Communication-Efficient Federated Learning with Adaptive Parameter Freezing

***Abstract*—Federated learning allows edge devices to collaboratively train a global model by synchronizing their local updates without sharing private data. Yet, with limited network bandwidth at the edge, communication often becomes a severe bottleneck. In this paper, we find that it is not necessary to synchronize all the model parameters in the training process, because many of them gradually stabilize prior to the ultimate model convergence, and can thus be excluded from synchronization at an early stage. This allows us to reduce the communication overhead without compromising the model accuracy. However, the challenges are that the parameters excluded from global synchronization may diverge locally, 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 if the previously-frozen parameters remain stable in subsequent iterations. We implemented APF as a Python module in PyTorch. Our extensive array of experimental results show that APF can reduce data transfer by over 60%.**

I. Introduction

Federated learning (FL) [27], [29] emerges as a promising paradigm that allows edge devices (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 rounds of communication. 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 [33], [27], [7], and some other works propose to *sparsify* model updates by filtering out small values [36], [17], [23], [37]. Nonetheless, those works assume that all the model parameters should be updated indiscriminately in each communication round, which we find is unnecessary. In our testbed measurements, many parameters become stable long before the model converges. After these parameters reach their optimal values, subsequent updates for them are simply oscillations with little changes, and can be safely excluded without harming the model accuracy.

Therefore, an intuitive idea we want to explore is, can we reduce FL communication amount by no longer synchronizing those stabilized parameters? Realizing this intuition raises two challenges: (1) How to efficiently identify the stabilized parameters in resource-constrained edge devices? (2) Can we preserve the model convergence accuracy after eliminating the stabilized parameters from being synchronized?

Regarding the first challenge, we propose a stability metric called effective perturbation to quantify—from a historical point of view—how closely the consecutive updates of a parameter follow an oscillation pattern. With an exponential moving average method, we make the effective perturbation metric efficient in computation and memory consumptions, and rely on it to identify the stable parameters.

For the second challenge, we experimentally find that some strawman designs 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 and cause model inconsistency. Another method that permanently fixing (i.e., freezing) the stabilized parameters can ensure model consistency at the cost of model accuracy: some parameters may stabilize only *temporally* during the model 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), that adaptively freezes and unfreezes parameters with the objective of reducing communication volume while preserving convergence. Under APF, each stable parameter is frozen for a certain number of rounds and then unfrozen to see if they need further training; the length of such freezing period is *additively increased* or *multiplicatively decreased* based on whether that parameter is still stable after resuming updating. This way, APF enables temporarily-stable parameters to resume training timely while keeping the converged parameters frozen for most of the time. Additionally, for large models that are over-parameterized [30], [40], we design an extended version of APF, called *Aggressive APF*, that can attain a much larger performance improvement by allowing some unstable parameters to enter frozen state randomly.

We implement APF as a pluggable Python module named `APF_Manager` atop the PyTorch framework. The module takes over the model update from edge clients and communicates only the non-frozen part 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% without compromising the model convergence. In some cases APF can even improve the convergence accuracy because parameter freezing avoids over-fitting and improves the model generalization capability. Meanwhile, our breakdown measurements show that the

computation and memory overheads of APF are negligible compared to its benefits in training speedup.

II. Background

Given a neural network model, the objective of training is to find the optimal parameters $\omega^\star$ that can minimize a global loss function, $F(\omega)$, over the entire training dataset D. Let $f(s,\omega)$ be the loss value computed from sample s with parameters ω, then that objective can be expressed as:

$$\omega^\star = \arg\min F(\omega) = \arg\min \frac{1}{|D|} \sum\nolimits_{s\in D} f(s,\omega). \quad (1)$$

A common algorithm to find $\omega^\star$ is *stochastic gradient descent* (SGD) [28]. SGD works in an iterative manner: in iteration k, ω_k is updated with a gradient g_{ξ_k} that is calculated from a random sample batch ξ_k, i.e., $\omega_{k+1} = \omega_k - \eta g_{\xi_k}$.

In FL scenarios, there is a central server maintaining the global model, and the training samples are locally stored among a number of edge clients. Those clients update the global model in communication *rounds* [27], [29]. In each round, clients pull the latest model parameters from the central server and, after locally refining those parameters for *multiple iterations*, push their updated model parameters back to the server for global aggregation.

A salient performance bottleneck for FL is the communication between edge clients and the central server [27], [29], [9]. As an enabling technique that supports apps like Google Gboard with global user [39], 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]. When training LeNet-5 (a classical convolutional neural network (CNN) for image classification [28]) or ResNet-18 under a FL setup with 50 clients (see Sec. VI-A for detailed setup), 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.

There has been much research on communication compression in distributed learning scenarios. They can essentially be categorized into two types: *quantization* and *sparsification*. To be concise, here we only summarize their key insights and defer the detailed description of each category to Sec. VII. Quantization means to use lower bits to represent gradients which is originally represented in 32 or 64 bits [33], [27], [7]. Sparsification means to reduce the number of elements that are transmitted in model aggregation, and a key idea for sparsification is to have each worker transfer only the significant gradient—those that are larger than a given threshold [36], [17], [23].

Nonetheless, both approaches treat the model convergence process as a *black-box*: they implicitly assume that synchronization shall be conducted for *all* parameters in *all* communication rounds, and focus merely on reducing the resultant synchronization cost—either by adopting lower-bit representation or by filtering out small gradients. There still exists an uncharted territory: *is it necessary to synchronize all the parameters in each round*? If there exist some parameters that do not deserve synchronization during certain periods, we can fundamentally eliminate the resultant cost of their synchronization. Motivated by the potential benefits, we next explore the answer to this question with theoretical analysis and testbed measurements.

III. Motivation

In this section, we seek to find out whether each parameter needs to be synchronized in each round. We first theoretically analyze how parameters evolve during the model training process, and then verify that with testbed measurements. Finally we summarize the research objectives and challenges.

Theoretical analysis. In general, during the iterative model training process, the model parameters go through a *transient* phase where their values vary dramatically, and move towards a *stationary* phase where their values do not change substantially. Before testbed measurements, we present a theorem that supports the existence of transient phase and stationary phase. The theorem indicates that the mean squared error of SGD has a bias term originating from distance to the starting point, and a variance term from noise in stochastic gradients.

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

$$\mathbb{E}(||\omega_k - \omega^\star||^2) \leq A^k ||\omega_0 - \omega^\star||^2 + B.$$

The proof can be found in Appendix. As implied by Theorem 1, in the transient phase, ω approaches $\omega^\star$ exponentially fast in the number of iterations, and in the stationary phase, ω oscillates around $\omega^\star$. Therefore, model parameters should change remarkably in the beginning and then gradually stabilize. Next we confirm that with testbed measurements.

Testbed Measurements on Parameter Variation. We train *LeNet-5* [28] locally on the CIFAR-10 dataset with a batch size of 100, and Fig. 1 shows the values of two randomly sampled parameters after each epoch, along with the test accuracy[1] 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 plateaus, they gradually *stabilize* after around 200 and 300 epochs, respectively.

In standard FL scheme [27], [29], parameters are still updated regularly even after they stabilize. Although their values barely change in later rounds and their updates are mostly oscillatory random noises, they still consume the same communication bandwidth as before. Such communication cost is largely unnecessary, because machine learning algorithms are resilient against parameter perturbations [15], [18], [23], which can be confirmed by the success of recent advances like Dropout [6], [20], [35] and Stochastic Depth [24]. Thus, to reduce communication cost in FL, intuitively we should avoid synchronizing the stabilized parameters.

[1]For clarity, we present the best-ever accuracy instead of instantaneous accuracy (which fluctuates drastically) in this paper.

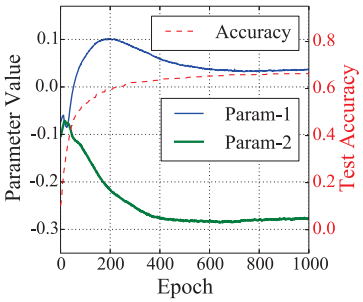


Fig. 1: Evolution of two randomly selected parameters during LeNet-5 training (with the test accuracy for reference.)

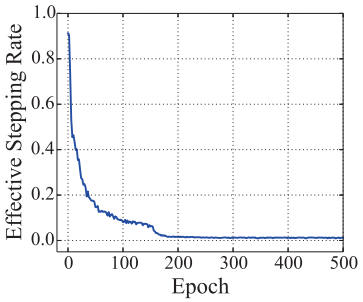


Fig. 2: Variation of the average *effective stepping rate* of all the parameters during the LeNet-5 training process.

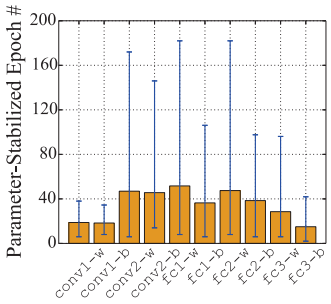


Fig. 3: Average epoch number where parameters in different layers become stable (error bars show the 5-th/95-th percentiles).

Statistical Analysis on Parameter Stability. 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. To begin with, we quantitatively describe a parameter's stability using a new metric called *effective perturbation*. It represents how a parameter's value changes out of the zigzagging steps, i.e., how the consecutive updates counteract with each other. Effective perturbation is defined over an observation window that contains a certain amount of consecutive model updates in the recent past. Formally, let $\mathbf{u}_k$ represent parameter update in iteration k (i.e., $\mathbf{u}_k = \omega_k - \omega_{k-1}$), and r be the number of recent iterations the observation window contains, then P_k^r, the *effective perturbation* at iteration k, can be defined as:

$$P_k^r = \frac{||\sum_{i=0}^{r-1} \mathbf{u}_{k-i}||}{\sum_{i=0}^{r-1} ||\mathbf{u}_{k-i}||}. \tag{2}$$

Clearly, 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, and P_k^r would be 1; if any two consecutive model updates well counteract each other, then P_k^r would be 0. Thus, with a small threshold on effective perturbation, we can identify those stabilized parameters effectively.

We then look at the evolution of 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 very slowly after the model converges at around 200 epochs (see the accuracy result in Fig. 1), suggesting that most parameters gradually stabilize before the model converges.

Granularity of Parameter Manipulation: Tensor or Scalar? By now careful readers may have noticed that a parameter in our figure presentation means a *scalar* (our theoretical analysis can be extended to scalar parameters), yet it is more natural and efficient to represent a neural network model at the granularity of *tensors* (e.g., `weight` matrix, `bias` matrix in a CNN model) instead of a vast array of scalars. Thus one may ask: do all scalars in the same tensor share the same stabilizing trend, so that we can directly deal with tensors instead?

To answer this question, we group all the scalar parameters according to the tensor they belong to, as depicted in Fig. 3. 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. For each tensor, we present the average epoch number at which its scalar values become stable. Here a scalar is deemed stable when its effective perturbation drops below 0.01, and the error bars in Fig. 3 represent the 5^{th} and 95^{th} percentiles. As indicated by Fig. 3, different tensors exhibit different stability trends; meanwhile, there also exist large gaps between the 5^{th} and 95^{th} 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 from the input may be easier to learn than the rest, and the corresponding connections in the neural network would converge faster [41]. Therefore, the granularity for parameter synchronization control needs to be individual scalars instead of tensors.

Objective and Challenges. Our objective in this work is to reduce the overall transmission volume for FL while preserving the accuracy performance. Motivated by above observations, an intuitive idea is to identify stabilized parameters and avoid their synchronization in the remaining training process.

Nonetheless, there are two immediate challenges for making this intuition a reality. First, how to efficiently detect the stabilized parameters in resource-constrained edge devices? It would be impractical to maintain in memory all the historical updates within the observation window. Second, how to guarantee model convergence when the seemingly stabilized parameters are no longer synchronized? We address these challenges in the next section.

IV. Adaptive Parameter Freezing

In this section, we present a novel mechanism to reduce FL transmission amount while preserving convergence. We first propose a resource-efficient metric to identify stable parameters and then, after exploring some candidate solutions, develop the algorithm of Adaptive Parameter Freezing (APF).

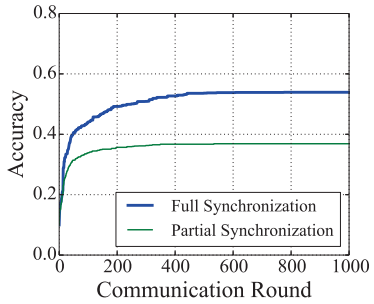


Fig. 4: *Partial synchronization* causes severe accuracy loss on non-IID data.

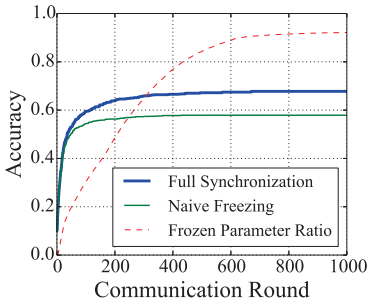


Fig. 5: *Permanent freezing* also causes accuracy loss.

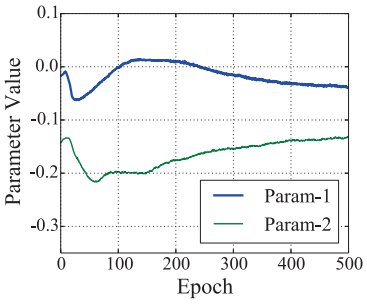


Fig. 6: The two sampled parameters stabilize only *temporarily* between epoch 100 and 200.

A. Identify Stabilized Parameters

Identifying the stabilized parameters is the first step of avoiding unnecessary parameter synchronization. Doing this on the central server would incur additional communication cost of sending the results back to clients. Therefore, we perform parameter stability check on the client side. Considering the memory and computation resource limitation on edge devices, calculating the effective perturbation defined in Eq. 2 is resource prohibitive, because it requires clients to locally store a series of past model snapshots and process all of them in each round.

To make our solution efficient in computation and memory consumption, we need to modify the previous definition of effective perturbation. First for computational efficiency, we relax the frequency of stability check from *once-per-round* to *once-for-multiple-rounds*, using the sum of all the updates since the last check. This is feasible because a stabilized parameter would remain so when observed at a coarse time granularity. Further for memory efficiency, instead of maintaining a window of previous updates, we adopt the Exponential Moving Average (EMA) method to calculate effective perturbation with the latest model update. For a parameter in the K-th stability check, let Δ_K represent the cumulative update since the last check, then the *effective perturbation* of that individual parameter, P_K, can be defined as:

$$P_K = \frac{|E_K|}{E_K^{\text{abs}}}, \text{ where } E_K = \alpha E_{K-1} + (1-\alpha)\Delta_K, \quad E_K^{\text{abs}} = \alpha E_{K-1}^{\text{abs}} + (1-\alpha)|\Delta_K|. \tag{3}$$

Here E_K is the moving average of the parameter updates, and E_K^{abs} 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 (Eq. 2)—close to 1 if model updates are of the same direction, and close to 0 if the model updates oscillate—still hold for P_K, with a much lower computation and memory cost. And in fact decaying the earlier updates with EMA is reasonable, because recent behavior is more significant. In the following part of this paper, we judge a parameter as stable if its effective perturbation under this new definition is smaller than a given stability threshold.

Regarding the setup of the stability threshold, ideally, it should be *loose enough* to include all stabilized parameters (i.e., avoid false negative), and should in the meantime be *tight enough* to avoid judging unstable parameters as being stable (i.e., avoid false positive). However, we can not assume that the stability threshold be always set appropriately. To be robust, we introduce a mechanism that adaptively tunes the stability threshold at runtime: each time the majority (e.g., 80%) of the parameters have been judged as stable, we decrease the stability threshold by one half. This method is similar with learning rate decay. This way, the negative effect of improper threshold can be gradually rectified, and the effectiveness of this approach would be verified later in evaluation.

B. Explore Candidate Solutions

Given the stabilized parameters identified with the above metric, how to treat them so that we can reap communication reduction without compromising model convergence? In this part, we explore some intuitive approaches for that and derive some key principles a valid solution must follow.

Deficiency of 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 [27], [29], [9]. This implies that the local loss function $F(\omega)$ and the corresponding local optima $\omega^\star$ on different clients are usually different [43]. 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. After each round, the stabilized parameters are excluded from later synchronization. As shown in Fig. 4, compared to FL with full synchronization, partial synchronization suffers an accuracy loss of more than 10%. Therefore, one principle any valid solution must follow

is that, *stabilized (i.e., unsynchronized) parameters must be kept unchanged on each client* (**Principle-1**).

Deficiency of Permanent Freezing. Given Principle-1, an intuitive method next would be to fix (i.e., freeze) the stabilized parameters to their current values. This should solve the parameter divergence problem. However, when training LeNet-5 with such a *permanent freezing* method with 2 clients, we find that the accuracy performance, as depicted in Fig. 5, is still suboptimal compared to full synchronization.

To understand the reasons behind we look into the parameter evolution process during training. Interestingly, we find that some parameters stabilize only temporally. Fig. 6 showcases two such parameters: they start to change again after a temporary 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 [35], and this may cause a "stabilized" parameter to drift away from its current value to the ultimate optimum. The permanent freezing solution, however, would prevent parameters from converging to the true optimum after they are frozen, which leads to accuracy loss eventually. It is inherently difficult to distinguish the temporally stabilized parameters by devising a better stability metric based on local observations, since their behavior before drifting is virtually identical to that of those who already converge. Hence, as another principle (**Principle-2**), *any effective solution must handle the possibility of temporary stabilization*.

C. Adaptive Parameter Freezing

To summarize, we learned two lessons from the previous explorations. First, to ensure model consistency, we have to freeze the non-synchronized stabilized parameters. Second, to ensure that the frozen parameters can converge to the true optima, we shall allow them to resume being updated (i.e., *unfreeze* them) at some point. Therefore, instead of freezing stabilized parameters 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 after the freezing period expires, that parameter should be updated as normal until it stabilizes again.

Control Mechanism of Freezing Period. Yet a remaining question is, given that there is no complete knowledge of the model convergence process a prior, when to unfreeze a frozen parameter? Or how to set the *freezing period* once a parameter becomes stable? Since each parameter has 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; 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 prior knowledge of the model being 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 being 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 [11], [8]: 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 [4], [5].

Similarly, under APF, the associated freezing period of each stabilized parameter starts with a small value (e.g., one round as in our evaluation). When the freezing period expires, the parameter re-joins 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 reacting to the individual 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.

D. Aggressive APF: Extension for Over-parameterized Models

Note that our previous analysis in Sec. III is based on the assumption of convex model (i.e., the loss function $F(\omega)$ be convex). Analyzing neural networks with convex assumptions is a common practice for the deep learning community, because non-convex analysis is very complex and many conclusions from conventional convex analysis can approximately hold for DNN models. However, we have to admit that DNN models are mostly non-convex and may distinct from convex models in certain aspects. In practice we have observed that, for large DNN models like ResNet, some parameters may not converge (i.e., stabilize) due to irregular landscapes like *flat minima* [22] or *saddle points* [26]: they keep moving in a certain direction even after the model converges. This phenomena is indeed quite common for large models which are usually *over-parameterized* [30], [40]. For those models, the fraction of stable parameters that are frozen under APF may be quite limited (as shown later in Fig. 7b of Sec. VI), although it is in fact possible to aggressively freeze many unstable parameters with the training accuracy preserved.

To exploit such unstable but accuracy-immaterial parameters, we introduce an extended version of APF: *Aggressive APF*. That is, in addition to the standard APF, we randomly associate some unstable parameters with a non-zero freezing period and put them into frozen status. Essentially, such an aggressive freezing method is similar with the famous Dropout technique [6], [20], [35], which, by randomly disabling some neural connections in each iteration, has been shown to saliently improve the model

generalization capability. Thus, since our previous adaptive freezing method can well adapt to the prematurely-frozen parameters, for over-parameterized models, *Aggressive APF* may attain a much better communication compression level without additional accuracy loss. We will confirm this in later evaluations.

V. Implementation

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 Alg. 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.

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.

A key implementation challenge is that, PyTorch operates parameters at the granularity of a full tensor, with no APIs supporting parameter freezing at the *scalar* granularity. This also holds in other machine learning frameworks like TensorFlow [3] and MXNet [13]. 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 `APF_Manager`). Meanwhile, all APF operations are implemented with the built-in tensor-based APIs of PyTorch for fast processing.

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 Alg. 1 is linear to the model size, i.e., $O(|\omega|)$. This is acceptable because when training neural network models, the memory is mostly consumed by input data and feature maps, which are usually *orders-of-magnitude* larger than the model itself [32]. Meanwhile, the additional latency of APF is also negligible, because the tensor traversal operations with PyTorch APIs are much faster than the convolution or deconvolution operations in forward and backward propagations. We will evaluate APF overheads later in Sec. VI-D.

Algorithm 1 Workflow with Adaptive Parameter Freezing

Require: F_s, F_c, T_s ▷ F_s: synchronization frequency; F_c: stability check frequency; T_s: threshold on P_k below which to judge a parameter as being stable

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

1: **procedure** ClientIterate(k)
2: $\omega_k^i \leftarrow \omega_{k-1}^i + u_k^i$ ▷ conduct regular local update
3: $\omega_k^i \leftarrow$`APF_Manager.Sync(`ω_k^i`)` ▷ pass model to `APF_Manager`

Central Server:

1: **procedure** Aggregate($\breve{\omega}_k^1$, $\breve{\omega}_k^2$, ..., $\breve{\omega}_k^N$)
2: **return** $\frac{1}{N}\sum_{i=1}^{N} \breve{\omega}_k^i$ ▷ aggregate all the non-frozen parameters

APF Manager:

1: **procedure** Sync(ω_k^i)
2: $\bar{\omega}_k^i \leftarrow M_{\text{is\_frozen}} \ ? \ \omega_{k-1}^i : \omega_k^i$ ▷ emulate freezing by parameter rollback; $M_{\text{is\_frozen}}$: freezing mask; $\bar{\omega}_k^i$: parameters with selectively-frozen effect
3: **if** $k \bmod F_s = 0$ **then**
4: $\breve{\omega}_k^i \leftarrow \bar{\omega}_k^i$.masked_select($!M_{\text{is\_frozen}}$)
▷ $\breve{\omega}_k^i$: a compact tensor composed of only the unstable parameters
5: $\breve{\omega}_k \leftarrow$`Central_Server.aggregate(`$\breve{\omega}_k^i$`)`
▷ synchronize the tensor containing unstable parameters
6: $\bar{\omega}_k^i \leftarrow \bar{\omega}_k^i$.masked_fill($!M_{\text{is\_frozen}}, \breve{\omega}_k$)
▷ 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($\bar{\omega}_k^i$,$M_{\text{is\_frozen}}$) ▷update $M_{\text{is\_frozen}}$
9: **return** $\bar{\omega}_k^i$

10: **function** StabilityCheck($\bar{\omega}_k^i$,$M_{\text{is\_frozen}}$)
▷ Operations below are *tensor-based*, supporting `where` selection semantics
11: update E, E^{abs}, P_k with $\bar{\omega}_k^i$ where $M_{\text{is\_frozen}}$ is False
▷ update *effective perturbation* based on Eq. 3
12: $L_{\text{freezing}} \leftarrow L_{\text{freezing}} + F_c$ where $P_k \leq T_s$ and $M_{\text{is\_frozen}}$ is False
13: $L_{\text{freezing}} \leftarrow L_{\text{freezing}}/2$ where $P_k > T_s$ and $M_{\text{is\_frozen}}$ is False
▷ L_{freezing}: freezing period lengths, updated in TCP-style
14: $I_{\text{unfreeze}} \leftarrow k + L_{\text{freezing}}$ where $M_{\text{is\_frozen}}$ is False
▷ I_{unfreeze}: iteration ids representing freezing deadlines of all the parameters
15: $M_{\text{is\_frozen}} \leftarrow (k < I_{\text{unfreeze}})$ ▷ update freezing mask for all parameters
16: **return** $M_{\text{is\_frozen}}$

VI. Evaluation

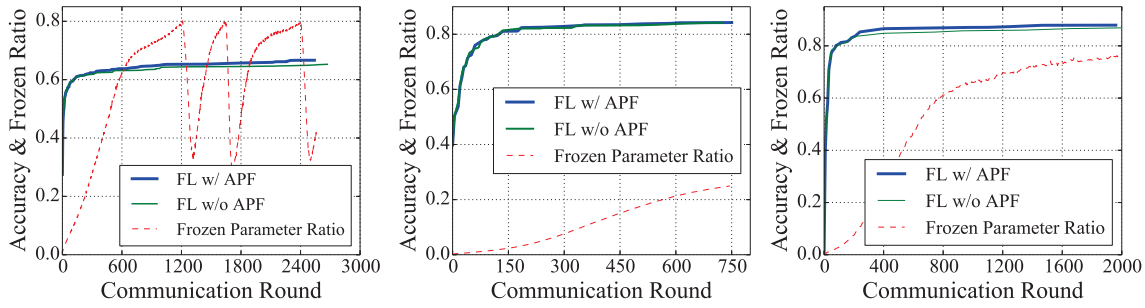
In this section, we systematically evaluate the performance of our APF algorithm. We start with end-to-end comparisons between APF and the standard FL scheme, and then resort to micro-benchmark evaluations to justify the stability and superiority of APF algorithm in various scenarios. Finally we examine the extra overheads incurred by APF operations.

A. Experimental Setup

Hardware Setup. We create a FL architecture[2] with 50 EC2 `m5.xlarge` instances as edge clients, each with 2 vCPU cores and 8GB memory (similar with that of 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.

Model and Dataset Setup. The datasets in our experiments are CIFAR-10 dataset and KeyWord Spotting dataset (KWS)—a subset of the Speech Commands dataset [38] with 10 keywords.

[2]In real-world FL scenarios, due to poor connection and client instability, some clients may dynamically leave or join the FL process [29]. Yet this is only an engineering concern and does not affect the effectiveness of our APF algorithm, because with admission control [9], 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.



(a) LeNet-5 (b) ResNet-18 (c) LSTM

Fig. 7: Test accuracy curves when training different models with and without APF.

Model	LeNet-5	ResNet-18	KWS-LSTM
Transmission-Volume w/ APF	239 MB	2.62 (1.44) GB	194 MB
Transmission-Volume w/o APF	651 MB	3.12 GB	428 MB
APF Improvement	63.3%	16.0% (53.8%)	54.7%

TABLE I: Cumulative transmission volume (Values in parentheses are from Aggressive APF).

Model	LeNet-5	ResNet-18	KWS-LSTM
Per-round Time w/APF	0.74 s	139 (95) s	1.8 s
Per-round Time w/o APF	1.02 s	158 s	2.2 s
Improvement	27.5%	12.1% (39.8%)	18.2%

TABLE II: Average per-round time (Values in parentheses are from Aggressive APF).

These datasets are randomly partitioned to the 50 clients. 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, hereafter referred as KWS-LSTM), all with a batch size of 100. Meanwhile, for diversity we use the Adam [16] optimizer for LeNet-5, and SGD optimizer for ResNet-18 and KWS-LSTM; their learning rates are set to 0.001, 0.1, and 0.01, respectively (default for each model-optimizer combination), with a weight decay of 0.01. Moreover, the synchronization and stability check frequencies—F_s and F_c in Alg. 1—are set as 10 and 50. Regarding the stability check, 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%. Note that large models like VGG are not employed considering the limited resources on FL clients, as in existing works [27], [28].

B. Overall Benefits

Convergence Validity. Fig. 7 shows the test accuracy curves for training the three models with and without APF. 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. In particular, we notice that

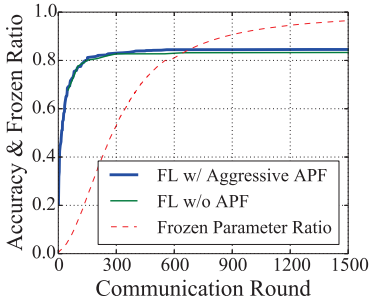


Fig. 8: *Aggressive APF* attains a much larger communication improvement for ResNet-18 without accuracy loss.

when training LeNet-5 and KWS-LSTM, APF actually achieves a better accuracy. This is because like dropout [35], intermittent parameter freezing under APF can break up parameter co-adaptations and avoid overfitting, which improves the model's generalization capability.

Efficiency. We next turn to the efficiency gain of APF. Table I summarizes the cumulative transmission volume of each client up to convergence during training. APF can greatly reduce the transmission volume: for LeNet-5 for example, it provides a saving of 63.3%. Table II 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%. This gain, amortized by the computation time, is less salient than the transmission volume savings. Yet, 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 further improved.

Effectiveness of Aggressive APF. Note that for ResNet-18 in Fig. 7b, the performance benefit of APF is quite limited, because ResNet-18 is over-parameterized and many parameters in convergence state may not be stable (Sec. IV-D). Recall that we have introduced *Aggressive APF* to address this problem. To evaluate its performance, we set the possibility of freezing an unstable parameter as $\texttt{min}(\frac{K}{2000}, 0.5)$, where K is the round id. As shown in Fig. 8, under Aggressive APF, the overall transmission reduction increases to 53.8% (in contrast to 16.0%

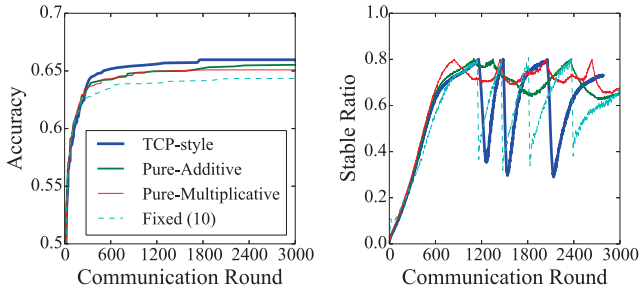


Fig. 9: A comparison of the TCP-style scheme in APF against other schemes on freezing period control. *Pure-Additively* means to additively increase or decrease the freezing period by 1; *Pure-Multiplicatively* means to multiplicatively increase or decrease the freezing period by $2\times$; *Fixed (10)* means to freeze each stabilized parameters by 10 (in stability checks, i.e., $10 \times F_c$ iterations).

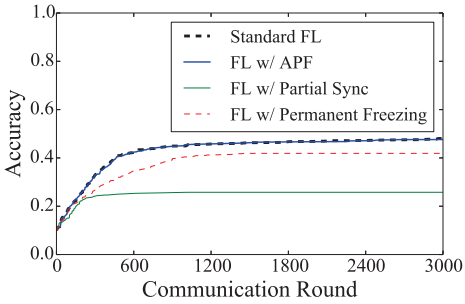


Fig. 10: APF performance for LeNet-5 with non-IID data.

in Fig.7b) with the same accuracy performance. In Table I and Table II, as noted by the values in parentheses, Aggressive APF largely outperforms APF in both transmission volume and per-round time for ResNet-18.

C. Micro-Benchmark Evaluations

Necessity of the TCP-style control mechanism in APF. 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 evaluate that, we replace it with three different control mechanisms: *pure-additive*—increase/decrease the freezing period always additively; *pure-multiplicative*—increase/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. 9 shows the validating accuracy and instantaneous stable ratio under each scheme. While different schemes achieve similar communication reduction, our TCP-style design, by freezing parameters tentatively and unfreezing them agilely once parameter shifting occurs, can yield the best learning accuracy.

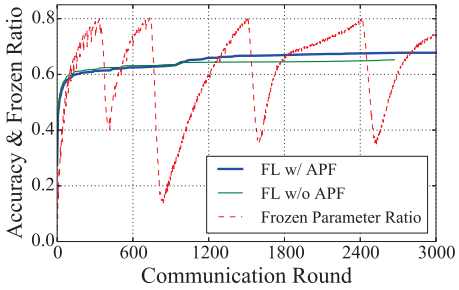


Fig. 11: When the initial stability threshold is set loosely, LeNet-5 can still converge to the optimal accuracy after several *tighten-up* actions.

Performance with Non-IID Data. So far our evaluation is based on IID data. Recall that in Sec. IV-B, partial synchronization method performs poorly on non-IID data due to parameter divergence. Then, can APF perform well on non-IID data? To evaluate that, we train LeNet-5 with 5 workers, and each worker is configured with only 2 distinct classes of CIFAR-10. As shown in Fig. 10, APF achieves the same accuracy as regular FL after convergence. Note that with non-IID data the model accuracy is much worse, which is consistent with observations from prior work [43] and beyond the scope of this work.

Hyper-parameter Robustness. As discussed in Sec. IV-C, to make APF robust to stability threshold, we have introduced a mechanism that tightens the stability threshold when most (80% in our evaluation) parameters stabilize. 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. Fig. 11 depicts the corresponding accuracy curve, together with the frozen parameter ratio in training. Comparing Fig. 11 to Fig. 7a we can easily find that, with a loose initial stability threshold, number of frozen parameters increase more rapidly, but the accuracy curve is slightly below that without APF. However, after several tightening actions, APF gradually catches up with—and finally outperforms—standard FL in accuracy. Therefore, even if the initial stability threshold is inappropriate, our APF algorithm can still yield good performance.

D. Overheads

Computation and Memory Overheads. The last question we seek to answer here is, what are the memory and computation overheads of APF? We measure the extra computation time (in average per-round training time) and physical memory consumption incurred by APF. Table III lists the overheads for different models. As implied in Table III, for small models like LeNet and KWS-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 data and feature map, APF's memory overhead becomes larger (8.51%). Overall, considering the salient reduction

Model	LeNet-5	ResNet-18	KWS-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%

TABLE III: Computation (average per-round training time) and memory overheads of APF.

of network transmission volume and convergence time, the overhead of APF is indeed acceptable.

VII. Related Work

Mitigating communication bottlenecks for distributed machine learning is recently a very hot research topic. In the *scheduling* perspective, the communication efficiency can be improved by relaxing the synchronization collisions [21], [14], [12], by pipelining the layer-wise parameter transmission [42], [34], or by controlling the transmission order of different parameter groups [19], [25], [31]. In the *communication compression* perspective which is more closely related to our work, existing works can be broadly classified into two categories: quantization and sparsification.

Quantization reduces bandwidth consumption by transmitting low-precision updates. Seide et al. [33] proposed to aggressively quantize the gradients into only 1 bit, and achieved a $10\times$ communication speedup. A later work, QSGD [7], sought to balance the trade-off between accuracy and gradient compressing ratio. Further, Konečnỳ et al. [27] adopted a probabilistic quantization method, which made the quantized value an unbiased estimator of the initial one. These quantization approaches are orthogonal to and can be integrated with APF.

Sparsification aims to send only the important gradients. Storm [36] and Gaia [23] proposed to only send gradients larger than a threshold. Dryden et al. [17] designed an approach that given a preset compression ratio, selected a fraction of positive and negative gradient updates separately. Nonetheless, in these approaches it is usually difficult to determine the appropriate threshold or ratio beforehand, and the accuracy is often suboptimal. Meanwhile as elaborated in Sec. II, sparsification algorithms, assuming that all the model parameters should always be synchronized, seek to filter small gradients based on *client-local* and *instantaneous* knowledge, yet APF fundamentally eliminates parameter synchronization based on a *global* and *historical* point of view. In this sense, sparsification is also orthogonal to our solution.

Moreover, parameter freezing has already appeared in recent research literature. Brock et al. [10] proposed the FreezeOut mechanism that gradually freezes the first few layers of a deep neural network—which were observed to converge faster—to avoid the computation cost of calculating their gradients. However, its operation granularity is an entire layer, which is too coarse given the analysis in Sec. III. Worse, because there is no parameter unfreezing mechanism, FreezeOut degrades the accuracy performance despite the computation speedup [10].

VIII. Conclusion

In this work, to reduce the communication overhead in FL, we presented Adaptive Parameter Freezing (APF) which eliminates the synchronization of stabilized parameters. APF identifies the stabilized parameters based on their effective perturbation and tentatively freezes the stable parameters for an interval of time, which is adjusted in an additively-increase and multiplicatively-decrease manner. We implemented APF based on PyTorch, and testbed experiments confirmed that APF can largely improve the communication efficiency of FL, with identical or even better accuracy performance.

Appendix

A. Proof of Theorem 1

Assumption 1. (Strong Convexity.) *The global loss function $F(\omega)$ is $\mu-$strongly convex, i.e.,*

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

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

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

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

Theorem 1. *Under Assumptions 1 and 2, choose $\eta \in (0, \frac{1}{2\mu}]$, then we have*

$$\mathbb{E}(||\omega_k - \omega^\star||^2) \leq (1-2\mu\eta)^k||\omega_0-\omega^\star||^2 + \frac{\eta\sigma^2}{2\mu}.$$

Proof.

$$\begin{aligned}||\omega_{k+1}-\omega^\star||^2 &= ||\omega_k - \omega^\star + \omega_{k+1} - \omega_k||^2 \\ &= ||\omega_k - \omega^\star - \eta g_{\xi_k}(\omega_k)||^2 \\ &= ||\omega_k-\omega^\star||^2 - 2\eta\langle\omega_k-\omega^\star, g_{\xi_k}(\omega_k)\rangle + \eta^2||g_{\xi_k}(\omega_k)||^2\end{aligned}$$

Taking expectation conditioned on ω_k we can obtain:

$$\begin{aligned}\mathbb{E}_{\omega_k}(||\omega_{k+1}-\omega^\star||^2) = ||\omega_k-\omega^\star||^2 - 2\eta\langle\omega_k-\omega^\star, \nabla F(\omega_k)\rangle \\ + \eta^2\mathbb{E}_{\omega_k}(||g_{\xi_k}(\omega_k)||^2)\end{aligned}$$

Given that $F(\omega)$ is $\mu-$strongly convex, we have

$$\begin{aligned}\langle\nabla F(\omega_k), \omega_k-\omega^\star\rangle &= \langle\nabla F(\omega_k) - \nabla F(\omega^\star), \omega_k-\omega^\star\rangle \\ &\geq \mu||\omega_k-\omega^\star||^2\end{aligned}$$

Applying total expectation, we can obtain

$$\begin{aligned}\mathbb{E}(||\omega_{k+1}-\omega^\star||^2) &= \mathbb{E}||\omega_k-\omega^\star||^2 - 2\eta\langle\mathbb{E}(\omega_k-\omega^\star), \nabla F(\omega_k)\rangle \\ &\quad + \eta^2\mathbb{E}(||g_{\xi_k}(\omega_k)||^2) \\ &\leq (1-2\mu\eta)\mathbb{E}||\omega_k-\omega^\star||^2 + \eta^2\sigma^2\end{aligned}$$

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

$$\begin{aligned}\mathbb{E}(||\omega_k-\omega^\star||^2) &\leq (1-2\mu\eta)^k||\omega_0-\omega^\star||^2 + \sum_{j=0}^{k-1}(1-2\mu\eta)^j\eta^2\sigma^2 \\ &\leq (1-2\mu\eta)^k||\omega_0-\omega^\star||^2 + \frac{\eta\sigma^2}{2\mu}\end{aligned}$$

This completes our proof. □

REFERENCES

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

[2] PyTorch. https://pytorch.org/.

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A system for large-scale machine learning. In *USENIX OSDI*, 2016.

[4] ACM. Turning Award Laureates: ROBERT ("BOB") ELLIOT KAHN. https://amturing.acm.org/award_winners/kahn_4598637.cfm, 2005.

[5] ACM. Turning Award Laureates: VINTON ("VINT") GRAY CERF. https://amturing.acm.org/award_winners/cerf_1083211.cfm, 2005.

[6] Geoffrey E. Hinton Alex Krizhevsky. ImageNet Classification with Deep Convolutional Neural Networks. In *NeurIPS*, 2012.

[7] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. QSGD: Communication-efficient SGD via gradient quantization and encoding. In *Advances in Neural Information Processing Systems*, pages 1709–1720, 2017.

[8] Mark Allman, Vern Paxson, and Ethan Blanton. TCP congestion control. Technical report, 2009.

[9] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloe Kiddon, Jakub Konecn Konečny, Stefano Mazzocchi, H Brendan Mcmahan, Timon Van Overveldt, David Petrou, Daniel Ramage, and Jason Roselander. Towards Federated Learning at Scale: System Design. In *SysML*, 2019.

[10] Andrew Brock, Theodore Lim, James M Ritchie, and Nick Weston. Freezeout: Accelerate training by progressively freezing layers. *arXiv preprint arXiv:1706.04983*, 2017.

[11] Vinton G. Cerf and Robert E. Kahn. A Protocol for Packet Network Intercommunication. *IEEE Transactions on Communications*, pages 637–648, May 1974.

[12] Chen Chen, Wei Wang, and Bo Li. Round-Robin Synchronization: Mitigating Communication Bottlenecks in Parameter Servers. In *IEEE INFOCOM*, 2019.

[13] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv:1512.01274*, 2015.

[14] Henggang Cui, Alexey Tumanov, Jinliang Wei, Lianghong Xu, Wei Dai, Jesse Haber-Kucharsky, Qirong Ho, Gregory R Ganger, Phillip B Gibbons, Garth A Gibson, et al. Exploiting iterative-ness for parallel ml computations. In *ACM SoCC*, 2014.

[15] Misha Denil, Babak Shakibi, Laurent Dinh, Nando De Freitas, et al. Predicting parameters in deep learning. In *Advances in neural information processing systems*, pages 2148–2156, 2013.

[16] Jimmy Ba Diederik P. Kingma. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2017.

[17] Nikoli Dryden, Tim Moon, Sam Ade Jacobs, and Brian Van Essen. Communication quantization for data-parallel training of deep neural networks. In *2016 2nd Workshop on Machine Learning in HPC Environments (MLHPC)*, pages 1–8. IEEE, 2016.

[18] Zidong Du, Krishna Palem, Avinash Lingamneni, Olivier Temam, Yunji Chen, and Chengyong Wu. Leveraging the error resilience of machine-learning applications for designing highly energy efficient accelerators. In *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 201–206. IEEE, 2014.

[19] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H Campbell. TICTAC: Accelerating Distributed Deep Learning with Communication Scheduling. In *MLSys*, 2019.

[20] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. 2012.

[21] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *NIPS*, 2013.

[22] Sepp Hochreiter and Jürgen Schmidhuber. Flat minima. *Neural Computation*, 9(1):1–42, 1997.

[23] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R Ganger, Phillip B Gibbons, and Onur Mutlu. Gaia: Geo-distributed machine learning approaching LAN speeds. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 629–647, 2017.

[24] Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q Weinberger. Deep networks with stochastic depth. In *European conference on computer vision*, pages 646–661. Springer, 2016.

[25] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. Priority-based Parameter Propagation for Distributed DNN Training. In *MLSys*, 2019.

[26] Kenji Kawaguchi. Deep learning without poor local minima. In *Advances in neural information processing systems*, pages 586–594, 2016.

[27] Jakub Konečnỳ, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492*, 2016.

[28] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[29] H Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, et al. Communication-efficient learning of deep networks from decentralized data. *arXiv preprint arXiv:1602.05629*, 2016.

[30] Behnam Neyshabur, Zhiyuan Li, Srinadh Bhojanapalli, Yann LeCun, and Nathan Srebro. The role of over-parametrization in generalization of neural networks. In *ICLR*, 2018.

[31] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019.

[32] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *IEEE/ACM Micro*, 2016.

[33] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[34] Shaohuai Shi and Xiaowen Chu. MG-WFBP: Efficient data communication for distributed synchronous sgd algorithms. In *IEEE INFOCOM*, 2019.

[35] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[36] Nikko Strom. Scalable distributed dnn training using commodity gpu cloud computing. In *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

[37] L. WANG, W. WANG, and B. LI. Cmfl: Mitigating communication overhead for federated learning. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2019.

[38] Pete Warden. Speech commands: A dataset for limited-vocabulary speech recognition. *arXiv preprint arXiv:1804.03209*, 2018.

[39] Timothy Yang, Galen Andrew, Hubert Eichner, Haicheng Sun, Wei Li, Nicholas Kong, Daniel Ramage, and Françoise Beaufays. Applied Federated Learning: Improving Google Keyboard Query Suggestions. 2018.

[40] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. In *NIN*, 2016.

[41] Matthew D Zeiler. ADADELTA: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[42] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P. Xing. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *USENIX ATC*, 2017.

[43] Yue Zhao, Meng Li, Liangzhen Lai, Naveen Suda, Damon Civin, and Vikas Chandra. Federated learning with non-iid data. *arXiv preprint arXiv:1806.00582*, 2018.