Zeno++: Robust Fully Asynchronous SGD

Cong Xie [1] Oluwasanmi Koyejo [1] Indranil Gupta [1]

Abstract

We propose Zeno++, a new robust asynchronous Stochastic Gradient Descent (SGD) procedure, intended to tolerate Byzantine failures of workers. In contrast to previous work, Zeno++ removes several unrealistic restrictions on worker-server communication, now allowing for fully asynchronous updates from anonymous workers, for arbitrarily stale worker updates, and for the possibility of an unbounded number of Byzantine workers. The key idea is to estimate the descent of the loss value after the candidate gradient is applied, where large descent values indicate that the update results in optimization progress. We prove the convergence of Zeno++ for non-convex problems under Byzantine failures. Experimental results show that Zeno++ outperforms existing Byzantine-tolerant asynchronous SGD algorithms.

1. Introduction

Synchronous training and asynchronous training are the two most common paradigms of distributed machine learning. On the one hand, synchronous training requires the global updates at the server to be blocked until all the workers respond (after each period). In contrast, for asynchronous training, the server can update the global model immediately after each worker's response. Theoretical and experimental analysis (Dutta et al., 2018) suggests that synchronous training is more stable and has lower noise, but is slower due to the global barrier across all the workers (after each period). Since asynchronous training is comparatively faster especially for heterogeneous systems with stragglers, in this paper, we focus on asynchronous training. Doing so requires us to tackle challenges including instability and noisiness that arise from asynchony.

[1]Department of Computer Science, University of Illinois, Urbana-Champaign, USA. Correspondence to: Cong Xie <cx2@illinois.edu>.

Proceedings of the 37th International Conference on Machine Learning, Online, PMLR 119, 2020. Copyright 2020 by the author(s).

Concretely, we study the security of distributed asynchronous Stochastic Gradient Descent (SGD) in a centralized worker-server architecture, also known as the Parameter Server (or PS) architecture. In the PS architecture, there are server nodes and worker nodes. Each worker pulls the global model from the servers, estimates the gradients using the local portion of the training data, then sends the gradient estimates to the servers. The servers update the model asynchronously, as soon as a new gradient is received from any worker.

The security of machine learning has gained increasing attention in recent years. In particular, tolerance to Byzantine failures (Blanchard et al., 2017; Chen et al., 2017; Yin et al., 2018; Feng et al., 2014; Su & Vaidya, 2016; 2018; Xie et al., 2019b; Alistarh et al., 2018; Cao & Lai, 2018; Xie et al., 2019c) has become an important topic in the distributed machine learning literature. Byzantine nodes can behave arbitrarily, capturing behavior ranging from crash failures to malicious and arbitrary or compromised actions. In brief, the goal of Byzantine workers is to prevent convergence of the model training. By construction, Byzantine failures (Lamport et al., 1982) assume the worst case, i.e., the Byzantine workers can behave arbitrarily. Such failures may be caused by a variety of reasons including but not limited to: hardware/software bugs, vulnerable communication channels, poisoned datasets, or malicious attackers. To make things worse, groups of Byzantine workers may collude, resulting in more harmful attacks.

Byzantine failures have been well-studied for classical distributed systems, especially for consensus (Lamport et al., 1982). These results include well-known bounds on how many Byzantine workers can be tolerated. Yet, in distributed machine learning, Byzantine failures have a different impact than in classical consensus (Damaskinos et al., 2018). Unlike previous work, we consider Byzantine tolerance with minimal assumptions—an unbounded number of Byzantine workers, and full asynchrony with unbounded delay.

Byzantine tolerance of asynchronous SGD is challenging due to several reasons:

- **Full asynchrony.** Lack of synchrony introduces additional noise into the stochastic gradients. This makes it more difficult to distinguish gradients sent by Byzantine workers from gradients sent by benign ones, especially

as Byzantine behavior may exacerbate staleness.

- **Unpredictable successive updates.** The lack of synchronous scheduling means "fast" workers may send updates, to the server, more frequently than "slower" workers. Byzantine workers may exploit this by suffocating the server with their wrong gradients.

- **Unbounded number of Byzantine workers.** While traditional Byzantine consensus assumes an upper bound on the number of malicious workers (typically one-third or half (Lamport et al., 1982)), for fully asynchronous training, the assumption of a bounded number of Byzantine workers is impractical. In Byzantine-tolerant synchronous training (Blanchard et al., 2017; Chen et al., 2017; Yin et al., 2018; Xie et al., 2019b;a; 2018), the servers can compare the candidate gradients with each other, and either utilize a majority to filter out the harmful gradients, or use robust aggregation to bound the error. However, such strategies are infeasible in asynchronous training, since there may be nothing to compare against, or aggregate with. Aggregating the successive gradients is also meaningless since the successive gradients could all be pushed by the same Byzantine worker. Furthermore, although most of the previous work (Blanchard et al., 2017; Chen et al., 2017; Yin et al., 2018; Feng et al., 2014; Su & Vaidya, 2016; 2018; Alistarh et al., 2018; Cao & Lai, 2018) assumes a majority of honest workers, this requirement is not guaranteed to be satisfied in practice.

We propose Zeno++, a new algorithm that validates the gradients with low computation overhead on the server via lazy updates. Our key idea is to estimate the descent of the loss value after the candidate gradient is applied to the model parameters–for the latter we leverage the Byzantine-tolerant synchronous SGD algorithm called Zeno (Xie et al., 2019b). Intuitively, if the loss value decreases, the candidate gradient is likely to result in optimization progress. We also propose a mechanism of lazy updates to reduce the computation overhead.

To the best of our knowledge, this paper is the first to theoretically *and* empirically study Byzantine-tolerant fully asynchronous SGD with anonymous workers, and potentially an unbounded number of Byzantine workers. In summary, our contributions are:

- We propose Zeno++, a new approach for Byzantine-tolerant fully asynchronous SGD with anonymous workers, and low overhead of the additional computation for the validation on the servers.

- We show that Zeno++ tolerates Byzantine workers without any limit on either the staleness or the number of Byzantine workers.

- We prove the convergence of Zeno++ for non-convex problems.

- Experimental results validate that 1) existing algorithms may fail in practical scenarios, and 2) Zeno++ gracefully handles such cases.

2. Related Work

Most of the existing Byzantine-tolerant SGD algorithms focus on synchronous training. Chen et al. (2017); Su & Vaidya (2016; 2018); Yin et al. (2018); Xie et al. (2018) use robust statistics (Huber, 2004) including the geometric median, coordinate-wise median, and trimmed mean as Byzantine-tolerant aggregation rules. Blanchard et al. (2017); Mhamdi et al. (2018) propose Krum and its variants, which select the candidates with minimal local sum of Euclidean distances. Alistarh et al. (2018) utilize historical information to identify harmful gradients. Chen et al. (2018) use coding theory and majority voting to recover correct gradients. Most of these synchronous algorithms assume that most of the workers are non-Byzantine. However, in practice, there are no guarantees that the number of Byzantine workers can be controlled. Xie et al. (2019b); Cao & Lai (2018) propose synchronous SGD algorithms for an unbounded number of Byzantine workers.

Recent years have witnessed an increasing number of large-scale machine learning algorithms, including asynchronous SGD (Zinkevich et al., 2009; Lian et al., 2018; Zheng et al., 2017; Zhou et al., 2018). Damaskinos et al. (2018) proposed Kardam, which to our knowledge is the only prior work to address Byzantine-tolerant asynchronous training. Kardam utilizes the Lipschitzness of the gradients to filter out outliers. However, Kardam assumes a threat model much weaker than ours. The major differences in the threat model are listed as follows:

- **Verification of worker identity.** Unlike Kardam, we do not require verifying the identities of the workers when the server receives gradients. Kardam uses the so-called *empirical Lipschitz coefficient*, to test the benignity of the gradient sent by a specific worker. Such a mechanism keeps the record of the *empirical Lipschitz coefficient* of each worker. Thus, whenever a gradient is received, the Kardam server must be able to identify the identity/index of the worker. However, since Byzantine workers can behave arbitrarily, they can fake their identities/indices when sending gradients to the servers. Thus, Kardam assumes a threat model much weaker than the traditional Byzantine failure/threat model. Note that for synchronous training, the server can partially counter the index spoofing attack by simply filtering out all the gradients with duplicated indices. However, such an approach is infeasible for asynchronous training.

- **Bounded staleness of workers/limit of successive gradients.** Unlike Kardam, we do not require bounded staleness of the workers. Kardam requires that the number of gradients successively received from a single worker is bounded above. To be more specific, on the server, any sequence of successively received gradients of length $2q+1$ must contain at least $q+1$ gradients from honest workers. However, in real-world asynchronous training, such an assumption is very difficult to satisfy.

- **A majority of honest workers.** Unlike Kardam, we do not require a majority of honest workers. Kardam requires that the number of Byzantine workers is less than one-third of the total number of workers – much stronger restriction than the standard setting that allows for the number of Byzantine workers to be up to 50% of the total number of workers. Zeno++ further extends this guarantee to allow for not only 50%, but also a majority of Byzantine workers.

3. Model

We consider the following optimization problem: $\min_{x \in \mathbb{R}^d} F(x)$, where $F(x) = \frac{1}{m}\sum_{i \in [m]} \mathbb{E}_{z_i \sim \mathcal{D}_i} f(x; z_i)$, for $\forall i \in [m]$, z_i is sampled from the local data $\mathcal{D}_i$ on the ith device.

We solve this problem in a distributed manner with m workers. Each worker trains the model on local data. In each iteration, the ith worker will sample n independent data points from the dataset $\mathcal{D}_i$, and compute the gradient of the local empirical loss $F_i(x) = \frac{1}{n}\sum_{j=1}^n f(x; z_{i,j}), \forall i \in [m]$, where $z_{i,j} \sim \mathcal{D}_i$ is the jth sampled data on the ith worker. When there are no Byzantine failures, the servers update the model whenever a new gradient is received:

$$x_{t+1} = x_t - \gamma_t g_\tau,$$
$$g_\tau = \frac{1}{n}\sum_{j \in [n]} \nabla f(x_\tau; z_{i,j}), \tau \le t, i \in [m].$$

When there are Byzantine failures, g_τ can be replaced by arbitrary value (Damaskinos et al., 2018). Formally, we define the threat model as follows.

Definition 1. *(Threat Model). When the server receives a gradient estimator $\tilde{g}_\tau$, it is either correct or Byzantine. If sent by a Byzantine worker, $\tilde{g}_\tau$ is assigned arbitrary value. If sent by an honest worker, the correct gradient is $\frac{1}{n}\sum_{j=1}^n \nabla f(x_\tau; z_{i,j}), \tau \le t, i \in [m]$. Thus, we have*

$$\tilde{g}_\tau = \begin{cases} \textit{arbitrary value}, & \textit{if the worker is Byzantine}, \\ \frac{1}{n}\sum_{j=1}^n \nabla f(x_\tau; z_{i,j}), & \textit{otherwise.} \end{cases}$$

We assume that q out of m workers are Byzantine, where $q \le m$. Furthermore, the indices of Byzantine workers can change across different iterations.

Table 1. Notation

Notation	Description
m, $[m]$	Number of workers, set of integers $\{1, \ldots, m\}$
q	Number of Byzantine workers
$\mathcal{D}_i$	$\mathcal{D}_i$ is the training dataset on the ith worker
$\mathcal{S}$	The validation dataset on *Zeno++* server
n	Mini-batch size of workers
n_s	Mini-batch size of Zeno++ server
T, t	Number of global iterations, global iteration index
γ	Learning rate
ρ, ϵ	Hyperparameters of Zeno++
k	Maximum delay of g_r, i.e., "server delay"
k_w	Maximum delay of workers, i.e., "worker delay"
$\lVert \cdot \rVert$	All the norms in this paper are l_2-norms

4. Methodology

In this section, we introduce Zeno++, a Byzantine-tolerant asynchronous SGD algorithm based on inner-product validation. Zeno++ is a computationally efficient version of its prototype: Zeno+.

4.1. Zeno+

For completeness, we first introduce an auxiliary algorithm: Zeno+. Note that Zeno+ is hard to be applied in practice, due to its heavy computation overhead on the server for validation. We introduce Zeno+ only to help understand the concept of Zeno++.

Inspired by Zeno (Xie et al., 2019b), we compute a score for each candidate gradient estimator by using the stochastic zero-order oracle. However, in contrast to the existing synchronous SGD with majority-based aggregation methods, we need a hard threshold to decide whether a gradient is accepted, as sorting is not meaningful in asynchronous settings. This descent score is described next.

Definition 2. *(Stochastic Descent Score (Xie et al., 2019b)) Denote $f_s(x) = \frac{1}{n_s}\sum_{j=1}^{n_s} f(x; z_j)$, where z_j's are i.i.d. samples drawn from $\mathcal{S}$, where $\mathcal{S} \neq \mathcal{D}_i, \forall i \in [m]$, and n_s is the batch size of $f_s(\cdot)$. For any gradient estimator (correct or Byzantine) g, model parameter x, learning rate γ, and a constant weight $\rho > 0$, we define its stochastic descent score as follows:*

$$Score_{\gamma,\rho}(g, x) = f_s(x) - f_s(x - \gamma g) - \rho \|g\|^2.$$

Remark 1. *Note that we assume that the dataset $\mathcal{S}$ for computing $f_s(\cdot)$ is different from the training dataset, e.g., can be a separated validation dataset. In other words, $\mathcal{S} \neq \mathcal{D}_1 \neq \cdots \neq \mathcal{D}_m \neq \cup_{i=1}^m \mathcal{D}_i$.*

The score defined in Definition 2 is composed of two parts: the estimated descent of the loss function, and the magnitude of the update. The score increases when the estimated descent of the loss function, $f_s(x) - f_s(x - \gamma g)$, gets larger.

We penalize the score by $-\rho\|g\|^2$, so that the change of the model parameter will not be too large. A large descent suggests faster convergence. Observe that even when a gradient is Byzantine, a small magnitude indicates that it will be less harmful to the model.

Using the *stochastic descent score*, we can set a hard threshold parameterized by ϵ to filter out candidate gradients with relatively small scores. The detailed algorithm is outlined in Algorithm 1.

Algorithm 1 Zeno+

Server:
- $x_0 \leftarrow rand(), t \leftarrow 1$
- **repeat**
  - Randomly sample $z_j \sim \mathcal{S},\ \forall j \in [n_s]$ to compute f_s (Note: $\mathcal{S} \neq \mathcal{D}_1 \neq \cdots \neq \mathcal{D}_m$)
  - Receive $\tilde{g}$ from an arbitrary worker
  - Normalize $g = c\tilde{g}$ such that $\|g\|^2 = \|\nabla f_s(x_{t-1})\|^2$
  - **if** $Score_{\gamma,\rho}(g, x_{t-1}) \geq -\gamma\epsilon$ **then**
    - $x_t \leftarrow x_{t-1} - \gamma g, t \leftarrow t+1$
  - **end if**
- **until** Convergence

Worker $i = 1, \ldots, m$:
- **if** The worker is honest **then**
  - **repeat**
    - Pull x_τ from the server
    - Draw random samples $z_{i,j} \sim \mathcal{D}_i, \forall j \in [n]$, compute $\tilde{g} \leftarrow \frac{1}{n}\sum_{j\in[n]} \nabla f(x_\tau; z_{i,j})$
    - Push $\tilde{g}$ to the server
  - **until** Convergence
- **end if**

4.2. Zeno++

Calculating the *stochastic descent score* for every candidate gradient can be computationally expensive. To reduce the computation overhead, we approximate it by its first-order Taylor's expansion with stale validation gradients.

Definition 3. *(Approximated Stochastic Descent Score) Denote $f_s(x) = \frac{1}{n_s}\sum_{j=1}^{n_s} f(x; z_j)$, where z_j's are i.i.d. samples drawn from $\mathcal{S}$, where $\mathcal{S} \neq \mathcal{D}_i, \forall i \in [m]$, and n_s is the batch size of $f_s(\cdot)$. For any gradient estimator (correct or Byzantine) g, model parameter x, learning rate γ, and a constant weight $\rho > 0$, we approximate its stochastic descent score as follows:*

$$Score_{\gamma,\rho}(g, x) \approx \gamma \langle \nabla f_s(x), g \rangle - \rho\|g\|^2,$$

where x is a stale version of model on the server.

In brief, `Zeno++` is a computation-efficient version of `Zeno+` which reduces the computation overhead via two techniques: *approximated stochastic descent score*, and lazy updates of the validation gradient v. The detailed algorithm is shown in Algorithm 2. Compared to `Zeno`, we highlight several new techniques in `Zeno++` (Algorithm 2), specially designed for asynchronous training: 1) re-scaling the candidate gradient (Line (1)); 2) first-order Taylor's expansion (Line (2)); 3) hard threshold instead of comparison with the others (Line (2)); 4) lazy update for reducing the computation overhead (Line (3)).

Algorithm 2 Zeno++

Server:
- $x_0 \leftarrow rand(), t \leftarrow 1$
- **repeat**
  - **repeat**
    - Receive $\tilde{g}$ from an arbitrary worker
    - Read v with lock (v may be from an old version of x: $v = \nabla f_s(x_\tau), \tau \leq t-1$)
    - Normalize $g = c\tilde{g}$ such that $\|g\|^2 = \|v\|^2$ (1)
  - **until** $\gamma \langle v, g \rangle - \rho\|g\|^2 \geq -\gamma\epsilon$ (2)
  - $x_t \leftarrow x_{t-1} - \gamma g, \quad t \leftarrow t+1$
  - Lazy update of v: Run non-blocking $ZenoUpdater(x_t)$, if idle, or after every k iterations (3)
- **until** Convergence

ZenoUpdater(x) on server:
- Randomly sample $z_j \sim \mathcal{S},\ \forall j \in [n_s]$ to compute f_s (Note: $\mathcal{S} \neq \mathcal{D}_1 \neq \cdots \neq \mathcal{D}_m$)
- Write with lock: $v \leftarrow \nabla f_s(x) = \frac{1}{n_s}\sum_{j=1}^{n_s} \nabla f(x; z_j)$

Worker $i = 1, \ldots, m$:
- **if** The worker is honest **then**
  - **repeat**
    - Pull x_τ from the server
    - Draw random samples $z_{i,j} \sim \mathcal{D}_i, \forall j \in [n]$, compute $\tilde{g} \leftarrow \frac{1}{n}\sum_{j\in[n]} \nabla f(x_\tau; z_{i,j})$
    - Push $\tilde{g}$ to the server
  - **until** Convergence
- **end if**

Before moving forward, we wish to highlight several practical remarks for `Zeno++`:

- **Preparing the validation dataset for Zeno++:** The dataset $\mathcal{S}$ used for calculating v (the validation gradient of `Zeno++`) can be collected in many ways. As a machine-learning routine, it is common to separate the entire dataset into 3 parts: training data, validation data, and testing data. Such partition naturally provides the validation dataset for `Zeno++`. It can also be a separate validation dataset provided by a trusted third party. Another reasonable choice is that, a group of trusted workers can upload local data perturbed by additional noise (to

help protect the users' privacy). Typically, the validation dataset is small and different from the training dataset, thus can only be used to validate the gradients, and cannot be directly used for training, as shown in Section 6.

- **Scheduling** $ZenoUpdater(x)$**:** $ZenoUpdater(x)$ updates v in the background. It will only be triggered when the global model x_t is updated and the server is idle. Another scheduling strategy is to trigger $ZenoUpdater(x)$ after every k iterations. Thus, k is the upper bound of the delay of v. A reasonable choice is $k = m$, so that ideally v is updated after all the m workers respond.

- **Computational efficiency:** We can reduce the computation overhead of the `Zeno++` server by decreasing the mini-batch size n_s, or the frequency of the activation of $ZenoUpdater(x)$. However, doing so will potentially incur larger noise for v, which makes a trade-off.

5. Theoretical Guarantees

In this section, we prove the convergence of `Zeno++` (Algorithm 2) under Byzantine failures. We start with definitions used in the convergence analysis.

Definition 4. *(Smoothness) Differentiable $f(x)$ satisfies L-smoothness if there exists $L > 0$ such that $\forall x, y$, $f(y) - f(x) \leq \langle \nabla f(x; z), y - x\rangle + \frac{L}{2}\|y - x\|^2$.*

Definition 5. *(Polyak-Łojasiewicz (PL) inequality) Differentiable $f(x)$ satisfies the PL inequality (Polyak, 1963) if there exists $\mu > 0$, such that $\forall x$: $f(x) - f(x_*) \leq \frac{1}{2\mu}\|\nabla f(x)\|^2$.*

5.1. Convergence Guarantees

We prove the convergence of Algorithm 2 for non-convex problems with the following assumption.

Assumption 1. *(Bounded server delay) For `Zeno++`, we assume that the delay of the validation gradient v is upper-bounded. Without loss of generality, suppose the current model is x_t, and $v = \nabla f_s(x_\tau)$, where $\tau \leq t$. We assume that $\forall t$, $t - \tau \leq k$.*

Remark 2. *Zeno++ does not require bounded delay for the workers. The bounded delay requirement in Assumption 1 is only for the validation gradient v on the server.*

Assumption 2. *There exists at least one global minimum x_*, where $F(x_*) \leq F(x), \forall x$.*

We first analyze the convergence of functions that satisfy the PL inequality.

Theorem 1. *Assume that $F(x)$ and $f_s(x)$ are L-smooth and satisfy the PL inequality. Assume that $\forall x$, the correct gradients and validation gradients are upper-bounded: $\|\nabla F(x)\|^2 \leq V_1$, $\|\nabla f_s(x)\|^2 \leq V_1$, and the validation gradients are always non-zero and lower-bounded: $\|\nabla f_s(x)\|^2 \geq V_2$, where $0 < V_2 \leq V_1$. Furthermore, we assume that the validation set is close to the training set, which implies bounded variance: $\mathbb{E}\left[\|\nabla f_s(x) - \nabla F(x)\|^2\right] \leq V_3, \forall x$. Taking $\gamma < \min(1, \frac{1}{L})$ and $\rho \geq \frac{\alpha\sqrt{\gamma}V_1}{2\mu V_2}$, after T global updates, Algorithm 2 has the error bound: $\mathbb{E}\left[F(x_T) - F(x_*)\right] \leq (1 - \alpha\sqrt{\gamma})^T \left[F(x_0) - F(x_*)\right] + \frac{\sqrt{\gamma}}{\alpha}\mathcal{O}(k^2 V_1 + V_3 + \epsilon)$.*

Remark 3. *The assumption of the lower bounded gradient $\|\nabla f_s(x)\|^2 \geq V_2$ is necessary. We need $\nabla f_s(x) \neq 0$, so that the normalization in Line 6 and inner product in Line 7 of Algorithm 2 are feasible. In practice, if we have a mini-batch with zero gradient $\nabla f_s(x) = 0$ on server, we can simply draw additional samples and add them to the mini-batch, until such gradient is non-zero.*

For general smooth but non-convex functions, we have the following convergence guarantee.

Theorem 2. *Assume that $F(x)$ and $f_s(x)$ are L-smooth and potentially non-convex. Assume that $\forall x$, the true gradients and validation gradients are upper-bounded: $\|\nabla F(x)\|^2 \leq V_1$, $\|\nabla f_s(x)\|^2 \leq V_1$, and the validation gradients are always non-zero and lower-bounded: $\|\nabla f_s(x)\|^2 \geq V_2$, where $0 < V_2 \leq V_1$. Furthermore, we assume that the validation set is close to the training set, which implies bounded variance: $\mathbb{E}\left[\|\nabla f_s(x) - \nabla F(x)\|^2\right] \leq V_3, \forall x$. Taking $\gamma < \min(1, \frac{1}{L})$ and $\rho \geq \frac{\alpha\sqrt{\gamma}V_1}{V_2}$, after T global updates, Algorithm 2 has the error bound: $\frac{\mathbb{E}\left[\sum_{t \in [T]} \|\nabla F(x_{t-1})\|^2\right]}{T} \leq \frac{\mathbb{E}[F(x_0) - F(x_*)]}{\alpha\sqrt{\gamma}T} + \frac{\sqrt{\gamma}}{\alpha}\mathcal{O}(k^2 V_1 + V_3 + \epsilon)$.*

Furthermore, if we take $\gamma = \frac{1}{LT}$, then we have $\frac{\mathbb{E}\left[\sum_{t \in [T]} \|\nabla F(x_{t-1})\|^2\right]}{T} \leq \mathcal{O}\left(\frac{1}{\alpha\sqrt{T}}\right)$.

Remark 4. *ρ controls the trade-off between the acceptance ratio and the convergence rate. Large positive ρ makes the convergence faster, but fewer candidate gradients pass the test of `Zeno++`. Small positive ρ increases the acceptance ratio, but may also potentially slow down the convergence or incur larger variance. We use $\alpha > 0$ to bridge ρ to the convergence rate and the variance. Larger α makes ρ larger, which improves the convergence rate, but also enlarges the variance. Using non-zero ϵ potentially results in negative thresholds, which enlarges the acceptance ratio, but also increases the false negative ratio (the ratio of Byzantine gradients that are not filtered out by `Zeno++`).*

6. Experiments

In this section, we evaluate the proposed algorithm, `Zeno++`. Note that we do not evaluate the auxiliary algorithm `Zeno+`, since its computation overhead is too large for practical settings. Due to the space limitation, zoomed figures and additional experiments (including evaluation on an additional types of attacks, and testing the sensitivity to

hyperparameters) are presented in the appendix. We conduct experiments on two benchmarks: CIFAR-10 image classification dataset (Krizhevsky, 2009), and WikiText-2 language modeling dataset (Merity et al., 2017).

6.1. Baselines

We use the asynchronous SGD without attacks as the gold standard, referred to as `AsyncSGD without attack`. Since `Kardam` is the only previous work on Byzantine-tolerant asynchronous SGD, we use it as the baseline.

One may conjecture that `Zeno++` is analogous to training on the validation data. To explore this, we consider training only on $\mathcal{S}$ – assumed to be clean data on the server, i.e., update the model only using $v = \nabla f_s(x)$ on the server, without using any workers. We call this baseline `Server-only`. We fine-tune the learning rate and show the best results of `Server-only`.

6.2. CIFAR-10

The CIFAR-10 image classification dataset (Krizhevsky, 2009) is composed of 50k images for training and 10k images for testing. We use a convolutional neural network (CNN) with 4 convolutional layers followed by 2 dense layers. The detailed network architecture can be found in our submitted source code (will be released upon publication). From the training set, we randomly extracted 2.5k of them as the validation set for `Zeno++`, the remaining are randomly partitioned onto all the workers. In each experiment, we launch 10 worker processes. We repeat each experiment 10 times and take the average. Each experiment is composed of 200 epochs, where each epoch is a full pass of the training dataset. We simulate asynchrony by drawing random delay from a uniform distribution in the range of $[0, k_w]$, where k_w is the maximum worker delay (different from the maximum server delay k of `Zeno++`). In all the experiments, we take the learning rate $\gamma = 0.1$, mini-batch size $n = n_s = 128$, $\rho = 0.002$, $\epsilon = 0.1$, $k = 10$.

We use the top-1 accuracy on the testing set and the cross-entropy loss function on the training set as the evaluation metrics. We also report the false positive rate (FP), which is the ratio of correct gradients that are recognized as Byzantine and filtered out by `Zeno++` or the `Kardam` baseline.

6.2.1. Empirical Results

We first test the convergence when there are no attacks. For `Kardam`, we take $q = 2$ (i.e. here `Kardam` assumes that there are 2 Byzantine workers). The result is shown in Figure 1. `Zeno++` converges a little bit slower than `AsyncSGD`, but faster than `Kardam`, especially when the worker delay is large. When $k_w = 10$, `Zeno++` converges much faster than Kardam. `Server-only` performs badly on both training and testing data.

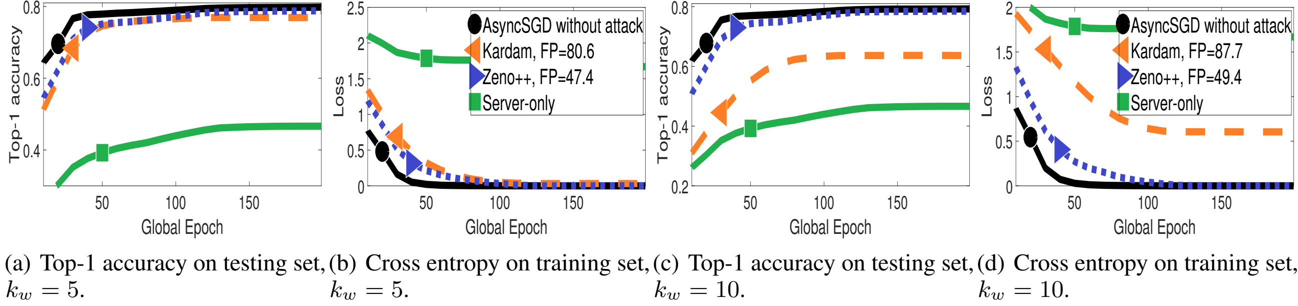
We test the Byzantine-tolerance to the “sign-flipping” attack, which was proposed in (Damaskinos et al., 2018). In such attacks, the Byzantine workers send $-10\nabla f(x)$ instead of the correct gradient $\nabla f(x)$ to the server. The result is shown in Figure 2, with different number of Byzantine workers q. It is shown that when $q = 4$, `Zeno++` converges slightly slower than `AsyncSGD` without attacks, and much faster than `Kardam`. Actually, we observe that `Kardam` fails to make progress when the worker delay is large. When the number of Byzantine workers gets larger ($q = 8$), the convergence of `Zeno++` gets slower, but it still makes reasonable progress, while `AsyncSGD` and `Kardam` fail. Note that `Kardam` performs even worse than `Server-only`, which means that `Kardam` is not even as good as training on a single honest worker. Thus, when there are Byzantine workers, distributed training with `Kardam` is meaningless.

In Figure 3, we show how the hyperparameters ρ, ϵ, and k affect the convergence. In general, `Zeno++` is insensitive to ϵ. Larger ρ and k slow down the convergence.

6.2.2. Discussion

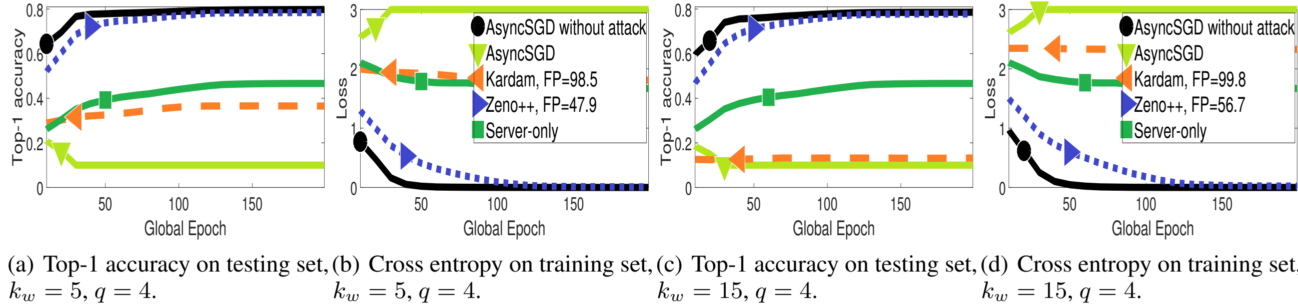
`Kardam` performs surprisingly badly in our experiments. The experiments in (Damaskinos et al., 2018) focus on dampening staleness when there are no Byzantine failures. For Byzantine tolerance, Damaskinos et al. (2018) only reports that `Kardam` filters out 100% of the Byzantine gradients, which matches the results in our experiments. However, we observe that in addition to filtering out 100% of the Byzantine gradients, `Kardam` also filters nearly 100% of the correct gradients. In Figure 2, we report that the false positive rate of `Kardam` is nearly 99%, which makes the convergence extremely slow. To make things worse, `Kardam` does not even perform as good as `Server-only`. For these reasons, we discourage the use of `Kardam` for distributed training. One reason why `Kardam` performs badly is that we use a more general threat model in this paper, which does not guarantee an important assumption of `Kardam`, namely “any sequence of successively received gradients of length $2q + 1$ must contain at least $q + 1$ gradients from honest workers”. It is clear that this assumption is quite strong, as in an asynchronous setting, Byzantine workers can easily send long sequences of erroneous responses. Our approach does not depend on such a strong assumption.

In all the experiments, `Zeno++` converges faster than the baselines when there are Byzantine failures. Although the convergence of `Zeno++` is slower than `AsyncSGD` when there are no attacks, we find that it provides a reasonable trade-off between security and convergence speed. In general, larger worker delay k_w and more Byzantine workers q add more error and noise to the gradients, which slows down the convergence, because there are fewer valid gradients for

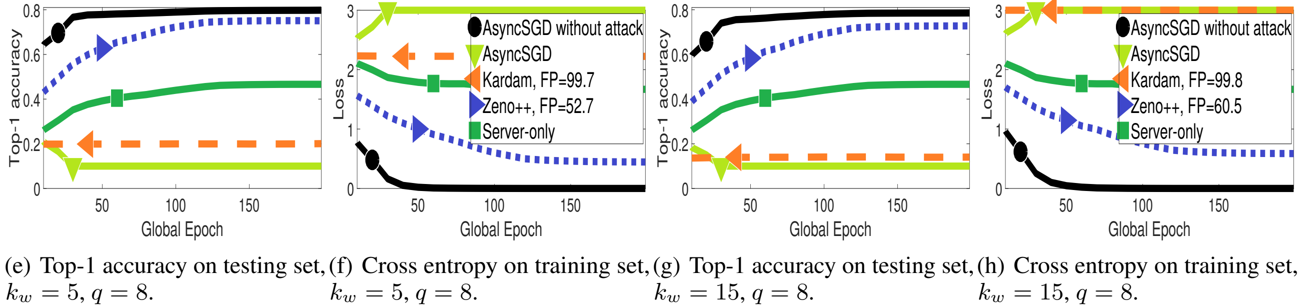


(a) Top-1 accuracy on testing set, $k_w = 5$. (b) Cross entropy on training set, $k_w = 5$. (c) Top-1 accuracy on testing set, $k_w = 10$. (d) Cross entropy on training set, $k_w = 10$.

Figure 1. Results on CNN and CIFAR-10, without attacks, with different maximum worker delays k_w. $\rho = 0.002, \epsilon = 0.1, k = 10$ for Zeno++. *FP* refers to the fraction of false positive detect ions i.e. incorrect prediction that a message is Byzantine.

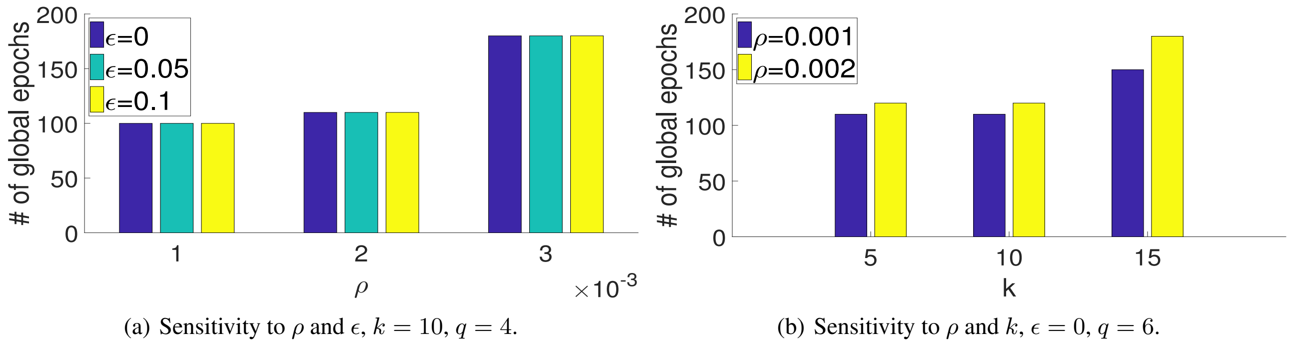


(a) Top-1 accuracy on testing set, $k_w = 5, q = 4$. (b) Cross entropy on training set, $k_w = 5, q = 4$. (c) Top-1 accuracy on testing set, $k_w = 15, q = 4$. (d) Cross entropy on training set, $k_w = 15, q = 4$.



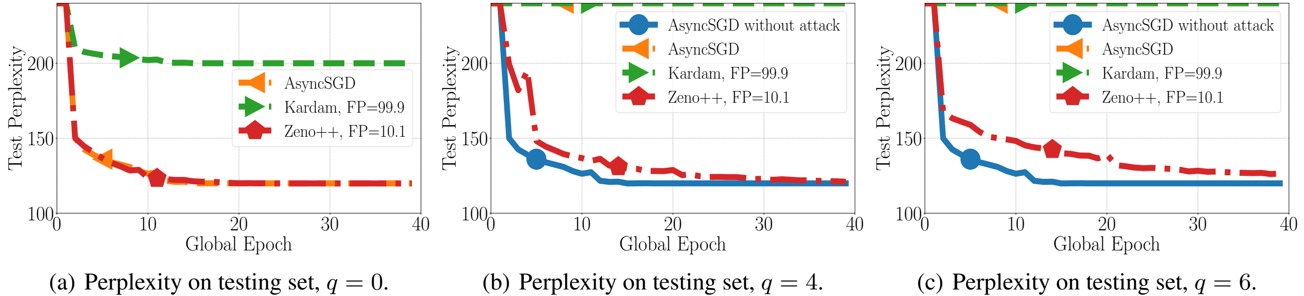
(e) Top-1 accuracy on testing set, $k_w = 5, q = 8$. (f) Cross entropy on training set, $k_w = 5, q = 8$. (g) Top-1 accuracy on testing set, $k_w = 15, q = 8$. (h) Cross entropy on training set, $k_w = 15, q = 8$.

Figure 2. Results on CNN and CIFAR-10, with sign-flipping attacks, and different maximum worker delays k_w. For any correct gradient g, if selected to be Byzantine, g will be replaced by $-10g$. $q \in \{4, 8\}$ out of the 10 workers are Byzantine. $\rho = 0.002, \epsilon = 0.1, k = 10$ for Zeno++. *FP* refers to the fraction of false positive detect ions i.e. incorrect prediction that a message is Byzantine.



(a) Sensitivity to ρ and ϵ, $k = 10$, $q = 4$. (b) Sensitivity to ρ and k, $\epsilon = 0$, $q = 6$.

Figure 3. Sensitivity to the hyperparameters on CNN and CIFAR-10. We test the number of global epochs to reach training loss value 0.2, with sign-flipping attacks. $k_w = 15$.



(a) Perplexity on testing set, $q = 0$. (b) Perplexity on testing set, $q = 4$. (c) Perplexity on testing set, $q = 6$.

Figure 4. Perplexity (the lower the better) on LSTM-based language model and WikiText-2 dataset. We show the convergence under sign-flipping attacks with maximum worker delays $k_w = 10$. For any correct gradient g, if selected to be Byzantine, g will be replaced by $-10g$. $q \in \{0, 4, 6\}$ out of the 10 workers are Byzantine. $\gamma = 20$ for all the algorithms. $\rho = 10, \epsilon = 2.0, k = 10$ for `Zeno++`. *FP* refers to the fraction of false positive detect ions i.e. incorrect prediction that a message is Byzantine. Note that when $q = 4$ or 6, the results of Kardam are off the charts.

the server to use. `Zeno++` can filter out most of the harmful gradients at the cost of FP $\approx 50\%$.

Note that `Server-only` is an extreme case that only uses the server and the validation dataset to train the model in a non-distributed manner, which will not be affected by Byzantine workers. However, only using the validation data is not enough for training, as shown in Figure 1. Similarly in practice, we can use a small dataset separated from the training data for cross-validation, but will never directly train the model only on such validation dataset. Furthermore, as shown in Figure 2, `Zeno++` performs much better than `Server-only`. Thus, we can draw the conclusion that `Zeno++` is efficiently training the model on the honest workers in a distributed manner, which is not equivalent to training on the validation dataset only.

On average, the server computes $\frac{n_s}{k} = 12.8$ gradients in each iteration, since the validation gradient v of `Zeno++` is updated after every $k = 10$ iterations. Thus, the workload on the server is much smaller than a worker. Furthermore, since we can parallelize the workload on the server and workers, the computation overhead of v can be hidden, so that `Zeno++` can benefit from distributed training.

6.3. WikiText-2

The WikiText-2 dataset contains over 2 million tokens from Wikipedia articles, and annotations from Wikidata. We train a LSTM-based language model with one LSTM layer, and one dense layer. The dimension of the hidden state is 200. The back propagation through time (BPTT) is 35. In all the experiments, we take the learning rate $\gamma = 20$, mini-batch size $n = n_s = 20$, $k = k_w = 10$, $\rho = 10$, $\epsilon = 2$. Each experiment runs 40 epochs, where each epoch is a full pass of the training data. The other basic settings are the same as the experiments on CIFAR-10.

We use the perplexity (the exponential of the loss value) on the testing set as the evaluation metric. We also report the false positive rate (FP), which is the ratio of correct gradients that are recognized as Byzantine and filtered out by `Zeno++` or the `Kardam` baseline.

6.3.1. Empirical Results

In Figure 4, we test the Byzantine-tolerance to the "sign-flipping" attack, on LSTM-based language models and WikiText-2 dataset. It is shown that when there are no Byzantine workers, `Zeno++` converges as fast as vanilla asynchronous SGD. When there are Byzantine workers, `Zeno++` converges slightly slower than `AsyncSGD` without attacks, and much faster than `Kardam`. When the number of Byzantine workers gets larger, the convergence of `Zeno++` gets slower. In overall, on the language models, `Zeno++` achieves performance similar to what we observe for CNN.

6.3.2. Discussion

We show that `Zeno++` can protect asynchronous SGD from Byzantine failures on the workers in multiple applications, including image classification and language models. Compared to CIFAR-10, with smaller k_w and q for WikiText-2, the FP of `Zeno++` can be as low as 10%. Note that the choices of ρ and ϵ depends on the learning rate γ. Large γ requires large ρ and ϵ. Kardam still performs badly due to the large false positive rates.

7. Conclusion

We propose a novel Byzantine-tolerant fully asynchronous SGD algorithm: Zeno++. Zeno++ provably converges. Our empirical results show good performance compared to previous work. In future work, we will explore variations of our approach for other settings such as federated learning.

Acknowledgements

This work was funded in part by the following grants: NSF IIS 1909577, NSF CNS 1908888, and a JP Morgan Chase Fellowship, along with computational resources donated by Intel, AWS, and Microsoft Azure.

References

Alistarh, D., Allen-Zhu, Z., and Li, J. Byzantine Stochastic Gradient Descent. In *NeurIPS*, 2018.

Blanchard, P., Mhamdi, E. M. E., Guerraoui, R., and Stainer, J. Machine Learning with Adversaries: Byzantine Tolerant Gradient Descent. In *NeurIPS*, 2017.

Cao, X. and Lai, L. Robust Distributed Gradient Descent with Arbitrary Number of Byzantine Attackers. In *ICASSP*. IEEE, 2018.

Chen, L., Wang, H., Charles, Z. B., and Papailiopoulos, D. S. DRACO: Byzantine-resilient Distributed Training via Redundant Gradients. In *ICML*, 2018.

Chen, Y., Su, L., and Xu, J. Distributed Statistical Machine Learning in Adversarial Settings: Byzantine Gradient Descent. *POMACS*, 1:44:1–44:25, 2017.

Damaskinos, G., Mhamdi, E. M. E., Guerraoui, R., Patra, R., and Taziki, M. Asynchronous Byzantine Machine Learning (the case of SGD). In *ICML*, 2018.

Dutta, S., Joshi, G., Ghosh, S., Dube, P., and Nagpurkar, P. Slow and Stale Gradients Can Win the Race: Error-Runtime Trade-offs in Distributed SGD. In *AISTATS*, 2018.

Feng, J., Xu, H., and Mannor, S. Distributed Robust Learning. *arXiv preprint arXiv:1409.5937*, 2014.

Huber, P. J. *Robust Statistics*, volume 523. John Wiley & Sons, 2004.

Krizhevsky, A. Learning Multiple Layers of Features from Tiny Images. 2009.

Lamport, L., Shostak, R., and Pease, M. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

Lian, X., Zhang, W., Zhang, C., and Liu, J. Asynchronous Decentralized Parallel Stochastic Gradient Descent. In *ICML*, 2018.

Merity, S., Xiong, C., Bradbury, J., and Socher, R. Pointer Sentinel Mixture Models. In *ICLR*, 2017.

Mhamdi, E. M. E., Guerraoui, R., and Rouault, S. The Hidden Vulnerability of Distributed Learning in Byzantium. In *ICML*, 2018.

Polyak, B. T. Gradient methods for minimizing functionals. *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki*, 3(4):643–653, 1963.

Su, L. and Vaidya, N. H. Fault-Tolerant Multi-Agent Optimization: Optimal Iterative Distributed Algorithms. In *PODC*, 2016.

Su, L. and Vaidya, N. H. Defending non-Bayesian learning against adversarial attacks. *Distributed Computing*, pp. 1–13, 2018.

Xie, C., Koyejo, O., and Gupta, I. Phocas: Dimensional Byzantine-resilient Stochastic Gradient Descent. *arXiv preprint arXiv:1805.09682*, 2018.

Xie, C., Koyejo, O., and Gupta, I. SLSGD: Secure and Efficient Distributed On-device Machine Learning. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pp. 213–228. Springer, 2019a.

Xie, C., Koyejo, S., and Gupta, I. Zeno: Distributed Stochastic Gradient Descent with Suspicion-based Fault-tolerance. In *ICML*, 2019b.

Xie, C., Koyejo, S., and Gupta, I. Fall of Empires: Breaking Byzantine-tolerant SGD by Inner Product Manipulation. In *UAI*, 2019c.

Yin, D., Chen, Y., Ramchandran, K., and Bartlett, P. Byzantine-Robust Distributed Learning: Towards Optimal Statistical Rates. In *ICML*, 2018.

Zheng, S., Meng, Q., Wang, T., Chen, W., Yu, N., Ma, Z., and Liu, T.-Y. Asynchronous Stochastic Gradient Descent with Delay Compensation. In *ICML*, 2017.

Zhou, Z., Mertikopoulos, P., Bambos, N., Glynn, P. W., Ye, Y., Li, L.-J., and Fei-Fei, L. Distributed Asynchronous Optimization with Unbounded Delays: How Slow Can You Go? In *ICML*, 2018.

Zinkevich, M., Langford, J., and Smola, A. J. Slow Learners are Fast. In *NeurIPS*, 2009.