Genuinely Distributed Byzantine Machine Learning

El-Mahdi El-Mhamdi*
Rachid Guerraoui*
Arsany Guirguis*
Lê Nguyên Hoang*
Sébastien Rouault*
firstname.lastname@epfl.ch
EPFL
Lausanne (VD), Switzerland

ABSTRACT

Machine Learning (ML) solutions are nowadays distributed, according to the so-called *server/worker* architecture. One *server* holds the model parameters while several *workers* train the model. Clearly, such architecture is prone to various types of component failures, which can be all encompassed within the spectrum of a Byzantine behavior. Several approaches have been proposed recently to tolerate Byzantine workers. Yet all require trusting a central parameter server. We initiate in this paper the study of the *"general" Byzantine-resilient* distributed machine learning problem where no individual component is trusted. In particular, we distribute the parameter server computation on several nodes.

We show that this problem can be solved in an asynchronous system, despite the presence of $\frac{1}{3}$ Byzantine parameter servers and $\frac{1}{3}$ Byzantine workers (which is optimal). We present a new algorithm, ByzSGD, which solves the general Byzantine-resilient distributed machine learning problem by relying on three major schemes. The first, *Scatter/Gather*, is a communication scheme whose goal is to bound the maximum drift among models on correct servers. The second, *Distributed Median Contraction* (DMC), leverages the geometric properties of the median in high dimensional spaces to bring parameters within the correct servers back close to each other, ensuring learning convergence. The third, *Minimum–Diameter Averaging* (*MDA*), is a statistically–robust gradient aggregation rule whose goal is to tolerate Byzantine workers. *MDA* requires loose bound on the variance of non–Byzantine gradient estimates, compared to existing alternatives (e.g., Krum [12]). Interestingly, ByzSGD ensures Byzantine resilience without adding communication rounds (on a normal path), compared to vanilla non-Byzantine alternatives. ByzSGD requires, however, a larger number of messages which, we show, can be reduced if we assume synchrony.

We implemented ByzSGD on top of TensorFlow, and we report on our evaluation results. In particular, we show that ByzSGD achieves convergence in Byzantine settings with around 32% overhead compared to vanilla TensorFlow. Furthermore, we show that ByzSGD's throughput overhead is 24–176% in the synchronous case and 28–220% in the asynchronous case.

KEYWORDS

distributed machine learning, Byzantine fault tolerance, Byzantine parameter servers

ACM Reference Format:
El-Mahdi El-Mhamdi, Rachid Guerraoui, Arsany Guirguis, Lê Nguyên Hoang, and Sébastien Rouault. 2020. Genuinely Distributed Byzantine Machine Learning. In *PODC '20: The ACM Symposium on Principles of Distributed Computing, August 03–07, 2020, Salerno, Italy*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3382734.3405695

1 INTRODUCTION

Distributed ML is fragile. Distributing Machine Learning (ML) tasks seems to be the only way to cope with ever-growing datasets [17, 30, 38]. A common way to distribute the learning task is through the now classical *parameter server* architecture [35, 36]. In short, a central *server* holds the model parameters (e.g., weights of a neural network) whereas a set of *workers* perform the backpropagation computation [28], typically following the standard optimization algorithm: *stochastic gradient descent* (SGD) [43], on their local data, using the latest model they pull from the server. This server in turn gathers the updates from the workers, in the form of *gradients*, and aggregates them, usually through averaging [31]. This scheme is, however, very fragile because averaging does not tolerate a single corrupted input [12], whilst the multiplicity of machines increases the probability of a misbehavior somewhere in the network.

This fragility is problematic because ML is expected to play a central role in safety-critical tasks such as driving and flying people, diagnosing their diseases, and recommending treatments to their doctors [23, 33]. Little room should be left for the routinely reported [11, 27, 39] vulnerabilities of today's ML solutions.

Byzantine-resilient ML. Over the past three years, a growing body of work, e.g., [5, 9, 12, 16, 20, 46, 48, 51], took up the challenge of *Byzantine-resilient ML*. The Byzantine failure model, as originally introduced in distributed computing [32], encompasses crashes, software bugs, hardware defects, message omissions, corrupted data, and even worse, hacked machines [10, 49]. So far, all the work on Byzantine-resilient ML assumed that a fraction of workers could be Byzantine. But all assumed the central parameter server to be always *honest* and *failure-free*. In other words, none of the previous approaches considered a genuinely Byzantine–resilient distributed ML system, in the distributed computing sense, where

*Equal contribution. Authors are listed alphabetically.

PODC '20, August 03–07, 2020, Salerno, Italy

ACM ISBN 978-1-4503-XXXX-X/18/06.
https://doi.org/10.1145/3382734.3405695

no component is trusted. Consider a multi–branch organization with sensitive data, e.g., a hospital or a bank, that would like to train an ML model among its branches. In such a situation, the worker machines, as well as the central server, should be robust to the worst: the adversarial attacks.

A natural way to prevent the parameter server from being a *single point–of–failure* is to *replicate* it. But this poses the problem of how to synchronize the replicas. The classical synchronization technique, *state machine replication* [14, 44] (SMR), enforces a *total order for updates* on all replicas through *consensus*, providing the abstraction of a single parameter server, while benefiting from the resilience of the multiplicity of underlying replicas. Applying SMR to distributed SGD would however lead to a potentially huge *overhead*. In order to maintain the same state, replicas would need to agree on a total order of the model updates, inducing frequent exchanges (and retransmissions) of gradients and parameter vectors, that can be several hundreds of MB large [30]. Given that a distributed ML setup is network bound [29, 52], SMR is impractical in this context.

The key insight underlying our paper is that the *general Byzantine SGD problem*, even when neither the workers nor the servers are trusted, is easier than consensus, and total ordering of updates is not required in the context of ML applications; only convergence to a good final cross-accuracy is needed. We thus follow a different route where we do not require all the servers to maintain the same state. Instead, we allow mildly diverging parameters (which have proven beneficial in other contexts [6, 53]) and present new ways to *contract* them in a *distributed* manner.

Contributions. In short, we consider, for the first time in the context of distributed SGD, the general Byzantine ML problem, where no individual component is trusted. First, we show that this problem can be solved in asynchronous settings, and then we show how we can utilize synchrony to boost the performance of our asynchronous solution.

Our main algorithm, ByzSGD, achieves *general* resilience without assuming bounds on communication delays, nor inducing additional communication rounds (on a normal path), when compared to a non-Byzantine resilient scheme. We prove that ByzSGD tolerates $\frac{1}{3}$ Byzantine servers and $\frac{1}{3}$ Byzantine workers, which is optimal in an asynchronous setting. ByzSGD employs a novel communication scheme, *Scatter/Gather*, that bounds the maximum drift between models on correct servers. In the *scatter* phase, servers work independently (they do not communicate among themselves) and hence, their views of the model could drift away from each other. In the *gather* phase, correct servers communicate and apply collectively a *Distributed Median-based Contraction* (DMC) module. This module is crucial for it brings the diverging parameter vectors back closer to each other, despite each parameter server being only able to gather a fraction of the parameter vectors. Interestingly, ByzSGD ensures Byzantine resilience without adding communication rounds (on a normal path), compared to non-Byzantine alternatives. ByzSGD requires, however, a larger number of messages which we show we can reduce if we assume synchrony. Essentially, in a synchronous setting, workers can use a novel filtering mechanism we introduce to eliminate replies from Byzantine servers without requiring to communicate with all servers.

ByzSGD[1] uses a *statistically–robust gradient aggregation rule* (GAR), which we call *Minimum–Diameter Averaging* (*MDA*) to tolerate Byzantine workers. Such a choice of a GAR has two advantages compared to previously–used GARs in the literature to tolerate Byzantine workers, e.g., [12, 50]. First, *MDA* requires a loose bound on the variance of the non–Byzantine gradient estimates, which makes it practical[2]. Second, *MDA* makes the best use of the variance reduction resulting from the gradient estimates on multiple workers, unlike Krum [12] and Median [50].

We prove that ByzSGD guarantees convergence despite the presence of Byzantine machines, be they workers or servers. We implemented ByzSGD on top of TensorFlow [3], while achieving transparency: applications implemented with TensorFlow need not to change their interfaces to be made Byzantine–resilient. We report on our evaluation of ByzSGD. We show that ByzSGD tolerates Byzantine failures with a reasonable convergence overhead ($\sim 32\%$) compared to the non Byzantine-resilient vanilla TensorFlow deployment. Moreover, we show that the throughput overhead of ByzSGD ranges from 24% to 220% compared to vanilla TensorFlow.

The paper is organized as follows. Section 2 provides some background on SGD, the problem settings and the threat model. Section 3 describes our ByzSGD algorithm, while Section 4 sketches its correctness proof. Section 5 discusses how ByzSGD leverages synchrony to boost performance. Section 6 reports on our empirical evaluation of ByzSGD. Section 7 concludes the paper by discussing related work and highlighting open questions. In the Appendix to the main paper, we give the convergence proof of ByzSGD (Appendix C), we empirically validate our assumptions (Appendix D), and we assess ByzSGD's performance in a variety of cases (Appendix E).

2 BACKGROUND AND MODEL

2.1 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) [43] is a widely-used *optimization* algorithm in ML applications [3, 17, 35]. Typically, SGD is used to minimize a *loss function* $L(\theta) \in \mathbb{R}$, which measures how accurate the model θ is when classifying an input. Formally, SGD addresses the following optimization problem:

$$\arg\min_{\theta \in \mathbb{R}^d} L(\theta) \tag{1}$$

SGD works iteratively and consists, in each step t, of:

(1) Estimating the gradient $G\left(\theta^{(t)}, \xi\right)$, with a subset ξ of size b of the training set, called *mini–batch*. Such a gradient is a *stochastic* estimation of the real, uncomputable one $\nabla L\left(\theta^{(t)}\right)$.
(2) Updating the parameters following the estimated gradient:

$$\theta^{(t+1)} = \theta^{(t)} - \eta_t \cdot G\left(\theta^{(t)}, \xi\right) \tag{2}$$

The sequence $\{\eta_t \in \mathbb{R}_{>0}\}$ is called the *learning rate*.

[1] We use the same name for our asynchronous and synchronous variants of the algorithm when there is no ambiguity.

[2] In Appendix D, we report on our experimental validation of such a requirement. We compare the bound on the variance require by *MDA* to the one required by *Multi–Krum* [12], a state–of–the–art GAR. For instance, we show that with a batch–size of 100 and assuming 1 Byzantine failure, the requirement of *MDA* is satisfied in our experiments, while that of *Multi–Krum* is not.

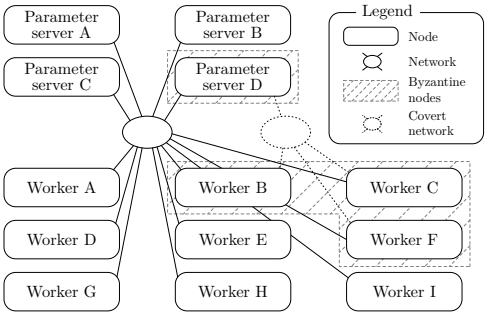

Figure 1: A distributed ML setup with 4 parameter servers and 9 workers, including respectively 1 and 3 Byzantine nodes, which all can be viewed as a single adversary.

2.2 The Parameter Server Architecture

Estimating one gradient is computationally expensive, as it consists in computing b estimates of $G(\theta, \xi_i)$, where ξ_i is the i^{th} pair (input, label) from the mini–batch, and where each $G(\theta, \xi_i)$ involves one *backpropagation* computation [28]. Hence, the amount of arithmetic operations to carry out to estimate $G(\theta, \xi) \approx \nabla L\left(\theta^{(t)}\right)$ is $O(b \cdot d)$.

However, this gradient estimation can be easily distributed: the b computations of $G\left(\theta^{(t)}, \xi_i\right)$ can be executed in parallel on n machines, where the aggregate of such computations gives $G\left(\theta^{(t)}, \xi\right)$. This corresponds to the now standard *parameter server* architecture [36], where a central server holds the parameters θ.

Each training step includes two communication rounds: the server first broadcasts the parameters to workers, which then estimate the gradient $G\left(\theta^{(t)}, \xi_i\right)$ (i.e., each with a mini–batch of $\frac{b}{n}$). When a worker completes its estimation, it sends it back to the parameter server, which in turn *averages* these estimations and updates the parameters θ, as in Equation 2.

2.3 Byzantine Machine Learning

The Byzantine failure abstraction [32] models any arbitrary behavior and encompasses software bugs, hardware defects, message omissions, or even hacked machines. We then typically assume that a subset of the machines can be Byzantine and controlled by an adversary whose sole purpose is to defeat the computation.

A Byzantine machine can, for instance, send a biased estimate of a gradient to another machine, which leads to a corrupted learning model accordingly or even to learning divergence [8]. Byzantine failures also abstract the *data poisoning* problem [10], which happens when a machine owns maliciously–labeled data. This may result in learning a corrupted model, especially–crafted by the adversary. Clearly, assuming a central machine controlling the learning process (as with the standard parameter server architecture [36]) is problematic if such a machine is controlled by an adversary, for this machine can write whatever it wants to the final model, jeopardizing the learning process.

2.4 System Model

Computation and communication models. We build on the standard parameter server model, with two main variations (Fig. 1).

Table 1: The notations used throughout this paper.

n_{ps}	Total number of parameter servers
f_{ps}	Maximal number of Byzantine parameter servers
q_{ps}	Number of parameter vectors a node waits for from servers, $2f_{ps} + 2 \leq q_{ps} \leq n_{ps} - f_{ps}$
n_w	Total number of workers
f_w	Declared, maximal number of Byzantine workers
q_w	Number of gradients a node waits for from workers, $2f_w + 1 \leq q_w \leq n_w - f_w$
d	Dimension of the parameter space $\mathbb{R}^d$
L	Loss function we aim to minimize
l	Lipschitz constant of the loss function
$\theta_t^{(i)}$	Parameter vector (i.e. model) at the parameter server i at step t
$\mathcal{G}_t^{(i)}$	Gradient distribution at the worker i at step t
$g_t^{(i)}$	Stochastic gradient estimation of worker i at step t
$\nabla L(\theta)$	Real gradient of the loss function L at θ
$\widehat{\nabla L}(\theta)$	A stochastic estimation of $\nabla L(\theta)$
η_t	Learning rate at step t

Without loss of generality in the analysis, we note:

$[1 \,..\, n_{ps} - f_{ps}]$	the indexes of the correct servers
$[1 \,..\, n_w - f_w]$	the indexes of the correct workers

1. We assume a subset of the nodes (i.e., machines) involved in the distributed SGD process to be *adversarial*, or *Byzantine*, in the parlance of [12, 16, 22]. The other nodes are said to be *correct*. Clearly, which node is correct and which is Byzantine is not known ahead of time. **2.** We consider several *replicas* of the parameter server (we call them *servers*), instead of a single one, preventing it from being a single *point–of–failure*, unlike in classical ML approaches.

In our new context, workers send their gradients to all servers (instead of one in the standard case), which in turn send their parameters to all workers. Periodically, servers communicate with each other, as we describe later in Section 3. We consider *bulk–synchronous training*: only the gradients computed at a learning step t are used for the parameters update at the same step t.

Adversary capabilities. The adversary is an entity that controls all Byzantine nodes (Figure 1), and whose goal is to prevent the SGD process from converging to a state that could have been achieved if there was no adversary. As in [12, 16, 22], we assume an omniscient adversary that can see the full training datasets as well as the packets being transferred over the network (i.e., gradients and models). However, the adversary is not omnipotent: it can only send arbitrary messages from the nodes it controls, or force them to remain silent. We assume nodes can authenticate the source of a message, so no Byzantine node can forge its identity or create multiple fake ones [15].

2.5 Convergence Conditions

We assume the classical conditions[3] for convergence in non–convex optimization [13]. For instance, we assume that the training data is identically and independently distributed (*i.i.d*) over the workers, and the sequence of learning rates η_t is monotonically decreasing

[3]The exhaustive list, with their formal formulations, is available in Appendix C.1.

(i.e., $\lim_{t\to+\infty} \eta_t = 0$). We also assume that correct workers compute unbiased estimates of the true gradient with sufficiently low variance, namely (see Table 1 for notations):

$$\exists \kappa \in]1, +\infty[\,,\ \forall (i,t,\theta) \in [1\,..\,n_w - f_w] \times \mathbb{N} \times \mathbb{R}^d,$$
$$\kappa \frac{2f_w}{n_w - f_w} \sqrt{\mathbb{E}\left(\left\|g_t^{(i)} - \mathbb{E}\, g_t^{(i)}\right\|_2^2\right)} \leq \left\|\nabla L\left(\theta\right)\right\|_2. \tag{3}$$

Such an equation bounds the ratio of the standard deviation of the stochastic gradient estimations to the norm of the real gradient with $\frac{n_w - f_w}{2f_w}$. This assumption is now classical in the Byzantine ML literature [12, 22]. We also empirically verify this assumption in Appendix D in our experimental setup.

In addition, we assume: (1) L is l-Lipschitz continuous, and (2) no (pattern of) network partitioning among the correct servers lasts forever. The standard l-Lipschitz continuity assumption [7, 41, 47] acts as the only bridge between the parameter vectors $\theta_t^{(i)}$ and the stochastic gradients $g_t^{(i)}$ at a given step t. This is a *liveness* assumption: the value of l can be arbitrarily high and is only used to bound the expected maximal distance between any two correct parameter vectors. The second assumption ensures that eventually the correct parameter servers can communicate with each other in order to pull their views of the model back close to each other; this is crucial to achieve Byzantine resilience.

Denote by $S_j \triangleq \left\{s \subset \left[1\,..\,n_{ps} - f_{ps}\right] - \{j\} \;\middle|\; |s| \in [q - f - 1\,..\,q - 1]\right\}$ the subset of correct parameter server indexes that parameter server j can deliver at a given step. We then call $S \triangleq \prod_{j \in [1\,..\,n_{ps} - f_{ps}]} S_j$ the set of all correct parameter server indexes the $n_{ps} - f_{ps}$ correct parameter servers $i \in \left[1\,..\,n_{ps} - f_{ps}\right]$ can deliver. We further assume $\exists \rho > 0,\ \forall s \in S,\ P\left(X = s\right) \geq \rho$. For any server j, $\left\|\nabla L\left(\theta_t^{(j)}\right)\right\|_2 \geq 9\left(n_w - f_w\right)\sigma' + \frac{Cdl^2}{\rho}$, for some constant C, and where σ' is the upper bound on the distance between the estimated and the true gradients (see Appendix C.1). Finally, ByzSGD requires $n_w \geq 3f_w + 1$ and $n_{ps} \geq 3f_{ps} + 2$.

3 BYZSGD: GENERAL BYZANTINE RESILIENCE

We present here ByzSGD, the first algorithm to tolerate Byzantine workers and servers without making any assumptions on node relative speeds and communication delays. ByzSGD does not add, on average, any communication rounds compared to the standard parameter server communication model (Section 2.2). However, periodically, ByzSGD adds a communication round between servers to enforce contraction and convergence, as we show in this section.

We first describe the fundamental technique to tolerate Byzantine servers: *Distributed Median-based Contraction* (DMC). Then, we describe the Byzantine–resilient *gradient aggregation rule* we use to tolerate Byzantine workers. Finally, we explain the overall functioning of ByzSGD, highlighting our novel *Scatter/Gather* communication scheme.

3.1 *Distributed Median–based Contraction*

The fundamental problem addressed here is induced by the multiplicity of servers and consists of bounding the drift among correct parameter vectors $\theta_t^{(1)} \ldots \theta_t^{(n_{ps} - f_{ps})}$, as t grows. The problem is particularly challenging because of the combination of three constraints: (a) we consider a Byzantine environment, (b) we assume an asynchronous network, and (c) we do not want to add communication rounds, compared to those done by vanilla non–Byzantine deployments, given the expensive cost of communication in distributed ML applications [29, 52]. The challenging question can then be formulated as follows: given that the correct parameter servers should not expect to receive more than $n-f-1$ messages per round, how to keep the correct parameters close to each other, knowing that a fraction of the received messages could be Byzantine?

Our solution to this issue is, what we call, *Distributed Median–based Contraction (DMC)*, whose goal is to decrease the expected maximum distance between any two honest parameter vectors (i.e., *contract* them). DMC is a combination of (1) the application of *coordinate–wise* Median (which is Byzantine–resilient as soon as $q_{ps} \geq 2f_{ps} + 1$) on the parameter vectors and (2) the over–provisioning of 1 more correct parameter server (i.e., $q_{ps} \geq 2f_{ps} + 2$); both constitute the root of what we call the *contraction effect*. Assuming each honest parameter server can deliver a subset of $q_{ps} - f_{ps} - 1$ *honest* parameters, the expected median of the gathered parameters is then both (1) *bounded* between the gathered honest parameters and (2) *different* from any extremum among the gathered honest parameters (as 1 correct parameter server was over-provisioned). Since each subset of the gathered honest parameters contains a subset of all the $n_{ps} - f_{ps}$ honest parameters, the expected maximum distance between two honest parameters is thus decreased after applying DMC.

3.2 *Minimum–Diameter Averaging*

To tolerate Byzantine workers, we use a statistically–robust *Gradient Aggregation Rule* (GAR). A GAR is merely a function of $\left(\mathbb{R}^d\right)^n \to \mathbb{R}^d$. Reminiscent of the *Minimum Volume Ellipsoid* [42], we use *Minimum–Diameter Averaging* (*MDA*) as our Byzantine–resilient GAR. Given n input gradients (with f possible Byzantine ones), *MDA* averages only a subset of $n - f$ gradients. The selected subset must have the *minimum diameter* among all the subsets of size $n - f$ taken from the n input gradients; hence the name. The *diameter* of a subset is defined as the maximum ℓ_2 distance between any two gradients of this subset.

MDA can be formally described as follows. Let $\left(x_1 \ldots x_q\right) \in \left(\mathbb{R}^d\right)^q$, and $Q \triangleq \left\{x_1 \ldots x_q\right\}$ the set of all the input gradients. Let $\mathcal{R} \triangleq \left\{\mathcal{X} \mid \mathcal{X} \subset Q,\ |\mathcal{X}| = q - f\right\}$ the set of subsets of Q with cardinality $q - f$. Let $\mathcal{S} \triangleq \arg\min_{\mathcal{X} \in \mathcal{R}} \left(\max_{(x_i, x_j) \in \mathcal{X}^2} \left(\left\|x_i - x_j\right\|_2\right)\right)$. Then, the aggregated gradient is $\mathit{MDA}\left(x_1 \ldots x_q\right) \triangleq \frac{1}{q-f} \sum_{x \in \mathcal{S}} x$.

3.3 The ByzSGD Algorithm

ByzSGD operates iteratively in two phases: *scatter* and *gather*. One *gather* phase is entered every T steps (lines 8 to 11 in Algorithm 2); we call the whole T steps the *scatter* phase.

Algorithms 1 and 2 depict the training loop applied by workers and servers respectively. As an initialization step, correct servers

Figure 2: ByzSGD: worker and parameter server logic.

Algorithm 1 Worker

1: $t \leftarrow 0$
2: **repeat**
3: ms ← read_models()
4: m.set(*Median*(ms))
5: g ← compute_grad()
6: $t \leftarrow t + 1$
7: **until** $t >$ max_steps

Algorithm 2 Parameter Server

1: Calculate T & *seed*
2: m ← init_model(seed)
3: $t \leftarrow 0$
4: **repeat**
5: gs ← read_gradients()
6: m.update(*MDA*(gs))
7: **if** $t \bmod T = 0$ **then**
8: m ← read_models()
9: m ← *Median*(ms)
10: **end if**
11: $t \leftarrow t + 1$
12: **until** $t >$ max_steps

initialize the model with the same random values, i.e., using the same *seed*. Moreover, the servers compute the value of T.

The subsequent steps $t \in \mathbb{N}$ work as follows. The algorithm starts with the *scatter* phase, which includes doing a few learning steps. In each step, each server i broadcasts its current parameter vector $\theta_t^{(i)}$ to every worker. Each worker j then (1) *aggregates* with *coordinate–wise* Median (hereafter simply, *Median*) the first q_{ps} received $\theta_t^{(i)}$ (line 4 in Algorithm 1), and (2) computes an estimate $g_t^{(j)}$ of the gradient at the aggregated parameter vector, i.e., model. Then, each worker j broadcasts its computed gradient estimation $g_t^{(j)}$ to all parameter servers. Each parameter server i in turn *aggregates* with *MDA* the first q_w received $g_t^{(j)}$ (line 6 in Algorithm 2) and then performs a local parameter update with the aggregated gradient, resulting in $\overline{\theta}_t^{(i)}$. In normal steps (in the *scatter* phase), $\theta_{t+1}^{(i)} = \overline{\theta}_t^{(i)}$.

Every T steps (i.e., in the *gather* phase), correct servers apply DMC: each parameter server i broadcasts $\overline{\theta}_t^{(i)}$ to every other server and then aggregates with *Median* the first q_{ps} received $\overline{\theta}_t^{(k)}$; this aggregated parameter vector is $\theta_{t+1}^{(i)}$.

4 CORRECTNESS OF BYZSGD

This section sketches the convergence proof ByzSGD. The full proof is detailed in Appendix C.

Theorem 1 (Correctness of ByzSGD). *Under the assumptions of Section 2.5, ByzSGD guarantees the convergence of all correct parameter servers, i.e.*

$$\forall i \in \left[1 \,..\, n_{ps} - f_{ps}\right], \lim_{t \to +\infty} \left\| \nabla L\left(\theta_t^{(i)}\right) \right\|_2 = 0. \tag{4}$$

Equation 4 is the commonly admitted termination criteria for non–convex optimization tasks [13]. We provide here the key intuitions and steps of the proof, which relies on two key observations: (1) assuming that the learning rate η_t converges to 0, *Median* guarantees that all correct servers' parameters θ_t all eventually reach the same values (i.e., *contract*) and (2) any given server i's parameters $\theta_t^{(i)}$ eventually essentially follows a stochastic gradient descent dynamics, which guarantees its convergence as described by Equation 4. In the sequel, we detail these two points.

4.1 Contraction by *Median*

The first key element of the proof is to prove that *Median* contracts the servers' parameters in expectation, despite the Byzantines' attacks. This turns out to be nontrivial. In fact, the diameter of servers' parameters does not decrease monotonically. The trick is to follow the evolution of another measure of the spread of the servers' parameters, namely the sum of coordinate-wise diameters. We prove that, using *Median*, Byzantines can never increase this quantity.

Lemma 1 (Safety of *Median*). *Let Δ_{θ_t} the sum of coordinate-wise diameters, i.e.: $\Delta_{\theta_t} = \sum_{i=1}^{d} \max_{j,k \in [1 \,..\, n_{ps} - f_{ps}]} \left| \theta_t^{(j)}[i] - \theta_t^{(k)}[i] \right|$. Then, no matter which gradients are delivered to which servers, and no matter how Byzantines attack, we have $\Delta_{\theta_{t+1}} \leq \Delta_{\theta_t}$.*

Sketch of proof. On each dimension j, since $q_{ps} \geq 2f_{ps}+2$, for any server i, there is a majority of correct servers whose parameters have been delivered to i. Thus, the median computed by server i must belong to the interval containing the j-coordinates of all correct servers. Since this holds for all servers i, the diameter along dimension j cannot increase. □

But interestingly, assuming that all delivering configurations can occur with a positive probability, we can show that there will be contraction in expectation.

Lemma 2 (Expected contraction by *Median*). *There is a constant $m < 1$ such that $\mathbb{E}\left[\Delta_{\theta_{t+1}}\right] \leq m\Delta_{\theta_t}$. Note that the expectation is taken over all delivering configurations, and given Byzantines' attacks for a delivering configuration.*

Sketch of proof. Using the assumption $q_{ps} \geq 2f_{ps} + 2$, for any coordinate, we show the existence of a delivering configuration such that the diameter along this coordinate is guaranteed to decrease by a factor 3/4. The trick to identify this configuration is to divide, for each coordinate i, the set of servers into those that have a high coordinate i and those that have a low coordinate i. One of these two subsets, call it X^*, must contain a majority of correct servers.

Given that $q_{ps} \leq \lfloor \frac{n_{ps} - f_{ps}}{2} \rfloor$, we can thus guarantee the existence of a delivering configuration such that all parameter servers all contain a majority of inputs that come from X^*. If this happens, then all servers' parameters after application of *Median* must then have their i's coordinates closer to the subset X^*. In fact, we show that their new diameters along the coordinate i is at most 3/4 of their previous diameter, which proves the contraction along coordinate i for at least one delivering configuration. Taking the expectation implies a strict expected contraction along this coordinate. Summing up over all coordinates yields the lemma. □

Unfortunately, this is still not sufficient to guarantee the convergence of the servers' parameters, because of a potential drift during learning. However, as the learning rate decreases, we show that contraction eventually becomes inevitable, which guarantees eventual convergence.

Lemma 3 (Bound on the drift). *There exists constants A and B such that $\mathbb{E}\left[\Delta_{\theta_{t+1}}\right] \leq (m + A\eta_t)\Delta_{\theta_t} + B\eta_t$. Here, the expectation is over delivering configurations, from parameter servers to workers, from workers to parameter servers and in-between parameter servers.*

The expectation is also taken over stochastic gradient estimates by workers, and for any attacks by the Byzantines.

Sketch of proof. This bound requires to control the drift despite Byzantine attacks on two different levels, namely when parameter servers deliver their parameters to workers, and when workers deliver their gradients to servers.

In the former case, we exploit the fact that the spread of all correct parameters along, measured in terms of sum of coordinate-wise diameters, is at most Δ_{θ_t}, to guarantee that the coordinate-wise medians of the parameters computed by workers are also close to one another.

Using Lipschitz continuity, this then implies that the gradients they compute are also close to one another, in the sense that the diameter of gradients is at most $l\Delta_{\theta_t}$ plus some deviation $2h_w\sigma'$ due to the noise in the stochastic gradient estimates.

Then, using the Byzantine resilience property of *MDA*, we show that the diameter of servers' aggregated gradients is at most that of workers' estimated gradients. Combining it all yields the lemma.

Note that the constant A depends on the Lipschitz continuity of the gradient of the loss function, while the constant B comes from the randomness of the gradient estimates by workers. See more details in Appendix C.4. □

Lemma 4 (Eventual contraction of *Median*). *As a corollary, assuming $\eta_t \to 0$, we have $\mathbb{E}[\Delta_{\theta_t}] \to 0$.*

Sketch of proof. This comes from the eventual contraction of Δ_{θ_t}. Note that it implies that some other measures of the spread of servers' parameters, like their diameter measured in ℓ_2, also converge to zero. □

4.2 Liveness of Server Parameters

The previous section showed that eventually, the parameters will all have nearly identical values. In this section, we show that, as a result, and thanks to a Byzantine–resilient GAR like *MDA*, any parameter's update is nearly an update that would be obtained by vanilla Byzantine-free SGD, for which the guarantee of Theorem 1 has been proven.

First, we note that *MDA* ensures that its output gradient lies within the correct set of gradients submitted to a correct server, as stated by the following lemma.

Lemma 5 (*MDA* bounded deviation from majority). *The distance between the output of* MDA *and (at least) one of the correct gradients is bounded below the diameter of the set of correct gradients.*

Formally, let $(d, f) \in (\mathbb{N} - \{0\})^2$ and let $q \in \mathbb{N}$ such that $q \geq 2f + 1$. Let note $H \triangleq [1\,..\,q - f]$.

It holds that, $\forall \left(g_1 \dots g_q\right) \in \left(\mathbb{R}^d\right)^q$, $\exists k \in H$, :

$$\left\|\text{MDA}\left(g_1 \dots g_q\right) - g_k\right\|_2 \leq \max_{(i,j)\in H^2} \left\|g_i - g_j\right\|_2 \tag{5}$$

Hence, SGD, with *MDA*, alone would converge.

Now, we show that a server j's update can be written $\theta_{t+1}^{(j)} = \theta_t^{(j)} - \eta_t \hat{G}_t^{(j)}$, where $\hat{G}_t^{(j)}$ satisfies the classical assumptions of Byzantine machine learning. Namely, we show that $\mathbb{E}\,\hat{G}_t^{(j)} \cdot \nabla L\left(\theta_t^{(j)}\right)$ is positive, under our assumptions.

Theorem 2 (Liveness and safety of parameter servers). *Under our assumptions for any parameter server j, $\mathbb{E}\,\hat{G}_t^{(j)} \nabla L\left(\theta_t^{(j)}\right) > 0$.*

Sketch of proof. To guarantee this condition, we first prove that the expected maximum distance between an estimate $G_t^{(j)}$ by worker j and server k's parameters $\nabla L\left(\theta_t^{(k)}\right)$ is upper-bounded by the spread of all servers' parameters and a term due to the randomness of the worker's gradient estimates.

We then show that the servers' updates are all similar, which allows to say that after applying *Median*, all servers are guaranteed to move along the direction of the true gradient, given our assumptions. □

Complexity. The communication complexity of ByzSGD is $O(dn_w n_{ps})$ when T is very large. With small values for T, the communication complexity is $O(dn_{ps}(n_w + n_{ps}))$. The computation complexities of *Median* and *MDA* are $O(n_{ps}d)$ and $O\left(\binom{n_w}{f_w} + n_w^2 d\right)$. Notably, distributed ML problem is network–bound [29, 52] and hence, the (possibly) exponential complexity of *MDA* does not aggressively harm the performance, as we show in Section 6.

5 BYZSGD: REDUCING MESSAGES WITH SYNCHRONY

We show here that assuming network synchrony, we can boost ByzSGD's performance while keeping the same resilience guarantees. In particular, the number of communicated messages can be reduced as follows: instead of pulling an updated model from q_{ps} servers (line 3 in Algorithm 1), each worker pulls only one model and then checks its legitimacy using two filters: *Lipschitz* and *Outliers* filters. In this case, ByzSGD requires $n_w \geq 2f_w + 1$ while keeping $n_{ps} \geq 3f_{ps} + 2$. The full proof of ByzSGD, while considering synchronous networks, is in Appendix C.

5.1 *Lipschitz* Filter

Based on the standard Lipschitz continuity of the loss function assumption [12, 13], previous work used empirical estimations for the Lipschitz coefficient to filter out gradients from Byzantine workers in asynchronous learning [19]. We use a similar idea, but we now apply it to filter out *models* from Byzantine *servers*. The filter works as follows: consider a worker j that owns a model $\theta_t^{(j)}$ and a gradient it computed $g_t^{(j)}$ based on that model at some step t. A correct server i should include $g_t^{(j)}$ while updating its model $\theta_t^{(i)}$, given network synchrony. Worker j then: (1) estimates the updated model locally $\theta_{t+1}^{(j(l))}$ based on its own gradient and (2) pulls a model $\theta_{t+1}^{(i)}$ from a parameter server i. If server i is correct then the growth of the pulled model $\theta_{t+1}^{(i)}$ (with respect to the local gradient $g_t^{(j)}$) should be close to that of the estimated local model $\theta_{t+1}^{(j(l))}$, based on the guarantees given by *MDA* (see Appendix C.2.1). Such a growth rate is encapsulated in the *Lipschitz coefficient* of the loss function. If the pulled model is correct then, the worker expects that the Lipschitz coefficient computed based on that model be close to those of the other correct models received before by the worker. Concretely, a worker computes an empirical estimation of

the Lipschitz coefficient $k = \frac{\left\|g_{t+1}^{(j)} - g_t^{(j)}\right\|_2}{\left\|\theta_{t+1}^{(j(l))} - \theta_t^{(j)}\right\|_2}$ and then, ensures that it follows the condition $k \leq K_p \triangleq \text{quantile}_{\frac{n_{ps}-f_{ps}}{n_{ps}}}\{K\}$, where K is the list of all previous Lipschitz coefficients k (i.e., with $t_{prev} < t$). Note the Lipschitz filter requires $n_{ps} > 3f_{ps}$ (see Appendix C.2.3).

5.2 *Outliers* Filter

Although the Lipschitz filter can bound the model growth with respect to gradients, a server can still trick this filter by sending a well-crafted model that is arbitrarily far from the other correct models [8]. To overcome this problem, we use another filter, which we call Outliers *filter*, to bound the distance between models in any two successive steps. In short, a worker j assumes the distance between a local estimate of a model $\theta_{t+1}^{(j(l))}$ and a pulled model $\theta_{t+1}^{(i)}$ to be upper–bounded as follows: $\left\|\theta_{t+1}^{(j(l))} - \theta_{t+1}^{(i)}\right\|_2 < \eta_{T\cdot(t \bmod T)} \left\|g_{T\cdot(t \bmod T)}\right\|_2 \left(\frac{(3T+2)(n_w - f_w)}{4f_w} + 2\big((t-1) \bmod T\big)\right)$.

Sketch of proof. Each worker j at time t computes a local updated model $\theta_t^{(j(l))}$ as follows:

$$\theta_t^{(j(l))} = \theta_{t-1}^{(j)} - \eta_t g_t^{(j)}.$$

Simultaneously, a correct server i computes an update as follows:

$$\theta_t^{(i)} = \theta_{t-1}^{(i)} - \eta_t MDA\left(g_t^{(1)} \ldots g_t^{(n_w)}\right).$$

Based on the guarantees given by *MDA* [22], the following holds:

$$\left\|MDA\left(g_t^{(1)} \ldots g_t^{(n_w)}\right) - g_t^{(j)}\right\|_2 \leq \frac{n_w - f_w}{2f_w} \left\|g_t^{(j)}\right\|_2.$$

Hence, based on the Lipschitzness of the loss function, we have:

$$\left\|\theta_t^{(j(l))} - \theta_t^{(i)}\right\|_2 \leq \eta_{(T\cdot(t \bmod T))} \left\|g_{(T\cdot(t \bmod T))}\right\|_2 \cdot \left(2\left((t \bmod T) - 1\right) + \frac{(n_w - f_w)(3T+2)}{4f_w}\right) \quad (6)$$

□

Such a bound is also based on the *scatter*/*gather* scheme we are using. The details of deriving this term is in Appendix C.2.3.

5.3 ByzSGD: The Synchronous Version

Keeping the parameter server algorithm as is (Algorithm 2), Algorithm 3 presents the workers' training loop in the synchronous case. We focus here on the changes in the ByzSGD algorithm, compared to the asynchronous case (Section 3.3).

In the initialization phase, each worker j chooses a random integer r_j with $1 \leq r_j \leq n_{ps}$ before doing one backpropagation computation to estimate the gradient at the initial model.

While parameter servers are updating the model (line 7 in Algorithm 2), each worker j speculates the updated model by computing a local view of it, using its local computed gradient and its latest local model (line 7 in Algorithm 3). Then, each worker j pulls *one* parameter vector from server i where, $i = (r_j + t + 1) \bmod n_{ps}$. Such a worker computes the new gradient, using backpropagation, based on the pulled model. Based on this computation and the local estimate of the updated model, the worker applies the *Lipschitz* and the *Outliers* filters to check the legitimacy of the pulled model. If the model fails to pass the filters, the worker j pulls a new model from the parameter server i_{++}, where $i_{++} = (r_j + t + 2) \bmod n_{ps}$. This process is repeated until a pulled model passes both filters. Every T steps (i.e., in the *gather* phase), each worker j pulls models from *all* servers and aggregates them using *Median*, completing the DMC computation.

Algorithm 3 ByzSGD: worker logic (synchronous)

```
1:  Calculate the values of T & seed
2:  model ← init_model(seed)
3:  r ← random_int(1,n_ps)
4:  t ← 0
5:  grad ← model.backprop()
6:  repeat
7:    local_model ← apply_grad(model,grad)
8:    if t mod T = 0 then
9:      models ← read_models()
10:     model ← MeaMed(models)
11:   else
12:     i ← 0
13:     repeat
14:       new_model ← read_model((r+t+i) mod n_ps)
15:       new_grad ← new_model.backprop()
16:       i ← i + 1
17:     until pass_filters(new_model)
18:     model ← new_model
19:     grad ← new_grad
20:   end if
21:   t ← t + 1
22: until t > max_steps
```

Table 2: Models used throughout the evaluation.

NN architecture	# parameters	Size (MB)
MNIST_CNN	79510	0.3
CifarNet	1756426	6.7
Inception	5602874	21.4
ResNet-50	23539850	89.8
ResNet-200	62697610	239.2

6 EXPERIMENTAL EVALUATION

We implemented our algorithms on top of TensorFlow [3], and we report here on some of our empirical results. Appendix E presents additional experiments, highlighting the effect of changing n_{ps} and assessing the efficiency of our filtering mechanism.

6.1 Evaluation Setting

Testbed. We use Grid5000 [2] as an experimental platform. We employ up to 20 worker nodes and up to 6 parameter servers. Each node has 2 CPUs (Intel Xeon E5-2630 v4) with 10 cores, 256 GiB RAM, and 2×10 Gbps Ethernet.

Experiments. We consider an image classification task due to its wide adoption as a benchmark for distributed ML systems, e.g., [17]. We use MNIST [34] and CIFAR-10 [1] datasets. MNIST consists of handwritten digits. It has 70,000 28×28 images in 10 classes. CIFAR-10 is a widely–used dataset in image classification [45, 52]. It consists of 60,000 32×32 colour images in 10 classes.

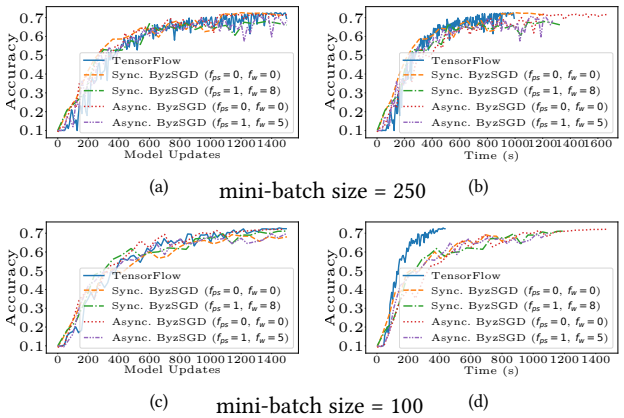

Figure 3: Convergence in a non-Byzantine environment.

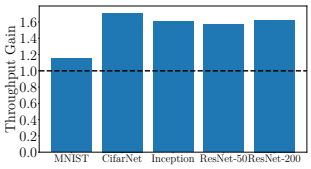

Figure 4: Throughput gain: sync. relative to async.

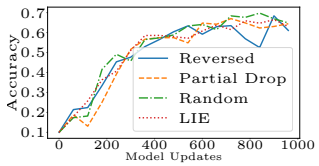

Figure 5: Convergence with one Byzantine server.

We employ several NN architectures with different sizes ranging from a small convolutional neural network (CNN) for MNIST, training <100k parameters, to big architectures like ResNet-200 with around 63M parameters (see Table 2). We use *CIFAR10* (as a dataset) and *CifarNet* (as a model) as our default experiment.

Metrics. We evaluate the performance of ByzSGD using the following standard metrics. *1. Accuracy (top–1 cross–accuracy).* The fraction of correct predictions among all predictions, using the *test* dataset. We measure accuracy with respect to time and model updates. *2. Throughput.* The total number of updates that the deployed system can do per second.

Baseline. We consider vanilla TensorFlow (vanilla TF) as our baseline. Given that such a baseline does not converge in Byzantine environments [18], we use it only to quantify the overhead in non–Byzantine environments.

6.2 Evaluation Results

First, we show ByzSGD's performance, highlighting the overhead, in a non–Byzantine environment. Then, we compare the throughput of the synchronous variant to that of the asynchronous variant in a Byzantine–free environment. Finally, we report on the performance of ByzSGD in a Byzantine environment, i.e., with Byzantine workers and Byzantine servers. For the Byzantine workers, we show the effect of a recent attack [8] on ByzSGD, and then we show the results of 4 different attacks in the case of Byzantine servers. In all experiments, and based on our setup, we use $T = 333$. We discuss the effect of changing T on ByzSGD's performance in Appendix E.2.

Non–Byzantine environment. Figure 3 shows the convergence (i.e., progress of accuracy) of all experimented systems with both time and model updates (i.e., training steps). We experiment

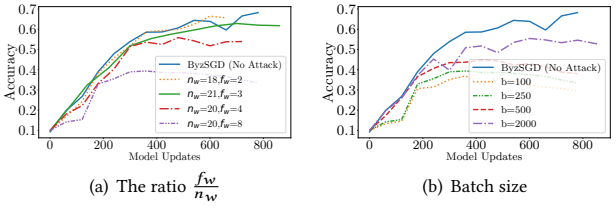

Figure 6: Convergence in the presence of Byzantine workers.

with two batch sizes and different values for declared Byzantine servers and workers (only for the Byzantine–tolerant deployments). Figure 3(a) shows that all deployments have almost the same convergence behavior, with a slight loss in final accuracy for the Byzantine–tolerant deployments, which we quantify to around 5%. Such a loss is emphasized with the smaller batch size (Figure 3(c)). This accuracy loss is admitted in previous work [50] and is inherited from using statistical methods (basically, *MDA* in our case) for Byzantine resilience. In particular, *MDA* ensures convergence only to a ball around the optimal solution, i.e., local minimum [22]. Moreover, the figures confirm that using a higher batch size gives a more stable convergence for ByzSGD. Figures 3(a) and 3(c) show that both variants of ByzSGD almost achieve the same convergence.

The cost of Byzantine resilience is more clear when convergence is observed over time (Figure 3(b)), especially with the lower batch size (Figure 3(d)). We define the convergence overhead by the ratio of the time taken by ByzSGD to reach some accuracy level compared to that taken by TensorFlow to reach the same accuracy level. For example, in Figure 3(b), TensorFlow reaches 60% accuracy in 268 seconds which is around 32% better than the slowest deployment of ByzSGD. We draw two main observations from these figures. First, changing the number of declared Byzantine machines affects the progress of accuracy, especially with the asynchronous deployment of ByzSGD. This is because servers and workers in such case wait for replies from only $n - f$ machines. Hence, decreasing f forces the receiver to wait for more replies, slowing down convergence. Second, the synchronous variant always outperforms the asynchronous one, especially with non–zero values for declared Byzantine machines, be they servers and workers. Such a result is expected as the synchronous algorithm uses less number of messages per round compared to the asynchronous one. Given that distributed ML systems are network–bound [29, 52], reducing the communication overhead significantly boosts the performance (measured by convergence speed in this case) and the scalability of such systems.

Throughput. We do the same experiment again, yet with different state–of–the–art models so as to quantify the throughput gain of the synchronous variant of ByzSGD. Figure 4 shows the throughput of synchronous ByzSGD divided by the throughput of the asynchronous ByzSGD in each case. From this figure, we see that synchrony helps ByzSGD achieve throughput boost (up to 70%) in all cases, where such a boost is emphasized more with large models. This is expected because the main advantage of synchronous ByzSGD is to decrease the number of communication messages, where bigger messages are transmitted with bigger models.

Byzantine workers. We study here the convergence of ByzSGD in the presence of Byzantine workers. Simple misbehavior like

message drops, unresponsive machines, or reversed gradients are well-studied and have been shown to be tolerated by Byzantine–resilient GARs, e.g., [50], which ByzSGD also uses. Thus, here we focus on a more recent attack that is coined as *A little is enough attack* [8]. This attack focuses on changing each dimension in gradients of Byzantine workers to trick some of Byzantine–resilient GARs, e.g., [12, 22].

We apply this attack to multiple deployments of ByzSGD. In each scenario, we apply the strongest possible change in gradients' coordinates so as to hamper the convergence the most. We study the effect of this attack on the convergence of ByzSGD with both the ratio of Byzantine workers to the total number of workers (Figure 6(a)) and the batch size (Figure 6(b)). We use the deployment with no Byzantine behavior (*No Attack*) as a baseline.

Figure 6(a) shows that the effect of the attack starts to appear clearly when the number of Byzantine workers is a significant fraction (more than 20%) of the total number of workers. This is intuitive as the attack tries to increase the variance between the submitted gradients to the parameter servers and hence, increases the ball (around the local minimum) to which the used GAR converges (see e.g., [12, 22] for a theoretical analysis of the interplay between the variance and the Byzantine resilience). Stretching the number of Byzantine workers to the maximum ($f_w = 8$) downgrades the accuracy to around 40% (compared to 67% in "No Attack" case). This can be explained by the large variance between honest gradients, above what *MDA* requires, as we discuss in Appendix D.

Increasing the batch size not only improves the accuracy per training step but also the robustness of ByzSGD (by narrowing down the radius of the ball around the convergence point, where the model will fluctuate as proven in [12, 22]). Figure 6(b) fixes the ratio of f_w to n_w to the biggest allowed value to see the effect of using a bigger batch size on the convergence behavior. This figure confirms that increasing the batch size increases the robustness of ByzSGD. Moreover, based on our experiments, setting 25% of workers to be Byzantine while using a batch size of (up to) 256 does not experimentally satisfy the assumption on the variance of *MDA* in this deployment, which leads to a lower accuracy after convergence (see Appendix D).

Byzantine servers. Figure 5 shows the convergence of ByzSGD in the presence of 1 Byzantine server. We experimented with 4 different adversarial behaviors: **1.** *Reversed:* the server sends a correct model multiplied by a negative number, **2.** *Partial Drop:* the server randomly chooses 10% of the weights and set them to zero (this simulates using unreliable transport protocol in the communication layer, which was proven beneficial in some cases, e.g., [18]), **3.** *Random:* the server replaces the learned weights by random numbers, and **4.** *LIE*, an attack inspired from the *little is enough* attack [8], in which the server multiplies each of the individual weights by a small number z, where $|z - 1| < \delta$ with δ close to zero; $z = 1.035$ in our experiments. Such a figure shows that ByzSGD can tolerate the experimented Byzantine behavior and guarantee the learning convergence to a high accuracy.

7 CONCLUDING REMARKS

Summary. This paper is a first step towards genuinely distributed Byzantine–resilient Machine Learning (ML) solutions that do not trust any network component. We present ByzSGD that guarantees learning convergence despite the presence of Byzantine machines. Through the introduction of a series of novel ideas, the *Scatter/Gather* protocol, the *Distributed Median-based Contraction (DMC)* module, and the filtering mechanisms, we show that ByzSGD works in an asynchronous setting, and we show how we can leverage synchrony to boost performance. We built ByzSGD on top of TensorFlow, and we show that it tolerates Byzantine behavior with 32% overhead compared to vanilla TensorFlow.

Related work. With the impracticality (and sometimes impossibility [26]) of applying exact consensus to ML applications, the approximate consensus [25] seems to be a good candidate. In approximate consensus, all nodes try to decide values that are arbitrarily close to each other and that are within the range of values proposed by correct nodes. Several approximate consensus algorithms were proposed with different convergence rates, communication/computation costs, and supported number of tolerable Byzantine nodes, e.g., [4, 21, 24, 37].

Inspired by approximate consensus, several Byzantine–resilient ML algorithms were proposed yet, all assumed a single correct parameter server: only workers could be Byzantine. Three Median-based aggregation rules were proposed to resist Byzantine attacks [50]. Krum [12] and Multi-Krum [18] used a distance-based algorithm to eliminate Byzantine inputs and average the correct ones. Bulyan [22] proposed a meta-algorithm to guarantee Byzantine resilience against a strong adversary that can fool the aforementioned aggregation rules. Draco [16] used coding schemes and redundant gradient computation for Byzantine resilience, where Detox [40] combined coding schemes with Byzantine–resilient aggregation for better resilience and overhead guarantees. Kardam [19] is the only proposal to tolerate Byzantine workers in asynchronous learning setup. ByzSGD augments these efforts by tolerating Byzantine servers in addition to Byzantine workers.

Open questions. This paper opens interesting questions.

First, the relation between the frequency of entering the *gather* phase (i.e., the value of T) and the variance between models on correct servers is both data and model dependent. In our analysis, we provide safety guarantees on this relation that always ensure Byzantine resilience and convergence. However, we believe that in some cases, entering the *gather* phase more frequently may lead to a noticeable improvement in the convergence speed (see Appendix E.2). The trade-off between this gain and the corresponding communication overhead is an interesting open question.

Second, the Lipschitz filter requires $n_{ps} > 3f_{ps}$ (see Appendix C.2.3). There is another tradeoff here between the communication overhead and the required number of parameter servers. One can use Byzantine–resilient aggregation of models, which requires only $n_{ps} > 2f_{ps}$, yet requires communicating with *all* servers in each step. In our design, we strive for reducing the communication overhead, given that communication is the bottleneck [29, 52].

Third, it is interesting to explore similar Byzantine–resilient solutions in the context of decentralized settings, i.e., in which all machines are considered as workers with no central servers and also with non-iid data. Although it seems that we can directly apply the same algorithms presented in this paper to such settings, designing better algorithms with low communication overhead with provable resilience guarantees remains an open question.

ACKNOWLEDGMENTS

This work has been supported in part by the Swiss National Science Foundation (FNS grant 200021_182542/1).

Most experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

REFERENCES

[1] [n.d.]. Cifar dataset. https://www.cs.toronto.edu/~kriz/cifar.html.
[2] [n.d.]. Grid5000. https://www.grid5000.fr/.
[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 265–283.
[4] Ittai Abraham, Yonatan Amit, and Danny Dolev. 2004. Optimal resilience asynchronous approximate agreement. In *International Conference On Principles Of Distributed Systems*. Springer, 229–239.
[5] Dan Alistarh, Zeyuan Allen-Zhu, and Jerry Li. 2018. Byzantine stochastic gradient descent. In *Advances in Neural Information Processing Systems*. 4613–4623.
[6] Dan Alistarh, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2016. QSGD: Randomized Quantization for Communication-Optimal Stochastic Gradient Descent. *arXiv preprint arXiv:1610.02132* (2016).
[7] Peter L Bartlett, Dylan J Foster, and Matus J Telgarsky. 2017. Spectrally-normalized margin bounds for neural networks. In *Neural Information Processing Systems*. 6241–6250.
[8] Moran Baruch, Gilad Baruch, and Yoav Goldberg. 2019. A Little Is Enough: Circumventing Defenses For Distributed Learning. *arXiv preprint arXiv:1902.06156* (2019).
[9] Jeremy Bernstein, Jiawei Zhao, Kamyar Azizzadenesheli, and Anima Anandkumar. 2018. signSGD with majority vote is communication efficient and fault tolerant. *arXiv preprint arXiv:1810.05291* (2018).
[10] Battista Biggio, Blaine Nelson, and Pavel Laskov. 2012. Poisoning attacks against support vector machines. *arXiv preprint arXiv:1206.6389* (2012).
[11] Battista Biggio and Fabio Roli. 2017. Wild Patterns: Ten Years After the Rise of Adversarial Machine Learning. *arXiv preprint arXiv:1712.03141* (2017).
[12] Peva Blanchard, El Mahdi El Mhamdi, Rachid Guerraoui, and Julien Stainer. 2017. Machine Learning with Adversaries: Byzantine Tolerant Gradient Descent. In *Neural Information Processing Systems*. 118–128.
[13] Léon Bottou. 1998. Online learning and stochastic approximations. *Online learning in neural networks* 17, 9 (1998), 142.
[14] Christian Cachin, Rachid Guerraoui, and Luis Rodrigues. 2011. *Introduction to reliable and secure distributed programming*.
[15] Miguel Castro, Barbara Liskov, et al. 1999. Practical Byzantine fault tolerance. In *OSDI*, Vol. 99. 173–186.
[16] Lingjiao Chen, Hongyi Wang, Zachary Charles, and Dimitris Papailiopoulos. 2018. DRACO: Byzantine-resilient Distributed Training via Redundant Gradients. In *International Conference on Machine Learning*. 902–911.
[17] Trishul M Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project Adam: Building an Efficient and Scalable Deep Learning Training System.. In *OSDI*, Vol. 14. 571–582.
[18] Georgios Damaskinos, El Mahdi El Mhamdi, Rachid Guerraoui, Arsany Guirguis, and Sébastien Rouault. 2019. AggregaThor: Byzantine Machine Learning via Robust Gradient Aggregation. In *SysML*.
[19] Georgios Damaskinos, El Mahdi El Mhamdi, Rachid Guerraoui, Rhicheek Patra, and Mahsa Taziki. 2018. Asynchronous Byzantine Machine Learning (the case of SGD). In *ICML*. 1153–1162.
[20] Ilias Diakonikolas, Gautam Kamath, Daniel M Kane, Jerry Li, Ankur Moitra, and Alistair Stewart. 2018. Robustly learning a gaussian: Getting optimal error, efficiently. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. 2683–2702.
[21] Danny Dolev, Nancy A Lynch, Shlomit S Pinter, Eugene W Stark, and William E Weihl. 1986. Reaching approximate agreement in the presence of faults. *Journal of the ACM (JACM)* 33, 3 (1986), 499–516.
[22] El Mahdi El Mhamdi, Rachid Guerraoui, and Sébastien Rouault. 2018. The Hidden Vulnerability of Distributed Learning in Byzantium. In *International Conference on Machine Learning*. 3521–3530.
[23] Andre Esteva, Brett Kuprel, Roberto A Novoa, Justin Ko, Susan M Swetter, Helen M Blau, and Sebastian Thrun. 2017. Dermatologist-level classification of skin cancer with deep neural networks. *Nature* 542, 7639 (2017), 115.
[24] AD Fekete. 1986. Asymptotically optimal algorithms for approximate agreement. In *Proceedings of the fifth annual ACM symposium on Principles of distributed computing*. 73–87.
[25] Alan David Fekete. 1987. Asynchronous approximate agreement. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*. 64–76.
[26] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. 1985. Impossibility of distributed consensus with one faulty process. *JACM* 32, 2 (1985), 374–382.
[27] Justin Gilmer, Luke Metz, Fartash Faghri, et al. 2018. Adversarial Spheres. *arXiv preprint arXiv:1801.02774* (2018).
[28] Robert Hecht-Nielsen. 1992. Theory of the backpropagation neural network. In *Neural networks for perception*. Elsevier, 65–93.
[29] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, et al. 2017. Gaia: Geo-Distributed Machine Learning Approaching LAN Speeds.. In *NSDI*. 629–647.
[30] Larry Kim. 2012. How many ads does Google serve in a day? http://goo.gl/oIidXO (November 2012).
[31] Jakub Konečnỳ, Brendan McMahan, and Daniel Ramage. 2015. Federated optimization: Distributed optimization beyond the datacenter. *arXiv preprint arXiv:1511.03575* (2015).
[32] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine generals problem. *TOPLAS* 4, 3 (1982), 382–401.
[33] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436–444.
[34] Yann Lecunn. 1998. MNIST dataset. http://yann.lecun.com/exdb/mnist/.
[35] Mu Li, David G Andersen, Jun Woo Park, et al. 2014. Scaling distributed machine learning with the parameter server. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 583–598.
[36] Mu Li, Li Zhou, Zichao Yang, et al. 2013. Parameter server for distributed machine learning. In *Big Learning NIPS Workshop*, Vol. 6. 2.
[37] Hammurabi Mendes, Maurice Herlihy, Nitin H. Vaidya, and Vijay K. Garg. 2015. Multidimensional agreement in Byzantine systems. *Distributed Computing* 28, 6 (2015), 423–441. https://doi.org/10.1007/s00446-014-0240-5
[38] Xiangrui Meng, Joseph Bradley, Burak Yavuz, et al. 2016. Mllib: Machine learning in apache spark. *JMLR* 17, 1 (2016), 1235–1241.
[39] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. 2017. Practical black-box attacks against machine learning. In *Asia Conference on Computer and Communications Security*. 506–519.
[40] Shashank Rajput, Hongyi Wang, Zachary Charles, and Dimitris Papailiopoulos. 2019. DETOX: A Redundancy-based Framework for Faster and More Robust Gradient Aggregation. *arXiv preprint arXiv:1907.12205* (2019).
[41] Lorenzo Rosasco, Ernesto De Vito, Andrea Caponnetto, Michele Piana, and Alessandro Verri. 2004. Are loss functions all the same? *Neural Computation* 16, 5 (2004), 1063–1076.
[42] Peter J Rousseeuw. 1985. Multivariate estimation with high breakdown point. *Mathematical statistics and applications* 8 (1985), 283–297.
[43] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1986. Learning representations by back-propagating errors. *nature* 323, 6088 (1986), 533–536.
[44] Fred B Schneider. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *CSUR* 22, 4 (1990), 299–319.
[45] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A simple way to prevent neural networks from overfitting. *JMLR* 15, 1 (2014), 1929–1958.
[46] Lili Su and Shahin Shahrampour. 2018. Finite-time Guarantees for Byzantine-Resilient Distributed State Estimation with Noisy Measurements. *arXiv preprint arXiv:1810.10086* (2018).
[47] Aladin Virmaux and Kevin Scaman. 2018. Lipschitz regularity of deep neural networks: analysis and efficient estimation. In *Advances in Neural Information Processing Systems*. 3835–3844.
[48] Pooja Vyavahare, Lili Su, and Nitin H Vaidya. 2019. Distributed Learning with Adversarial Agents Under Relaxed Network Condition. *arXiv preprint arXiv:1901.01943* (2019).
[49] Huang Xiao, Battista Biggio, Gavin Brown, Giorgio Fumera, Claudia Eckert, and Fabio Roli. 2015. Is feature selection secure against training data poisoning?. In *ICML*. 1689–1698.
[50] Cong Xie, Oluwasanmi Koyejo, and Indranil Gupta. 2018. Generalized Byzantine-tolerant SGD. *arXiv preprint arXiv:1802.10116* (2018).
[51] Cong Xie, Oluwasanmi Koyejo, and Indranil Gupta. 2018. Zeno: Byzantine-suspicious stochastic gradient descent. *arXiv preprint arXiv:1805.10032* (2018).
[52] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P. Xing. 2017. Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on GPU Clusters. In *USENIX ATC*. 181–193.
[53] Sixin Zhang, Anna E Choromanska, and Yann LeCun. 2015. Deep learning with elastic averaging SGD. In *NIPS*. 685–693.