Pisces: Efficient Federated Learning via Guided Asynchronous Training

Zhifeng Jiang
HKUST
zjiangaj@cse.ust.hk

Wei Wang
HKUST
weiwa@cse.ust.hk

Baochun Li
University of Toronto
bli@ece.toronto.edu

Bo Li
HKUST
bli@cse.ust.hk

ABSTRACT

Federated learning (FL) is typically performed in a synchronous parallel manner, and the involvement of a slow client delays the training progress. Current FL systems employ a participant selection strategy to select fast clients with quality data in each iteration. However, this is not always possible in practice, and the selection strategy has to navigate a knotty tradeoff between the speed and the data quality.

This paper makes a case for *asynchronous FL* by presenting Pisces, a new FL system with intelligent participant selection and model aggregation for accelerated training despite slow clients. To avoid incurring excessive resource cost and stale training computation, Pisces uses a novel scoring mechanism to identify suitable clients to participate in each training iteration. It also adapts the aggregation pace dynamically to bound the progress gap between the participating clients and the server, with a provable convergence guarantee in a smooth non-convex setting. We have implemented Pisces in an open-source FL platform, Plato, and evaluated its performance in large-scale experiments with popular vision and language models. Pisces outperforms the state-of-the-art synchronous and asynchronous alternatives, reducing the time-to-accuracy by up to 2.0× and 1.9×, respectively.

CCS CONCEPTS

• **Computing methodologies** → **Supervised learning**; • **Security and privacy** → *Database and storage security*.

KEYWORDS

Federated Learning, Asynchronous Training, Efficiency

SoCC '22, November 7–11, 2022, San Francisco, CA, USA

ACM ISBN 978-1-4503-9414-7/22/11.
https://doi.org/10.1145/3542929.3563463

ACM Reference Format:
Zhifeng Jiang, Wei Wang, Baochun Li, and Bo Li. 2022. Pisces: Efficient Federated Learning via Guided Asynchronous Training. In *SoCC '22: ACM Symposium on Cloud Computing (SoCC '22), November 7–11, 2022, San Francisco, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3542929.3563463

1 INTRODUCTION

Federated learning [48] (FL) enables multiple clients to collaboratively and privately train a shared model, under the orchestration of a central server. At its core, FL lets clients keep their private data onsite, while only transferring (protected) local updates containing minimum information, such as the gradients or the model weights [33], to the server. By evading the privacy risks of centralized learning, FL has gained increasing popularity in a multitude of applications, such as mobile services [9, 17, 18, 54, 55, 70], financial business [46, 65], and medical care [44, 52].

Current FL systems orchestrate the training process following a *synchronous parallel scheme*, where the server waits for all participating clients to finish local training and then uses the aggregated updates to refine the global model [6, 48]. While synchronous FL is easy to implement, the *time-to-accuracy*, measured by the wall clock time needed to train a model to the target accuracy, can be substantially delayed in the presence of *stragglers* – those whose updates arrive much later than others. In a typical FL setting, clients have a wide range of computing capabilities and their training speed may differ by orders of magnitude [38, 67, 69], leading to a salient straggler problem (§2.1).

A common straggler mitigation approach in FL is to identify slow clients and exclude them from participating into the training. Existing works propose various metrics to quantify the computing capabilities of clients, based on which they reduce the involvement of slow clients [8, 38, 51]. Yet, participant selection may not always work well. Consider a pathological case where the client's speed and data quality are *inversely correlated* [24, 38]. In this case, prioritizing fast clients inevitably excludes those slower, yet informative clients possessing quality data. As the time-to-accuracy depends on both the speed and data quality of participants, reconciling the two objectives often requires navigating an *unpleasant, knotty tradeoff*. We show in §2.2 that even the state-of-the-art

participant selection strategy, namely Oort [38], can make an inefficient decision that underperforms the naive random selection strategy by 2.7×.

In this paper, we turn to a more effective solution that fundamentally mitigates the straggler problem by switching to an *asynchronous parallel scheme*. In asynchronous FL, the server can both (1) select a subset of idle clients to run and (2) aggregate the received local updates at any time, regardless of the training progress of other clients. Without a synchronization barrier, there is no need to wait for straggling clients, thus relieving the potential tension between the client speed and the data quality. While the intuition is simple, a few challenges remain open to solve.

First, asynchronous FL results in more frequent client participations, many of which are not very helpful, leading to wasted computations and low *resource efficiency*. Existing approaches commonly involve all clients and keep them running throughout the training process [10, 57, 68]. Yet, our experiments show that compared with selecting only a small portion (e.g., 25%) of clients, keeping everyone busy yields marginal performance gains (e.g., ≤1.5×) yet substantial resource overhead (e.g., 3.6-6.7×) (§4.1). While this points to *controlled concurrency*, i.e., limiting the maximum amount of clients allowed to run concurrently [50], it remains unclear how to fully utilize each quota for running clients to maximize resource efficiency in asynchronous FL.

Second, eliminating the synchronization barrier results in *stale computation*. When a client reports its local update, the global model maintained on the server may have gone far ahead of the local version based on which the update is computed. The *staleness* of the computation, measured by the progress gap between the server and the client, harms the quality of update: both theoretical analysis [50] and empirical results [68] show that incorporating stale local updates can slow the training convergence. It is hence desirable to bound the staleness at a low level. Existing approaches, notably buffered aggregation [50], though effective in staleness control, provide no guaranteed staleness bound and rely on manual tuning to adapt to different system settings.

Tackling the above challenges, we present Pisces[1], an end-to-end asynchronous parallel scheme for efficient FL training (§3). To make a good use of each quota under controlled concurrency, Pisces conducts *guided participant selection*. In a nutshell, Pisces prioritizes clients with high data quality, which is measured by the clients' training loss based on the approximation of importance sampling [29, 34]. Given that the training loss may be misleading in case of corrupted data or malicious clients, Pisces clusters the loss values of clients, based on which it identifies outliers and excludes them from being selected. Moreover, Pisces predicts the staleness of clients to avoid inducing stale computation. Our design eliminates the unpleasant tradeoff between the client's speed and the data quality, achieving high resource efficiency under the asynchronous settings (§4).

To limit the impacts of stale computation that is already induced, Pisces further adopts an *adaptive aggregation pace control* with a novel online algorithm. By dynamically adjusting the aggregation interval to match the speed of running clients, Pisces balances the aggregation load over time for being both steady in convergence and scalable to a large client base. The algorithm automatically adapts to different distributions of clients' speed without manual tuning (§5). It can also bound the clients' staleness under any target value, with a provable convergence guarantee in a smooth non-convex setting (§6.1).

We have implemented Pisces atop Plato [64], an open-source FL platform (§7), under practical settings of system and data heterogeneity across 100-400 clients (§8). Extensive experiments over image classification and language model applications show that, compared to the state-of-the-art synchronous and asynchronous FL designs, namely Oort [38] and FedBuff [50], Pisces accelerates the time-to-accuracy by up to 2.0× and 1.9×, respectively. Pisces is also shown to be insensitive to the choice of hyperparameters.

In summary, we make the following contributions:

(1) We highlight the knotty tradeoff between clients' speed and data quality faced by synchronous FL.
(2) We propose new algorithms to automate participant selection and aggregation pace control in asynchronous FL for improved resource efficiency and reduced stale computation.
(3) We implement and evaluate Pisces through large-scale deployment to show its performance advantages over the state-of-the-art solutions under practical settings.
(4) We open-source system implementation for Pisces [1] and hereby invite the community to contribute more to FL research.

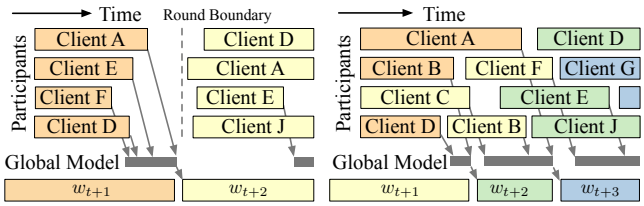
2 BACKGROUND AND MOTIVATION

In this section, we briefly introduce synchronous FL and its inefficiency in the presence of stragglers (§2.1). We then discuss the limitations of current participant selection strategies when navigating the tradeoff of clients' speeds and data quality in synchronous FL, thus motivating the need for relaxing the synchronization barrier (§2.2).

2.1 Synchronous Federated Learning

Federated learning (FL) has become an emerging approach for building a model from decentralized data. To orchestrate

[1]Symbolized by two fish swimming, Pisces is a constellation of the zodiac. We use it as a metaphor for the two design knobs that we optimize in asynchronous FL.



(a) Synchronous **(b) Asynchronous**

Figure 1: Sychronous and asynchronous FL [24].

the training process, current FL systems employ the parameter server architecture [42, 59] with a *synchronous parallel* design [6, 28, 48], where a server refines the global model on a round basis, as illustrated in Figure 1a. In each round, the server randomly selects a subset of online clients to participate and sends the current global model to them. These clients then perform a certain number of training steps using their local datasets. Finally, the server waits for all participants to report their local updates and uses the aggregated updates to refine the global model before advancing to the next round.

Performance bottlenecks. The performance of synchronous FL can be significantly harmed by *straggling clients* that process much slower than non-stragglers. This problem becomes particularly salient in cross-device scenarios, where the computing power and data amount vary across clients by orders of magnitude [7, 38, 67, 69]. In our testbed experiments (detailed in Section 8.1), the server running FedAvg [48] algorithm remains idle for 33.2-57.2% of the training time waiting for the slowest clients to report updates.

To mitigate stragglers, simple solutions include periodic aggregation or over-selection [6]. The former imposes a deadline for participants to report updates and ignores late submissions; the latter selects more participants than needed but only waits for the reports from a specific number of early arrivals. Both solutions waste the computing efforts of slow clients, leading to suboptimal resource efficiency. Developers then seek to improve over random selection by identifying and excluding stragglers in the first place.

2.2 Participant Selection

By prioritizing fast clients, the average round latency will be shortened compared to random selection. However, this is not sufficient to achieve a shorter *time-to-accuracy*, which is the product of average round latency and the number of rounds taken to reach the target accuracy. To avoid inflating the number of rounds when handling stragglers, participant selection should also account for the clients' data quality. As data quality may not be positively correlated with speeds, a good strategy must strike a balance in-between.

Prior arts. To navigate the tradeoff between the two factors, extensive research efforts such as FedCS [51], TiFL [8], Oort [38] and AutoFL [35] have been made in the literature, among which Oort is the state-of-the-art for its fine-grained navigation and training-free nature. To guide participant selection, Oort ranks each client i with a utility score U_i^{Oort} that jointly considers the client's speed and data quality:

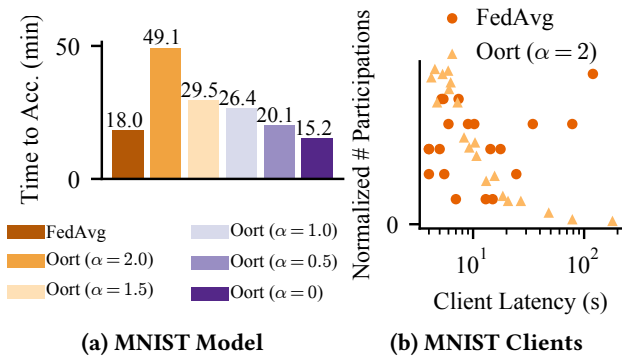
$$U_i^{Oort} = \underbrace{|B_i|\sqrt{\frac{1}{|B_i|}\sum_{k\in B_i} Loss(k)^2}}_{Data\ quality} \times \underbrace{(\frac{T}{t_i})^{\mathbb{1}(T<t_i)\times\alpha}}_{System\ speed}. \quad (1)$$

In a nutshell, the first component is the aggregate training loss that reflects the volume and distribution of the client's dataset B_i. The second component compares the client's completion time t_i with the developer-specified duration T and penalizes any delay (which makes the indicator $\mathbb{1}(T < t_i)$ outputs 1) with an exponent $\alpha > 0$. Oort prioritizes the use of clients with high utility scores.

Inefficiency. We briefly explain the *strict penalty effect* that Equation (1) imposes on slow clients. Following the evaluation setting in Oort, assume $\alpha = 2$. The quantified data quality of a straggler i will then be divided by a factor proportional to the square of its latency t_i. Given that in Oort a client is selected with probability in proportion to its utility score, such a penalty implies that straggling clients have much less chance of being selected in the training than non-stragglers.

However, imposing such a heavy penalty is not always desirable. Consider a pathological case where the clients' speeds and data quality are inversely correlated, i.e., faster clients are coupled with fewer data of poorer quality. Note that this is not uncommon in practice [24, 38]; for example, it usually takes a client with a larger dataset a longer time to finish training. In this case, strictly penalizing slow clients can lead to using an insufficient amount or quality of data, which can impair the time-to-accuracy compared to no optimization. To illustrate this problem, we compare Oort over FedAvg [48] (that uses random selection) in a small-scale training task where 5 out of 20 clients are selected at each round to train over the MNIST dataset. In this emulation experiment (detailed settings in Section §8.1), the completion time of clients are configured following the Zipf distribution ($a = 1.2$) [26, 41, 63] so that the majority are fast while the rest are extremely slow. Accordingly, fast clients are associated with fewer data samples of more unbalanced label distribution and vice versa, leveraging latent Dirichlet allocation (LDA) [3, 5, 23, 56].

Figure 2a depicts the time taken to reach 95% accuracy. Oort with straggler penalty factor $\alpha = 2.0$ suffers 2.7× slowdown than FedAvg. According to Figure 2b, Oort's poor



(a) MNIST Model (b) MNIST Clients

Figure 2: Slow clients are overlooked by Oort [38] when clients' speed and data quality are at odds.

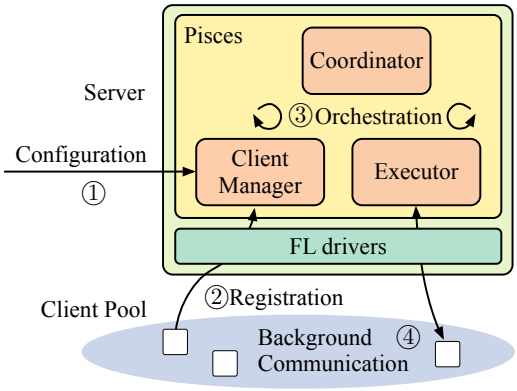


Figure 3: Pisces architecture.

performance results from its bias toward fast clients. Under strict penalties, stragglers in Oort get way less attention than others, despite having rich data of good quality. Instead, FedAvg evenly picks each client, which involves stragglers for enough times and thus leads to faster convergence.

Generality. To confirm that the strict penalty effect generally exists, we evaluate Oort across different small α's: 2.0, 1.5, 1.0, 0.5, and 0. As Figure 2a shows, while using a more gentle straggler penalty factor (i.e., smaller α) does yield a shorter time-to-accuracy, the use of nonzero factors is still harmful. For Oort to improve over FedAvg, it should ignore the speed disparity and purely focus on prioritizing clients with high data quality (i.e., $\alpha = 0$). This strategy, however, deviates from Oort's original design and mandates manual tuning with prior knowledge. This limitation also generally applies to other optimization alternatives due to the tricky tradeoff between clients' speeds and data quality.

In short, participant selection does not completely address the performance bottlenecks in synchronous FL. The limited tolerance for stragglers is responsible for such inefficiency.

3 PISCES OVERVIEW

To sidestep the above tradeoff faced by synchronous FL, we design Pisces, an asynchronous FL framework that improves training efficiency with novel algorithmic principles. We first give an overview of Pisces, including its design knobs and architecture.

Design knobs. The advantages of switching to an asynchronous design are two-fold. *First*, it can inform an available client to train whenever it is idle, as illustrated in Figure 1b. As such, fast clients do not have to wait for stragglers to finish their tasks. *Second*, the server can conduct model aggregation as soon as a local update becomes available. Thus, the pace at which the global model evolves can be lifted by fast clients, instead of being constrained by stragglers.

Given the above degrees of freedom, Pisces needs to address the following practical issues in order to unleash the performance potential in asynchronous training.

(1) How many clients are selected to run concurrently? If there is an available quota for running clients at some point, how to decide whether it is time to launch local training, and at which client (§4)?
(2) When receiving local updates from clients, how to decide whether to aggregate available updates right away or wait for more updates to arrive (§5)?

Architecture. Pisces is a practical and efficient FL framework with new implementations on both the server and clients. Figure 3 depicts how Pisces fits in the existing FL workflow. ① *Configuration*: given a training plan, the client manager configures itself and waits for clients to arrive. ② *Registration*: upon a client's arrival, the client manager registers its meta-information (e.g., dataset size) and starts to monitor its runtime metrics (e.g., response latency). ③ *Orchestration*: during training, the coordinator iterates over a control loop that interacts with both the client manager and the executor. ④ *Background Communication*: in the background, the executor handles the server-client communication, maintains a buffer that stores non-aggregated local updates, and validates the model with hold-out datasets.

We further zoom in on the coordinator's control loop, as shown in Algorithm 1. At each iteration, the coordinator first asks the client manager whether to perform model aggregation (Line 3). If necessary, it delegates the task to the executor for completion. The coordinator then consults the client manager on whether any idle client needs to be selected (Line 7). The client manager will either reply no, or yes with a plan to instruct the executor on whom to select. On meeting a certain termination condition, e.g., reaching a target accuracy, the control loop will stop and output a final trained model (Line 5).

```
Function AsynchronousTraining()
    /* Repeat every time window. */
    while True do
        /* Perform model aggregation if necessary. */
        if ManagerToAggregate() then
            ExecutorAggregate()
        /* Exit the loop if applicable. */
        if ExecutorToTerminate() then
            break
        /* Select participants if necessary. */
        if ManagerToSelect() then
            clients = ManagerSelectClients()
            ExecutorStartTraining(clients)
```

Algorithm 1: Coordinator's control loop in Pisces.

In the subsequent sections, we delve into Pisces's algorithmic principles on how to turn the above design knobs for efficient asynchronous training.

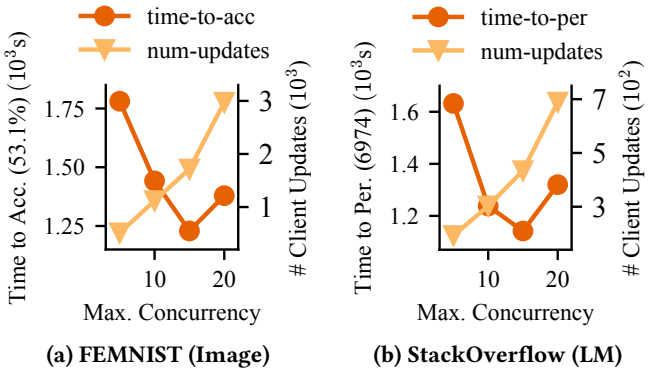
4 PARTICIPANT SELECTION

In this section, we consolidate the need for controlling the maximum number of clients allowed to run concurrently (i.e., concurrency) (§4.1), and then introduce how Pisces selects clients for fully utilizing each quota for running clients (§4.2).

4.1 Controlled Concurrency

Idle clients can be invoked at any time in asynchronous FL. Many existing works continuously invoke all clients for maximizing the speedup [10, 57, 68]. Having all clients training concurrently, however, can *saturate* the server's memory capacity and network bandwidth. Even with abundant resources, it remains a question whether excessive resource usage can translate into a *proportional* runtime reduction. There exists a work that evaluates the runtime performance of FL training where the maximum allowed concurrency is controlled below a certain portion of the population [50]. However, it neither characterizes the relationship between the runtime gain and resource cost in asynchronous FL, nor does it suggest how to fully utilize each quota for running clients given a particular concurrency limit.

Resource cost v.s. runtime gain. To empirically examine the relationship between resource cost and runtime gain in asynchronous FL, we evaluate FedBuff, the state-of-the-art asynchronous FL approach deployed in Facebook AI [24], in the same 20-client testbed as mentioned in Section 2.2. Essentially, FedBuff selects clients randomly and employs buffered aggregation where model aggregation is conducted when the number of received local updates exceeds a certain predefined threshold (more details in Section 5.1). Here we consistently set the aggregation threshold to be 40% of the concurrency limit across all cases.



(a) FEMNIST (Image) **(b) StackOverflow (LM)**

Figure 4: Scaling up the concurrency leads to diminishing gain in run time with escalating network usage.

In Figure 4, the time-to-accuracy/-perplexity reduces as the maximum allowed concurrency grows (dark lines), though at a *diminishing* speed. After a *turning* point (around 15 clients), the time-to-accuracy instead starts to inflate. On the other hand, we constantly observe a *superlinear* increase in the accumulated number of clients' updates when scaling up the maximum concurrency (light lines). Such a growing pattern indicates an escalating burden on network bandwidth and limitations in scalability. Thus, there exists a tradeoff between resource cost and runtime gain, and it is plausible to limit the concurrency to a small portion of the population.

4.2 Utility Profiling

While directly limiting the concurrency can improve resource efficiency, we take a step further by considering how to select clients for fully utilizing each concurrency quota.

Does random selection suffice? As stragglers can be tolerated, it first raises the question of whether random selection suffices for asynchronous FL before resorting to more complex methods. Our answer is no, as clients widely exhibit heterogeneous data distributions [22]. Even if clients have the same speed, it is still desirable to focus on clients with the most utility to improve global model quality for faster convergence. We empirically show the inferiority of random selection in Section 8.2.

Which clients to prioritize? We have left off the question of how to identify clients with the most *utility* to improve the global model. Starting from the state-of-the-art solution established for synchronous FL (§2.2), we need to address the following concerns arising from asynchronous FL:

- Given lifted tolerance for stragglers, do they still have to be strictly penalized in the chance of being selected?

- Given new training dynamics, does clients' data quality influence the global model in the same way?

Both of our answers are no. First, no individual client can impede the pace at which the global model updates. With the removal of synchronization barriers, there is no need to wait for stragglers to finish. Also, strictly penalizing stragglers risks precluding informative clients given the coupled nature of speeds and data quality (§2.2). Thus, *strictly penalizing stragglers can do more harm than good.*

On the other hand, a client's speed still affects its utility to improve the global model in an indirect way. The longer it takes a client to train, the more changes that the global model that it bases on is likely to undergo, because of other clients' contributions in the interim. In the literature, it is dubbed as *staleness* the lag between the version of the current global model and that of the locally used one. Empirical studies show that as the staleness of an update grows, the accuracy gained from incorporating that update will diminish [12, 21, 68]. Hence, *it is not helpful to select clients with high-quality data but also a large chance to produce stale updates.*

Combining the two insights, we formulate the utility of a client by respecting the roles that its data quality and staleness play in improving the global model:

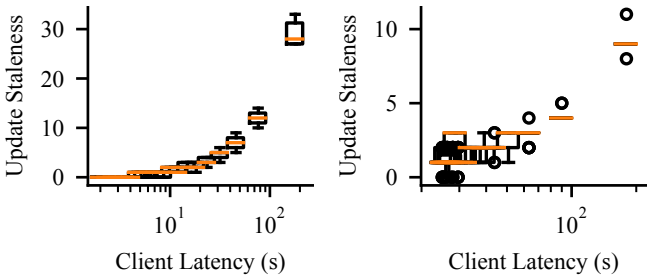
$$U_i^{Pisces} = \underbrace{|B_i|\sqrt{\frac{1}{|B_i|}\sum_{k\in B_i} Loss(k)^2}}_{Data\ quality} \times \underbrace{\frac{1}{(\tilde{\tau}_i+1)^{\beta}}}_{Staleness}, \qquad (2)$$

where $Loss(k)$ is the training loss of sample k from the client i's local dataset B_i, $\tilde{\tau}_i \geq 0$ is the estimated staleness of the client's updates and $\beta > 0$ is the staleness penalty factor. Based on the profiled utilities, Pisces sorts clients and selects the clients with the highest utilities to train.

Robustness against training loss outliers. The first term of Equation (2) stems from Oort's utility formula, i.e., Equation (1), that approximates the ML principle of *importance sampling* [29, 34] to sketch a client's data quality. We do not reinvent the formulation as its effectiveness does not vary across synchronization modes. Moreover, it features negligible computation overhead and privacy leakage.

Beyond the formulation, however, we need to tackle a *robustness challenge* unique to asynchronous FL. By definition, clients with high training loss are taken as possessing important data. In practice, a high loss could also result from the client's *corrupted data* or *malicious manipulation*. As a quick fix, Oort mitigates such cases by (i) adding randomness by probabilistic sampling and (ii) blacklisting clients who have been selected over a threshold of times.

Unfortunately, both strategies do not generalize to asynchronous FL. *First*, the server performs selection way more frequently, eliminating the benefits of probabilistic sampling. Suppose that asynchronous FL selects clients every 5 seconds and synchronous FL every 60 seconds. For a client with a probability of 0.01 to be selected in an attempt, the probability that it must be selected within five minutes is $1-(1-0.01)^5 \approx 5\%$ in synchronous FL, while being $1-(1-0.01)^{60} \approx 45\%$ in asynchronous FL. *Further*, as a client is generally involved more frequently, its participation can quickly reach the blacklist threshold. The client pool for selection can thus be exhausted in the late stage of training, hindering convergence.



(a) FEMNIST (Image) **(b) StackOverflow (LM)**

Figure 5: Clients' staleness varies slightly throughout the training regardless of their execution time.

Given the intuition that (i) the loss values of benign clients evolve in roughly the same direction, while (ii) those resulting from corrupted or malicious clients tend to be *outliers* consistently, we propose to blacklist clients whose losses have been outliers over a threshold of time. Initially, each client is given r *reliability credits*. For a client update which uses the global model of version w_t, Pisces pools the received client updates that are trained from similar initial versions of the global model, namely $\{w_{t-k}, w_{t-k+1}, \ldots, w_t\}$ for some $k > 0$, and runs DBSCAN [15] to *cluster* their associated losses. Each time a client's loss value is identified as an outlier, its credit gets deducted by one. A client will be removed from the client pool when it runs out of reliability credit. As validated, Pisces can reduce anomalous updates while avoiding blindly blacklisting benign clients (§8.3).

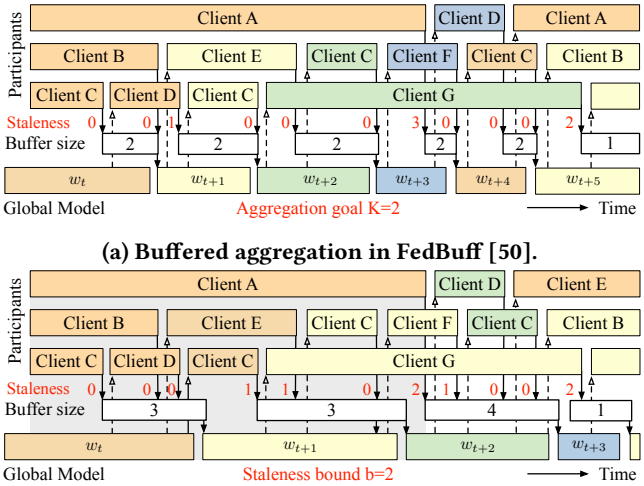
Staleness-aware discounting. As the second term in Equation (2) shows, Pisces *discounts* a client's utility based on its estimated staleness of its local updates. We adopt the reciprocal of a polynomial function for realizing: 1) *functionality*: given the same data quality, the client with larger staleness should get a larger discount for being selected less likely; and 2) *numerical stability*: the speed at which the discount inflates should decrease as the staleness goes infinite.

It remains a question of how to estimate the actual staleness τ_i in practice. By definition, its exact value is unknown until the client returns its update, which happens after the

selection. While it may be precisely inferred through elaborate simulation of the federation, doing so is impractical due to the need for accurate knowledge of clients' speeds. Given that Equation (2) is not designed to work with an accurate estimation on τ_i, we advocate *approximating it with historical records*. Specifically, Pisces lets $\tilde{\tau}_i$ be the moving average of the most recent k actual values $\tau_{i,t-k+1}, \tau_{i,t-k+2}, \ldots \tau_{i,t}$, i.e.,

$$\tilde{\tau}_i = \frac{1}{k} \sum_{j=t-k+1}^{t} \tau_{i,j}. \tag{3}$$

The intuition behind the approximation is that the staleness of a client's updates is usually *stable* over time, given the stability of (1) clients' execution times and (2) the frequency of model aggregation. To exemplify, we study the staleness behaviors associated with the experiments mentioned in Section 4.1. We use 15 as the concurrency limit without loss of generality. As depicted in Figure 5, during the training, the staleness of each client slightly fluctuates around the median, with the maximum range of individual values being 6 and 3 for FEMNIST and StackOverflow, respectively.



(a) Buffered aggregation in FedBuff [50].

(b) Pisces's adaptive pace control.

Figure 6: Two methods for steering aggregation pace.

5 MODEL AGGREGATION

While Pisces's optimizations in participant selection can reduce stale computation, it cannot thoroughly prevent its occurrences. As for stale updates already generated, we further optimize model aggregation to prevent the global model from being arbitrarily impaired. In this section, we first start with the goal of bounded staleness and discuss the limitations of existing fixes (§5.1). We then elaborate on Pisces's principles on performing adaptive aggregation pace control for efficiently bounding clients' staleness (§5.2).

5.1 Bounded Staleness

As mentioned in Section 4.2, local updates with larger staleness empirically bring less gain in model convergence. This fact has a theoretical ground as stated below, consolidating the first-order goal of bounding staleness in aggregation.

Why desire bounded staleness? The mainstream way to derive a convergence guarantee for asynchronous FL is based on the perturbed iterate framework [47, 50]. In addition to the assumptions that are commonly made in synchronous FL, it also requires the following assumption to hold:

Assumption 1. *(Bounded Staleness) For all clients $i \in [N]$ and for each Pisces server's loop step, the staleness τ_i between the model version in which client i uses to start local training, and the model version in which the respective update is used to modify the global model is not larger than $\tau_{\max}$.*

We here sketch the intuition on how bounded staleness helps convergence. By limiting the staleness of each client's updates all the time, we can *limit the model divergence* between any version of the global model w_t and the initial model that any corresponding contributor of w_t uses. This implies that the contributors' gradients do not deviate much from each other. Consequently, each model aggregation attempt does take an effective step towards the training objective, and the training can thus terminate with finite times of aggregation. We provide more details in Section 6.1.

Does buffered aggregation suffice? To control clients' staleness, the state-of-the-art asynchronous FL design, i.e., FedBuff (mentioned in Section 4.1), adopts *buffered aggregation (BA)*. The server uses a buffer to store local updates and only aggregates when the buffer size reaches a certain aggregation goal $K > 1$, as illustrated in Figure 6a.

Still, due to the lack of explicit control, the maximum staleness across clients in BA can *go unbounded*, to which extent depends on the heterogeneity degree of clients' speeds. For example, for the experiments reported by Figure 5, the fastest client is 90× faster than the slowest in FEMNIST and 7.2× in StackOverflow. Thus, despite using the same aggregation goal ($K = 6$), the maximum staleness values differ vastly (33 in FEMNIST and 11 in StackOverflow). This implies the need for *manually tuning* the aggregation goal across different federation environments and learning tasks, which can be too expensive or even prohibited.

5.2 Adaptive Pace Control

To generally enforce a staleness bound, we develop an adaptive strategy for steering the pace of model aggregation.

Input: Running clients R, target staleness bound $b > 0$, last aggregation time $t_{\tau_{j-1}}$, current time T_l, clients' profiled latencies $\{L_i\}_{i\in[n]}$
Output: Aggregation decision for the loop step l

```
Function ManagerToAggregate()
    /* Set the aggregation interval proportionate to the
       profiled latency of the slowest running client. */
    L_max = max_{i∈R} L_i
    I = L_max / b
    /* Aggregate if the interval currently ends. */
    return T_l − t_{τ_{j−1}} > I
```

Algorithm 2: Pisces's adaptive aggregation pace control.

How to be adaptive to complex dynamics? In general, the distribution of the staleness values across clients is determined by the interplay between (1) the algorithms used (for both selection and aggregation), and (2) the dynamic environment (e.g., clients' speeds or data quality). While accounting for the whole population is overwhelming, we can reduce the problem to a narrowly scoped one that only considers a single client. Our insight is that to enforce a staleness bound for all clients throughout the training, it suffices to bound the staleness of the slowest running client for each time unit. We thus propose to keep track of the running client with the largest profiled latency and adjust the *aggregation interval*, i.e., the time between two consecutive model aggregations, in a way that the aggregation pace can match the slowest client's speed.

How to adjust the aggregation interval? We further develop the idea with a concrete example as shown in Figure 6b. We focus on the time period where Client A performs local training (highlighted with grey shadow). During this process, A is consistently the slowest running client. Thus, by ensuring that $m \leq 2$ model updates take place during A's training, we can guarantee the same staleness bound for any other clients running within this period.

It only leaves off the question of when should these m aggregations happen. Our intuition is that, arranging them *evenly in terms of time* can balance the numbers of contributors across different aggregation attempts in expectation, helping the global model evolve smoothly. Also, a (nearly) uniform distribution of the server's aggregation workload can sustain better scalability.

Latency-aware aggregation interval. By generalizing the above idea to real deployments where the identity of the slowest running client changes over time, we obtain Pisces's principles on steering the model aggregation pace. In essence, Pisces periodically examines the necessity of aggregation in the control loop (§3). As outlined in Algorithm 2, for a loop step l that begins at time T_l, Pisces first fetches the profiled latency L_{max} of the slowest client that is currently running. It then determines the aggregation interval I suitable to bound the client's staleness based on the above intuition. Finally, it computes the elapsed time $e = T_l - t_{\tau_{j-1}}$ since the last model aggregation happened. Only when e is larger than the interval I will Pisces suggest performing model aggregation in this step. In practice, clients' latencies can be profiled with historical records. As Section 6.1 shows, assuming accurate latency predictions, Algorithm 2 helps achieve bounded staleness.

6 THEORETICAL ANALYSIS

In this section, we first prove that Pisces guarantees bounded staleness and then present Pisces's convergence guarantee (§6.1). Next, we analyze the complexity of computation, communication, and storage of Pisces (§6.2).

6.1 Convergence Guarantee

Notation. We let T_l denote the start time of a loop step l, I_l denote the aggregation interval generated by Algorithm 2 at step l, n denote the total number of clients, b denote the target staleness bound used by Pisces, $\nabla F_i(w)$ denote the gradient of model w with respect to the loss of client i's data, $f(w) = \frac{1}{n}\sum_{i=1}^{n} p_i F_i(w)$ denote the global learning objective with $p_i > 0$ weighting each client's contribution, f^* denote the optimum of $f(w)$, $g_i(w;\xi_i)$ denote the stochastic gradient on client i with randomness ξ_i, and Q denote the number of steps in local SGD.

Why does Pisces achieve bounded staleness? We first state a useful lemma that globally holds for each loop step.

Lemma 1. *For any loop step l where model aggregation happens, there must be no model aggregation happening during the time range $[T_l - I_l, T_l)$.*

Proof. Assume that model aggregation takes place once at time $t \in [T_l - I_l, T_l)$, the elapsed time since this aggregation starts is then $T_l - t \leq I_l$. However, by the design of Algorithm 2, there should be no aggregation in step l, which leads to a contradiction. □

We are then able to derive the bounded staleness property.

Theorem 1. *Executing Algorithm 2 for all loop steps $l = 1, \cdots, L$, the maximum number of model aggregations happening during any training process of any client $i \in [n]$ is no more than b, if the profiled latencies $\{L_i\}_{i\in[n]}$ are accurate.*

Proof. Consider a training process of client i that lasts for L_i. In the interim, assume that there are $m \in \mathbb{N}$ model

aggregations performed by the server, each of which happens in the loop step l_k where $k \in [1, m]$.

Now, applying Lemma 1, the duration between the 1st and the m-th aggregation has a lower bound $T_{l_m} - T_{l_1} > \sum_{j=2}^{m} I_{l_j}$. By definition, $I_{l_j} = L_{max,l_j}/b$ where L_{max,l_j} is the end-to-end latency of the slowest client running at step l_j. Given that $L_{max,l_j} \geq L_i$, we further have $T_{l_m} - T_{l_1} > (m-1)L_i/b$.

Meanwhile, we also have $L_i \geq T_{l_m} - T_{l_1}$, as the two aggregations all happen in this training process. Combining the two inequalities yields $1 > (m-1)/b$, i.e., $m < b$. □

What is the convergence guarantee? We additionally make the following assumptions which are commonly made in analyzing FL algorithms [45, 56, 61, 71].

Assumption 2. *(Unbiasedness of client stochastic gradient)* $\mathbb{E}_{\zeta_i}[g_i(w;\zeta_i)] = \nabla F_i(w)$.

Assumption 3. *(Bounded local and global variance) for all clients* $i \in [1, n]$, $\mathbb{E}_{\zeta_i|i}[\|g_i(w;\zeta_i) - \nabla F_i(w)\|^2] \leq \sigma_\ell^2$, *and* $\frac{1}{n}\sum_{i=1}^{n} \|\nabla F_i(w) - \nabla f(w)\|^2 \leq \sigma_g^2$.

Assumption 4. *(Bounded gradient)* $\|\nabla F_i\|^2 \leq G$, $i \in [1, n]$.

Assumption 5. *(Lipschitz gradient) for all client* $i \in [1, n]$, *the gradient is L-smooth* $\|\nabla F_i(w) - \nabla F_i(w')\|^2 \leq L\|w - w'\|^2$.

Given the five assumptions, by substituting the use of maximum delay $\tau_{max,K}$ with our enforced staleness bound b and using a constant server learning rate $\eta_g = 1$ (as we execute Federated Averaging [48]) in [50]'s proof, we can derive the convergence guarantee for Pisces as follows.

Theorem 2. *Let* $\eta_\ell^{(q)}$ *be the local learning rate of client SGD in the q-th step, and define* $\alpha(Q) := \sum_{q=0}^{Q-1} \eta_\ell^{(q)}$, $\beta(Q) := \sum_{q=0}^{Q-1} (\eta_\ell^{(q)})^2$. *Choosing* $\eta_\ell^{(q)} Q \leq \frac{1}{L}$ *for all local steps* $q = 0, \cdots, Q-1$, *the global model iterates in Pisces achieves the following ergodic convergence rate*

$$\frac{1}{T}\sum_{t=0}^{T-1} \left\|\nabla f(w^t)\right\|^2 \leq \frac{2\left(f(w^0) - f^*\right)}{\alpha(Q)T} + \frac{L}{2}\frac{\beta(Q)}{\alpha(Q)}\sigma_\ell^2 + 3L^2 Q\beta(Q)\left(b^2+1\right)\left(\sigma_\ell^2 + \sigma_g^2 + G\right). \tag{4}$$

Theorem 2 derives an upper bound for the ergodic norm-squared of the gradient, which diminishes at a rate slightly slower than $O(1/T)$ as the number of global aggregations T grows. Thus, Pisces can converge towards a first-order stationary point in non-convex stochastic optimization.

6.2 Performance Analysis

We next analyze the additional cost that Pisces imposes on an FL workflow. All calculations below assume a single server and n clients in total. We ignore clients' cost as Pisces induces negligible overhead to each local training process,[2] and focus on each loop step performed by the server.

Computation cost: $O(n^2)$. The computation cost can be broken down as: (1) running Algorithm 2 to decide whether to aggregate, which is $O(1)$, (2) updating the utility score for clients who are involved in model aggregation, if any, which is dominated by $O(1)$ attempts of detecting training loss outliers with DBSCAN and thus takes $O(n^2)$ time, and (3) selecting clients to train based on the rankings of their utility scores, which is $O(n \log n)$.

Communication cost: $O(n)$. The server additionally collects the aggregate training loss from each client who reports her local updates, which costs $O(n)$ in total.

Storage cost: $O(n^2)$. The storage cost stems from (1) caching clients' profiled latencies, estimated staleness values, and computed utility scores, which is $O(n)$, (2) ranking clients by their utility scores, which is $O(n)$, and (3) running DBSCAN to detect training loss outliers, which is $O(n^2)$.

7 IMPLEMENTATION

We implement Pisces atop Plato [64], an open-source framework for scalable FL research, with 2.1k lines of code. Plato abstracts away the underlying ML driver with handy APIs, with which we can seamlessly port Pisces to infrastructures including PyTorch [53], Tensorflow [2], and MindSpore [49]. On the other hand, at the time of engineering, Plato did not support asynchronous FL and participant selection. Thus, we first implement the coordinator's control loop and client manager for enhanced functionality. For fairness of model testing overhead across different synchronization modes, we take the testing logic from the main loop to a concurrent background process. Further, to emulate arbitrary distributions of clients' speeds given finite hardware specifications, we instrument Plato to simulate client latencies by controlling exactly when a received local update is "visible" to the FL protocol. This enables us to control clients' latencies in a fine-grained manner, as shown in Section 8.1. The entire codebase of Pisces is open-sourced at [1].

8 EVALUATION

We evaluate Pisces's effectiveness in various FL training tasks. The highlights of our evaluation are listed as follows.

[2]The training loss of each sample is inherently available in ML training processes. Also, both the cost of aggregating training losses and that of reporting the aggregate do not grow with more clients.

(1) Pisces **outperforms Oort and FedBuff** by 1.2−2.0× in time-to-accuracy performance. It gains high training efficiency by automating participant selection and model aggregation in a principled way (§8.2).
(2) Pisces is superior to its counterparts over different **participation scales** and **degrees of system heterogeneity** across clients. Its efficiency is also shown to be **insensitive** to choice of the hyperparameters (§8.3).

8.1 Methodology

Datasets and models. We run two categories of applications with four real-world datasets of diverse scales.

- *Image Classification:* the CIFAR-10 dataset [37] with 60k colored images in 10 classes, the MNIST dataset [13] with 60k greyscale images in 10 classes, and a larger dataset, FEMNIST [7], with 805k greyscale images in 62 categories collected from 3.5k data owners. We train LeNet5 [40] to classify the images in MNIST and FEMNIST and use ResNet-18 [20] for CIFAR-10.
- *Language Modeling:* a large-scale StackOverflow [30] dataset contributed by 0.3 million clients. We train Albert [39] over it for next-word prediction.

Experimental Setup. We launch an AWS EC2 `c5.4xlarge` instance (16 vCPUs and 32 GB memory) for the server. We use another 20 `c5.4xlarge` instances to emulate 200 clients, where each instance is used to run 10 clients for high utilization without resource contention. As each instance has the same resource configuration, all clients are created with uniform processing speed. To simulate system heterogeneity, we configure their processing latency to follow the Zipf distribution [26, 41, 63] with $a = 1.2$ using the execution barriers mentioned in §7, i.e., the end-to-end latency of the i-th slowest client is proportional to i^{-a}. This models a practical scenario where most clients are fast and only a few are stragglers. To additionally introduce data heterogeneity, we resort to either of the two solutions:

- *Synthetic Datasets.* For datasets that are used in conventional ML (MNIST and CIFAR-10), we apply latent Dirichlet allocation (LDA) over data labels for each client as in the literature [3, 5, 23, 56]. The concentration parameters are all set to be 1.0 such that label distributions are highly skewed across clients.
- *Realistic Datasets.* For datasets collected in real distributed scenarios (FEMNIST and StackOverflow), as they have been partitioned by the original data owners, we directly allocate one partition to a client to preserve the native non-IID properties.

Note that data quality and system speed are configured independently here, instead of inversely as in Section 2.2.

Table 1: Summary of the hyperparameters.

Parameters	MNIST	FEMNIST	CIFAR-10	StackOverflow
Local epochs	5	5	1	2
Batch size	32	32	128	20
Learning rate	0.01	0.01	0.01	0.00008
Weight decay	0	0	0.0001	0.0001

Hyperparameters. The concurrency limit is $C = 20$ (§4.1; shared with Oort and FedBuff), and staleness penalty factor is $\beta = 0.5$ (§4.2). **The target staleness bound b (§5.2) always equates** C. We use SGD with momentum set as 0.9 except for StackOverflow where Adam [36] is used, unless otherwise stated. Other configurations are listed at Table 1.

Baseline Algorithms. We evaluate Pisces against Oort [38], the state-of-the-art optimized solution for synchronous FL, and FedBuff [50], the cutting-edge asynchronous FL algorithm. We use the default set of hyperparameters for Oort. The aggregation goal of FedBuff is set to 20% of the concurrency limit, according to the authors' suggestions.

Metrics. We primarily care about the *elapsed time* taken to reach the target accuracy. To identify the source of performance differences across algorithms, we also record the *number of involvements* of each client and the *number of aggregations* performed by the server. We further measure the *network footprint* for assessing the practicality.

8.2 End-to-End Performance

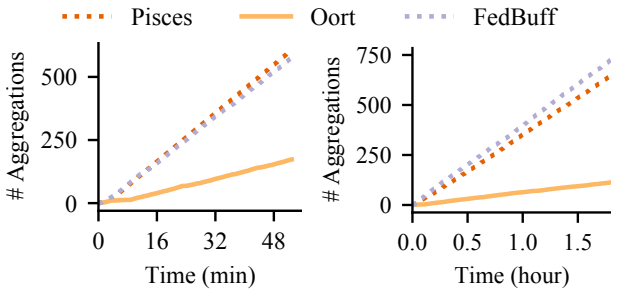
We start with Pisces' time-to-accuracy performance. We first highlight its improvements over Oort and FedBuff. We next break down the source of efficiency in Pisces. We further access the network footprint of all algorithms.

Pisces improves time-to-accuracy. Table 2 summarizes the speedups of Pisces over the baselines, where Pisces reaches the target 1.2-2.0× faster on the three image classification tasks. A consistent speedup of 1.9× can be observed in language modeling. To understand the source of such improvements, we first compare Pisces against Oort. As depicted in Figure 7, the accumulated number of model aggregations in Pisces is consistently larger than that in Oort (as is also the case when FedBuff is compared to Oort). This confirms the advantages of removing the synchronization barriers.

On the other hand, albeit with comparable update frequency as in Pisces, FedBuff improves over Oort to a milder degree. The key downside of FedBuff lies in its random selection method. As shown in Figure 8, Pisces prefers clients with large datasets, while FedBuff shows barely any difference in the interests of clients. Given the intuition that clients with larger datasets have higher potential to improve the global model, Pisces makes more efficient use of concurrency

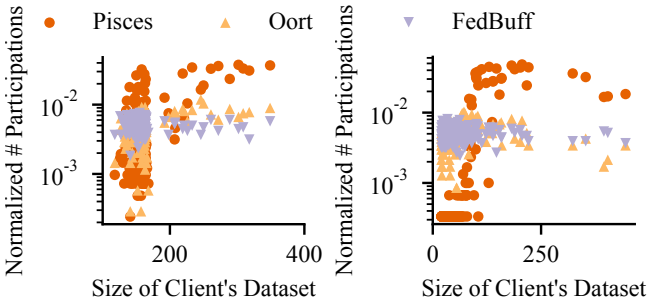
Table 2: Summary of Pisces's improvements on the time-to-accuracy performance.

Task	Dataset	Target Accuracy	Model	Time-to-Accuracy		
				Pisces	Oort [38]	FedBuff [50]
Image Classification	MNIST [13]	97.8%	LeNet-5 [40]	6.2min	12.8min (2.0×)	7.6min (1.2×)
	FEMNIST [7]	60.0%	LeNet-5	8.9min	16.0min (1.8×)	12.6min (1.4×)
	CIFAR-10 [37]	65.1%	ResNet-18 [20]	24.5min	40.3min (1.6×)	26.5min (1.1×)
Language Modeling	StackOverflow [30]	800 perplexity	Albert [39]	25.0min	48.4min (1.9×)	47.2min (1.9×)



(a) FEMNIST (Image) (b) StackOverflow (LM)

Figure 7: Pisces performs model aggregation more frequently than Oort for being asynchronous.

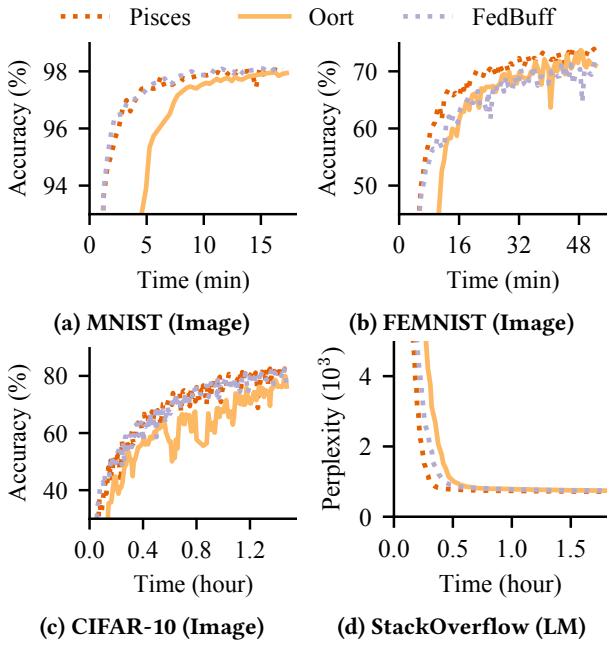


(a) FEMNIST (Image) (b) StackOverflow (LM)

Figure 8: Pisces selects informative clients more frequently than FedBuff with principled selection.

quotas than FedBuff does. Oort also differentiates clients by data quality, though to a more moderate extent as it has to reconcile for clients' speeds.

Pisces exhibits stable convergence behaviors. Figure 9 further plots the learning curves of different protocols. First, when reaching the corresponding time limit, Pisces achieves 0.2%, 2.9%, and 0.8% higher accuracy on MNIST, FEMNIST, and CIFAR-10, respectively, and 23 lower perplexity on StackOverflow, as compared to Oort. In other words, apart from theoretical convergence guarantees on convergence, Pisces



(a) MNIST (Image) (b) FEMNIST (Image)
(c) CIFAR-10 (Image) (d) StackOverflow (LM)

Figure 9: Elaboration on the convergence behaviors.

is also empirically shown to achieve comparative final model quality with that in Oort (synchronous FL).

Moreover, in all evaluated cases, Pisces' model quality evolves more stably than that in Oort, especially in the middle and late stages. Pisces' stability advantage probably stems from the noise in local updates induced by moderately stale computation, which is analogous to the one introduced to avoid overfitting in traditional ML [60]. This conjecture complies with the fact that FedBuff also exhibits stable convergence, though sometimes to an inferior model quality (3.1% and 0.4% lower accuracy than Pisces on FEMNIST and CIFAR-10, respectively, and 24 higher perplexity on StackOverflow).

Pisces' optimizations on participant selection are effective. To examine which part of Pisces's participant selection optimizations to accredit with, we break it down and formulate two variants: (i) *Pisces w/o slt.*: we disable Pisces's selection strategies and sample randomly instead; (ii) *Pisces*

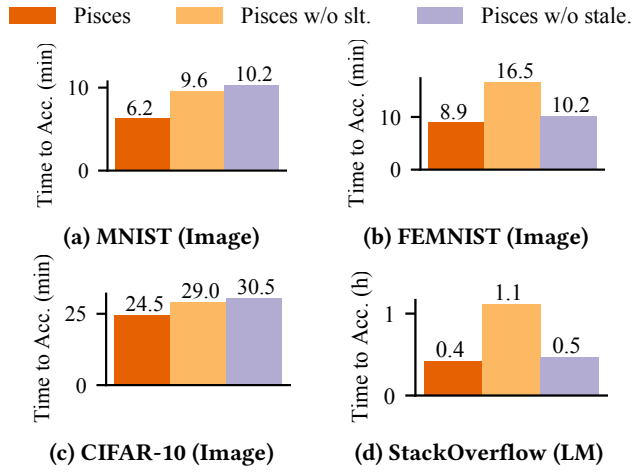


Figure 10: Breakdown of participant selection designs.

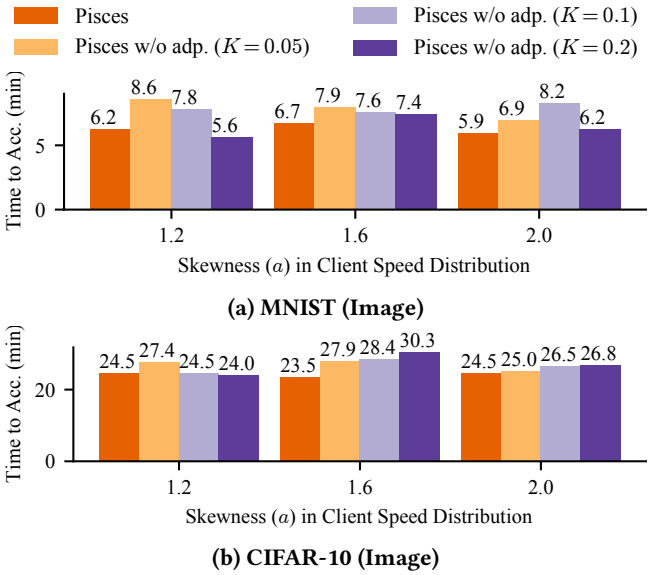


Figure 11: Aggregation adapts to speed heterogeneity.

w/o stale.: while we still select clients based on their data quality, we ignore the impact that clients' staleness has on clients' utilities, as if the second term in Equation (2) is consistently set to be one. Figure 10 reports the comparison of complete Pisces with these two variants.

As Pisces selects clients with high data quality and low potential of inducing stale computation, it improves the time-to-accuracy over *Pisces w/o slt.* by 1.1-2.7×. Further, the criteria on data quality are more critical than those on staleness dynamics, as Pisces is shown to enhance the performance of *Pisces w/o stale.* less significantly (1.1-1.6×). Understandably, with adaptive pace control in model aggregation, the pressure of avoiding stale computation in participant selection is partially released. Still, the combined considerations of the two factors yield the best efficiency.

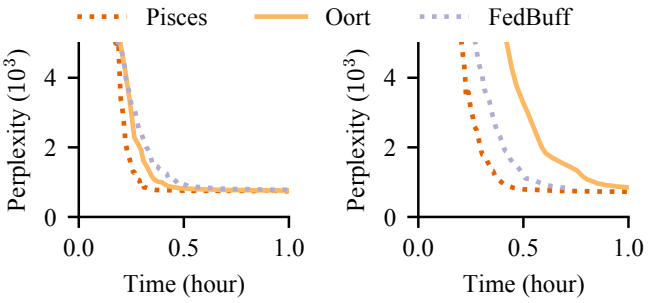
Pisces's optimizations on aggregation adapt to various settings. To understand the benefits of adaptive pace control, we also compare Pisces against another variant: (i) *Pisces w/o adp.*: we disable Pisces's adaptive pace control and instead resort to buffered aggregation (§5.1) with various choices of the aggregation goal K: 5%, 10% and 20% of the concurrency limit C, representing gradually decreasing aggregation frequencies. As mentioned in Section 5.2, clients' staleness dynamics partly depend on the distributions of clients' speeds. We thus also vary the skewness of client latency distribution by using different a's in the Zipf distribution: 1.2 (moderate), 1.6 (high), and 2.0 (heavy).

As shown in Figure 11, Pisces improves over *Pisces w/o adp.* by up to 1.4× both when $a = 1.6$ and when $a = 2.0$, while it slightly slows down *Pisces w/o adp.* by at most 1.1× when $a = 1.2$. Further, across various degrees of client speed heterogeneity, *Pisces w/o adp.* exhibits unstable efficiency as sticking to any of K does not yield the best performance for all cases. Thus, buffered aggregation relies on the manual tuning of K to adapt to different tasks and environments, while Pisces's adaptive pace control is more preferable with (1) a deterministic bound of clients' staleness for theoretically guaranteed efficiency and (2) one parameter b which does not require to tune.

Table 3: Total network footprint (GB) at the server end when reaching the target accuracy.

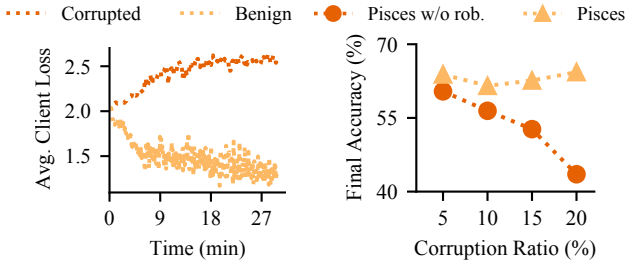
Task	MNIST	FEMNIST	CIFAR-10	StackOverflow
Pisces	**0.162**	**0.346**	108.501	**77.668**
Oort	0.197	0.406	**96.742**	119.229
FedBuff	0.199	0.474	114.631	193.466

Pisces is also competitive in network overhead. Apart from the runtime, Table 3 further compares the server's accumulated network footprint (including sent and received packets) in different algorithms when the global model reaches the target accuracy. Although Pisces performs participant selection and model aggregation more frequently than Oort per time unit (Figure 7), it achieves the target accuracy with a slightly higher training time of 12% in CIFAR-10 or even shortening it by 15-35% in the other tasks, thanks to its significantly reduced time-to-accuracy (Table 2). On the other hand, despite also improving the time-to-accuracy, FedBuff consistently results in a higher network footprint than Oort (by up to 60%). Thus, Pisces is still competitive in terms of network overhead when compared to Oort and FedBuff.



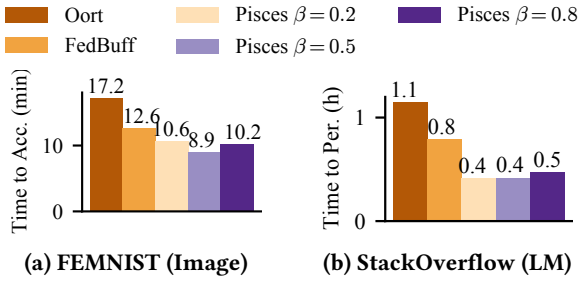
(a) StackOverflow (C = 100) (b) StackOverflow (C = 400)

Figure 12: Pisces outperforms in various scales.



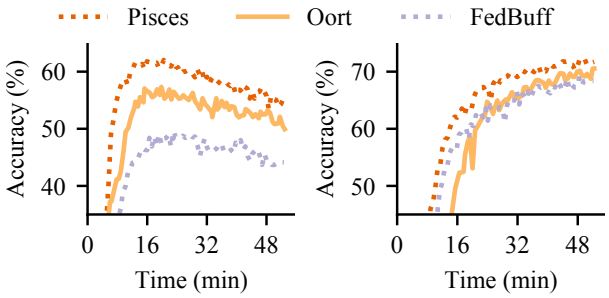
(a) Loss Dynamics (b) Robustness

Figure 13: Pisces is robust against corrupted clients.



(a) FEMNIST (Image) (b) StackOverflow (LM)

Figure 14: Selection with various staleness penalties.



(a) Adam [36] (b) FedProx [43]

Figure 15: Training FEMNIST with various optimizers.

8.3 Sensitivity Analysis

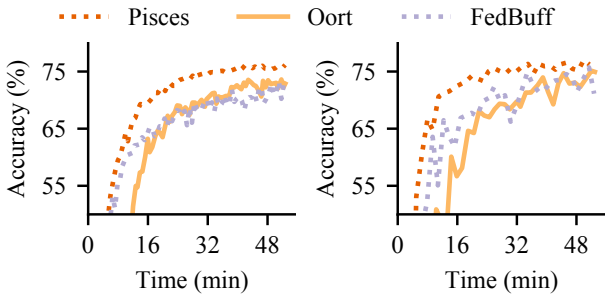
We also examine Pisces's effectiveness across various environments and configurations. All the results are based on the same accuracy targets mentioned in Section 8.2.

Impact of participation scale. Besides a pool of 200 clients with the concurrency limit being 20 (N=200 with C=20 in §8.2), we evaluate Pisces on two more scales of participation: N=100 with C=10, and N=400 with C=40. As shown in Figure 12, as the number of clients scales up, Pisces outperforms Oort (resp. FedBuff) in time-to-perplexity by 1.7× (resp. 2.1×), 1.9× (resp. 1.9×), and 2.4× (resp. 1.6×) for N=100, N=200, and N=400, respectively. The key reason for Pisces's stable performance benefits is that its algorithmic designs are agnostic to the population size. Further, Pisces enhances its scalability by (1) enforcing a concurrency limit (§4.1), and (2) adaptively balancing the aggregation workload (§5.2).

Impact of training loss outliers. To validate the robustness to training loss outliers, we insert manual corruption into the FEMNIST dataset. We follow the literature on adversarial attacks [16, 38] to randomly flip all the labels of a subset of clients. This leads to consistently higher losses from corrupted clients than the majority, as exemplified in Figure 13a where 5% clients are corrupted. We compare Pisces against *Pisces w/o rob.*, a variant of Pisces where anomalies are not identified and precluded. As shown in Figure 13b, Pisces outperforms it in the final accuracy for accurately precluding outlier clients (§4.2) across various scales of corruption.

Impact of staleness penalty factor. We next examine Pisces under different choices of the staleness penalty factor, β, which is introduced to prevent stale computation in participant selection (§4.2). We set $\beta = 0.2$ and $\beta = 0.8$ in addition to the primary choice where $\beta = 0.5$, where a large β can be interpreted as a stronger motivation for Pisces to avoid stale computation. As depicted in Figure 14, while different FL tasks may have different optimal choices of β (e.g., around 0.5 for FEMNIST and 0.2 for StackOverflow), Pisces still improves performance across different uses of β.

Impact of optimizers. We next show the superior performance of Pisces over its counterparts under different choices of optimizers. In addition to SGD (with learning curves shown in Figure 9b), we additionally train over FEMNIST with the Adam [36] and FedProx [43] optimizers. We configure the penalty constant $\mu = 1.0$ in FedProx following the authors' empirical suggestion [43]. In Figure 15, we observe that Pisces still improves convergence speed and final model accuracy across different optimizers. We additionally note that Adam displays a faster initial process than SGD and FedProx but quickly fails to keep up the progress. This complies with the observations made in the literature [58, 66],



(a) Customized CNN [4, 31] **(b) ResNet-18 [20]**

Figure 16: Training FEMNIST with various models.

which suggests that adaptive optimizers may not always excel but falls out of the scope of this paper.

Impact of model architectures. In this experiment, we investigate the effectiveness of Pisces when training diverse models over FEMNIST. Beyond LeNet5 (0.06M parameters with learning curves shown in Figure 9b), we also evaluate two models of different scales: a customized CNN [4, 31] (1M parameters) and the ResNet-18 [20] model (11M parameters). As shown in Figure 16, Pisces's superiority in the time-to-accuracy performance is still noticeable across different uses of models. Also, Pisces exhibits higher final model accuracy and more stable convergence behaviors.

9 DISCUSSION AND FUTURE WORK

Generic FL training framework. The model of computation that Pisces's architecture embodies, i.e., the three components (Figure 3) and the interactions (Algorithm 1), can be used to instantiate a wide range of FL protocols. Synchronous FL also fits in this model by ensuring that (1) aggregation starts when no client is running, and (2) selection starts right after an aggregation. Given the expressiveness, we present it as a *generic FL training framework* for helping FL developers compare various protocols more clearly and fairly.

Privacy compliance. During a client's participation in Pisces, the possible sources of information leakage are two-fold–the average training loss and local update that it reports to the server. For the current release of the plaintext training loss, we do not enlarge the attack surface compared to synchronous FL deployments [19, 38, 70], and thus can also adopt the same techniques as theirs to enhance privacy.

To prevent the local update from leaking sensitive information, Pisces can resort to differential privacy (DP) [14], a rigorous measure of information disclosure about clients' participation in FL. Traditional DP methods [4, 27, 31] require precise control over which clients to contribute to a server update, which is not the case in asynchronous FL. However, Pisces is compatible with DP-FTRL [32], a newly emerged DP solution for privately releasing the prefix sum of a data stream. While we focus on improving FL efficiency and scalability, we plan to integrate Pisces with DP as future work.

10 RELATED WORK

To accommodate data heterogeneity in asynchronous FL, many efforts propose to regularize local optimization for controlling the deviations of local models from the global one. For example, FedAsync [68] applies a surrogate objective to each client to regularize the L2-Norm between her local model and the global one. ASO-Fed [10] considers a similar design in the context of online FL learning. Pisces complements these studies with informed participant selection and adaptive aggregation pace control.

To reduce stale computation, researchers study how to steer the pace of aggregation. For example, both HySync [57] and FedBuff [50] consider buffered aggregation (BA) (§5.1). In particular, HySync assumes full client concurrency and adapts the aggregation interval to avoid aggregating too many updates at a time. FedBuff deals with controlled concurrency and uses a fixed-size buffer for aggregation. Moreover, Port [62] handles stale computation by forcing stale clients to abort halfway and report their intermediate local updates for aggressive aggregation. Unlike Pisces, these systems neither guarantee bounded staleness in model aggregation, nor do they devise participant selection strategies for improved resource efficiency.

There also exist several proposals which adopt principled participant selection. For example, both FedAR [25] and Chen's work [11] select clients based on their resource capacity and contributions to improving the global model. Such work does not optimize model aggregation and thus cannot reap the most benefits of asynchrony as Pisces does.

11 CONCLUSION

While the tradeoff between clients' speeds and data quality is knotty to navigate in synchronous FL, resorting to asynchronous FL requires retaining resource efficiency and avoiding stale computation. In this paper, we present practical principles for automating participant selection and model aggregation with Pisces. Compared to prior arts, Pisces features noticeable speedups, robustness, and flexibility.

ACKNOWLEDGEMENT

We thank our shepherd Patrizio Dazzi and the anonymous reviewers for their valuable feedback that helps improve the quality of this work. This research was supported in part by ACCESS – AI Chip Center for Emerging Smart Systems, Hong Kong SAR, RGC RIF grant R6021-20, and RGC GRF grants under the contracts 16209120 and 16200221.

REFERENCES

[1] [n.d.]. Pisces codebase. https://github.com/SamuelGong/Pisces.

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

[3] Durmus Alp Emre Acar, Yue Zhao, Ramon Matas Navarro, Matthew Mattina, Paul N Whatmough, and Venkatesh Saligrama. 2021. Federated learning based on dynamic regularization. In *ICLR*.

[4] Naman Agarwal, Peter Kairouz, and Ziyu Liu. 2021. The skellam mechanism for differentially private federated learning. In *NeurIPS*.

[5] Maruan Al-Shedivat, Jennifer Gillenwater, Eric Xing, and Afshin Rostamizadeh. 2020. Federated Learning via Posterior Averaging: A New Perspective and Practical Algorithms. In *ICLR*.

[6] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloe Kiddon, Jakub Konečnỳ, Stefano Mazzocchi, H Brendan McMahan, et al. 2019. Towards federated learning at scale: System design. In *MLSys*.

[7] Sebastian Caldas, Sai Meher Karthik Duddu, Peter Wu, Tian Li, Jakub Konečnỳ, H Brendan McMahan, Virginia Smith, and Ameet Talwalkar. 2019. Leaf: A benchmark for federated settings. In *NeurIPS Workshop*.

[8] Zheng Chai, Ahsan Ali, Syed Zawad, Stacey Truex, Ali Anwar, Nathalie Baracaldo, Yi Zhou, Heiko Ludwig, Feng Yan, and Yue Cheng. 2020. Tifl: A tier-based federated learning system. In *HPDC*.

[9] Mingqing Chen, Rajiv Mathews, Tom Ouyang, and Françoise Beaufays. 2019. Federated learning of out-of-vocabulary words. *arXiv:1903.10635* (2019).

[10] Yujing Chen, Yue Ning, Martin Slawski, and Huzefa Rangwala. 2020. Asynchronous online federated learning for edge devices with non-iid data. In *Big Data*.

[11] Zheyi Chen, Weixian Liao, Kun Hua, Chao Lu, and Wei Yu. 2021. Towards asynchronous federated learning for heterogeneous edge-powered internet of things. In *DCN*.

[12] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R Ganger, Phillip B Gibbons, et al. 2014. Exploiting bounded staleness to speed up big data analytics. In *ATC*.

[13] Li Deng. 2012. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine* 29, 6 (2012), 141–142.

[14] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006. Calibrating noise to sensitivity in private data analysis. In *Theory of cryptography conference*. Springer.

[15] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise.. In *KDD*.

[16] Minghong Fang, Xiaoyu Cao, Jinyuan Jia, and Neil Gong. 2020. Local model poisoning attacks to {Byzantine-Robust} federated learning. In *USENIX Security*.

[17] Google. 2021. Your chats stay private while messages improves suggestions. https://support.google.com/messages/answer/9327902.

[18] Andrew Hard, Kanishka Rao, Rajiv Mathews, Swaroop Ramaswamy, Françoise Beaufays, Sean Augenstein, Hubert Eichner, Chloé Kiddon, and Daniel Ramage. 2018. Federated learning for mobile keyboard prediction. *arXiv:1811.03604* (2018).

[19] Florian Hartmann, Sunah Suh, Arkadiusz Komarzewski, Tim D Smith, and Ilana Segall. 2019. Federated learning for ranking browser history suggestions. *arXiv:1911.11807* (2019).

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

[21] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. 2013. More effective distributed ml via a stale synchronous parallel parameter server. In *NeurIPS*.

[22] Kevin Hsieh, Amar Phanishayee, Onur Mutlu, and Phillip Gibbons. 2020. The non-iid data quagmire of decentralized machine learning. In *ICML*.

[23] Tzu-Ming Harry Hsu, Hang Qi, and Matthew Brown. 2019. Measuring the effects of non-identical data distribution for federated visual classification. *arXiv:1909.06335* (2019).

[24] Dzmitry Huba, John Nguyen, Kshitiz Malik, Ruiyu Zhu, Mike Rabbat, Ashkan Yousefpour, Carole-Jean Wu, Hongyuan Zhan, Pavel Ustinov, Harish Srinivas, et al. 2022. Papaya: Practical, Private, and Scalable Federated Learning. In *MLSys*.

[25] Ahmed Imteaj and M Hadi Amini. 2020. Fedar: Activity and resource-aware federated learning model for distributed mobile robots. In *2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA)*.

[26] Zhipeng Jia and Emmett Witchel. 2021. Boki: Stateful Serverless Computing with Shared Logs. In *SOSP*.

[27] Zhifeng Jiang, Wei Wang, and Ruichuan Chen. 2022. Taming Client Dropout for Distributed Differential Privacy in Federated Learning. *arXiv:2209.12528* (2022).

[28] Zhifeng Jiang, Wei Wang, Bo Li, and Qiang Yang. 2022. Towards Efficient Synchronous Federated Training: A Survey on System Optimization Strategies. *IEEE Transactions on Big Data* (2022).

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

[30] Kaggle. 2021. Stack Overflow Data. https://www.kaggle.com/stackoverflow/stackoverflow.

[31] Peter Kairouz, Ziyu Liu, and Thomas Steinke. 2021. The distributed discrete gaussian mechanism for federated learning with secure aggregation. In *ICML*.

[32] Peter Kairouz, Brendan McMahan, Shuang Song, Om Thakkar, Abhradeep Thakurta, and Zheng Xu. 2021. Practical and private (deep) learning without sampling or shuffling. In *ICML*.

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

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

[35] Young Geun Kim and Carole-Jean Wu. 2021. Autofl: Enabling heterogeneity-aware energy efficient federated learning. In *MICRO*.

[36] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv:1412.6980* (2014).

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

[38] Fan Lai, Xiangfeng Zhu, Harsha V Madhyastha, and Mosharaf Chowdhury. 2021. Oort: Efficient Federated Learning via Guided Participant Selection. In *OSDI*.

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

[40] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. 1989. Backpropagation applied to handwritten zip code recognition. *Neural computation* 1, 4 (1989), 541–551.

[41] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. 2018. PRETZEL: Opening the black box of machine learning prediction serving systems. In *OSDI*.

[42] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *OSDI*.

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

[44] Wenqi Li, Fausto Milletarì, Daguang Xu, Nicola Rieke, Jonny Hancox, Wentao Zhu, Maximilian Baust, Yan Cheng, Sébastien Ourselin, M Jorge Cardoso, et al. 2019. Privacy-preserving federated brain tumour segmentation. In *MLMI Workshop*.

[45] Xiang Li, Kaixuan Huang, Wenhao Yang, Shusen Wang, and Zhihua Zhang. 2020. On the convergence of fedavg on non-iid data. In *ICLR*.

[46] Heiko Ludwig, Nathalie Baracaldo, Gegi Thomas, Yi Zhou, Ali Anwar, Shashank Rajamoni, Yuya Ong, Jayaram Radhakrishnan, Ashish Verma, Mathieu Sinn, et al. 2020. Ibm federated learning: an enterprise framework white paper v0. 1. *arXiv:2007.10987* (2020).

[47] Horia Mania, Xinghao Pan, Dimitris Papailiopoulos, Benjamin Recht, Kannan Ramchandran, and Michael I Jordan. 2017. Perturbed iterate analysis for asynchronous stochastic optimization. *SIAM Journal on Optimization* 27, 4 (2017), 2202–2229.

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

[49] MindSpore-AI. 2021. MindSpore. https://gitee.com/mindspore/mindspore.

[50] John Nguyen, Kshitiz Malik, Hongyuan Zhan, Ashkan Yousefpour, Michael Rabbat, Mani Malek Esmaeili, and Dzmitry Huba. 2022. Federated Learning with Buffered Asynchronous Aggregation. In *AISTATS*.

[51] Takayuki Nishio and Ryo Yonetani. 2019. Client selection for federated learning with heterogeneous resources in mobile edge. In *ICC*.

[52] NVIDIA. 2020. Triaging COVID-19 Patients: 20 Hospitals in 20 Days Build AI Model that Predicts Oxygen Needs. https://blogs.nvidia.com/blog/2020/10/05/federated-learning-covid-oxygen-needs/.

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

[54] Matthias Paulik, Matt Seigel, Henry Mason, Dominic Telaar, Joris Kluivers, Rogier van Dalen, Chi Wai Lau, Luke Carlson, Filip Granqvist, Chris Vandevelde, et al. 2021. Federated Evaluation and Tuning for On-Device Personalization: System Design & Applications. *arXiv:2102.08503* (2021).

[55] Swaroop Ramaswamy, Rajiv Mathews, Kanishka Rao, and Françoise Beaufays. 2019. Federated learning for emoji prediction in a mobile keyboard. *arXiv:1906.04329* (2019).

[56] Sashank J Reddi, Zachary Charles, Manzil Zaheer, Zachary Garrett, Keith Rush, Jakub Konečnỳ, Sanjiv Kumar, and Hugh Brendan McMahan. 2020. Adaptive Federated Optimization. In *ICLR*.

[57] Guomei Shi, Li Li, Jun Wang, Wenyan Chen, Kejiang Ye, and ChengZhong Xu. 2020. HySync: Hybrid Federated Learning with Effective Synchronization. In *HPCC/SmartCity/DSS*.

[58] Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V Le. 2018. Don't decay the learning rate, increase the batch size. In *ICLR*.

[59] Alexander Smola and Shravan Narayanamurthy. 2010. An architecture for parallel topic models. In *VLDB*.

[60] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research* 15, 1 (2014), 1929–1958.

[61] Sebastian U Stich. 2018. Local SGD converges fast and communicates little. *arXiv:1805.09767* (2018).

[62] Ningxin Su and Baochun Li. 2022. How Asynchronous can Federated Learning Be?. In *IWQoS*.

[63] Huangshi Tian, Minchen Yu, and Wei Wang. 2021. CrystalPerf: Learning to Characterize the Performance of Dataflow Computation through Code Analysis. In *ATC*.

[64] TL-System. 2021. Plato: A New Framework for Federated Learning Research. https://github.com/TL-System/plato.

[65] WeBank. 2020. Utilization of FATE in Anti Money Laundering Through Multiple Banks. https://www.fedai.org/cases/utilization-of-fate-in-anti-money-laundering-through-multiple-banks/.

[66] Ashia C Wilson, Rebecca Roelofs, Mitchell Stern, Nati Srebro, and Benjamin Recht. 2017. The marginal value of adaptive gradient methods in machine learning. In *NeurIPS*.

[67] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, et al. 2019. Machine learning at facebook: Understanding inference at the edge. In *HPCA*.

[68] Cong Xie, Sanmi Koyejo, and Indranil Gupta. 2020. Asynchronous federated optimization. In *OPT*.

[69] Chengxu Yang, QiPeng Wang, Mengwei Xu, Zhenpeng Chen, Kaigui Bian, Yunxin Liu, and Xuanzhe Liu. 2021. Characterizing Impacts of Heterogeneity in Federated Learning upon Large-Scale Smartphone Data. In *WWW*.

[70] Timothy Yang, Galen Andrew, Hubert Eichner, Haicheng Sun, Wei Li, Nicholas Kong, Daniel Ramage, and Françoise Beaufays. 2018. Applied federated learning: Improving google keyboard query suggestions. *arXiv:1812.02903* (2018).

[71] Hao Yu, Sen Yang, and Shenghuo Zhu. 2019. Parallel restarted SGD with faster convergence and less communication: Demystifying why model averaging works for deep learning. In *AAAI*.