BASGD: Buffered Asynchronous SGD for Byzantine Learning

Yi-Rui Yang YANGYR@LAMDA.NJU.EDU.CN
Wu-Jun Li LIWUJUN@NJU.EDU.CN
Department of Computer Science and Technology
Nanjing University, China

arXiv:2003.00937v2 [cs.LG] 3 Mar 2020

Abstract

Distributed learning has become a hot research topic, due to its wide application in cluster-based large-scale learning, federated learning, edge computing and so on. Most distributed learning methods assume no error and attack on the workers. However, many unexpected cases, such as communication error and even malicious attack, may happen in real applications. Hence, Byzantine learning (BL), which refers to distributed learning with attack or error, has recently attracted much attention. Most existing BL methods are synchronous, which will result in slow convergence when there exist heterogeneous workers. Furthermore, in some applications like federated learning and edge computing, synchronization cannot even be performed most of the time due to the offline workers (clients or edge servers). Hence, asynchronous BL (ABL) is more general and practical than synchronous BL (SBL). To the best of our knowledge, there exist only two ABL methods. One of them cannot resist malicious attack. The other needs to store some training instances on the server, which has the privacy leak problem. In this paper, we propose a novel method, called buffered asynchronous stochastic gradient descent (BASGD), for BL. BASGD is an asynchronous method. Furthermore, BASGD has no need to store any training instances on the server, and hence can preserve privacy in ABL. BASGD is theoretically proved to have the ability of resisting against error and malicous attack. Moreover, BASGD has a similar theoretical convergence rate to that of vanilla asynchronous SGD (ASGD), with an extra constant variance. Empirical results show that BASGD can significantly outperform vanilla ASGD and other ABL baselines, when there exists error or attack on workers.

1. Introduction

Due to the wide application in cluster-based large-scale learning, federated learning (Kone**v**cnỳ et al., 2016; Kairouz et al., 2019), edge computing (Shi et al., 2016) and so on, distributed learning has recently become a hot research topic (Zinkevich et al., 2010; Yang, 2013; Jaggi et al., 2014; Shamir et al., 2014; Zhang and Kwok, 2014; Ma et al., 2015; Lee et al., 2017; Lian et al., 2017; Zhao et al., 2017; Sun et al., 2018; Wangni et al., 2018; Zhao et al., 2018; Zhou et al., 2018; Yu et al., 2019a,b). Most existing distributed learning methods are based on stochastic gradient descent (SGD) and its variants (Bottou, 2010; Xiao, 2010; Duchi et al., 2011; Johnson and Zhang, 2013; Shalev-Shwartz and Zhang, 2013; Zhang et al., 2013; Lin et al., 2014; Schmidt et al., 2017; Zhao et al., 2018; Yu et al., 2019b). Furthermore, most existing distributed learning methods assume no error and attack on the workers.

However, in real distributed learning applications with multiple networked machines (nodes), different kinds of hardware or software errors may happen. Representative errors include bit-flipping in the communication media and the memory of some workers (Xie et al., 2019b). In this case, a small error on some machines (workers) might cause a distributed learning method to fail. In addition, malicious attack should not be neglected in an open network where the manager (or server) generally has not much control on the workers, such as the cases of edge computing and federated learning. Some malicious workers may behave arbitrarily or even adversarially. Hence, *Byzantine learning* (BL), which refers to distributed learning with attack or error, has recently attracted much attention (Blanchard et al., 2017; Alistarh et al., 2018; Damaskinos et al., 2018; Xie et al., 2019b).

Existing BL methods can be divided into two main categories: synchronous BL (SBL) methods and asynchronous BL (ABL) methods. In SBL methods, the learning information, such as the gradient in SGD, of all workers will be aggregated in a synchronous way. On the contrary, in ABL methods the learning information of workers will be aggregated in an asynchronous way. Existing SBL methods mainly take two different ways to achieve resilience against *Byzantine workers* which refer to those workers with attack or error. One way is to replace the simple averaging aggregation operation with some more robust aggregation operations, such as median, trimmed-mean (Yin et al., 2018) and Krum (Blanchard et al., 2017). The other way is to filter the suspicious learning information (gradients) before averaging. Representative examples include ByzantineSGD (Alistarh et al., 2018) and Zeno (Xie et al., 2019b). The advantage of SBL methods is that they are relatively simple and easy to be implemented. But SBL methods will result in slow convergence when there exist heterogeneous workers. Furthermore, in some applications like federated learning and edge computing, synchronization cannot even be performed most of the time due to the offline workers (clients or edge servers). Hence, ABL is more general and practical than SBL.

To the best of our knowledge, there exist only two ABL methods: Kardam (Damaskinos et al., 2018) and Zeno++ (Xie et al., 2019a). Kardam introduces two filters to drop out suspicious learning information (gradients), which can still achieve good performance when the communication delay is heavy. However, when in face of malicious attack, some work (Xie et al., 2019a) finds that Kardam also drops out most correct gradients in order to filter all faulty (error) gradients. Hence, Kardam cannot resist malicious attack. Zeno++ scores each received gradient, and determines whether to accept it according to the score. But Zeno++ needs to store some training instances on the server for scoring. In practical applications, storing data on the server will increase the risk of privacy leak or even face legal risk. Therefore, under the general setting where the server has no access to any training instances, there have not existed ABL methods to resist malicious attack.

In this paper, we propose a novel method, called buffered asynchronous stochastic gradient descent (BASGD), for BL. The main contributions of BASGD are listed as follows:

- BASGD is an asynchronous method, and hence BASGD is more general and practical than existing SBL methods.
- BASGD has no need to store any training instances on the server, and hence can preserve privacy in ABL.
- BASGD is theoretically proved to have the ability of resisting against error and malicious attack.

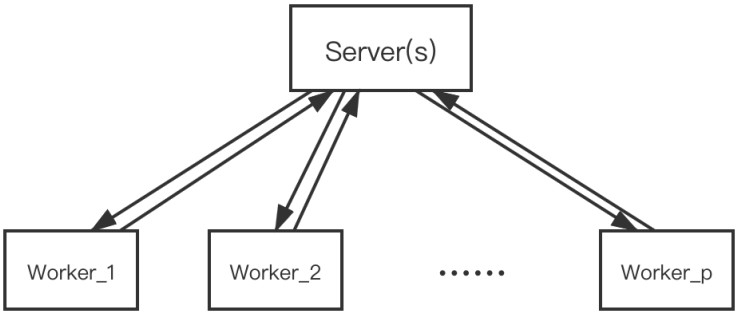


Figure 1: Parameter Server framework

- BASGD has a similar theoretical convergence rate to that of vanilla asynchronous SGD (ASGD), with an extra constant variance.
- Empirical results show that BASGD can significantly outperform vanilla ASGD and other ABL baselines when there exist error or malicious attack on workers. In particular, BASGD can still converge under cases with malicious attack in which ASGD and other ABL methods fail.

2. Preliminary

This section presents the preliminary of this paper, including the distributed learning framework used in this paper and the definition of Byzantine worker.

2.1 Distributed Learning Framework

Many machine learning models, such as logistic regression and deep neural networks, can be formulated as the following finite sum optimization problem:

$$\min_{\mathbf{w}\in\mathbb{R}^d} F(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} f(\mathbf{w}; z_i), \tag{1}$$

where $\mathbf{w}$ is the parameter to learn, d is the dimension of parameter, n is the number of training instances, $f(\mathbf{w}; z_i)$ is the empirical loss on the training instance z_i. The goal of this work is to solve the optimization problem in (1), by designing distributed learning algorithms on multiple networked machines.

Although there have appeared many distributed learning frameworks, in this paper we focus on the widely used Parameter Server (PS) framework (Li et al., 2014) which is shown in Figure 1. In a PS framework, there are several workers and one server or multiple servers. Each worker can only communicate with the server(s). There may exist more than one server in a PS framework, but for the problem of this paper the servers can be logically conceived as a unity. Without loss of generality, we will assume there is only one server in this paper.

Algorithm 1 Asynchronous SGD (ASGD)

Server:
Initialization: initial parameter $\mathbf{w}^0$, learning rate η;
Send initial $\mathbf{w}^0$ to all workers;
for $t = 0$ **to** $t_{max} - 1$ **do**
  Wait until $\mathbf{g}_{k_t}^t$ is received from some worker_k_t;
  Execute SGD step: $\mathbf{w}^{t+1} \leftarrow \mathbf{w}^t - \eta \cdot \mathbf{g}_{k_t}^t$;
  Send $\mathbf{w}^{t+1}$ back to worker_k_t;
end for
Notify all workers to stop;

Worker_k**:** $(k = 0, 1, ..., m-1)$
repeat
  Wait until receiving the latest parameter $\mathbf{w}$ from server;
  Randomly sample an index i from $\mathcal{D}_k$;
  Compute $\nabla f(\mathbf{w}; z_i)$;
  Send $\nabla f(\mathbf{w}; z_i)$ to server;
until receive server's notification to stop

Training instances are disjointedly distributed across m workers. Let $\mathcal{D}_k$ denote the index set of training instances on worker_k, we have $\cup_{k=1}^m \mathcal{D}_k = \{1, 2, \ldots, n\}$ and $\mathcal{D}_k \cap \mathcal{D}_{k'} = \emptyset$ if $k \neq k'$. In this paper, we assume that the server has no access to any training instances.

One popular asynchronous method to solve the problem in (1) under the PS framework is ASGD which is presented in Algorithm 1. Here, each worker sample one instance for gradient computation each time. Each worker can also sample a mini-batch of instances for gradient computation each time. The effect of batch size is not the focus of this work, and the analysis of this paper can also be easily adapted for cases with mini-batch. Hence, in this paper we do not separately discuss the mini-batch case.

In PS based ASGD, the server is responsible for updating and maintaining the latest parameter. We use the number of iterations that the server has already executed as the current iteration number of the server. At the very beginning, the iteration number $t = 0$. Each time a SGD step is executed, t will be increased by 1 immediately. Iteration number can also be seen as the version of parameter, and we denote the parameter after t iterations as $\mathbf{w}^t$.

The server may have executed several SGD steps between the time when worker k receives parameter $\mathbf{w}^{t_1}$ and the time when worker k sends back the gradient computed based on $\mathbf{w}^{t_1}$. We use t_2 to denote the iteration number on the server when receiving the gradient computed based on $\mathbf{w}^{t_1}$. Delay τ is defined as $\tau = t_2 - t_1$.

2.2 Byzantine Worker

Between two iterations on the server, a worker may send nothing, send gradient only once, or send gradient more than once. Though the last case is impossible in ASGD (see Algorithm 1), but it may happen in BASGD (refer to Section 3).

For workers that have sent gradients to server at iteration t, we call some worker a *loyal worker* if the worker has finished all the tasks without any fault and each sent gradient is correctly received by the server with delay $\tau \leq \tau_{max}$, where τ_{max} is a constant. Otherwise, worker_k is called a *Byzantine worker*. If worker_k is a Byzantine worker, it means the received gradient from worker_k is not credible, which may be an arbitrary value.

We use $\mathcal{G}^t$ to denote the index set of workers that the server has received gradient from at iteration t, namely, between the t-th SGD step and the $(t+1)$-th SGD step. $\forall k \in \mathcal{G}^t$, we denote the gradient received from worker_k at iteration t as $\mathbf{g}_k^t$.

Please note that a worker may not be always loyal or always Byzantine. For example, a loyal worker at iteration t_1 may suffer from a bit-flipping at iteration t_2, so it will be identified as a Byzantine worker at iteration t_2. Also, a malicious worker may sometimes behave as loyal ones to hide itself, and will be seen as loyal at these normally working iterations.

Furthermore, we define the index set of loyal workers at iteration t as follows:

$$\mathcal{L}^t = \{k \in \mathcal{G}^t \mid \exists t' \in [t - \tau_{max},\ t],\ \mathbf{g}_k^t = \nabla f(\mathbf{w}^{t'}; z_i),\ i \text{ is uniformly sampled from } \mathcal{D}_k\}.$$

Thus, worker_k is Byzantine at iteration t if $k \in \mathcal{G}^t \setminus \mathcal{L}^t$. Then, we have:

$$\mathbf{g}_k^t = \begin{cases} \nabla f(\mathbf{w}^{t'}; z_i), & \text{if } k \in \mathcal{L}^t; \\ \textit{arbitrary value}, & \text{if } k \in \mathcal{G}^t \setminus \mathcal{L}^t, \end{cases}$$

where $0 \leq t - t' \leq \tau_{max}$, and i is randomly sampled from $\mathcal{D}_k$.

Our definition of Byzantine worker and loyal worker is consistent with most previous works (Blanchard et al., 2017; Xie et al., 2019b) under the setting of synchronous Byzantine learning which actually corresponds to the case $\tau_{max} = 0$. But our definition is more general since it includes the cases with time delays, i.e., $\tau_{max} \geq 0$, which cannot be neglected in an asynchronous method. In particular, there are mainly two types of Byzantine workers in ABL:

- *Workers with malicious attack*: This type of workers are controlled or hacked by an adversarial party. They may send wrong or malicious gradients to the server on purpose, and try to make learning method fail. This type of workers can be appeared in some applications with open networks, such as edge computing and federated learning, where the manager (or server) generally has not much control on the workers.

- *Workers with accidental error*: Although not necessarily malicious, this type of workers may go wrong during the learning process, due to accidental errors such as bit flipping and network failure. For cases with this type of workers, the gradient received by the server might be too stale or wrongly transmitted. Although unintentionally, the stale or faulty (error) gradients will slow down the convergence or even cause learning methods to fail.

3. Buffered Asynchronous SGD

In synchronous BL, all gradients are received at the same time for updating parameters. During this process, we can compare the gradients with each other, and then filter suspicious

ones, or use more robust aggregation rules such as median and trimmed-mean for updating. However, in asynchronous BL, only one gradient is received by the server at a time. Without any training instances stored on the server, it is difficult for the server to identify whether a received gradient is credible or not.

In order to deal with this problem in asynchronous BL, we propose a novel ABL method called buffered asynchronous SGD (BASGD). BASGD introduces B buffers ($0 < B \leq m$) on the server, and the gradient used for updating parameters will be aggregated from these buffers. The learning procedure of BASGD is presented in Algorithm 2. We can find that BASGD degenerates to vanilla ASGD when buffer number $B = 1$.

In the following content of this section, we will introduce the details of the two key components of BASGD: buffer and aggregation function.

3.1 Buffer

In BASGD, the workers do the same job as that in ASGD, while the updating rule on server is modified. More specifically, there are B buffers ($0 < B \leq m$) on the server. When the server receives a gradient $\mathbf{g}$ from worker_s, the parameter will not be updated immediately. The gradient will be stored in a buffer b temporarily, where $b = s \ mod \ B$. A concrete example is illustrated in Figure 2. Only when all buffers have got changed since the last SGD step, a new SGD step will be executed.

For each buffer b, more than one gradient may have been received between two iterations. We will store the average of these gradients, denoted by $\mathbf{h}_b$, in buffer b. Assume that there are already $(N_b - 1)$ gradients $\mathbf{g}_1, \mathbf{g}_2, \ldots, \mathbf{g}_{N_b-1}$ which should be stored in buffer b, and

$$\mathbf{h}_{b(old)} = \frac{1}{N_b - 1} \sum_{i=1}^{N_b-1} \mathbf{g}_i.$$

When the N_b-th gradient $\mathbf{g}_{N_b}$ is received, the new average value in buffer b should be:

$$\mathbf{h}_{b(new)} = \frac{1}{N_b} \sum_{i=1}^{N_b} \mathbf{g}_i = \frac{N_b - 1}{N_b} \cdot \mathbf{h}_{b(old)} + \frac{1}{N_b} \cdot \mathbf{g}_{N_b}.$$

This is the updating rule for each buffer b when a gradient is received. After the parameter $\mathbf{w}$ is updated, all buffers will be zeroed out at once.

With the benefit of buffers, the server has access to B candidate gradients when updating parameter. Thus, a more reliable (robust) gradient can be aggregated from the B gradients of buffers, if a proper aggregation function $Aggr(\cdot)$ is chosen.

3.2 Aggregation Function

When a SGD step is ready to be executed, there are B buffers providing candidate gradients. An aggregation function is needed to get the final gradient for updating. A simple function is to take the mean of all candidate gradients. However, mean value is sensitive to outliers which are common in BL.

For designing proper aggregation functions, we first define the q-Byzantine Robust (q-BR) condition to quantitatively describe the Byzantine resilience ability of an aggregation function.

Algorithm 2 Buffered Asynchronous SGD (BASGD)

Server:
Input: learning rate η, buffer number B,
aggregation function: $Aggr(\cdot)$;
Initialization: initial parameter $\mathbf{w}^0$, learning rate η;
Send initial $\mathbf{w}^0$ to all workers;
Set $t \leftarrow 0$;
Set buffer: $\mathbf{h}_b \leftarrow \mathbf{0}$, $N_b \leftarrow 0$;
repeat
Wait until receiving $\mathbf{g}$ from some worker_s;
Choose buffer: $b \leftarrow s \ mod \ B$;
$N_b \leftarrow N_b + 1$;
$\mathbf{h}_b \leftarrow \frac{(N_b-1)\mathbf{h}_b+\mathbf{g}}{N_b}$;
if $N_b > 0$ for each $b \in [B]$ **then**
Aggregate: $\mathbf{G}^t = Aggr([\mathbf{h}_1, \ldots, \mathbf{h}_B])$;
Execute SGD step: $\mathbf{w}^{t+1} \leftarrow \mathbf{w}^t - \eta \cdot \mathbf{G}^t$;
for $b = 1$ **to** B **do**
Zero out buffer: $\mathbf{h}_b \leftarrow \mathbf{0}$, $N_b \leftarrow 0$;
end for
$t \leftarrow t + 1$;
end if
Send $\mathbf{w}^t$ back to worker_s;
until stop criterion is satisfied
Notify all workers to stop;

Worker_k**:** $(k = 0, 1, ..., m-1)$
repeat
Wait until receiving the latest parameter $\mathbf{w}$ from server;
Randomly sample an index i from $\mathcal{D}_k$;
Compute $\nabla f(\mathbf{w}; z_i)$;
Send $\nabla f(\mathbf{w}; z_i)$ to server;
until receive server's notification to stop

Definition 1 (q-Byzantine Robust)**.** *For an aggregation function $Aggr(\cdot)$: $Aggr([\mathbf{h}_1, \ldots, \mathbf{h}_B]) = \mathbf{G}$, where $\mathbf{G} = [G_1, \ldots, G_d]^T$ and $\mathbf{h}_b = [h_{b1}, \ldots, h_{bd}]^T, \forall b \in [B]$, we call $Aggr(\cdot)$* q-Byzantine Robust *($q \in \mathbb{Z}, 0 < q < B/2$), if it satisfies the following two properties:*

a). $\forall \mathbf{h}_1, \ldots, \mathbf{h}_B \in \mathbb{R}^d, \forall \mathbf{h}' \in \mathbb{R}^d$,

$$Aggr([\mathbf{h}_1 + \mathbf{h}', \ldots, \mathbf{h}_B + \mathbf{h}']) = Aggr([\mathbf{h}_1, \ldots, \mathbf{h}_B]) + \mathbf{h}';$$

b). $\forall j \in [d]$, $\forall \mathcal{S} \subset [B]$ *with* $|\mathcal{S}| = B - q$,

$$\min_{s \in \mathcal{S}} \{h_{sj}\} \leq G_j \leq \max_{s \in \mathcal{S}} \{h_{sj}\}.$$

Intuitively, property a) in Definition 1 says that if all candidate gradients $\mathbf{h}_i$ are added by a same vector $\mathbf{h}'$, the aggregated gradient will also be added by $\mathbf{h}'$. Property b) says that

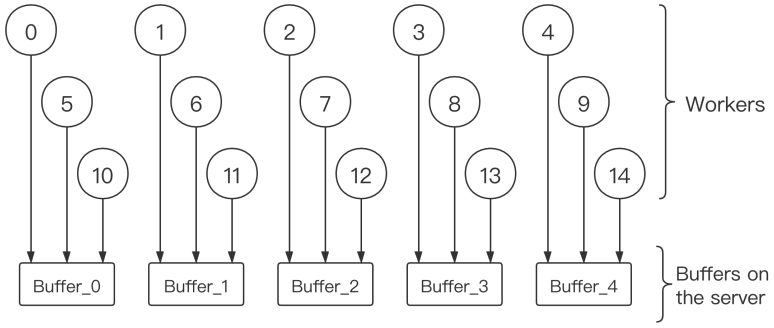


Figure 2: An example of buffers. A circle represents a worker, and the number in a circle is the worker ID. In this example, there are 15 workers and 5 buffers. The gradient received from worker_s is stored in a determinate buffer according to the worker ID s. For example, the gradient received from worker_0, worker_5 and worker_10 will be stored in buffer_0.

for each coordinate j, the aggregated value G_j will be between the $(q+1)$-th smallest value and the $(q+1)$-th largest value among the j-th coordinates of all candidate gradients. Thus, the gradient aggregated by a q-BR function is insensitive to at least q outliers.

We can find that q-BR condition gets stronger when q increases. In other words, if $Aggr(\cdot)$ is q-BR, then for any $0 < q' < q$, $Aggr(\cdot)$ is also q'-BR.

Remark 1. *It is not hard to find that when $B > 1$, mean function is not q-Byzantine Robust for any $q > 0$. We will illustrate this by a simple one-dimension example: $h_1, \ldots, h_{B-1} \in [0, 1] \subset \mathbb{R}$, while $h_B = 10 \times B$. Then $\frac{1}{B}\sum_{b=1}^{B} h_b \geq \frac{h_B}{B} = 10 \notin [0, 1]$. Namely, the mean is larger than any of the first $B - 1$ values.*

We find that the following two aggregation functions satisfy Byzantine Robust condition.

Definition 2 (Coordinate-wise median (Yin et al., 2018))**.** *For B candidate gradients $\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_B \in \mathbb{R}^d$, $\mathbf{h}_b = [h_{b1}, h_{b2}, \ldots, h_{bd}]^T$, $\forall b = 1, 2, \ldots, B$.* Coordinate-wise median *is defined as:*

$$Med([\mathbf{h}_1, \ldots, \mathbf{h}_B]) = [Med(h_{\cdot 1}), \ldots, Med(h_{\cdot d})]^T,$$

where $Med(h_{\cdot j})$ is the scalar median of the j-th coordinates of all candidate gradients, $\forall j = 1, 2, \ldots, d$.

Definition 3 (Coordinate-wise q-trimmed-mean (Yin et al., 2018))**.** *For any positive interger $q < B/2$ and candidate gradients $\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_B \in \mathbb{R}^d$, $\mathbf{h}_b = [h_{b1}, h_{b2}, \ldots, h_{bd}]^T$, $\forall b = 1, 2, \ldots, B$.* Coordinate-wise q-trimmed-mean *is defined as:*

$$Trm([\mathbf{h}_1, \ldots, \mathbf{h}_B]) = [Trm(h_{\cdot 1}), \ldots, Trm(h_{\cdot d})]^T,$$

where $Trm(h_{\cdot j})$ is the scalar q-trimmed-mean:

$$Trm(h_{\cdot j}) = \frac{1}{B - 2q} \sum_{b \in \mathcal{M}_j} h_{bj}.$$

$\mathcal{M}_j$ is the subset of $\{h_{bj}\}_{b=1}^B$ obtained by removing the q largest elements and q smallest elements, $\forall j = 1, 2, \ldots, d$.

In the following content, coordinate-wise median and coordinate-wise q-trimmed-mean are also called *median* and *trmean*, respectively. Proposition 1 shows the q-BR property of these two functions.

Proposition 1. *With B candidate gradients, coordinate-wise q-trimmed-mean is q-BR, and coordinate-wise median is $\lfloor \frac{B-1}{2} \rfloor$-BR.*

Here, $\lfloor x \rfloor$ represents the maximum integer not larger than x. According to Proposition 1, either median or trmean is a proper choice for aggregation function in BASGD.

3.3 Complexity

The time complexity for computing the average value of all buffers in each iteration is $O(Bd)$. If trmean or median is chosen as $Aggr(\cdot)$, the time complexity for each iteration is $O(Bd(B-2q))$ and $O(Bd)$ for trmean and median, respectively. Hence, the total time complexity is $O(TBd(B-2q))$ and $O(TBd)$ for trmean and median respectively, where T is the total number of iterations. For space complexity, B buffers are introduced in BASGD. Hence, the extra space complexity of BASGD is $O(Bd)$.

4. Convergence

In this section, we theoretically prove the convergence and resilience of BASGD against attack or error. Here we only present the main Lemmas and Theorems.

We make the following assumptions, which also have been widely used in stochastic optimization methods like SGD-based methods. Please note that we do not give any assumption about the behavior of Byzantine workers, which may behave arbitrarily.

Assumption 1 (Lower bound)**.** *Global loss function $F(\mathbf{w})$ is bounded below: $\exists F^* \in \mathbb{R}, F(\mathbf{w}) \geq F^*, \forall \mathbf{w} \in \mathbb{R}^d$.*

Assumption 2 (Unbiased estimation)**.** *For any loyal worker, it can use locally stored training instances to obtain an estimated gradient of the global loss function with no bias: $\mathbb{E}[\nabla f(\mathbf{w}; z_i)] = \nabla F(\mathbf{w}),\ \forall \mathbf{w} \in \mathbb{R}^d$.*

Assumption 3 (Limited second order moment)**.** *The gradient received from any loyal worker has limited second order moment: $\mathbb{E}[||\nabla f(\mathbf{w}; z_i)||^2 | \mathbf{w}] \leq D^2, \forall \mathbf{w} \in \mathbb{R}^d$.*

Assumption 4 (L-smoothness)**.** *Global loss function $F(\mathbf{w})$ is differentiable and L-smooth: $||\nabla F(\mathbf{w}) - \nabla F(\mathbf{w}')|| \leq L||\mathbf{w} - \mathbf{w}'||,\ \forall \mathbf{w}, \mathbf{w}' \in \mathbb{R}^d$.*

Assumption 5 (Limited number of Byzantine workers)**.** *The number of Byzantine workers at each iteration is not larger than r.*

Remark 2. *Please note that we do not explicitly assume limited delay here, because it can be guaranteed by the definition of loyal workers. Workers with too heavy delay would be seen as Byzantine workers in our analysis.*

Remark 3. *Please also note that we do not give any assumption about convexity. The analysis in this section is suitable for both convex and non-convex models in machine learning, such as logistic regression and deep neural networks.*

Before formally giving theoretical results about convergence, we define a type of constant $C_{M,K}$, which will be used in our theoretical results.

Definition 4. $\forall M \in \mathbb{Z},\ K \in \mathbb{Z},\ 0 < K \leq \frac{M}{2}$, *constant* $C_{M,K}$ *is defined as:*

$$C_{M,K} = \begin{cases} M, & K = 1; \\ \frac{M!(K-1)^{K-1}(M-K)^{M-K}}{(K-1)!(M-K)!(M-1)^{M-1}}, & 1 < K \leq \frac{M}{2}. \end{cases}$$

Lemma 1. *If* $Aggr(\cdot)$ *is* q*-Byzantine Robust, and there are no more than* r *Byzantine workers* ($r \leq q$)*, then:*

$$\mathbb{E}[||\mathbf{G}^t||^2 \mid \mathbf{w}^t] \leq C_{B-r,q-r+1} D^2 d.$$

Lemma 2. *If* $Aggr(\cdot)$ *is* q*-Byzantine Robust, then:*

$$||\mathbb{E}[\mathbf{G}^t - \nabla F(\mathbf{w}^t) \mid \mathbf{w}^t]|| \leq C_{B-r,q-r+1} Dd \cdot (\tau_{max} L \sqrt{C_{B-r,q-r+1} d} + 1).$$

Theorem 1. *If* $Aggr(\cdot)$ *is* q*-Byzantine Robust and* $r \leq q$*, taking learning rate* $\eta = \frac{1}{L\sqrt{T}}$*, we have:*

$$\begin{aligned} \frac{\sum_{t=0}^{T-1} \mathbb{E}[||\nabla F(\mathbf{w}^t)||^2]}{T} \leq & O\left(\frac{1}{\sqrt{T}}\right) + O\left(C_{B-r,q-r+1} D^2 d\right) \\ & + O\left([C_{B-r,q-r+1}]^{\frac{3}{2}} D^2 d^{\frac{3}{2}} \tau_{max} L\right). \end{aligned}$$

We can find that BASGD has a similar theoretical convergence rate as that of vanilla ASGD, with an extra constant variance which corresponds to the constant $C_{B-r,q-r+1}$.

Lemma 3. $\forall B, q, r \in \mathbb{Z}_+, 0 \leq r \leq q < \frac{B}{2}$,

$$C_{B-r,q-r+1} \leq \begin{cases} B \cdot \frac{e}{2\pi} \frac{\sqrt{B-1}}{\sqrt{(B-q-1)(q-r)}}, & r < q; \\ B - q, & r = q. \end{cases}$$

Then we have the following conclusions:

- When B and q are fixed, the upper bound of $C_{B-r,q-r+1}$ will increase when r (number of Byzantine workers) increases. Namely, the upper bound will be larger if there are more Byzantine workers.
- When B and r are fixed, q measures the Byzantine Robust degree of aggregation function $Aggr(\cdot)$. The factor $\frac{1}{\sqrt{(B-q-1)(q-r)}}$ is monotonically decreasing with respect to q, when $q < \frac{B-1+r}{2}$. Since $r \leq q < \frac{B}{2}$, the upper bound will decrease when q increases. Also, $B - q$ decreases when q increases. Namely, the upper bound will be smaller if $Aggr(\cdot)$ has a stronger q-BR property.
- In the worst case ($q = r$), the upper bound of $C_{B-r,q-r+1}$ is linear to B. Even in the best case ($r = 0, q = \lfloor \frac{B-1}{2} \rfloor$), the denominator is about $\frac{B}{2}$ and the upper bound of $C_{B-r,q-r+1}$ is linear to $\sqrt{B}$. That is to say, larger buffer number B might result in slower convergence and higher loss. Hence, unless necessary, we should choose B as small as possible.

5. Experiment

In this section, we empirically evaluate the performance of BASGD and baselines. Our experiments are conducted on a distributed platform with dockers. Each docker is bound to an NVIDIA Tesla V100 (32G) GPU. In all experiments, we choose 30 dockers as workers, and one extra docker as the server. All algorithms are implemented with PyTorch 1.3.

5.1 Experimental Setting

The algorithms are evaluated on the CIFAR-10 image classification dataset (Krizhevsky et al., 2009) with a deep learning model ResNet-20 (He et al., 2016). Each worker is manually set to have a delay k_{del} which is randomly sampled from a truncated standard normal distribution within interval $[0, +\infty)$.

We use cross-entropy loss on training set (training loss) and top-1 accuracy on test set to quantitatively measure the performance. In an asynchronous algorithm, the epoch number on different workers may differ. Hence, we use the average cross-entropy loss and average top-1 accuracy on all workers w.r.t. epochs as the final metrics.

We set initial learning rate $\eta = 0.1$ for each algorithm, and multiply η by 0.1 at the 80-th epoch and the 120-th epoch respectively. The weight decay is set to 10^{-4}. We run each algorithm for 180 epochs, but only the results of the first 160 epochs will be taken into account because some workers may finish earlier than others. Training set is randomly and equally distributed to different workers, and the batch size on each worker is set to 25.

Because the focus of this paper is on ABL, SBL methods cannot be directly compared with BASGD. The ABL method Zeno++ (Xie et al., 2019a) either cannot be directly compared with BASGD, because Zeno++ needs to store some training instances on the server. Hence, in our experiments, we compare BASGD with vanilla ASGD and the ABL baseline Kardam (Damaskinos et al., 2018). For Kardam, we set the dampening function to be $\Lambda(\tau) = \frac{1}{1+\tau}$ as suggested in (Damaskinos et al., 2018).

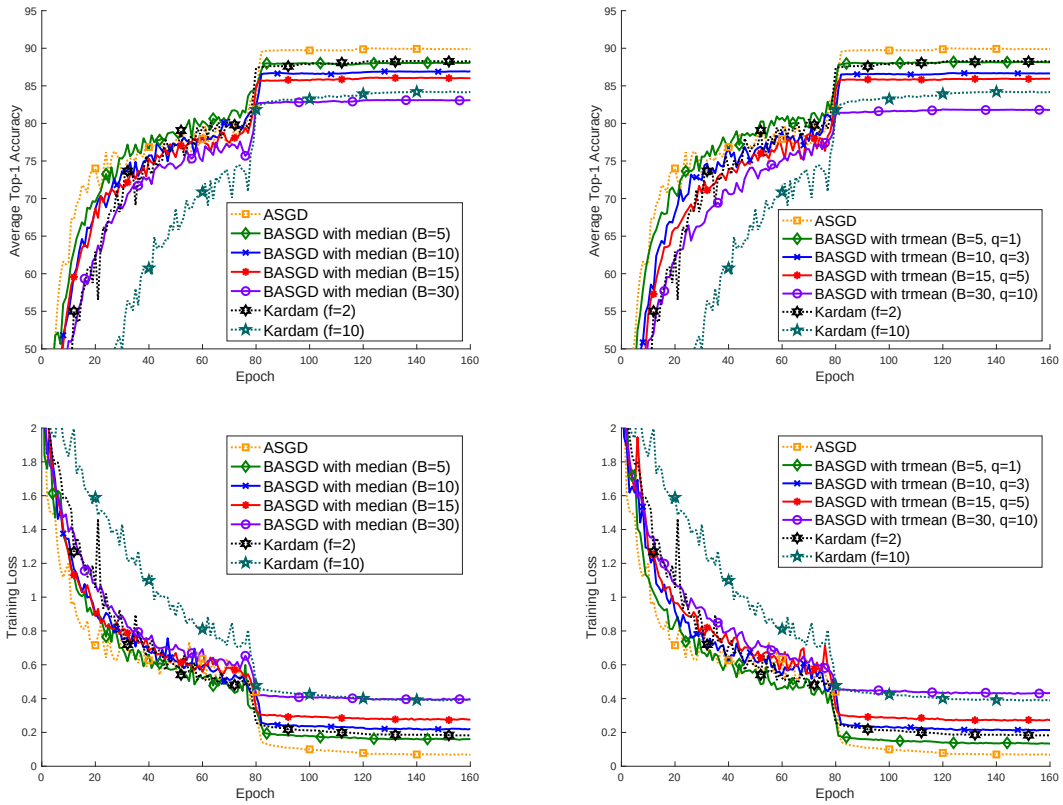
5.2 Cases without Byzantine Workers

We compare the performance of different methods when there are no Byzantine workers. Experimental results with median and trmean aggregation functions are illustrated in Figure 3(a) and Figure 3(b), respectively.

We can find that ASGD achieves the best performance. BASGD ($B > 1$) and Kardam have similar convergence rate as ASGD, but both sacrifice a little accuracy. Furthermore, the performance of BASGD gets worse when the buffer number B increases, which is consistent with the theoretical results. Please note that ASGD is a degenerated case of BASGD with $B = 1$. Hence, in the cases without attack or error, BASGD can achieve the same performance as ASGD by setting $B = 1$.

5.3 Cases with Byzantine Workers

We compare the performance of different methods under two types of attack: negative gradient attack (NG-attack) and random disturbance attack (RD-attack). In NG-attack, Byzantine workers will send $\tilde{\mathbf{g}}_{NG} = -k_{atk} \cdot \mathbf{g}$ to the server, where $\mathbf{g}$ is the correctly computed gradient based on its training data. In RD-attack, Byzantine workers will send $\tilde{\mathbf{g}}_{RD} = \mathbf{g} + \mathbf{g}_{rad}$

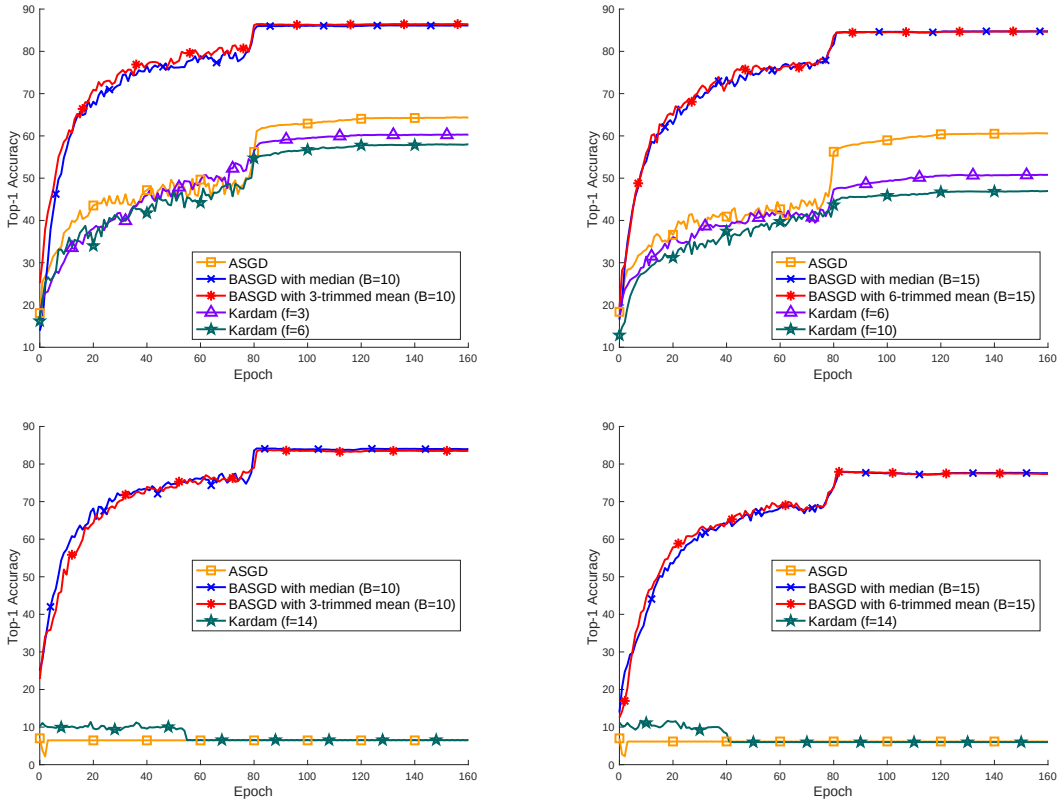


(a) median (b) trmean

Figure 3: Average top-1 test accuracy and training loss w.r.t. epochs when there are no Byzantine workers. The aggregation function in BASGD is median and q-trimmed mean. f is the hyper-parameter about the assumed number of Byzantine workers in Kardam.

to the server, where $\mathbf{g}_{rad}$ is a random vector with each coordinate randomly sampled from a normal distribution $\mathcal{N}(0, \sigma_{atk}^2)$. We set $k_{atk} = 10$ for NG-attack, and $\sigma_{atk}^2 = ||\frac{1}{5}\mathbf{g}||^2$ for RD-attack. NG-attack is a typical kind of malicious attack, while RD-attack can be seen as an accidental error with expectation $\mathbf{0}$. For each type of attack, we conduct two experiments in which there are 3 and 6 Byzantine workers, respectively. We respectively set 10 and 15 buffers for BASGD in these two experiments.

Figure 4(a) (for 3 Byzantine workers) and Figure 4(b) (for 6 Byzantine workers) illustrate the average top-1 test accuracy w.r.t. epochs. Figure 5(a) and Figure 5(b) illustrate the average training loss w.r.t. epochs. In Figure 5, some curves do not appear, because the value of loss function is extremely large or even not a number (NaN), due to the Byzantine attack.

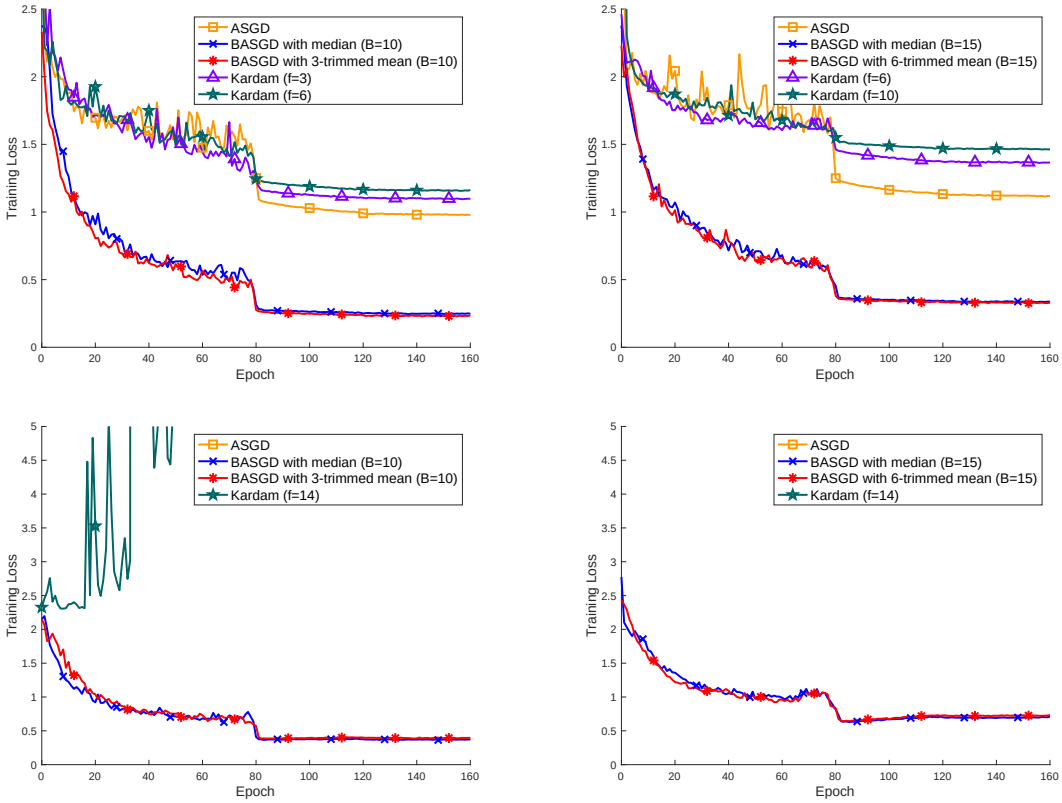


(a) Case of 3 Byzantine workers

(b) Case of 6 Byzantine workers

Figure 4: Average top-1 test accuracy w.r.t. epochs in face of random disturbance attack (above) and negative gradient attack (below), when the number of Byzantine workers $r = 3$ and 6. f is the hyper-parameter about the assumed number of Byzantine workers in Kardam.

We can find that BASGD significantly outperforms ASGD and Kardam under both RD-attack (accidental error) and NG-attack (malicious attack). Although ASGD and Kardam can still converge under the less harmful RD-attack, they both suffer a significant loss on accuracy. Under the NG-attack, even if we have set the number of assumed Byzantine workers to the maximum value for Kardam ($f = 14$), both ASGD and Kardam cannot converge. Hence, both ASGD and Kardam cannot resist malicious attack. On the contrary, both types of attack have little effect on the performance of BASGD. Furthermore, in our experiments we find that Kardam filters more than 80% of the gradients, which means that Kardam also filters most of the correct gradients in order to filter the faulty (error) gradients. This might explains why Kardam has a poor performance under malicious attack.



(a) Case of 3 Byzantine workers

(b) Case of 6 Byzantine workers

Figure 5: Average training loss w.r.t. epochs in face of random disturbance attack (above) and negative gradient attack (below), when the number of Byzantine workers $r = 3$ and 6. f is the hyper-parameter about the assumed number of Byzantine workers in Kardam. Some curves do not appear in the figure, because the value of loss function is extremely large or even not a number (NaN).

6. Conclusion

In this paper, we propose a novel method called BASGD for Byzantine learning. BASGD is asynchronous, which has more practical applications than synchronous methods. Furthermore, BASGD has no need to store any training instances on the server, which provides a potential solution for preserving privacy in distributed learning. In addition, BASGD is theoretically proved to have the ability of resisting against error and malicious attack. Empirical results show that BASGD can significantly outperform vanilla ASGD and other asynchronous Byzantine learning baselines, when there exists error or attack on workers.

References

Dan Alistarh, Zeyuan Allen-Zhu, and Jerry Li. Byzantine stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 4613–4623, 2018.

Peva Blanchard, Rachid Guerraoui, Julien Stainer, et al. Machine learning with adversaries: Byzantine tolerant gradient descent. In *Advances in Neural Information Processing Systems*, pages 119–129, 2017.

Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

Georgios Damaskinos, Rachid Guerraoui, Rhicheek Patra, Mahsa Taziki, et al. Asynchronous Byzantine machine learning (the case of SGD). In *Proceedings of the International Conference on Machine Learning*, pages 1145–1154, 2018.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul): 2121–2159, 2011.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

Martin Jaggi, Virginia Smith, Martin Takác, Jonathan Terhorst, Sanjay Krishnan, Thomas Hofmann, and Michael I Jordan. Communication-efficient distributed dual coordinate ascent. In *Advances in Neural Information Processing Systems*, pages 3068–3076, 2014.

Rie Johnson and Tong Zhang. Accelerating stochastic gradient descent using predictive variance reduction. In *Advances in Neural Information Processing Systems*, pages 315–323, 2013.

Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Keith Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. Advances and open problems in federated learning. *arXiv:1912.04977*, 2019.

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

Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

Jason D Lee, Qihang Lin, Tengyu Ma, and Tianbao Yang. Distributed stochastic variance reduced gradient methods by sampling extra data with replacement. *The Journal of Machine Learning Research*, 18(1):4404–4446, 2017.

Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. Communication efficient distributed machine learning with the parameter server. In *Advances in Neural Information Processing Systems*, pages 19–27, 2014.

Xiangru Lian, Ce Zhang, Huan Zhang, Cho-Jui Hsieh, Wei Zhang, and Ji Liu. Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 5330–5340, 2017.

Qihang Lin, Zhaosong Lu, and Lin Xiao. An accelerated proximal coordinate gradient method. In *Advances in Neural Information Processing Systems*, pages 3059–3067, 2014.

Chenxin Ma, Virginia Smith, Martin Jaggi, Michael Jordan, Peter Richtárik, and Martin Takác. Adding vs. averaging in distributed primal-dual optimization. In *Proceedings of the International Conference on Machine Learning*, pages 1973–1982, 2015.

Mark Schmidt, Nicolas Le Roux, and Francis Bach. Minimizing finite sums with the stochastic average gradient. *Mathematical Programming*, 162(1-2):83–112, 2017.

Shai Shalev-Shwartz and Tong Zhang. Stochastic dual coordinate ascent methods for regularized loss minimization. *Journal of Machine Learning Research*, 14(Feb):567–599, 2013.

Ohad Shamir, Nati Srebro, and Tong Zhang. Communication-efficient distributed optimization using an approximate newton-type method. In *Proceedings of the International Conference on Machine Learning*, pages 1000–1008, 2014.

Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

Shizhao Sun, Wei Chen, Jiang Bian, Xiaoguang Liu, and Tie-Yan Liu. Slim-dp: a multi-agent system for communication-efficient distributed deep learning. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, pages 721–729, 2018.

Jianqiao Wangni, Jialei Wang, Ji Liu, and Tong Zhang. Gradient sparsification for communication-efficient distributed optimization. In *Advances in Neural Information Processing Systems*, pages 1299–1309, 2018.

Lin Xiao. Dual averaging methods for regularized stochastic learning and online optimization. *Journal of Machine Learning Research*, 11(Oct):2543–2596, 2010.

Cong Xie, Oluwasanmi Koyejo, and Indranil Gupta. Zeno++: robust asynchronous SGD with arbitrary number of Byzantine workers. *arXiv:1903.07020*, 2019a.

Cong Xie, Sanmi Koyejo, and Indranil Gupta. Zeno: Distributed stochastic gradient descent with suspicion-based fault-tolerance. In *Proceedings of the International Conference on Machine Learning*, pages 6893–6901, 2019b.

Tianbao Yang. Trading computation for communication: Distributed stochastic dual coordinate ascent. In *Advances in Neural Information Processing Systems*, pages 629–637, 2013.

Dong Yin, Yudong Chen, Ramchandran Kannan, and Peter Bartlett. Byzantine-robust distributed learning: Towards optimal statistical rates. In *Proceedings of the International Conference on Machine Learning*, pages 5650–5659, 2018.

Hao Yu, Rong Jin, and Sen Yang. On the linear speedup analysis of communication efficient momentum SGD for distributed non-convex optimization. In *Proceedings of the International Conference on Machine Learning*, pages 7184–7193, 2019a.

Hao Yu, Sen Yang, and Shenghuo Zhu. Parallel restarted SGD with faster convergence and less communication: Demystifying why model averaging works for deep learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 5693–5700, 2019b.

Lijun Zhang, Mehrdad Mahdavi, and Rong Jin. Linear convergence with condition number independent access of full gradients. In *Advances in Neural Information Processing Systems*, pages 980–988, 2013.

Ruiliang Zhang and James Kwok. Asynchronous distributed admm for consensus optimization. In *Proceedings of the International Conference on Machine Learning*, pages 1701–1709, 2014.

Shen-Yi Zhao, Ru Xiang, Ying-Hao Shi, Peng Gao, and Wu-Jun Li. SCOPE: scalable composite optimization for learning on spark. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, pages 2928–2934. AAAI Press, 2017.

Shen-Yi Zhao, Gong-Duo Zhang, Ming-Wei Li, and Wu-Jun Li. Proximal SCOPE for distributed sparse learning. In *Advances in Neural Information Processing Systems*, pages 6551–6560, 2018.

Yi Zhou, Yingbin Liang, Yaoliang Yu, Wei Dai, and Eric P Xing. Distributed proximal gradient algorithm for partially asynchronous computer clusters. *The Journal of Machine Learning Research*, 19(1):733–764, 2018.

Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. Parallelized stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 2595–2603, 2010.