y-axis values over 1000 runs given the same x-axis input; each line reports the median.***

natural effect of violating developer specification is bias in results (Figure 4(b)), where we test the accuracy of the same model on these participants. We observe that a biased testing set results in high uncertainties in testing accuracy.

3 Oort Overview

Oort improves FL training and testing performance by judiciously selecting participants while enabling FL developers to specify data selection criteria. In this section, we provide an overview of how Oort fits in the FL life cycle to help the reader follow the subsequent sections.

3.1 Architecture

At its core, Oort is a participant selection framework that identifies and cherry-picks valuable participants for FL training and testing. It is located inside the coordinator of an FL framework and interacts with the driver of an FL execution (e.g., PySyft [7]). Given developer-specified criteria, it responds with a list of participants, whereas the driver is in charge of initiating and managing execution on the Oort-selected remote participants.

Figure 5 shows how Oort interacts with the developer and FL execution frameworks. ① *Job submission*: the developer submits and specifies the participant selection criteria to the FL coordinator in the cloud. ② *Participant selection*: the coordinator enquires the clients meeting eligibility properties (e.g., battery level), and forwards their characteristics (e.g., liveness) to Oort. Given the developer requirements (and execution feedbacks in case of training (2a)), Oort selects participants based on the given criteria and notifies the coordinator of this participant selection ((2b)). ③ *Execution*: the coordinator distributes relevant profiles (e.g., model) to these participants, and then each participant independently computes results (e.g., model weights in training) on her data; ④ *Aggregation*: when participants complete the computation, the coordinator aggregates updates from participants.

During federated training, where the coordinator initiates the next training round after aggregating updates from enough number of participants [20], it iterates over ②-④ in each round. Every a few training rounds, federated testing is often

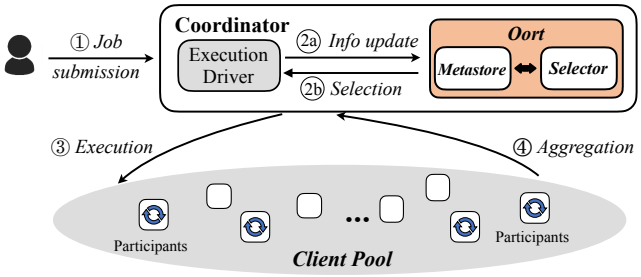


Figure 5: ***Oort architecture. The driver of the FL framework interacts with Oort using a client library.***

used to detect whether the cut-off accuracy has been reached.

3.2 Oort Interface

Oort employs two distinct selectors that developers can access via a client library during FL training and testing. We explain all Oort APIs in Appendix B.

Training selector. This selector aims to improve the time-to-accuracy performance of federated training. To this end, it captures the utility of clients in training, and efficiently explores and selects high-utility clients at runtime.

```
1  import Oort
2
3  def federated_model_training():
4    selector = Oort.create_training_selector(config)
5
6    # Train to target testing accuracy
7    while federated_model_testing() < target:
8
9      # Train 50 rounds before testing
10     for _ in range(50):
11       # Collect feedbacks of last round
12       feedbacks = engine.get_participant_feedback()
13
14       # Update the utility of clients
15       for clientId in feedbacks:
16         selector.update_client_util(
17                     clientId, feedbacks[clientId])
18
19       # Pick 100 high-utility participants
20       participants = selector.select_participant(100)
21       ... # Activate training on remote clients
```

Figure 6: ***Code snippet of Oort interaction during FL training.***

Figure 6 presents an example of how FL developers and frameworks interact with Oort during training. In each training round, Oort collects feedbacks from the engine driver, and updates the utility of individual clients (Line 15-17). Thereafter, it cherry-picks high-utility clients to feed the underlying execution (Line 20). We elaborate more on client utility and the selection mechanism in Section 4.

Testing selector. This selector currently supports two types of selection criteria. When the individual client data characteristics (e.g., categorical distribution) are not provided, the testing selector determines the number of participants needed to cap the data deviation of participants from the global. Otherwise, it cherry-picks participants to serve the exact developer-specified requirements on data while minimizing the duration of testing. We elaborate more on selection

for federated testing in Section 5.

4 Federated Model Training

In this section, we first outline the trade-off in selecting participants for FL training (§4.1), and then describe how Oort quantifies the client utility while respecting privacy (§4.2 and §4.3), how it selects high-utility clients at scale despite staleness in client utility as training evolves (§4.4).

4.1 Tradeoff Between Statistical and System Efficiency

Time-to-accuracy performance of FL training relies on two aspects: (i) *statistical efficiency*: the number of rounds taken to reach target accuracy; and (ii) *system efficiency*: the duration of each training round. The data stored on the client and the speed with which it can perform training determine its utility with respect to statistical and system efficiency, which we respectively refer to as statistical and system utility.

Due to the coupled nature of client data and system performance, cherry-picking participants for better time-to-accuracy performance requires us to jointly consider both forms of efficiency. We visualize the trade-off between these two with our breakdown experiments on the MobileNet model with OpenImage dataset (§7.2.1). As shown in Figure 7, while optimizing the system efficiency (i.e., "Opt-Sys. Efficiency") can reduce the duration of each round (e.g., picking the fastest clients), it can lead to more rounds than random selection as that client data may have already been overrepresented by other participants over past rounds. On the other hand, using a client with high statistical utility (i.e., "Opt-Stat. Efficiency") may lead to longer rounds if that client turns out to be the system bottleneck in global model aggregation.

Challenges. To improve time-to-accuracy performance, Oort aims to find a sweet spot in the trade-off by associating with every client its *utility* toward optimizing each form of efficiency (Figure 7). This leads to three challenges:

- In each round, how to determine which clients' data would help improve the statistical efficiency of training the most while respecting client privacy (§4.2)?
- How to take a client's system performance into account to optimize the global system efficiency (§4.3)?
- How to account for the fact that we don't have up-to-date utility values for all clients during training (§4.4)?

Next, we integrate system designs with ML principles to tackle the heterogeneity, the massive scale, the runtime uncertainties and privacy concerns of clients for practical FL.

4.2 Client Statistical Utility

An ideal design of statistical utility should be able to efficiently capture the client data utility toward improving model performance for various training tasks, and respect privacy.

To this end, we leverage importance sampling [47, 79]. Say each client i has a bin B_i of training samples locally stored. Then, to improve the round-to-accuracy performance

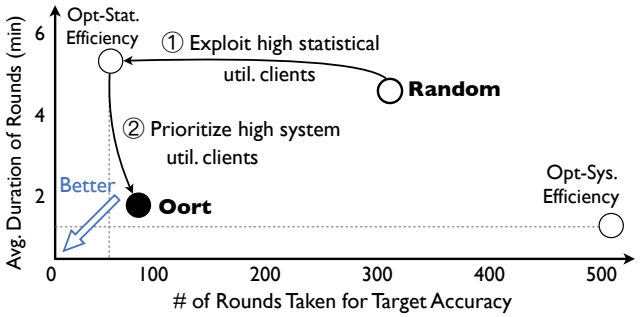


Figure 7: ***Existing FL training randomly selects participants, whereas Oort navigates the sweet point of statistical and system efficiency to optimize their circled area (i.e., time to accuracy). Numbers are from the MobileNet on OpenImage dataset (§7.2.1).***

via importance sampling, the optimal solution would be to pick bin B_i with a probability proportional to its importance $|B_i|\sqrt{\frac{1}{|B_i|}\sum_{k\in B_i} \| \nabla f(k) \|^2}$, where $\| \nabla f(k) \|$ is the L2-norm of the unique sample k's gradient $\nabla f(k)$ in bin B_i. Intuitively, this means selecting the bin with larger aggregate gradient norm across all of its samples.

However, taking this importance as the statistical utility is impractical, since it requires an extra time-consuming pass over the client data to generate the gradient norm of every sample,[4] and this gradient norm varies as the model updates.

To avoid extra cost, we introduce a pragmatic approximation of statistical utility instead. At its core, the gradient is derived by taking the derivative of training loss with respect to current model weights, wherein training loss measures the estimation error between model predictions and the ground truth. Our insight is that a larger gradient norm often attributes to a bigger loss [44]. Therefore, we define the statistical utility $U(i)$ of client i as $U(i) = |B_i|\sqrt{\frac{1}{|B_i|}\sum_{k\in B_i} Loss(k)^2}$, where the training loss $Loss(k)$ of sample k is automatically generated during training with negligible collection overhead. As such, we consider clients that currently accumulate a bigger loss to be more important for future rounds.

Our statistical utility can capture the heterogeneous data utility across and within categories and samples for various tasks. We present the theoretical proof for its effectiveness over random sampling in Appendix C, and empirically show its close-to-optimal performance (§7.2.2).

How Oort respects privacy? Training loss measures the prediction confidence of a model without revealing the raw data and is often collected in real FL deployments [35, 76]. We further provide three ways to respect privacy. First, we rely on *aggregate* training loss, which is computed locally by the client across *all* of her samples without revealing the loss distribution of individual samples either. Second, when even the aggregate loss raises a privacy concern, clients can add noise to their loss value before uploading, like in existing

[4] ML models generate the training loss of each sample during training, but calculate the gradient of the mini-batch instead of individual samples.

differentially private FL [31]. Third, we later show that Oort can flexibly accommodate other definitions of statistical utility used in our generic participant selection framework (§4.4). We provide detailed analyses for each strategy (e.g., using gradient norm of batches) of how Oort can respect privacy (e.g., amenable under noisy utility value) while improving performance in Appendix D, theoretically and empirically.

4.3 Trading off Statistical and System Efficiency

Simply selecting clients with high statistical utility can hamper the system efficiency. To reconcile the demand for both efficiencies, we should maximize the statistical utility we can achieve per unit time (i.e., the product of statistical utility and its system speed). As such, we formulate the utility of client i by associating her statistical utility with a global system utility in terms of the duration of each training round:

$$Util(i) = \underbrace{|B_i|\sqrt{\frac{1}{|B_i|}\sum_{k\in B_i} Loss(k)^2}}_{\text{Statistical utility } U(i)} \times \underbrace{\left(\frac{T}{t_i}\right)^{\mathbb{1}(T<t_i)\times\alpha}}_{\text{Global sys utility}} \quad (1)$$

where T is the developer-preferred duration of each round, t_i is the amount of time that client i takes to process the training, which has already been collected by today's coordinator from past rounds,[5] and $\mathbb{1}(x)$ is an indicator function that takes value 1 if x is true and 0 otherwise. This way, the utility of those clients who may be the bottleneck of the desired speed of current round will be penalized by a developer-specified factor α, but we do not reward the non-straggler clients because their completions do not impact the round duration.

This formulation assumes that all samples at a client are processed in that training round. Even if the estimated t_i for a client is greater than the desired round duration T, Oort might pick that client if the statistical utility outweighs its slow speed. Alternatively, if the developer wishes to cap every round at a certain duration [56], then either only clients with $t_i < T$ can be considered (e.g., by setting $\alpha \to \infty$) or a subset of a participant's samples can be processed [49, 64].

Navigating the trade-off. Determining the preferred round duration T in Equation (1), which strikes the trade-off between the statistical and system efficiency in aggregations, is non-trivial. Indeed, the total statistical utility (i.e., $\sum U(i)$) achieved by picking high utility clients can decrease round by round, because the training loss decreases as the model improves over time. If we persist in suppressing clients with high statistical utility but low system speed, the model may converge to suboptimal accuracy (§7.2.2).

To navigate the optimal trade-off – maximizing the total statistical utility achieved without greatly sacrificing the system efficiency – Oort employs a pacer to determine the preferred duration T at runtime. The intuition is that, when the accumulated statistical utility in the past rounds decreases, the pacer allows a larger $T \leftarrow T + \Delta$ by Δ to bargain with the statistical efficiency again. We elaborate more in Algorithm 1.

4.4 Adaptive Participant Selection

Given the above definition of client utility, we need to address the following practical concerns in order to select participants with the highest utility in each training round.

- *Scalability*: a client's utility can only be determined after it has participated in training; how to choose from clients at scale without having to try all clients once?
- *Staleness*: since not every client participates in every round, how to account for the change in a client's utility since its last participation?
- *Robustness*: how to be robust to outliers in the presence of corrupted clients (e.g., with noisy data)?

To tackle these challenges, we develop an exploration-exploitation strategy for participant selection (Algorithm 1).

Online exploration-exploitation of high-utility clients. Selecting participants out of numerous clients can be modeled as a multi-armed bandit problem, where each client is an "arm" of the bandit, and the utility obtained is the "reward." To maximize the long-term reward, we can adaptively balance the exploration and exploitation of different arms [15].

Similar to the bandit design, Oort efficiently explores potential participants under spatial variation, while intelligently exploiting observed high-utility participants under temporal variation.[6] At the beginning of each selection round, Oort receives the feedback of the last training round, and updates the statistical utility and system performance of clients (Line 6). For the explored clients, Oort calculates their client utility and narrows down the selection by exploiting the high-utility participants (Line 9-15). Meanwhile, Oort samples $\varepsilon(\in [0, 1])$ fraction of participants to explore potential participants that had not been selected before (Line 16), which turns to full exploration as $\varepsilon \to 1$. Although we cannot learn the statistical utility of not-yet-tried clients, one can decide to prioritize the unexplored clients with faster system speed when possible (e.g., by inferring from device models), instead of performing random exploration (Line 16).

Exploitation under staleness in client utility. Oort employs two strategies to account for the dynamics in client utility over time. First, motivated by the confidence interval used to measure the uncertainty in bandit reward, we introduce an incentive term, which shares the same shape of the confidence in bandit solutions [42], to account for the staleness (Line 10), whereby we gradually increase the utility of a client if she has been overlooked for a long time. So those clients accumulating high utility since their last trial can still be rediscovered again. Second, instead of picking clients with

[5] We only care whether a client can complete by the expected duration T. So, a client can even mask its precise speed by deferring its report.

[6] We borrow from the bandit design because, in contrast to sophisticated models (e.g., reinforcement learning), it is scalable and flexible even when the solution space (e.g., number of clients) varies dramatically over time.

Input: Client set $\mathbb{C}$, sample size K, exploitation factor ε, pacer step Δ, step window W, penalty α
Output: Participant set $\mathbb{P}$

/* Initialize global variables. */
1 $\mathbb{E} \leftarrow \emptyset$; $\mathbb{U} \leftarrow \emptyset$ ▷ Explored clients and statistical utility.
2 $\mathbb{L} \leftarrow \emptyset$; $\mathbb{D} \leftarrow \emptyset$ ▷ Last involved round and duration.
3 $R \leftarrow 0$; $T \leftarrow \Delta$ ▷ Round counter and preferred round duration.

/* Participant selection for each round. */
4 **Function** SelectParticipant ($\mathbb{C}$, K, ε, T, α)
5 $Util \leftarrow \emptyset$; $R \leftarrow R+1$

/* Update and clip the feedback; blacklist outliers. */
6 UpdateWithFeedback($\mathbb{E}$, $\mathbb{U}$, $\mathbb{L}$, $\mathbb{D}$)

/* Pacer: Relaxes global system preference T if the statistical utility achieved decreases in last W rounds. */
7 **if** $\sum \mathbb{U}(R-2W : R-W) > \sum \mathbb{U}(R-W : R)$ **then**
8 $T \leftarrow T+\Delta$

/* Exploitation #1: Calculate client utility. */
9 **for** *client* $i \in \mathbb{E}$ **do**
10 $Util(i) \leftarrow \mathbb{U}(i) + \sqrt{\frac{0.1 \log R}{\mathbb{L}(i)}}$ ▷ Temporal uncertainty.
11 **if** $T < \mathbb{D}(i)$ **then** ▷ Global system utility.
12 $Util(i) \leftarrow Util(i) \times (\frac{T}{\mathbb{D}(i)})^{\alpha}$

/* Exploitation #2: admit clients with greater than c% of cut-off utility; then sample $(1-\varepsilon)K$ clients by utility. */
13 $Util \leftarrow$ SortAsc(Util)
14 $\mathbb{W} \leftarrow$ CutOffUtil($\mathbb{E}$, $c \times Util((1-\varepsilon) \times K)$)
15 $\mathbb{P} \leftarrow$ SampleByUtil($\mathbb{W}$, $Util$, $(1-\varepsilon) \times K$)

/* Exploration: sample unexplored clients by speed. */
16 $\mathbb{P} \leftarrow \mathbb{P} \cup$ SampleBySpeed($\mathbb{C} - \mathbb{E}$, $\varepsilon \times K$)
17 **return** $\mathbb{P}$

Alg. 1: Participant selection w/ exploration-exploitation.

top-k utility deterministically, we allow a confidence interval c on the cut-off utility (95% by default in Line 13-14). Namely, we admit clients whose utility is greater than the c% of the top $((1-\varepsilon) \times K)$-th participant. Among this high-utility pool, Oort samples participants with probability proportional to their utility (Line 15). This adaptive exploitation mitigates the uncertainties in client utility by prioritizing participants opportunistically while preserving a high quality as a whole.

Robust exploitation under outliers. Simply prioritizing high utility clients can be vulnerable to outliers in unfavorable settings. For example, corrupted clients may have noisy data, leading to high training loss, or even report arbitrarily high training loss intentionally. For robustness, Oort (i) removes the client in selection after she has been picked over a given number of rounds. This helps to remove the perceived outliers in terms of participation (Line 6); (ii) clips the utility value of a client by capping it to no more than an upper bound (e.g., 95% value in utility distributions). With probabilistic participant selection among the high-utility client pool (Line 15), the chance of selecting outliers is significantly decreased under the scale of clients in FL. We show that Oort outperforms existing mechanisms while being robust (§7.2.3).

```
def federated_model_testing():
  selector = Oort.create_testing_selector()

  # Type 1: subset w/ < X deviation from the global
  participants = selector.select_by_deviation(
    dev_target, range_of_capacity, total_num_clients)

  # Provide individual client data characteristics
  selector.update_client_info(client_id, client_info)
  # Type 2: [5k, 5k] samples of category [i, j]
  participants = selector.select_by_category(
                 request_list, testing_config)
```

Figure 8: ***Key Oort APIs for supporting federated testing.***

Accommodation to diverse selection criteria. Our adaptive participant selection is generic for different utility definitions of diverse selection criteria. For example, developers may hope to reconcile their demand for time-to-accuracy efficiency and fairness, so that some clients are not underrepresented (e.g., more fair accuracy distribution or resource usage across clients) [45, 50]. Indeed, we show that Oort can achieve a more fair accuracy distribution across clients by prioritizing high-loss clients than the existing (Appendix E). More subtly, although developers may have various fairness criterion $fairness(\cdot)$, Oort can enforce their demands by replacing the current utility definition of client i with $(1-f) \times Util(i) + f \times fairness(i)$, where $f \in [0,1]$ and Algorithm 1 will naturally prioritize clients with the largest fairness demand as $f \to 1$. For example, $fairness(i) = max\_resource\_usage - resource\_usage(i)$ motivates fair resource usage for each client i. Note that existing participant selection provides no support for fairness, whereas we show that Oort can efficiently enforce diverse developer-preferred fairness while improving performance in Appendix E.

5 Federated Model Testing

Enforcing developer-defined requirements on data distribution is a first-order goal in FL testing, whereas existing mechanisms lead to biased testing results (§2.3). In this section, we elaborate on how Oort serves the two primary types of queries. As shown in Figure 8, we start with how Oort preserves the representativeness of testing set even without individual client data characteristics (§5.1), and how it efficiently enforces developer's testing criteria for specific data distribution when the individual information is provided (§5.2).

5.1 Preserving Data Representativeness

Learning the individual data characteristics (e.g., categorical distribution) can be too expensive or even prohibited [29, 65].

Without knowing data characteristics, the developer has to be conservative and selects many participants to gain more confidence for query *"a testing set with less than X% data deviation from the global"*, as selecting too few can lead to a biased testing result (§2.3). However, admitting too many may inflate the budget and/or take too long because of the system heterogeneity. Next, we show how Oort can enable guided participant selection by determining the number of participants needed to guarantee this deviation target.

We consider the deviation of the data formed by all participants from the global dataset (i.e., representative) using L1-distance, a popular distance metric in FL [39, 40, 59]. For category X, its L1-distance ($|\bar{X} - E[\bar{X}]|$) captures how the average number of samples of all participants (i.e., empirical value $\bar{X}$) deviates from that of all clients (i.e., expectation $E[\bar{X}]$). As the number of samples X_n that client n holds is independent across clients,[7] it can be viewed as a random instance sampled from the distribution of variable X.

Given the developer-specified tolerance ε on data deviation and confidence interval δ (95% by default [58]), our goal is to estimate the number of participants needed such that the deviation from the representative categorical distribution is bounded (i.e., $Pr[|\bar{X} - E[\bar{X}]| < \varepsilon] > \delta$). To this end, we formulate it as a problem of sampling stochastic variables, and apply the concentration theory [17] to capture how this data deviation varies with different number of participants. We attach our detailed results and proof in Appendix F.

Estimating the number of participants to cap deviation. Even when the individual data characteristics are not available, the developer can specify her tolerance ε on the deviation from the global categorical distribution, whereby Oort outputs the number of participants needed to preserve this preference. To use our model, the developer needs to input the global range (i.e., global maximum - global minimum) of the number of samples that one client can hold, and the total number of clients. Learning this global information securely is well-established [25, 65], and the developer can assume a plausible limit (e.g., according to the capacity of device models) too.

Our model does not require any collection of the distribution of global or participant data. As a straw-man participant selection design, the developer can randomly distribute her model to this Oort-determined number of participants. After collecting results from this number of participants, she can confirm the representativeness of computed data.

5.2 Enforcing Diverse Data Distribution

When the individual data characteristics are provided (e.g., FL across enterprise AI cameras [40, 53]), Oort can enforce the exact data preference on specific categorical distribution, and improve the duration of testing by cherry-picking participants.

Satisfying queries like *"[5k, 5k] samples of class [x, y]"* can be viewed as a multi-dimensional bin covering problem, where a subset of data bins (i.e., participants) are selected to cover the requested quantity of data. We first formulate a mixed-integer linear programming formulation (MILP) to decide the optimal participant selection subject to the preference and budget constraints, while optimizing the testing duration by accounting for system heterogeneity (see Appendix G).

Scalable participant selection. For better scalability, we present a greedy heuristic to scale down the search space of this strawman. We (1) first group a subset of feasible clients to satisfy the preference constraint. To this end, we iteratively add to our subset the client which has the most number of samples across all not-yet-satisfied categories, and deduct the preference constraint on each category by the corresponding capacity of this client. We stop this greedy grouping until the preference is met, or request a new budget if we exceed the budget; and (2) then optimize job duration with a simplified MILP among this subset of clients, wherein we have removed the budget constraint and reduced the search space of clients. We show that our heuristic can outperform the straw-man MILP model in terms of the end-to-end duration of model testing owing to its small overhead (§7.3.2).

6 Implementation

We have implemented Oort as a Python library, with 2617 lines of code, to friendly support FL developers. Oort provides simple APIs to abstract away the problem of participant selection, and developers can import Oort in their application codebase and interact with FL engines (e.g., PySyft [7] or TensorFlow Federated [11]). Readers can refer to Appendix B for detailed APIs and corresponding use cases.

We have integrated Oort with PySyft. Oort operates on and updates its client metadata (e.g., data distribution or system performance) fed by the FL developer and PySyft at runtime. The metadata of each client in Oort is an object with a small memory footprint. Oort caches these objects in memory during executions and periodically backs them up to persistent storage. In case of failures, the execution driver will initiate a new Oort selector, and load the latest checkpoint to catch up. We employ Gurobi solver [5] to solve the MILP. The developer can also initiate a Oort application beyond coordinators to avoid resource contention. We use *xmlrpc* library to connect to the coordinator, and updates from the coordinator will activate Oort to write these updates to its metastore. In the coordinator, we use the PySyft API *model.send(client_id)* to direct which client to run given the Oort decision, and *model.get(client_id)* to collect the feedback.

7 Evaluation

We evaluate Oort's effectiveness for four different ML models on four CV and NLP datasets.[8] We organize our evaluation by the FL activities with the following key results.

[7] The number of samples that one client holds will not be affected by the selection of any other clients at that time.

[8] We will make Oort implementation and the workloads open-source.

Task	Dataset	Accuracy Target	Model	Speedup for Prox [49]			Speedup for YoGi [64]		
				Stats.	Sys.	Overall	Stats.	Sys.	Overall
Image Classification	OpenImage-Easy [3]	74.9%	MobileNet [66]	3.8×	3.2×	12.1×	2.4×	2.4×	5.7×
			ShuffleNet [78]	2.5×	3.5×	8.8×	1.9×	2.7×	5.1×
	OpenImage [3]	53.1%	MobileNet	4.2×	3.1×	13.0×	2.3×	1.5×	3.3×
			ShuffleNet	4.8×	2.9×	14.1×	1.8×	3.2×	5.8×
Language Modeling	Reddit [8]	39 perplexity	Albert [48]	1.3×	6.4×	8.4×	1.5×	4.9×	7.3×
	StackOverflow [9]	39 perplexity	Albert	2.1×	4.3×	9.1×	1.8×	4.4×	7.8×
Speech Recognition	Google Speech [72]	62.2%	ResNet-34 [36]	1.1×	1.1×	1.2×	1.2×	1.1×	1.3×

Table 1: ***Summary of improvements on time to accuracy. We tease apart the overall improvement with statistical and system ones, and take the highest accuracy that Prox can achieve as the target, which is moderate due to the high task complexity and lightweight models.***

FL training results summary:

- Oort outperforms existing random participant selection by 1.2×-14.1× in time-to-accuracy performance, while achieving 1.3%-9.8% better final model accuracy (§7.2.1).
- Oort achieves close-to-optimal model efficiency by adaptively striking the trade-off between statistical and system efficiency with different components (§7.2.2).
- Oort outperforms its counterpart over a wide range of parameters and different scales of experiments, while being robust to outliers (§7.2.3).

FL testing results summary:

- Oort can serve developer testing criteria on data deviation while reducing costs by bounding the number of participants needed even without individual data characteristics (§7.3.1).
- With the individual information, Oort improves the testing duration by 4.7× w.r.t. Mixed Integer Linear Programming (MILP) solver, and is able to efficiently enforce developer preferences across millions of clients (§7.3.2).

7.1 Methodology

Experimental setup. Oort is designed to operate in large deployments with potentially millions of edge devices. However, such a deployment is not only prohibitively expensive, but also impractical to ensure the reproducibility of experiments. As such, we resort to a cluster with 68 NVIDIA Tesla P100 GPUs, and emulate up to 1300 participants in each round. We provide details of our simulation platform and used profiles in Appendix A. Clients are running with PySyft [7] using PyTorch v1.5.0 backend. We emulate the *heterogeneous* system performance with hundreds of profiles of end-device computational capacity and network bandwidth, and use a large-scale real-world user behavior dataset [75] to emulate the dynamics of client availability over time. To mitigate stragglers in each training round, we employ the widely-used mechanism specified in real FL deployments [20], where we collect updates from the first K completed participants out of $1.3K$ participants, and K is 100 by default.

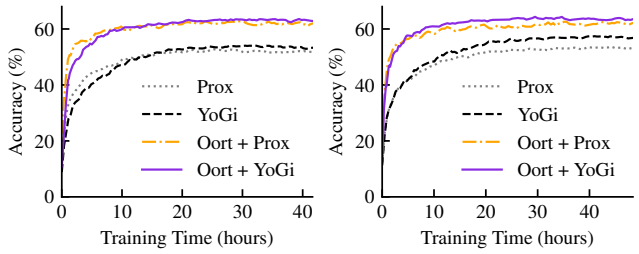
Datasets and models. We run three categories of applications with four real-world datasets of different scales:

- *Speech Recognition*: the small-scale Google speech dataset [72], with 105k speech commands over 3k clients. We train a convolutional neural network model (ResNet-34 [36]) to recognize the command among 35 categories.
- *Image Classification*: the middle-scale OpenImage [3] dataset, with 1.5 million images spanning 600 categories across 14k clients, and a simpler dataset (OpenImage-Easy) with images from the most popular 60 categories. We train MobileNet [66] and ShuffleNet [78] models to classify the image.
- *Language Modeling*: the large-scale StackOverflow [9] and Reddit [8] dataset, with 0.3 and 1.6 million clients respectively. We train next word predictions with Albert.

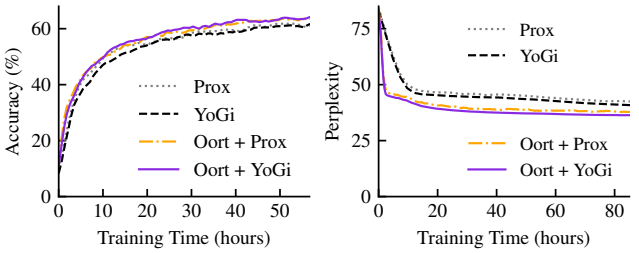
These applications are widely used in real end-device applications [73], and these models are designed to be lightweight.

Parameters. The minibatch size of each participant is 16 in speech recognition, and 32 in other tasks. The initial learning rate for Albert model is 4e-5, and 0.04 for other models. These configurations are consistent with those reported in the literature [34]. In configuring the training selector, Oort uses the popular time-based exploration factor [15], where the initial exploration factor is 0.9, and decreased by a factor 0.98 after each round when it is larger than 0.2. The step window of pacer W is 20 rounds. We set the pacer step Δ in a way that it can cover the duration of next $W \times K$ clients in the descending order of explored clients' duration, and the straggler penalty α to 2. We remove a client from Oort's exploitation list once she has been selected over 10 times.

Metrics. We care about the *time-to-accuracy* performance and *final model accuracy* for model training tasks. For model testing, we measure the *end-to-end* testing duration, which consists of the computation overhead of the solution and the duration of actual computation.



(a) MobileNet (Image). (b) ShuffleNet (Image).



(c) ResNet (Speech). (d) Albert (LM).

Figure 9: ***Time-to-Accuracy performance. A lower perplexity is better in the language modeling (LM) task.***

For each experiment, we report the mean value over 5 runs, and error bars show the standard deviation.

7.2 FL Training Evaluation

In this section, we evaluate Oort's performance on model training, and employ Prox [49] and YoGi [64]. We refer Prox as Prox running with existing random participant selection, and Prox + Oort is Prox running atop Oort. We use a similar denotation for YoGi. Note that Prox and YoGi optimize the statistical model efficiency for the given participants, while Oort cherry-picks participants to feed them.

7.2.1 End-to-End Performance

Table 1 summarizes the key time-to-accuracy performance of all datasets. In the rest of the evaluations, we report the ShuffleNet and MobileNet performance on OpenImage, and Albert performance on Reddit dataset for brevity. Figure 9 reports the timeline of training to achieve different accuracy.

Oort improves time-to-accuracy performance. We notice that Oort achieves large speedups to reach the target accuracy (Table 1). Oort reaches the target 3.3×-14.1× faster in terms of wall clock time on the middle-scale OpenImage dataset; speedup on the large-scale Reddit and StackOverflow dataset is 7.3×-9.1×. Understandably, these benefits decrease when the total number of clients is small, as shown on the small-scale Google Speech dataset (1.2×-1.3×).

These time-to-accuracy improvements stem from the comparable benefits in statistical model efficiency and system efficiency (Table 1). Oort takes 1.8×-4.8× fewer training rounds on OpenImage dataset to reach the target accuracy, which is better than that of language modeling tasks (1.3×-2.1×). This is because real-life images often exhibit greater

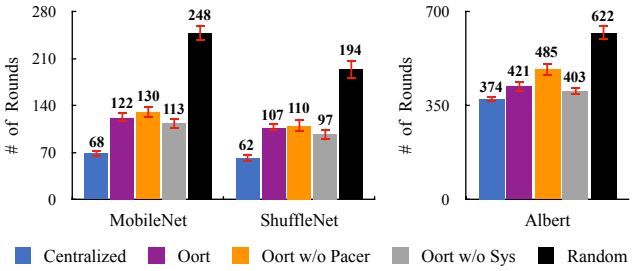


Figure 10: ***Number of rounds to reach the target accuracy.***

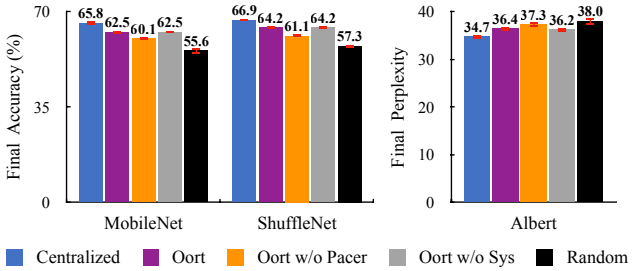


Figure 11: ***Breakdown of final model accuracy.***

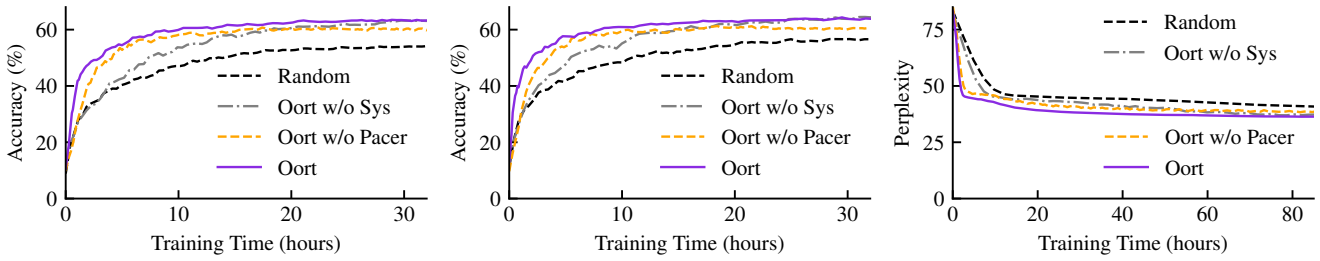
heterogeneity in data characteristics than the language dataset, whereas the large population of language datasets leaves a great potential to prioritize clients with faster system speed.

Oort improves final model accuracy. When the model converges, Oort achieves 6.6%-9.8% higher final accuracy on OpenImage dataset, and 3.1%-4.4% better perplexity on Reddit dataset (Figure 9). Again, this improvement on Google Speech dataset is smaller (1.3% for Prox and 2.2% for YoGi) due to the small scale of clients. These improvements attribute to the exploitation of high statistical utility clients. Specifically, the statistical model accuracy is determined by the quality of global aggregation. Without cherry-picking participants in each round, clients with poor statistical model utility can dilute the quality of aggregation. As such, the model may converge to suboptimal performance. Instead, models running with Oort concentrate more on clients with high statistical utility, thus achieving better final accuracy.

7.2.2 Performance Breakdown

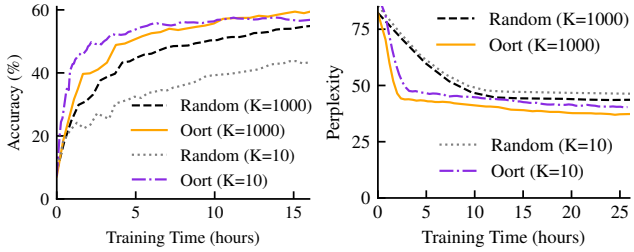
We next delve into the improvement on middle- and large-scale datasets, as they are closer to real FL deployments. We break down our knobs designed for striking the balance between statistical and system efficiency: (i) (*Oort w/o Pacer*): We disable the pacer that guides the aggregation efficiency. As such, it keeps suppressing low-speed clients, and the training can be restrained among low-utility but high-speed clients; (ii) (*Oort w/o Sys*): We further totally remove our benefits from system efficiency by setting α to 0, so Oort blindly prioritizes clients with high statistical utility. We take YoGi for analysis, because it outperforms Prox most of the time.

Breakdown of time-to-accuracy efficiency. Figure 12 reports the breakdown of time-to-accuracy performance, where Oort achieves comparable improvement from statistical and



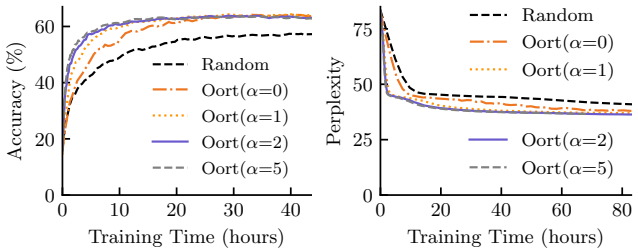
(a) MobileNet (Image). (b) ShuffleNet (Image). (c) Albert (LM).

Figure 12: ***Breakdown of Time-to-Accuracy performance with YoGi, when using different participant selection strategies.***



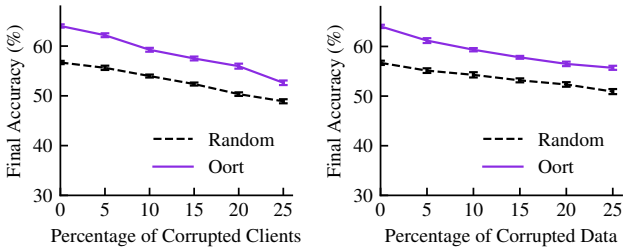
(a) ShuffleNet (Image). (b) Albert (LM).

Figure 13: ***Oort outperforms in different scales of participants.***



(a) ShuffleNet (Image). (b) Albert (LM).

Figure 14: ***Oort improves performance across penalty factors.***



(a) Corrupted clients. (b) Corrupted data.

Figure 15: ***Oort still improves performance under outliers.***

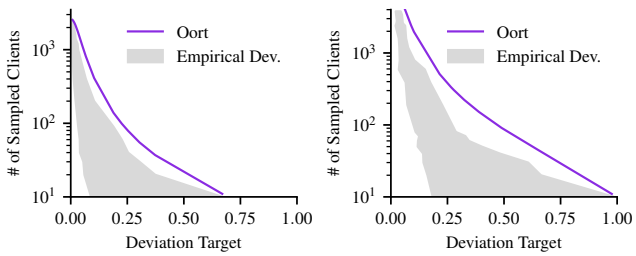
system optimizations. Taking Figure 12(b) as example, (i) At the beginning of training, both Oort and (Oort w/o Pacer) improve the model accuracy quickly, because they penalize the utility of stragglers and select clients with higher statistical utility and system efficiency. In contrast, (Oort w/o Sys) only considers the statistical utility, resulting in longer rounds. (ii) As training evolves, the pacer in Oort gradually relaxes the constraints on system efficiency, and admits clients with relatively low speed but higher statistical utility, which ends up with the similar final accuracy of (Oort w/o Sys). However, (Oort w/o Pacer) relies on a fixed system constraint and suppresses valuable clients with high statistical utility but low speed, leading to suboptimal final accuracy.

Breakdown of statistical model efficiency. We consider an *upper-bound* statistical efficiency by creating a centralized case, where all data are evenly distributed to K participants. Using the target accuracy in Table 1, Oort can efficiently approach this upper bound by incorporating different components (Figure 10). Oort is within 2× of the upper-bound to achieve the target accuracy, and (Oort w/o Sys) performs the best in statistical model efficiency, because (Oort w/o Sys) always grasps clients with higher statistical utility. However, it is suboptimal in our targeted time-to-accuracy performance because of ignoring the system efficiency. Moreover, by introducing the pacer, Oort achieves 2.4%-3.1% better accuracy than (Oort w/o Pacer), and is merely about 2.7%-3.3% worse than the upper-bound final model accuracy (Figure 11).

7.2.3 Sensitivity Analysis

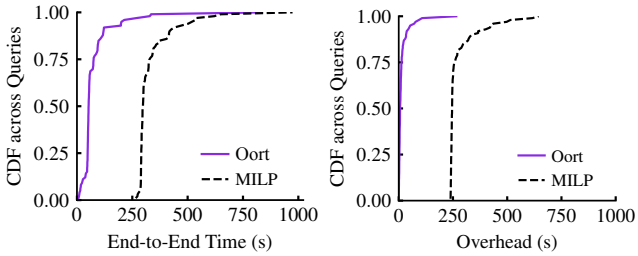
Impact of number of participants K. We evaluate Oort across different scales of participants in each round, where we cut off the training after 200 rounds given the diminishing rewards. We observe that Oort improves time-to-accuracy efficiency across different number of participants (Figure 13), and having more participants in FL indeed receives diminishing rewards. This is because taking more participants (i) is similar to having a large batch size, which is confirmed to be even negative to round-to-accuracy performance [52]; (ii) can lead to longer rounds due to stragglers when the number of clients is limited (e.g., K=1000 on OpenImage dataset).

Impact of penalty factor α on stragglers. Oort uses the penalty factor α to penalize the utility of stragglers in participant selection, whereby it adaptively prioritizes high system efficiency participants. Figure 14 shows that Oort achieves persistent improvements and outperforms its counterparts on different models across different non-zero penalty factors.



(a) Google Speech. (b) Reddit.

Figure 16: ***Oort can cap data deviation for all targets. Shadow indicates the empirical [min, max] range of the x-axis values over 1000 runs given the y-axis input.***



(a) OpenImage (Testing duration). (b) OpenImage (Overhead).

Figure 17: ***Oort outperforms MILP in clairvoyant FL testing.***

Impact of outliers. We investigate the robustness of Oort by introducing outliers manually. Following the popular adversarial ML setting [30], we randomly flip the ground-truth data labels of the OpenImage dataset to any other categories, resulting in artificially high utility. We consider two practical scenarios with the ShuffleNet model: (i) Corrupted clients: labels of all training samples on these clients are flipped (Figure 15(a)); (ii) Corrupted data: each client uniformly flips a subset of her training samples (Figure 15(b)). We notice Oort still outperforms across all degrees of corruption.

7.3 FL Testing Evaluation

7.3.1 Preserving Data Representativeness

Oort can cap data deviation. Figure 16 reports Oort's performance on serving different deviation targets, with respect to the global distribution. We sweep the number of selected clients from 10 to 4k, and randomly select each given number of participants over 1k times to empirically search their possible deviation. We notice that for a given deviation target, (i) different workloads require distinct number of participants. For example, to meet the target of 0.05 divergence, the Speech dataset uses 6× less participants than the Reddit attributing to its smaller heterogeneity (e.g., tighter range of the number of samples); (ii) with the Oort-determined number of participants, no empirical deviation exceeds the target, showing the effectiveness of Oort in satisfying the deviation target, whereby Oort reduces the cost of expanding participant set arbitrarily and improves the testing duration.

7.3.2 Enforcing Diverse Data Distribution

Oort is scalable and outperforms MILP. We start with the middle-scale OpenImage dataset and compare the end-to-end testing duration of Oort and MILP. Here, we generate 200 queries using the form "Give me X representative samples", where we sweep X from 4k to 200k and budget B from 100 participants to 5k participants. We report the validation time of MobileNet on participants selected by these strategies.

Figure 17(a) shows the end-to-end testing duration. We observe Oort outperforms MILP by 4.7× on average. This is because Oort suffers little computation overhead by greedily reducing the search space of MILP. As shown in Figure 17(b), MILP takes 274 seconds on average to complete the participant selection, while Oort only takes 15 seconds.

Moreover, our experiments on the large-scale dataset show that Oort can efficiently serve developer requirements in a few minutes at the scale of millions of clients (Appendix H).

8 Related Work

Federated Learning Federated learning [45] is a distributed machine learning paradigm in a network of end devices, wherein Prox [49] and YoGi [64] are state-of-the-art optimizations in tackling data heterogeneity. Recent efforts in FL have been focusing on improving communication efficiency [38, 56] or compression schemes [14], ensuring privacy by leveraging multi-party computation (MPC) [21] and differential privacy [31], or tackling heterogeneity by reinventing ML algorithms [50, 70]. However, they underperform in FL because of the suboptimal participant selection they rely on, and lack systems supports for developers to specify their participant selection criteria.

Datacenter Machine Learning Distributed ML in datacenters has been well-studied [41, 60, 62], wherein they assume relatively homogeneous data and workers [33, 54]. While developer requirements and models can still be the same, the client heterogeneity makes FL much more challenging. We aim at enabling them in FL. While bearing some resemblance in prioritizing important training samples [44, 47, 79], we consider both statistical and system efficiency at scale.

Privacy-preserving Data Analytics To gather sensitive statistics from user devices, several differentially private systems add noise to user inputs locally to ensure privacy [29], whereas some assume a trusted third party, which only adds noise to the aggregated raw inputs [19], or use MPC to enable global differential privacy without a trusted party [65]. Our work is orthogonal to them, but developers can leverage them to collect client information for the clairvoyant model testing.

9 Conclusion

In this paper, we presented Oort to enable guided participant selection for FL developers. Compared to existing participant selection mechanisms, Oort achieves large speedups in time-to-accuracy performance for federated training by picking clients with high data and system utility, and it allows

developers to specify their selection criteria on data while efficiently serving developer requirements on data distribution during testing even at the scale of millions of clients.

References

[1] AI Benchmark: All About Deep Learning on Smartphones. http://ai-benchmark.com/ranking_deeplearning_detailed.html.

[2] Federated AI Technology Enabler. https://www.fedai.org/.

[3] Google Open Images Dataset. https://storage.googleapis.com/openimages/web/index.html.

[4] Google's Sundar Pichai: Privacy Should Not Be a Luxury Good. https://www.nytimes.com/2019/05/07/opinion/google-sundar-pichai-privacy.html.

[5] Gurobi. https://www.gurobi.com/.

[6] MobiPerf. https://www.measurementlab.net/tests/mobiperf/.

[7] PySyft. https://github.com/OpenMined/PySyft.

[8] Reddit Comment Data. https://files.pushshift.io/reddit/comments/.

[9] Stack Overflow Data. https://cloud.google.com/bigquery/public-data/stackoverflow.

[10] Stanford Puffer. https://puffer.stanford.edu/.

[11] TensorFlow Federated. https://www.tensorflow.org/federated.

[12] Transformers. https://github.com/huggingface/transformers.

[13] Martín Abadi, Andy Chu, Ian Goodfellow, and et al. Deep learning with differential privacy. In *CCS*, 2016.

[14] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. QSGD: Communication-efficient sgd via gradient quantization and encoding. In *NeurIPS*, 2017.

[15] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. In *Machine Learning*, 2002.

[16] Sean Augenstein, H. Brendan McMahan, and et al. Generative models for effective ML on private, decentralized datasets. In *ICLR*, 2020.

[17] Rémi Bardenet and Odalric-Ambrym Maillard. *Concentration inequalities for sampling without replacement*. Bernoulli Society for Mathematical Statistics and Probability, 2015.

[18] Patrice Bertail, Emmanuelle Gautherat, and Hugo Harari-Kermadec. Exponential bounds for multivariate self-normalized sums. *Electron. Commun. Probab.*, 13:no. 57, 628–640, 2008.

[19] Andrea Bittau, Úlfar Erlingsson, and et al. Prochlo: Strong privacy for analytics in the crowd. In *SOSP*, 2017.

[20] Keith Bonawitz, Hubert Eichner, and et al. Towards federated learning at scale: System design. In *MLSys*, 2019.

[21] Keith Bonawitz, Vladimir Ivanov, and et al. Practical secure aggregation for privacy-preserving machine learning. In *CCS*, 2017.

[22] Eric Breck, Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. Data validation for machine learning. In *MLSys*, 2019.

[23] Mingqing Chen, Rajiv Mathews, Tom Ouyang, and Françoise Beaufays. Federated learning of out-of-vocabulary words. In *arxiv.org/abs/1903.10635*, 2019.

[24] Mingqing Chen, Ananda Theertha Suresh, Rajiv Mathews, Adeline Wong, Cyril Allauzen, Francoise Beaufays, and Michael Riley. Federated learning of n-gram language models. In *ACL*, 2019.

[25] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *NSDI*, 2017.

[26] Apple Differential Privacy Team. Learning with privacy at scale. In *Apple Machine Learning Journal*, 2017.

[27] S. Dutta, D. Wei, H. Yueksel, P. Y. Chen, S. Liu, and K. R. Varshney. Is there a trade-off between fairness and accuracy? a perspective using mismatched hypothesis testing. In *ICML*, 2020.

[28] Hubert Eichner, Tomer Koren, H. Brendan McMahan, Nathan Srebro, and Kunal Talwar. Semi-cyclic stochastic gradient descent. In *ICML*, 2019.

[29] Ulfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. RAPPOR: Randomized aggregatable privacy-preserving ordinal response. In *CCS*, 2014.

[30] Minghong Fang, Xiaoyu Cao, Jinyuan Jia, and Neil Zhenqiang Gong. Local model poisoning attacks to byzantine-robust federated learning. In *USENIX Security Symposium*, 2020.

[31] Robin C. Geyer, Tassilo Klein, and Moin Nabi. Differentially private federated learning: A client level perspective. In *NeuIPS*, 2017.

[32] Siddharth Gopal. Adaptive sampling for SGD by exploiting side information. In *ICML*, 2016.

[33] Juncheng Gu, Mosharaf Chowdhury, and et al. Tiresias: A GPU cluster manager for distributed deep learning. In *NSDI*, 2019.

[34] Andrew Hard, Kanishka Rao, and et al. Federated learning for mobile keyboard prediction. In *arxiv.org/abs/1811.03604*, 2018.

[35] Florian Hartmann, Sunah Suh, Arkadiusz Komarzewski, Tim D. Smith, and Ilana Segall. Federated learning for ranking browser history suggestions. In *arxiv.org/abs/1911.11807*, 2019.

[36] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[37] Kevin Hsieh, Ganesh Ananthanarayanan, and et al. Focus: Querying large video datasets with low latency and low cost. In *OSDI*, 2018.

[38] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R. Ganger, Phillip B. Gibbons, and Onur Mutlu. Gaia: Geo-distributed machine learning approaching LAN speeds. In *NSDI*, 2017.

[39] Kevin Hsieh, Amar Phanishayee, Onur Mutlu, and Phillip B. Gibbons. The Non-IID data quagmire of decentralized machine learning. In *ICML*, 2020.

[40] Harry Hsu, Hang Qi, and Matthew Brown. Federated visual classification with real-world data distribution. In *ECCV*, 2020.

[41] Zhihao Jia, Oded Padon, and et al. TASO: Optimizing deep learning computation with automatic generation of graph substitutions. In *SOSP*, 2019.

[42] Junchen Jiang, Rajdeep Das, and et al. Via: Improving internet telephony call quality using predictive relay selection. In *SIGCOMM*, 2016.

[43] Junchen Jiang, Yuhao Zhou, Ganesh Ananthanarayanan, Yuanchao Shu, and Andrew A. Chien. Networked cameras are the new big data clusters. In *HotEdgeVideo*, 2019.

[44] Tyler B. Johnson and Carlos Guestrin. Training deep models faster with robust, approximate importance sampling. In *NeurIPS*, 2018.

[45] Peter Kairouz, H. Brendan McMahan, and et al. Advances and open problems in federated learning. In *arxiv.org/abs/1912.04977*, 2019.

[46] Angelos Katharopoulos and Francois Fleuret. Biased importance sampling for deep neural network training. In *arxiv.org/abs/1706.00043*, 2017.

[47] Angelos Katharopoulos and Francois Fleuret. Not all samples are created equal: Deep learning with importance sampling. In *ICML*, 2018.

[48] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. ALBERT: A lite BERT for self-supervised learning of language representations. In *ICLR*, 2020.

[49] Tian Li, Anit Kumar Sahu, Manzil Zaheer, Maziar Sanjabi, Ameet Talwalkar, and Virginia Smith. Federated optimization in heterogeneous networks. In *MLSys*, 2020.

[50] Tian Li, Manzil Zaheer, Ahmad Beirami, and Virginia Smith. Fair resource allocation in federated learning. In *ICLR*, 2020.

[51] Wenqi Li, Fausto Milletari, and Daguang Xu. Privacy-preserving federated brain tumour segmentation. In *Machine Learning in Medical Imaging*, 2019.

[52] Tao Lin, Sebastian U. Stich, Kumar Kshitij Patel, and Martin Jaggi. Don't use large mini-batches, use local SGD. In *ICLR*, 2020.

[53] Jiahuan Luo, Xueyang Wu, Yun Luo, Anbu Huang, Yunfeng Huang, Yang Liu, and Qiang Yang. Real-world image datasets for federated learning. In *arxiv.org/abs/1910.11089*, 2019.

[54] Kshiteej Mahajan, Arjun Balasubramanian, and et al. Themis: Fair and efficient GPU cluster scheduling. In *NSDI*, 2020.

[55] Yishay Mansour, Mehryar Mohri, Jae Ro, and Ananda Suresh. Three approaches for personalization with applications to federated learning. In *arxiv.org/abs/2002.10619*, 2020.

[56] H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *AISTATS*, 2017.

[57] William Mendenhall, Robert J Beaver, and Barbara M Beaver. *Introduction to probability and statistics*. Cengage Learning, 2012.

[58] William Mendenhall, Robert J Beaver, and Barbara M Beaver. *Introduction to probability and statistics*. Cengage Learning, 2012.

[59] Mehryar Mohri, Gary Sivek, and Ananda Theertha Suresh. Agnostic federated learning. In *ICML*, 2019.

[60] Deepak Narayanan, Aaron Harlap, and et al. Pipedream: Generalized pipeline parallelism for dnn training. In *SOSP*, 2019.

[61] Xingchao Peng, Zijun Huang, Yizhe Zhu, and Kate Saenko. Federated adversarial domain adaptation. In *ICLR*, 2020.

[62] Yanghua Peng, Yibo Zhu, and et al. A generic communication scheduler for distributed dnn training acceleration. In *SOSP*, 2019.

[63] Swaroop Ramaswamy, Om Thakkar, Rajiv Mathews, Galen Andrew, H. Brendan McMahan, and Françoise Beaufays. Training production language models without memorizing user data. In *arxiv.org/abs/2009.10031*, 2020.

[64] Sashank Reddi, Zachary Charles, and et al. Adaptive federated optimization. In *arxiv.org/abs/2003.00295*, 2020.

[65] Edo Roth, Daniel Noble, Brett Hemenway Falk, and Andreas Haeberlen. Honeycrisp: Large-scale differentially private aggregation without a trusted core. In *SOSP*, 2019.

[66] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR*, 2018.

[67] Supreeth Shastri, Vinay Banakar, Melissa Wasserman, Arun Kumar, and Vijay Chidambaram. Understanding and benchmarking the impact of GDPR on database systems. In *VLDB*, 2020.

[68] Supreeth Shastri, Melissa Wasserman, and Vijay Chidambaram. The seven sins of personal-data processing systems under GDPR. In *HotCloud*, 2019.

[69] Ananda Theertha Suresh, Felix X. Yu, Sanjiv Kumar, and H. Brendan McMahan. Distributed mean estimation with limited communication. In *ICML*, 2017.

[70] Jianyu Wang and Gauri Joshi. Adaptive communication strategies to achieve the best error-runtime trade-off in local-update SGD. In *MLSys*, 2019.

[71] Kangkang Wang, Rajiv Mathews, Chloe Kiddon, Hubert Eichner, Francoise Beaufays, and Daniel Ramage. Federated evaluation of on-device personalization. In *arxiv.org/abs/1910.10252*, 2019.

[72] Pete Warden. Speech commands: A dataset for limited-vocabulary speech recognition. In *arxiv.org/abs/1804.03209*, 2018.

[73] Mengwei Xu, Jiawei Liu, Yuanqiang Liu, Felix Xiaozhu Lin, Yunxin Liu, and Xuanzhe Liu. A first look at deep learning apps on smartphones. In *WWW*, 2019.

[74] Francis Y. Yan, Hudson Ayers, and et al. Learning in situ: a randomized experiment in video streaming. In *NSDI*, 2020.

[75] Chengxu Yang, Qipeng Wang, and et al. Heterogeneity-aware federated learning. In *arxiv.org/abs/2006.06983*, 2020.

[76] Timothy Yang, Galen Andrew, Hubert Eichner, Haicheng Sun, Wei Li, Nicholas Kong, Daniel Ramage, and Françoise Beaufays. Applied federated learning: Improving Google keyboard query suggestions. In *arxiv.org/abs/1812.02903*, 2018.

[77] Felix X. Yu, Ankit Singh Rawat, Aditya Krishna Menon, and Sanjiv Kumar. Federated learning with only positive labels. In *ICML*, 2020.

[78] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *CVPR*, 2018.

[79] Peilin Zhao and Tong Zhang. Stochastic optimization with importance sampling for regularized loss minimization. In *ICML*, 2015.

A Analysis of Real-world Dataset

Here we provide a detailed description of the datasets used in the paper.

A.1 Dataset of Samples

A summary of statistics for these datasets can be found in Table 2.

Dataset	# of Clients	# of Samples
Google Speech [72]	2,618	105,829
OpenImage [3]	14,477	1,672,231
StackOverflow [9]	342,477	135,818,730
Reddit [8]	1,660,820	351,523,459

Table 2: ***Dataset statistics.***

Google Speech Commands. A speech recognition dataset [72] with over ten thousand clips of one-second-long duration. Each clip contains one of the 35 common words (e.g., digits zero to nine, "Yes", "No", "Up", "Down") spoken by thousands of different people.

OpenImage. OpenImage [3] is a vision dataset collected from Flickr, an image and video hosting service. It contains a total of 16M bounding boxes for 600 object classes (e.g., Microwave oven). We clean up the dataset according to the provided indices of clients. In our evaluation, the size of each image is 96×96.

Reddit and StackOverflow. Reddit [8] (StackOverflow [9]) consists of comments from the Reddit (StackOverflow) website. It has been widely used for language modeling tasks, and we consider each user as a client. In our experiments, we restrict to the 30k most frequently used words, and represent each sentence as a sequence of indices corresponding to these 30k frequently used words. We use Transformers [12] to tokenize these sequences with a block size 64.

In our experiments, we use 90% of the dataset as the training set, and the rest 10% as the held-out testing set.

A.2 Dataset of System Performance and Availability

Heterogeneous system performance. We use the AIBench [1] dataset and MobiPerf [6] dataset, and visualize these numbers in Fig 2. AIBench dataset provides the training time and inference time of different models across a wide range of devices. As specified in real FL deployments [20,34], we focus on the capability of mobile devices that have > 2GB RAM in this paper. To understand the network capacity of these devices, we clean up the MobiPerf dataset, and analyze the available bandwidth when they are connected with WiFi, which is preferred for FL as well [45]. In our evaluations, we configure the system performance of each client via sampling by the distribution of these profiles.

Availability of clients. We use a large-scale real-world user behavior dataset [75]. It comes from a popular input method app (IMA) that can be downloaded from Google Play, and covers 136k users and spans one week from January 31st to February 6th in 2020. This dataset includes 180 million trace items (e.g., battery charge or screen lock) and we consider user devices that are in charging to be available, as specified in real FL deployments [20].

A.3 Simulation Setup

We employ the traditional deployment of parameter server and workers, which is the mainstream simulation design used in Google [11] and existing FL researches [28, 49, 50]. The parameter server acts as the central coordinator in FL, and the worker works as the client. Communications between coordinators and clients will travel through the network, though the communication latency is simulated by the profile of available bandwidth. There exists a global virtual clock on the parameter server to mimic different behaviors in FL (e.g., some clients become offline).

The statistical performance (i.e., round-to-accuracy) performs in the same way of that in traditional ML, while the duration of each round is simulated with our real-world profiles. After receiving the global model by communicating with the coordinator (i.e., parameter server), the worker will first dump this model to the storage. For the simulation run of each client, the worker will load the latest model from the storage, and perform training on the dataset of that client. Note that the worker acts as an individual client at a time, and the computation time is also measured in terms of her profile. When the training of each individual client completes, the worker will push updates to the coordinator, including the updated model and virtual clock time spent on this training round. After receiving these updates, the coordinator will pad this clock time to the global clock. Dynamics of system performance are simulated by specifying different computation/communication latency in profile.

B Oort APIs

```
class Oort:
  # the APIs for Federated Training Selector
  def create_training_selector(self, config)
  def update_client_util(self, client_id, utility)
  def select_participants(self, num_of_participants)
  # the APIs for Federated Testing Selector
  def create_testing_selector(self, config)
  def update_client_info(self, client_id, client_info)
  def select_representative(self, num_of_samples,
     testing_config)
  def select_by_category(self, request_list,
     testing_config)
  def select_by_deviation(self, dev_target,
     capacity_range, total_num_clients)
```

Figure 18: ***Oort APIs for federated training and testing.***

Oort APIs. We have implemented Oort as a Python library. Oort provides simple APIs to abstract away the problem of participant selection, and developers can import Oort in their application. Figure 18 summarizes APIs used in training and testing selector. And we explain these APIs with usage exam-

ples of both training selector (Figure 6) and testing selector (Figure 19).

```
import Oort

def federated_model_testing():
  selector = Oort.create_testing_selector()
  # [Clairvoyant] Update client data and sys info.
  for clientId in available_clients:
    selector.update_client_info(
                clientId, client_info[clientId])

  ''' [Clairvoyant Query] Give me 10k representative
      samples given data transfer size and budget'''
  participants = selector.select_representative(
                  num_of_samples=10000,
                  testing_config={
                  'data_transfer_size':size,
                  'budget':budget})

  ''' [Clairvoyant Query] Give me 5k samples of
     category A, 5k samples of category B'''
  participants = selector.select_by_category(
                  request_list={'A': 5000, 'B': 5000},
                  testing_config={
                  'data_transfer_size':size,
                  'budget':budget})

  ''' [Non-Clairvoyant Query] Give me a subset w/
      less than 10 deviation from the global'''
  participants = selector.select_by_deviation(
                  dev_tolerance=10, capacity_range=500
                  total_num_clients=1000000)
  ... # Activate execution and return accuracy
```

Figure 19: ***Code snippets of FL testing support in Oort.***

Usage Examples of Testing Selector. Figure 19 shows example code snippets for both scenarios. With the individual data characteristics, the developer can use the clairvoyant selector by dumping the client information to Oort. Note that we use stratified sampling [57] to serve queries like "*50k representative samples*", which will determine the number of samples in each category, thus paraphrasing such queries in a form of *"[p_i, p_j] samples of class [i, j]"*. For the non-clairvoyant scenario, Oort receives the developer-specified deviation tolerance and the range of number of samples that a client can hold.

C Proving Benefits of Statistical Utility

We follow the proof of importance sampling to show the advantage of our statistical utility in theory. The convergence speed of Stochastic Gradient Descent (SGD) can be defined as the reduction R of the divergence of model weight $\mathbf{w}$ from its optimal $\mathbf{w}^*$ in two consecutive round t and $t+1$ [47, 79] :

$$R = \mathbb{E}\big[\ \| \mathbf{w_t} - \mathbf{w}^* \|_2^2 - \| \mathbf{w_{t+1}} - \mathbf{w}^* \|_2^2\ \big] \tag{2}$$

How does oracle sampling help in theory? If the learning rate of SGD is η and we use loss function L to measure the training loss between input features x and the label y, then $\mathbf{w_{t+1}} = \mathbf{w_t} - \eta \nabla L(\mathbf{w_t}(x_i), y_i)$. We now set the gradient $G_t = \nabla L(\mathbf{w_t}(x_i), y_i)$ for brevity, then from Eq. (2):

$$\begin{aligned}
R &= -\mathbb{E}\big[(\mathbf{w_{t+1}} - \mathbf{w}^*)^T(\mathbf{w_{t+1}} - \mathbf{w}^*) - (\mathbf{w_t} - \mathbf{w}^*)^T(\mathbf{w_t} - \mathbf{w}^*)\big] \\
&= -\mathbb{E}\big[\mathbf{w_{t+1}^T w_{t+1}} - 2\mathbf{w_{t+1}}\mathbf{w}^* - \mathbf{w_t^T w_t} + 2\mathbf{w_t}\mathbf{w}^*\big] \\
&= -\mathbb{E}\big[(\mathbf{w_t} - \eta\mathbf{G_t})^T(\mathbf{w_t} - \eta\mathbf{G_t}) + 2\eta G_t^T \mathbf{w}^* - \mathbf{w_t}^T\mathbf{w_t}\big] \\
&= -\mathbb{E}\big[-2\eta(\mathbf{w_t} - \mathbf{w}^*)G_t + \eta^2 G_t^T G_t\big] \\
&= 2\eta(\mathbf{w_t} - \mathbf{w}^*)\mathbb{E}[G_t] - \eta^2\mathbb{E}[G_t]^T\mathbb{E}[G_t] - \eta^2 Tr(\mathbb{V}[G_t])
\end{aligned} \tag{3}$$

It has been proved that optimizing the first two terms of Eq. (3) is intractable due to their joint dependency on $\mathbb{E}[G_t]$, however, one can gain a speedup over random sampling by intelligently sampling important data bins to minimize $Tr(\mathbb{V}[G_t])$ (i.e., reducing the variance of gradients while respecting the same expectation $\mathbb{E}[G_t]$) [44, 47]. Here, the oracle is to pick bin B_i with a probability proportional to its importance $|B_i|\sqrt{\frac{1}{|B_i|}\sum_{k\in B_i} \| G(k) \|^2}$, where $\| G(k) \|$ is the L2-norm of the unique sample k's gradient $G(k)$ in bin B_i (Please refer to [32] for detailed proof).

How does loss-based approximation help? We have shown the advantage of importance sampling by sampling the larger gradient norm data, and next we present a theoretical proof that motivates our loss-based utility design.

Corollary 0.1. *(Theorem 2 in [46]). Let $\| G(k) \|$ denote the gradient norm of any sample k, $M = \max \| G(k) \|$. There exists $K > 0$ and $C < M$ such that $\frac{1}{K}L(\mathbf{w}(x_k), y_k) + C \geq \| G(k) \|$.*

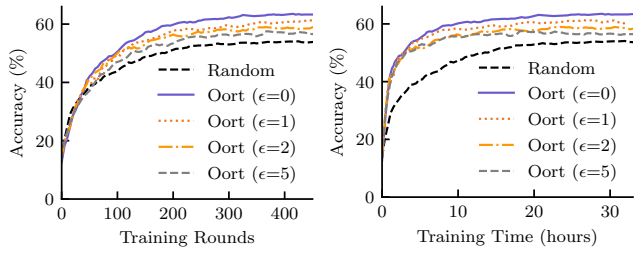
This corollary implies that a bigger loss leads to a large upper bound of the gradient norm. To sample data with a larger gradient norm, we prefer to pick the one with bigger loss. Moreover, it has been empirically shown that sampling high loss samples exhibits similar variance reducing properties to sampling according to the gradient norm, resulting in better convergence speed compared to naive random sampling [47].

By taking account of the oracle and the effectiveness of loss-based approximation, we propose our loss-based statistical utility design, whereby we achieve the close-to-optimal statistical performance (§7.2.2).

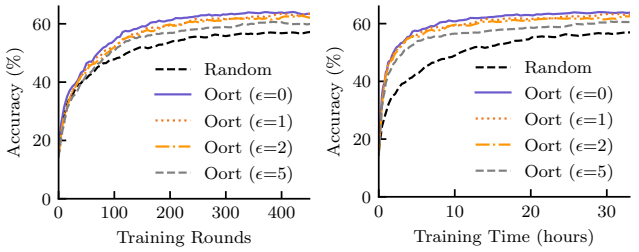
D Privacy Concern in Collecting Feedbacks

Depending on different requirements on privacy, we elaborate on how Oort respects privacy while outperforming existing mechanisms (§4.2).

Compute aggregated training loss on clients locally. Our statistical utility of a client relies on the aggregated training loss of all samples on that client. The training loss of each sample measures the prediction uncertainties of model on every possible output (e.g., category), and even the one with a



(a) Round to accuracy (MobileNet). (b) Time to accuracy (MobileNet).



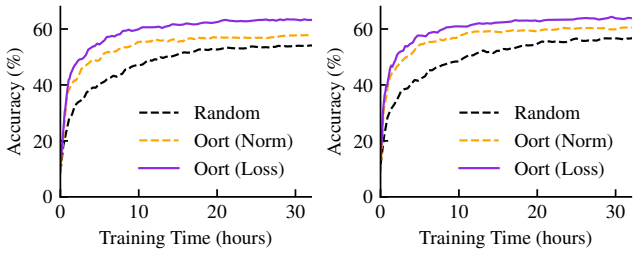
(c) Round to accuracy (ShuffleNet). (d) Time to accuracy (ShuffleNet).

Figure 20: ***Oort improves performance even under noise.***

correct prediction can generate non-zero training loss [27]. So it does not reveal raw data inputs. Moreover, it does not leak the categorical distribution either, since samples of the same category can own different training losses due to their heterogeneous input features. Note that even when we consider homogeneous input features, a high total loss can be from several high loss samples, or many moderate loss samples.

Add noise to hide the real aggregated loss. Even when clients have very stringent privacy concerns on their aggregated loss, clients can add noise to the exact value. Similar to the popular differentially private FL [31], clients can disturb their real aggregated loss by adding Gaussian noise (i.e., noise from the Gaussian distribution). In fact, Oort can tolerate this noisy statistical utility well, owing to its probabilistic selection from the pool of high-utility clients, wherein teasing apart the top-k% utility clients from the rest is the key.

We first prove that Oort is still very likely to select high-utility clients even in the presence of noise. To pick K participants out of N all feasible clients, there are totally $\binom{N}{K}$ possible combinations. We denote these combinations as X_i and sort them $X_1 \leq \ldots \leq X_n$ by the ascending order of total utility. Adding noise to each client ends up with an accumulated noise on X_i. Thereafter, X_i turns to random variables $\mathbf{X}_i$ that follow the distribution of accumulated noise. Specifically, distribution of $\mathbf{X}_i$ is equivalent to shifting the distribution of noise horizontally by a constant X_i. Given that noise added to X_i follows the same distribution, (i) $\mathbf{X}_i$ experiences the same standard deviation for every i; (ii) the expectation of $\mathbf{X}_i$ is the sum of X_i and the expectation of noise. Note that adding a constant (i.e., the expectation of noise here) to the inequality does not change its properties, so we still have $\mathbb{E}[\mathbf{X}_1] \leq \ldots \leq \mathbb{E}[\mathbf{X}_n]$. As such, we are more likely to select



(a) MobileNet. (b) ShuffleNet.

Figure 21: ***Oort outperforms with different utility definitions.***

high-utility clients (i.e., combination $\mathbf{X}_i$ with higher $\mathbb{E}[\mathbf{X}_i]$) in sampling when picking i with the highest value of X_i.

We next show the superior empirical performance of Oort over its counterparts under noise. In this experiment, we add noise from the Gaussian distribution $Gaussian(0, \sigma^2)$, and investigate Oort's performance with different σ. Similar to differential FL [31], we define $\sigma = \varepsilon \times Mean(real\_value)$, where $Mean(real\_value)$ is the average of real value without noise. Note that we take this $real\_value$ as reference for the ease of presentations, and developers can refer to other values. As such, a large ε implies larger variance in noise, thus providing better privacy by disturbing the real value significantly. We report the statistical efficiency after adding noise to the statistical utility (Fig 20(a) and Fig 20(c)), as well as the time-to-accuracy performance (Fig 20(b) and Fig 20(d)). We observe that Oort still improves performance across different amount of noise, and is robust even when the noise is large (e.g., $\varepsilon = 5$ is often considered to be very large noise [13]).

Rely on gradient norm of batches. For case where clients are even reluctant to report the noisy loss, we introduce an alternative statistical utility to drive our exploration-exploitation based client selection. Our intuition is to use the gradient norm of batches to approximate the gradient norm of individual samples $\nabla f(k)$ in the oracle importance $|B_i|\sqrt{\frac{1}{|B_i|}\sum_{k\in B_i} \| \nabla f(k) \|^2}$. In mini-batch SGD, we have

$$\mathbf{w}_{t+1} = \mathbf{w}_t - learning\_rate \times \frac{1}{batch\_size} \times \sum_{k\in batch} \nabla f(k)$$

where w_t is the model weights at time t. Now, we can use the gradient norm of batches (i.e., $\| \mathbf{w}_{t+1} - \mathbf{w}_t \|^2$) to approximate $\| \nabla f(k) \|^2$, and they become equivalent when the batch size is 1. Note that today's FL is already collecting the model updates (i.e., $\mathbf{w}_{t+1} - \mathbf{w}_t$), so we are not introducing additional information. As such, we consider the client with larger accumulated gradient norm of batches to be more important.

We report the empirical performance of this approximation and the loss-based statistical utility using YoGi. As shown in Fig 21, Oort achieves superior performance over the random selection, and the loss-based utility is better than its counterparts. This is because the approximation accuracy with the norm of batches decreases when using mini-batch SGD,

Strategy	Average (%)	Worst 5% (%)	Best 5% (%)	Var. of Accuracy
Prox	67.0	30.0	88.5	170.7
Oort + Prox	72.1	37.6	93.5	153.1
YoGi	63.6	27.8	85.7	173.0
Oort + YoGi	74.8	42.9	92.9	134.1
Centralized	79.5	49.8	95.4	114.6

Table 3: ***Oort achieves a more fair accuracy distribution across clients (e.g., smaller variance of model accuracy across clients).***

Strategy	TTA (h)	Final Accuracy (%)	Var. (Rounds)
Random	36.3	57.3	0.39
f=0	5.8	64.2	6.52
f=0.25	6.1	62.4	5.1
f=0.5	13.1	59.7	2.03
f=0.75	25.4	58.6	0.65
f=1	30.1	57.2	0.31

Table 4: ***Oort can enforce diverse fairness criteria, where Random reports the performance of random participant selection and the rest reports Oort's performance given different fairness knobs (f). The variance of rounds reports how fairness is enforced in terms of the number of participating rounds across clients, so a smaller variance implies better fairness. We measure the time to accuracy (TTA) using 57.2% as the target accuracy.***

whereas mini-batch SGD is more popular than the single-sample batch in ML.

E Fairness of Participant Selection

Oort improves the fairness of model accuracy. While the FL developer may have different fairness criteria, achieving a more fair accuracy distribution across interested clients for that given model is mostly the paramount [45]. We follow the popular metrics to measure the fairness of model accuracy for different strategies [45, 50]. As shown in Table 3, where we test the ShuffleNet model accuracy across all clients after training completes, compared with the random participant selection, Oort improves the fairness of model accuracy by boosting the model accuracy for both the worst 5% clients and the best 5% clients, while achieving a smaller variance of model accuracy across all clients.

Oort can enforce developer-preferred fairness. Moreover, we show that Oort can flexibly respect other developer preferences on diverse fairness. In this experiment, we expect all clients should have participated training with the same number of rounds (Table 4), implying a fair resource usage [45]. We train ShuffleNet model on OpenImage dataset with YoGi. To this end, we sweep different knobs f to accommodate the developer demands for the time-to-accuracy efficiency and fairness. Namely, we replace the current utility definition of client i with $(1-f)\times Util(i)+f\times fairness(i)$, where $fairness(i)=max\_resource\_usage-resource\_usage(i)$. Understandably, time-to-accuracy efficiency will significantly decrease as $f\to 1$, since we gradually end up with round-robin participant selection, which totally ignores the data utility of clients. Note that Oort still achieves better time-to-accuracy even when $f\to 1$ as it prioritizes high system utility clients in such a round-robin scenario. Moreover, we note that Oort can enforce these fairness criteria while improving efficiency.

F Determining Size of Participants

We next introduce Lemma 1, which captures how the empirical value of $\bar{X}$ (i.e., average number of samples of participants for category X) deviates from the expectation $E[\bar{X}]$ (i.e., average number of samples of all clients) as the size of participants n varies.

Lemma 1. *For a given tolerance on deviation ε and confidence interval δ for category X, the number of participants n we need to achieve $Pr[|\bar{X}-E[\bar{X}]|<\varepsilon]>\delta$ requires:*

$$n \geq (N+1)\times \frac{1}{1-\frac{2N}{\log(1-\delta)}\times\left(\frac{\varepsilon}{\max\{X\}-\min\{X\}}\right)^2} \quad (4)$$

where N is the total number of feasible clients, and $\max\{X\}$ and $\min\{X\}$ denote the global maximum and minimum possible number of samples that all clients can hold, respectively.

Lemma 1 is a corollary of Hoeffding-Serfling Bound [17], and we omit the detailed proof for brevity. Intuitively, when we have an extremely stringent requirement (i.e., $\varepsilon\to 0$), we have to include more participants (i.e., $n\to N$). When more information of the client data characteristics is available, one can refine this range better. For example, the bound of Eq. (4) can be improved with Chernoff's inequality [17] when the distribution of sample quantities is provided.

Similarly, the multi-category scenario proves to be an instance of multi-variate Hoeffding Bound. Given the developer-specific requirement on each category, the developer may want to figure out how many participants needed to satisfy all these requirements simultaneously (e.g., $Pr[|\bar{X}-E[\bar{X}]|<\varepsilon_x\wedge|\bar{Y}-E[\bar{Y}]|<\varepsilon_y]>\delta$). More discussions are out of the scope of this paper, but readers can refer to [18] for detailed discussions and a complete solution.

G MILP Formula for Testing Selector

For each category $i(\in I)$ of interest, the developer has preference p_i (preference constraint), and an upper limit B (referred to as budget) on how many participants she can have [16].[9] Each participant $n(\in N)$ can contribute n_i samples out of her capacity $c_n{}^i$ (capacity constraint). Given her compute speed s_n, the available bandwidth b_n and the size of data transfers d_n, we aim to minimize the duration of model testing:

[9] $B\to$ inf represents a hypothetical case where FL can run on all clients.

$$\min \left\{ \max_{n \in N} \left(\frac{\sum_{i \in I} n_i}{s_n} + \frac{d_n}{b_n} \right) \right\} \quad \triangleright \text{ Minimize duration}$$

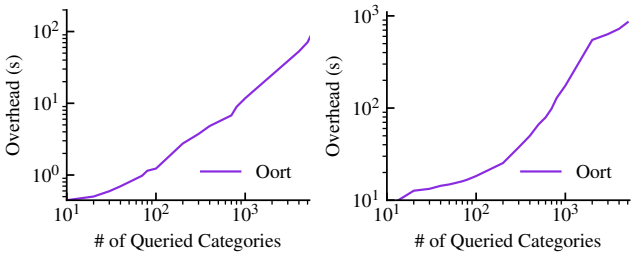
$$\text{s.t.} \quad \forall i \in I, \sum_{n \in N} n_i = p_i \quad \triangleright \text{ Preference Constraint}$$

$$\forall i \in I, \forall n \in N, n_i \leq c_n^i \quad \triangleright \text{ Capacity Constraint}$$

$$\forall i \in I, \sum_{n \in N} \mathbb{1}(n_i > 0) \leq B \quad \triangleright \text{ Budget Constraint}$$

The max-min formulation stems from the fact that testing completes after aggregating results from the last participant. To enforce fairness (i.e., each client group should contribute a certain number of samples) while optimizing the duration of model testing in our clairvoyant selector, Oort can still apply our greedy heuristic in §5.2 to select participants by treating each given quota of samples (e.g., the product of the total number of samples and fair share of each group) for each group as an individual query.

H Model Testing on Large-Scale Datasets



(a) StackOverflow (0.3M clients). (b) Reddit (1.6M clients).

Figure 22: ***Oort scales to millions of clients, while MILP did not complete any query.***

Using the same setting used on OpenImage dataset (§7.3.2), we investigate Oort's performance on the large-scale StackOverflow and Reddit dataset with millions of clients, where we take 1% of the global data as the requirement, and sweep the number of interested categories from 1 to 5k. Figure 22 shows even though we gradually magnify the search space of participant selection by introducing more categories, Oort can serve our requirement in a few minutes at the scale of millions of clients, while MILP fails to generate the solution decision for any query.