HandoffFL: Efficient Communication in Federated Learning Via Asynchronous Layer-Wise Handoff Training

HandoffFL: Efficient Communication in Federated Learning via Asynchronous Layer-Wise Handoff Training

Wenjun Zhang*, Xiaoli Liu†*, Sasu Tarkoma*
*University of Helsinki, Finland. Email: {firstname.lastname}@helsinki.fi
†University of Vaasa, Finland. Email: xiaoli.liu@uwasa.fi

***Abstract*—Federated Learning (FL) enables collaborative training without sharing raw data, yet conventional synchronous approaches suffer from poor scalability and high communication costs, imposing heavy burdens on resource-constrained clients in heterogeneous edge environments. While asynchronous FL reduces heterogeneous stragglers, it introduces severe staleness and convergence degradation. Recent layer-wise FL methods reduce resource consumption by decomposing training into sequential layer-level tasks; however, they still rely on rigid round-based synchronization and therefore lack adaptability to dynamic client availability and network fluctuations. We propose HandoffFL, an event-driven asynchronous layer-wise FL framework built on a real-time message-passing monitoring architecture. HandoffFL organizes training into layer-level stages, where clients train one layer at a time with frozen prefixes. Transitions between layers are triggered by adaptive convergence events, enabling clients to progressively hand off converged layers and advance at their own pace. To support convergence detection at intermediate layers, the framework introduces temporal classifiers and further applies layer-specific staleness-aware aggregation with exponential decay and truncation. Extensive experiments on MNIST, Fashion-MNIST, and CIFAR-10 with multiple model architectures demonstrate that HandoffFL substantially reduces wall-clock training time and communication overhead compared to synchronous layer-wise FL, while maintaining comparable accuracy and robustness under heterogeneous client conditions compared with asynchronous FL.**

***Index Terms*—Asynchronous Federated Learning, Efficient Communication, Event-driven Mechanism, Layer-wise Handoff Training**

I. Introduction

The rapid proliferation of Internet of Things (IoT) has unlocked massive potential for distributed intelligence, yet centralized processing is increasingly constrained by stringent privacy regulations, such as the General Data Protection Regulation (GDPR), and prohibitive communication costs. While Federated Learning (FL) [1] has emerged as the standard paradigm to address these issues, its deployment in real-world edge environments remains severely limited by the resource constraints and heterogeneity of client devices [2]. Conventional synchronous FL deployed in healthcare [3], natural language processing (NLP) [4], human activity recognition [5], and transportation scenarios [6], which mandates full-model local training and strict global synchronization, imposes excessive computational burdens and suffers critically from straggler effects, resulting in unacceptable latency and energy inefficiency.

To mitigate these challenges, many approaches have been proposed, including model compression and quantization techniques [7] to reduce communication costs, as well as asynchronous [8], and semi-asynchronous FL [9] to alleviate synchronization bottlenecks. Specifically, event-driven FL [10, 11] has been explored to improve flexibility and responsiveness in asynchronous settings. Event-driven paradigms enable the server to aggregate model updates triggered by specific events (e.g., client arrival, deadline expiration, convergence threshold) rather than waiting for fixed synchronization rounds. This approach naturally accommodates irregular client arrival patterns, dynamic network, and computational conditions. It can significantly reduce idle waiting time and improve system throughput by exploiting asynchronous parallelism. While these event-driven approaches effectively mitigate synchronization bottlenecks and improve system throughput, they address only half the challenge. Crucially, they still premise that clients can afford the prohibitive memory and computational overhead of training the full model under an assumption that often fails in IoT scenarios.

To tackle this fundamental resource constraint, layer-wise FL has emerged as a promising paradigm. Instead of training the entire model, layer-wise FL decomposes the training process into sequential layer-level sub-tasks, enabling lightweight local updates with significantly reduced computational and memory costs [12–14]. By training one layer at a time while keeping previous layers frozen, clients can substantially lower per-iteration resource consumption. However, these existing layer-wise FL approaches predominantly rely on synchronous round-based protocols, where layer transitions are rigidly scheduled by the server at fixed global rounds, and client participation patterns remain static and predetermined. The framework does not adapt to real-time device state changes, such as irregular arrival times and network fluctuations. To the best of our knowledge, a systematic integration of layer-wise training within an event-driven federated optimization framework remains missing. There exists a critical methodological gap in combining the fine-grained resource efficiency of layer-wise training with the adaptive responsiveness of event-driven aggregation.

This research was supported by NordForsk Nordic University Cooperation on Edge Intelligence (Grant No. 168043). (Corresponding author: Xiaoli Liu.)

However, bridging this gap is non-trivial and introduces three distinct technical challenges: First, different layers (e.g., convolutional vs. fully-connected) exhibit vastly different computational costs and communication sizes, indicating resource heterogeneity across layers. Designing an event-driven scheduling policy that respects these heterogeneous resource profiles to minimize wall-clock time remains unclear. Second, there exists a conflict between sequential dependency and asynchronous ambiguity. Unlike full-model training, layer-wise training imposes a strict dependency ($l \rightarrow l+1$) requiring specific per-layer convergence criteria. In event-driven settings without global barriers, determining exactly when a layer is sufficiently trained to trigger a handoff becomes highly ambiguous, requiring a robust and adaptive detection method. Third, asynchronous updates naturally introduce staleness. Directly applying existing mitigation techniques is insufficient, as staleness must now be managed at the fine-grained layer level rather than for the entire model to prevent version conflicts during transitions.

To address these challenges, we propose HandoffFL, an event-driven layer-wise FL framework that systematically integrates layer-wise training with event-driven optimization. Overall, our main contributions are summarized as follows:

- We propose HandoffFL, the first framework that systematically integrates layer-wise training with event-driven optimization, addressing the critical gap between resource efficiency and adaptive asynchronous aggregation.
- We design a novel event-driven architecture based on message-passing and priority queuing, enabling fully asynchronous layer-wise training with explicit modeling of communication delays, computational heterogeneity, and dynamic client participation.
- We introduce temporal classifiers and staleness-aware aggregation specifically tailored for layer-wise training, enabling robust per-layer convergence evaluation and version-aware aggregation in asynchronous settings.
- We demonstrate through extensive experiments on MNIST, Fashion-MNIST, and CIFAR-10 using LeNet and VGG9 architectures that HandoffFL outperforms state-of-the-art baselines, reducing training latency by 64% compared to synchronous approaches while maintaining comparable accuracy.

II. RELATED WORK

A. Layer-wise Training of Deep Networks

Although deep networks are known to be significantly more expressive than shallow architectures, training very deep networks in practice has long remained challenging. As depth increases, optimization often becomes slower and more unstable, with issues such as vanishing gradients and a strong dependence on careful initialization or pretraining. When trained directly from random initialization using standard backpropagation, deep networks are prone to converging to suboptimal local minima.

To address these challenges, Hinton, Osindero, and Teh [15] proposed a greedy layer-wise unsupervised learning algorithm for Deep Belief Networks (DBNs), a class of generative models composed of multiple layers of latent variables. In this framework, the network is trained incrementally, with each layer optimized using an unsupervised objective to learn representations that best explain the output of the previous layer, followed by a global fine-tuning stage using backpropagation. Bengio et al. [16] further analyzed this paradigm and provided both theoretical and empirical explanations for why greedy layer-wise unsupervised training is effective and they showed that the advantages of layer-wise pretraining primarily arise from improved optimization and parameter initialization, rather than from any particular model choice.

These early studies established greedy layer-wise training as a practical approach for enabling the optimization of deep architectures, typically by progressively adding and training layers under unsupervised objectives. A related but more explicitly supervised line of work is deep supervision, which retains end-to-end backpropagation while introducing auxiliary classification branches at intermediate layers. Wang et al. [17] proposed that by injecting additional supervision signals at multiple depths, deep supervision helps alleviate gradient vanishing and improves the trainability of very deep convolutional networks, without fundamentally altering the global learning objective. In parallel, Jaderberg et al. [18] indicated that decoupling approaches such as Decoupled Neural Interfaces introduce synthetic gradients to relax backward-locking, enabling modules to update without waiting for exact backpropagated gradients, thereby moving from sequential dependence toward more independent (potentially asynchronous) module updates. More recently, Arild Nøkland and Lars H. Eidnes [19] proposed a local learning paradigm that explicitly replaces the single global loss with a combination of layer-wise local loss functions. Specifically, each hidden layer is supervised by locally generated error signals, without relying on globally back-propagated gradients.

The development of layer-wise paradigms has naturally aligned with challenges of FL, particularly under limited memory, communication, and computation resources. By decoupling end-to-end optimization into layer-level updates, layer-wise methods reduce backward dependency and memory overhead, making them well-suited for resource-constrained FL settings. Mo et al. [12] further extended it to trusted execution environments (TEEs) by adopting greedy layer-wise, enabling deep model training within the limited secure memory available in TEEs while maintaining competitive performance. At the algorithmic level, Chen et al. [13] proposed FL-Joint, which incorporates auxiliary loss functions into local optimization to jointly align feature and label distributions across heterogeneous clients, while Liu et al. [14] proposed PFPS-LWC, which leverages lightweight local classifiers as auxiliary components to support personalized training and alleviate classifier overfitting and catastrophic forgetting.

B. Event-driven Paradigm in Asynchronous FL

Traditional FL methods are predominantly evaluated under a round-based synchronous paradigm, where the server selects

a subset of clients, each client performs local training, and the server aggregates model updates only after all selected clients have uploaded their results. While round-based FL is convenient for simulation and theoretical analysis, it suffers from well-known limitations in practical deployments. In particular, straggler effects, heterogeneous computation and communication capabilities, and unreliable network conditions can significantly delay each training round. As a result, fast devices are forced to idle while waiting for slow clients, leading to poor system utilization and wasted CPU/GPU resources at both the server and client sides.

To alleviate strict synchronization, asynchronous FL has been proposed as an alternative. Xie et al. [8] proposed FedAsync that replaces round-based synchronization with an arrival-driven update mechanism, where the server immediately incorporates each incoming client update into the global model. Building on this idea, Chen et al. [20] introduced FedSA, which observes that in realistic settings with non-IID data distributions and unreliable or slow clients, stale updates can be particularly harmful. FedSA therefore employed staleness-aware asynchronous aggregation strategies to improve robustness under heterogeneous systems. However, each arriving update is immediately aggregated; the global model is frequently perturbed by noisy or severely stale updates, which may harm convergence stability.

To address this issue, Nguyen et al. [10] proposed FedBuff that introduces buffered asynchronous aggregation where the server accumulates a buffer of multiple client updates and triggers aggregation only when the buffer is filled. However, buffering also introduces a new challenge. When client updates finally arrive at the server, they are often computed based on outdated global models, especially in highly asynchronous environments, exacerbating the stale-model problem.

More recently, Liu et al. [11] introduced FedASMU that further refines staleness handling by dynamically adjusting the importance of client updates using both staleness information and local training loss. In addition, complementary to fully asynchronous and buffered designs, semi-asynchronous FL has emerged as a middle ground between strict synchronization and pure asynchrony. Islam et al. [9] proposed SEAFL that targeted this setting by combining adaptive aggregation with selective training. Instead of aggregating every update immediately or waiting for full synchronization, SEAFL assigns aggregation weights based on both staleness and the estimated importance of updates to the current global model. Furthermore, its selective training mechanism allows slower or resource-constrained devices to contribute without forcing the system to wait excessively, thereby mitigating both straggler effects and stale-model issues.

III. SYSTEM MODEL AND PROBLEM DEFINITION

A. FL Environment Definition

We consider an FL system consisting of a central server $\mathcal{S}$ and N distributed clients $\mathcal{C} = \{c_1, c_2, \ldots, c_N\}$. Each client c_i holds a local private dataset $\mathcal{D}_i = \{(x_j, y_j)\}_{j=1}^{n_i}$, where n_i denotes the number of local samples. In conventional FL, the global objective is to jointly optimize all model parameters θ to minimize:

$$\min_{\theta} F(\theta) = \sum_{i=1}^{N} \frac{n_i}{n} F_i(\theta), \quad F_i(\theta) = \frac{1}{n_i} \sum_{(x,y)\in\mathcal{D}_i} \ell(\theta; x, y), \tag{1}$$

where $n = \sum_{i=1}^{N} n_i$ is the total number of samples and $\ell(\cdot)$ is the loss function. The global model is a deep neural network composed of L sequential layers $\mathcal{L} = \{l_0, l_1, \ldots, l_{L-1}\}$. The complete model parameters are decomposed as:

$$\theta = \{\theta_0, \theta_1, \ldots, \theta_{L-1}\}, \tag{2}$$

where θ_l represents the parameters of layer l. For convolutional neural networks, layers typically include convolutional layers, pooling layers, and fully-connected layers. The forward propagation can be expressed as:

$$h_l = f_l(h_{l-1}; \boldsymbol{\theta}_l), \quad l = 0, 1, \ldots, L-1, \tag{3}$$

where h_l denotes the output feature map of the l-th layer, with $h_{-1} = \boldsymbol{x}$ representing the network input, and $f_l(\cdot)$ denotes the layer-wise transformation parameterized by $\boldsymbol{\theta}_l$, such as convolution, nonlinear activation, or linear mapping.

B. Communication Latency in FL

Following the Shannon capacity principle adopted in wireless FL [21], we model the heterogeneous network bandwidth available to each client c_i. To capture channel variability across clients, the effective bandwidth R_i (in megabits per second) is modeled as a log-normal random variable:

$$R_i \sim \mathrm{LogNormal}(\mu_R, \sigma_R^2), \tag{4}$$

where the parameters μ_R and σ_R could be chosen such that the mean bandwidth is 5 Mbps (representative of heterogeneous wireless networks including 4G/WiFi) and $\sigma_R = 0.5$ to capture moderate heterogeneity.

This statistical formulation is motivated by the Shannon capacity expression $C = B\log_2(1+\mathrm{SNR})$, where the signal-to-noise ratio (SNR) varies multiplicatively with distance, shadowing, and allocated spectrum. The log-normal distribution reflects these multiplicative effects commonly observed in wireless channels [21].

For layer l with parameter tensor $\boldsymbol{\theta}_l \in \mathbb{R}^{d_l}$, the communication payload size is:

$$|\boldsymbol{\theta}_l| = d_l \times 4 \text{ bytes}, \tag{5}$$

assuming 32-bit floating-point (FP32) representation. In layer-wise training, only the parameters of layer l are transmitted, significantly reducing communication overhead compared to full-model transmission.

The downlink transmission time for the server to broadcast layer l parameters to a client c_i is:

$$t_i^{\mathrm{down}}(l) = \frac{8|\boldsymbol{\theta}_l|}{R_i}, \tag{6}$$

where the factor 8 converts bytes to bits. Similarly, the uplink transmission time from client c_i to upload its trained layer l parameters to the server is:

$$t_i^{\mathrm{up}}(l) = \frac{8|\boldsymbol{\theta}_l|}{R_i}. \tag{7}$$

We assume symmetric bandwidth for simplicity, though the model can be extended to asymmetric settings by introducing separate R_i^{up} and R_i^{down} parameters. The total communication latency for client c_i for layer l is:

$$t_i^{\mathrm{comm}}(l) = t_i^{\mathrm{down}}(l) + t_i^{\mathrm{up}}(l) = \frac{2 \cdot 8|\boldsymbol{\theta}_l|}{R_i}. \tag{8}$$

Due to the heterogeneity in bandwidth R_i, communication latency is client-specific, creating natural asynchrony in parameter exchange even when clients have identical computational capabilities.

IV. HandoffFL Design

A. Layer-wise Optimization Framework

Unlike conventional FL, which jointly optimizes all model parameters $\theta = \{\theta_0, \ldots, \theta_{L-1}\}$ in an end-to-end manner, our HandoffFL framework reformulates the training process as a sequential layer-wise optimization problem. Instead of updating the entire model simultaneously, the training is decomposed into a series of subproblems, each focusing on a single layer while keeping previously trained layers fixed.

For each layer $l = 0, 1, \ldots, L-1$, the global optimization objective is defined as

$$\theta_l^* = \arg\min_{\theta_l} F_l(\theta_l|\theta^*_{\text{prefix}}) = \arg\min_{\theta_l} \sum_{i=1}^{N} \frac{n_i}{n} F_{i,l}(\theta_l|\theta^*_{\text{prefix}}), \tag{9}$$

where $\theta^*_{\text{prefix}} = \{\theta_0^*, \theta_1^*, \ldots, \theta_{l-1}^*\}$ denotes the frozen prefix parameters obtained from previously converged layers, n_i is the local dataset size of client c_i, and $n = \sum_{i=1}^{N} n_i$.

The local objective function of client c_i for training layer l is given by

$$F_{i,l}(\theta_l|\theta^*_{\text{prefix}}) = \frac{1}{n_i} \sum_{(x,y)\in\mathcal{D}_i} \ell_l(\theta_l, \theta^*_{\text{prefix}}; x, y), \tag{10}$$

where $\mathcal{D}_i$ denotes the local dataset of client c_i. The sample-wise loss $\ell_l(\cdot)$ is computed using a partial model composed of the frozen prefix θ^*_{prefix}, the trainable layer θ_l, and a layer-specific temporal classifier ϕ_l, i.e.,

$$\ell_l(\theta_l, \theta^*_{\text{prefix}}; x, y) = \mathcal{L}(\phi_l(h_l), y), \tag{11}$$

where $h_l = f_l(h_{l-1}; \theta_l)$ is the feature representation at layer l, obtained by forward propagation through the frozen prefix and the current trainable layer.

As shown in Fig. 1, the optimization proceeds sequentially across layers following a layer-wise training protocol. Specifically, training starts from the first layer ($l = 0$), where θ_0 is optimized without any preceding layers. Once θ_0^* is obtained, it is frozen and treated as fixed when optimizing the next layer. For a general layer l, the optimization focuses exclusively on θ_l, while all previously trained parameters $\{\theta_0^*, \ldots, \theta_{l-1}^*\}$ remain fixed. This process continues iteratively until all L layers are trained and their corresponding optimal parameters obtained.

At any given time, each client c_i participates in the training of at most one layer l. During this process, all layers preceding l remain frozen and are fixed to the latest globally aggregated parameters. Formally, when client c_i is assigned to train layer l, its local training operates on a partial model parameterized as

$$\theta_i^{(\text{partial})} = \{\theta_0^{\text{global}}, \theta_1^{\text{global}}, \ldots, \theta_{l-1}^{\text{global}}, \theta_l^{(i)}\}, \tag{12}$$

where θ_j^{global} for $j < l$ denote the globally aggregated and frozen parameters of previous layers, and $\theta_l^{(i)}$ represents the trainable parameters of layer l at client c_i, initialized from the latest global model.

B. Temporal Classifiers for Intermediate Convergence

A critical challenge in layer-wise training lies in assessing the convergence of intermediate layers without access to subsequent layers. Since the representations produced by an intermediate layer cannot be directly evaluated by the final task loss, a dedicated mechanism is required to provide meaningful supervision during training. To address this challenge, we introduce a set of temporal classifiers ϕ_l, which are temporarily attached to each layer l during training and removed once the corresponding layer converges.

The temporal classifiers are constructed based on the Global Average Pooling (GAP) mechanism [22, 23], enabling lightweight and layer-adaptive supervision. Specifically, the temporal classifier ϕ_l for layer l is defined as

$$\phi_l(h_l) = \begin{cases} \text{Linear}(\text{GAP}(h_l)), & l \in \mathcal{L}_{\text{conv}}, \\ \text{FC}_{l+1} \circ \cdots \circ \text{FC}_{L-1}, & l \in \mathcal{L}_{\text{fc}}, \end{cases} \tag{13}$$

where $\mathcal{L}_{\text{conv}}$ and $\mathcal{L}_{\text{fc}}$ denote the sets of convolutional and fully-connected layers, respectively. Notably, the temporal classifiers ϕ_l are used exclusively for local training and convergence evaluation; they are discarded after the corresponding layer converges and neither uploaded nor aggregated at the server.

Upon receiving a layer assignment for layer l, client c_i performs the following local training procedure.

Initialization: The client loads the frozen prefix parameters $\{\theta_0^{\text{global}}, \ldots, \theta_{l-1}^{\text{global}}\}$ stored own together with the current layer parameters θ_l^{global} downloaded from server.

Temporal Classifier Attachment: A temporal classifier ϕ_l is constructed and attached to the output of layer l to provide auxiliary supervision.

Convergence Evaluation: The convergence of layer l is evaluated on the local validation set $\mathcal{D}_i^{\text{val}}$ using the temporal classifier, based on validation loss or prediction accuracy.

Local Optimization: The client performs local training for E epochs using stochastic gradient descent (SGD),

$$\theta_l^{(i)} \leftarrow \theta_l^{(i)} - \eta \nabla_{\theta_l} \mathcal{L}(\theta_i^{(\text{partial})}; \mathcal{B}), \tag{14}$$

where $\mathcal{B} \subset \mathcal{D}_i$ denotes a mini-batch sampled from the local dataset, and η is the learning rate.

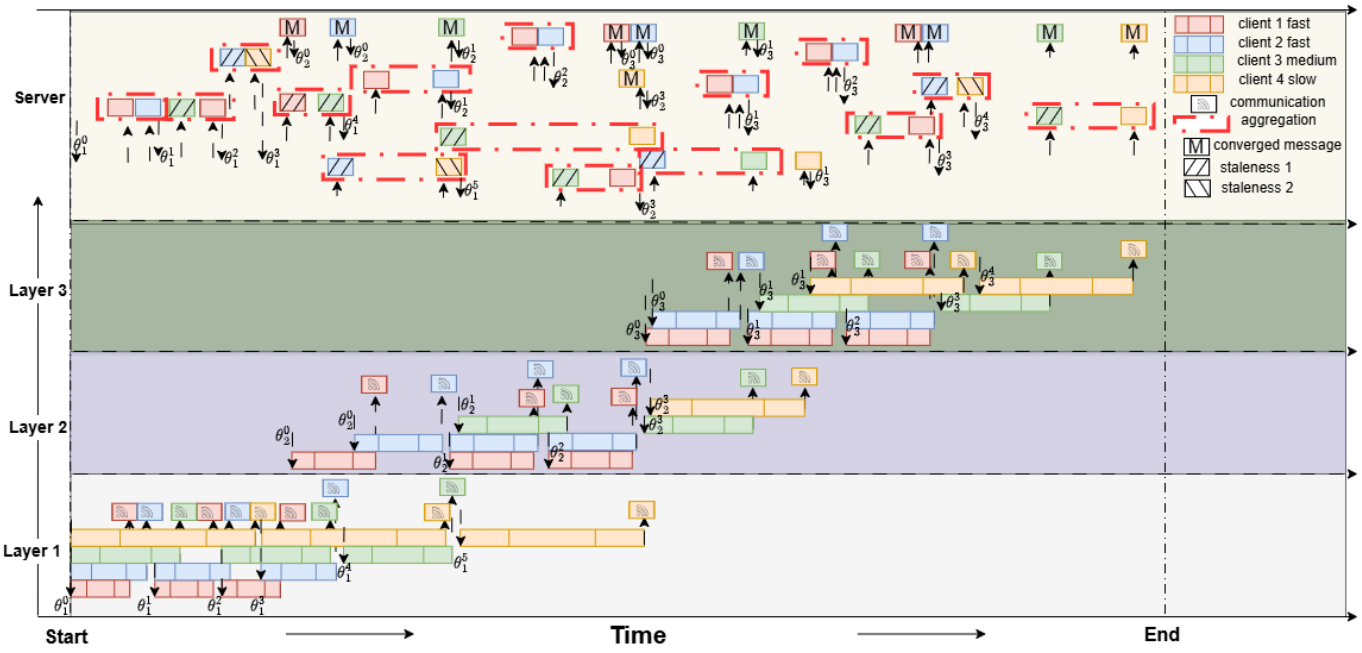

Fig. 1: Timeline Visualization Example of Event-Driven Layer-Wise HandoffFL Framework. It captures the asynchronous training dynamics of four heterogeneous clients across three sequential layers over wall-clock time. Each horizontal bar represents a complete training iteration, decomposed into download (left), training (center), and upload (right) phases, reflecting different communication and computation costs consumed. The server assigns aggregated weights considering staleness to clients layer by layer, and clients evaluate and train locally as well as send updates to the server. Clients asynchronously transition to next layer without synchronization barriers once it converges.

Update Transmission: Upon completion of local training, the client sends the updated layer parameters $\theta_l^{(i)}$ along with relevant training metadata to the server.

C. Layer Convergence and Handoff Protocol

As discussed earlier, real-world FL systems are characterized by client heterogeneity in terms of both computational capability and communication bandwidth. While resource-rich clients can complete local training and model updates rapidly, resource-constrained edge devices often experience slow training progress or even convergence difficulties. This heterogeneity motivates a collaborative training paradigm in which well-trained layer parameters produced by faster clients can be shared via the server with slower clients, allowing the latter to reuse pretrained layers and bypass expensive local optimization of early stages.

Motivated by this observation, our proposed event-driven layer-wise HandoffFL framework adopts a layer-wise convergence mechanism instead of relying on a centralized validation criterion across all clients. In highly asynchronous settings, global convergence tests may prematurely declare convergence for many clients that have not yet fully converged. In contrast, layer-wise convergence more accurately reflects practical training dynamics in heterogeneous edge systems, where different clients train different layers at different times, and completed layers can be handed off to others.

Accordingly, we design a layer-wise convergence criterion to determine when a given layer has been sufficiently optimized. The convergence of layer l is assessed based on the relative improvement rate of its performance metric. Specifically, layer l is considered to have converged if the relative improvement over a lookback window falls below a predefined threshold:

$$\text{converged}(l) \iff \frac{\text{acc}_l^{(t)} - \text{acc}_l^{(t-k)}}{\text{acc}_l^{(t-k)}} < \delta, \qquad (15)$$

where k denotes the lookback window size (e.g., $k = 4$ rounds) and δ is the convergence threshold (e.g., $\delta = 0.01$ corresponding to a 1% improvement rate). This criterion captures diminishing returns in layer-wise optimization and enables timely progression to subsequent layers.

Each client c_i independently evaluates the convergence condition of layer l using its local validation set $\mathcal{D}_i^{\text{val}}$. Once a client determines that layer l has converged locally, it immediately sends a CONVERGED message to the server and proceeds to train the next layer $l + 1$ without waiting for other clients. To ensure system-wide progress, the server enforces global convergence for layer l when the number of remaining active clients falls below $K_{\min}$.

Upon convergence of layer l, either through local client decisions or server-enforced global convergence, the following actions are performed:

Algorithm 1 Event-Driven Layer-Wise HandoffFL

Input: $\mathcal{S}$, $\mathcal{C} = \{c_1, \ldots, c_N\}$, $\mathcal{L} = \{l_0, \ldots, l_{L-1}\}$
Input: Client capabilities: $\{\alpha_i, \text{bw}_i\}$ for $c_i \in \mathcal{C}$
Input: Staleness threshold $S_{\max}$, half-life h, contributor $K_{\min}$
Output: Trained model $\{\theta_0, \ldots, \theta_{L-1}\}$

// Initialization
$\mathcal{S}$.init($\mathcal{L}$) ▷ Initialize global parameters
$\mathcal{Q} \leftarrow \emptyset$ ▷ Priority queue sorted by timestamp
$V[l] \leftarrow 0$ for all $l \in \mathcal{L}$ ▷ Layer version counters
// Calibrate client heterogeneity
for each client $c_i \in \mathcal{C}$ **do**
  $t_0 \leftarrow c_i$.measure_training_time($l = 0$, epochs $= 1$)
  $\alpha_i \leftarrow t_{\text{baseline}}/t_0$ ▷ Speed factor (2.0=fast, 0.5=slow)
end for
// Client enrollment
for each client $c_i \in \mathcal{C}$ **do**
  $\mathcal{Q}$.push(JOIN, $c_i, t = 0$)
end for
// Event-driven training loop
while $\mathcal{Q} \neq \emptyset$ **and** not all layers converged **do**
  $m \leftarrow \mathcal{Q}$.pop() ▷ Get earliest message
  $\mathcal{S}.t \leftarrow \max(\mathcal{S}.t, m.t)$ ▷ Update wall-clock
  if m.type = JOIN **then**
    $c_i \leftarrow m$.sender
    Send ASSIGN($l = 0, \theta_0$) to c_i ▷ Assign first layer
  else if m.type = ASSIGN **then**
    $c_i \leftarrow m$.rcv
    $l \leftarrow m$.layer
    $\theta_l \leftarrow m$.params
    $t_d \leftarrow 4|\theta_l|/\text{bw}_i$ ▷ Download time (client-specific)
    if local_val **and** c_i.converged(l) **then**
      $\mathcal{Q}$.push(CONV, $c_i, l, m.t + t_d$)
    else
      θ'_l, info $\leftarrow c_i$.train(l, θ_l) ▷ Train layer (info.t uses α_i)
      $t_u \leftarrow 4|\theta'_l|/\text{bw}_i$ ▷ Upload time (client-specific)
      $t_c \leftarrow m.t + t_d +$ info.$t + t_u$ ▷ Total completion time
      $\mathcal{Q}$.push(UPDATE, c_i, θ'_l, t_c)
    end if
  else if m.type $\in$ {UPDATE, CONV} **then**
    HANDLEUPDATE(m) ▷ Algorithm 2
  end if
end while
return $\{\theta_0, \ldots, \theta_{L-1}\}$

Algorithm 2 Server: HandleUpdate(m)

Input: Message m from client c_i
$c_i \leftarrow m$.sender
$l \leftarrow m$.layer
// Handle converged clients
if m.type = CONV **then**
  Mark c_i converged for layer l
  Assign c_i to layer $l + 1$ ▷ Move to next layer
  check force satus of l ▷ Check force convergence
  return
end if
// Compute staleness and weight
$v_{\text{train}} \leftarrow m$.ver ▷ Training version
$s \leftarrow V[l] - v_{\text{train}}$ ▷ Staleness
if $s > S_{\max}$ **then** ▷ Hard truncation
  Reject update; **return**
end if
$w \leftarrow 2^{-s/h}$ ▷ Exponential decay weight
$\mathcal{U}[l] \leftarrow \mathcal{U}[l] \cup \{(c_i, m.\text{params}, w, m.t)\}$
// Trigger aggregation if enough updates
if $|\mathcal{U}[l]| \geq K_{\min}$ **then**
  $W \leftarrow \sum_{u \in \mathcal{U}[l]} u.w$ ▷ Total weight
  $\theta_l \leftarrow \frac{1}{W} \sum_{u \in \mathcal{U}[l]} u.w \cdot u.\theta$ ▷ Weighted avg
  $V[l] \leftarrow V[l] + 1$ ▷ Increment version
  $R[l] \leftarrow R[l] + 1$ ▷ Aggregation round counter
  // Check layer convergence
  converged $\leftarrow$ check convergence threshold l
  if converged **then**
    Broadcast: assign all active clients to $l + 1$
  else
    Assign participants to layer l ▷ Continue training
  end if
  $\mathcal{U}[l] \leftarrow \emptyset$ ▷ Clear pending updates
end if

- **Layer Freezing:** The global parameters θ_l^{global} are frozen and no longer subject to further updates.
- **Layer Transition:** Active clients are notified to transition to layer $l + 1$ and receive the updated prefix parameters $\{\theta_0^{\text{global}}, \ldots, \theta_l^{\text{global}}\}$, along with the initialization parameters $\theta_{l+1}^{\text{global}}$ for the next layer.
- **Training Resumption:** Clients resume local training on layer $l + 1$ with the frozen prefix $\{\theta_0, \ldots, \theta_l\}$ fixed.

This layer-wise convergence and handoff procedure is repeated sequentially for all L layers until the model converges.

D. Heterogeneity-Aware Wall-Clock Time Modeling

Building upon the communication latency model established in Section III-B, HandoffFL implements a practical heterogeneity-aware timing mechanism to simulate realistic wall-clock execution in an event-driven framework. Each client c_i is characterized by a capability profile $\mathcal{R}_i = (\alpha_i, \text{bw}_i)$, where α_i captures computational heterogeneity and bw_i represents effective network bandwidth. These parameters enable accurate wall-clock time estimation for training and communication events.

1) Calibration-Based Computational Profiling: Unlike analytical models that require detailed hardware specifications, HandoffFL employs empirical calibration to measure client computational capability. During system initialization, each client c_i executes a standardized calibration task, training one epoch on layer $l = 0$ with fixed hyperparameters (batch size B, learning rate η). Let $t_i^{(0)}$ denote the measured wall-clock time for client i, and t_{baseline} be the time from a designated reference client. The computational speed factor is computed as:

$$\alpha_i = \frac{t_{\text{baseline}}}{t_i^{(0)}}. \tag{16}$$

This empirical approach captures the aggregate effect of hardware characteristics (CPU/GPU architecture, memory hierarchy, compiler optimizations) on the specific deep learning workload, without requiring explicit hardware profiling. The speed factor satisfies $\alpha_i = 1.0$ for the baseline, $\alpha_i > 1.0$ for faster clients, and $\alpha_i < 1.0$ for slower clients.

2) Wall-Clock Time Estimation: For a given training task involving N gradient descent steps, the computational cost is

estimated as:

$$T_{\text{comp}}^{(i)} = \frac{t_{\text{step}} \cdot N}{\alpha_i}, \tag{17}$$

where t_{step} is the reference per-step time obtained from calibration. Communication latency for layer l is derived from the bandwidth parameter bw_i and layer size $|\boldsymbol{\theta}_l|$ (as defined in Section III-B):

$$T_{\text{comm}}^{(i)}(l) = \frac{2 \cdot 8 \cdot |\boldsymbol{\theta}_l|}{\text{bw}_i \cdot 10^6}, \tag{18}$$

where the factor 2 accounts for bidirectional parameter exchange (download + upload). The total wall-clock time for client i to complete training on layer l is then:

$$T_{\text{total}}^{(i)}(l) = T_{\text{comp}}^{(i)} + T_{\text{comm}}^{(i)}(l). \tag{19}$$

These timing estimates are used by the event-driven simulator to schedule message arrivals and maintain causal consistency in asynchronous execution.

E. Staleness-Aware Aggregation

In HandoffFL framework, the server maintains a layer-wise version counter $V[l] \in \mathbb{N}$ for each layer $l = 0, 1, \ldots, L-1$. The version counter is incremented whenever an aggregation is performed on the corresponding layer, i.e., $V[l] \leftarrow V[l] + 1$. These version counters enable explicit tracking of update staleness in the asynchronous training process.

When a client c_i submits an update for layer l, the update is associated with the version $v_{\text{train}}^{(i)}$ of the global model on which the client started its local training. Upon arrival at the server, the staleness of the update is computed as

$$s_i = V[l] - v_{\text{train}}^{(i)}, \tag{20}$$

which quantifies the number of aggregation events occurred for layer l since the client initiated its local optimization.

To mitigate the adverse impact of stale updates, the server assigns a staleness-aware aggregation weight w_i to each update according to

$$w_i = \begin{cases} 2^{-s_i/h}, & \text{if } s_i \le S_{\max}, \\ 0, & \text{if } s_i > S_{\max}, \end{cases} \tag{21}$$

where $h > 0$ is a half-life parameter controlling the exponential decay rate of the aggregation weight, and $S_{\max}$ is a staleness threshold for hard truncation. Under this weighting scheme, the contribution of an update decays exponentially with its staleness, and the weight is reduced by half for every h additional versions of staleness. Updates whose staleness exceeds $S_{\max}$ are discarded to prevent outdated information from adversely affecting the global model.

Once at least $K_{\min}$ updates are available for layer l, the server performs weighted aggregation over the set of pending updates $\mathcal{U}_l$:

$$\theta_l^{\text{new}} = \frac{1}{W} \sum_{i \in \mathcal{U}_l} w_i \theta_l^{(i)}, \tag{22}$$

where $W = \sum_{i \in \mathcal{U}_l} w_i$ denotes the normalization factor.

After aggregation, the global parameters of layer l are updated as $\theta_l^{\text{global}} \leftarrow \theta_l^{\text{new}}$, the corresponding version counter is incremented, i.e., $V[l] \leftarrow V[l] + 1$, and the set of pending updates is cleared $\mathcal{U}_l \leftarrow \emptyset$.

F. Event-Driven Execution Model

1) Message-Passing Architecture: The event-driven layer-wise HandoffFL is implemented as a discrete event simulation (DES), detailed in Algorithm 1. All interactions between the server and clients are encapsulated as timestamped messages. The server maintains a global priority queue $\mathcal{Q}$ that stores pending messages ordered by their arrival times:

$$\mathcal{Q} = \{(m_1, t_1), (m_2, t_2), \ldots\}, \quad \text{where } t_1 \le t_2 \le \cdots. \tag{23}$$

Prior to the training loop, the system performs a calibration phase to quantify client heterogeneity. As shown in Algorithm 1 (lines 5-8), each client c_i is assigned a speed factor α_i relative to a baseline, and its network bandwidth bw_i is recorded. These capabilities determine the simulation of physical delays.

Each message m in the queue is characterized by its type $m.\text{type} \in \{\text{JOIN}, \text{ASSIGN}, \text{UPDATE}, \text{CONVERGED}\}$, sender/receiver identifiers, and a payload containing layer parameters θ_l. Crucially, the timestamp $m.t$ simulates the wall-clock time. For instance, when a client receives an ASSIGN message at time t, it does not respond immediately. Instead, the system calculates the expected completion time t_c by summing the simulated download time (t_d), scaled local training time ($\alpha_i \cdot t_{\text{train}}$), and upload time ($t_u$). An UPDATE event is then pushed to $\mathcal{Q}$ with timestamp t_c, accurately modeling the asynchronous arrival of stragglers.

2) Asynchronous Aggregation and Time Tracking: The framework maintains a global wall-clock time variable t_{current}, which ensures causal consistency by advancing only when the server processes the earliest message m from $\mathcal{Q}$:

$$t_{\text{current}} \leftarrow \max(t_{\text{current}}, m.t). \tag{24}$$

Upon processing an UPDATE message, the server executes the staleness-aware aggregation logic described in Algorithm 2. To address the "stale gradient" problem inherent in asynchronous FL, the server computes the staleness $s = V[l] - v_{\text{train}}$ of the received model. Updates exceeding a staleness threshold $S_{\max}$ are rejected (hard truncation), while valid updates are assigned an exponential decay weight $w = 2^{-s/h}$ to down-weight older gradients, detailed also in Section IV-E.

In contrast to synchronous FL, our event-driven workflow exhibits true asynchrony in three key aspects:

- **No Synchronization Barriers:** Clients fetch the latest global parameters and upload updates independently without waiting for a global round to finish.
- **Partial Aggregation:** As defined in Algorithm 2 (lines 14-18), the server performs a weighted aggregation immediately once a buffer of $K_{\min}$ updates is collected, rather than waiting for all N clients.
- **Seamless Layer Handoff:** Straggler effects are naturally mitigated. Faster clients that complete the current layer l are not held back by slower ones; once layer convergence

is detected, the system triggers a handoff, allowing active participants to progress to layer $l + 1$.

V. Experiments and Evaluation

A. Experiment Setup

1) Datasets and Models: In our experiments, we employ three standard benchmark datasets widely utilized in FL research: MNIST, Fashion-MNIST, and CIFAR-10. The MNIST [24] dataset, a cornerstone benchmark for handwritten digit recognition, comprises 60k training samples and 10k test samples of 28×28 grayscale pixels distributed across 10 digit classes (0-9). As a more challenging alternative, we utilize Fashion-MNIST [25], which shares the same dataset size and resolution as MNIST but features 10 distinct fashion product categories, i.e., T-shirts, dresses, and sneakers. Finally, we select the CIFAR-10 [26] dataset. This dataset consists of 32×32 RGB images divided into 50k training and 10k testing samples, spanning 10 diverse object classes such as airplanes, automobiles, and animals. Meanwhile, we deploy two Convolutional Neural Network (CNN) architectures of varying depth and computational complexity: LeNet and VGG9. LeNet [27], a lightweight architecture well-suited for grayscale image tasks, consists of two convolutional layers followed by pooling and two fully connected layers. For more complex classification tasks, we employ a deeper architecture VGG9 [28], which adopts the design philosophy of the VGG family by utilizing small 3×3 convolutional filters. It is structured with six convolutional layers organized into blocks, each followed by max-pooling, and then three fully connected layers.

Instead of relying on traditional centralized evaluation, HandoffFL implements a local validation strategy to ensure raw data remains decentralized, thereby reducing both privacy risks and communication costs. In this setup, the dataset is distributed across 50 clients, and each client performs an 8:2 split on their local data for training and validation purposes.

2) Client Heterogeneity Initialization: To evaluate HandoffFL under realistic conditions, we incorporate client heterogeneity across two aspects:

Computational Heterogeneity: We simulate a diverse hardware ecosystem, ranging from resource-constrained mobile units ($0.5\times$ baseline speed) to high-performance edge servers ($2.0\times$ baseline speed). By dynamically scaling local training time based on specific compute speed factors, we accurately model environments containing IoT sensors, smartphones, and edge nodes.

Network Heterogeneity: Furthermore, we model a broad spectrum of network conditions by configuring different μ_R, simulating connectivity environments ranging from IoT Edge (5–15 Mbps), through 4G/WiFi 4 (10–30 Mbps) to 5G/WiFi 5 (20–100 Mbps). This heterogeneity-aware architecture ensures fair aggregation through staleness-aware weighting and prevents resource bottlenecks by intelligently scheduling layers.

To operationalize these heterogeneity factors, we implement a wall-clock time simulation. This mechanism advances global time incrementally as events are processed detailed in Section IV-F, utilizing the precise computation and communication latency models defined in Section IV-D.

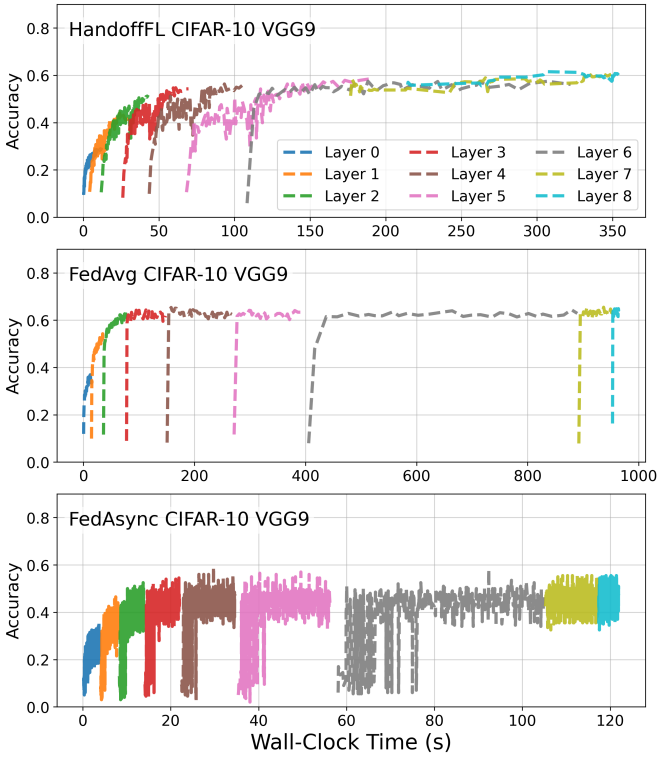

Fig. 2: Performance Comparison with SOTA FL Methods

3) Baselines: We benchmark HandoffFL against FedAvg [1] and FedAsync [8] by adapting their logic to our layer-wise architecture. Since our framework introduces a novel event-driven layer-wise training structure, we adapt the fundamental mechanisms of these baselines to fit this specific context while ensuring fair comparison. For FedAvg, we replicate a synchronous environment where the server has to wait for the last one and aggregates only after all selected clients complete the current round task. For FedAsync, we implement an immediate update strategy where the server performs weighted aggregation upon receiving any single client's updates, instantly triggering the subsequent training step. All other experimental configurations remain the same across methods when making a comparison.

4) Implementation: We implement our event-driven layer-wise Handoff FL using PyTorch 2.5 on Python 3.11. All experiments were conducted on a Windows workstation equipped with an Intel Core i9-13980HX CPU (2.20 GHz), 32GB of RAM, and an NVIDIA GeForce RTX 4080 GPU.

B. Performance Comparison with SOTA FL

To validate the effectiveness of HandoffFL, we benchmark it against two SOTA methods, specifically focusing on accuracy convergence and wall-clock time consumption. Fig. 2 illustrates the accuracy convergence trends for training VGG9 on the CIFAR-10 dataset, and lines of the same color represent the convergence trajectory of the same layer.

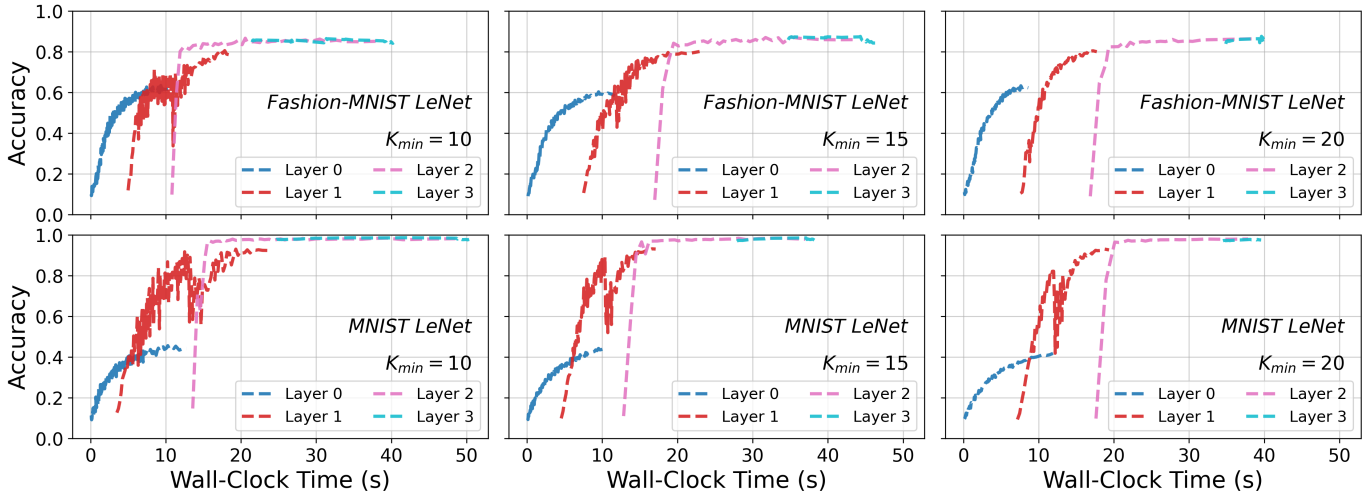

Fig. 3: Impact of Minimum Contributors on Performance

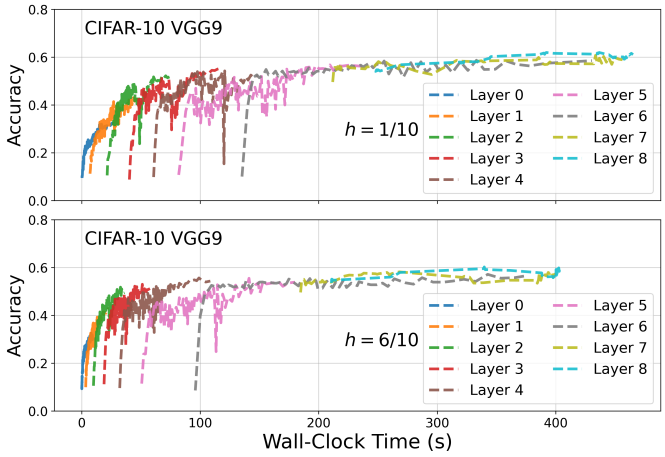

Fig. 4: Impact of Staleness Penalty on Performance

For the experimental setup, both HandoffFL and FedAvg are set to select 10 clients out of 50 as minimum contributors. Namely, in HandoffFL, aggregation is triggered immediately once the server collects updates from the first 10 clients, allowing for the immediate assignment of new tasks. In contrast, under the same configuration, FedAvg dictates that for every training round, the server must wait for the slowest client, i.e., the straggler to complete its task before aggregation and subsequent assignment can start. Regarding FedAsync, since the server updates the global model upon receiving a single client upload, the raw convergence plots exhibited excessive volatility; therefore, a 10-round moving average was applied to visualize the trends. Note that the convergence criterion was determined using Function 15 with δ set to 0.01.

As evident in Fig. 2, while both HandoffFL and FedAvg achieve comparable final model accuracy, HandoffFL significantly reduces the total training time from $975s$ (FedAvg) to $351s$, a reduction of approximately 64%. This result highlights a critical limitation of FedAvg that the synchronization bottleneck caused by waiting for stragglers severely hampers training efficiency in realistic heterogeneous scenarios.

Meanwhile, the comparison between HandoffFL and FedAsync reveals that although FedAsync is substantially faster in completing in $125s$ due to its lack of waiting mechanisms, it suffers from severe training instability. While the general trend suggests convergence, the final model accuracy for individual clients fluctuates drastically between 0.3 and 0.5. This outcome corroborates findings in existing literature, such as SEAFL [9], which indicate that FedAsync often fails to achieve stable convergence in many practical scenarios.

C. Effect of Parameter Setting

Configuring HandoffFL involves tuning key hyperparameters that dictate its performance. In this section, we examine the impact of the minimum aggregation contributors K_{min} and the staleness penalty h on the framework's effectiveness.

1) Impact of Minimum Contributors: Fig. 3 illustrates the performance of LeNet on the Fashion-MNIST and MNIST datasets under varying $K_{\min}$. The results from both datasets indicate a trade-off that while a higher $K_{\min}$ reduces convergence fluctuations within each layer, it significantly delays the initialization of subsequent layers. Specifically, taking Fashion-MNIST as an example, when $K_{\min} = 10$, the second, third, and final layers begin training at approximately $5s$, $11s$, and $21s$, respectively. In contrast, increasing $K_{\min}$ to 15 or 20 delays these start times substantially; for instance, the final layer initializes at $35.5s$ and $36s$, respectively, compared to $21s$ in the $K_{\min} = 10$ setting. Consequently, the third layer achieves convergence much earlier with a lower threshold ($12s$) compared to the higher thresholds (approximately $20s$). These findings demonstrate that enforcing a high $K_{\min}$ forces fast clients to idle while waiting for stragglers, thereby adversely affecting the overall training efficiency.

2) Impact of Staleness Penalty: Fig. 4 presents a performance comparison of VGG9 on the CIFAR-10 dataset under varying values of the half-life parameter h explained

TABLE I: Performance Comparison of Heterogeneous Clients: VGG9 on CIFAR-10

	Layer 0				Layer 1				Layer 2				Layer 3			
Client ID	*Start*	*End*	*Rounds*	*Acc*	*Start*	*End*	*Rounds*	*Acc*	*Start*	*End*	*Rounds*	*Acc*	*Start*	*End*	*Rounds*	*Acc*
client 37 (fast)	0.00052	6.40	30	0.33	6.66	14.35	31	0.39	14.65	26.93	32	0.52	27.48	50.08	43	0.62
client 43 (fast)	0.00089	3.95	22	0.26	4.12	8.57	21	0.35	8.82	21.27	25	0.43	21.85	41.33	28	0.56
client 18 (medium)	0.00187	8.63	26	0.28	9.28	19.39	24	0.41	19.93	42.29	31	0.54	43.38	60.57	16	0.56
client 32 (medium)	0.00139	4.02	18	0.32	4.29	13.69	31	0.44	14.14	24.98	21	0.46	25.64	41.20	20	0.53
client 5 (slow)	0.00363	11.57	15	0.26	12.13	30.07	18	0.42	30.31	43.22	10	0.48	43.67	70.16	13	0.56
client 23 (slow)	0.00384	12.06	16	0.28	12.13	30.08	19	0.36	30.33	43.24	10	0.54	43.70	70.19	12	0.51

	Layer 4				Layer 5				Layer 6				Layer 7				Layer 8			
Client ID	*Start*	*End*	*Rounds*	*Acc*	*Start*	*End*	*Rounds*	*Acc*	*Start*	*End*	*Rounds*	*Acc*	*Start*	*End*	*Rounds*	*Acc*	*Start*	*End*	*Rounds*	*Acc*
client 37 (fast)	50.85	66.75	19	0.62	67.64	84.33	13	0.59	87.28	159.65	14	0.63	164.85	213.19	19	0.66	214.23	216.54	9	0.65
client 43 (fast)	42.17	67.41	26	0.59	68.91	95.67	17	0.62	99.41	155.16	10	0.61	160.77	185.41	11	0.60	186.30	261.83	12	0.60
client 18 (medium)	62.82	90.05	16	0.61	92.82	156.12	20	0.57	164.27	254.36	10	0.57	264.95	304.09	11	0.62	305.64	333.26	9	0.62
client 32 (medium)	42.42	70.98	21	0.50	73.29	126.74	22	0.49	133.03	291.16	19	0.57	298.54	330.97	10	0.61	333.03	350.03	15	0.62
client 5 (slow)	70.88	106.19	11	0.64	107.91	191.13	14	0.62	198.50	315.49	7	0.60	329.33	347.73	6	0.62	349.08	353.20	5	0.61
client 23 (slow)	70.95	104.15	10	0.56	108.05	191.27	13	0.54	198.98	315.97	7	0.63	329.39	350.18	6	0.63	349.08	352.80	5	0.61

TABLE II: Client Heterogeneity Profile

Client ID	**Device Type**	**Compute Speed**	**Bandwidth**
Client 37	Fast	1.61×	54.78 Mbps
Client 43	Fast	2.11×	32.23 Mbps
Client 18	Medium	1.02×	15.35 Mbps
Client 32	Medium	1.50×	20.58 Mbps
Client 5	Slow	0.41×	7.89 Mbps
Client 23	Slow	0.43×	7.47 Mbps

in Section IV-E, which governs the exponential decay rate of the aggregation weights. A smaller h (e.g., 1.0) induces rapid decay, imposing a strict penalty on stale updates from stragglers. Conversely, a larger h (e.g., 8) results in slower decay, exhibiting greater tolerance towards staleness.

As illustrated in Fig. 4, in an experimental setting where 10 clients participate in staleness-weighted aggregation per round, setting $h = 1$ implies an aggressive decay strategy that a client merely one version behind the freshest update faces a 50% reduction in aggregation weight. The visual results demonstrate that, compared to a moderate baseline of $h = 6$, the model with $h = 1$ exhibits significantly higher fluctuation during the initial layers and requires a prolonged period to achieve convergence.

Specifically, timeline comparisons reveal that at the $100s$, the model with $h = 6$ has already initiated training on layer 6, whereas under $h = 1$, layer 6 training is delayed until after $130s$. Furthermore, the initialization of the final layer is delayed by approximately $50s$, resulting in a total training time increase of $65s$. These findings indicate that setting h too low imposes an excessive penalty on slow clients (stragglers), which negatively impacts both convergence stability and overall training efficiency.

D. Client Heterogeneity Influence

In real-world scenarios, clients serve as the primary entities for data collection and inference using local models. However, the adaptability of a client to the global model is constrained by its inherent heterogeneity, as detailed in Section IV-D. Consequently, analyzing performance at the individual client level is crucial. To address this, HandoffFL is configured to simulate realistic computational and communication heterogeneity. Table II profiles six representative clients, two selected randomly from each of the slow, medium, and fast types out of the 50 clients participating in the VGG9 on CIFAR-10 experiment. As described in Section V-A2, these clients are initialized with random attributes. To mirror realistic conditions, we deploy that clients with the highest bandwidth do not necessarily possess the highest computational resources.

Table I presents a layer-wise performance analysis of these selected clients, including metrics such as starting and ending wall-clock time (s), total aggregation rounds participated in, and maximum accuracy achieved. First, Table I demonstrates that under the HandoffFL framework, even stragglers can participate in a sufficient number of training rounds due to the asynchronous mechanism. Furthermore, by leveraging knowledge transferred and the handoff mechanism from faster clients, stragglers can maintain competitive accuracy even with fewer local training rounds. For instance, in layer 3, slow client 5 completes only 13 rounds in approximately $27s$, achieving a maximum accuracy of 0.56. This is only 0.06 lower than the highest accuracy achieved by client 43, who completes 43 rounds in roughly $23s$. Similarly, in layer 7, slow client 23 achieves an accuracy only 0.03 below the top performer despite participating in only 6 rounds. These results confirm that HandoffFL significantly enhances model refinement for stragglers.

Additionally, as the experiment utilizes an improvement rate-based convergence threshold, defined by Function 15 in Section IV-C, a distinct trend is observable in the table that selected clients require a high number of training rounds in the initial layers to meet the accuracy improvement threshold. However, the required number of rounds decreases significantly in subsequent layers. For example, client 37 drops from over 30 rounds in the early layers to approximately 15 rounds starting from layer 4. This represents a novel finding of HandoffFL. Unlike most existing round-based research that employs a fixed number of rounds, HandoffFL adapts to the network architecture. As deeper layers typically demand greater computational and communication resources, the observed reduction in rounds for deeper layers suggests that

HandoffFL effectively mitigates the computational and communication burden associated with training deep networks.

VI. Conclusion

In this paper, we introduce HandoffFL, an event-driven asynchronous layer-wise FL framework built on a real-time message-passing architecture to mitigate the straggler problem in heterogeneous environments. By utilizing a continuous handoff mechanism for task assignment and aggregation, HandoffFL allows the server to efficiently leverage computational resources from clients with varying capabilities without being bottlenecked by the slowest devices. Extensive experiments on the CIFAR-10 dataset using the VGG9 model demonstrate the superiority of our approach. HandoffFL reduces the total training time by approximately 64% compared to FedAvg while maintaining comparable accuracy. Furthermore, unlike FedAsync, which suffers from severe instability and accuracy fluctuations, HandoffFL achieves stable convergence by effectively managing client contributions through the K_{min} threshold. Our layer-wise analysis reveals that HandoffFL is particularly effective in deep networks where computational demands vary significantly across layers. In future work, we aim to investigate the integration of differential privacy (DP) mechanisms into HandoffFL to further enhance data security in asynchronous settings.

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 *Artificial intelligence and statistics*, pp. 1273–1282, PMLR, 2017.

[2] W. Zhang, X. Liu, C. Zhu, S. Varjonen, F. Wang, and S. Tarkoma, "Federated learning meets urban opportunistic crowdsensing in 6g networks: Opportunities, challenges, and optimization potentials," *IEEE Network*, vol. 39, no. 2, pp. 36–43, 2025.

[3] M. J. Sheller, G. A. Reina, B. Edwards, J. Martin, and S. Bakas, "Multi-institutional deep learning modeling without sharing patient data: A feasibility study on brain tumor segmentation," in *International MICCAI Brainlesion Workshop*, pp. 92–104, Springer, 2018.

[4] M. Liu, S. Ho, M. Wang, L. Gao, Y. Jin, and H. Zhang, "Federated learning meets natural language processing: A survey," *arXiv preprint arXiv:2107.12603*, 2021.

[5] X. Ouyang, Z. Xie, J. Zhou, J. Huang, and G. Xing, "Clusterfl: a similarity-aware federated learning system for human activity recognition," in *Proceedings of the 19th annual international conference on mobile systems, applications, and services*, pp. 54–66, 2021.

[6] W. Zhang, X. Liu, R. Zhang, C. Zhu, and S. Tarkoma, "Fedvisual: Heterogeneity-aware model aggregation for federated learning in visual-based vehicular crowdsensing," *IEEE Internet of Things Journal*, vol. 11, no. 22, pp. 36191–36202, 2024.

[7] J. Konečnỳ, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon, "Federated learning: Strategies for improving communication efficiency," *arXiv preprint arXiv:1610.05492*, 2016.

[8] C. Xie, S. Koyejo, and I. Gupta, "Asynchronous federated optimization," *arXiv preprint arXiv:1903.03934*, 2019.

[9] M. S. Islam, S. Panta, F. Xu, X. Yuan, L. Chen, and N.-F. Tzeng, "Seafl: Enhancing efficiency in semi-asynchronous federated learning through adaptive aggregation and selective training," 2025.

[10] J. Nguyen, K. Malik, H. Zhan, A. Yousefpour, M. Rabbat, M. Malek, and D. Huba, "Federated learning with buffered asynchronous aggregation," 2022.

[11] J. Liu, J. Jia, T. Che, C. Huo, J. Ren, Y. Zhou, H. Dai, and D. Dou, "Fedasmu: Efficient asynchronous federated learning with dynamic staleness-aware model update," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, pp. 13900–13908, 2024.

[12] F. Mo, H. Haddadi, K. Katevas, E. Marin, D. Perino, and N. Kourtellis, "Ppfl: privacy-preserving federated learning with trusted execution environments," in *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '21, (New York, NY, USA), p. 94–108, Association for Computing Machinery, 2021.

[13] W. Chen, J. Zhang, and D. Zhang, "Fl-joint: joint aligning features and labels in federated learning for data heterogeneity," *Complex & Intelligent Systems*, vol. 11, no. 1, p. 52, 2025.

[14] J. Liu, W. Gong, Z. Fang, J. Gonzàlez, and J. Rodrigues, "Personalized federated learning with progressive local training strategy and lightweight classifier.," *Applied Sciences (2076-3417)*, vol. 15, no. 5, 2025.

[15] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural computation*, vol. 18, no. 7, pp. 1527–1554, 2006.

[16] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, "Greedy layer-wise training of deep networks," *Advances in neural information processing systems*, vol. 19, 2006.

[17] L. Wang, C.-Y. Lee, Z. Tu, and S. Lazebnik, "Training deeper convolutional networks with deep supervision. arxiv 2015," *arXiv preprint arXiv:1505.02496*, 2015.

[18] M. Jaderberg, W. M. Czarnecki, S. Osindero, O. Vinyals, A. Graves, D. Silver, and K. Kavukcuoglu, "Decoupled neural interfaces using synthetic gradients," in *International conference on machine learning*, pp. 1627–1635, PMLR, 2017.

[19] A. Nøkland and L. H. Eidnes, "Training neural networks with local error signals," in *International conference on machine learning*, pp. 4839–4850, PMLR, 2019.

[20] M. Chen, B. Mao, and T. Ma, "Fedsa: A staleness-aware asynchronous federated learning algorithm with non-iid data," *Future Generation Computer Systems*, vol. 120, pp. 1–12, 2021.

[21] T. Nishio and R. Yonetani, "Client selection for federated learning with heterogeneous resources in mobile edge," in *ICC 2019-2019 IEEE international conference on communications (ICC)*, pp. 1–7, IEEE, 2019.

[22] M. Lin, Q. Chen, and S. Yan, "Network in network," *arXiv preprint arXiv:1312.4400*, 2013.

[23] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9, 2015.

[24] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[25] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," *arXiv preprint arXiv:1708.07747*, 2017.

[26] A. Krizhevsky, "Learning multiple layers of features from tiny images," 2009.

[27] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[28] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.