Synchronize Only the Immature Parameters: Communication-Efficient Federated Learning By Freezing Parameters Adaptively

Chen Chen, *Member, IEEE*, Hong Xu, *Senior Member, IEEE*, Wei Wang, *Member, IEEE*, Baochun Li, *Fellow, IEEE*, Bo Li, *Fellow, IEEE*, Li Chen, *Member, IEEE*, and Gong Zhang, *Member, IEEE*

***Abstract*—Federated learning allows edge devices to collaboratively train a global model without sharing their local private data. Yet, with limited network bandwidth at the edge, communication often becomes a severe bottleneck. In this article, we find that it is unnecessary to always synchronize the full model in the entire training process, because many parameters already become mature (i.e., stable) prior to model convergence, and can thus be excluded from later synchronizations. This allows us to reduce the communication overhead without compromising the model accuracy. However, challenges are that the local parameters excluded from global synchronization may diverge on different clients, and meanwhile some parameters may stabilize only temporally. To address these challenges, we propose a novel scheme called Adaptive Parameter Freezing (APF), which fixes (freezes) the non-synchronized stable parameters in intermittent periods. Specifically, the freezing periods are tentatively adjusted in an additively-increase and multiplicatively-decrease manner—depending on whether the previously-frozen parameters remain stable in subsequent iterations. We also extend APF into APF# and APF++, which freeze parameters in a more aggressive manner to achieve larger performance benefit for large complex models. We implemented APF and its variants as Python modules with PyTorch, and extensive experiments show that APF can reduce data transfer amount by over 60%.**

***Index Terms*—Distributed computing, federated learning, machine learning algorithms.**

Manuscript received 18 July 2022; revised 31 December 2022; accepted 24 January 2023. Date of publication 3 February 2023; date of current version 15 May 2024. This work was supported in part by the National Natural Science Fundation of China (NSFC) under Grant 62202300, in part by Shanghai Pujiang Program under Grant 22PJ1404600, in part by RGC RIF under Grant R6021-20, in part by RGC GRF under Grants 16209120, 16200221, 16213120, 16202121 and 11209520, and in part by CUHK under Grants 4937007, 4937008, 5501329, and 5501517. Recommended for acceptance by A. C. Zhou. *(Corresponding author: Bo Li.)*

Chen Chen is with the John Hopcroft Center for Computer Science, Shanghai Jiao Tong University, Shanghai 200240, China (e-mail: chen-chen@sjtu.edu.cn).

Hong Xu is with the Department of Computer Science and Engineering, Chinese University of Hong Kong, Central Ave, Hong Kong (e-mail: hongxu@cuhk.edu.hk).

Wei Wang and Bo Li are with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong (e-mail: weiwa@cse.ust.hk; bli@cse.ust.hk).

Baochun Li is with the Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON M5S, Canada (e-mail: bli@ece.toronto.edu).

Li Chen is with the Zhongguancun Laboratory, Beijing 100081, China (e-mail: lichen@zgclab.edu.cn).

Gong Zhang is with the Theory Lab, Huawei Hong Kong Research Center, Pak Shek Kok, Hong Kong (e-mail: nicholas.zhang@huawei.com).

Digital Object Identifier 10.1109/TPDS.2023.3241965

I. Introduction

FEDERATED learning (FL) [32], [40] emerges as a promising paradigm that allows edge clients (e.g., mobile and IoT devices) to collaboratively train a model without sharing their local private data. In FL, there is a central server maintaining the global model, and edge devices synchronize their model updates in communication *rounds*. However, the Internet connections have limited bandwidth [1], which makes the parameter synchronization between the edge and server an inherent bottleneck that severely prolongs the training process.

A rich body of research works have been proposed to reduce the communication amount for distributed model training. For example, some works propose to *quantize* the model updates into fewer bits [5], [32], [49], and some other works propose to *sparsify* local updates by filtering out certain values [20], [27], [53], [55]. Nonetheless, those works assume that all the model parameters should be synchronized indiscriminately in all the communication rounds, which we find, however, is unnecessary. In our testbed measurements, many parameters become "mature" (i.e., *stable*) long before the model convergence; after these parameters reach their optimal values, their subsequent updates are random oscillations with no substantial changes, and can be excluded without harming the model accuracy.

Therefore, an intuitive question we want to explore is, can we reduce FL communication amount by no longer synchronizing those stabilized parameters? We first propose a metric called *effective perturbation* to identify the stabilized parameters—by quantifying how closely the consecutive updates of a parameter follow an oscillating pattern. A key challenge, then, is to treat the non-synchronized parameters properly—with preserved model convergence validity.

Delving into that regard, we empirically find that some strawman solutions however do not work well. A straightforward method—having the stabilized parameters updated only locally—fails to guarantee convergence, because the local updates on different clients may diverge, causing model inconsistency. Another method that permanently fixes (i.e., freezes) the stabilized parameters can ensure model consistency, but it still hurts the model convergence accuracy. This is because some parameters may stabilize only *temporally* during the training process; after being frozen prematurely, they are unable to reach the true optima.

Based on the above explorations, we design a novel mechanism, *Adaptive Parameter Freezing* (APF), with the objective of reducing communication volume while guaranteeing convergence validity. The key design philosophy is to tentatively freeze the stabilized parameters as guided by a feedback control loop. Under APF, each stable parameter is frozen for a certain period and then unfrozen for regular updating. The length of such freezing period is not fixed but adjusted adaptively—*additively increased* or *multiplicatively decreased* based on whether that parameter is still stable after being updating regularly. This way, APF enables temporarily-stable parameters to resume training in a timely manner, while keeping the converged parameters unsynchronized for most of the time. Our theoretical analysis further confirms that APF can guarantee convergence validity for general non-convex models.

Furthermore, for large models that are over-parameterized [43], [65], we have empirically observed that some parameters may not be stable even after model convergence (due to irregular landscapes). This may largely limit the communication compression benefit of APF. Considering the statistical redundancy of those complex neural network models, we further propose two APF extensions, APF# and APF++, which can freeze parameters in a more aggressive manner. The former (APF#) forces unstable parameters to be frozen for one round with a fixed probability (similar to Dropout [24], [33], [52]), and the latter (APF++) lets that freezing probability and length gradually increase during training, so as to exploit statistical redundancy for more communication-reduction benefits.

We have implemented APF (including the extensions) as a pluggable Python module named `APF_Manager`, atop the PyTorch framework. The module takes over the vanilla updates of edge clients and communicates only the non-frozen components with the server. Meanwhile, it also maintains the freezing period of each parameter based on the metric of effective perturbation. Our evaluation on Amazon EC2 platform demonstrates that, APF can reduce the overall transmission volume by over 60%—with comparable or even-better generalization performance than vanilla FL. Meanwhile, it outperforms a series of competing communication compression methods in the literature, with good hyper-parameter robustness and low overheads.

Our contributions can be summarized as follows:

- To our knowledge, we are the first that discover the opportunity to reduce the communication volume of FL by not synchronizing the stabilized parameters.
- After exploring candidate solutions, we propose APF, which attains large communication compression with convergence validity preserved, as supported by both empirical observations and theoretical analysis.
- We further propose APF# and APF++ that can attain even larger communication compression benefit with more aggressive parameter freezing methods.
- We have implemented APF in PyTorch, and verified its effectiveness with extensive testbed experiments.

II. Background

A. Federated Learning: The Basics

Deep learning techniques based on neural network models have tremendously improved the state-of-the-art performance of many modern applications, including image classification [23], [33] and language processing [15], [18]. However, in many real-world scenarios, the training samples are privacy-sensitive and dispersed on distributed edge clients, like cellphones and IoT equipments. To train models without centralizing such private data, an increasingly popular technique is federated learning (FL) [8], [32], [40]. In FL, there is a central server that maintains the global model, and clients periodically communicate with the server to pull the latest model parameters and, after local refinements for multiple iterations, push back the updates for global aggregation.

However, communication between clients and the FL server has become a severe performance bottleneck [7], [32], [40]. As an enabling technique that supports apps like Google Gboard with global user [63], FL entails that edge clients need to communicate with the server through the Internet, where the average bandwidth is less than 10*Mbps* globally [1]. In our testbed measurements (see Section VII-A for detailed setup), when training LeNet-5 (a classical convolutional neural network for image classification [34]) or ResNet-18 under a FL setup with 50 clients, we find that over half of the training time is spent on parameter communication. Therefore, it is of great significance to reduce the communication volume in FL.

B. Prior Arts and Their Limitations

To reduce the total transmission volume in distributed machine learning, there have been a bewildering array of research works [5], [12], [16], [20], [22], [25], [27], [29], [32], [46], [49], [50], [53], [67]. Among them the two most typical methodologies are *quantization* and *sparsification*.

Quantization [5], [49], [59] works by using lower bits to represent the transmitted gradients which are originally represented by 32 bits. For example, Seide et al. [49] proposed to aggressively quantize the gradients into only 1 bit, and achieved a $10\times$ communication speedup. A later work, QSGD [5], sought to balance the trade-off between accuracy and gradient compressing ratio, and provided a family of quantization schemes. Wen et al. [59] developed another communication compression scheme named TernGrad, which quantizes gradients into three levels $\{-1, 0, 1\}$.

Sparsification aims to reduce the number of elements to transmit in each iteration—by selecting only a "significant" portion of the original content. Compared with quantization, sparsification has the potential to achieve a much higher compression level. In designing sparsification solutions, a key challenge is how to identify the *significant* gradient components. For example, Storm [53] and Gaia [27] proposed to only send gradients whose absolute or relative values were larger than a given threshold. Dryden et al. [20] chose to separately select a fraction of positive and negative gradients for transmission—with the objective to meet a given compression ratio while preserving the expected gradient values. Additionally, CMFL [55] sought to report a portion of the initial gradients that were consistent enough with the global one in the positive/negative directions. Those methods have proven effective in reducing the communication amount.

Limitation of Existing Practice. However, the above sparsification methods are still far from optimal, because there exists a remarkable deficiency in their criteria to identify the so-called "significant" gradients. In those works, the significant gradients are identified purely with instantaneous information (e.g., by comparing the values or directions of different gradients within each individual round), but are blind to high-level model convergence status. In fact, users primarily care about model convergence rather than the exact parameter values; sometimes (as shown later in Section III) the updates of certain parameters are not needed by the long-term model convergence—regardless of their gradient magnitude. In that case, we can try eliminate their gradient synchronization to further reduce overall communication volume, even though their gradients are "significant" according to traditional criteria (e.g., in aspects of quantity or consistency). Obviously, existing sparsification methods cannot leverage such opportunity for better performance.

In this work, we ask a bold question: Is it necessary to keep updating all the parameters during the entire model training process? Intuitively, if there exist some parameters that no longer require being updated, we can fundamentally eliminate the communication cost of synchronizing them later. Next, we will explore the feasibility of that idea with both theoretical analysis and testbed measurements.

III. Motivating Exploration

In this section, we focus on parameter evolutions in model training, identifying an intriguing parameter stabilizing pattern which can be exploited for communication reduction. We first confirm its existence with theoretical analysis and empirical observations, and then conduct a series of in-depth study on its microscopic characteristics.

A. Parameter Evolution During Training

Given a neural network model, the objective of model training is to find the optimal parameters $\mathbf{x}^\star$ that can minimize the global loss function, $F(\mathbf{x})$. A common algorithm to find $\mathbf{x}^\star$ is *stochastic gradient descent* (SGD) [34], which works in an iterative manner. In iteration k, the model parameter $\mathbf{x}_k$ is updated with a gradient $g_{\xi_k}(\mathbf{x})$ calculated from a random sample batch ξ_k, i.e., $\mathbf{x}_{k+1} = \mathbf{x}_k - \eta g_{\xi_k}(\mathbf{x}_k)$.

In this work, we are curious on how the sequence of $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_k, \ldots\}$ evolves during the model training process. Our analysis is based on two common assumptions: strong convexity and bounded gradients.

Assumption 1. (Strong Convexity.) The global loss function $F(\mathbf{x})$ is μ-strongly convex, i.e.,

$$F(\mathbf{y}) \geq F(\mathbf{x}) + \nabla F(\mathbf{x})^T(\mathbf{y}-\mathbf{x}) + \frac{\mu}{2}\|\mathbf{y}-\mathbf{x}\|^2.$$

An equivalent form of $\mu-$strongly convexity is

$$(\nabla F(\mathbf{x}) - \nabla F(\mathbf{y}))^T(\mathbf{x}-\mathbf{y}) \geq \mu\|\mathbf{x}-\mathbf{y}\|^2.$$

Assumption 2. (Bounded Gradient.) The stochastic gradient calculated from a mini-batch ξ is bounded as $\mathbb{E}\|\mathbf{g}_\xi(\mathbf{x})\|^2 \leq \sigma^2$.

Based on the above assumptions, we can derive the following theorem[1]:

Theorem 1. Suppose $F(\mathbf{x})$ is μ-strongly convex, σ^2 is the upper bound of the variance of $\|\nabla F(\mathbf{x})\|^2$, then there exist two constants $A = 1 - 2\mu\eta$ and $B = \frac{\eta\sigma^2}{2\mu}$, such that

$$\mathbb{E}(\|\mathbf{x}_k - \mathbf{x}^\star\|^2) \leq A^k\|\mathbf{x}_0 - \mathbf{x}^\star\|^2 + B.$$

Proof.

$$\begin{aligned}\|\mathbf{x}_{k+1} - \mathbf{x}^\star\|^2 &= \|\mathbf{x}_k - \mathbf{x}^\star + \mathbf{x}_{k+1} - \mathbf{x}_k\|^2\\ &= \|\mathbf{x}_k - \mathbf{x}^\star - \eta\mathbf{g}_{\xi_k}(\mathbf{x}_k)\|^2\\ &= \|\mathbf{x}_k - \mathbf{x}^\star\|^2 - 2\eta\langle\mathbf{x}_k - \mathbf{x}^\star, \mathbf{g}_{\xi_k}(\mathbf{x}_k)\rangle + \eta^2\|\mathbf{g}_{\xi_k}(\mathbf{x}_k)\|^2.\end{aligned}$$

Taking expectation conditioned on $\mathbf{x}_k$ we can obtain:

$$\begin{aligned}\mathbb{E}_{\mathbf{x}_k}(\|\mathbf{x}_{k+1} - \mathbf{x}^\star\|^2) = \|\mathbf{x}_k - \mathbf{x}^\star\|^2 - 2\eta\langle\mathbf{x}_k - \mathbf{x}^\star, \nabla F(\mathbf{x}_k)\rangle\\ + \eta^2\mathbb{E}_{\mathbf{x}_k}(\|\mathbf{g}_{\xi_k}(\mathbf{x}_k)\|^2).\end{aligned}$$

Given that $F(\mathbf{x})$ is $\mu-$strongly convex, we have

$$\begin{aligned}\langle\nabla F(\mathbf{x}_k), \mathbf{x}_k - \mathbf{x}^\star\rangle &= \langle\nabla F(\mathbf{x}_k) - \nabla F(\mathbf{x}^\star), \mathbf{x}_k - \mathbf{x}^\star\rangle\\ &\geq \mu\|\mathbf{x}_k - \mathbf{x}^\star\|^2.\end{aligned}$$

Applying total expectation, we can obtain

$$\begin{aligned}\mathbb{E}(\|\mathbf{x}_{k+1} - \mathbf{x}^\star\|^2) &= \mathbb{E}\|\mathbf{x}_k - \mathbf{x}^\star\|^2 - 2\eta\langle\mathbb{E}(\mathbf{x}_k - \mathbf{x}^\star), \nabla F(\mathbf{x}_k)\rangle\\ &\quad + \eta^2\mathbb{E}(\|\mathbf{g}_{\xi_k}(\mathbf{x}_k)\|^2)\\ &\leq (1 - 2\mu\eta)\mathbb{E}\|\mathbf{x}_k - \mathbf{x}^\star\|^2 + \eta^2\sigma^2.\end{aligned}$$

Recursively applying the above and summing up the resulting geometric series gives

$$\begin{aligned}\mathbb{E}(\|\mathbf{x}_k - \mathbf{x}^\star\|^2) &\leq (1-2\mu\eta)^k\|\mathbf{x}_0 - \mathbf{x}^\star\|^2 + \sum_{j=0}^{k-1}(1-2\mu\eta)^j\eta^2\sigma^2\\ &\leq (1-2\mu\eta)^k\|\mathbf{x}_0 - \mathbf{x}^\star\|^2 + \frac{\eta\sigma^2}{2\mu}.\end{aligned}$$

This completes the proof.

As implied by Theorem 1, the training gap is determined by two components: a exponentially-decaying "bias" term related to parameter initialization, and a stable "noise" term related to the gradient variance bound which gradually dominates as training proceeds. This is consistent with previous observations that the model parameters first go through a *transient* phase and then a *stationary* phase [42], [66]: In the transient phase, $\mathbf{x}$ approaches $\mathbf{x}^\star$ exponentially fast in the number of iterations, and in the stationary phase, $\mathbf{x}$ oscillates around $\mathbf{x}^\star$. Therefore, model parameters should change remarkably in the beginning

1.This is not our key contribution and similar theorems have already been derived in existing works [11], [41]. We put it here primarily for clarity; besides, our theorem form is more neat than existing ones.

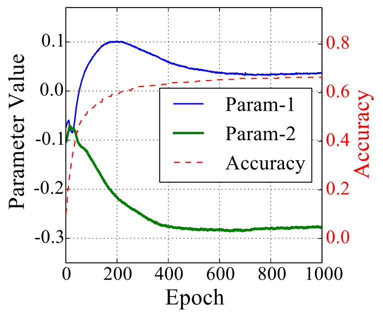


Fig. 1. Evolution of two randomly selected parameters during LeNet-5 training (with the test accuracy for reference.)

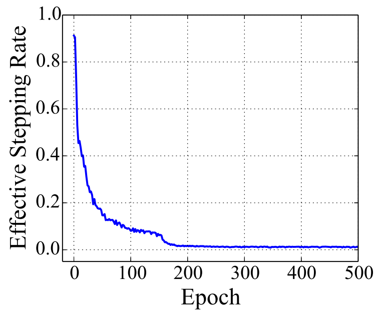


Fig. 2. Variation of the average *effective perturbation* of all the parameters during the LeNet-5 training process.

and then gradually stabilize. We further confirm that with testbed measurements.

Empirical Observations. We train *LeNet-5* [34] locally on the CIFAR-10 dataset with a batch size of 100, and Fig. 1 shows the values of two randomly sampled scalar parameters after each epoch, along with the test accuracy[2] for reference. In the figure, the two parameters change significantly in the beginning, accompanied by a rapid rise in the test accuracy; then, as the accuracy curve plateaus, they gradually *stabilize* after around 200 and 300 epochs, respectively.

We note that those stabilized parameters provision a promising opportunity to mitigate FL communication bottleneck. In current FL practices [32], [40], parameters are still updated regularly even after they stabilize. Although the updates of stabilized parameters are mostly oscillatory random noises, transmitting them still consumes the same communication bandwidth as before. Such a communication cost is in fact unnecessary, because machine learning algorithms are in general resilient against small parameter perturbations [17], [21], [27]. This can also be confirmed by the success of recent techniques like Dropout [?], [24], [52] and Stochastic Depth [28]. Thus, to reduce communication cost in FL, as an intuitive method, we can try avoid synchronizing such stabilized parameters.

Yet, how to identify the so-called *stable* parameters? And how common is the resource wastage caused by transmitting their oscillating gradient? We next quantitatively analyze the parameter stability characteristics over the entire model training process through testbed measurements.

B. Deep Dive Analysis on Parameter Stability

In this subsection, we propose a metric to quantify the parameter stability level, and dive deep into the inner

1) Identifying Stable Parameters: To see if the parameter variation pattern in Fig. 1 generally holds for other parameters, we conduct a statistical analysis of all the LeNet-5 parameters over the entire training process.

Regarding gradient statistics, several metrics have already been proposed in the literature to help setup certain training hyper-parameters. For example, to instruct the batch size setup, Qiao et. al [47] and McCandlish et. al [39] proposed a metric called Gradient Noise Scale (GNS), which calculates the consistent level among gradients of different samples. Meanwhile, to instruct the learning rate setup, Johnson et. al [30] proposed to calculate the Gradient Variance (GR) metric, and Kazuki et. al further resorted to a second-order metric [44]. Nonetheless, those metrics do not meet our demand in this article: We care not about the gradient variance or gradient noise scale, but on the parameter stability level, i.e., to what extent the gradients from different iterations oscillate around zero. To that end, we devise a new gradient metric called *effective perturbation*.

Effective perturbation is defined over an observation window containing a certain amount of consecutive model updates in the recent past. Formally, let $\mathbf{u}_k = \mathbf{x}_k - \mathbf{x}_{k-1}$ represent parameter update in iteration[3] k, and $\mathcal{S}$ be the number of recent iterations the observation window contains, then $\mathcal{P}_k$—the *effective perturbation* at iteration k—can be defined as:

$$\mathcal{P}_k = \frac{\| \sum_{i=k-\mathcal{S}+1}^{k} \mathbf{u}_i \|}{\sum_{i=k-\mathcal{S}+1}^{k} \|\mathbf{u}_i\|}. \tag{1}$$

Clearly, $\mathcal{P}_k$ represents how a parameter's trajectory follows the zigzagging pattern, i.e., how the consecutive updates counteract with each other. The more stable a parameter is, the more conflicting its consecutive updates are, and the smaller its effective perturbation is. If all the model updates are of the same direction, $\mathcal{P}_k$ would be 1; if any two consecutive model updates well counteract each other, then $\mathcal{P}_k$ would be 0. Thus, with a small threshold on effective perturbation, we can identify those stabilized parameters effectively.

We then look at the average effective perturbation of all the parameters during the LeNet-5 training process, as depicted in Fig. 2, where the observation window W spans one epoch (i.e., 500 updates). We observe that the average effective perturbation decays rapidly at first and then quite slowly after the model converges at around epoch-200 (see the accuracy result in Fig. 1), suggesting that most parameters gradually stabilize before the model converges.

2.For clarity, we present the *best-ever* accuracy instead of instantaneous accuracy (which fluctuates drastically) in this article.

3.While an *update* here corresponds to one iteration for SGD analysis, for FL setups it is naturally extended to be an accumulated update over an entire round. A parameter judged stable at per-iteration granularity would remain so at per-round granularity. For simplicity, in later parts we skip the intra-round microscopic details, and treat each FL round as a SGD iteration when analyzing parameter trajectories.

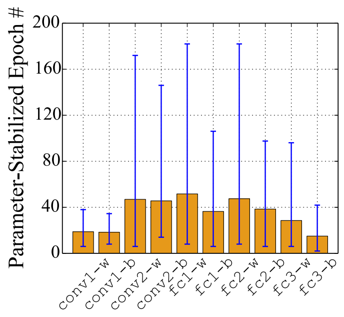


Fig. 3. Average epoch number where parameters in different layers become stable (error bars show the 5th/95th percentiles).

2) Manipulating Granularity: Tensor or Scalar?: It is well-known that a neural network model is a list of *tensors* (e.g., weight tensor, bias tensor), where each tensor corresponds to a multi-dimensional array of *scalars*. When compressing communication by skipping synchronizing the stable parameters, we have two choices regarding the parameter manipulation granularity: *tensor*-granularity and *scalar*-granularity. The former is more concise by synchronizing scalars of a tensor in an *all-or-nothing* manner, yet it assumes that all the scalars in a tensor follow the same stability pattern. To find out the appropriate manipulating granularity, we resort to testbed measurements that compare the stability behaviors of different parameters in a model.

As in Fig. 3, we group all scalars into different buckets according to the tensor they belong to, and calculate the average epoch number at which the scalars in that tensor bucket become stable. In LeNet-5 there are 10 tensors, e.g., `conv1-w`—the weight tensor in the first convolutional layer, and `fc2-b`—the bias tensor in the second fully connected layer. Here a scalar is deemed stable when its effective perturbation drops below 0.01, and the error bars in Fig. 3 represent the 5th and 95th percentiles.

As indicated by Fig. 3, different tensors exhibit different stability trends; meanwhile, there also exist large gaps between the 5th and 95th error bars of each tensor, implying that parameters within the same layer may exhibit vastly different stabilization characteristics. This is in fact reasonable, because some features of the input samples may be easier to learn than the rest, and thus in a neural network layer, some scalar parameters move faster in their optimal dimensions and would converge faster [66]. This is essentially a property called *non-uniform convergence* [27].

Therefore, the granularity for parameter synchronization control needs to be *individual scalar* instead of the entire tensor. That is, let $\mathbf{x}_k^j$ and $\mathbf{u}_k^j$ respectively represent the j-th scalar in the flattened parameter[4] $\mathbf{x}_k$ and flattened update $\mathbf{u}_k$, then given a stability threshold γ, a scalar $\mathbf{x}_k^j$ would judged as stable if

$$\mathcal{P}_k^j = \frac{|\sum_{i=k-\mathcal{S}+1}^{k} \mathbf{u}_i^j|}{\sum_{i=k-\mathcal{S}+1}^{k} |\mathbf{u}_i^j|} < \gamma. \tag{2}$$

4.For simplicity, in the remaining part of this article, we use a vector symbol $\mathbf{x}$ to represent all the parameters in a model. In implementation, that vector can be obtained by first expanding all the model tensors into a vector (with the PyTorch API `Tensor.view(-1)`) and then concatenating those vectors together.

Note that our objective in this work is to reduce the communication volume for FL without compromising the accuracy performance. In the next section (Section IV), we will explore how to treat those stabilized parameter so as to harvest communication compression with accuracy preserved, whose applicability is further extended in Section V. Later in Section VI, we will discuss how to integrate that solution into realistic FL systems in an efficient manner, and finally we will evaluate its performance in Section VII.

IV. Adaptive Parameter Freezing

In this section, we present a novel mechanism called Adaptive Parameter Freezing (APF), which can reduce the communication cost in FL process with the model convergence validity preserved. We first explore some strawman solutions, with the lessons learned we then develop our APF solution.

A. Lessons Learned From Strawman Solutions

Given the stabilized parameters identified with the effective perturbation metric, how to exploit them so that we can reap communication reduction without compromising model convergence? We find that some intuitive approaches however do exhibit some fatal deficiencies, from which we can learn the key principles that a valid solution must follow.

Intuitive Solution 1—Partial Synchronization. To exploit the stabilized parameters for communication compression, an intuitive idea is to exclude them from synchronization (but still update them locally), and only synchronize the rest of the model to the central server. We find that such a *partial synchronization* method may cause severe accuracy loss.

In typical FL scenarios, the local training data on an edge client is generated under particular device environment or user preference. Thus the local data is usually *not identically and independently distributed* (i.e., non-IID) across different clients [7], [32], [40]. This implies that the local loss function $F(\mathbf{x})$ and the corresponding local optima $\mathbf{x}^\star$ on clients are usually different [68]. Therefore, a parameter that is updated only locally would eventually diverge to different local optima on different clients; such inconsistency of unsynchronized parameters would deteriorate the model accuracy.

To confirm this, we train the LeNet-5 model in a distributed manner with non-IID data. There are two clients each with 5 distinct classes of the CIFAR-10 dataset, and stabilized parameters are excluded from later synchronization. Fig. 4 shows the local variation of two randomly selected parameters: Once they are excluded from global synchronization, their values on different clients would diverge remarkably. The model accuracy under such a *partial synchronization* method is further revealed in Fig. 5: Compared to full-model synchronization, there is an accuracy loss of more than 10%. Therefore, the first principle our design must follow is that, *stabilized (i.e., unsynchronized) parameters must be kept unchanged on each client* (*Principle-1*).

Intuitive Solution 2—Permanent Freezing. Given Principle-1, another intuitive method is to simply fix the stabilized parameters to their current values. Such a *permanent freezing* method can naturally prevent parameter divergence. However, when

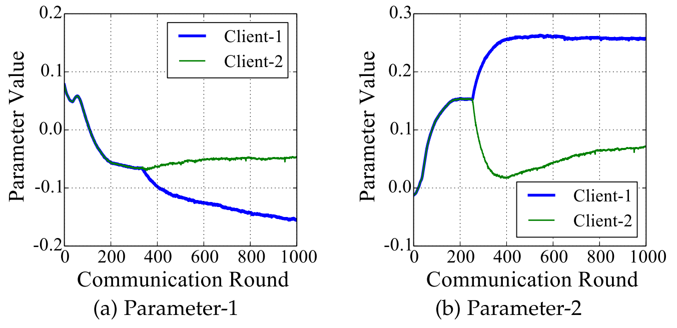


Fig. 4. When training LeNet-5 under a non-IID setup with two clients, the local values of two randomly selected parameters (stabilizing at around round-300 and then updated purely locally) would diverge on different clients.

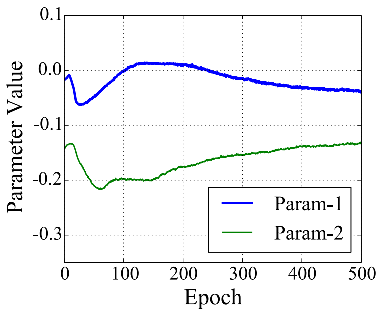


Fig. 7. The two sampled parameters stabilize only *temporarily* between epoch 100 and 200.

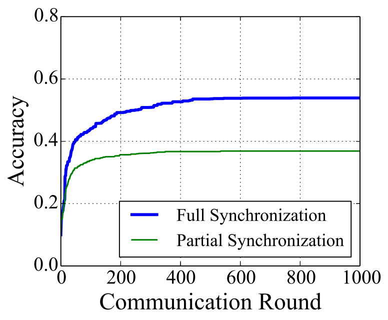


Fig. 5. *Partial synchronization* causes severe accuracy loss on non-IID data.

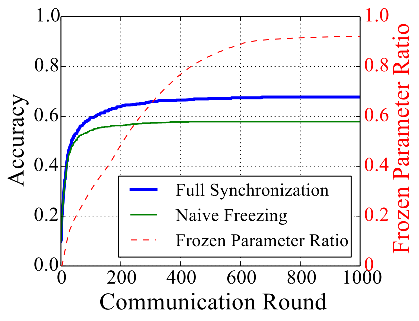


Fig. 6. *Permanent freezing* also causes accuracy loss.

training LeNet-5 with permanent freezing with two clients, we find that the accuracy performance as depicted in Fig. 6 is still suboptimal compared to full-model synchronization.

To understand the reasons behind we look into the parameter evolution process during training. Interestingly, we find that some parameters stabilize only temporarily. Fig. 7 showcases two such parameters: they start to change again after a transient stable period. In fact, parameters in neural network models are not independent of each other. They may have complex interactions with one another in different phases of training [52], and this may cause a seemingly-stabilized parameter to drift away from its current value (to ultimate optimum). The permanent freezing solution, however, prevents such parameters from converging to the true optimum once they are frozen, which eventually compromises the model accuracy. It is inherently difficult to distinguish the temporarily stabilized parameters by devising a better stability metric based on local observations, since their behavior before drifting is virtually identical to that of those that truly converge. Hence, as another principle (*Principle-2*), *any effective solution must handle the possibility of temporary stabilization.*

B. Adaptive Parameter Freezing

Parameter Freezing in a Periodical Manner. To summarize, we learned two lessons from the previous explorations. First, to ensure model consistency, we have to freeze the non-synchronized stable parameters. Second, to ensure that the frozen parameters can converge to the true optima, we shall allow them to resume being updated (i.e., be *unfrozen*) when necessary.

Therefore, for those stabilized parameters, instead of freezing them forever, we freeze them only for a certain time interval, which we call *freezing period*. Within the freezing period, the parameter is fixed to its previous value, and once the freezing period expires, that parameter should be updated as normal until it stabilizes again. That is, suppose the parameter $\mathbf{x}_k^j$ is judged as being stable (i.e., $\mathcal{P}_k^j < \gamma$), then it would be associated with a freezing period of L_k^j, such that $\mathbf{x}_{t+1}^j = \mathbf{x}_t^j$ for $t \in \{k, k+1, \ldots, k+L_k^j-1\}$.

A remaining question is, given that there is no complete knowledge of the model convergence process a priori, when to unfreeze a frozen parameter? Or how to set the *freezing period* once a parameter becomes stable? We next propose a novel mechanism to control the parameter freezing period.

Control Mechanism of Parameter Freezing Period. Since each parameter has a distinct convergence behavior, our solution must adapt to each individual parameter instead of using an identical freezing period for all. There is a clear *trade-off* here: If the freezing period is set too large, we can compress communication significantly, but some *should-be-drifting* parameters cannot escape frozen state timely, which may compromise the model convergence accuracy; on the other hand, if the freezing period is set too small, performance benefit from parameter freezing would be quite limited.

Therefore, we need to *adaptively* set the freezing period of each stabilized parameter. For generality we do not assume any

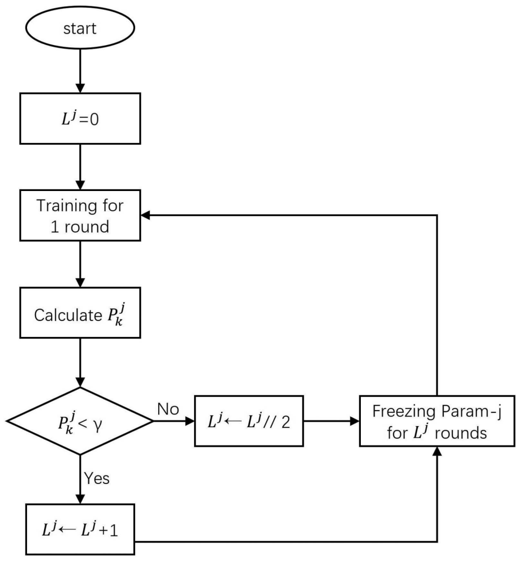


Fig. 8. A flowchart showing the control mechanism of parameter freeing period in APF. If a frozen parameter-j keeps stable (i.e., $P_k^j < \gamma$) after its freezing period expires, its freezing period L^j is increased linearly (e.g., added by 1); otherwise if that parameter is no longer stable, its freezing period is decreased multiplicatively (e.g., being halved).

prior knowledge of the model trained, and choose to adjust parameter freezing period in a *tentative* manner. In particular, we shall greedily increase the freezing period as long as the frozen parameter keeps stable after being unfrozen, and shall meanwhile react agilely to potential parameter variation.

To this end, we design a novel control mechanism called *Adaptive Parameter Freezing* (APF). APF is inspired by the classical control mechanism of TCP (*Transmission Control Protocol*) in computer networking [6], [10]: It *additively increases* the sending rate of a flow upon the receipt of a data packet in order to better utilize the bandwidth, and *multiplicatively decreases* the sending rate when a loss event is detected, to quickly react to congestion. This simple control policy has been working extremely robustly as one of the cornerstones of the Internet.

Similarly, as shown in Fig. 8, under APF the associated freezing period of each stabilized parameter is also adjusted in an "additively-increase, multiplicative decrease" manner. For a stable parameter, its freezing period starts with a small value (e.g., one round as in our evaluation). When the freezing period expires, the parameter resumes regular training in the following round, after which we update its effective perturbation and re-check its stability. If the parameter is still stable, we *additively increase*—and otherwise *multiplicatively decrease* (e.g., halve)—the duration of its freezing period, thereby adapting to the dynamics of each parameter. This way, the converged parameters would stay frozen for most of the time, whereas the temporally stabilized ones can swiftly resume regular synchronization.

To integrate the above algorithm into realistic FL systems, we create a dedicated Python module called `APF_manager`. That `APF_manager` makes our communication optimization transparent to clients: It compresses (uncompresses) vanilla gradients before (after) synchronization based on the parameter freezing status, and also automatically maintains the parameter freezing periods based on the algorithm in Fig. 8. We have also adopted a series of engineering optimizations to reduce the extra overhead of APF, and the elaboration of such implementation details is deferred to Section VI. Next, we mathematically prove that APF can preserve model convergence for general models.

C. Convergence Analysis

Note that our motivating analysis in Section III-A assumes the models be convex, which is acceptable for the community [11], [27], [41], [55]. Yet, we admit that deep neural networks are essentially non-convex. In this part we prove that APF can ensure convergence validity even for non-convex models. We first state the smoothness assumption.

Assumption 3. (β-smoothness) The global loss function $F(\mathbf{x})$ is β-smooth, i.e.,

$$F(\mathbf{y}) \leq F(\mathbf{x}) + \nabla F(\mathbf{x})^T(\mathbf{y}-\mathbf{x}) + \frac{\beta}{2}\|\mathbf{y}-\mathbf{x}\|^2.$$

Inspired by the perturbed iterate framework of [38], we define a virtual sequence $\{\tilde{\mathbf{x}}_k\}$ based on the true sequence $\{\mathbf{x}_k\}$ in the following way:

$$\tilde{\mathbf{x}}_0 := \mathbf{x}_0, \qquad \tilde{\mathbf{x}}_{k+1} := \tilde{\mathbf{x}}_k - \eta_t \nabla F(\mathbf{x}_k). \tag{3}$$

Our analysis is based on the following assumption which bounds the cumulative updates of a frozen parameter if otherwise refined regularly without APF.

Assumption 4. (Parameter Gap incurred by Freezing.) Suppose $\mathbf{x}_k^j$ is judged as being stable (i.e., $\mathcal{P}_{k,j}^{(\mathcal{S})} < \gamma$) and associated with a freezing period L_k^j under APF, then under Assumption 2 (let $\mathbb{E}|\mathbf{u}^j| \leq \sigma^j$ be the bound of the j-th update, where $\sum_j (\sigma^j)^2 = \sigma^2$), there exists a constant ν such that $\forall l \in 1, 2, \ldots, L_k^j$,

$$\mathbb{E}|\sum_{l'=0}^{l} \mathbf{u}_{k+l'}^j| \leq \nu\gamma\eta_k\sigma^j. \tag{4}$$

Justification of Assumption 4. Admittedly, given the stochastic nature of the model training process, there is no deterministic knowledge on the virtual behavior of a frozen parameter in a "parallel world" without freezing. To handle this challenge, given the adaptive monitoring functionality of APF, we faithfully assume that the *expected* behavior of a parameter within the freezing period resembles that within the previous observation window, i.e., (2) also hold for the excluded updates $\{\mathbf{u}_k^j, \mathbf{u}_{k+1}^j, \ldots, \mathbf{u}_{k+L_k^j-1}^j\}$. That is,

$$\mathbb{E}|\sum_{l'=0}^{L_k^j-1} \mathbf{u}_{k+l'}^j| = \mathcal{P}_k^j \sum_{l'=0}^{L_k^j-1} |\mathbf{u}_{k+l'}^j| \leq \gamma L_k^j \eta_k \sigma^j. \tag{5}$$

Further, in APF the freezing period L_k^j is not set arbitrarily as an independent constant, but by adapting to the model landscape (L_k^j is increased only when the previous stability trend persists). Therefore, we can actually get rid of L_k^j and use a constant

ν to bound the expected cumulative error after skipping those updates—this then yields Assumption 4.

Based on the above two assumptions (note that the convexity assumption is not required), we can obtain the following theorem:

Theorem 2 (Convergence Property). Under the Assumption 3 and Assumption 4, after running T iterations under APF, we have:

$$\frac{1}{\sum_{k=1}^{T}\eta_k}\sum_{k=1}^{T}\eta_k\mathbb{E}[\|\nabla F(\mathbf{x}_k)\|^2] \le \frac{4(F(\mathbf{x}_0)-F(\mathbf{x}^\star))}{\sum_{k=1}^{T}\eta_k} + \frac{4\sigma^2\beta^2\nu^2\gamma^2\sum_{k=1}^{T}\eta_k^3}{\sum_{k=1}^{T}\eta_k} + \frac{2\beta\sigma^2\sum_{k=1}^{T}\eta_k^2}{\sum_{k=1}^{T}\eta_k}. \tag{6}$$

Proof. Under the Assumption of β-smooth of F, we have

$$\begin{aligned}F(\tilde{\mathbf{x}}_{k+1})-F(\tilde{\mathbf{x}}_k) &\le \nabla F(\tilde{\mathbf{x}}_k)^T(\tilde{\mathbf{x}}_{k+1}-\tilde{\mathbf{x}}_k)+\frac{\beta}{2}\|\tilde{\mathbf{x}}_{k+1}-\tilde{\mathbf{x}}_k\|^2\\ &= -\eta_k\nabla F(\tilde{\mathbf{x}}_k)^T g_k(\mathbf{x}_k)+\frac{\eta_k^2\beta}{2}\|g_k(\mathbf{x}_k)\|^2.\end{aligned} \tag{7}$$

Taking the expectation at iteration k, we have

$$\begin{aligned}&\mathbb{E}[F(\tilde{\mathbf{x}}_{k+1})]-F(\tilde{\mathbf{x}}_k)\\ &\le -\eta_k\nabla F(\tilde{\mathbf{x}}_k)^T\mathbb{E}[g_k(\mathbf{x}_k)]+\frac{\eta_k^2\beta}{2}\mathbb{E}[\|g_k(\mathbf{x}_k)\|^2]\\ &= -\eta_k\nabla F(\tilde{\mathbf{x}}_k)^T\nabla F(\mathbf{x}_k)+\frac{\eta_k^2\beta}{2}\mathbb{E}[\|g_k(\mathbf{x}_k)\|^2]\\ &= -\frac{\eta_k}{2}\|\nabla F(\tilde{\mathbf{x}}_k)\|^2-\frac{\eta_k}{2}\|\nabla F(\mathbf{x}_k)\|^2\\ &\quad+\frac{\eta_k}{2}\|\nabla F(\tilde{\mathbf{x}}_k)-\nabla F(\mathbf{x}_k)\|^2+\frac{\eta_k^2\beta}{2}\mathbb{E}[\|g_k(\mathbf{x}_k)\|^2]\\ &\le -\frac{\eta_k}{2}\|\nabla F(\tilde{\mathbf{x}}_k)\|^2+\frac{\eta_k\beta^2}{2}\|\tilde{\mathbf{x}}_k-\mathbf{x}_k\|^2+\frac{\eta_k^2\beta}{2}\mathbb{E}[\|g_k(\mathbf{x}_k)\|^2]\\ &= -\frac{\eta_k}{2}(\|\nabla F(\tilde{\mathbf{x}}_k)\|^2+\beta^2\|\mathbf{x}_k-\tilde{\mathbf{x}}_k\|)\\ &\quad+\eta_k\beta^2\|\mathbf{x}_k-\tilde{\mathbf{x}}_k\|^2+\frac{\eta_k^2\beta}{2}\mathbb{E}[\|g_k(\mathbf{x}_k)\|^2]\\ &\le -\frac{\eta_k}{2}(\|\nabla F(\tilde{\mathbf{x}}_k)\|^2+\beta^2\|\mathbf{x}_k-\tilde{\mathbf{x}}_k\|)\\ &\quad+\eta_k\beta^2\|\mathbf{x}_k-\tilde{\mathbf{x}}_k\|^2+\frac{\eta_k^2\beta\sigma^2}{2}.\end{aligned} \tag{8}$$

Taking the expectation before k, it yields

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

Here we first focus on the term of $\mathbb{E}\|\mathbf{x}_k-\tilde{\mathbf{x}}_k\|^2$. For those frozen parameters, the gap is quantified by Assumption 4; for those non-frozen parameters which is updated under vanilla SGD without any APF interference, there is no gap incurred in expectation. Therefore, let l^j represent the frozen length so far of the j-th scalar, we have

$$\begin{aligned}\mathbb{E}\|\mathbf{x}_k-\tilde{\mathbf{x}}_k\|^2 &= \mathbb{E}(\sum_j|\mathbf{x}_k^j-\tilde{\mathbf{x}}_k^j|^2)\\ &= \sum_j\mathbb{E}|\sum_{l'=0}^{l^j}\mathbf{u}_{k+l'}^j|^2 \le \nu^2\eta_k^2\gamma^2\sum_j(\sigma^j)^2=\nu^2\eta_k^2\gamma^2\sigma^2.\end{aligned} \tag{10}$$

Applying the above inequality to (9), we have

$$\begin{aligned}\mathbb{E}[F(\tilde{\mathbf{x}}_{k+1})]-\mathbb{E}[F(\tilde{\mathbf{x}}_k)] &\le \eta_k^2\sigma^2(\eta_k\beta^2\nu^2\gamma^2+\frac{\beta}{2})\\ &\quad-\frac{\eta_k}{2}\mathbb{E}[(\|\nabla F(\tilde{\mathbf{x}}_k)\|+\beta^2\|\mathbf{x}_k-\tilde{\mathbf{x}}_k\|^2)].\end{aligned} \tag{11}$$

Then we can obtain

$$\begin{aligned}&\eta_k\mathbb{E}[(\|\nabla F(\tilde{\mathbf{x}}_k)\|^2+\beta^2\|\mathbf{x}_k-\tilde{\mathbf{x}}_k\|^2)]\\ &\le 2(\mathbb{E}[F(\tilde{\mathbf{x}}_k)]-\mathbb{E}[F(\tilde{\mathbf{x}}_{k+1})])+\eta_k^2\sigma^2(2\eta_k\beta^2\nu^2\gamma^2+\beta).\end{aligned} \tag{12}$$

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

$$\begin{aligned}\|\nabla F(\mathbf{x}_k)\|^2 &= \|\nabla F(\mathbf{x}_k)-\nabla F(\tilde{\mathbf{x}}_k)+\nabla F(\tilde{\mathbf{x}}_k)\|^2\\ &\le 2\|\nabla F(\mathbf{x}_k)-\nabla F(\tilde{\mathbf{x}}_k)\|^2+2\|\nabla F(\tilde{\mathbf{x}}_k)\|^2\\ &\le 2\beta^2\|\mathbf{x}_k-\tilde{\mathbf{x}}_k\|^2+2\|\nabla F(\tilde{\mathbf{x}}_k)\|^2.\end{aligned} \tag{13}$$

Combine with (12), we obtain

$$\begin{aligned}&\eta_k\mathbb{E}[\|\nabla F(\mathbf{x}_k)\|^2] \le 2\eta_k\mathbb{E}[\beta^2\|\mathbf{x}_k-\tilde{\mathbf{x}}_k\|^2+\|\nabla F(x)\|^2]\\ &\le 4(\mathbb{E}[F(\tilde{\mathbf{x}}_k)]-\mathbb{E}[F(\tilde{\mathbf{x}}_{k+1})])+2\eta_k^2\sigma^2(2\eta_k\beta^2\nu^2\gamma^2+\beta).\end{aligned} \tag{14}$$

Summing up the above inequality for $k=1,2,\ldots,T$, we have

$$\sum_{k=1}^{T}\eta_k\mathbb{E}[\|\nabla F(\mathbf{x}_k)\|^2] \le 4(F(\mathbf{x}_0)-F(\mathbf{x}^\star)) + 4\sigma^2\beta^2\nu^2\gamma^2\sum_{k=1}^{T}\eta_k^3+2\beta\sigma^2\sum_{k=1}^{T}\eta_k^2. \tag{15}$$

By dividing the summation of learning rates, we have:

$$\frac{1}{\sum_{k=1}^{T}\eta_k}\sum_{k=1}^{T}\eta_k\mathbb{E}[\|\nabla F(\mathbf{x}_k)\|^2] \le \frac{4(F(\mathbf{x}_0)-F(\mathbf{x}^\star))}{\sum_{k=1}^{T}\eta_k} + \frac{4\sigma^2\beta^2\nu^2\gamma^2\sum_{k=1}^{T}\eta_k^3}{\sum_{k=1}^{T}\eta_k} + \frac{2\beta\sigma^2\sum_{k=1}^{T}\eta_k^2}{\sum_{k=1}^{T}\eta_k}.$$

Theorem 2 implies that model training under APF can converge to 0 if T is large enough, when η_k is set to satisfy the following condition:

$$\lim_{T\to\infty}\sum_{k=1}^{T}\eta_k=\infty \quad\text{and}\quad \lim_{T\to\infty}\frac{\sum_{k=1}^{T}\eta_k^2}{\sum_{k=1}^{T}\eta_k}=0. \tag{16}$$

Therefore, if we set $\eta_k=\mathcal{O}(\frac{1}{\sqrt{T}})$ which satisfies the conditions in (16), then model convergence can be guaranteed under APF.

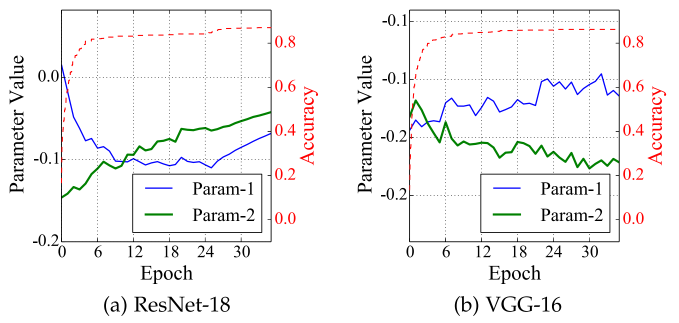


Fig. 9. For over-parameterized models, the sampled parameters may drift or perform random walk after convergence.

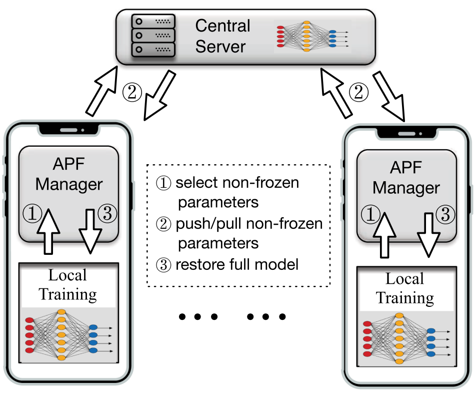


Fig. 10. FL workflow with `APF_Manager`.

V. More Aggressive Parameter Freezing: APF# and APF++

In standard APF, we focus on identifying and freezing those stable parameters: The more parameters becoming stable, the larger communication reduction benefit APF can yield. However, for large over-parameterized models like ResNet and VGG [43], [65], many parameters may not converge to a fixed point (i.e., stabilize) due to irregular landscapes like *flat minima* [26] or *saddle points* [31]: They are not stable even after the model converges, which may saliently limit the performance benefit of APF.

We first reveal that problem with testbed measurements. In Fig. 9, we show the variations of two randomly sampled parameters in ResNet and VGG. Compared with the parameter variations of smaller models (as in Figs. 1, 4 and 7), we find that the parameters in large over-parameterized models exhibit more salient fluctuations. Moreover, even after the model trained reaches the highest accuracy, the sampled parameters are still not stable; instead, they conduct random walk in fixed or ad-hoc directions without affecting accuracy. In fact, this is a natural result caused by irregular landscapes like flat minima and saddle points [26], [31]. For such over-parameterized models, the fraction of stable parameters that could be frozen under APF may be quite small (as shown later in Fig. 11b of Section VII).

Therefore, for over-parameterized large models, we can try freeze the model parameters more aggressively to harvest larger performance benefit. To this end, we propose two APF extensions—APF# and APF++, which randomly associate some unstable parameters with a non-zero freezing period.

APF#. APF# is directly motivated by the famous Dropout technique [24], [33], [52]. By randomly (typically with a probability of 0.5) disabling some neural connections in each iteration, Dropout has been shown to saliently improve the model generalization capability. Similarly, for each unfrozen parameter in APF#, we randomly (under a fixed probability) associate it with a freezing period of 1. In this way, it can be expected that the transmission amount can be further reduced with model accuracy still preserved.

APF++. Note that to design an aggressive version of APF, we need to answer two questions: What is the probability to put an unstable parameter into freezing? And how long is that freezing period? In APF# the two answers are both fixed (0.5 and 1) to resemble Dropout, yet in APF++ we let both of them gradually increase during the training process. That is, given the round number K, we set the probability to freeze a unstable parameter as $a_1 K$, and the length of the freezing period is randomly sampled from $[1, 1 + a_2 K]$, where a_1 and a_2 are predefined coefficients. APF++ is motivated by the fact that the earlier training phase is usually more important for model convergence [4], [62], implying that the freezing probability can be larger in the end. Moreover, over-parameterized models with irregular landscapes like flat minima may allow for longer random freezing, with no need to limit it to merely one round. This way, it is expected that APF++ can attain even larger communication reduction with comparable accuracy performance for over-parameterized models.

VI. Implementation

In this section we focus on implementing the APF algorithm in a practical manner. We first customize the vanilla APF algorithm to make it efficient in communication, computation and memory, as well as to make it more roust against improper hyper-parameter setups. Then we elaborate the implementation details with the PyTorch framework.

A. Customizing APF for Efficiency and Robustness

For Better Communication Efficiency—Updating Parameter Freezing Status on Clients. Identifying stabilized parameters is the first step to eliminate unnecessary parameter synchronization. Doing this on the central server would incur additional communication cost of sending the results back to clients. Therefore, we choose to perform parameter stability check on the *client* side. Given the memory and computing-power limitations of edge devices, calculating the effective perturbation defined in (1) is resource prohibitive, as it requires each client to locally store a series of model snapshots and process them frequently. Therefore, we need to further mitigate the computation and memory overhead incurred by APF.

For Better Computation Efficiency—Checking Stability Once-for-Multiple-Rounds. To reduce the computation cost, we allow the frequency of stability check to be relaxed from *once-for-per-round* to *once-for-multiple-rounds*. Correspondingly, we judge stability based on the *accumulated* model updates between two consecutive checks. This is feasible because a stabilized

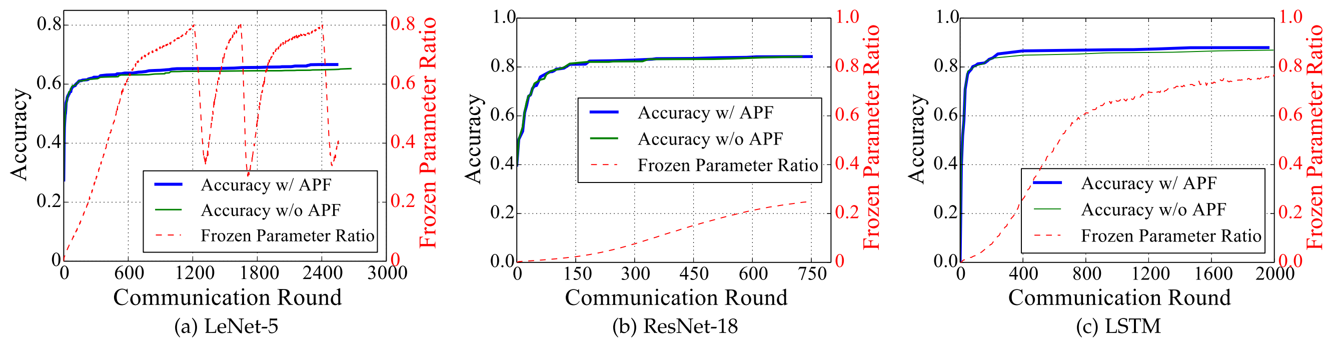


Fig. 11. Test accuracy curves when training different models with and without APF.

parameter would remain so when observed at a coarser time granularity.

For Better Memory Efficiency—Incorporating EMA Smoothing in Calculating Effective Perturbation. To be memory-efficient, instead of maintaining a window of previous updates, we adopt the Exponential Moving Average (EMA) method to calculate effective perturbation. For a parameter in the K^{th} stability check, let Δ_K^j represent the cumulative update since the last check for parameter j. Then the *effective perturbation* of this parameter, $\mathcal{P}_K^j$, can be defined as:

$$\mathcal{P}_K^j = \frac{|E_K^j|}{A_K^j}, \text{ where } E_K^j = \alpha E_{K-1}^j + (1-\alpha)\Delta_K^j,$$
$$A_K^j = \alpha A_{K-1}^j + (1-\alpha)|\Delta_K^j|. \quad (17)$$

Here E_K is the moving average of the parameter updates, and A_K^j is the moving average of the *absolute value* of parameter updates. The smoothing factor α is set close to 1. This way, the nice properties of effective perturbation under the previous definition of (1)—close to 1 if model updates are of the same direction, and close to 0 if they oscillate—still hold for P_K yet at a much lower compute and memory cost. In addition, decaying the earlier updates with EMA is sensible because recent behavior is more significant. Therefore, in the following, we determine a parameter as stable if its effective perturbation under this new definition is smaller than a given stability threshold.

For Better Robustness—Enabling Stability Threshold Decay. Ideally, the stability threshold should be *loose enough* to include all stabilized parameters (i.e., avoid false negatives), and in the meantime be *tight enough* to not mistakenly include any unstable parameters (i.e., avoid false positives). However, we cannot assume that the stability threshold is always set appropriately. To be robust, we introduce a mechanism that adaptively tunes the stability threshold at runtime: Each time when most (e.g., 80%) parameters have been categorized as stable, we decrease the stability threshold by one half—a method similar to learning rate decay. This way, the negative effect of improper stability thresholds can be gradually rectified, which will be verified later in Section VII-H.

B. Implementing APF Atop PyTorch Framework

Workflow Overview. We implement APF as a pluggable Python module, named `APF_Manager`, atop the PyTorch framework [2]. The detailed workflow with `APF_Mananger` is elaborated in Fig. 10 and Algorithm 1. In each iteration, after updating the local model, each client calls the `APF_Manager.Sync()` function to handle all the synchronization-related issues. The `APF_Manager` wraps up all the APF-related operations (stability checking, parameter freezing, model synchronization, etc.), rendering the APF algorithm transparent and also pluggable to edge users. Next we introduce the key techniques in our APF implementation.

Using a Bitmap to Represent Parameter Freezing Status. When doing global synchronization, the `APF_Manager` selects and packages all the unstable parameters into a tensor for fast transmission. The selection is dictated by a bitmap $M_{\text{is\_frozen}}$ representing whether each (scalar) parameter should be frozen or not, which is updated in the function `StabilityCheck()` based on the latest value of effective perturbation. To avoid extra communication overhead, the `APF_Manager` on each client refreshes $M_{\text{is\_frozen}}$ independently. Note that, since $M_{\text{is\_frozen}}$ is calculated from synchronized model parameters, the values of $M_{\text{is\_frozen}}$ calculated on each worker would be always identical.

Emulating the Effect of Fine-Grained Parameter Freezing. A key implementation challenge is that, PyTorch operates parameters at the granularity of a full tensor, with no APIs supporting parameter freezing at a granularity of per dimension (scalar). This also holds in other machine learning frameworks like TensorFlow [3] and MXNet [14]. To achieve fine-grained parameter freezing, we choose to *emulate* the freezing effect by rolling back: those *should-be-frozen* scalar parameters participate model update normally, but after each iteration, their values are rolled back to their previous values (Line-2 in Algorithm 1). Meanwhile, all APF operations are implemented with the built-in tensor-based APIs of PyTorch for fast processing.

Complexity analysis. Given that edge devices are resource-constrained, we need to be careful with the overheads incurred by APF operations. It is easy to see that the space and computation complexity of Algorithm 1 is linear to the model size, i.e., $O(|\mathbf{x}|)$. This is acceptable because memory consumption in model training is mostly incurred by input data and feature maps, compared to which the memory consumption of the model itself is usually *orders-of-magnitude* smaller [48]. Meanwhile, with Tensor-processing APIs provided by PyTorch, the tensor

Algorithm 1: Workflow With Adaptive Parameter Freezing.

```
Require: $F_s$, $F_c$, $T_s$ ▷ $F_s$: synchronization frequency (i.e., synchronization is conducted once for $F_s$ iterations); $F_c$: stability check frequency; $T_s$: threshold on $P_k$ below which to judge a parameter as being stable
Client: $i = 1, 2, \ldots, N$:
1: procedure ClientIterate$k$
2:     $\mathbf{x}_k^i \leftarrow \mathbf{x}_{k-1}^i + u_k^i$ ▷ update local model $\mathbf{x}_k^i$ on client-$i$
3:     $\mathbf{x}_k^i \leftarrow$APF_Manager.Sync($\mathbf{x}_k^i$) ▷ pass model to APF_Manager
Central Server:
1: procedure Aggregate$\breve{\mathbf{x}}_k^1, \breve{\mathbf{x}}_k^2, \ldots, \breve{\mathbf{x}}_k^N$
2:     return $\frac{1}{N}\sum_{i=1}^N \breve{\mathbf{x}}_k^i$ ▷ aggregate all the non-frozen parameters
APF Manager:
1: procedure Sync$\mathbf{x}_k^i$
2:     $\bar{\mathbf{x}}_k^i \leftarrow M_{\text{is\_frozen}}\ ?\ \mathbf{x}_{k-1}^i : \mathbf{x}_k^i$ ▷ emulate freezing by parameter rollback; $M_{\text{is\_frozen}}$: freezing mask
3:     if $k \bmod F_s = 0$ then
4:         $\breve{\mathbf{x}}_k^i \leftarrow \bar{\mathbf{x}}_k^i$.masked_select($!M_{\text{is\_frozen}}$)
           ▷ $\breve{\mathbf{x}}_k^i$: a compact tensor composed of only the unstable parameters
5:         $\breve{\mathbf{x}}_k \leftarrow$Central_Server.aggregate($\breve{\mathbf{x}}_k^i$)
           ▷ synchronize the tensor containing unstable parameters
6:         $\bar{\mathbf{x}}_k^i \leftarrow \bar{\mathbf{x}}_k^i$.masked_fill($!M_{\text{is\_frozen}}, \breve{\mathbf{x}}_k$)
           ▷ restore the full model by merging the frozen (stable) parameters
7:         if $k \bmod F_c = 0$ then
8:             $M_{\text{is\_frozen}} \leftarrow$StabilityCheck($\bar{\mathbf{x}}_k^i, M_{\text{is\_frozen}}$) ▷ update $M_{\text{is\_frozen}}$
9:     return$\bar{\mathbf{x}}_k^i$
10: function StabilityCheck$\bar{\mathbf{x}}_k^i, M_{\text{is\_frozen}}$
       ▷ Operations below are tensor-based, supporting where selection semantics
11:     update $E$, $A$, $\mathcal{P}_k$ with $\bar{\mathbf{x}}_k^i$
12:     $L_{\text{freezing}} \leftarrow L_{\text{freezing}} + F_c$ where $\mathcal{P}_k \leq T_s$
13:     $L_{\text{freezing}} \leftarrow L_{\text{freezing}}/2$ where $\mathcal{P}_k > T_s$
       ▷ $L_{\text{freezing}}$: freezing period lengths, updated in TCP-style
14:     $I_{\text{unfreeze}} \leftarrow k + L_{\text{freezing}}$
       ▷ $I_{\text{unfreeze}}$: round ids representing freezing deadlines
15:     $M_{\text{is\_frozen}} \leftarrow (k < I_{\text{unfreeze}})$ ▷ update freezing mask
16:     return$M_{\text{is\_frozen}}$
```

traversal operations of APF are much faster than the already-existing convolution operations in forward or backward propagations. Such a light-weight nature allows APF to be deployed in resource-constraint scenarios like on IoT devices, and we will evaluate APF overheads later in Section VII-I.

VII. Evaluation

In this section, we systematically evaluate the performance of our APF algorithm family. We start with end-to-end comparisons between APF and the standard FL scheme, and then resort to a series of micro-benchmark evaluations to justify the superiority of APF in various scenarios, including its extended versions (APF#, APF++ and APF with Quantization). Finally we examine the hyper-parameter sensitivity as well as the extra overheads incurred.

A. Experimental Setup

Hardware Setup. We create a FL architecture[5] with 50 EC2 `m5.xlarge` instances as edge clients, each with 2 vCPU cores and 8GB memory (similar with a smart phone). Meanwhile, following the global Internet condition [1], the download and upload bandwidth of each client is configured to be 9Mbps and 3Mbps, respectively. The central server is a `c5.9xlarge` instance with a bandwidth of 10Gbps.

Dataset setup. Datasets in our experiments are CIFAR-10 and the KeyWord Spotting dataset (KWS)—a subset of the Speech Commands dataset [58] with 10 keywords, which are partitioned across the 50 clients. To synthesize non-IID data distribution, we independently draw each client's training samples following Dirichlet distribution [64], which controls local class evenness via a concentration parameter α ($\alpha \to \infty$ means IID data distribution). In particular, we set $\alpha = 1$ on each worker (under which the expected max-min ratio among sample numbers of different classes is over 50).

Model Setup. Models trained upon the CIFAR-10 dataset are LeNet-5 and ResNet-18, and upon the KWS dataset is a LSTM network with 2 recurrent layers (the hidden size is 64), all with a batch size of 100. Meanwhile, for diversity we use the Adam [19] optimizer for LeNet-5, and SGD optimizer for ResNet-18 and LSTM; their learning rates are respectively set to 0.001, 0.1, and 0.01 (default for each model-optimizer combination), with a weight decay of 0.01.

APF setup. Moreover, the default synchronization and stability check frequency are respectively set as 10 and 50. Regarding the stability check of APF, the EMA parameter α is 0.99, and the stability threshold on effective perturbation is 0.05, which is halved once the fraction of frozen parameters reaches 80%, a method we introduced in Section VI-A.

B. End-to-End Evaluations of Standard APF

Convergence Validity. Fig. 11 shows the test accuracy curves for training the three models with and without APF. The synchronization and stability check frequencies (F_s and F_c) are set as 10 and 50. Here a model converges if its test accuracy has not changed for 100 rounds, and the dashed red lines represent the ratio of frozen parameters in each round. These results demonstrate that APF does not compromise convergence. We list the best-ever testing accuracy in each case in Table I. In particular, we notice that when training LeNet-5 and LSTM,

5.In real-world FL scenarios, due to poor connection and client instability, some clients may dynamically leave or join the FL process [40]. Yet this is only an engineering concern and does not affect the effectiveness of our APF algorithm, because with admission control [7], active workers in each round always start with the latest global model (as well as $M_{\text{is\_frozen}}$ for APF). For simplicity, all the clients can work persistently in our FL architecture setup.

TABLE I
THE BEST TESTING ACCURACY FOR EACH MODEL

Model	LeNet-5	ResNet-18	LSTM
Accuracy w/APF	0.666	0.842	0.879s
Accuracy w/o APF	0.652	0.842	0.869 s

TABLE II
CUMULATIVE TRANSMISSION VOLUME

Model	LeNet-5	ResNet-18	LSTM
Transmission-Volume w/ APF	239 MB	2.62 GB	194 MB
Transmission-Volume w/o APF	651 MB	3.12 GB	428 MB
APF Improvement	63.3%	16.0%	54.7%

TABLE III
AVERAGE PER-ROUND TIME

Model	LeNet-5	ResNet-18	LSTM
Per-round Time w/APF	0.74s	139 s	1.8s
Per-round Time w/o APF	1.02s	158s	2.2s
Improvement	27.5%	12.1%	18.2%

APF actually achieves a better accuracy. This is because, like Dropout [52], intermittent parameter freezing under APF can break up parameter co-adaptations and avoid overfitting, which improves the model's generalization capability.

Communication Efficiency. We next turn to the efficiency gain of APF. Table II summarizes the cumulative transmission volume of each client up to convergence, showing that APF can greatly reduce the transmission volume. For LeNet-5 for example, it provides a saving of 63.3%. Table III further lists the average *per-round time* (i.e., training time divided by the number of rounds) in each case, showing that APF can speed up FL by up to 27.5%. For extreme cases where the bandwidth (e.g., with cellular networks) is much less than the global average, it is expected that the speedup would be even larger.

C. APF Performance on Extremely Non-IID Data

Recall that in Section IV-A we explored two strawman solutions to avoid transmitting stable parameters: partial synchronization (i.e., only synchronizing the unstable parameters and having the stable ones updated locally) and permanent freezing (i.e., no longer updating the stable parameters). Here we compare their performance with standard FL and APF in a setup with extremely non-IID data. In particular, we train LeNet-5 with 5 workers, and each worker is configured to host only 2 distinct classes of CIFAR-10.

As shown in Fig. 12a, APF achieves the same accuracy as standard FL for LeNet-5. More importantly, because parameter freezing in APF can avoid overfitting and improve the model generalization capability, in Fig. 12b it attains an accuracy improvement of 18% over standard FL, which is consistent with our previous conclusion from Fig. 11. Meanwhile, in each case, the accuracy performance of APF is much better than both partial synchronization and permanent freezing methods.

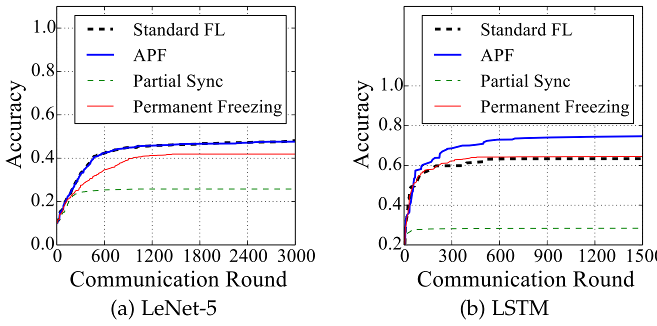


Fig. 12. Performance comparison among different schemes when training LeNet-5 and LSTM on extremely non-IID data.

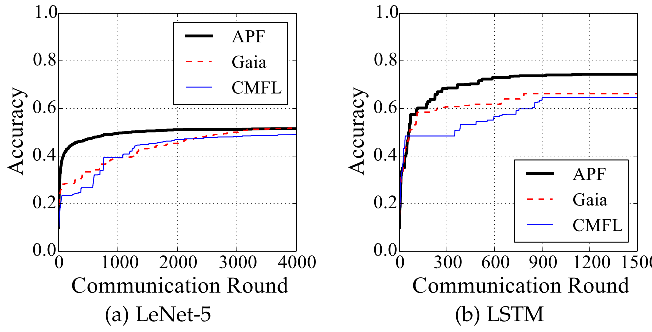


Fig. 13. Accuracy performance of different sparsification methods.

D. Comparison Against Other Sparsification Methods

We further compare the performance of APF with two typical sparsification methods in the literature: Gaia [27] and CMFL [55], which have been introduced in Section II. We implement[6] Gaia and CMFL also with PyTorch, and in our evaluation their hyper-parameters are set to the default values as in their papers: The significance threshold in Gaia is 0.01 (meaning that updates with less-than-1% change will not be reported) and the relevance threshold in CMFL is 0.8 (meaning that updates with less-than-80% direction consistency will not be reported).

Fig. 13 shows the accuracy curves when training LeNet-5 and LSTM with 5 clients each hosting two distinct sample classes. As shown in Fig. 13, in each case APF can always make the best model accuracy. This is because Gaia and CMFL are blind to the long-term model training process, and updates that are temporally insignificant (in value) or irrelevant (in update direction) may still need to be transmitted to ensure validity. That is, purely local sparsification decisions may not be accurate for long-term model convergence.

Furthermore, APF can also surpass Gaia and CMFL in the communication benefit. Fig. 14 shows the cumulative transmission amount when training LeNet-5 and LSTM respectively

6.We implement Gaia and CMFL with two respective Python modules—`Gaia_Manager` and `CMFL_Manager`. Their workflows are similar with that of `APF_Manager`: First select the valuable parameters for transmission, second to conduct remote synchronization, and third to restore the full model from the synchronized portion.

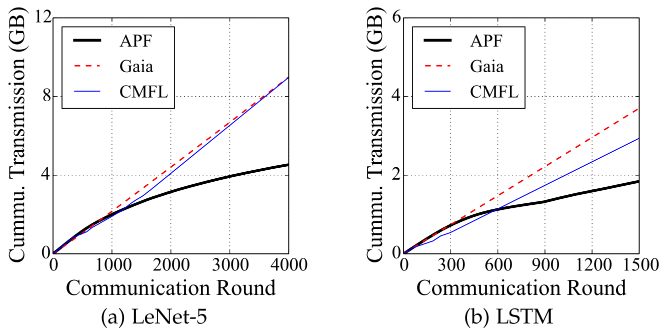


Fig. 14. Cumulative transmission amount under different sparsification methods.

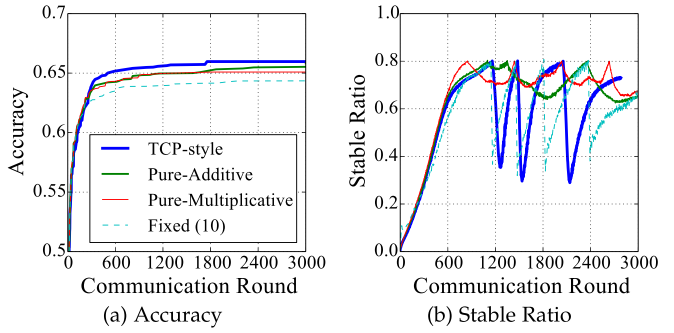


Fig. 15. A comparison of the TCP-style control scheme in APF against other potential design choices. *Pure-Additively* means to additively increase or decrease the freezing period by 1; *Pure-Multiplicatively* means to multiplicatively increase or decrease the freezing period by $2\times$; *Fixed (10)* means to freeze each stabilized parameters for 10 (in number of stability checks, i.e., for $10 \times F_c$ iterations).

under the three schemes (with both pull and push transmission considered). For Gaia and CMFL (with decaying thresholds as elaborated in their papers), their communication reduction performance is stable, rendering the cumulative transmission amount almost linear to the round number; yet, APF can achieve larger reductions in later rounds by freezing more stable parameters. More importantly, Gaia and CMFL only compress the parameter transmission in the *push* phase, yet APF eliminates the transmission of stable parameters in *both pull and push* phases, which helps to make a larger communication reduction.

E. Necessity of the TCP-Style Control Mechanism

Note that a key building block of APF is a TCP-style mechanism to control the parameter freezing period, i.e., *additively increase* or *multiplicatively decrease* the freezing period of each parameter based on whether that parameter keeps stable after being unfrozen. Then, is it necessary or can we replace it with a simpler mechanism? To explore that, we replace it with three different control mechanisms: *pure-additive*—increase or decrease the freezing period always additively; *pure-multiplicative*—increase or decrease the freezing period always multiplicatively; and *fixed*—freeze each stabilized parameter for a fixed period (across 10 stability checks or for $10 \times F_c$ iterations in our experiment).

Fig. 15 shows the validating accuracy and instantaneous stable ratio under each scheme for LeNet-5. From Fig. 15b, we find that different control schemes exhibit a similar performance on stable ratio– - meaning that they can reduce the overall communication volume to a similar extent; yet on the other hand, our TCP-style mechanism—by freezing parameters tentatively and unfreezing them agilely once parameter shifting occurs—can avoid the negative impact of parameter freezing, and as as shown in Fig. 15a, thus yield the best testing accuracy.

F. Effectiveness of APF# and APF++

Recall that in Section V we propose two APF extensions, APF# and APF++, to attain larger transmission reduction with more aggressive parameter freezing. In this subsection we evaluate their effectiveness with micro-benchmark measurements in a cluster of 10 nodes (under the SGD optimizer with $F_c = F_s$).

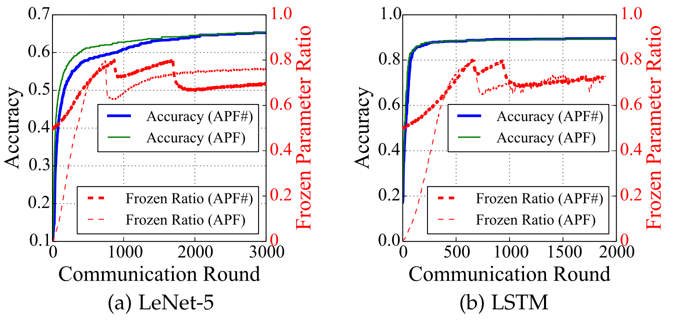


Fig. 16. Compared with vanilla APF, APF# can further increase the average parameter freezing ratio (from 66.5% to 70.2% for LeNet-5, and from 62.4% to 71.3% for LSTM) with accuracy preserved.

In Fig. 16, we show the training performance of LeNet-5 and LSTM under APF# against that under vanilla APF. To be specific, in APF# we randomly freeze those unstable parameters for 1 round with a probability of 0.5. Our measurements show that APF# achieves a similar accuracy performance as APF (the training improvement in the earlier phase under APF# is slower but would catch up in later phase, similar to the well-known behavior of Dropout), with a better communication compression level. For LeNet-5 that additional performance gain is 5.5% and for LSTM that is 14.2%.

In Fig. 17, we show the training performance of LeNet-5 and ResNet-18 under APF++ against vanilla APF. In our setup, an active parameter is put to freezing with a probability of $\frac{K}{4000}$ (for LeNet-5) or $\frac{K}{2000}$ (for ResNet-18), and the associated freezing length is randomly sampled from the interval $[1, 1 + \frac{1}{20}K]$, where K is the round number. In Fig. 17a, while APF++ can remarkably reduce the transmission amount, its aggressiveness unfortunately compromises the model accuracy because LeNet-5 is not over-parameterized for CIFAR-10 dataset; in contrast, in Fig. 17b APF++ can substantially improve the parameter frozen ratio of ResNet-18 (up to an average value of 77%), without hurting the accuracy performance. Therefore, for over-parameterized large models, it is suitable to adopt APF++ for much better communication efficiency.

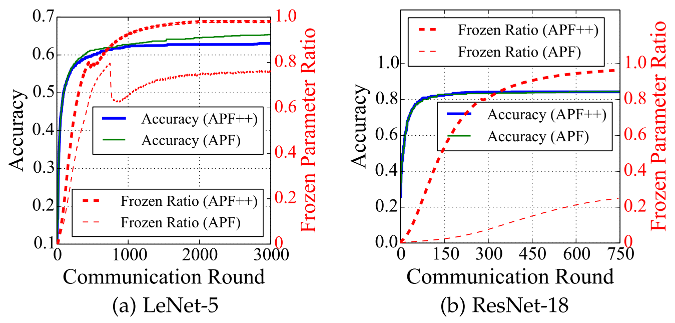


Fig. 17. APF++ can substantially enhance the communication compression level, without hurting ResNet accuracy.

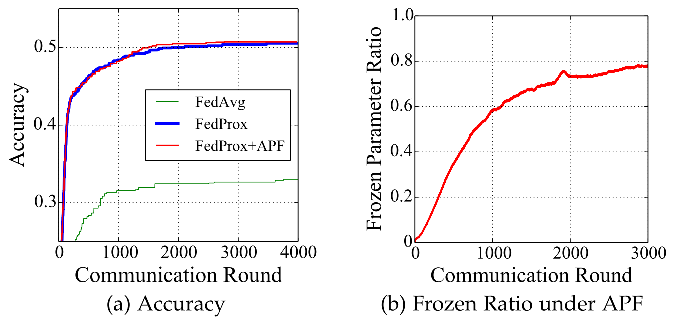


Fig. 19. Training performance respectively under FedAvg, FedProx and FedProx+APF.

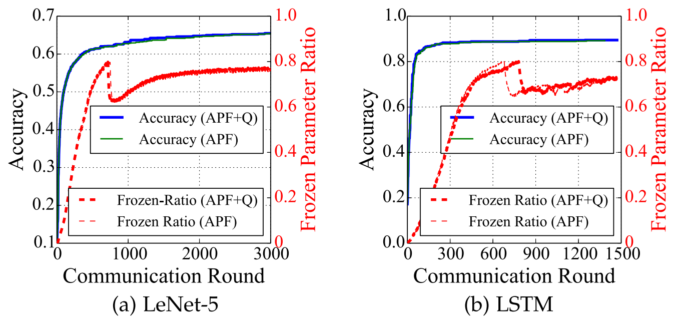


Fig. 18. Training performance respectively under APF and APF+Quantization (APF+Q).

G. Combination With Other Optimization Techniques

As a sparsification method, APF can be combined with other non-sparsification methods to further enhance FL performance. In this subsection, we evaluate the APF performance when combined with a simple quantization method as well as the FedProx optimization method [35].

Combining APF With Quantization. While APF works by reducing the number of parameters to be transmitted, quantization means to reduce the cost to transmit each parameter, and thus the two methods can be adopted simultaneously. To verify that feasibility, we integrate APF with a simple quantization method—using 16 bits instead of the default 32 bits to represent each parameter.

In our implementation to realize such a combination, we create a `Quantization_Manager` and stack it atop the `APF_Manager` in Fig. 10. In the pushing phase, after the filtering operation of APF, the `Quantization_Manager` takes over the compressed parameters and calls `Tensor.half()` provided by PyTorch to perform a second compression. In the pulling phase, the `Quantization_Manager` first restores the full precision, then the `APF_Manager` restores the full model.

In Fig. 18, we show the APF convergence curve of LeNet and LSTM respectively with and without Quantization. We find that APF with Quantization (APF+Q)—with only a half of the initial communication cost—exhibits a very similar accuracy and stability performance as vanilla APF, echoing the observation of quantization impact in [5]. Overall, APF+Q can reduce the communication amount by 83.2% for LeNet model and 81.6% for LSTM model. It can be expected that, such improvements can be even larger if more aggressive quantization levels are adopted.

Combination With FedProx. FedProx [35] is a federated optimization algorithm to address the challenges related to *system* and *statistical* heterogeneity. In typical FL scenarios, participants with inferior resources would become stragglers, and they can only process a portion of the local samples at the expected synchronization barriers. At that moment, both *dropping* (as in `FedAvg`) or *naively incorporating* those stragglers' intermediate results would increase the statistical heterogeneity and adversely impact convergence behavior. To improve training efficiency and stability, FedProx also includes the intermediate results from stragglers—yet with a special proximal term added to the objective function to ensure convergence validity. That is, in round $k+1$ starting from global model $\mathbf{x}_k$, each client i under FedProx would optimize $h^i(\mathbf{x}, \mathbf{x}_k) = F^i(\mathbf{x}) + \frac{\mu}{2} \parallel \mathbf{x} - \mathbf{x}_k \parallel^2$ instead of $F^i(\mathbf{x})$.

To confirm the feasibility of integrating APF with FedProx, we create a FL micro-benchmark with both system and statistical heterogeneity, following a method similar to the FedProx paper [35]. In our measurements, we train LeNet-5 on CIFAR-10 datasets with 5 clients each hosting 2 label classes. Among the clients there are two stragglers that can only process 25% and 50% of the expected workloads in each round. In Fig. 19, we respectively measure the training performance under (1) FedAvg, (2) FedProx, and (3) FedProx+APF; here the FedProx hyperparameter μ is set to the recommended value 0.01. From Fig. 19a, by fully incorporating the results of all the clients, FedProx achieves much higher accuracy performance over FedAvg. Moreover, when APF is adopted with FedProx, we can attain similar accuracy performance with much less communication cost: In average, APF freezes around 55.0% of all the parameters over the training process. Therefore, APF can be effectively combined with FedProx to achieve high accuracy as well as low overhead for FL.

H. Hyper-Parameter Sensitivity Analysis

In this subsection, we systematically evaluate the sensitivity of APF performance against various hyper-parameters (e.g., stability threshold T_s, stability check frequency F_c, learning

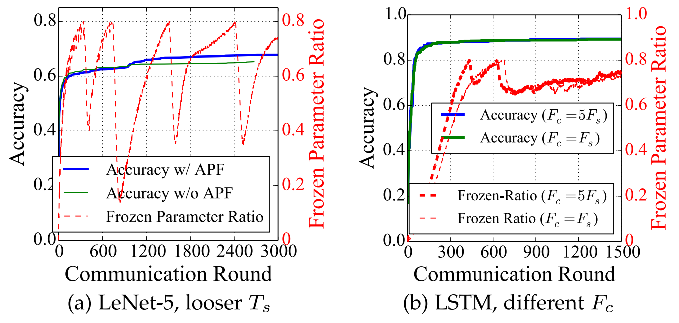


Fig. 20. When the initial stability threshold is more loose, or when the stability check is less frequent, LeNet-5 and LSTM can still converge well.

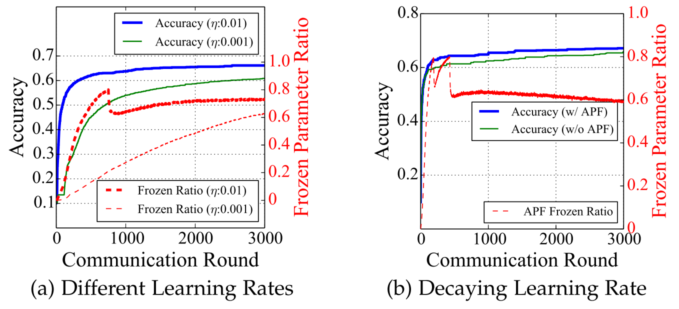


Fig. 21. APF performance when training LeNet-5 with different or decaying learning rates.

rate η, and the synchronization frequency F_s). Unless otherwise specified, we experiment with the LeNet-5 model which is more common in the literature.

Stability Threshold. As discussed in Section IV-B, to make APF robust to stability threshold, we have introduced a mechanism that tightens the stability threshold each time most (e.g., 80% in our setup) parameters become stable. To verify the effectiveness of this approach, we purposely loosen the initial stability threshold on effective perturbation from 0.05 to 0.5, and then train LeNet-5 again following the setup in Section VII-A. Fig. 20a depicts the corresponding accuracy curve, together with the instantaneous ratio of frozen parameter. Comparing Fig. 20a with Fig. 11a we can easily find that, under a looser initial stability threshold, the number of frozen parameters increases more rapidly, but the price paid is that the instantaneous accuracy is slightly below that of standard FL (before round-600). However, after several tightening actions, APF gradually catches up with—and finally outperforms—standard FL in accuracy. Therefore, our APF algorithm can still yield good performance even with inadequate hyper-parameters.

Stability Check Frequency. To evaluate the impact of the stability check frequency, we resort to an experiment as shown in Fig. 20b. In Fig. 20b, we train the LSTM model with F_c respectively be F_s and $5F_s$, the later meaning that we update the freezing status once for 5 rounds. Correspondingly, for fair comparison, when F_c is 5 we increase the freezing round by a constant of 5 (instead of 1) if the parameter stability continues, and adopt a scale-down factor of 5 (instead of 2) if the stability trend no longer persists. From Fig. 20b, the two different F_c setups yield a similar training performance in both accuracy and stability ratio, confirming the robustness of our APF algorithm against the F_c hyper-parameter.

Learning Rate. We further evaluate the performance sensitivity of APF against different learning rate setups. In Fig. 21a, we respectively set the learning rate η to 0.01 and 0.001, and measure the accuracy and frozen ratio of APF in each case. Fig. 21a reveals that with a larger learning rate, the model accuracy is enhanced at a faster pace, and in the meantime the model parameters get stable more rapidly.

Recall that Theorem 2 suggests APF can yield better convergence guarantee with a decaying learning rate (although our experiments empirically show that APF does not compromise the model accuracy even without such a decay). To set up the decaying learning rate, we initialize η to 0.1 and multiply it with a factor of 0.99 once after every 10 epochs, and Fig. 21b shows the training performance with and without APF. We notice that, the general trend of model accuracy and parameter frozen ratio with learning rate decay are similar as before except for two minor differences. First, with a larger learning rate in the beginning (0.1 in Fig. 21b against 0.01 in Fig. 21a), the model parameters become stable much more rapidly. Second, during the later phase the parameter frozen rate begins to drop slowly—this is because a decaying learning rate would allow the model parameters to keep refined subtly. More importantly, with such a decay learning rate, the accuracy benefit of APF against vanilla FedAvg becomes even larger: After 3000 rounds, APF attains an accuracy superiority of 0.033—with an accumulated communication reduction of 61.9%. To summarize, APF can make salient performance improvement regardless of the particular learning rate setup.

Synchronization Frequency. Note that in this article we use synchronization frequency F_s to control the number of local iterations within each round; here F_s is equivalent to the *number-of-local-epochs* (E) hyper-parameter in other FL papers [32], [35]—because all clients share the same batch size and local dataset size. We then evaluate how F_s would affect the APF performance. Our experiments are conducted under a non-IID setup with 5 clients each hosting two CIFAR-10 classes, because in the literature [36], [56] F_s may severely impact the training performance for cases with non-IID data. In our measurements, we set F_s respectively to 10, 100 and 500, and depict the accuracy and parameter frozen ratio in Fig. 22. On the one hand, with less frequent synchronization ($F_s = 500$), parameters would be refined more extensively within each round, therefore the model accuracy and the parameter frozen ratio would increase faster in the number of communication rounds. On the other hand, less frequent synchronization would make the aggregated model updates less accurate, thus in Fig. 22a the model training process with $F_s = 500$ stagnates at a suboptimal accuracy.

I. Computation and Memory Overheads

The last question we seek to answer is, what are the memory and computation overheads of APF? We measure the extra computation time and physical memory consumption incurred

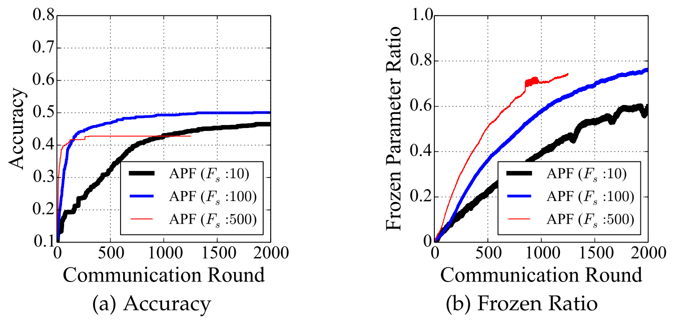


Fig. 22. APF performance under different synchronization frequency setups.

TABLE IV
COMPUTATION (EXTRA TIME REQUIRED FOR EACH ROUND IN AVERAGE) AND MEMORY OVERHEADS OF APF

Model	LeNet-5	ResNet-18	LSTM
Computation Time Incurred by APF (per-round)	0.009 s	1.278 s	0.011 s
Computation Time Inflation Ratio incurred by APF	1.93%	4.50%	1.42%
Memory Occupied for APF Processing	1.2 MB	142 MB	4.8 MB
Memory Inflation Ratio incurred by APF	0.18%	8.51%	2.35%

by APF. Table IV lists the overheads for different models. As implied in Table IV, for small models like LeNet and LSTM, both the computation and memory overheads are fairly small ($< 2.35\%$); for larger models like ResNet-18, because its model size is more comparable to that of input and feature map, APF's memory overhead becomes larger (8.51%). Overall, considering the salient reduction of network transmission volume and convergence time, the overhead of APF is indeed acceptable.

VIII. Additional Related Work

In this article, we developed a parameter-freezing method with the objective of mitigating the model synchronization bottleneck. In this section, we briefly survey the related work that shares a similar objective (i.e., speeding up model synchronization with either communication compression or transmission scheduling), or share a similar solution methodology (i.e., with parameter-freezing).

Communication Compression Techniques. Recall that in Section II we have discussed a series of sparification and quantization methods. Apart from sparsification and quantization, a third solution category for communication compression is *matrix factorization* [61], [67]. Matrix factorization works by decoupling the original model parameters as the outer product of two vectors (called *sufficient factors*). In model synchronization, only those sufficient factors are transmitted to reduce the bandwidth consumption. Yet, that decoupling method is mainly designed for dense fully-connected layers, thus suffering restricted applicability for general models.

Communication Scheduling Techniques. To speed up model synchronization, another common methodology is to optimize the transmission schedule of gradients from different clients or layers.

Regarding the transmission coordination of different clients, by default all clients shall communicate their gradients in synchronous mode (with Bulk Synchronous Parallel or BSP) [67], yet it may cause severe resource wastage due to stragglers and network collisions. To avoid such wastage, some works respectively proposed SSP [16], [25] and R ^{2} SP [12] to properly relax the synchronization barrier, trading a little bit gradient accuracy for improved resource efficiency.

Regarding the transmission coordination of different layers, to reduce the time blocked by gradient transmission, Zhang et. al [50], [67] proposed wait-free-backpropagation, which pipelines gradient communication with the layer-wise gradient computation. Besides, parameters that are updated later in backward propagation would be accessed earlier in the next iteration; noticing that property, several works [22], [29], [46] proposed to assign proper priorities to gradients of different layers, so that gradients to be accessed earlier do not need to wait behind others in network contention. These works achieve good performance in practice, yet they are orthogonal to our work.

Parameter Freezing Techniques. Parameter freezing is a technique already adopted in the literature—yet mainly for computation acceleration. Brock et al. [9] proposed the FreezeOut mechanism that gradually froze the first few layers of a deep neural network—which were observed to converge faster—to avoid the computation cost of calculating their gradients. KGT [57] and AutoFreeze [37] go further by using an adaptive approach to choose which layers are frozen, so as to accelerate model training while preserving accuracy. However, their operation granularity is an entire layer, which is too coarse given the analysis in Section III. Worse, because there was no parameter unfreezing mechanism, it has been shown that such methods like FreezeOut would degrade the accuracy performance despite the computation speedup [9].

IX. Discussions

APF compatibility with differential privacy. So far, we are assuming that the original gradients are accessible over the FL process; yet, to avoid privacy leakage, gradients are often encrypted before being collected to the FL server, and a typical encryption method is *differential privacy* (DP) [45], [51], [60]. Differential privacy means to add random noise to the original gradients. That noise follows a Gaussian or Laplace distribution whose mean is set to zero, and the noise scale is controlled by a hyper-parameter ε. Regarding the impact of DP on our APF algorithm, we note that the added noise—which oscillates around 0—would make the resultant gradients look more stable, i.e., rendering the calculated effective perturbation metric smaller. For example, if in extreme case the noise component dominates the gradient value, the effective perturbation would also be 0. Yet, large noise would severely impair the model accuracy, and in practice the injected noise is set much smaller than the original gradient values [60]. Therefore, when DP is adopted, we can choose a tighter stability threshold to counteract its impact.

Placement of Freezing Mask Computation. Note that in our implementation (Section VI-B), the freezing mask $M_{\text{is\_frozen}}$ is maintained purely on clients, with the objective of trading slightly more computation for communication efficiency (no extra communication cost). In other cases where computation is more expensive on clients (like IoT devices), we can also place mask computations on the FL server (or an edge server). Besides, instead of transmitting the full mask vector, we can otherwise transfer a dense representation (including change-indexes and change-values), for mask updating would be relatively slow and a dense representation is more efficient.

Applicability in cluster environments. While APF is proposed for FL scenarios, it can also be applied to cluster environments with cutting-edge hardware like GPUs, where communication remains to be a bottleneck compared to the large GPU throughput. Yet, compared with FL scenarios, models trained in such clusters are usually much more complex, including various structure types like GNNs [54] and Transformers [18]; meanwhile, achieving SOTA accuracy is often a primary concern, rendering manufactured training interference more risky. We plan to customize our work to GPU clusters later, with much broader evaluations on advanced deep learning models.

X. Conclusion

In this work, to reduce the communication overhead in FL, we have proposed a novel scheme called Adaptive Parameter Freezing (APF), which seeks to identify the stable parameters and then avoid their synchronization. APF identifies the stabilized parameters based on their effective perturbation and tentatively freezes the stable parameters for certain time intervals, which are adjusted in an additively-increase and multiplicatively-decrease manner. We implemented APF and its more aggressive variants based on PyTorch, and testbed experiments have confirmed that it can largely improve the FL communication efficiency, with comparable or even better accuracy performance.

Acknowledgments

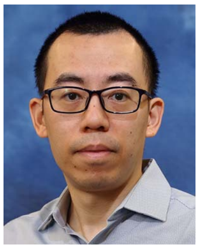
The preliminary version of this article, titled "Communication -Efficient Federated Learning with Adaptive Parameter Freezing" [13], was published in IEEE ICDCS 2021.

References

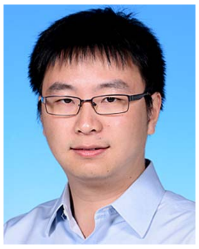
[1] Global Internet Condition, 2019. [Online]. Available: https://www.atlasandboots.com/remote-jobs/countries-with-the-fastest-internet-in-the-world
[2] PyTorch, 2022. [Online]. Available: https://pytorch.org/
[3] M. Abadi et al., "TensorFlow: A system for large-scale machine learning," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2016, pp. 265–283.
[4] S. Agarwal, H. Wang, K. Lee, S. Venkataraman, and D. Papailiopoulos, "Adaptive gradient communication via critical learning regime identification," in *Proc. Mach. Learn. Syst.*, vol. 3, 2021, pp. 55–80.
[5] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "QSGD: Communication-efficient SGD via gradient quantization and encoding," in *Proc. 31st Int. Conf. Neural Inf. Process. Syst.*, 2017, pp. 1707–1718.
[6] M. Allman, V. Paxson, and E. Blanton, "TCP congestion control," Tech. Rep. Rfc 5681, 2009.
[7] K. Bonawitz et al., "Towards federated learning at scale: System design," in *Proc. Mach. Learn. Syst.*, vol. 1, 2019, pp. 374–388.
[8] K. Bonawitz et al., "Towards federated learning at scale: System design," 2019, *arXiv:1902.01046*.
[9] A. Brock, T. Lim, J. M. Ritchie, and N. Weston, "Freezeout: Accelerate training by progressively freezing layers," 2017, *arXiv:1706.04983*.
[10] V. G. Cerf and R. E. Kahn, "A protocol for packet network intercommunication," *IEEE Trans. Commun.*, vol. 22, no. 5, pp. 637–648, May 1974.
[11] J. Chee and P. Toulis, "Convergence diagnostics for stochastic gradient descent with constant learning rate," in *Proc. 21st Int. Conf. Artif. Intell. Statist.*, 2018, pp. 1476–1485.
[12] C. Chen, W. Wang, and B. Li, "Round-robin synchronization: Mitigating communication bottlenecks in parameter servers," in *Proc. IEEE Conf. Comput. Commun.*, 2019, pp. 532–540.
[13] C. Chen et al., "Communication-efficient federated learning with adaptive parameter freezing," in *Proc. IEEE Int. Conf. Distrib. Comput. Syst.*, 2021, pp. 1–11.
[14] T. Chen et al., "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," 2015, *arXiv:1512.01274*.
[15] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, "Natural language processing (almost) from scratch," *J. Mach. Learn. Res.*, vol. 12, pp. 2493–2537, 2011.
[16] H. Cui et al., "Exploiting iterative-ness for parallel ML computations," in *Proc. ACM Symp. Cloud Comput.*, 2014, pp. 1–14.
[17] M. Denil et al., "Predicting parameters in deep learning," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2013, pp. 2148–2156.
[18] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics: Hum. Lang. Technol.*, 2019, pp. 4171–4186.
[19] J. Ba Diederik and P. Kingma, "Adam: A method for stochastic optimization," 2017, *arXiv:1412.6980*.
[20] N. Dryden, T. Moon, S. A. Jacobs, and B. Van Essen, "Communication quantization for data-parallel training of deep neural networks," in *Proc. IEEE 2nd Workshop Mach. Learn. HPC Environ.*, 2016, pp. 1–8.
[21] Z. Du, K. Palem, A. Lingamneni, O. Temam, Y. Chen, and C. Wu, "Leveraging the error resilience of machine-learning applications for designing highly energy efficient accelerators," in *Proc. IEEE 19th Asia South Pacific Des. Automat. Conf.*, 2014, pp. 201–206.
[22] S. H. Hashemi, S. A. Jyothi, and R. H. Campbell, "TICTAC: Accelerating distributed deep learning with communication scheduling," in *Proc. Mach. Learn. Syst.*, vol. 1, 2019, pp. 418–430.
[23] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.
[24] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," 2012, *arXiv:1207.0580*.
[25] Q. Ho et al., "More effective distributed ML via a stale synchronous parallel parameter server," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2013, pp. 1223–1231.
[26] S. Hochreiter and J. Schmidhuber, "Flat minima," *Neural Comput.*, vol. 9, no. 1, pp. 1–42, 1997.
[27] K. Hsieh et al., "Gaia: Geo-distributed machine learning approaching LAN speeds," in *Proc. 14th USENIX Symp. Netw. Syst. Des. Implementation*, 2017, pp. 629–647.
[28] G. Huang, Y. Sun, Z. Liu, D. Sedra, and K. Q. Weinberger, "Deep networks with stochastic depth," in *Proc. Eur. Conf. Comput. Vis.*, 2016, pp. 646–661.
[29] A. Jayarajan, J. Wei, G. Gibson, A. Fedorova, and G. Pekhimenko, "Priority-based parameter propagation for distributed DNN training," in *Proc. Mach. Learn. Syst.*, vol. 1, 2019, pp. 132–145.
[30] T. Johnson, P. Agrawal, H. Gu, and C. Guestrin, "AdaScale SGD: A user-friendly algorithm for distributed training," in *Proc. Int. Conf. Mach. Learn.*, 2020, pp. 4911–4920.
[31] K. Kawaguchi, "Deep learning without poor local minima," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2016, pp. 586–594.
[32] J. Konečný, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon, "Federated learning: Strategies for improving communication efficiency," 2016, *arXiv:1610.05492*.
[33] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2012, pp. 1106–1114.
[34] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

[35] T. Li, A. K. Sahu, M. Zaheer, M. Sanjabi, A. Talwalkar, and V. Smith, "Federated optimization in heterogeneous networks," in *Proc. Mach. Learn. Syst.*, vol. 2, 2020, pp. 429–450.
[36] X. Li, K. Huang, W. Yang, S. Wang, and Z. Zhang, "On the convergence of FedAvg on non-IID data," 2019, *arXiv:1907.02189*.
[37] Y. Liu, S. Agarwal, and S. Venkataraman, "Autofreeze: Automatically freezing model blocks to accelerate fine-tuning," 2021, *arXiv:2102.01386*.
[38] H. Mania, X. Pan, D. Papailiopoulos, B. Recht, K. Ramchandran, and M. I. Jordan, "Perturbed iterate analysis for asynchronous stochastic optimization," 2015, *arXiv:1507.06970*.
[39] S. McCandlish, J. Kaplan, D. Amodei, and OpenAI Dota Team, "An empirical model of large-batch training," 2018, *arXiv:1812.06162*.
[40] H. B. McMahan et al., "Communication-efficient learning of deep networks from decentralized data," 2016, *arXiv:1602.05629*.
[41] E. Moulines and F. Bach, "Non-asymptotic analysis of stochastic approximation algorithms for machine learning," in *Proc. Adv. Neural Inf. Process. Syst.*, 2011, pp. 451–459.
[42] N. Murata, "A statistical study of on-line learning," *Online Learning and Neural Networks*. ,Cambridge, UK: Cambridge Univ. Press, 1998, pp. 63–92.
[43] B. Neyshabur, Z. Li, S. Bhojanapalli, Y. LeCun, and N. Srebro, "Towards understanding the role of over-parametrization in generalization of neural networks," 2018, *arXiv:1805.12076*.
[44] K. Osawa, Y. Tsuji, Y. Ueno, A. Naruse, R. Yokota, and S. Matsuoka, "Large-scale distributed second-order optimization using kronecker-factored approximate curvature for deep convolutional neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2019, pp. 12359–12367.
[45] M. A. Pathak, S. Rane, and B. Raj, "Multiparty differential privacy via aggregation of locally trained classifiers," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2010, pp. 1876–1884.
[46] Y. Peng et al., "A generic communication scheduler for distributed dnn training acceleration," in *Proc. Symp. Operating Syst. Princ.*, 2019, pp. 16–29.
[47] A. Qiao et al., "Pollux: Co-adaptive Cluster scheduling for goodput-optimized deep learning," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, vol. 21, 2021, pp. 1–18.
[48] 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 *Proc. IEEE/ACM 49th Annu. Int. Symp. Microarchitecture*, 2016, pp. 18:1–18:13.
[49] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu, "1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns," in *Proc. 15th Annu. Conf. Int. Speech Commun. Assoc.*, 2014, pp. 1058–1062.
[50] S. Shi and X. Chu, "MG-WFBP: Efficient data communication for distributed synchronous sgd algorithms," in *IEEE Conf. Comput. Commun.*, 2019, pp. 172–180.
[51] R. Shokri and V. Shmatikov, "Privacy-preserving deep learning," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 1310–1321.
[52] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Machi. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, 2014.
[53] N. Strom, "Scalable distributed DNN training using commodity GPU cloud computing," in *Proc. Annu. Conf. Int. Speech Commun. Assoc.*, 2015, pp. 1488–1492.
[54] J. Thorpe et al., "Dorylus: Affordable, scalable, and accurate GNN training with distributed CPU servers and serverless threads," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2021, pp. 495–514.
[55] L. Wang, W. Wang, and B. Li, "Cmfl: Mitigating communication overhead for federated learning," in *Proc. IEEE Int. Conf. Distrib. Comput. Syst.*, 2019, pp. 954–964.
[56] S. Wang et al., "Adaptive federated learning in resource constrained edge computing systems," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 6, pp. 1205–1221, Jun. 2019.
[57] Y. Wang, D. Sun, K. Chen, F. Lai, and M. Chowdhury, "Efficient DNN training with knowledge-guided layer freezing," 2022, *arXiv:2201.06227*.
[58] P. Warden, "Speech commands: A dataset for limited-vocabulary speech recognition," 2018, *arXiv:1804.03209*.
[59] W. Wen et al., "TernGrad: Ternary gradients to reduce communication in distributed deep learning," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2017, pp. 1508–1518.
[60] N. Wu, F. Farokhi, D. Smith, and M. A. Kaafar, "The value of collaboration in convex machine learning with differential privacy," in *Proc. IEEE Symp. Secur. Privacy*, 2020, pp. 304–317.
[61] P. Xie et al., "Lighter-communication distributed machine learning via sufficient factor broadcasting," in *Proc. 32nd Conf. Uncertainty Artif. Intell.*, 2016, pp. 795–804.
[62] G. Yan, H. Wang, and J. Li, "Seizing critical learning periods in federated learning," in *Proc. AAAI Conf. Artif. Intell.*, 2022, pp. 8788–8796.
[63] T. Yang et al., "Applied federated learning: Improving google keyboard query suggestions," 2018, *arXiv:1812.02903*.
[64] M. Yurochkin, M. Agarwal, S. Ghosh, K. Greenewald, T. N. Hoang, and Y. Khazaeni, "Bayesian nonparametric federated learning of neural networks," in *Proc. Int. Conf. Mach. Learn.*, 2019, pp. 7252–7261.
[65] S. Zagoruyko and N. Komodakis, "Wide residual networks," 2016, *arXiv:1605.07146*.
[66] M. D. Zeiler, "ADADELTA: An adaptive learning rate method," 2012, *arXiv:1212.5701*.
[67] H. Zhang et al., "Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters," in *Proc. USENIX Annu. Tech. Conf.*, 2017, pp. 181–193.
[68] Y. Zhao, M. Li, L. Lai, N. Suda, D. Civin, and V. Chandra, "Federated learning with non-iid data," 2018, *arXiv:1806.00582*.

Chen Chen (Member, IEEE) received the BEng degree from Tsinghua University, in 2014, and the PhD degree from the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, in 2018. He is an associate professor with John Hopcroft Center for Computer Science, Shanghai Jiao Tong University. His recent research interests include distributed deep learning, big data systems and networking. He previously worked as a researcher with the Theory Lab, Huawei Hong Kong Research Center.



Hong Xu (Senior Member, IEEE) received the BEng degree from the Chinese University of Hong Kong, in 2007, and the MASc and PhD degrees from the University of Toronto, in 2009 and 2013, respectively. He is an associate professor with the Department of Computer Science and Engineering, The Chinese University of Hong Kong. His research area is computer networking and systems, particularly big data systems and data center networks. From 2013 to 2020 he was with City University of Hong Kong. He was the recipient of an Early Career Scheme Grant from the Hong Kong Research Grants Council in 2014. He received three best paper awards, including the IEEE ICNP 2015 best paper award. He is a senior member of ACM.



Wei Wang (Member, IEEE) received the BEngr and MEngr degrees from the Department of Electrical Engineering, Shanghai Jiao Tong University, China, in 2007 and 2010, respectively, and the PhD degree from the Department of Electrical and Computer Engineering, University of Toronto, Canada, in 2015. Since 2015, he has been with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology (HKUST), where he is currently an associate professor. He is also affiliated with the HKUST Big Data Institute. His research interests cover the broad area of distributed systems, with focus on serverless computing, machine learning systems, and cloud resource management. He published extensively in the premier conferences and journals of his fields. His research has won the Best Paper Runner Up awards of IEEE ICDCS 2021 and USENIX ICAC 2013.

Baochun Li (Fellow, IEEE) received the PhD degree from the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, in 2000. Since then, he has been with the Department of Electrical and Computer Engineering at the University of Toronto, where he is currently a professor. He holds the Bell Canada Endowed Chair in computer engineering since August 2005. His research interests include large-scale distributed systems, machine learning, security, cloud computing, and wireless networks. He is a member of ACM.

Bo Li (Fellow, IEEE) received the BEng. (summa cum laude) degree in computer science from Tsinghua University, Beijing, and the PhD degree in electrical and computer engineering from the University of Massachusetts at Amherst. He is a chair professor with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology. He held a Cheung Kong visiting chair professor in Shanghai Jiao Tong University between 2010 and 2016, and was the chief technical advisor for ChinaCache Corp. (NASDAQ:CCIH), a leading CDN provider. He was an adjunct researcher with the Microsoft Research Asia (MSRA) (1999-2006) and with the Microsoft Advanced Technology Center (2007-2008). He made pioneering contributions in multimedia communications and the Internet video broadcast, in particular Coolstreaming system, which was credited as first large-scale Peer-to-Peer live video streaming system in the world. It attracted significant attention from both industry and academia and received the Test-of-Time Best Paper Award from IEEE INFOCOM (2015). He has been an editor or a guest editor for more than a two dozen of IEEE and ACM journals and magazines. He was the Co-TPC chair for IEEE INFOCOM 2004.

Li Chen (Member, IEEE) received the BE , MPhil and PhD degrees from the Hong Kong University of Science and Technology (HKUST), in 2011, 2013 and 2018, respectively. He previously worked as a systems researcher with Huawei Theory Lab. His research interests include data center networks, distributed systems, vert computing, machine learning and OAM. He now works with Zhongguancun Laboratory.

Gong Zhang (Member, IEEE) is a principal researcher with Huawei 2012 Labs. He has more than 18 years research experience in network communication and distributed system, and has contributed more than 90 patents globally. He was a team leader for future Internet and cooperative communication research since 2005, was in charge of Advance Network Technology Research Department, in 2009, and led the System Group in data mining and machine learning since 2012. His recent research directions are network architecture and large-scale distributed systems.