FedSU: Communication-efficient Federated Learning with Speculative Updating

Wei Yu[1], Chen Chen[1], Qinbin Li[2], Jieru Zhao[1], Shixuan Sun[1], Bo Li[3], Minyi Guo[1],
[1]Shanghai Jiao Tong University, Shanghai, China
[2]Huazhong University of Science and Technology, Wuhan, China
[3]Hong Kong University of Science and Technology, Hong Kong, China
Email: {yuweivvv, chen-chen}@sjtu.edu.cn, qinbin@hust.edu.cn, {zhao-jieru, sunshixuan}@sjtu.edu.cn,
bli@cse.ust.hk, guo-my@cs.sjtu.edu.cn

***Abstract*—Federated learning enables mobile devices to collaboratively learn a global model in iterative communication rounds. Many sparsification methods have been proposed for communication compression of FL, working by not synchronizing insignificant updates. However, we find there still exist unexploited sparsification opportunities: given the update similarity across different rounds, parameters often exhibit a linear updating pattern; motivated by speculative execution in computer architecture domain, it is promising to use the predicted gradients to refine the linearly-updating parameters without synchronization. To that end, we propose Federated Learning with Speculative Updating, or FedSU, to attain larger sparsification ratio without compromising model accuracy. In particular, to identify the linearly-updating parameters efficiently at runtime, we devise a regression-free method that diagnoses parameter linearity based on whether the second-order parameter difference is oscillating around 0. Meanwhile, to ensure convergence validity, FedSU leverages the prediction error as a feedback signal—so as to timely return to regular updating if the parameter no longer follows the linear pattern in reality. We have implemented FedSU as a Python module, and large-scale experiments in an emulated FL setup confirm that FedSU can remarkably improve the communication efficiency of FL, with a convergence speedup of over 40%.**

I. Introduction

Federated Learning (FL) [1]–[3] is nowadays an increasingly popular paradigm that allows edge clients to collaborate in model training without disclosing their local private data. In FL, the participating clients iteratively pull the latest model parameters from the FL server and then push back the updated ones after multiple local iterations (called a *round*). However, due to the limited bandwidth resources of typical edge devices and the increasing complexity of the neural network models, communication between the FL clients and server has become a major performance bottleneck.

To mitigate the network bottleneck in FL, a significant amount of research works [4]–[9] have been proposed, of which the *sparsification-style* methods [10]–[13] stand out because of their potentially high communication compression ratio. Sparification means to selectively synchronize only a portion of the model parameters, and CMFL [11] and APF [12] are two classical methods in that regard that can reduce the communication volume without accuracy loss. Nonetheless, those methods are essentially filtering out only those *insignificant* updates (i.e., close to 0) at synchronization time, and we notice that there still exists large unexploited territory that can be leveraged for making more aggressive lossless sparisification.

In fact, during the FL process, there usually exists strong similarity among the updates of consecutive rounds, as indicated by both existing works [11], [14] and our testbed measurements (Sec. III-A). Given such cross-round update similarity, our microscopic observations further show that, the evolution trajectories of individual parameters often exhibit a *linear* pattern. Therefore, it is promising to use the predicted gradient instead of conducing real synchronization to harvest the desired model updating effect. We note that such gradient prediction task is similar to the CPU *branch prediction* problem in computer architecture design, for which *speculative execution* [15], [16]—directly executing a probable branch and rectifying the results in cases with wrong prediction—is a highly effective solution. We therefore transfer the concept of speculative execution into Federated Learning (FL), and propose *Federated Learning with Speculative Updating* (FedSU)—seeking to reduce communication cost by leveraging the predicted updates for model refinement.

However, when enforcing FedSU into practice, there are two challenges. First, given the resource constraints on edge clients and potential gradient fluctuations, how to efficiently and accurately identify the linearly-updating parameters at runtime? Second, given that the linear evolution pattern may terminate at any time during training, how to guarantee convergence validity by timely resuming regular updating once the linear pattern no longer holds?

Regarding the first challenge, we adopt an indirect linearity-diagnosis method instead of conducting costly linear regression. We note that for a parameter updating linearly, its gradient (i.e., first-order difference) would be stable, and the second order difference would thus be close to (i.e., oscillating around) 0. Therefore, we propose to make linearity diagnosis based on a metric we call *second-order oscillation ratio*, which further incorporates the smoothing technique to attain better efficiency. For a parameter whose second-order oscillation ratio is lower than a given threshold, we can judge it as in linear mode, and can potentially use the predicted gradients instead of the synchronized ones for its refinement. Regarding the second challenge, we notice that the gradient prediction

error can be collected as a feedback signal indicating whether the linear updating pattern still persists in reality. We also mathematically prove that FedSU can guarantee convergence validity for general non-convex models.

We have implemented FedSU with a Python Module called `FedSU_Manager`. On each client, that module maintains a predictability mask and uses it to sparsify (restore) the model updates before (after) synchronization. Our evaluations in a 128-node EC2 cluster emulating realistic FL setup show that, FedSU can remarkably improve FL communication efficiency without accuracy loss. For example, it can speed up the convergence of the ResNet model by 46.1% (with a communication volume reduction of 71.7%). We also demonstrate that the negative overhead of FedSU is negligible compared with its performance benefit in efficiency optimization.

II. Background

A. Federated Learning Basics

Data privacy is a key concern when adopting machine learning technology in many real-world scenarios like hospitals and banks. Federated Learning (FL) [1], [2] comes as a popular paradigm that allows multiple clients to jointly train a model with minimum privacy leakage. In FL, there is a server node that maintains the global model. Regarding the interaction between the FL server and the clients, FedAvg [2] has been commonly adopted as a norm for FL. In FedAvg, the FL clients communicate with the FL server in communication *rounds*. In each round, the FL clients pull the latest model parameters, refine those parameters for *multiple* local iterations with their private data, and finally report the updates[1] back to the FL server; the FL server then refreshes the global model with the average value of client updates.

In particular, server-client communication is a severe performance bottleneck of FL. Compared with distributed machine learning in dedicated clusters, the typical network bandwidth between the FL server and clients is much smaller. For example, when conducing FL with cellphones for mobile applications like Gboard [17], the sever-client connection (over the Internet) has a typical bandwidth of tens of Mega-bits per second [18]. Existing studies [19]–[21] have shown that, when training typical neural network model with Megabit-level bandwidth, a substantial portion (e.g., over a half for ResNet model) of the per-round time might be spent on communication. Therefore, it is in urgent need to improve the communication efficiency of FL.

B. Prior Arts and Their Limitations

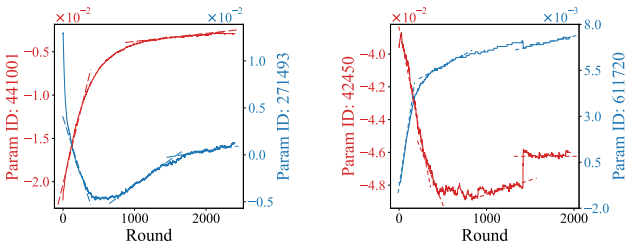
In the literature, a bewildering array of research works have already been proposed to improve the communication efficiency of FL. As a classical method, *quantization* [4]–[6] means to use fewer bits to represent each parameter update value in communication (i.e., from 32bits to 16bits); meanwhile, *matrix factorization* [7]–[9] is also an effective method that works by decomposing the vanilla gradient matrix into two lower-rank vectors, the communication volume of the latter is much smaller. However, the maximum communication compression level of quantization is relatively limited (constrained by the minimum bit number required to ensure convergence validity), and the matrix factorization methods are only applicable to certain over-parameterized models [7], both exhibiting noticeable limitations.

In recent years, the *sparsification-style* communication compression techniques [10]–[13] have become increasingly popular, which are the focus of this paper. Sparsification means to only transmit the updates that are *significant* enough when conducting global aggregation. In that regard, CMFL [11] and APF [12] are two classical algorithms that can mitigate communication bottleneck without accuracy loss. Under CMFL, a client reports its local gradients to the FL server only when a sufficient portion of the (positive/negative) element directions are consistent with the global ones. APF makes sparsification decisions based on the convergence status of individual parameters: it excludes a parameter from being synchronized if that parameter converges earlier than others.

However, we find that the above sparsification methods fail to thoroughly exploit the potential communication-optimization opportunities. By not transmitting the insignificant gradients, these methods are essentially replacing the value of the skipped gradients to 0 at aggregation time, with the expectation that such replacement would (almost) not affect the training accuracy. Note that, to preserve the training accuracy is to preserve the expected parameter updating trajectory; existing works actually seek to avoid communicating *insignificant* gradients (e.g., the small-scale [11] or temporally-zigzagging [12] gradients) as long as the macroscopic parameter evolution pattern still holds. Nonetheless, those works only focus on the pattern that the parameter *stagnates*, which is *over-conservative* and renders the volume of gradients that can be skipped for transmission quite limited. In fact, if a parameter is not stagnating at a fixed value but *changing at a fixed slope*, we can also train models effectively by directly emulating the original updating trajectory without conducting realistic gradient transmission. In that sense, stagnating pattern is a special case of linearly-changing pattern; with such generalization, we can open up a broader optimization space and attain better communication efficiency.

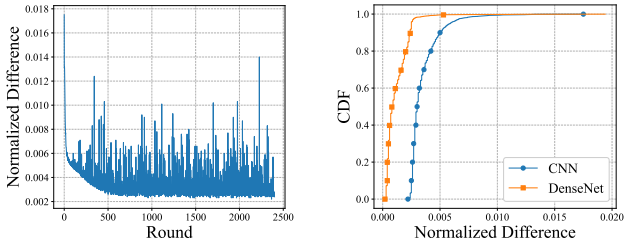
Objective. Our objective of this paper is to further improve FL communication efficiency with more aggressive gradient sparsification. In the next section, we will conduct a series of empirical measurements to systematically study the parameter evolution characteristics during the FL process, quantitatively exposing the potential sparsification opportunities therein.

[1]Here "updates" mean the *gradients* instead of the parameters. While it is the parameters that are synchronized by default under FedAvg [1], the two schemes are essentially equivalent and we choose the former for simplicity. Besides, in this paper we interchangeably use "gradient" and "update", both representing the parameter variation accumulated over an entire round.



(a) CNN (b) DesNet

Fig. 1: Evolution trajectory curves of two randomly-selected parameters when training CNN and DesNet. There exist widespread training periods with strong trajectory linearity.



(a) Instantaneous Norm. Difference (b) CDF of CNN and DenseNet

Fig. 2: The instantaneous values of Normalization Difference under CNN, as well as the cumulative distribution function (CDF) of the Normalization Difference values for both CNN and DenseNet.

III. Motivation

A. Prevalence of Linear Parameter Evolution Pattern

As previously discussed, we believe the linear parameter evolution pattern is a promising angle to yield more aggressive communication compression. Before exploring how to leverage such a linear pattern, we first measure how prevalent such a linear pattern is during typical FL processes.

We first depict the instantaneous values of two randomly-selected parameters respectively in a CNN model and the DenseNet model (for the detailed description of the training setup, please refer to Sec. VI). As shown in Fig. 1, the displayed parameters first change dramatically and then only gently as training proceeds. In particular, the parameter updating trajectories are indeed highly predictable: as marked in the dashed lines, the parameter value curves exhibit strong linearity for a large time portion of the entire training process. Moreover, we also notice that the linearity periods for different (scalar) parameters are not identical even in the same model, indicating that our later sparsification decisions shall be made in a fine-grained manner—independently for each parameter. Such fine-grained behavior heterogeneity is also observed in some existing works [10], [22]. For example, some features may be easier to learn, and their corresponded parameters in the neural network model may thus converge faster (called non-uniform convergence in [10]).

In fact, linear parameter variation is not a coincidence but a common pattern as endorsed by existing works. Wang et al. [11] have pointed out that the model parameters usually converge steadily and smoothly during training, with the differences between any two sequential global updates being quite small. They devise a metric called *normalized difference*, which is defined as $\frac{\|\delta_{t+1}-\delta_t\|}{\|\delta_t\|}$; here δ_t is the (global) update vector in iteration t. As reported by Wang et al. [11], when training CNN and LSTM models, the normalized difference values for more than 93% global updates are smaller than 0.05; such small values indicate that the updates of consecutive iterations are highly stable, and the parameter variation curves thus possess strong linearity. In fact, it is widely accepted in analysis that DNN models are smooth (see Assumption 1 in Sec. IV-D); an equivalent expression of the smoothness property is that the second-order gradients be bounded by a modest smoothness constant, which to certain extent can endorse the existence of parameter linearity in a theoretical perspective.

We further calculate the above normalized difference metric from the original *cross-iteration* scale to the *cross-round* scale for FL. We train the CNN and DenseNet model under a FL setup, where each round contains 50 iterations. In Fig. 2a we measure the variation of the normalized difference value over the entire training process for CNN, and in Fig. 2b we further show the cumulative distribution function (CDF) of the normalized difference metric for both CNN and DesNet. We find that the normalized difference values are even smaller at such coarser-grained per-round granularity: Fig. 2a suggests that the normalized difference value for CNN is almost always smaller than 0.01, and Fig. 2b further shows that more than 90% of the updates have a normalized difference less than 0.005. This is because the gradient fluctuation caused by mini-batch randomness—when accumulated over multiple iterations composing a round—is in fact smoothed by a certain extent, thus rendering the per-round updates even more stable.

In a word, per-round model updates are quite similar across consecutive rounds in FL. In fact, with a linearity diagnosis mechanism we will elaborate later (Sec. IV-A), we find that over 80% parameters exhibit a linear evolution pattern for at least a half of the total training time (Fig. 7 in Sec. VI-C), consistent with the sampled trajectories in Fig. 1. Therefore, by exploiting such linear-pattern for sparsified communication, we can potentially attain remarkable enhancement in FL communication efficiency.

B. Insight and Challenges

Motivated by the prevalence of linear parameter updating pattern, our insight in this paper is to conduct linearity-oriented gradient sparsification. That is, we keep monitoring the global parameter updating pattern at runtime; for those parameters with linear trajectory, we can manually emulate the expected parameter updating result without conducting realistic gradient synchronization, so as to reach ideal accuracy performance with much less communication.

In essential, we need to estimate parameter update prior to realistic gradient aggregation, in hopes that our estimation is valid and the communication cost can be saved without any

negative impact; otherwise we have to take special measures for remediation. This is similar to the famous *speculative execution* technique in computer architecture design [15], [16]. To avoid resource wastage in pipelined execution, speculative execution allows processors to aggressively execute instructions beyond unresolved branches: the processor uses dynamic branch prediction to select a likely branch to execute; the execution results are temporally buffered and later accepted or revoked after the control dependency is resolved (i.e., valid branch determined). Given that dynamic branch prediction is usually highly accurate in practice [23], [24], speculation can effectively enhance CPU utilization and is nowadays a cornerstone technique in modern high-performance processors.

We therefore borrow the concept of speculative execution to FL and propose *Federated Learning with Speculative Updating* (FedSU), which deems gradients as the hidden truth to predict, just like the CPU branch. As long as we can predict the parameter update at an accuracy high enough, we can effectively reduce the overall communication volume. To that end, we limit such prediction to the parameters already exhibiting a strong linear pattern, whose updating trajectories are likely to be persistently linear in the near future.

Challenges. Nonetheless, in designing FedSU, we face two immediate challenges. First, identifying those parameters with a linear pattern is a non-trivial task for FL given the well-known resource constraints on typical FL devices. We need to devise an effective method that can diagnose trajectory linearity with high efficiency in both time (computation) and memory consumption. Second, as can be observed in Fig. 1, linear parameter evolution is only a temporal pattern that may last for any arbitrary length, suggesting that the predicted update may diverge from the true one at any time in the midst of a FL process. To ensure training validity, we need to know the ground truth on how the gradients would evolve under standard FL, and leverage it to instruct whether to continue speculative updating or switch to regular updating at execution time. We will address such challenges in the next section.

IV. Solution

In this section, we will elaborate the design details of our proposed FedSU algorithm. In general, the workflow of FedSU is composed of three parts: first to efficiently identify the parameters with linear evolution pattern, second to use the predicted gradient to refine those parameters, and third to introduce an error-feedback mechanism to ensure convergence validity. We will respectively elaborate each part, and conclude this section with the convergence analysis of FedSU.

A. Efficient Identification of Parameters with Strong Linearity

The need for efficient linearity identification. To realize FedSU, a premier task is to identify the parameters with strong linear updating pattern at runtime, whose gradients would then be the spasification target to be exempted from being synchronized. Specifically, after each round completes, we need to conduct linearity diagnosis for each parameter based on its latest value and the historical values (all being post-synchronization global values). Although sounding straightforward, it is nonetheless a non-trivial task. For communication efficiency, the task of linearity diagnosis is located respectively on each client (feasible because each client share the same model parameters after synchronization); otherwise the diagnosis results have to be transferred from the FL server to each client, which brings extra communication overhead. However, the computation and memory resources on FL clients are quite limited, and we thus need to design an accurate and also resource-efficient method for linearity diagnosis.

Deficiency of regression-style solutions. For the problem of linearity diagnosis, a classical method goes to linear regression analysis [25], [26]: first collect enough historical data in an observation window, and then fit out the coefficients of a linear function with the *least square* method [27]—the sum of residual error squares working as a quantitative signal of the linearity level. However, such an intuitive method have two remarkable deficiencies. First, it is hard to determine the observation window size. An over-large window may make the diagnosis misled by out-of-dated data, and an over-small window is prone to the recent fluctuations caused by factors like mini-batch randomness, both rendering the diagnosis result inaccurate. Second, that method requires maintaining plenty of historical model snapshots in local inventory, incurring large storage overhead. Meanwhile, it also consumes large computation power because, to refresh the coefficients, the least square algorithm must be repeated over the entire window after each round.

Insight: linearity diagnosis with second-order parameter differences. To attain accurate and efficient linearity diagnosis, we propose a *regression-free* second-order differential method. We note that, when a parameter exhibits a linear updating pattern, its gradient value (i.e., *first-order parameter difference*) would stabilize around a fixed value; further, the *second-order parameter difference* would stabilize (*oscillate* at the microscopic level, in fact) around 0. Although mathematically equivalent, it is however much easier to judge whether a variable is oscillating around 0 than to judge whether it is changing linearly. Intuitively, let $\{x_{k-K+1}, x_{k-K+2}, ..., x_k\}$ be the historical values of a (scalar) parameter at round-k (K is the observation window size), then we can define a metric called *second-order oscillation ratio*:

$$\mathcal{R} = \frac{|\sum_{r=k-K+1}^{k} g'_r|}{\sum_{r=k-K+1}^{k} |g'_r|}, \tag{1}$$

where $g'_k = g_k - g_{k-1}$ is the second-order parameter difference and $g_k = x_k - x_{k-1}$ the first-order. When $\mathcal{R}$ is close to 0 (i.e., $\{g'_k\}$ oscillating around 0), we can learn that $\{g_k\}$ is stable and x is updated under a linear pattern.

Formal solution with the metric of *second-order oscillation ratio*. Exempted from the need for linear regression, we further enhance the above method with the EMA smoothing technique (so as to avoid relying on the observation window size K). That is, letting $\langle\cdot\rangle_\theta$ denote the exponential moving average

operation with a decay factor θ, we redefine the *second-order oscillation ratio* of a given parameter as:

$$\mathcal{R} = \frac{|\langle g'_k \rangle_\theta|}{\langle |g'_k| \rangle_\theta}, \text{ where } \langle g'_k \rangle_\theta = \theta * g'_{k-1} + (1-\theta) * g'_k. \quad (2)$$

Here $\langle g'_k \rangle_\theta$ and $\langle |g'_k| \rangle_\theta$ are respectively the EMA value of g'_k and $|g'_k|$. With a proper θ value close to 1, we can well approximate the effect of window-based smoothing (a smaller $\mathcal{R}$ also indicates a higher linearity), yet we have two additional benefits. First, with θ, we keep decaying the impact of old values modestly, the diagnosis results being more timely and accurate. Second, instead of maintaining all the historical values in the observation window, for each parameter we now only need to maintain a single value $\mathcal{R}$, thereby remarkably reducing the memory consumption.

With the above $\mathcal{R}$ metric, we can faithfully judge a parameter as predictable (i.e., updating in a linear manner) if $\mathcal{R} < T_\mathcal{R}$, where $T_\mathcal{R}$ is a predefined threshold close to 0. In this way (combining second-order difference with EMA), we can make accurate and efficient linearity diagnosis. We next elaborate how to conduct communication-efficient FL with this diagnosis method.

B. Speculative Parameter Updating with Predicted Gradient

After identifying the linearly-updating parameters, we need to manually approximate the realistic parameter updating effect without synchronizing them. To that end, we need to speculatively update the values of those parameters following the same linearity pattern as before (i.e., with the same *slope*). Note that, the gradient value of a linearly-updating parameter would be stable across different rounds; for such parameters, we can simply use the update of the *last round* as the per-round update to be applied in the future.

Speculative updating dictated by predictability mask. In particular, as discussed in Sec. III-A, sparsification decisions shall be made independently for each (scalar) parameter. Given that existing frameworks like TensorFlow [28] or PyTorch [29] do not support selective parameter training at such a fine granularity, we further design a technique called *masked replacement*. That is, each client maintains a mask vector called *predictability mask*, which is identical across different clients. Each bit of the predictability mask represents whether the corresponded parameter is now updated in a linear pattern (i.e., predictable in our sense) or not. We first refine all the parameters regularly within a round; for those parameters masked as predictable, after round completion, we replace their current value to the predicted one—calculated by persistently applying the previously-profiled per-round update.

Yet, a remaining problem is that, given that the linear parameter updating is not an everlasting pattern but occurs only in interleaved time periods each with an arbitrary length (as shown in Fig. 1), how do we know the desired time to terminate speculative updating and switch to regular synchronization? If we persistently enforce linear updating (with no synchronization), once the linear-updating period in standard FL setup terminates, the parameter updating trajectory we get would diverge from the realistic one, compromising the model training accuracy. Therefore, we need to keep sensing the true parameter updating trend in reality.

C. Local Error Feedback to Ensure Convergence Validity

To sense the true parameter updating trend, we choose to leverage the local updating results as a feedback signal. In the previous design, we simply replace the local value of each target parameter to the predicted one, regardless of what the replaced value is. Yet, given factors like mini-batch randomness and loss function irregularities, the predicted value is usually not exactly the same with the true value after synchronization—meaning that there is a non-zero gap between the two values, which we call *prediction error*. In fact, such errors provide valuable information on the true parameter updating pattern: if the linear pattern persists, that errors would be caused by mini-batch randomness and exhibit an oscillating pattern; otherwise, the errors for consecutive rounds would stably skew to a fixed direction.

Monitoring updating linearity with error feedback signal. Therefore, we can use the accumulated prediction error to judge whether the predicted linear pattern persists in reality. In particular, confronting heterogeneous parameter fluctuation extents, we set the expected per-round update as the normalization baseline. That is, let $\{e_{k+1}, e_{k+2}, ..., e_{k+l}\}$ denote the list of prediction errors since speculative updating phase launches at round k; here $e_r = \tilde{g}_r - g_k$ where $\tilde{g}_r$ is the true gradient if conducting regular synchronization at round r ($r = k+1, k+2, ..., k+l$), and g_k is the estimated per-round update used for that phase. Then we calculate

$$\mathcal{S} = \frac{|\sum_{r=k+1}^{k+l} e_r|}{|g_k|} \quad (3)$$

as the feedback signal. As long as $\mathcal{S} < T_\mathcal{S}$ where $T_\mathcal{S}$ is a modest threshold to control the error scale, we can deem that the linear updating pattern holds; otherwise we shall stop speculative updating and switch to regular updating.

Note that, however, obtaining the above prediction error (e_r) is not a free lunch. It is well-known that clients in FL differ in local data distributions [1], meaning that the local error on a client cannot represent the global one. Therefore, we have to conduct global synchronization to get the prediction errors. Obviously, it is unacceptable to keep monitoring the prediction error *after each round*, because the communication cost incurred would counteract the potential communication savings from spasification. That is, there exists a trade-off in setting up the error-check frequency: more frequent error aggregation can yield more timely reaction, but the cost is larger communication overhead.

We therefore choose a tentative strategy to setup the checking frequency: we assign each predictable parameter with a *no-checking* period, and error aggregation is conducted only after *no-checking period* expires. Moreover, the no-checking period of a parameter is adjusted based on the latest feedback

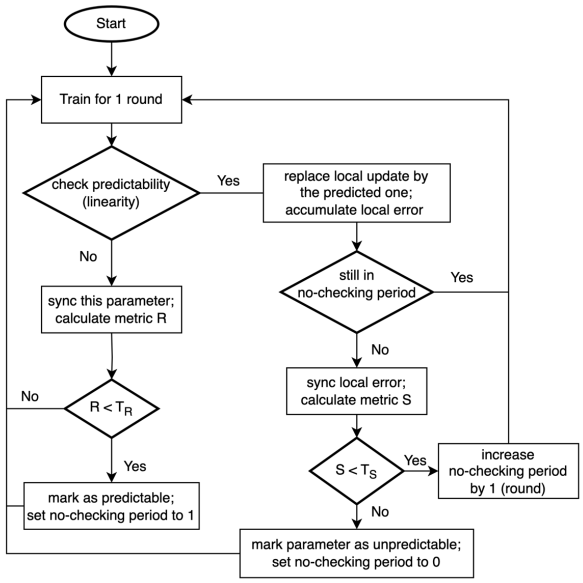


Fig. 3: Flow chart describing the workflow of FedSU.

signal $\mathcal{S}$: if $\mathcal{S}$ is still smaller than $T_{\mathcal{S}}$, we increase the no-checking period by one round; otherwise we reset the no-checking period to 0 and mask that parameter as unpredictable. In this way, we can effectively reduce the communication amount without compromising accuracy.

In Fig. 3, we summarize the overall workflow of FedSU. After each round, clients would check the linearity of each parameter based on the predictability mask. Those unpredictable parameters would be synchronized normally, after which we calculate their second-order oscillation ratio and accordingly update the predictability status (Sec. IV-A). Otherwise, for those predictable parameters within their respective no-checking period, we update their values based on the expected gradient without synchronization (Sec. IV-B). In particular, for those parameters, if the no-checking period expires after the current round, we would collect their local errors to update the no-checking period (Sec. IV-C).

D. Convergence Analysis

We further prove that FedSU can ensure convergence validity for general non-convex DL models. We begin by stating the smoothness and bounded gradient assumptions:

Assumption 1. (Smoothness) *The global loss function $F(\mathbf{x})$ is β-smooth convex, i.e., $\|\nabla F(\mathbf{x}) - \nabla F(\mathbf{y})\| \leq \beta \|\mathbf{x} - \mathbf{y}\|$.*

Assumption 2. (Bounded Gradient) *During the model training process, the model gradients are bounded as $\|\mathbf{g}\|^2 \leq \sigma^2$. In particular, for the j-th (scalar) parameter, its gradient bound is specifically $|g^j| \leq \sigma^j$, where $\sum_j (\sigma^j)^2 = \sigma^2$.*

Inspired by the perturbed iterate framework of [30], we define a virtual sequence $\{\tilde{\mathbf{x}}_k\}$ based on the true sequence[2] $\{\mathbf{x}_k\}$: $\tilde{\mathbf{x}}_0 := \mathbf{x}_0, \quad \tilde{\mathbf{x}}_{k+1} := \tilde{\mathbf{x}}_k - \eta_k \nabla F(\mathbf{x}_k) = \tilde{\mathbf{x}}_k - \eta_k \tilde{\mathbf{g}}_k$. Then, we can derive the following theorem:

Theorem 1. (Convergence Property) *Under Assumption 1 and Assumption 2, after running T iterations in FedSU, we have:*

$$\frac{1}{\sum_{k=1}^{T} \eta_k} \sum_{k=1}^{T} \eta_k \mathbb{E}[||\nabla F(\mathbf{x}_k)||^2] \leq \frac{4(F(\mathbf{x}_0) - F(\mathbf{x}^*))}{\sum_{k=1}^{T} \eta_k} + \frac{(4\sigma^2\beta^2 T_{\mathcal{S}}^2) \sum_{k=1}^{T} \eta_k^3}{\sum_{k=1}^{T} \eta_k} + \frac{(2\sigma^2\beta) \sum_{k=1}^{T} \eta_k^2}{\sum_{k=1}^{T} \eta_k}. \tag{4}$$

Proof. With Assumption 1, we have $F(\tilde{\mathbf{x}}_{k+1}) - F(\tilde{\mathbf{x}}_k) \leq \nabla F(\tilde{\mathbf{x}}_k)^T (\tilde{\mathbf{x}}_{k+1} - \tilde{\mathbf{x}}_k) + \frac{\beta}{2} \|\tilde{\mathbf{x}}_{k+1} - \tilde{\mathbf{x}}_k\|^2 = -\eta_k \nabla F(\tilde{\mathbf{x}}_k)^T g_k(\mathbf{x}_k) + \frac{\eta_k^2 \beta}{2} \|g_k(\mathbf{x}_k)\|^2$.

Taking the expectation at iteration k we have:

$$\begin{aligned}
&\mathbb{E}[F(\tilde{\mathbf{x}}_{k+1})] - F(\tilde{\mathbf{x}}_k) \\
\leq& -\eta_k \nabla F(\tilde{\mathbf{x}}_k)^T \mathbb{E}[g_k(\mathbf{x}_k)] + \frac{\eta_k^2 \beta}{2} \mathbb{E}\left[\|g_k(\mathbf{x}_k)\|^2\right] \\
=& -\eta_k \nabla F(\tilde{\mathbf{x}}_k)^T \nabla F(\mathbf{x}_k) + \frac{\eta_k^2 \beta}{2} \mathbb{E}\left[\|g_k(\mathbf{x}_k)\|^2\right] \\
=& -\frac{\eta_k}{2} \|\nabla F(\tilde{\mathbf{x}}_k)\|^2 - \frac{\eta_k}{2} \|\nabla F(\mathbf{x}_k)\|^2 \\
&+ \frac{\eta_k}{2} \|\nabla F(\tilde{\mathbf{x}}_k) - \nabla F(\mathbf{x}_k)\|^2 + \frac{\eta_k^2 \beta}{2} \mathbb{E}\left[\|g_k(\mathbf{x}_k)\|^2\right] \\
\leq& -\frac{\eta_k}{2} \|\nabla F(\tilde{\mathbf{x}}_k)\|^2 + \frac{\eta_k}{2} \|\tilde{\mathbf{x}}_k - \mathbf{x}_k\|^2 + \frac{\eta_k^2 \beta}{2} \mathbb{E}\left[\|g_k(\mathbf{x}_k)\|^2\right] \\
=& -\frac{\eta_k}{2} \left(\|\nabla F(\tilde{\mathbf{x}}_k)\|^2 + \beta^2 \|\mathbf{x}_k - \tilde{\mathbf{x}}_k\|^2\right) \\
&+ \eta_k \beta^2 \|\mathbf{x}_k - \tilde{\mathbf{x}}_k\| + \frac{\eta_k^2 \beta}{2} \mathbb{E}\left[\|g_k(\mathbf{x}_k)\|^2\right] \\
\leq& -\frac{\eta_k}{2} \left(\|\nabla F(\tilde{\mathbf{x}}_k)\|^2 + \beta^2 \|\mathbf{x}_k - \tilde{\mathbf{x}}_k\|^2\right) \\
&+ \eta_k \beta^2 \|\mathbf{x}_k - \tilde{\mathbf{x}}_k\| + \frac{\eta_k^2 \beta \sigma^2}{2}.
\end{aligned}$$

Taking the expectation before k, it yields

$$\begin{aligned}
&\mathbb{E}[F(\tilde{\mathbf{x}}_{k+1})] - \mathbb{E}[F(\tilde{\mathbf{x}}_k)] \leq \eta_k \beta^2 \mathbb{E}\left[\|\mathbf{x}_k - \tilde{\mathbf{x}}_k\|^2\right] \\
&+ \frac{\eta_k^2 \beta \sigma^2}{2} - \frac{\eta_k}{2} \mathbb{E}\left[\left(\|\nabla F(\tilde{\mathbf{x}}_k)\|^2 + \beta^2 \|\mathbf{x}_k - \tilde{\mathbf{x}}_k\|^2\right)\right].
\end{aligned} \tag{5}$$

Here, we first focus on $\mathbb{E}\|\mathbf{x}_k - \tilde{\mathbf{x}}_k\|^2$. For those predictable parameters, the gap is quantified by Assumption 3; for those unpredictable parameters updated under vanilla SGD, there is no expected gap without any FedSU interference. Therefore, let k_0 denote the round where the last no-checking period starts, then we have:

$$\mathbb{E}\|\mathbf{x}_k - \tilde{\mathbf{x}}_k\| = \mathbb{E}(\sum_j |x_k^j - \tilde{x}_k^j|^2) = \eta_k^2 \sum_j \mathbb{E}| \sum_{r=k_0}^{k-1} (g_r^j - \tilde{g}_r^j)|^2. \tag{6}$$

[2]For simplicity we skip the intra-round updating details and assume that there is only one iteration in each round. We note that this does not invalidate our analysis, because a parameter updating linearly when observed at a coarse time granularity would also follow a linear pattern when observed at a finer granularity. Our analysis thus also holds for general cases (except that the bounds in Assumption 1 and Assumption 2 have different scales).

Based on the no-checking condition in Eq. 3 and Assumption 2, we further have:

$$\mathbb{E}\|\mathbf{x}_k - \tilde{\mathbf{x}}_k\| = \sum_j \mathbb{E}|\sum_{r=k_0}^{k-1} \eta_r e_r^j|^2 \leq \sum_j \eta_k^2 T_{\mathcal{S}}^2 |g_{r_0}^j|^2 \leq \eta_k^2 T_{\mathcal{S}}^2 \sigma^2. \tag{7}$$

Applying the above inequality to Eq. 5, we have

$$\begin{aligned} &\mathbb{E}[F(\tilde{\mathbf{x}}_{k+1})] - \mathbb{E}[F(\tilde{\mathbf{x}}_k)] \leq \eta_k^2 \sigma^2 (\eta_k \beta^2 T_{\mathcal{S}}^2 + \frac{\beta}{2}) \\ &- \frac{\eta_k}{2} \mathbb{E}[(||\nabla F(\tilde{\mathbf{x}}_k)||^2 + \beta^2 ||\mathbf{x}_k - \tilde{\mathbf{x}}_k||^2)]. \end{aligned} \tag{8}$$

Then we can obtain

$$\begin{aligned} &\eta_k \mathbb{E}[(||\nabla F(\tilde{\mathbf{x}}_k)||^2 + \beta^2 ||\mathbf{x}_k - \tilde{\mathbf{x}}_k||^2)] \\ &\leq 2(\mathbb{E}[F(\tilde{\mathbf{x}}_k)] - \mathbb{E}[F(\tilde{\mathbf{x}}_{k+1})]) + \eta_k^2 \sigma^2 (2\eta_k \beta^2 T_{\mathcal{S}}^2 + \beta). \end{aligned} \tag{9}$$

Using the β-smooth property of $F(\mathbf{x})$, we have

$$\begin{aligned} &||\nabla F(\mathbf{x}_k)||^2 = ||\nabla F(\mathbf{x}_k) - \nabla F(\tilde{\mathbf{x}}_k) + \nabla F(\tilde{\mathbf{x}}_k)||^2 \\ &\leq 2||\nabla F(\mathbf{x}_k) - \nabla F(\tilde{\mathbf{x}}_k)||^2 + 2||\nabla F(\tilde{\mathbf{x}}_k)||^2 \\ &\leq 2\beta^2 ||\mathbf{x}_k - \tilde{\mathbf{x}}_k||^2 + 2||\nabla F(\tilde{\mathbf{x}})_k||^2. \end{aligned} \tag{10}$$

Combine with Eq. 9, we obtain

$$\begin{aligned} &\eta_k \mathbb{E}[||\nabla F(\mathbf{x}_k)||^2] \leq 2\eta_k \mathbb{E}[\beta^2 ||\mathbf{x}_k - \tilde{\mathbf{x}}_k||^2 + ||\nabla F(\tilde{\mathbf{x}})_k||^2] \\ &\leq 4(\mathbb{E}[F(\tilde{\mathbf{x}}_k)] - \mathbb{E}[F(\tilde{\mathbf{x}}_{k+1})]) + 2\eta_k^2 \sigma^2 (2\eta_k \beta^2 T_{\mathcal{S}}^2 + \beta). \end{aligned} \tag{11}$$

Summing up the inequalities for $k = 1, 2, ..., T$, we have

$$\begin{aligned} &\sum_{k=1}^{T} \eta_k \mathbb{E}[||\nabla F(\mathbf{x}_k)||^2] \leq 4(F(\mathbf{x}_0) - F(\mathbf{x}^*)) \\ &+ (4\sigma^2 \beta^2 T_{\mathcal{S}}^2) \sum_{k=1}^{T} \eta_k^3 + (2\sigma^2 \beta) \sum_{k=1}^{T} \eta_k^2. \end{aligned} \tag{12}$$

By dividing the summation of learning rates, we have:

$$\begin{aligned} &\frac{1}{\sum_{k=1}^T \eta_k} \sum_{k=1}^{T} \eta_k \mathbb{E}[||\nabla F(\mathbf{x}_k)||^2] \leq \frac{4(F(\mathbf{x}_0) - F(\mathbf{x}^*))}{\sum_{k=1}^T \eta_k} \\ &+ \frac{(4\sigma^2 \beta^2 T_{\mathcal{S}}^2) \sum_{k=1}^T \eta_k^3}{\sum_{k=1}^T \eta_k} + \frac{(2\sigma^2 \beta) \sum_{k=1}^T \eta_k^2}{\sum_{k=1}^T \eta_k}. \end{aligned}$$

□

Theorem 1 implies that model training under FedSU can converge when η_k satisfies:

$$\lim_{T\to\infty} \sum_{k=1}^{T} \eta_k = \infty \quad \text{and} \quad \lim_{T\to\infty} \frac{\sum_{k=1}^T \eta_k^2}{\sum_{k=1}^T \eta_k} = 0. \tag{13}$$

Therefore, we can set $\eta_k = \mathcal{O}(\frac{1}{\sqrt{T}})$ which satisfies the conditions in Eq. 13, and the model convergence can be guaranteed under FedSU.

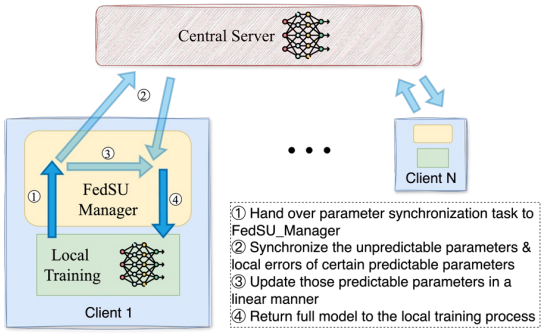


Fig. 4: System architecture with FedSU.

V. Implementation

Overview. As shown in Fig. 4, we have implemented FedSU with a Python module named `FedSU_Manager`, which wraps up all our FedSU algorithm details and makes them transparent to end users. Algorithm 1 summarizes the main workflow of FedSU. After each round, FL clients only need to call `FedSU_Manager.sync()` for model synchronization. `FedSU_Manager` maintains the predictability mask ($M_{\text{predictable}}$) as well as the no-checking mask ($M_{\text{no\_check}}$, which can be derived from the no-checking period) for each parameter, and further uses them to decide whether to conduct realistic transmission (following the workflow described in Fig. 3). Note that to avoid incurring extra communication cost, each client maintains its own copy of the two masks (identical across different clients since they are calculated from global parameters). Here the remote communication is conducted via RPyC (Remote Python Call) [31].

Handling system dynamicity. Note that in realistic FL scenarios, the clients may dynamically join or leave the FL process at any time. To ensure that FedSU can work smoothly against such participant dynamicity, for each new participant just joining the FL process, we let it download—besides the latest model—also the latest predictability mask and no-checking period information.

Complexity analysis. Typical edge devices for FL bear poor hardware capability, and we need to ensure that our FedSU implementation is resource efficient. Regarding the computation complexity, for each parameter the operations required to conduct linearity diagnosis and error feedback are both constant. Meanwhile, the additional memory overheads incurred by FedSU (i.e., the predictability mask and the local error) are proportional to the model size (acceptable because the model itself only consumes a small portion of the total memory—which is mostly consumed by the input data, feature-map and optimizer status [32]). That is, both the computation and space complexity can be captured by $O(|\mathbf{x}|)$. We will further evaluate the overhead of FedSU in Sec. VI-F.

VI. Evaluation

In this section, we systematically evaluate the performance of our FedSU algorithm. We first conduct an end-to-end performance comparison of FedSU, and then demonstrate the

Algorithm 1 FedSU Workflow

Require: $F_s, T_{\mathcal{R}}, T_{\mathcal{S}}$ ▷ F_s: the number of iterations in a round

Client: $i = 1, 2, ..., N$:

procedure CLIENTITERATE(k)
  $\mathbf{x}_k^i \leftarrow \mathbf{x}_{k-1}^i - \eta \mathbf{g}_k^i$ ▷ conduct regular local update
  if $k \mod F_s = 0$ **then**
    $\mathbf{x}_k^i \leftarrow$ `FedSU_Manager.`SYNC($\mathbf{x}_k^i$)

Central_Server:

procedure AGGREGATE_MODEL($\mathbf{x}_k^1, ..., \mathbf{x}_k^N$)
  return $\frac{1}{N}\sum_{i=1}^{N} \mathbf{x}_k^i$
procedure AGGREGATE_ERROR($\mathbf{e}_k^1, ..., \mathbf{e}_k^N$)
  return $\frac{1}{N}\sum_{i=1}^{N} \mathbf{e}_k^i$

FedSU_Manager:

Init: $\mathbf{e} = 0$, $M_{\text{predictable}} = 0$, $M_{\text{no\_check}} = 0$

procedure SYNC($\mathbf{x}$)
  $\hat{\mathbf{x}} \leftarrow \mathbf{x}$`.masked_select(`$\neg M_{\text{predictable}}$`)` ▷ $\hat{\mathbf{x}}$: parameters that are updated in a non-linear (unpredictable) manner
  $\hat{\mathbf{x}} \leftarrow$ `Central_Server.`AGGREGATE_MODEL($\hat{\mathbf{x}}$)
  $\hat{\mathbf{x}} \leftarrow \mathbf{x}$`.masked_fill(`$\neg M_{\text{predictable}}, \hat{\mathbf{x}}$`)` ▷ restore full size
  $\mathbf{e} \leftarrow \mathbf{e} + (\mathbf{x} - \acute{\mathbf{x}})$`.masked_select(`$\neg M_{\text{no\_check}}$`)` ▷ $\acute{\mathbf{x}}$ is the parameter value if all refined following the linear pattern
  $\mathbf{e} \leftarrow$ `Central_Server.`AGGREGATE_ERROR($\mathbf{e}$)
  $\mathbf{e} \leftarrow (\mathbf{x} - \acute{\mathbf{x}})$`.masked_fill(`$\neg M_{\text{no\_check}}, \mathbf{e}$`)`
  $\mathbf{x} \leftarrow M_{\text{predictable}} ? \acute{\mathbf{x}} : \hat{\mathbf{x}}$
  with $\mathbf{e}$, calculate $\mathcal{S}$ for checked parameters and update M_{check} and $M_{\text{predictable}}$ based on $T_{\mathcal{S}}$ (following Fig. 3)
  with $\hat{\mathbf{x}}$, calculate $\mathcal{R}$ for unpredictable parameters and update $M_{\text{predictable}}$ based on $T_{\mathcal{R}}$ (following Fig. 3)
  return $\mathbf{x}$

stability of our algorithm through sensitivity analysis and ablation studies, assessing the role and importance of each component of the algorithm. Finally, we present an evaluation of the overhead associated with the FedSU algorithm.

A. Experimental Setup

Hardware setup. We build an EC2 cluster with 128 `c6i.large` instances and one `c5a.8xlarge` instance. Each `c6i.large` instance has 2 vCPUs and 4GB RAM (similar to a smart phone), working as a FL client. Following the average network condition of FedScale [33]—a comprehensive FL benchmark including real-world mobile device measurements, we set the link bandwidth of each client to 13.7 Mbps (with the wondershaper [34] tool). Meanwhile, the `c5a.8xlarge` instance—with 32 vCPUs, 64GB RAM and 10 Gbps link bandwidth—works as the FL server. Meanwhile, to emulate participation dynamicity, in each round, we collect 70% of the clients returned the earliest.

Training Setup. The datasets we use in our experiment are EMNIST [35], FMNIST (Fashion-MNIST) [36] and CIFAR-10 [37]. In particular, to simulate non-IID data settings, we generated the local dataset distribution for each client based on a Dirichlet distribution [38]. The Dirichlet distribution controls the concentration parameter α for the combination of label classes ($\alpha \to \infty$ indicates IID data, while $\alpha = 0$ indicates that the dataset on each client contains only one label class). Specifically, we set $\alpha = 1$ for each client, emulating a modest[3] non-IID level. With the EMNIST dataset we train a CNN [41] model with two convolutional layers (kernel size 5×5) and two fully-connected layers; with the FMNIST dataset we train the ResNet-18 [42] model; with the CIFAR-10 dataset we train the DesNet-121 [43] model. These models all adopt the SGD optimizer, with the learning rate respectively set to 0.01, 0.001, and 0.01. The weight decay coefficient is 0.001 for each model. Meanwhile, the batch size of each iteration is 32 and each round has 50 iterations.

TABLE I: Time to reach the target accuracy for each model, together with the per-round time and number of rounds required.

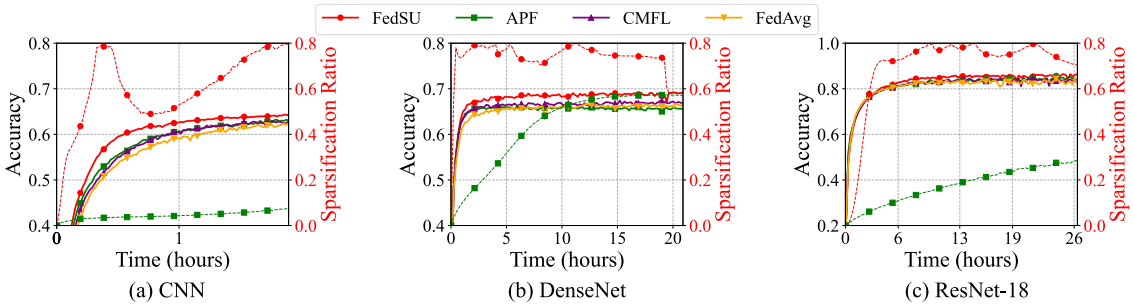
Model	Scheme	Per-round Time (s)	# of Rounds	Total Time (h)
CNN (0.60)	FedSU	7.23	264	0.53
	APF	12.04	265	0.89
	CMFL	13.05	248	0.90
	FedAvg	15.05	273	0.91
DenseNet (0.65)	FedSU	40.36	96	1.08
	APF	50.46	108	1.51
	CMFL	55.57	96	1.48
	FedAvg	60.34	174	2.92
ResNet-18 (0.85)	FedSU	109.19	283	8.58
	APF	144.96	379	15.26
	CMFL	145.38	379	15.31
	FedAvg	150.39	381	15.92

Algorithm Setup. With the above hardware and workload setup, we evaluate four algorithms: FedAvg, CMFL, APF and FedSU. FedAvg conducts full-model synchronization, and CMFL [11] and APF [12] are two typical sparsification algorithms previously introduced in Sec. II-B. In CMFL, we set the relevance threshold to the default value 0.8, meaning that updates with less-than-80% identical directions with the global one will not be transmitted. In APF, we set the stability threshold (used to eliminate the parameters that have already converged from transmission) to the default value 0.05. In our FedSU setup, we set the predictability threshold ($T_{\mathcal{R}}$) and the error-feedback threshold ($T_{\mathcal{S}}$) respectively to 0.01 and 1.0.

B. End-to-End Performance

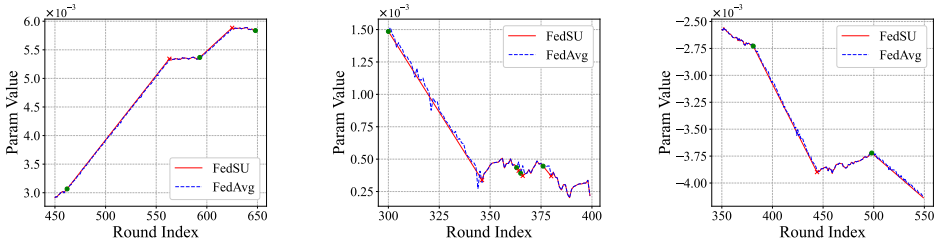
In Fig. 5, we show the time-to-accuracy curves of the three models under the four schemes above, accompanied by the communication compression ratio respectively under APF and FedSU. From Fig. 5, we can learn that FedSU can always make the most rapid accuracy improvement. In particular, Fig. 5 also shows that FedSU can attain a much higher communication compression ratio than APF. For example, when training the ResNet model, the average sparsification ratio under FedSU is 71.7%, whereas under APF that ratio is

[3]It is known that the model accuracy of FL would degrade under higher non-IID level [39]; our objective with FedSU is to reduce the communication amount while preserving (instead of improving) the training accuracy, thus we choose a typical non-IID level. That said, FedSU can be integrated with those methods aiming to enhancing FL accuracy over non-IID data (e.g., by changing the optimization function [40] or by data compensation [39]).



(a) CNN (b) DenseNet (c) ResNet-18

Fig. 5: Instantaneous time-to-accuracy curves when training CNN, DenseNet and ResNet-18 under different schemes. For APF and FedSU we also depict the instantaneous communication compression ratio (sparsification ratio) in dashed lines.



(a) CNN (b) DenseNet (c) ResNet-18

Fig. 6: Parameter evolution trajectories for a randomly-sampled parameter under FedSU as well as with regular synchronization like under FedAvg. Green dots and red crosses respectively represent the start and end points of the linear-updating periods of FedSU.

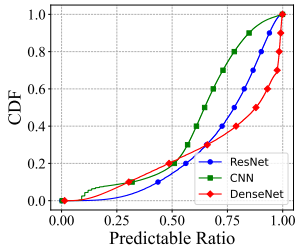


Fig. 7: Cumulative Distribution Function (CDF) of the predictable (linearly-updating) ratio when training the three models.

only 21.3%. This is consistent with our analysis in Sec. II-B: FedSU can exhaustively exploit the linear parameter evolution pattern for sparsified communication, of which the (static) converged pattern exploited by APF is only a special case.

We further resort to Table I to quantitatively compare the performance of FedSU against others. We first set a near-optimal accuracy target for the three models (0.6 for CNN, 0.65 for DenseNet, and 0.85 for ResNet-18), and then record the time required to reach such accuracy targets under the four schemes, accompanied with the number of rounds and the average per-round time. According to Table I, FedSU can substantially reduce the model training time of FL. Compared with the *second-best* scheme (APF), FedSU can attain a respective training speedup of 40.4%, 28.9% and 43.8% for the three models. In particular, for each model, the number of rounds required to reach the target accuracy under FedSU is similar[4] with that under FedAvg; we can thus confirm that the statistical (accuracy) performance of FedSU is not compromised by sparsification.

C. Microscopic Study

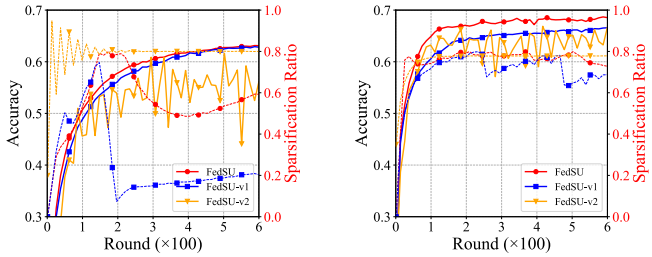
To better understand the performance benefit of FedSU, we further dive deep into the microscopic parameter trajectories during the FL training process. In Fig. 6, for each model, we depict the parameter evolution trajectories of a randomly-selected parameter during a randomly-selected period under FedSU—accompanied by the parameter trajectories during the same period if regular synchronizations are conducted like under FedAvg. As shown by Fig. 6, the parameter trajectories under our proposed FedSU algorithm can well approximate the vanilla ones under FedAvg. On the one hand, FedSU can accurately identify the training periods with strong updating linearity—thus attaining salient communication reduction; on the other hand, once such period expires, FedSU can swiftly sense that and switch to regular updating mode—thus with no hurt on the training accuracy.

We further resort to Fig. 7 to depict the overall linearity level (as diagnosed by FedSU) of all the parameters during the entire training process. We collect the proportions of *diagnosed-as-linear* periods for each parameter, and depict the cumulative distribution function (CDF) of such proportions over all the parameters of each model. Fig. 6 shows that a large portion of parameters have strong linearity during training: *more than 80%* parameters exhibit the linear pattern in *more than a half* of the model training time. This echos our previous motivating explorations in Sec. III-A and also corroborates the remarkable performance benefit of FedSU in Fig. 5.

D. Ablation Study

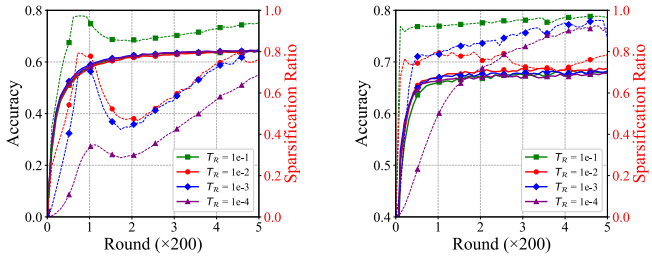
Note that our FedSU algorithm switches between normal updating and speculative updating based on two diagnoses: starting speculative updating when the linearity metric $\mathcal{R}$ decreases below a threshold ($T_{\mathcal{R}}$), and terminating it when the error metric $\mathcal{S}$ increases above a threshold ($T_{\mathcal{S}}$). Given the strong similarity between consecutive updates (as indicated in Fig. 2), we need to make it clear whether the two mechanisms are indeed indispensable. We then build two FedSU variants: FedSU-v1 (with linearity diagnosis but without error feedback)

[4]For the DenseNet model, our convergence speed in terms of the round number is even better than FedAvg. This is because we can avoid overfitting by updating with the predicted gradient and thus attain better generalization performance—a phenomena that also appeared in APF [12].



(a) CNN (b) DenseNet

Fig. 8: Ablation experiments with FedSU, FedSU-v1 (with linearity diagnosis but without error feedback) and FedSU-v2 (with neither linearity diagnosis nor error feedback).



(a) CNN (b) DenseNet

Fig. 9: FedSU performance with different linearity diagnosis thresholds ($T_{\mathcal{R}}$). The dashed lines show the sparsification ratio.

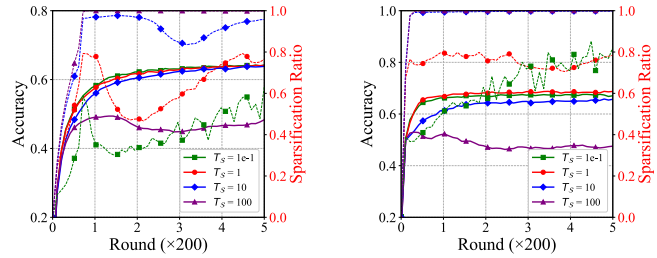
and FedSU-v2 (adopting neither linearity diagnosis nor error feedback). In FedSU-v1, for parameters diagnosed with a linear pattern, speculative updating would be conducted for a fixed period without error feedback; in FedSU-v2, each parameter randomly enters speculative updating with a preset probability, and the speculative updating period is also fixed.

In Fig. 8, we compare[5] FedSU with the two variants when training CNN and DenseNet models. The fixed speculative updating period and launching probability are respectively set to 43 (58) and 0.53% (0.81%) for CNN (DenseNet), following our measurements under standard FedSU. As shown in Fig. 8, the sparsification ratio curves of FedSU-v1 are in general remarkably lower than those of standard FedSU, and the accuracy curves are also slower (especially for DenseNet). Therefore, to fully exploit the communication compression opportunities without accuracy loss, we do need the error feedback mechanism in setting up the speculative updating period. Further, if we step forward by removing the linearity diagnosis, the accuracy performance would be substantially impacted: in Fig. 8, the accuracy curves of FedSU-v2 fluctuate drastically, much worse than those of FedSU. This confirms that speculative updating should be enforced only for those parameters with a strong linearity.

E. Sensitivity Analysis

Note that there are two key hyper-parameters in FedSU: $T_{\mathcal{R}}$ (linearity checking threshold introduced in Sec. IV-A) and $T_{\mathcal{S}}$ (error feedback threshold described in Sec. IV-C). Here we

[5]For simplicity and also due to space limitation, in our later experiments on ablation study and sensitivity analysis, we focus on the CNN and DenseNet model, and the general conclusions also hold for the ResNet model.



(a) CNN (b) DenseNet

Fig. 10: FedSU performance with different error-feedback thresholds ($T_{\mathcal{S}}$). The dashed lines show the sparsification ratio of each case.

explore how the two thresholds affect the FedSU performance, and the models we use are also CNN and DenseNet. In Fig. 9, we showcase the FedSU performance under different $T_{\mathcal{R}}$ values from 0.1 to 0.0001. From Fig. 9, we learn that the communication speedup from FedSU would be larger with a looser $T_{\mathcal{R}}$ threshold. Meanwhile, changing the $T_{\mathcal{R}}$ threshold does not significantly impact the accuracy performance of FedSU—thanks to the error feedback mechanism. That said, an over-large threshold ($T_{\mathcal{R}} = 0.1$) does lead to a slight accuracy degradation; our default setup $T_{\mathcal{R}} = 0.01$ can yield large communication speedup without accuracy loss.

In Fig. 10, we set the $T_{\mathcal{S}}$ threshold to four different values ranging from 0.1 to 100. Regarding the communication reduction effect, we find that the general trend resembles that of $T_{\mathcal{R}}$: the looser the threshold, the larger the communication speedup. Yet, we note that the training accuracy would deteriorate significantly under an improper $T_{\mathcal{S}}$ threshold: there is an accuracy loss of over 20% if setting $T_{\mathcal{S}}$ to 100. This is reasonable because $T_{\mathcal{S}}$ is a key hyper-parameter that directly bounds the accumulated gradient error brought by FedSU.

F. Overhead Analysis

TABLE II: The computation and memory overheads of FedSU.

Model	CNN	DenseNet	ResNet-18
Computation Time Inflation	0.034 s	0.916 s	1.258 s
Computation Time Inflation Ratio	0.47%	2.15%	1.01%
Memory Inflation	65 MB	70 MB	128 MB
Memory Inflation Ratio	3.75%	2.68%	8.27%

We further check the computation and memory overheads of FedSU. As explained in Sec. V, both the computation and memory overhead of FedSU are quite limited. In Table II, we measure the increase in per-round time and memory consumption after adopting FedSU. It shows that the computation time inflation ratio is always less than 2.15%, and the memory inflation ratio is also rather limited (less than 10%). In sum, the memory and computation overheads of FedSU are acceptable when compared with its performance benefit.

VII. Conclusion

In this work, we devise FedSU, a novel approach to mitigate the communication overhead in federated learning. Motivated by the observation that model parameters often exhibit a linear evolution pattern during the training process, FedSU can efficiently identify the linearly-updating parameters and use the predicted updates for model refinement without conducting realistic synchronization. To ensure convergence validity, FedSU further incorporates an error-feedback mechanism to timely react to the instantaneous training status. Extensive experiments confirm that FedSU can substantially reduce the communication volume without accuracy degradation.

Acknowledgement

The research was supported in part by the National Natural Science Fundation of China (NSFC) grant 62202300, a RGC RIF grant under contract R6021-20, RGC TRS grant under contract T43-513/23N-2, RGC CRF grants under contracts C7004-22G, C1029-22G and C6015-23G, and RGC GRF grants under contracts 16200221, 16207922 and 16207423. Chen Chen is the corresponding author.

References

[1] H. B. McMahan, E. Moore, D. Ramage, S. Hampson *et al.*, "Communication-efficient learning of deep networks from decentralized data," *arXiv preprint arXiv:1602.05629*, 2016.

[2] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *PMLR AISTATS*, 2017.

[3] C. Zhang, Y. Xie, H. Bai, B. Yu, W. Li, and Y. Gao, "A survey on federated learning," *Knowledge-Based Systems*, vol. 216, p. 106775, 2021.

[4] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "Qsgd: Communication-efficient sgd via gradient quantization and encoding," *Advances in neural information processing systems*, vol. 30, 2017.

[5] A. Reisizadeh, A. Mokhtari, H. Hassani, A. Jadbabaie, and R. Pedarsani, "Fedpaq: A communication-efficient federated learning method with periodic averaging and quantization," in *PMLR AISTATS*, 2020.

[6] D. Jhunjhunwala, A. Gadhikar, G. Joshi, and Y. C. Eldar, "Adaptive quantization of model updates for communication-efficient federated learning," in *IEEE ICASSP*, 2021.

[7] T. Vogels, S. P. Karimireddy, and M. Jaggi, "Powersgd: Practical low-rank gradient compression for distributed optimization," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[8] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing, "Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on GPU Clusters," in *USENIX ATC*, 2017.

[9] D. Yu, H. Zhang, W. Chen, J. Yin, and T.-Y. Liu, "Large scale private learning via low-rank reparametrization," in *PMLR ICML*, 2021.

[10] K. Hsieh, A. Harlap, N. Vijaykumar, D. Konomis, G. R. Ganger, P. B. Gibbons, and O. Mutlu, "Gaia:geo-distributed machine learning approaching LAN speeds," in *USENIX NSDI*, 2017.

[11] W. Luping, W. Wei, and L. Bo, "Cmfl: Mitigating communication overhead for federated learning," in *IEEE ICDCS*, 2019.

[12] C. Chen, H. Xu, W. Wang, B. Li, B. Li, L. Chen, and G. Zhang, "Communication-efficient federated learning with adaptive parameter freezing," in *IEEE ICDCS*, 2021.

[13] M. Ye, W. Shen, E. Snezhko, V. Kovalev, P. C. Yuen, and B. Du, "Vertical federated learning for effectiveness, security, applicability: A survey," *arXiv preprint arXiv:2405.17495*, 2024.

[14] J. Wu, S. Drew, and J. Zhou, "Fedle: Federated learning client selection with lifespan extension for edge iot networks," in *IEEE ICC*, 2023.

[15] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[16] A. Kejariwal, X. Tian, M. Girkar, W. Li, S. Kozhukhov, U. Banerjee, A. Nicolau, A. V. Veidenbaum, and C. D. Polychronopoulos, "Tight analysis of the performance potential of thread speculation using spec cpu 2006," in *ACM PPoPP*, 2007.

[17] T. Yang, G. Andrew, H. Eichner, H. Sun, W. Li, N. Kong, D. Ramage, and F. Beaufays, "Applied federated learning: Improving google keyboard query suggestions," *arXiv preprint arXiv:1812.02903*, 2018.

[18] "Global Internet Condition (2019)." https://www.atlasandboots.com/remote-jobs/countries-with-the-fastest-internet-in-the-world.

[19] Z. Gan, C. Chen, J. Zhang, G. Zeng, Y. Zhu, J. Zhao, Q. Chen, and M. Guo, "PAS: Towards Accurate and Efficient Federated Learning with Parameter-Adaptive Synchronization," in *IWQoS*, 2024.

[20] L. Luo, J. Nelson, L. Ceze, A. Phanishayee, and A. Krishnamurthy, "Parameter hub: a rack-scale parameter server for distributed deep neural network training," in *ACM SoCC*, 2018.

[21] F. Liang, Z. Zhang, H. Lu, V. Leung, Y. Guo, and X. Hu, "Communication-efficient large-scale distributed deep learning: A comprehensive survey," *arXiv preprint arXiv:2404.06114*, 2024.

[22] Z. Gan, C. Chen, J. Zhang, G. Zeng, Y. Zhu, J. Zhao, Q. Chen, and M. Guo, "Pas: Towards accurate and efficient federated learning with parameter-adaptive synchronization," in *IEEE IWQoS*, 2024.

[23] D. A. Jiménez and C. Lin, "Neural methods for dynamic branch prediction," *ACM TOCS*, vol. 20, no. 4, pp. 369–397, 2002.

[24] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn, "Evidence-based static branch prediction using machine learning," *ACM TOPLAS*, vol. 19, no. 1, pp. 188–222, 1997.

[25] D. C. Montgomery, E. A. Peck, and G. G. Vining, *Introduction to linear regression analysis*. John Wiley & Sons, 2021.

[26] G. A. Seber and A. J. Lee, *Linear regression analysis*. John Wiley & Sons, 2012.

[27] Å. Björck, "Least squares methods," *Handbook of numerical analysis*, vol. 1, pp. 465–652, 1990.

[28] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: a system for large-scale machine learning," in *USENIX OSDI*, 2016.

[29] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *NeurIPS*, vol. 32, 2019.

[30] H. Mania, X. Pan, D. Papailiopoulos, B. Recht, K. Ramchandran, and M. I. Jordan, "Perturbed iterate analysis for asynchronous stochastic optimization," *SIAM Journal on Optimization*, vol. 27, no. 4, pp. 2202–2229, 2017.

[31] "RPyC," https://rpyc.readthedocs.io/en/latest/.

[32] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *IEEE MICRO*, 2016.

[33] F. Lai, Y. Dai, S. Singapuram, J. Liu, X. Zhu, H. Madhyastha, and M. Chowdhury, "Fedscale: Benchmarking model and system performance of federated learning at scale," in *PMLR ICML*, 2022.

[34] "wondershaper," https://github.com/magnific0/wondershaper, 2020.

[35] G. Cohen, S. Afshar, J. Tapson, and A. e van Schaik, "Emnist: an extension of mnist to handwritten," *Proceedings of the IEEE*, vol. 4322, 2017.

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

[37] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[38] T.-M. H. Hsu, H. Qi, and M. Brown, "Measuring the effects of non-identical data distribution for federated visual classification," *arXiv preprint arXiv:1909.06335*, 2019.

[39] Y. Zhao, M. Li, L. Lai, N. Suda, D. Civin, and V. Chandra, "Federated learning with non-iid data," *arXiv preprint arXiv:1806.00582*, 2018.

[40] T. Li, A. K. Sahu, M. Zaheer, M. Sanjabi, A. Talwalkar, and V. Smith, "Federated optimization in heterogeneous networks," *Proceedings of Machine learning and systems*, vol. 2, pp. 429–450, 2020.

[41] 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.

[42] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *IEEE CVPR*, 2016.

[43] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *IEEE CVPR*, 2017.