2025 34th International Conference on Computer Communications and Networks (ICCCN) | 979-8-3315-0898-2/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICCCN65249.2025.11133903

PEGASUS: Efficient Asynchronous Three-Layer Federated Learning

Chen Ying
Department of Electrical and Computer Engineering
University of Toronto
10 King's College Road, Toronto, ON, M5S 3G4, Canada
Email: cying@ece.utoronto.ca

Baochun Li
Department of Electrical and Computer Engineering
University of Toronto
10 King's College Road, Toronto, ON, M5S 3G4, Canada
Email: bli@ece.toronto.edu

***Abstract*—Compared to conventional two-layer federated learning (FL), three-layer FL, which adds a layer of edge servers between the central server and clients, is less studied but could provide better training performance in terms of reducing elapsed wall-clock time to complete training. However, three-layer FL inherits and magnifies some inherent challenges in two-layer FL, such as client heterogeneity. With different computing capabilities and network connections, slow clients require exceedingly long training times. To alleviate the negative effect due to slow clients, this paper is the first to study asynchronous three-layer FL, where edge servers conduct local aggregation without waiting for slow clients and the central server aggregates updates from fast edge servers. Unfortunately, asynchronous three-layer FL could suffer from performance degradation as when aggregating updates from slow clients and edge servers, those updates are not computed based on the newest global model. Therefore, we propose a staleness-aware framework, PEGASUS, with newly designed client selection, compression, and update aggregation mechanisms to improve every important aspect during training. Our extensive evaluation of different training tasks demonstrates that PEGASUS can achieve a reduction in elapsed wall-clock training time by at least 46.8% with an increase in converged accuracy of the trained global model by up to 0.3% as compared to the state-of-the-art.**

***Index Terms*—Asynchronous Federated Learning, Three-Layer Federated Learning, Staleness-Aware Optimization**

I. INTRODUCTION

Conventional federated learning (FL) [1] employs a two-layer structure with a central server and a layer of numerous clients to train a global model while keeping private local data on clients. In each communication round, the central server sends the current global model to clients, who train locally and only return their local updates for aggregating a new global model. Despite preserving privacy, two-layer FL suffers from communication bottlenecks as clients share the limited bandwidth to communicate with the central server.

Hence, three-layer FL, which is shown in Fig. 1, has been proposed to add a layer of edge servers between the central server and clients. With this hierarchy, the central server only needs to communicate with edge servers and each client only communicates with its edge server, leading to faster convergence in theory [2]–[4] and in practice [5]. This structure, also known as *hierarchical FL* [2], [4], [6]–[8], improves communication efficiency and training speed.

However, three-layer FL inherits and can amplify challenges of client heterogeneity in two-layer FL. In reality, clients often have non-i.i.d. (not independent and identically distributed) local datasets, which could significantly reduce the accuracy and slow down the convergence of the global model [9], [10]. Their computing capabilities also vary widely, with slower clients — known as *stragglers* [11], [12] in conventional distributed machine learning — delaying training, as edge servers must wait for their updates. This delay propagates to the central server, further slowing down global aggregation.

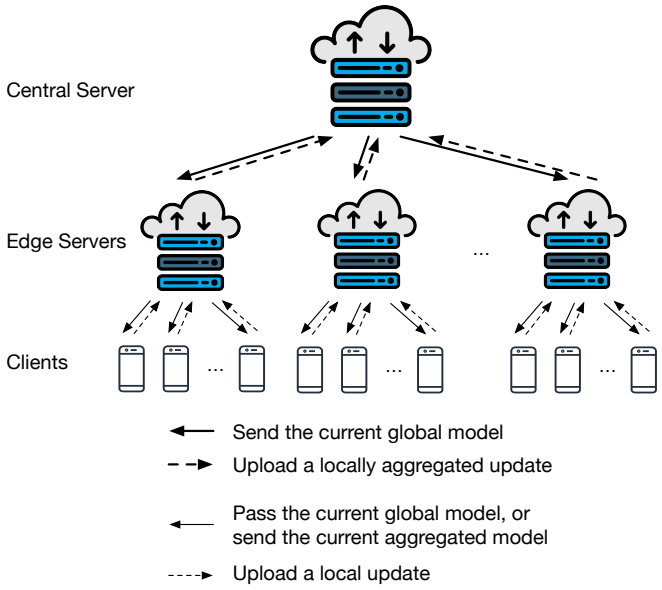


Fig. 1: Three-layer FL.

A key performance metric in FL is the *elapsed wall-clock training time* for the global model to converge to a target accuracy. To reduce this by alleviating the performance degradation due to stragglers, some works of two-layer FL have proposed to proceed FL server *asynchronously* with only a subset of timely client updates [13]–[16]. It has been shown that asynchronous two-layer FL can outperform synchronous two-layer FL with heterogeneous clients [17]. Hence, it is intuitive to use asynchronous central server and edge servers in three-layer FL. Yet, as far as we know, asynchronous three-layer FL remains unexplored.

Unfortunately, similar to asynchronous two-layer FL, asynchronous three-layer FL also suffers from degraded perfor-

mance due to *staleness* of updates [17]. Since the server may have progressed several rounds by aggregating updates from faster clients, the global model used by a slower client for local training can become *stale*. In three-layer FL, this issue worsens as edge servers perform multiple rounds of local aggregation before reporting to the central server. The more rounds an edge server with slow clients processes, the staler its aggregated update becomes. To address staleness at both client and edge levels, customized mechanisms are needed to improve training performance in asynchronous three-layer FL.

In this paper, we propose PEGASUS, a staleness-aware framework for asynchronous three-layer FL that reduces the elapsed wall-clock time required to reach target accuracy. PEGASUS introduces new client selection, pruning, quantization, and update aggregation methods, tailored for asynchronous three-layer FL to enhance all key aspects of training. For each local aggregation, edge servers record selection scores for clients, calculated using staleness and the quality of their latest updates. During aggregation, client updates receive weights based on staleness and divergence from the locally aggregated model. Similarly, when the central server aggregates, each edge server's update is weighted by its staleness and divergence from the current global model.

A key challenge in pruning updates from clients and edge servers is avoiding performance degradation from removing important parameters. However, pruning can do more than reduce update size: it can improve performance by eliminating parameters that would otherwise lower global model accuracy. PEGASUS leverages both pruning and quantization to compress updates while minimizing or even reversing accuracy loss. Designed for asynchronous three-layer FL, PEGASUS applies more aggressive compression to staler clients and edge servers, whose updates are typically less valuable to the global model.

Original contributions. Highlights of our original contributions in this paper are as follows. *First*, to our best knowledge, we are the first to study asynchronous three-layer FL. *Second*, we propose a new framework PEGASUS, to reduce the elapsed wall-clock training time with carefully designed client selection, pruning, quantization, and update aggregation mechanisms. Our mechanisms are *staleness-aware*, in that updates with higher degrees of staleness are pruned more aggressively and have smaller aggregation weights. Also, these clients are less likely to be selected in future rounds. *Finally*, with a scalable implementation and reproducible experiments, our evaluation demonstrated convincing evidence that PEGASUS consistently outperformed its state-of-the-art alternatives with respect to the wall-clock training time used for converging to a target accuracy over a variety of tasks. Towards even better reproducibility, our work will be released as open-source to the research community and open for cross-examinations.

II. PRELIMINARIES

A. Three-Layer Federated Learning

To formulate the three-layer FL model in Fig. 1, assume the bottom layer has N clients and the second layer has M edge servers. Each client communicates with one edge server, and the central server only communicates with edge servers. Suppose each client n, $n \in [N]$, has a local dataset $\mathcal{D}_n$. If client n connects to edge server m, $m \in [M]$, and is selected in the current round, then $n \in \mathcal{C}^{(m)}$, with $C^{(m)} = |\mathcal{C}^{(m)}|$. The dataset of edge server m's selected clients is $\mathcal{D}^{(m)} = \cup_{n\in\mathcal{C}^{(m)}}\mathcal{D}_n$, and the union of all data is $\mathcal{D} = \cup_{m=1}^{M}\mathcal{D}^{(m)}$.

We use the *FedAvg* algorithm [1], the most widely used algorithm for two-layer FL, which combines local stochastic gradient descent (SGD) on clients with update aggregation at a central server. For the three-layer setup, we adapt it as follows. In each global communication round t, the central server sends model $\mathbf{w}(t)$ to edge servers, which forward it to their clients. In local round t_e, client n at edge server m performs τ_c epochs of SGD on $\mathbf{w}^{(m)}(t,t_e)$ to produce $\mathbf{w}_n(t,t_e,\tau_c)$, and sends update $\Delta_n(t,t_e) = \mathbf{w}_n(t,t_e,\tau_c) - \mathbf{w}^{(m)}(t,t_e)$.

Edge server m aggregates updates from all $n \in \mathcal{C}^{(m)}$ into its aggregated model $\mathbf{w}^{(m)}(t,t_e+1)$:

$$\mathbf{w}^{(m)}(t,t_e+1) = \sum_{n\in\mathcal{C}^{(m)}} \frac{|\mathcal{D}_n|}{|\mathcal{D}^{(m)}|}\Delta_n(t,t_e) + \mathbf{w}^{(m)}(t,t_e). \quad (1)$$

For $t_e = 1, 2, \ldots, \tau_e - 1$, edge server m sends its aggregated weights $\mathbf{w}^{(m)}(t,t_e+1)$ to newly selected clients. After τ_e rounds of local aggregation, update $\Delta^{(m)}(t) = \mathbf{w}^{(m)}(t,\tau_e+1) - \mathbf{w}(t)$ is reported to the central server. After receiving aggregated updates of all the edge servers, the central server updates the global model $\mathbf{w}(t+1)$ by aggregating them:

$$\mathbf{w}(t+1) = \sum_{m=1}^{M} \frac{|\mathcal{D}^{(m)}|}{|\mathcal{D}|}\Delta^{(m)}(t) + \mathbf{w}(t). \quad (2)$$

B. Asynchronous Three-Layer FL

A limitation of the above *FedAvg*-based three-layer FL is that the central server (edge server m) aggregates only after receiving updates from all M edge servers (or all $C^{(m)}$ selected clients). Yet some clients may take longer to finish local training, delaying the entire process.

To deal with slow clients, we propose to let the central server and edge servers aggregate asynchronously. The central server updates after receiving V ($V < M$) edge server updates; each edge server aggregates after $K^{(m)}$ ($K^{(m)} < C^{(m)}$) client updates. However, as shown in prior work [15], [17], staleness in asynchronous FL can degrade performance. In the t-th global communication round, an update received may be based on an outdated global model different from $\mathbf{w}(t)$. Such a phenomenon is referred to as *client staleness*. Unfortunately, in asynchronous three-layer FL, staleness could exist on not only clients but also edge servers, and thus global model performance would have higher chances to be degraded. We define the *edge server staleness* and *client staleness* as follows:

Definition 1 (Edge server staleness)**.** *In global communication round t at the central server, edge server staleness $s^{(m)}(t)$ is the number of global communication rounds that has elapsed since the last time edge server m received the global model from the central server. If edge server m received the global model $\mathbf{w}(\kappa^{(m)})$ and passes it to its clients for local training, the edge server staleness $s^{(m)}(t) := t - \kappa^{(m)}$.*

Definition 2 (Client staleness)**.** *In global communication round t at the central server and local aggregation round t_e at edge server m, client staleness $s_n(t, t_e)$, $n \in \mathcal{C}^{(m)}$, is the number of local aggregations that have processed on edge server m since the last time client n received an aggregated model from edge server m. If client n received the aggregated model $\mathbf{w}^{(m)}(\kappa^{(m)}, \kappa_n)$ and uses it to conduct local training, the client staleness $s_n(t, t_e) := (t - \kappa^{(m)})\tau_e + t_e - \kappa_n$.*

III. Related Work

Most existing asynchronous FL mechanisms [13], [16]–[18] are designed for two-layer FL. The first proposed one is *FedAsync* [13], which lets the central server to aggregate whenever it receives one client update. However, stale updates from slower clients degrades the training performance and could even result in divergence [17]. To mitigate the issue of staleness, *FedBuff* [16] aggregates with more than one updates.

To improve training performance in FL, techniques such as client selection and pruning have been widely studied. *Oort* [19] selects clients based on statistical utility and system utility, measured by their most recent training loss and training time. *Hermes* [20] prunes channels with the lowest magnitudes in each local model. It adjusts the pruning amount based on each local model's test accuracy and its previous pruning amount. *Sub-FedAvg* [21] iteratively prunes the parameters and channels of each client's local model based on its test accuracy during its local training. However, all the above-mentioned mechanisms were designed for synchronous two-layer FL. Staleness, a major factor in global model performance in asynchronous FL, was not considered. Also, the training scheme of three-layer FL is different from two-layer FL. Thus, directly using them in asynchronous three-layer FL leads to degraded performance degradation, highlighting the need for mechanisms tailored to asynchronous three-layer FL.

IV. Overview

To reduce the elapsed wall-clock training time by enhancing all critical components of asynchronous three-layer FL training, we propose a new staleness-aware framework, PEGASUS.

Fig. 2 shows the overview of PEGASUS, which includes a *global update aggregator* on the central server, a *client selector*, *local update aggregator*, and a *compressor* on each edge server, and a *compressor* on each client. The *global update aggregator* is designed based on edge server staleness, while the edge server components and client-side *compressor* are designed based on client staleness.

Ideally, staleness should be measured using Definition 1 and Definition 2. Unfortunately, such values are not always available when needed. For example, an edge server cannot predict its staleness at the moment of compression, as it does not know the communication round in which its update will be aggregated at the central server. By the similar token, a client cannot know its staleness when compressing its local update. Hence, we need observable measures to substitute for edge server staleness and client staleness.

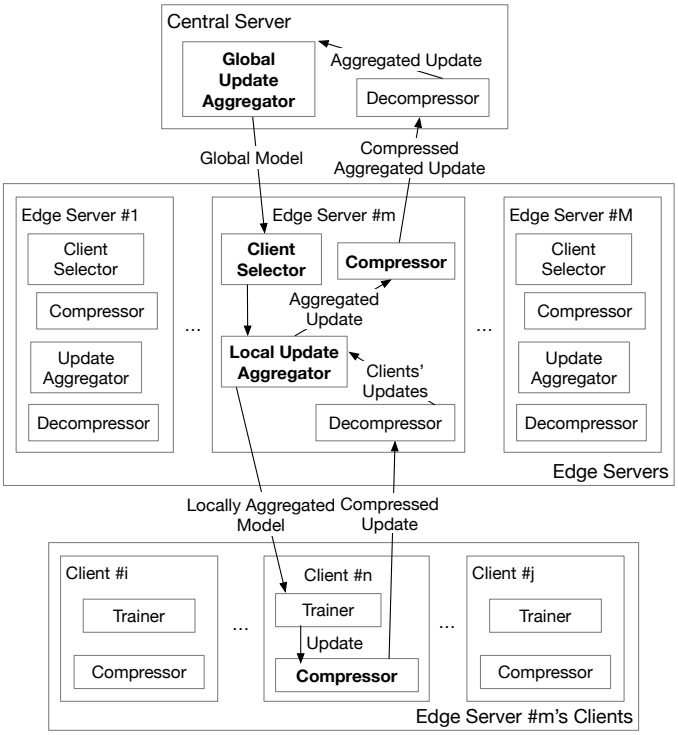


Fig. 2: PEGASUS: a design overview.

A substitute for client staleness. Longer local training typically leads to staler updates, as more aggregation rounds occur during training. Therefore, we use a client's local training time to substitute for its staleness. Edge server m, $m \in [M]$, records $\gamma_i^{(m)}$, the average local training time of i reported clients. After it receives the $(i+1)$-th update from its client n, it updates $\gamma_{i+1}^{(m)}$ as:

$$\gamma_{i+1}^{(m)} = \frac{\gamma_i^{(m)} \times i + \frac{l_n}{|\mathcal{D}_n|}}{i+1}, \tag{3}$$

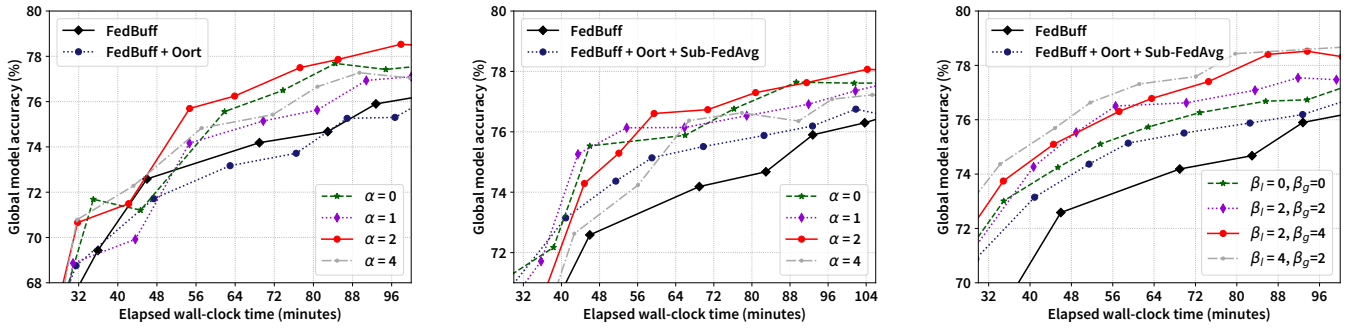
where l_n is client n's local training time, $|\mathcal{D}_n|$ is its dataset size, and $\gamma_0^{(m)} = 0$, $\forall m \in [M]$. As local dataset sizes usually vary among clients, a client with a larger local dataset should need a longer local training time than one with the same computing capability but a smaller local dataset. Thus, instead of l_n, $l_n/|\mathcal{D}_n|$ is used in Eq. (3) to update $\gamma^{(m)}$ for fairness.

A substitute for edge server staleness. Similarly, edge server staleness is substituted with the central server recording γ_j, the average training time of j reported edge servers. After receiving the $(j+1)$-th locally aggregated update from edge server m, the central server updates γ_{j+1} as:

$$\gamma_{j+1} = \frac{\gamma_j \times j + \gamma^{(m)}}{j+1}, \tag{4}$$

where $\gamma^{(m)}$ is the substitute for client staleness of edge server m that the central server received with edge server m's update.

The workflow. In the first global communication round, the central server sends all the M edge servers a randomly initialized global model and $\gamma_0 = 0$. Each edge server m then randomly selects $C^{(m)}$ clients and sends them the current global model and $\gamma_0^{(m)} = 0$.



(a) Comparing different client selectors. (b) Comparing different pruning mechanisms. (c) Comparing different update aggregators.

Fig. 3: Designing PEGASUS: motivational examples over the EMNIST dataset.

The selected client n of edge server m conducts local training based on received $\mathbf{w}^{(m)}$ to generate its update, which will be compressed based on client n's staleness estimated with its received $\gamma^{(m)}$. Client n then sends the compressed update, local training time l_n, and $|\mathcal{D}_n|$ to its edge server m.

After collecting $K^{(m)}(K^{(m)} < C^{(m)})$ updates, edge server m aggregates them and updates $\gamma^{(m)}$. Before finishing τ_e rounds of local aggregation, edge server m's client selector will select $K^{(m)}$ clients and send them its latest locally aggregated model, $\mathbf{w}^{(m)}$, and $\gamma^{(m)}$. After τ_e rounds, edge server m sends its update $\Delta^{(m)}$ and $\gamma^{(m)}$ to the central server.

On the central server, once it receives $V(V < M)$ reports, its global update aggregator assigns different weights to these locally aggregated updates based on their staleness and uses weighted averaging to generate a new global model. The central server then updates γ with Eq. (4) and starts a new communication round by sending the current global model and updated γ to the V edge servers that just reported.

V. MOTIVATION AND DESIGN

This section presents empirical studies that motivate our design of the client selector, compressor, local update aggregator, and global aggregator in PEGASUS. All experiments use the same setup: $M = 8$ edge servers and $N = 1000$ clients. Each edge server selects $C^{(m)} = 15$ clients from its 125 and aggregates updates from $K^{(m)} = 10$ per round. After $\tau_e = 3$ rounds of local aggregation, edge servers send updates to the central server, which aggregates once it receives $V = 6$ edge server updates. We use the `LeNet-5` model and `EMNIST` dataset. Clients train with batch size 32, 5 epochs, learning rate 0.01, momentum 0.9, and no weight decay.

A. Client Selector

Due to their different local data distributions and computing capabilities, some clients can quickly report their updates of high quality, i.e., updates that would increase the global model accuracy. We thus design a client selector on edge servers to give those clients higher chances of being selected.

Each edge server maintains a client selector that tracks a *selection score* for each client, with higher scores indicating higher selection probabilities. All scores are initialized as $K^{(m)}$, the number of updates required for local aggregation. Denote the set of clients whose updates are aggregated in local aggregation round t_e of global communication round t as $\mathcal{A}^{(m)}(t, t_e)$. After this local aggregation, the client selector updates the selection score of client n, $\forall n \in \mathcal{A}^{(m)}(t, t_e)$ as:

$$\text{Score}_n(t, t_e) = \underbrace{\frac{|\mathcal{D}^{(m)}(t, t_e)|}{|\mathcal{D}_n|}\Phi}_{\text{Update quality}} \times \underbrace{\text{sigmoid}\left(\frac{\gamma^{(m)}}{\frac{l_n}{|\mathcal{D}_n|}}\right)^{\alpha}}_{\text{Local training time}} \quad (5)$$

where $|\mathcal{D}^{(m)}(t, t_e)| = \sum_{n \in \mathcal{A}^{(m)}(t,t_e)} |\mathcal{D}_n|$ is the total number of data samples of reported clients in $\mathcal{A}^{(m)}(t, t_e)$. $\Phi = \frac{\Theta(\Delta_n(t,t_e), \mathbf{w}^{(m)}(t,t_e+1) - \mathbf{w}^{(m)}(t,t_e)) + 1}{2}$, where Θ is cosine similarity. $\Delta_n(t, t_e)$ is the local update of client n. $\gamma^{(m)}$ is the current average local training time.

The first component of Eq. (5) represents the quality of client update $\Delta_n(t, t_e)$. Since aggregating a client update that has the same angle as $\mathbf{w}^{(m)}(t, t_e + 1) - \mathbf{w}^{(m)}(t, t_e)$ would contribute the most to improving the newly aggregated local model $\mathbf{w}^{(m)}(t, t_e+1)$, we use the cosine similarity Θ to quantify the quality of $\Delta_n(t, t_e)$. With our design, a client update with a smaller angle between $\mathbf{w}^{(m)}(t, t_e + 1) - \mathbf{w}^{(m)}(t, t_e)$ would have a higher value of this component and thus have a higher selection score.

The second component captures local training time, a substitute for client staleness. We use the sigmoid function, which is monotonically increasing, to assign higher selection scores to clients with shorter local training times. A hyperparameter α is introduced as a tuning knob to control how much the local training time should contribute to the client selection score.

In the next local aggregation round t_e+1, client n is selected with probability $\psi_n(t, t_e + 1) = \frac{\text{Score}_n(t,t_e)}{\sum_{i \in \mathcal{S}^{(m)}(t,t_e+1)} \text{Score}_i(t,t_e)}$, where $\mathcal{S}^{(m)}(t, t_e + 1)$ are the set of currently available clients. For any client $i \in \mathcal{S}^{(m)}(t, t_e)$ and $i \notin \mathcal{A}^{(m)}(t, t_e)$, $\text{Score}_i(t, t_e) = \text{Score}_i(t, t_e - 1)$.

Fig. 3a illustrates our evaluation on our client selector with $\alpha = 0, 1, 2, 4$, compared to random selection, which is labeled as *FedBuff* in the figure, and *Oort* [19], a state-of-the-art client selection mechanism designed for synchronous two-layer FL. Here we apply the selection strategy of *Oort* on edge servers for them to select clients. Its performance is labeled as *FedBuff + Oort* in Fig. 3a. Our method consistently outperforms all

TABLE I: Elapsed wall-clock training times to reach different target accuracies with different client selection mechanisms.

Target Accuracy	Elapsed Wall-Clock Training Time (Hours)					
	FedBuff	*Oort*	$\alpha = 0$	$\alpha = 1$	$\alpha = 2$	$\alpha = 4$
76%	1.73	1.78 (-2.8%)	1.23 (28.9%)	1.51 (12.4%)	**1.06 (38.4%)**	1.35 (22.0%)
77%	2.51	2.49 (0.7%)	1.41 (43.9%)	1.67 (33.6%)	**1.29 (48.7%)**	1.49 (40.7%)
78%	2.86	4.22 (-47.7%)	2.03 (29.1%)	2.25 (21.4%)	**1.63 (43.0%)**	2.06 (28.0%)
79%	6.27	7.42 (-18.3%)	3.82 (39.0%)	4.71 (24.9%)	**3.40 (45.7%)**	3.46 (44.9%)

alternatives. *Oort*, however, at some training points had lower accuracies than using random selection.

Table I lists elapsed wall-clock training times to reach different target accuracies. Numbers in brackets are the percentages of reduced training time compared with *FedBuff*. We can see that *Oort* used a longer amount of time than random selection to reach 76%, 78%, and 79%. *Oort*'s unsatisfactory performance is due to its design for synchronous FL. It compares client n's local training time l_n with a predefined Γ. For any client n with $l_n > \Gamma$, *Oort* decreases their probability of being selected with the same amount. There are two major problems in this design. First, Γ is hard to decide before training while it largely affects *Oort*'s performance. Second, it is unreasonable to decrease slow clients' probability of being selected with the same amount. As clients could be highly heterogeneous in FL, some may only take training times that are slightly longer than Γ, while others may much longer. Our mechanism avoids using any predefined values and penalizes slow clients based on their local training time, thus leading to better performance than *Oort*.

Another notable observation in the results is that among the four values, $\alpha = 2$ yielded the shortest training time to reach the target accuracy. It shows that besides staleness, update quality is also an important factor in client selection. For the best performance, we need to balance these two factors.

B. Compressor: Pruning

To decrease training time, PEGASUS prunes both local updates of clients and aggregated updates of edge servers to reduce transmission overhead, which often dominates training time. We use L1-norm pruning [22], removing parameters with the smallest L1-norm. The intuition is that those parameters usually have less significant effects on the output, and hence are less likely to have a material impact on model performance if they are pruned.

Pruning is adaptive to staleness. The pruning amount of client n is formulated as:

$$\rho_n = 1 - \text{sigmoid}\left(\frac{\gamma^{(m)}}{\frac{l_n}{|\mathcal{D}_n|}}\right), \tag{6}$$

where $\gamma^{(m)}$ was sent along with the locally aggregated model from client n's edge server m.

After local training, client n prunes its update before sending it out. It zeros out ρ_n of parameters with the smallest L1-norm. As a client would not know its staleness when pruning, its pruning amount is computed based on its local training time. Our formulation of Eq. (6) follows the rule that a client with longer local training time should prune more, and the reason is two-fold. One is that if a client's local training time is long, its staleness will be high when its edge server receives its local update. Hence, the quality of this update is more likely to be low. The other is that as the client already took a long time for local training, it should compress its update more so that its edge server can receive it more quickly.

Each edge server also prunes its locally aggregated update before sending it to the central server. The pruning amount of edge server m is formulated as:

$$\rho^{(m)} = \text{sigmoid}\left(\frac{\gamma^{(m)} - \gamma}{\gamma}\right), \tag{7}$$

where γ is the average training time that edge server m received along with the global model from the central server.

Since an edge server m would not know its staleness when pruning, we use its $\gamma^{(m)}$ to determine its pruning amount. Basically, we assign a higher pruning amount to an edge server with a larger $\gamma^{(m)}$, because larger $\gamma^{(m)}$ implies more time spent on local aggregation and greater staleness. Thus, the update is likely of lower quality and should be pruned more aggressively for faster transmission.

To test the efficiency of our pruning mechanism with our client selection mechanism, we compare the combination of them with *FedBuff*, which uses random client selection and no pruning, as the baseline, and with *FedBuff* + *Oort* + *Sub-FedAvg*. Designed for synchronous two-layer FL, *Sub-FedAvg* [21] iteratively prunes the parameters and channels of each client's local model based on its test accuracy during its local training. We use the pruning strategy of *Sub-FedAvg* on both clients and edge servers for a fair comparison.

Fig. 3b highlights the advantage of our pruning mechanism combined with our selector, outperforming *Sub-FedAvg* across all tested α values.

C. Update Aggregators

In PEGASUS, two types of aggregators are used: local update aggregators at edge servers, and a global update aggregator at the central server.

Local update aggregator. In asynchronous three-layer FL, client updates vary in staleness and quality. For better training performance, each edge server should assign larger aggregation weights to updates with higher quality and lower staleness. Thus, we adopt the following Eq. (8) to compute

TABLE II: Elapsed wall-clock training times to reach different target accuracies with different β_l and β_g.

Target Accuracy	Elapsed Wall-Clock Training Time (Hours)				
	FedBuff	$\beta_l = 0, \beta_g = 0$	$\beta_l = 2, \beta_g = 2$	$\beta_l = 2, \beta_g = 4$	$\beta_l = 4, \beta_g = 2$
76%	1.73	1.21 (29.8%)	0.94 (45.5%)	0.95 (44.8%)	**0.86 (50.2%)**
77%	2.51	1.71 (32.0%)	1.39 (44.6%)	1.24 (50.6%)	**1.02 (59.4%)**
78%	2.86	2.27 (20.7%)	2.06 (28.0%)	1.43 (49.9%)	**1.33 (53.6%)**
79%	6.27	3.74 (40.3%)	3.40 (45.8%)	**1.81 (71.1%)**	2.30 (63.4%)

the aggregation weight of client n in local aggregation round t_e of communication round t:

$$p_n(t,t_e) = \underbrace{\frac{|\mathcal{D}_n|}{|\mathcal{D}^{(m)}(t,t_e)|}\Psi}_{\text{Estimation of update quality}} \times \underbrace{\Big(1 - \frac{s_n(t,t_e)}{s^{(m)}_{\max}(t,t_e)+1}\Big)^{\beta_l}}_{\text{Staleness}}, \tag{8}$$

where $\Psi = \frac{\Theta(\Delta_n(t,t_e), \mathbf{w}^{(m)}(t,t_e) - \mathbf{w}^{(m)}(t,t_e-1)) + 1}{2}$ and $s^{(m)}_{\max}(t,t_e)$ is the highest staleness of updates to be aggregated.

There are two components in Eq. (8). The first estimates the quality of the update. Since the new locally aggregated model $\mathbf{w}^{(m)}(t,t_e+1)$ is not yet available, the aggregator cannot compute $\Theta(\Delta_n(t,t_e), \mathbf{w}^{(m)}(t,t_e+1) - \mathbf{w}^{(m)}(t,t_e))$. Instead, it uses $\Theta(\Delta_n(t,t_e), \mathbf{w}^{(m)}(t,t_e) - \mathbf{w}^{(m)}(t,t_e-1))$ to estimate the update quality. The second component accounts for staleness, assigning higher aggregation weights to fresher updates. The hyperparameter β_l controls how strongly staleness influences the weights.

After normalizing all $p_n(t,t_e)$, $\forall n \in \mathcal{A}^{(m)}(t,t_e)$, so they sum to 1, the local update aggregator on edge server m computes a locally aggregated model as:

$$\mathbf{w}^{(m)}(t,t_e+1) = \mathbf{w}^{(m)}(t,t_e) + \sum_{n \in \mathcal{A}^{(m)}(t,t_e)} p_n(t,t_e)\Delta_n(t,t_e). \tag{9}$$

Global update aggregator. To generate an improved global model, the global update aggregator on the central server conducts weighted averaging by assigning a different weight to each aggregated update based on its staleness. Intuitively, aggregated updates with higher quality and lower staleness should have larger aggregation weights. Therefore, we formulate the following Eq. (10) to compute the aggregation weight of reported edge server m in communication round t:

$$p^{(m)}(t) = \underbrace{\frac{\Theta(\Delta^{(m)}(t), \mathbf{w}(t) - \mathbf{w}(t-1)) + 1}{2}}_{\text{Estimated quality of locally aggregated update}} \times \underbrace{\Big(1 - \frac{s^{(m)}(t)}{s_{\max}(t)+1}\Big)^{\beta_g}}_{\text{Staleness}}, \tag{10}$$

where $s_{\max}(t)$ is the highest staleness of locally aggregated updates to be aggregated.

Same as the client aggregation weight in Eq. (8), edge server aggregation weight also has two components. The first one is for update quality and the second one is for staleness. Similarly, we use $\Theta(\Delta^{(m)}(t), \mathbf{w}(t) - \mathbf{w}(t-1))$ to estimate the update quality. As the new global model $\mathbf{w}(t+1)$ has not been aggregated at this point, the aggregator cannot use $\Theta(\Delta^{(m)}(t), \mathbf{w}(t+1) - \mathbf{w}(t))$ to quantify the quality of the aggregated update $\Delta^{(m)}(t)$.

To increase aggregation weights for edge servers with lower staleness and decrease for those with higher staleness, we use the second component in Eq. (10). Similar to the hyperparameter β_l for local update aggregator, here we have hyperparameter β_g to control how significantly staleness should affect the aggregation weights of locally aggregated updates.

Denote $\mathcal{A}(t)$ as the set of edge servers whose updates are about to be aggregated. After normalizing all $p^{(m)}(t)$, $\forall m \in \mathcal{A}(t)$, such that their sum becomes 1, the global update aggregator aggregates an improved global model as:

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \sum_{m \in \mathcal{A}(t)} p^{(m)}(t)\Delta^{(m)}(t). \tag{11}$$

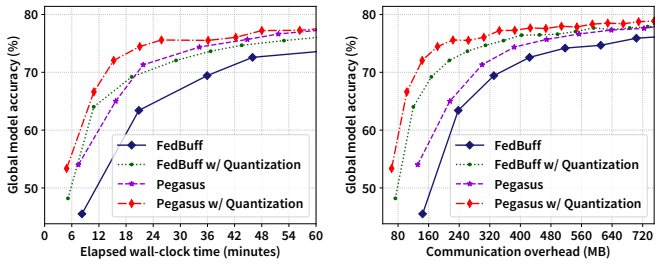
Fig. 3c shows the performance of the global model trained with our global update aggregator, local update aggregator, client selector, and pruning mechanism. As $\alpha = 2$ had the best performance in the previous two examples, we set $\alpha = 2$ and evaluated among differences values of β_l and β_g. It is obvious that adding the local and global update aggregators further improved the training performance of PEGASUS, indicating that our update aggregation mechanism is better than assigning weights only based on sizes of local datasets, as Eq. (1) and Eq. (2), in conventional FL.

Table II records elapsed wall-clock training times to reach target accuracies. Time reductions compared to *FedBuff* are shown in brackets. PEGASUS consistently accelerated training. With $\beta_l = 2$ and $\beta_g = 4$, it reduced time by 71.1% to reach 79% accuracy. With $\beta_l = 4$, $\beta_g = 2$, reductions ranged from 50.2% to 63.4%.

D. Additional Compressor: Quantization

To further boost PEGASUS's performance, we incorporate quantization as an additional compression method. Our empirical study shows that converting the global model, locally aggregated models, and both edge and client updates from 32-bit to 16-bit floating-point significantly reduces elapsed wall-clock training time without degrading the performance of the trained global model.

Fig. 4 demonstrates these performance improvements in both *FedBuff* and PEGASUS. Fig. 4a shows that quantization largely reduced the wall-clock time for both mechanisms. Regarding the communication overhead (accumulated sizes of transferred models and updates), the advantage of quantization became more noticeable as Fig. 4b illustrates.



(a) (b)

Fig. 4: Quantization can help reduce elapsed wall-clock training time and communication overhead.

Thus, PEGASUS combines quantization with pruning. The central server sends a quantized global model to edge servers, who forward it to their selected clients. Clients dequantize the model to 32-bit format before local training. After training, they prune and quantize their updates before sending them back. Upon receiving $K^{(m)}$ updates, edge server m dequantizes and aggregates them. In local aggregation round $t_e < \tau_e$, the edge server quantizes its aggregated model and sends the quantized model to its selected clients for another round of local aggregation. In the τ_e-th local aggregation round, the edge server prunes and quantizes its aggregated update before reporting to the central server. After receiving V quantized updates from edge servers, the central server dequantizes and aggregates them to generate a new global model.

VI. IMPLEMENTATION AND EVALUATION

A. Implementation and Preparation

We have implemented three-layer FL and PEGASUS in the open-source FL research framework, PLATO[1], which supports scalable, reproducible, and extensible FL research. While PLATO already supports conventional two-layer FL, extending it to three-layer FL required implementing edge servers, which is the most challenging part.

An edge server functions as both a server and a client: it receives the global model and reports aggregated updates to the central server (client role), while selecting clients, distributing models, and aggregating updates locally (server role). To enable this dual functionality, each edge server runs two asynchronous tasks, one as a client and one as a server, using the asynchronous event-driven programming paradigm, which is only supported since Python 3.4.

To make sure a training session can proceed correctly, there are two special timings when one of the two tasks of an edge server needs to signal the other. At the beginning of a global communication round, when an edge server is selected, its client task signals its server task to start a new round of local aggregation by selecting clients. After finishing τ_e rounds of local aggregation, the server task signals the client task to report to the central server. Using synchronization primitives *Event* from the Python library to notify tasks that some events have happened, we implement two *Event* objects, one for starting a new global communication round, and the other for completing τ_e rounds of local aggregation.

An *Event* object manages an internal flag that is set to false initially. When the event happens, the `set()` method set the flag to true. The `clear()` method resets the flag to false, and the `wait()` method blocks until the flag is set to true. For example, to correctly start an edge server's first round of local aggregation, the server task of this edge server calls `wait()` to block before selecting clients, so that it can wait for the client task to call `set()` right after the edge server is selected by the central server. The server task then uses `clear()` to set the flag to false for the next global communication round.

To launch a three-layer FL training session, the central server starts as a WebSocket server, then initiates all edge servers as WebSocket clients. Once all edge servers are running, the central server initializes clients. Each edge server also starts a WebSocket server to communicate with its clients.

Experimental settings. We evaluate PEGASUS on six FL training tasks: training `LeNet-5` with `MNIST`, `FashionMNIST`, `EMNIST`, and `FEMNIST`; `ResNet-18` with `CIFAR-10`; and `VGG-16` with `CINIC-10`.

There are $M = 8$ edge servers, each with an equal number of clients. Every edge server performs $\tau_e = 3$ local aggregation rounds before sending updates to the central server, which aggregates after receiving updates from $V = 6$ edge servers. Except for the `FEMNIST` dataset which naturally provides 3597 non-i.i.d. local datasets, the data distribution of local datasets is sampled with the symmetric Dirichlet distribution with a concentration of 5. For local training of clients, we use a batch size of 32, 5 local epochs, a learning rate of 0.01, and a momentum of 0.9 across all tasks. Additional key parameters are listed in Table III.

TABLE III: Parameter settings for evaluating PEGASUS.

Parameter	`MNIST` `FashionMNIST` `EMNIST`	`FEMNIST`	`CIFAR-10` `CINIC-10`
Total clients	1000	3597	1000
$C^{(m)}$	15	15	6
$K^{(m)}$	10	10	4
Weight decay	0	0	0.0001

Hyperparameter sweep. Before comparing PEGASUS with state-of-the-art mechanisms, we first perform a hyperparameter sweep for α in Eq. (5), β_l in Eq. (8), and β_g in Eq. (10). We simulate heavy-tailed local training times using the Pareto distribution and fix a random seed to ensure consistent training times across runs for fairness and reproducibility.

Using `LeNet-5` on the `EMNIST` dataset as an example, we tested 27 combinations with $\alpha, \beta_l, \beta_g = 0, 2, 4$. Fig. 5 compares representative combinations, showing that $\alpha = 2$, $\beta_l = 4$, and $\beta_g = 2$ consistently outperformed others across all tasks. We therefore adopt this setting for our performance evaluations.

B. Performance Evaluation

Evaluating PEGASUS without quantization. We first evaluate PEGASUS with its basic components: client selector,

[1]`https://github.com/TL-System/plato`

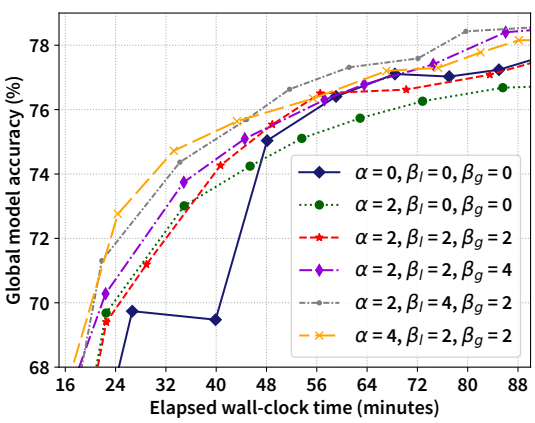


Fig. 5: Evaluating different combinations of hyperparameters α, β_l, and β_g.

adaptive pruning, local update aggregator, and global update aggregator, without quantization. Fig. 6 presents results, where the label *FedBuff* + represents *FedBuff* + *Oort* + *Sub-FedAvg*. *FedAvg* is for synchronous three-layer FL as another baseline.

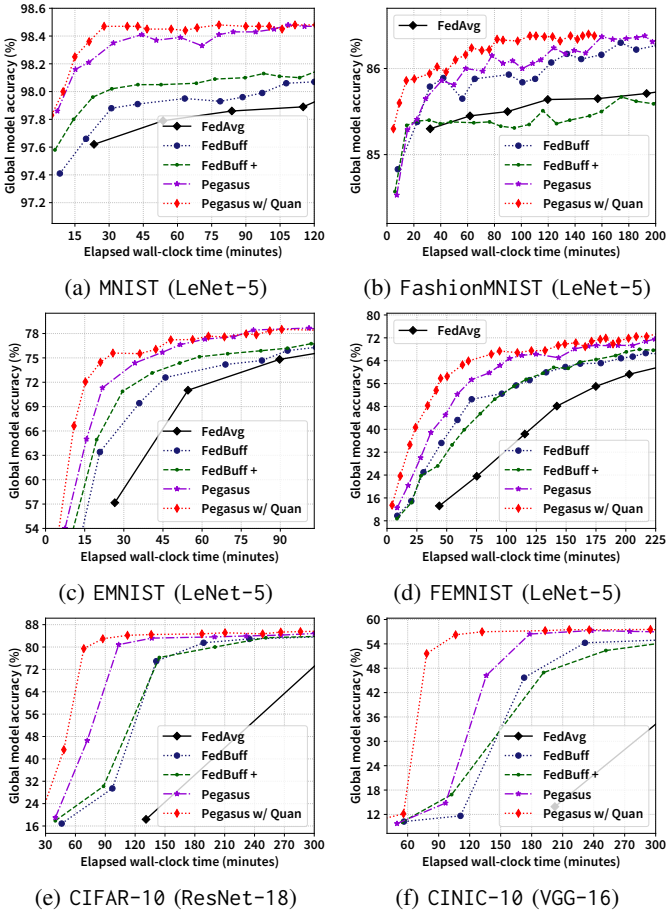
Across all six tasks, PEGASUS consistently outperformed its competitors, achieving the shortest time to any validation accuracy. By replacing random selection with its client selector, PEGASUS favored clients with higher-quality, less-stale updates, resulting in higher accuracy than *FedBuff* throughout training. Its adaptive pruning also reduced transmission time per round, further enhancing performance. In contrast, *FedBuff* with *Oort*'s client selection and *Sub-FedAvg*'s pruning performed only marginally better or even worse than vanilla *FedBuff* on the `FashionMNIST`, `CIFAR-10`, and `CINIC-10` datasets, reinforcing our view that both client and edge server staleness matter in asynchronous three-layer FL.

Evaluating PEGASUS with quantization. To further reduce wall-clock training time by cutting communication overhead, we applied quantization on the central server, edge servers, and clients, and evaluated its impact.

TABLE IV: Communication overhead to reach a target accuracy and converged global model accuracy.

Settings		Communication Overhead (GB); Converged Accuracy	
Dataset	Target Accuracy	PEGASUS	PEGASUS w/ Quantization
`MNIST`	98%	0.2; 98.49%	0.1 (50.0%); 98.49%
`FashionMNIST`	86%	0.6; 86.38%	0.3 (50.0%); 86.40%
`EMNIST`	75%	0.5; 79.70%	0.2 (60.0%); 79.39%
`FEMNIST`	72%	2.3; 76.34%	1.1 (52.2%); 76.47%
`CIFAR-10`	80%	21.1; 86.82%	12.8 (39.3%); 86.80%
`CINIC-10`	55%	35.7; 57.40%	16.9 (52.7%); 57.66%

As shown in Table IV, quantization significantly reduced communication overhead across all six training tasks by up to 60%. The percentages in brackets indicate the reduction achieved. Compared to PEGASUS without quantization, the converged global accuracy remained nearly unchanged, and for the `FashionMNIST`, `FEMNIST`, and `CINIC-10` tasks, it even



(a) `MNIST` (`LeNet-5`) (b) `FashionMNIST` (`LeNet-5`)

(c) `EMNIST` (`LeNet-5`) (d) `FEMNIST` (`LeNet-5`)

(e) `CIFAR-10` (`ResNet-18`) (f) `CINIC-10` (`VGG-16`)

Fig. 6: Performance comparison: PEGASUS vs. its competitors.

improved the accuracy.

Table V presents the wall-clock training times to reach target accuracy, with percentages in brackets indicating reductions compared to *FedBuff* + *Oort* + *Sub-FedAvg*. With quantization, PEGASUS further shortened training time, amplifying its superiority over its competitors.

Table VI shows the converged accuracies of global models under different three-layer FL mechanisms. *FedBuff* with *Oort* and *Sub-FedAvg* consistently converged to lower accuracies. In contrast, PEGASUS, both with and without quantization, often achieved higher accuracies (marked in blue cells). In other case, its accuracy was only slightly lower than that of *FedBuff*.

VII. CONCLUDING REMARKS

This paper studies asynchronous three-layer FL and aims to improve its training performance in terms of reducing wall-clock training time for the global model to converge to a target accuracy. By combining newly designed client selectors on edge servers, adaptive pruning mechanisms for clients and edge servers, local update aggregator on edge servers, and global update aggregator on the central server, we propose a staleness-aware framework, PEGASUS, to improve every important aspect of a training process. In our extensive evaluation of various training tasks, PEGASUS has indicated consistent and substantial superiority over its state-of-the-art

TABLE V: Elapsed wall-clock training time of global models trained with different mechanisms to reach a target accuracy.

Settings			Elapsed Wall-Clock Training Time (Hours)				
Dataset	Model	Target Accuracy	*FedAvg*	*FedBuff*	*FedBuff + Oort + Sub-FedAvg*	PEGASUS	PEGASUS w/ Quantization
MNIST	LeNet-5	98%	2.48	1.80	0.52	0.25 (51.92%)	0.16 (69.23%)
FashionMNIST	LeNet-5	85.5%	1.50	0.53	1.93	0.48 (75.13%)	0.15 (92.23%)
EMNIST	LeNet-5	75%	1.95	1.55	0.98	0.75 (23.47%)	0.43 (56.12%)
FEMNIST	LeNet-5	72%	8.95	7.22	6.54	4.44 (32.11%)	3.48 (46.79%)
CIFAR-10	ResNet-18	80%	6.78	3.15	3.34	1.72 (48.50%)	1.46 (56.29%)
CINIC-10	VGG-16	55%	6.77	5.37	5.75	2.97 (48.35%)	1.77 (69.22%)

TABLE VI: Converged accuracies of global models trained with different three-layer FL mechanisms.

Dataset	Model	Converged Global Model Accuracy				
		FedAvg	*FedBuff*	*FedBuff + Oort + Sub-FedAvg*	PEGASUS	PEGASUS w/ Quantization
MNIST	LeNet-5	98.21%	98.3%	98.22%	98.49%	98.49%
FashionMNIST	LeNet-5	86.02%	86.41%	85.74%	86.38%	86.40%
EMNIST	LeNet-5	79.29%	79.24%	78.56%	79.70%	79.39%
FEMNIST	LeNet-5	76.19%	76.10%	75.46%	76.34%	76.47%
CIFAR-10	ResNet-18	86.38%	86.76%	86.51%	86.82%	86.80%
CINIC-10	VGG-16	57.92%	56.02%	56.72%	57.40%	57.66%

competitors with respect to not only wall-clock training time but also converged accuracy of the global model.

REFERENCES

[1] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-Efficient Learning of Deep Networks from Decentralized Data," in *Proc. 20th International Conference on Artificial Intelligence and Statistics (AISTATS)*, vol. 54, 2017, pp. 1273–1282.

[2] L. Liu, J. Zhang, S. Song, and K. B. Letaief, "Client-Edge-Cloud Hierarchical Federated Learning," in *Proc. 2020 IEEE International Conference on Communications (ICC)*, 2020, pp. 1–6.

[3] J. Wang, S. Wang, R. Chen, and M. Ji, "Demystifying Why Local Aggregation Helps: Convergence Analysis of Hierarchical SGD," in *Proc. 36th AAAI Conference on Artificial Intelligence (AAAI)*, 2022, pp. 8548–8556.

[4] C. Liu, T. J. Chua, and J. Zhao, "Time Minimization in Hierarchical Federated Learning," in *Proc. 2022 ACM/IEEE Symposium on Edge Computing (SEC)*, 2022.

[5] L. Liu, J. Zhang, S. Song, and K. B. Letaief, "Hierarchical Federated Learning with Quantization: Convergence Analysis and System Design," *IEEE Trans. Wireless Communications*, 2022.

[6] M. S. H. Abad, E. Ozfatura, D. Gündüz, and Ö. Erçetin, "Hierarchical Federated Learning ACROSS Heterogeneous Cellular Networks," in *Proc. 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2020, pp. 8866–8870.

[7] C. Briggs, Z. Fan, and P. Andras, "Federated Learning with Hierarchical Clustering of Local Updates to Improve Training on Non-IID Data," in *Proc. 2020 International Joint Conference on Neural Networks (IJCNN)*, 2020, pp. 1–9.

[8] W. Wu, L. He, W. Lin, and R. Mao, "Accelerating Federated Learning Over Reliability-Agnostic Clients in Mobile Edge Computing Systems," *IEEE Trans. Parallel Distributed Syst.*, vol. 32, no. 7, pp. 1539–1551, 2021.

[9] Y. Zhao, M. Li, L. Lai, N. Suda, D. Civin, and V. Chandra, "Federated Learning with Non-IID Data," *arXiv preprint arXiv:1806.00582*, 2018.

[10] F. Sattler, S. Wiedemann, K.-R. Müller, and W. Samek, "Robust and Communication-Efficient Federated Learning from Non-iid Data," *IEEE Trans. Neural Networks and Learning Systems*, 2019.

[11] C. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. M. Hazelwood, E. Isaac, Y. Jia, B. Jia, T. Leyvand, H. Lu, Y. Lu, L. Qiao, B. Reagen, J. Spisak, F. Sun, A. Tulloch, P. Vajda, X. Wang, Y. Wang, B. Wasti, Y. Wu, R. Xian, S. Yoo, and P. Zhang, "Machine Learning at Facebook: Understanding Inference at the Edge," in *Proc. 25th IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 331–344.

[12] C. Yang, Q. Wang, M. Xu, Z. Chen, K. Bian, Y. Liu, and X. Liu, "Characterizing Impacts of Heterogeneity in Federated Learning upon Large-Scale Smartphone Data," in *Proc. The Web Conference 2021 (WWW)*, 2021, pp. 935–946.

[13] C. Xie, S. Koyejo, and I. Gupta, "Asynchronous Federated Optimization," in *Proc. NeurIPS Workshop on Optimization for Machine Learning (OPT)*, 2020.

[14] Y. Chen, Y. Ning, M. Slawski, and H. Rangwala, "Asynchronous Online Federated Learning for Edge Devices with Non-IID Data," in *Proc. 2020 IEEE International Conference on Big Data (IEEE BigData)*. IEEE, 2020, pp. 15–24.

[15] Y. Chen, X. Sun, and Y. Jin, "Communication-Efficient Federated Deep Learning With Layerwise Asynchronous Model Update and Temporally Weighted Aggregation," *IEEE Trans. Neural Networks and Learning Systems*, vol. 31, no. 10, pp. 4229–4238, 2020.

[16] J. Nguyen, K. Malik, H. Zhan, A. Yousefpour, M. Rabbat, M. Malek, and D. Huba, "Federated Learning with Buffered Asynchronous Aggregation," in *Proc. 25th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2022, pp. 3581–3607.

[17] N. Su and B. Li, "How Asynchronous can Federated Learning Be?" in *Proc. 29th IEEE/ACM International Symposium on Quality of Service (IWQoS)*, 2022.

[18] W. Wu, L. He, W. Lin, R. Mao, C. Maple, and S. A. Jarvis, "SAFA: A Semi-Asynchronous Protocol for Fast Federated Learning With Low Overhead," *IEEE Trans. Computers*, vol. 70, no. 5, pp. 655–668, 2021.

[19] F. Lai, X. Zhu, H. V. Madhyastha, and M. Chowdhury, "Oort: Efficient Federated Learning via Guided Participant Selection," in *Proc. 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021, pp. 19–35.

[20] A. Li, J. Sun, P. Li, Y. Pu, H. Li, and Y. Chen, "Hermes: An Efficient Federated Learning Framework for Heterogeneous Mobile Clients," in *Proc. 27th Annual International Conference on Mobile Computing and Networking (MobiCom)*, 2021, pp. 420–437.

[21] S. Vahidian, M. Morafah, and B. Lin, "Personalized Federated Learning by Structured and Unstructured Pruning under Data Heterogeneity," in *Proc. 41st IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 2021, pp. 27–34.

[22] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both Weights and Connections for Efficient Neural Network," in *Proc. 29th International Conference on Neural Information Processing Systems (NeurIPS)*, 2015, pp. 1135–1143.