PASTA: Training Acceleration for Vertical Federated Learning via Adaptive Pipeline Parallelism

Tian Wu*, Weijie Liu†, Han Liang*, Ziwei Zhan*, Jingpu Duan‡, Chuan Wu†,
Jinhang Zuo§, Xu Chen*, Xiaoxi Zhang*¶
*Sun Yat-sen University †The University of Hong Kong ‡Peng Cheng Laboratory §City University of Hong Kong
Email: *{wutian,liangh68,zhanzw}@mail2.sysu.edu.cn, †liuwj0817@connect.hku.hk, ‡duanjp@pcl.ac.cn
†cwu@cs.hku.hk, §jinhang.zuo@cityu.edu.hk, *{chenxu35,zhangxx89}@mail.sysu.edu.cn

***Abstract*—Vertical federated learning (VFL) enables collaborative model training among geo-distributed participants, each with different features of the same samples, but only one party possesses the labels. Communication delays between active and passive parties in VFL significantly hinder its training efficiency. Existing VFL methods adopt asynchronous schemes or multiple local updates per communication round, but they either introduce heavy computation overhead or fail to adapt to dynamic network conditions. This work proposes *PASTA*, a novel framework employing Adaptive Pipeline Parallelism with Staleness Control for VFL, designed to mitigate these delays and balance training efficiency and model performance. *PASTA* enables concurrent communication and computation, maximizing resource utilization and minimizing idle time by strategically using stale gradients. Each passive party can send one or more batches of embeddings per communication and conduct stale local training, so that computation times can overlap with communication latency. Since staleness impedes model accuracy despite its benefits in reducing time, a dynamic feedback-based mechanism is proposed to adjust the numbers of embeddings sent and local training iterations based on system heterogeneity. Extensive experiments across various datasets demonstrate that *PASTA* significantly enhances convergence speed by $1.8\times$ to $4.6\times$ compared to leading VFL systems, without compromising final accuracy. The source code is available at https://github.com/PointerA/PASTA.**

I. Introduction

Vertical Federated Learning (VFL) [1], [2] enables multiple parties to build machine learning models collaboratively with privacy preservation. Unlike horizontal federated learning (HFL) [3]–[6] where parties possess different training samples within the same feature space, parties in VFL hold disparate and complementary features of the same training samples. The labels are typically owned by a single party, often referred to as the *active party*, while the others are termed *passive parties*. VFL can therefore enhance cooperation among diverse parties from different industries by facilitating the exchange of domain knowledge without exposing their private data. This characteristic of VFL has endowed it with widespread attention in many privacy-sensitive domains, including collaborative healthcare research, risk assessment in finance, and inter-organizational data collaboration [7].

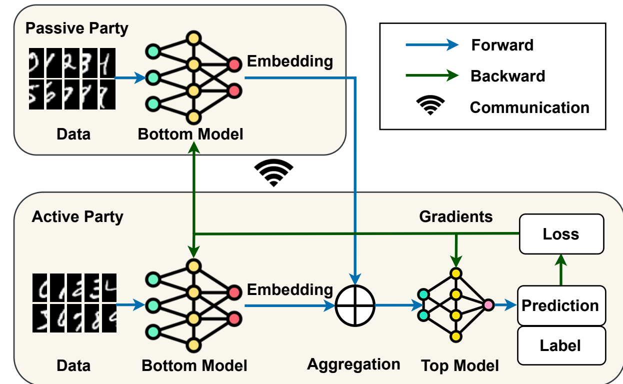

Fig. 1. Each batch of VFL training requires two communications: a passive party passes its embedding to the active party, who then sends back its gradient to the passive party.

Despite its advantages, different VFL parties have to exchange both their forward activations and backward gradients, resulting in more frequent inter-party communication and potentially longer training time compared to HFL. Besides, different parties are often geo-distributed, operating on heterogeneous network infrastructures and typically communicating through Wide-Area Network connections [8]–[10], as shown in Figure 1. This often leads to limited transmission bandwidth and higher network latency during the communication process, significantly reducing VFL training efficiency. One major bottleneck is propagation delay, which can range from 100 ms to 10 s in VFL due to long-distance transmission between geo-distributed parties. While compression and quantization [11] can reduce the amount of transmitted data to mitigate bandwidth-related delay, they do not address propagation delay, which is determined by the transmission medium and physical distance. These communication inefficiencies, there-

This work was supported by NSFC grant (62472460), Guangdong Basic and Applied Basic Research Foundation (2024A1515010161, 2023A1515012982), Young Outstanding Award under the Zhujiang Talent Plan of Guangdong Province, CityUHK 9610706, the PCL-CMCC Foundation for Science and Innovation under Grant 2024ZY1A0010, Hong Kong RGC under the contracts 17204423 (GRF), C7004-22G (CRF), C5032-23G (CRF), T43-513/23-N (TRS) and CRS_PolyU501/23.

¶Corresponding author: Xiaoxi Zhang.

979-8-3315-4940-4/25/$31.00 © 2025 IEEE

fore, restrict the broader implementation of VFL systems. To improve its communication efficiency, various research has been conducted. Some approaches focus on reducing the number of communication rounds. For example, FedBCD [12] performs multiple local model updates with stale gradients received from the active party. FlexVFL [13] and AdaVFL [14] further analyze the relationship between model convergence and the number of local updates, assigning various numbers of local updates to different passive parties based on their computation and communication capacities. Other works like C-VFL [11] incorporate gradient compression in order to directly reduce the amount of information exchanged, thereby decreasing the transmission time.

However, excessive use of the stale gradients in previous rounds may hinder the model's convergence toward the optimal solution. While existing approaches can reduce communication costs, they overlook *the interplay between the staleness of intermediate training statistics and communication overhead*, which is the focus of this paper and can be exploited to accelerate VFL training. Other novel VFL paradigms like asynchronous VFL [15] and one-shot VFL [16] also address latency issues but their applicability is often limited due to degraded model performance compared to synchronous VFL. This work proposes a new synchronous VFL training system, driven by a key observation: *participants often remain idle and are unable to effectively utilize resources due to inevitable communication delay in synchronous training among geo-distributed VFL parties*. *Our first idea* is to pipeline VFL by enabling additional local model computation executed in parallel with communication. Participants can perform more local updates using stale gradients or conduct forward propagation for the next mini-batch of training data. The communication time is therefore effectively utilized, improving VFL's training efficiency. In the meanwhile, since utilizing stale results inevitably impedes model accuracy, *our second idea* is to simultaneously control staleness along with pipeline training, in response to uncertain and dynamic network environments.

Although pipeline training has been common [17], [18], its implementation in VFL is complex due to the following **challenges**. First, *the latency-staleness tradeoff* requires balancing. To reduce idle time by leveraging pre-computation as in our aforementioned first idea, passive parties can conduct local backward updates using stale gradients and forward computation of future mini-batches using stale models. However, using outdated model gradients or parameters significantly undermines model accuracy [13]. Second, *system heterogeneity and network dynamics* present further complications in pipelining different stages of VFL training. Varying, uncertain capacities of communication and computation among parties can lead to inefficient workload distribution and increased delays, especially without real-time adjustments. Prior works such as FlexVFL [13] and AdaVFL [14] address system heterogeneity by pre-setting time limits of each round of local training at all VFL parties. This strategy fails to account for the varying training stages within each party and lacks adaptability to dynamic network conditions. To address the above challenges, we make the following technical contributions:

- *Novel pipeline parallelism framework for VFL training (Section III-A).* We integrate an adaptive synchronous VFL training framework, by pipelining stages of forward and backward propagations at different parties and overlaps embedding pre-computation of mini-batches with the extended communication time. To the best of our knowledge, this work presents the first attempt to implement pipeline parallelism in VFL, leveraging the nature of synchronous VFL training paradigm and improving its training efficiency.
- *Adaptive, feedback-based control of embedding pre-computation and stale local training iterations (Sections III-B and III-C).* Through experiments, we identify that different epochs of VFL training vary in sensitivity to staleness (see Fig. 3). We then design a novel epoch-specific staleness control mechanism to balance communication reduction and model accuracy preservation. Unlike methods such as CELU [9], which involve heavy computation to control staleness (e.g., using cosine similarities), *PASTA* operates more efficiently. It only requires the current number of training epochs as input and reduces *staleness* over time using a pre-defined metric. This metric integrates both pre-computed embeddings and stale gradients into local training, guided by a pre-set function. We finally employ real-time adjustments to refine the numbers of local training iterations and pre-computation batches, catering to varying, uncertain computation and communication capabilities among VFL parties.
- *Theoretical analysis and extensive experiments (Sections IV and V).* Through mathematical analysis, we demonstrate that our algorithm *PASTA* effectively reduces the idle time in VFL training. Besides, we compare *PASTA* with two state-of-the-art VFL paradigms using various datasets and models. Comprehensive experimental results verify that *PASTA* significantly reduces the training time required to achieve desired test accuracies. In the scenarios with high communication latencies, it achieves $1.8\times$ to $4.6\times$ training speedup on different datasets, exceeding leading VFL mechanisms.

II. Preliminaries and Related works

A. Vertical Federated Learning (VFL)

VFL [19] enables collaborative deep neural network (DNN) training between distributed clients using partitioned feature data. The objective of training is to minimize the average loss.

$$\min_{\Theta} \ell(\Theta; \mathcal{D}) \triangleq \frac{1}{N}\sum_{i=1}^{N} f(\Theta; \mathbf{x}_i, y_i), \tag{1}$$

where $\mathbf{x}_i \in \mathbb{R}^{1\times d}$ and y_i represent the feature vector and label of a single training sample $\mathcal{D}_i$ from the overall dataset $\mathcal{D}$, with N samples. Let $\mathcal{K}$ denote the set of K participating parties. Each party k in VFL possesses a portion of the feature vector of each data sample: $\{\mathbf{x}_i^k \in \mathbb{R}^{1\times d_k}\}_{i=1}^N$, where $\mathbf{x}_i^k$ is a sub feature vector of $\mathbf{x}_i$, with d_k dimensions. We define the K_{th} client as the *active party*, the only client that maintains the labels of all samples, whose dataset is defined as $\mathcal{D}^K \triangleq \{\mathbf{x}_i^K, y_i\}_{i=1}^N$. The other clients maintain no

labels and are called *passive parties*, each of which has a dataset $\mathcal{D}^k \triangleq \{\mathbf{x}_i^k\}_{i=1}^N$. Let $\Theta \triangleq \{\theta_1, \theta_2, \cdots, \theta_K, \theta_t\}$ denote the parameters of the DNN model in VFL, composed of local parameters θ_k of the *bottom model* at each VFL client $k, \forall k = 1, 2, \cdots, K$, and parameters θ_t of the *top model* exclusively owned by the active party K. We define $\mathcal{M}_k(\cdot, \theta_k)$ to denote the bottom model from party k and $\mathcal{M}_t(\cdot, \theta_t)$ to represent the top model exclusively owned by party K. Each bottom model serves as a feature extractor, learning characteristics of local features and providing embeddings for the top model at the active party. The top model utilizes embeddings from all passive parties and labels at the active party for predictions and loss calculation. Given a task loss function $\mathcal{L}$, e.g. cross-entropy loss, and $h_i^k \triangleq \mathcal{M}_k(\mathbf{x}_i^k, \theta_k)$, the loss function $f(\Theta; \mathbf{x}_i, y_i)$ in Eq. (1) is defined as:

$$f(\Theta; \mathbf{x}_i, y_i) = \mathcal{L}\left(\mathcal{M}_t\left(h_i^1, \cdots, h_i^K, \theta_t\right), y_i\right). \tag{2}$$

Workflow of vanilla VFL. Fig. 1 presents the training process of classic VFL [19]. First, each party k computes the output of the bottom model $h_k = \mathcal{M}_k(\mathbf{x}_b^k, \theta_k)$ using a mini-batch b of training samples $\mathbf{x}_b^k$ and sends it to the active party. Here, we omit the subscript b in h_k for simplicity, as it is always associated with the current mini-batch. In each iteration, the active party aggregates the received local model output $\{h_k\}_{k=1}^K$, which are the input of the top model $\mathcal{M}_t$, and calculates the training loss based on Eq. (1)–(2). Next, the active party updates the top model parameter θ_t by computing the gradient $\frac{\partial \ell}{\partial \theta_t}$ using loss from Eq. (1), computes the gradients $\{\frac{\partial \ell}{\partial h_k}\}_{k=1}^K$, and sends them back to each corresponding party k. Each party k then updates its bottom model parameter θ_k by computing gradients $\nabla_{\theta_k}\ell$:

$$\nabla_{\theta_k}\ell = \frac{\partial \ell}{\partial h_k} \cdot \frac{\partial h_k}{\partial \theta_k}. \tag{3}$$

The above procedure is repeated until the model converges.

B. Communication Mitigation in VFL

We categorize four types of related works aimed at improving VFL's communication efficiency.

Multiple local updates. FedBCD [12] allows passive parties to perform multiple local updates before each communication, defined as a round, based on Eq.(3) on the same mini-batch, reusing the stale gradient results $\frac{\partial \ell}{\partial h_k}$ computed by the active party in the previous round. FlexVFL [13] and AdaVFL [14] introduce a maximum time limit for every VFL round across all participating parties, and allow different party-specific numbers of local model updates based on their heterogeneous computing and communication capacities. While these methods do not reduce the cost per individual communication, they effectively decrease the total training communication time by leveraging multiple local updates to substantially diminish the overall number of required communication rounds. However, these works do not utilize pipeline parallelism and do not consider heterogeneous communication capabilities cross parties, lacking the agility to adapt to dynamic network environments.

Asynchronous VFL. With an asynchronous VFL design like VAFL [15] and FDML [20], the active party immediately performs top model updates and gradient computations upon receiving outputs from some passive parties (instead of all). This results in additional computation at the active party, as it performs more top model updates and gradient computation after receiving embeddings from subsets of passive parties. Also, most passive parties may not be able to obtain the latest gradients from the top model, hindering model convergence [21]. Further, existing asynchronous VFL methods [22]–[27] are mainly designed for Logistic Regression (LR) and Gradient Boosting Decision Trees rather than DNNs. We focus on synchronous VFL for DNN in this paper.

Compression. C-VFL [11] and Khan et al. [28] generate compressed representations from the training data using PCA (Principal Component Analysis) and auto-encoder modules. While they directly reduce the amount of data exchanged among parties, communication time is still long in case of high network latency. Besides, other compression techniques for VFL, e.g. AVFL [29] and SecureBoost [30] [31], are only applicable to LR and XGBoost algorithms. We consider these communication data compression methods orthogonal, which can be readily integrated into our framework.

One-shot and few-shot VFL. FedOnce [16] is a one-shot VFL algorithm that integrates an unsupervised learning method NAT (Noise As Targets) into VFL. Sun et al. [32] use semi-supervised learning during local training in one-shot/few-shot VFL, which enables passive parties to gain sufficient information through minimum communication rounds with the active party. Unfortunately, such unsupervised or self-supervised learning techniques may lead to degraded performance when communication is encrypted (common in practical VFL training environments). Our design works effectively in scenarios with or without encrypted communication, and our objective goals are not for one-/few-shot VFL.

C. Pipeline Parallelism

Pipeline parallelism has been widely adopted for DNN training in centralized clusters, that splits each mini-batch into smaller micro-batches and processes them in a pipelining manner across different model stages on different devices (e.g., Google's GPipe [17]). PipeDream [18] advocates 1F1B (one forward pass followed by one backward pass) pipeline training, and a weight stashing scheme to ensure the same version of parameters are used for both forward and backward passes for a given mini-batch. PipeDream-2BW [33] further reduces memory occupation with double-buffered weights by only storing two versions of parameters at each worker. PipeDream-Flush [33] only maintains a single version of model weights and introduces regular pipeline flush to ensure consistent model parameter versions, leading to lower memory occupation but lower training throughput.

To the best of our knowledge, no existing VFL systems for deep learning model training have implemented pipeline parallelism. In VFL, all parties maintain bottom models, while only the active party holds labels and the top model, introducing both data parallelism and model parallelism. The main bottleneck in VFL is the transmission of embeddings and

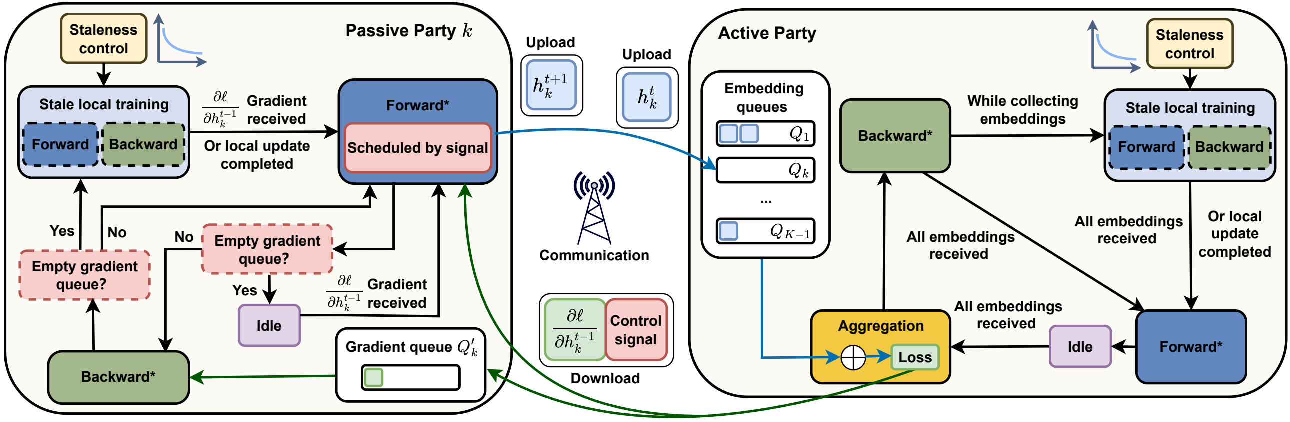

Fig. 2. The workflow of *PASTA*. Forward*, Backward*, Stale local training (consisting of forward and backward propagations for a certain number of iterations), and Aggregation of received embeddings are four main procedures of different parties.

gradients between geographically distributed parties. Traditional pipeline parallelism focuses on multi-GPU setups within the same cluster, requiring centralized control for profiling and batch management, which is incompatible with VFL's privacy-preserving requirements that prohibit such centralization. Instead, we can only rely on communication feedback such as received massage count to monitor the training process without introducing additional overhead. We also need to address the unique challenges of variable communication and computation that are not considered in conventional implementations.

III. System Design of *PASTA*

As introduced in our key observation (Section I), VFL participants typically collaborate by exchanging embeddings and gradients, which may lead to substantial idle waiting time due to potentially long communication latency, especially in scenarios where network latency is high or communication channels are unreliable. This idle time can significantly affect overall training efficiency. To address this, we introduce *PASTA*, a pipeline parallelism approach that focuses on reducing idle time periods, referred to as *time bubbles* between participants receiving messages via the idea of overlapping the processes of communication with computation. By pipelining the processes of local model updates, embedding uploading, and gradient broadcasting, across different parties, through lightweight interaction information as assistance, our algorithm can significantly reduce the overall training latency, thereby improving training efficiency. The design of *PASTA* consists of three main parts:

1) **Pipeline parallelism VFL.** Our basic idea is to enable pre-computation at VFL parties that are waiting for intermediate results, so that the computation and communication time periods can overlap. The specific strategies designed for passive parties and active party are elaborated in Section III-A respectively.
2) **A dynamic staleness decreasing method** is designed driven by our observation: repeatedly using outdated gradients may affect the convergence of training in the later stages more severely. Therefore, we propose to decrease the *staleness*, a metric designed as the product of the number of pre-computed embeddings and the number of stale local training iterations in each round.
3) **A feedback-based mechanism** is proposed to adaptively determine the number of embeddings of different batches to send each round, according to prevailing feedback from the active party and the staleness limit, better accommodating dynamic network status.

A. Pipeline Parallelism for VFL

By allowing passive parties to send several embedding of subsequent batches, we can overlap communication and computation to reduce time bubbles. As shown in Fig. 2, each passive party undergoes three primary phases: forward computation, local training, and backward updates. The active party additionally incorporates an aggregation step.

1) The action of each passive party k: In conventional VFL training, a single batch of embedding h_k^t is sent after the model update is completed. But considering the potentially long latency due to transmitting h_k^t to the active party and waiting for the new gradients received, *PASTA* enables pre-computation at passive parties so as to reduce its idle time. Starting from the Stale local training procedure shown in Fig. 2, two cases lead to stage transitions:
Case I. Gradient $\frac{\partial l}{\partial h_k^{t-1}}$ is received from the active party and stored into the gradient queue Q_k of passive party k, who hasn't computed h_k^t yet. Suppose that the previously decided number of in-flight embeddings, i.e., the expected number of batches of which the embeddings are sent but the corresponding gradients are not used, is one ($B_k^{t-1} = 1$). Currently, u_k, defined as the currently actual number of batches, of which the embeddings are sent but the corresponding gradients are not used, is one. Then, passive party k transitions to the Forward* procedure and checks the control signal, denoted by C_k^{t-1}, which is in the same packet of $\partial l/\partial h_k^{t-1}$. The value of C_k^{t-1} can be -1, 0, or 1. Based on the value of C_k^{t-1}, B_k^t is updated, and then the passive party k will send a number of $B_k^t - u_k + 1$

embeddings. For instance, it computes and sends h_k^t and h_k^{t+1} if $C_k^{t-1}=1$, meaning that it increases B_k^t to be 2, and it computes and sends h_k^t solely if $C_k^{t-1}=0$, meaning that it sticks with the current setting so that $B_k^t=B_k^{t-1}$.[1] Then, party k transitions to Backward*, using $\frac{\partial l}{\partial h_k^{t-1}}$ in Q_k to do one-iteration of backward propagation at the bottom model and update the model parameters accordingly. It discards $\frac{\partial l}{\partial h_k^{t-1}}$ from the gradient queue once it is used. Then, if the queue is still not empty, meaning that at least one newer gradient $\frac{\partial l}{\partial h_k^{t}}$ has also arrived into the queue from the active party, the passive party transitions to Forward* again and computes embedding(s) based on the control signal associated with $\frac{\partial l}{\partial h_k^{t}}$.

Case II. The passive party k hasn't received $\frac{\partial l}{\partial h_k^{t-1}}$ but has reached the maximum allowed iterations of Stale local training, e.g., using $\frac{\partial l}{\partial h_k^{t-2}}$. It also transitions to Forward* and decides the number of new embeddings to send according to $B_k^t-u_k+1=1-1+1=1$. After that, if $\frac{\partial l}{\partial h_k^{t-1}}$ still hasn't arrived yet, passive party k enters an idle time if Q_k is empty or proceeds to Backward* otherwise.

2) The action of the active party.: When the embeddings $\{h_k^{t-1}\}$ from all passive parties have arrived, the active party calculates its own embedding using its bottom model. It then aggregates all embeddings and calculates the corresponding gradients based on its labels. After sending them to passive parties, the active party updates its bottom model and top model. Before starting round the next round, the active party trains locally until all passive parties' embeddings are collected. The active party calculates the intermediate gradients and the control signals for the batches to be sent based on Algorithm 2. It then packages these results and send to the passive party. Finally, the active party updates the model parameters based on Eq. (3) and the number of local training times for the next round.

In summary, *PASTA* distinguishes itself from the conventional VFL by promptly transmitting a number of embeddings adaptively, thereby reducing the idle time at passive parties and the time for the active party to receive embeddings, compared to strategies without pre-computation. The numbers of embeddings sent each time and stale local training iterations are directly linked to the degree of system heterogeneity, which will be elaborated in Sections III-B and III-C.

B. Staleness Reduction Strategy

Motivated by the observation that using stale gradients potentially increases the required number of iterations/rounds or compromises final accuracy, we aim to design a strategy that can balance communication reduction and model performance. To ground our research, we first examine FedBCD [12], a classic algorithm that performs stale local training multiple times. As shown in Fig. 3, unlike FedSGD, a vanilla VFL method that allows per-iteration communication between passive and active parties, FedBCD exhibits pronounced loss

[1]As shown in Algorithm 1, more detailed determination of B_k^t given C_k^t is produced by CORRDINATOR($\cdot$) (Algorithm 2) and elaborated in Section III-C.

Algorithm 1: Passive Party Training in VFL

Input : Total number of communication rounds T, gradient queues $\{Q'_k\}$, training datasets $\{\mathcal{D}_k\}$
Output: Bottom model parameters $\{\theta_k^*\}$
Initialization: Initialize bottom model parameters $\{\theta_k\}$, max training time $\{\bar{l}_k^1\}$, the ideal number $\{B_k^1\}$ and the current number $\{u_k\}$ of in-flight embeddings

```
1  foreach party k ∈ K in parallel do
2      for t = 1 to T do
3          Sample private data i from D_k;
4          Forward*: compute embedding
             h_k^t = M_k(x_i^k, θ_k);
5          Send h_k^t and update u_k = u_k + 1;
6          while u_k > B_k^t do
7              Backward*, updating model θ_k^* using (3);
8              Use a gradient from Q'_k and u_k = u_k − 1;
9              while Waiting for gradient do
10                 Local training: using the latest stale
                     gradient dequeued from Q'_k;
11                 if Local training time > l̄_k^t then
12                     break
13         Update B_k^{t+1}, l̄_k^{t+1} by Coordinator(k, C_k^t);
14 return {θ_k^*} = {θ_k}
```

oscillation. More specifically, the oscillatory behavior becomes severe from the middle stage (5 <epoch <15) and becomes more pronounced in the later stage (epoch >15). Changing the learning rate doesn't significantly mitigate either. We then hypothesis that, in early training phases, while gradient directions may stagnate, they generally remain correct; however, as training progresses, stagnant gradients are more likely to miss optimal points. This can trap the model in local minima, particularly with excessive local updates, potentially hindering optimal performance. This issue is crucial when employing multiple local updates per batch to reduce latency.

Therefore, in order to mitigate the negative impact of staleness, we define a metric for quantifying staleness, s_e, to be the *product* of $\bar{l}_k^t$, the number stale local training iterations using the same gradient received from the active party, and B_k^t, the number of batches sent in each communication with the active party, since these two factors both contribute to stale gradients or embeddings. Driven by our observation shown in Fig. 3, we decrease s_e as more epochs of training proceed. Here, an epoch is one complete pass through the entire training dataset. The design of our staleness decay function is:

$$s_{e+1}=\frac{s_{max}}{\sqrt{e}}, \tag{4}$$

where e is the number of epochs finished, s_{e+1} is the staleness in epoch $e+1$, and s_{max} is an initial hyper-parameter controls the number of local model updates in the first round, which depends on different machine learning tasks. By determining

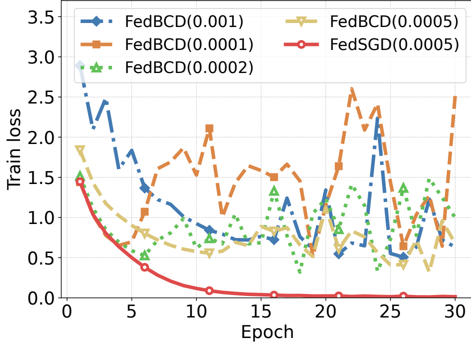

Fig. 3. The effects of using stale gradients. The loss curve of FedBCD initially declines but exhibits oscillations in later stages. Even after adjusting its learning rate (shown in parentheses), no improvement is observed. In contrast, FedSGD, which avoids stale gradients, demonstrates stable performance.

s_e, we can determine $\bar{l}_k^t$ once we set the value for B_k^t, which will be shown in Section III-C. After maintaining reasonable control over the parameter $\bar{l}_k^t$, the adverse effects of excessive stale updates on the convergence can be effectively mitigated.

C. A Feedback-based Mechanism to Decide B_k^t and $\bar{l}_k^t$

Given our pipeline parallelism framework and the staleness control strategy, the remaining details of our algorithm are to dynamically decide the actual value of B_k^t, the ideal number of in-flight embeddings, and $\bar{l}_k^t$, the number of iterations for stale local training, in round t. Choosing a static, effective B_k^t is challenging due to the uncertain and varying computational and communication capabilities of participants. A static B_k^t cannot adapt to dynamic changes in the training process, potentially making training either too slow if B_k^t is too small or inaccurate if B_k^t is too large. Moreover, a large B_k^t does not guarantee faster training: The increasing number of embeddings received from the passive parties can potentially introduce bottleneck at the active party with mediocre computing capacity. Conversely, passive parties can also become the bottleneck if the required forward or gradient computations exceed their computing capacity. Increasing B_k^t in both cases leads to zero speedup and degraded model performance.

Now, we introduce how to adjust B_k^t according to both the active party and the passive parties' feedback, which reflects their communication and computation statuses. The following descriptions correspond to Algorithm 2.

The active party. We adjust B_k^{t+1} based on B_k^t and our control signal C_k^t (lines 15 – 18 of Algorithm 2). If two or more embeddings are received, the active party signals the passive party with $C_k = -1$ to stop sending embeddings (lines 6, 7). Conversely, if it's computational capacity is available but hasn't received expected embeddings, it sends $C_k = 1$ with the gradient to prompt the passive party (lines 4, 5).

Each passive party. If the passive party doesn't receive a request to change B_k^t from the active party, it sets $B_k^{t+1} = B_k^t$. If instructed by C_k^t, it assesses whether to adjust batches accordingly. If $C_k^t = 1$, there are two cases: (i) party k is unable

Algorithm 2: Coordinator

```
   Initialization: calling party k, embedding queue
                   {Q_k}, control signal {C_k^t} = {0},
                   s_max, B_max
1  // If k = K (active party)
2  if active party is idle then
3      for k = 1 to K − 1 do
4          if embedding not received from party k then
5              C_k^t = 1;
6          if two or more embeddings from k are
             received then
7              C_k^t = −1;
8  if active party is not idle and begins to aggregate then
9      for k = 1 to K − 1 do
10         if two or more embeddings from k are
             received then
11             C_k^t = −1;
12 Active party sends signal C_k^t and gradient to party k;
13 if party k ≠ K receives C_k^t from the active party then
14     if C_k^t = 1 and party k executes the local training
         of round t − 1 then
15         B_k^{t+1} = min(B_k^t + C_k^t, B_max)
16     if C_k^t = −1 then
17         B_k^{t+1} = max(B_k^t + C_k^t, 1)
18 else
19     B_k^{t+1} = B_k^t
20 Party k updates l̄_k^{t+1} based on Eq. (4) and Eq. (5);
21 return
```

to process additional batches due to local limitations meaning it does not pass the condition in line 14, it applies $B_k^{t+1} = B_k^t$; (ii) party k is able to handle additional batches computation, indicated by executing local training at the previous round, B_k^{t+1} may increase (lines 14, 15 of Algorithm 2). Conversely, if the active party suggests $C_k^t = -1$, B_k^{t+1} is set as the maximum between $B_k^t - 1$ and 1 (line 17).

Adjusting $\bar{l}_k^t$. Since the staleness measured by s_e is a product of B_k^t and $\bar{l}_k^t$ for each k, t, and we have decided B_k^t and s_e ($B_K^t \equiv 1$), we then set $\bar{l}_k^t$ according to:

$$\bar{l}_k^t = \frac{s_e}{B_k^t}, \forall k = 1, 2, \cdots, K \tag{5}$$

In summary, *PASTA* dynamically adjusts embedding pre-computation and the number of local updates using stale gradients. We will provide analytical results in the next section.

IV. Performance Analysis

In this section, we compare our *PASTA* algorithm with FedBCD [12], a state-of-the-art synchronous VFL system that utilizes multiple local updates per round for accelerated training, compared to vanilla VFL. We choose FedBCD as

our baseline for theoretical performance analysis because its explicit use of stale gradients aligns closely with our research focus. We prove that *PASTA* can effectively reduce the total training time compared to FedBCD, by minimizing waiting time caused by communication delays among VFL parties. Communication round t begins when a party computes embeddings for aggregation and ends upon completing local training with round t's gradient. R denotes the total rounds.

Definition 1 (Maximum Time Interval for VFL)**.** The maximum time interval spans from when the active party sends gradients until it collects all embeddings in the next round, as exemplified by FedBCD's interval definition:

$$\mathcal{G}^t = \max_k \Big(T_{\mathrm{s2c},k}^{t-1} + T_{\mathrm{c2s},k}^{t} + T_{\mathrm{b},k}^{t-1} + l \times \left(T_{\mathrm{f},k}^{t-1} + T_{\mathrm{b},k}^{t-1} \right) + T_{\mathrm{f},k}^{t} \Big),$$

where T_{s2c} and T_{c2s} represent the communication time from the active party to the passive party and vice versa. Passive party k takes $T_{\mathrm{f},k}^{t}$ in round t for forward propagation and $T_{\mathrm{b},k}^{t}$ for backward propagation. We omit the subscript k and superscript t of iterations l due to consistency across the all parties and all rounds.

Theorem 1 (Size of Time Bubbles in FedBCD)**.** *Given Definition 1, the size of the time bubble of FedBCD for each passive party k in round t is:*

$$\hat{\mathcal{B}}_k^t = \mathcal{G}^t + T_{\mathrm{a}}^t + T_{\mathrm{s2c},k}^t - T_{\mathrm{b},k}^{t-1} - l \times \left(T_{\mathrm{f},k}^{t-1} + T_{\mathrm{b},k}^{t-1} \right) - T_{\mathrm{f},k}^{t} - T_{\mathrm{s2c},k}^{t-1},$$

where T_{a} represents the time for the active party to aggregate embeddings and compute the gradients for all passive parties.

Next, we analyze potential time bubbles in our algorithm for the passive party k, considering 3 complementary cases. Case 1: Passive party k has no time bubbles in round t; Case 2: Time Bubble exists, and the active party performs local training with $l_K^t \leq \bar{l}_K^t$ before conducting the aggregation operation; Case 3: Time Bubble exists, and the active party has to wait for the aggregation after reaching its local training limit $\bar{l}_K^t$.

We use $\mathcal{B}_{1,k}^t$ and $\mathcal{B}_{2,k}^t$ to represent the time bubbles in above Case 2 and Case 3. So we can calculate the size of the reduced time bubbles of our algorithm, compared to FedBCD, termed as *total reduced time*. We define $\tilde{k}$ as the slowest passive party that sends the embedding for round t in advance at the $\tilde{t}$ round, and $\tilde{t} + B_{\tilde{k}}^{\tilde{t}} = t$. The value of $\tilde{t}$ can be obtained by searching within the set $\{t-2, t-3, ..., t-B_{max}\}$.

Theorem 2 (Total Reduced Time in *PASTA*)**.** *The total size of the time that our algorithm reduces compared to FedBCD is:*

$$\Delta\mathcal{B}_k = \textstyle\sum_{t\in\mathcal{T}_1} \hat{\mathcal{B}}_k^t + \sum_{t\in\mathcal{T}_2} \left(\hat{\mathcal{B}}_k^t - \mathcal{B}_{1,k}^t \right) + \sum_{t\in\mathcal{T}_3} T_{B_{\tilde{k}}^{\tilde{t}}}^t,$$

where $T_{B_{\tilde{k}}^{\tilde{t}}}^t$ is the gap of the time bubbles between FedBCD and PASTA from $t - B_{\tilde{k}}^{\tilde{t}} + 1$ to t rounds. $\mathcal{T}_1$, $\mathcal{T}_2$ and $\mathcal{T}_3$ represent the three different cases mentioned earlier and ensures that $|\mathcal{T}_1| + |\mathcal{T}_2| + \sum_{t\in\mathcal{T}_3} B_{\tilde{k}}^{\tilde{t}} = R$. We define $\mathcal{I}$ as the set of indices that index the round to calculate, so that sets $\mathcal{T}_1, \mathcal{T}_2$ and $\mathcal{T}_3$ do not intersect with each other and form the complete set $\mathcal{I}$.

We have demonstrated that the all terms of the theorem are consistently greater than 0, and we can also observe that $T_{B_{\tilde{k}}^{\tilde{t}}}^t$ increases with the growth of $B_{\tilde{k}}^{\tilde{t}}$ [34]. Furthermore, if the communication latency is significant, the values of both $\hat{\mathcal{B}}_k^t$ and $T_{B_{\tilde{k}}^{\tilde{t}}}^t$ will also increase.

Therefore, our method achieves higher computational throughput and accomplishes more calculations than the synchronized FedBCD algorithm, which leads to improved training efficiency and faster convergence.

V. Evaluation

We evaluate our algorithm in this section, comparing *PASTA* with multiple baselines in various datasets.

A. Settings

1) Dataset and Model: We use representative datasets and suitable models for evaluation. The details are as follows:

- **MNIST**: Horizontally split into two parties (392 features each). Trained with 2-layer MLP (30 epochs).
- **Fashion-MNIST (FMNIST)**: Same partition as MNIST. Trained with CNN (30 epochs).
- **CIFAR10**: Evenly distributed in two-party setting (1536 features each) and four-party setting (768 features each). Trained with the VGG model (30 epochs).
- **NUSWIDE**: five web image categories. Active party: 634 low-level image features; passive parties: top-1000 tags. Trained with 2-layer MLP (10 epochs to convergence).

2) Baseline: We choose three state-of-the-art VFL methods as baselines and compare their performance with *PASTA*:

- **FedSGD** [19]: Each party in FedSGD updates its local model after processing each training batch.
- **FedBCD** [12]: FedBCD is a synchronous training method that makes multiple updates using stale gradients. In our experiments, each party in FedBCD performs one standard local update and four local updates per round.
- **FlexVFL-s**: We implement FlexVFL-s, a simplified version of FlexVFL [13] with local training timeout but without adaptive learning rate, to ensure synchronous embedding arrival while keeping other parameters (e.g., learning rate, optimizer) constant for fair scheduling comparison.

3) VFL Configurations: To evaluate the impact of whether the top model is trainable in the active party, we consider two distinct VFL settings for all the methods:

- **SplitVFL**: The top model is trainable. For MLP or CNN, the top model is defined as a fully connected layer. For VGG, the top model is the classification layer of VGG.
- **AggVFL**: The fixed top model directly aggregates bottom models' outputs as predictions, which is also a common setting introduced in [7].

Communi-cation time	Method	NUSWIDE		MNIST		FMNIST		CIFAR10	
		AggVFL	SplitVFL	AggVFL	SplitVFL	AggVFL	SplitVFL	AggVFL	SplitVFL
5	FedSGD	77.41±0.35	76.69±1.15	97.28±0.11	97.34±0.08	89.99±1.33	90.02±0.43	**78.40±2.28**	79.38±0.65
	FedBCD	75.82±0.71	76.04±0.56	97.27±0.35	97.05±0.18	90.04±0.57	90.14±0.43	77.89±1.41	**81.07±0.90**
	FlexVFL-s	75.69±0.79	75.85±0.46	97.29±0.16	97.09±0.32	89.95±0.52	90.05±0.42	77.96±1.21	80.45±0.62
	PASTA	**77.95±0.14**	**77.03±1.17**	**97.40±0.19**	**97.45±0.11**	**90.21±0.42**	**90.43±0.26**	78.30±0.56	79.42±0.45
50	FedSGD	**77.41±0.35**	**76.69±1.15**	**97.28±0.11**	97.34±0.08	89.99±1.33	90.02±0.43	**78.40±2.28**	79.38±0.65
	FedBCD	75.82±0.71	76.04±0.56	97.27±0.35	97.05±0.18	90.04±0.57	90.14±0.43	77.89±1.41	**81.07±0.90**
	FlexVFL-s	73.30±0.64	74.04±1.96	97.13±0.10	97.22±0.20	86.32±0.73	90.18±0.25	76.43±0.86	77.81±0.98
	PASTA	77.14±0.37	76.51±1.74	97.25±0.31	**97.43±0.23**	**90.07±0.24**	**90.38±0.18**	78.16±0.41	79.94±0.69
100	FedSGD	**77.41±0.35**	**76.69±1.15**	**97.28±0.11**	**97.34±0.08**	89.99±1.33	90.02±0.43	**78.40±2.28**	79.38±0.65
	FedBCD	75.82±0.71	76.04±0.56	97.27±0.35	97.05±0.18	**90.04±0.57**	90.14±0.43	77.89±1.41	**81.07±0.90**
	FlexVFL-s	73.30±0.64	74.04±1.96	97.13±0.10	97.22±0.20	86.32±0.73	90.18±0.25	76.43±0.86	80.45±0.62
	PASTA	77.10±0.39	76.47±1.63	97.09±0.21	97.30±0.18	**90.04±0.58**	**90.43±0.24**	77.39±0.58	79.36±0.74
1000	FedSGD	**77.41±0.35**	**76.69±1.15**	**97.28±0.11**	97.34±0.08	89.99±1.33	90.02±0.43	**78.40±2.28**	79.38±0.65
	FedBCD	75.82±0.71	76.04±0.56	97.27±0.35	97.05±0.18	**90.04±0.57**	90.14±0.43	77.89±1.41	**81.07±0.90**
	FlexVFL-s	73.30±0.64	74.04±1.96	97.13±0.10	97.22±0.20	86.32±0.73	90.18±0.25	76.43±0.86	80.45±0.62
	PASTA	76.96±0.39	76.53±1.57	97.19±0.10	**97.51±0.24**	89.80±0.31	**90.45±0.25**	77.09±0.59	79.51±0.65

TABLE I: Final accuracy with different datasets and communication latencies. Bold values represent the highest test accuracy, and underlined values represent the second best accuracy.

4) Training Setting: Each time unit is set to 10ms. There are two types of settings: (i) one active party and three passive parties for results in Section V-E, V-D, (ii) one active party and seven passive parties for Section V-E, and (iii) one active party and one passive party for Sections V-B, V-C and V-D. Note that the number of parties in VFL is generally much fewer than that in HFL [9], [35]–[37], as VFL parties in practice represent a hospital, a bank, or a internet company, rather than end devices commonly considered in HFL. The active and passive parties have forward/backward times of 10/20 time units, with aggregation taking 10 time units. Communication latency is set to 5, 50, 100, or 1000 time units. The maximum number of in-flight batch B_{max} and maximum staleness s_{max} parameters are set to 3 and 4 respectively. We utilized a open-source framework called VFLAIR [38] to implement our code.

B. Performance Evaluation across Various Datasets

For each dataset, we evaluate *PASTA* and baselines under two VFL configurations, while recording the duration to reach the target training epochs and corresponding test accuracy. Each experiment is repeated 5 times under the same setting.

Comparison across different datasets. Test accuracy across settings is summarized in Table I. Under low latency (5 or 50 time units), our algorithm *PASTA* achieves optimal/suboptimal accuracy due to frequent updates in training, outperforming FedSGD. Compared to FedBCD and FlexVFL-s, *PASTA* better controls staleness effects, yielding higher accuracy. As latency increases (100, 1000 time units), accuracy slightly declines due to operations like embedding pre-sending. At 1000 time units, pre-sent embeddings and stale training peak, but accuracy loss remains manageable, with minimal impact. The slight drop in accuracy is acceptable, as later we can see excellent performance in accelerating convergence in Fig. 4.

Comparison on convergence speedup. As shown in Fig. 4, *PASTA* significantly outperforms others in speedup as latency increases. On MNIST, it achieves speedups of 0.9, 2.0, 2.7, and 4.4 relative to FedSGD at latencies of 5, 50, 100, and 1000 time units, respectively. At 1000 time units, *PASTA* achieves speedups of 4.4, 4.6, 3.2, and 3.5 on MNIST, Fashion-MNIST, CIFAR10, and NUSWIDE, surpassing FedBCD (1.7, 1.1, 1.4, 2.3). For brevity, SplitVFL results are shown, and AggVFL exhibits similar speedup trends.

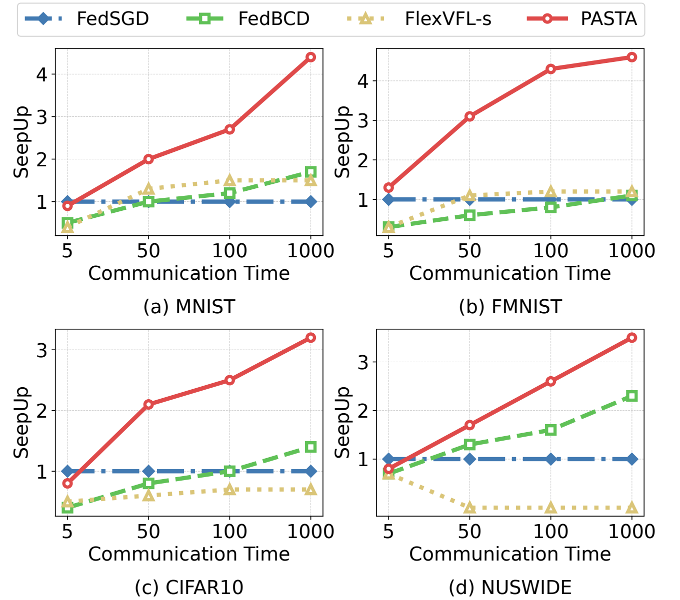

Fig. 4. Evaluation of speedup across multiple datasets under SplitVFL setting.

C. Sensitivity to Different Factors

In this subsection, we explore the influence of hyperparameters on the performance of *PASTA* using the FMNIST dataset. We configure the communication latency to 1000 time units to evaluate the performance comparison of various hyperparameters in high-latency scenarios.

Sensitivity of maximum staleness s_{max}. As shown in Table II, the initial staleness value has significant impact on

the accuracy in AggVFL. When the staleness value exceeds 4, the speedup ratio gradually decreases because it requires more rounds to achieve the same accuracy. In SplitVFL, the staleness control is relatively robust. The speedup ratio initially increases and then gradually decreases, suggesting that the algorithm convergence can be effectively improved by appropriately setting the maximum staleness value.

s_{max}	AggVFL		SplitVFL	
	Accuracy	**Time (to 89%)**	**Accuracy**	**Time (to 90%)**
2	**90.11±0.20**	**731040 (4.3×)**	90.21±0.29	2095700 (3.2×)
3	90.05±0.46	898040 (3.5×)	90.33±0.18	1538830 (4.4×)
4	89.80±0.31	819040 (3.8×)	90.45±0.25	1468620 (4.6×)
6	89.11±0.53	1797460 (1.7×)	90.34±0.19	**1391300 (4.8×)**
8	89.28±0.38	1883390 (1.7×)	**90.56±0.42**	1394870 (4.8×)
10	89.05±0.44	2213430 (1.4×)	90.52±0.55	1480650 (4.6×)
12	89.01±0.57	2461350 (1.3×)	90.20±0.49	1565410 (4.3×)

TABLE II: Comparison of final accuracy and time with various s_{max}. The numbers in parentheses represent the speedup of the time to reach the target accuracy compared to FedSGD.

Sensitivity of maximum number of in-flight batches B_{max}. Table III illustrates that in high-latency scenarios, setting B_{max}=1 results in slower training. However, with an increase in B_{max}, the speedup can be further enhanced. If communication latency is the primary bottleneck, increasing the number of batches sent can effectively improve the training time.

B_{max}	AggVFL		SplitVFL	
	Accuracy	**Time (to 89%)**	**Accuracy**	**Time (to 90%)**
1	89.06±0.90	2952240 (1.1×)	90.26±0.37	4651440 (1.4×)
2	89.57±0.12	1103120 (2.8×)	90.18±0.14	2078390 (3.2×)
3	89.80±0.31	819040 (3.8×)	90.45±0.25	1468620 (4.6×)
4	**90.00±0.25**	738900 (4.2×)	**90.48±0.31**	1042640 (6.5×)
5	89.82±0.35	**495070 (6.3×)**	90.35±0.10	**933400 (7.2×)**

TABLE III: Performance comparison with different B_{max}. The benchmark for speedup in parentheses is FedSGD.

D. Ablation Study

We evaluate the staleness decreasing strategy's effect on convergence using CIFAR10 with 2 and 4 parties setting. For 2 parties, communication latency is 1000 time units. For 4 parties, a heterogeneous setup includes one active party (forward: 10, backward: 20) and three passive parties: a quick trainer (forward: 8, backward: 15, latency: 5), a slow trainer (forward: 25, backward: 50, latency: 5), and one with high latency (forward: 10, backward: 20, latency: 100).

Number of parties	Framework	Decreasing strategy	Accuracy	Time (to 75%, 77%, 73%, 76%)
2	AggVFL	w/o	75.91±1.90	4109700 (1.9×)
		w/	**77.09±0.59**	**3514840 (2.3×)**
	SplitVFL	w/o	79.16±1.24	3561760(2.7×)
		w/	**79.51±0.65**	**2976350 (3.2×)**
4	AggVFL	w/o	73.32±1.13	1501815 (0.7×)
		w/	**73.57±0.58**	**682745 (1.5×)**
	SplitVFL	w/o	**77.19±0.85**	938775 (2.2×)
		w/	77.05±0.43	**682745 (3.0×)**

TABLE IV: The effectiveness of our staleness decreasing strategy. The benchmark for speedup in parentheses is FedSGD.

As shown in Table IV, the staleness decreasing strategy not only accelerates convergence but also helps protect model performance from the negative impact of staleness, leading to higher accuracy. If the staleness decreasing strategy is not adopted, the number of convergence rounds increases significantly, resulting in a longer time to achieve the target accuracy. Specifically, we observe a 16%-54% speed improvement in convergence time with this strategy across various scenarios.

E. Testbed Experiments and Scalability

Our testbed consists of four 4090 GPUs connected to a switch, with each party as a process on one GPU. We leverage the Linux Traffic Control program to configure the Network Interface Card. The transmission rates (Mbps) of the passive parties are $\{100, 50, 5\}$ due to communication heterogeneity.

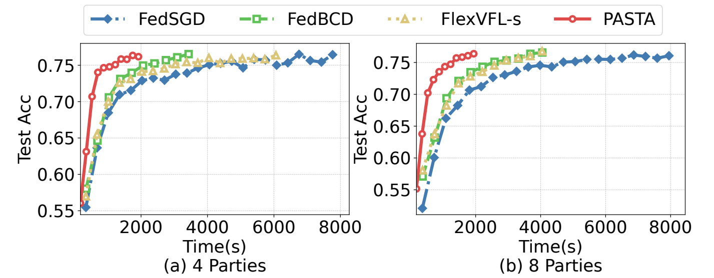

Fig. 5. Testing accuracy versus time (second) for CIFAR10 dataset in our distributed VFL testbed.

Each experiment is repeated three times, with average CIFAR10 results shown in Fig 5. FedSGD takes 7766s to reach 76% accuracy due to slow communication, while FedBCD and FlexVFL-s achieve it in 3431s and 6058s, respectively. *PASTA* outperforms both, reaching the target in 1922s with stable training—$4.04\times$ faster than FedSGD and $1.79\times$ faster than FedBCD—demonstrating its efficiency in real-world scenarios.

To further validate the scalability of *PASTA*, we expanded the setup to 8 parties, showing similar effectiveness. The transmission rates (Mbps) between the active party and the 7 passive parties are $\{100, 100, 50, 50, 50, 5, 5\}$. *PASTA* achieved a training time that was 4.09 times faster than FedSGD and 2.08 times faster than both FedBCD and FlexVFL-s.

VI. Conclusion

In this paper, we propose *PASTA*, a pipeline parallelism framework to accelerate training in vertical federated learning. Our approach optimizes the communication and computation pipeline, via exploiting the trade-off between using stale gradients for communication reduction and preserving model convergence, ensuring efficient utilization of resources and reducing idle times. In particular, our proposed feedback mechanism effectively enhances the training process under high latency and heterogeneity scenarios, ultimately improving the efficiency and effectiveness of the overall training framework. Extensive experimental results demonstrate the superiority and effectiveness of our *PASTA* algorithm, bringing significant speedup without sacrificing the final accuracy.

REFERENCES

[1] Q. Yang, Y. Liu, T. Chen, and Y. Tong, "Federated machine learning: Concept and applications," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 10, no. 2, pp. 1–19, 2019.

[2] J. Zhou, H. Liang, T. Wu, X. Zhang, Y. Jiang, and C. W. Tan, "Vfl-cafe: Communication-efficient vertical federated learning via dynamic caching and feature selection," *Entropy*, vol. 27, no. 1, p. 66, 2025.

[3] 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*. PMLR, 2017, pp. 1273–1282.

[4] J. Huang, F. Liu, and J. Zhang, "Multi-dimensional qos evaluation and optimization of mobile edge computing for iot: A survey," *Chinese Journal of Electronics*, vol. 33, no. 4, pp. 859–874, 2024.

[5] T. Haibo, L. Maonan, and R. Shuangyin, "Ese: Efficient security enhancement method for the secure aggregation protocol in federated learning," *Chinese Journal of Electronics*, vol. 32, no. 3, pp. 542–555, 2023.

[6] W. Liu, Z. Zhan, C. Joe-Wong, E. Ngai, J. Duan, D. Guo, X. Chen, and X. Zhang, "Taco: Tackling over-correction in federated learning with tailored adaptive correction," *arXiv preprint arXiv:2504.17528*, 2025.

[7] Y. Liu, Y. Kang, T. Zou, Y. Pu, Y. He, X. Ye, Y. Ouyang, Y.-Q. Zhang, and Q. Yang, "Vertical federated learning: Concepts, advances, and challenges," *IEEE Transactions on Knowledge and Data Engineering*, 2024.

[8] L. Zhu, H. Lin, Y. Lu, Y. Lin, and S. Han, "Delayed gradient averaging: Tolerate the communication latency for federated learning," *Advances in Neural Information Processing Systems*, vol. 34, pp. 29 995–30 007, 2021.

[9] F. Fu, X. Miao, J. Jiang, H. Xue, and B. Cui, "Towards communication-efficient vertical federated learning training via cache-enabled local updates," *arXiv preprint arXiv:2207.14628*, 2022.

[10] W. Liu, X. Zhang, J. Duan, C. Joe-Wong, Z. Zhou, and X. Chen, "Dynamite: Dynamic interplay of mini-batch size and aggregation frequency for federated learning with static and streaming dataset," *IEEE Transactions on Mobile Computing*, 2023.

[11] T. J. Castiglia, A. Das, S. Wang, and S. Patterson, "Compressed-vfl: Communication-efficient learning with vertically partitioned data," in *International Conference on Machine Learning*. PMLR, 2022, pp. 2738–2766.

[12] Y. Liu, X. Zhang, Y. Kang, L. Li, T. Chen, M. Hong, and Q. Yang, "Fedbcd: A communication-efficient collaborative learning framework for distributed features," *IEEE Transactions on Signal Processing*, vol. 70, pp. 4277–4290, 2022.

[13] T. Castiglia, S. Wang, and S. Patterson, "Flexible vertical federated learning with heterogeneous parties," *IEEE Transactions on Neural Networks and Learning Systems*, 2023.

[14] J. Zhang, S. Guo, Z. Qu, D. Zeng, H. Wang, Q. Liu, and A. Y. Zomaya, "Adaptive vertical federated learning on unbalanced features," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 4006–4018, 2022.

[15] T. Chen, X. Jin, Y. Sun, and W. Yin, "Vafl: a method of vertical asynchronous federated learning," *arXiv preprint arXiv:2007.06081*, 2020.

[16] Z. Wu, Q. Li, and B. He, "Practical vertical federated learning with unsupervised representation learning," *IEEE Transactions on Big Data*, 2022.

[17] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *Advances in neural information processing systems*, vol. 32, 2019.

[18] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: generalized pipeline parallelism for dnn training," in *Proceedings of the 27th ACM symposium on operating systems principles*, 2019, pp. 1–15.

[19] S. Hardy, W. Henecka, H. Ivey-Law, R. Nock, G. Patrini, G. Smith, and B. Thorne, "Private federated learning on vertically partitioned data via entity resolution and additively homomorphic encryption," *arXiv preprint arXiv:1711.10677*, 2017.

[20] Y. Hu, D. Niu, J. Yang, and S. Zhou, "Fdml: A collaborative machine learning framework for distributed features," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, New York, NY, USA, 2019, p. 2232–2240.

[21] J. Sun, A. Li, L. Duan, S. Alam, X. Deng, X. Guo, H. Wang, M. Gorlatova, M. Zhang, H. Li *et al.*, "Fedsea: A semi-asynchronous federated learning framework for extremely heterogeneous devices," in *Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems*, 2022, pp. 106–119.

[22] F. Fu, Y. Shao, L. Yu, J. Jiang, H. Xue, Y. Tao, and B. Cui, "Vf2boost: Very fast vertical federated gradient boosting for cross-enterprise learning," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 563–576.

[23] Y. Han, P. Du, and K. Yang, "Fedgbf: An efficient vertical federated learning framework via gradient boosting and bagging," *arXiv preprint arXiv:2204.00976*, 2022.

[24] B. Gu, Z. Dang, X. Li, and H. Huang, "Federated doubly stochastic kernel learning for vertically partitioned data," in *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, 2020, pp. 2483–2493.

[25] Q. Zhang, B. Gu, C. Deng, and H. Huang, "Secure bilevel asynchronous vertical federated learning with backward updating," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 12, 2021, pp. 10 896–10 904.

[26] Q. Zhang, B. Gu, C. Deng, S. Gu, L. Bo, J. Pei, and H. Huang, "Asysqn: Faster vertical federated learning algorithms with better computation resource utilization," in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, 2021, pp. 3917–3927.

[27] B. Gu, A. Xu, Z. Huo, C. Deng, and H. Huang, "Privacy-preserving asynchronous vertical federated learning algorithms for multiparty collaborative learning," *IEEE transactions on neural networks and learning systems*, vol. 33, no. 11, pp. 6103–6115, 2021.

[28] A. Khan, M. ten Thij, and A. Wilbik, "Communication-efficient vertical federated learning," *Algorithms*, vol. 15, no. 8, p. 273, 2022.

[29] D. Cai, T. Fan, Y. Kang, L. Fan, M. Xu, S. Wang, and Q. Yang, "Accelerating vertical federated learning," *IEEE Transactions on Big Data*, p. 1–10, 2024. [Online]. Available: http://dx.doi.org/10.1109/TBDATA.2022.3192898

[30] W. Chen, G. Ma, T. Fan, Y. Kang, Q. Xu, and Q. Yang, "Secureboost+: A high performance gradient boosting tree framework for large scale vertical federated learning," *arXiv preprint arXiv:2110.10927*, 2021.

[31] W. Xu, H. Fan, K. Li, and K. Yang, "Efficient batch homomorphic encryption for vertically federated xgboost," *arXiv preprint arXiv:2112.04261*, 2021.

[32] J. Sun, Z. Xu, D. Yang, V. Nath, W. Li, C. Zhao, D. Xu, Y. Chen, and H. R. Roth, "Communication-efficient vertical federated learning with limited overlapping samples," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2023, pp. 5203–5212.

[33] D. Narayanan, A. Phanishayee, K. Shi, X. Chen, and M. Zaharia, "Memory-efficient pipeline-parallel dnn training," in *International Conference on Machine Learning*. PMLR, 2021, pp. 7937–7947.

[34] "Source code and analysis," https://github.com/PointerA/PASTA.

[35] H. Liang, Z. Zhan, W. Liu, X. Zhang, C. W. Tan, and X. Chen, "Fedrema: Improving personalized federated learning via leveraging the most relevant clients," in *ECAI 2024*. IOS Press, 2024, pp. 2090–2097.

[36] S. Chen, W. Liu, X. Zhang, H. Xu, W. Lin, and X. Chen, "Adaptive personalized federated learning for non-iid data with continual distribution shift," in *2024 IEEE/ACM 32nd International Symposium on Quality of Service (IWQoS)*. IEEE, 2024, pp. 1–6.

[37] W. Liu, X. Zhang, J. Duan, C. Joe-Wong, Z. Zhou, and X. Chen, "Adacoopt: Leverage the interplay of batch size and aggregation frequency for federated learning," in *2023 IEEE/ACM 31st International Symposium on Quality of Service (IWQoS)*. IEEE, 2023, pp. 1–10.

[38] T. Zou, Z. Gu, Y. He, H. Takahashi, Y. Liu, G. Ye, and Y.-Q. Zhang, "Vflair: A research library and benchmark for vertical federated learning," *arXiv preprint arXiv:2310.09827*, 2023.