Dual-Way Gradient Sparsification for Asynchronous Distributed Deep Learning

Zijie Yan
Sun Yat-sen University
Guangzhou, China
yanzijie1996@gmail.com

Danyang Xiao
Sun Yat-sen University
Guangzhou, China
xiaody@mail2.sysu.edu.cn

Mengqiang Chen
Sun Yat-sen University
Guangzhou, China
chenmq9@mail2.sysu.edu.cn

Jieying Zhou
Sun Yat-sen University
Guangzhou, China
isszjy@mail.sysu.edu.cn

Weigang Wu
Sun Yat-sen University
Guangzhou, China
wuweig@mail.sysu.edu.cn

ABSTRACT

Distributed parallel training using computing clusters is desirable for large scale deep neural networks. One of the key challenges in distributed training is the communication cost for exchanging information, such as stochastic gradients, among training nodes. Recently, gradient sparsification techniques have been proposed to reduce the amount of data exchanged and thus alleviate the network overhead. However, most existing gradient sparsification approaches consider only synchronous parallelism and cannot be applied in asynchronous distributed training.

In this paper, we present the dual-way gradient sparsification (DGS) approach for asynchronous distributed training. Different from existing approaches, where workers download the global model from parameter server, our approach lets workers download model difference from the parameter server. With such design, both the gradients sent to server and the model parameters downloaded from server can be sparsified and the dual-way communication cost between server and workers can be significantly reduced. To preserve accuracy under the dual-way sparsification, we design a new momentum, SAMomentum, which is sparsification aware and offers significant optimization boost. Extensive experiments with different scales on ImageNet and CIFAR-10 show that, (1) compared with similar works, DGS with SAMomentum consistently achieves better performance; (2) DGS improves the scalability of asynchronous training, especially with limited networking infrastructure.

CCS CONCEPTS

• **Computing methodologies** → *Machine learning*; **Massively parallel algorithms**; • **Networks** → *Network algorithms*.

ICPP '20, August 17–20, 2020, Edmonton, AB, Canada

ACM ISBN 978-1-4503-8816-0/20/08…$15.00
https://doi.org/10.1145/3404397.3404401

KEYWORDS

Gradient Sparification, Deep Learning, Distributed Deep Learning, Asynchronous Training, Asynchronous Gradient Descent,

ACM Reference Format:
Zijie Yan, Danyang Xiao, Mengqiang Chen, Jieying Zhou, and Weigang Wu. 2020. Dual-Way Gradient Sparsification for Asynchronous Distributed Deep Learning. In *49th International Conference on Parallel Processing - ICPP (ICPP '20), August 17–20, 2020, Edmonton, AB, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3404397.3404401

1 INTRODUCTION

With the growing volume of training data and scale of deep neural networks (DNNs), training a large DNN model at a single machine may take an impractically long time. Distributed training, especially data parallelism, has become essential to reduce the training time of large DNNs on large data sets [7, 8, 17] . Distributed training relies on distributed optimizers to minimize the objective function of large-scale DNNs. Synchronous stochastic gradient descent (SSGD) [6, 26] is one of the most popular distributed optimizers, which distributes the workload among multiple workers and aggregates gradients computed by workers into the global model update equivalent to that of training with single worker but larger batch size. Since SSGD based distributed training may suffer from worker lags, which deteriorates efficiency and scalability, asynchronous stochastic gradient descent (ASGD) [14, 21, 28, 33] has been proposed to remove synchronization barrier among workers. ASGD is usually realized under the parameter server (PS) architecture [17]. PS refers to the node to collect gradients from workers and aggregate them into global model. Workers exchange gradients and model parameters with the server at their own pace. Since there is no longer synchronization barrier among workers, ASGD can significantly speed up the process of distributed training.

By increasing the number of worker nodes, distributed training via SSGD/ASGD can significantly reduce the total computation time of forward-backward passes on the same volume of data. However, either SSGD or ASGD introduces the communication overhead of exchanging model parameters and/or gradients in each iteration [30].

To reduce such communication cost, various solutions and techniques have been proposed. Roughly, communication cost among worker nodes can be reduced by increasing the batch size (i.e. reducing the frequency of communications) or reducing the volume of data

communicated in each iteration. Large batch training [10, 13, 29] tries to scale data-parallel SGD to more computing nodes without reducing the workload on each node. However, increasing batch size often leads to a significant loss in test accuracy [10, 12], and sophisticated hyper-parameter tuning like learning rate control [10, 15, 32] must be used to get better convergence accuracy. On the other hand, gradient compression is powerful method that can largely reduces the volume of exchanged data. There are two different ways to realize gradient compression, i.e., gradient quantization and gradient sparsification. Gradient quantization, e.g., 1-bit SGD [24], QSGD [3] and TernGrad [31], compresses the float-point values with prominent data representation and use fewer bits to represent each value. Gradient sparsification [2, 25], on the other hand, tries to exchange only essential gradient values. The importance of a gradient can be measured by the gradient magnitude or other factors. Storm et al. [26] prunes gradients using a fixed threshold, while Aji et al. [1] and others [5, 9, 30] proposed relative and adaptive thresholds to transmit only the essential gradients. Compare to gradient quantization, gradient sparsification can achieve much higher compression ratio in large scale DNN training. However, almost all existing gradient sparsification approaches are designed based on SSGD, i.e., they can be used for only synchronous training. In asynchronous training with ASGD, since workers may be using different model parameters at the same time, they need to download the whole model from server, and compression/sparsification is not applicable.

In this paper, we propose DGS, a novel approach for asynchronous training to overcome the communication bottleneck by compressing information exchanged. Different from existing asynchronous training, where workers need to download the whole model from the server, we let workers download the model difference between global and local from the server. Accordingly, DGS could sparsify both downward and upward communication to reduce communication volume. Such a dual-way compression approach can significantly reduce communication cost in asynchronous training. More importantly, to avoid loss of accuracy, we design, SAMomentum, a novel momentum suitable for asynchronous training. Compared with existing momentum, which can only be used under dense updates, our SAMomentum can achieve much better convergence performance in the sparse scenario.

We conduct three empirical studies to evaluate the proposed approach on Cifar10 and ImageNet, including accuracy analysis, scalability comparison, and system performance measurement. The experiment results show that our approach has better convergence performance and scalability than existing ones, inclusive of ASGD, Gradient Dropping [1], and Deep Gradient Compression [18]. Moreover, our approach works well with a low network bandwidth of 1Gbps, which is significant for asynchronous distributed training in mobile or wireless environments.

The rest of paper is organized as follows: Section 2 discusses related works on distributed training. The preliminaries are described in Section 3. Section 4 presents the design of our dual-way compression approach DGS and the design of the novel momentum SAMomentum. This section also provides proof of the correctness of our design, i.e., with our new momentum, the accuracy of our approach is equivalent to that of enlarging batch size for each model parameter. The experiments and results are reported in Section 5. Finally, Section 6 concludes the paper with future directions.

2 RELATED WORK

Researchers have proposed many approaches to optimize the SGD algorithm and communication pattern. The underlying idea is to relax the synchronization restriction to avoid waiting for slow workers. The HOGWILD algorithm [21] allows workers to read and write global model at will, which has been proven to converge for sparse learning problems. Downpour SGD [7] extended HOGWILLD to distributed-memory systems, which run multiple minibatches before exchanging gradients so as to reduce communication cost.

Another direction is to increase the minibatch size. Traditionally, due to memory constraints and accuracy degradation, minibatch size in deep learning usually less than 256. However, the scaling of data parallelism is limited by the size of minibatch. [10] proposed warmup approach and linear scaling rule to guarantee the convergence performance. [32] further introduce LARS, a method that changes the learning rate independently for each layer based on the norm of their weights and the norm of their gradient. It becomes possible to train with large minibatch sizes like 8k and 32k samples without significant injury on the accuracy, which makes the matrix operations more efficient and reducing the frequency of communication.

Gradient compression approaches, including gradient quantization and sparsification, are proposed to reduce communication data volume. Gradient quantization reduces the communication overhead by representing gradient values with fewer bits. [11] proposed the 16-bit float values representation for model parameters and gradients. 1-Bit SGD [24] and TernGrad [31] even quantize gradients to binary or ternary values, while still guarantee convergence with marginally reduced accuracy. QSGD [3] randomly quantizes gradients using uniformly distributed quantization points, which also explains the trade-off between model performance and gradient precision. However, even binary gradients can only achieve 32× reduced size, which is not really enough for large models and slow networks. Gradient sparsification approaches try to exchange selected valued rather than all of them. Storm et al. [26] proposed to prune gradients using a static threshold, and got up to 54× speedup with 80 nodes. However, it is hard to determine an appropriate threshold for a neural network in practice. [1] proposed Gradient Dropping, which sends only the top R% (R is fixed) gradients in terms of size, and accumulates the other gradients locally. [9] proposed to exchange only the important positive and negative gradients, based on their absolute value. DoubleSqueeze-async [27] performs the compression at both the worker side and the server side. It gathers m gradients at the server like HOGWILD [21], and then broadcasts compressed accumulated gradients to all workers. Lin et al. [18] proposed momentum correction to correct the disappearance of momentum discounting factor, along with some optimization tricks(including the warmup strategy and gradient clipping), which shows that Top-k sparsification SSGD can converge very closely to SGD.

3 PRELIMINARY AND MOTIVATION

3.1 Gradient Sparsification in SSGD

Various gradient sparsification approaches have been proposed to reduce the communication cost in distributed training. The key idea behind these approaches is to drop part of the stochastic gradient

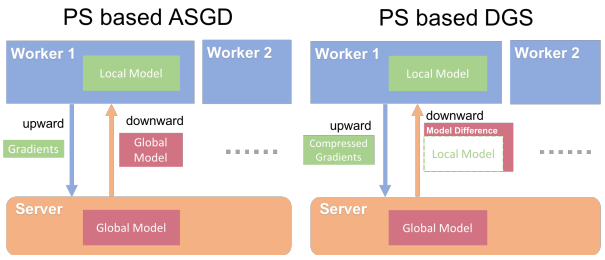

Figure 1: Architecture of PS Based Asynchronous Training

updates and only transmit the rest. For example, Aji et al. [1] propose to sparsify the gradients and transmit the elements with Top-k absolute values. Their sparsification method map the 99% smallest updates to zero then exchange sparse matrices, which significantly reduce the size of updates with marginally affecting the convergence performance. In order to avoid losing information, gradient sparsification usually accumulate the rest of the gradients locally, eventually, send all of the gradients over time. After each worker contributed the k largest gradients, we need average gradients from all workers than apply the averaged results to each worker. However, the sparsified gradients are generally associated with irregular indices (e.g., COO format), which makes it a challenge to accumulate the selected gradients from all workers efficiently. In decentralized SSGD, recent solutions uses the AllGather collective [22]. In parameter-server (PS) based SSGD, the server could do the average operation by adding support of sparse matrix. However, all the above methods to gather gradients are designed for SSGD. In ASGD, since different workers may be installed with different model versions, methods designed for SSGD will no longer work.

3.2 Asynchronous SGD

Same as other SGD algorithms, the goal of asynchronous SGD is to minimize an optimization problem $L(\theta)$, where L is the objective function, and the vector θ is the model's parameters. In asynchronous SGD, all N workers compute gradients asynchronously in parallel. After a worker k completes backpropagation with local model $\theta_{k,\text{prev}(k)}$, it will send gradients $\nabla L\left(\theta_{k,t}\right)$ to the parameter server and wait for the updated parameter θ back from server, where prev (k) denotes the last iteration that the worker k received update from the server. Once the server receives the gradient $\nabla L\left(\theta_{k,t}\right)$ from the worker k, it applies the gradient to its current set of parameters θ, and then sends θ back to the worker.

Figure 1 shows the overall architecture of existing asynchronous SGD. The upward communications are gradients from worker to the server, and the downward communications are global model parameters from the server to workers. We can compress upward communication by gradient sparsification methods. However, the downward communications of ASGD are unsuitable for gradient sparsification. This is because, different workers may keep different versions of the model at the same time, so the gradients aggregated at the server is meaningful for only the model version at the server. And if workers download the whole model at downward communications, the network bottleneck still exists. This motivates the proposal of DGS in this paper, a new gradient sparsification approach for ASGD with the sparsification aware momentum.

In DGS, we modify the update operations at the server. Instead of sending the global model to the worker k, DGS sends the model difference between global and local, which becomes compressible.

4 THE PROPOSED APPROACH DGS

In this section, we will introduce the detailed design of DGS. Firstly, we describe the dual-way sparsification operations, including the method to track the difference between global model and local model, and operations to do sparsification. Secondly, we present the design of sparsification aware momentum (SAMomentum), which is used to offer a significant optimization boost. At last, we discuss the equivalence between DGS and enlarged batch size, so as to show that DGS can guarantee convergence and model accuracy.

4.1 Architecture overview

Deep learning training is made up of millions of iterations. At the beginning of each iteration, our worker k fetches samples from the dataset, then propagates forward to calculate the loss, and backward to calculate the gradient update. Next, worker k calculates the threshold for sparsification, which we chose here as Top 1%, and of course some more advanced threshold selection methods can be used. Once the threshold is selected, worker$k$ compresses, encodes, and communicates the gradients. SAMomentum is calculated and updated based on sparse gradients.

Server receives and decodes gradients, then calculates the model difference according to the Model Difference Tracking mechanism. Finally, server performs secondary compression, encoding, and communication for model differences. Worker k updates the model after receiving the model differences. End of iteration.

While the worker k is processing this iteration, other workers are also processing their iterations concurrently and asynchronously.

Server side. In order to track model difference of each worker, the server must obtain the accumulation ($v_{k,\text{prev}(k)}$) of every update (M) and what has been sent to the worker k , so that the server can subtract the two values and get the model difference (G). After that, we can compress G by sending only the top R% values of each layer.

Worker side. We can simply regard the model difference (G) as accumulated gradients. Firstly, the worker receives G and updates its local model. Secondly, the worker can fetch data and compute gradients. Thirdly, it select the top R% gradients of each layer and compute SAMomentum. Finally, the worker sends sparsed gradients to the server.

4.2 Dual-way Gradient Sparsification for Asynchronous Training

4.2.1 Model Difference Tracking. In our dual-way gradient sparsification, the server maintains a separate vector v_k for each worker, which is the accumulation of the gradients that have been sent to worker k. The server no longer maintains the global model but maintains the accumulation of updates M_t. In the following, for simplicity of presentation, we denote the current stochastic gradient $\nabla L\left(\theta_{k,t}\right)$ by $\nabla_{k,t}$ for short:

$$M_{t+1} = M_t - \eta \nabla_{k,t} \tag{1}$$

Table 1: Notations and definitions

Notation	Description
k	The worker k.
t	The scalar timestamp that tracks the number of updates made to the server parameters (t starts at 0 and is incremented by one for each update).
c	Local timestamp of the worker.
T	The length of the sparse update interval.
prev (k)	The timestamp of the last update on worker k, which is also the timestamp of its local model.
$G_{k,t}$	Model difference between the server and the worker
v_k	Accumulation of model difference sent by the server to the worker k.
M_t	The accumulation of updates at time t.
θ	Parameters of the model.
$\theta[j]$	The jth layer of θ.
R	The ratio of sparsification. For example, R=1 means only top 1% of gradients/model difference values would be sent.
thr	Threshold calculated based on R.
η	The learning rate.
m	The momentum term.

η is the learning rate, and t is the timestamp of parameters on server. M_0 is a zero vector:

$$M_{t+1} = \theta_{t+1} - \theta_0 \tag{2}$$

After updating M, the server will send $G_{k,t+1}$ to the worker k and add $G_{k,t+1}$ on $v_{k,\text{prev}(k)}$:

$$G_{k,t+1} = M_{t+1} - v_{k,\text{prev}(k)} \tag{3}$$

$$\begin{aligned} v_{k,t+1} &= v_{k,\text{prev}(k)} + G_{k,t+1} \\ &= v_{k,\text{prev}(k)} + M_{t+1} - v_{k,\text{prev}(k)} \\ &= M_{t+1} \end{aligned} \tag{4}$$

In ASGD, a worker k receives the global model θ_{t+1} from server, replaces its local model $\theta_{k,\text{prev}(k)}$ with global model θ_{t+1}, and then moves to next iteration. However, DGS chooses to transmits $G_{k,t+1}$ rather than the global model:

$$\begin{aligned} \theta_{k,t+1} &= \theta_{k,\text{prev}(k)} + G_{k,t+1} \\ &= \theta_0 + M_{\text{prev}(k)} + M_{t+1} - v_{k,\text{prev}(k)} \\ &= \theta_0 + M_{\text{prev}(k)} + M_{t+1} - M_{\text{prev}(k)} \\ &= \theta_0 + M_{t+1} = \theta_{t+1} \end{aligned} \tag{5}$$

Eq.(5) indicates that DGS without sparsification is equivalent to ASGD. However, DGS transmits the difference between the global model and local model $G_{k,t+1}$, which can be sparsely compressed.

4.2.2 Dual-way Gradient Sparsification Operations. Algorithm 1 and Algorithm 2 present pseudo-code of dual-way gradient sparsification operations, at worker and server respectively. Algorithm 1 shows the gradient dropping scheme used at the workers in DGS (without SAMomentum), which is similar to the Top-k sparsification in distributed SGD [1]. Algorithm 2 shows the model difference based compression at the server. Please notice that, at line 5 of Algorithm 2, there is a switch for secondary compression. Under the small-scale setting, the server does not have the need for secondary compression of $G_{k,t+1}$, considering $G_{k,t+1}$ is the accumulation of several sparse updates and $G_{k,t+1}$ itself is highly sparse. However, under the circumstances of very limited communication resources (e.g., mobile devices) or a very large number of workers (e.g., federated learning), secondary compression (lines 5-11 of Algorithm 2) can be included to further reduce data exchanged in downward communication.

Secondary compression guarantees the sparsity of the send-ready model difference in downward communication, no matter how many workers are running. Furthermore, the secondary compression ratio can be adjusted according to the bandwidth. The function $sparsify$ () will zero out gradients less than the threshold thr and the function $unsparsify$ () will zero out gradients lager than the threshold (lines 9-12 of Algorithm 1). Substituting $M_{t+1} - v_{k,\text{prev}(k)}$ in Eq.(3) with $sparsify\left(M_{t+1} - v_{k,\text{prev}(k)}\right)$ yields the update rule of secondary compression:

$$G_{k,t+1} = sparsify\left(M_{t+1} - v_{k,\text{prev}(k)}\right) \tag{6a}$$

$$v_{k,t+1} = v_{k,\text{prev}(k)} + G_{k,t+1} \tag{6b}$$

Algorithm 1 DGS (without SAMomentum) at worker k

Require: Dataset $\mathcal{X}$
Require: Initial $\theta_0 = \{\theta[0], ..., \theta[J]\}$
Require: optimization function SGD
Require: $encode()$ function pack nonzero gradients to coordinate format.
Require: $decode()$ function unpack nonzero gradients from coordinate format.

1: $\theta_{k,0} = \theta_0$
2: $r_k \leftarrow \{0, ..., 0\}$
3: **for** $t = 0, 1, ...$ **do**
4: Sample data x from $\mathcal{X}$
5: $\nabla_{k,t} \leftarrow Backward(x, \theta_{k,\text{prev}(k)})$
6: $r_{k,t} \leftarrow r_{k,\text{prev}(k)} + \eta\nabla_{k,t}$
7: **for** $j = 0, ..., J$ **do**
8: // iterate over every layer
9: $thr \leftarrow R\%$ of $\left|r_{k,t}[j]\right|$
10: $Mask \leftarrow \left|r_{k,t}[j]\right| > thr$
11: $r_{k,t}[j] \leftarrow r_{k,t}[j] \odot \neg Mask$
12: $g_{k,t}[j] \leftarrow r_{k,t}[j] \odot Mask$
13: **end for**
14: Send $encode(g_{k,t})$ to the server
15: Recieve $G_{k,t+1}$ from the server
16: $\theta_{k,t+1} \leftarrow SGD\left(\theta_{k,\text{prev}(k)}, decode(G_{k,t+1})\right)$
17: **end for**

The server implicitly accumulates remaining gradient locally. Eventually, these gradients become large enough to be transmitted immediately. Lines 5-11 of algorithm 2 show how the server compresses $G_{k,t+1}$ in secondary compression, which eliminates the

Algorithm 2 DGS at server

Require: Initial $\theta_0 = \{\theta[0], ..., \theta[J]\}$
Require: $encode()$ function and $decode()$ function
$M^k_{t+1} \leftarrow \{0, ..., 0\}$
while Receive $encode(g_{k,t})$ from worker k **do**
  $M_{k,t+1} \leftarrow M_{k,t} - g_{k,t}$
  $G_{k,t+1} \leftarrow M_{t+1} - v_{k,\text{prev}(k)}$
  if Need secondary compression **then**
    for $j = 0, ..., J$ **do**
      // iterate over every layer
      $thr \leftarrow R\%$ of $\left|G_{k,t+1}[j]\right|$
      $Mask \leftarrow \left|G_{k,t+1}[j]\right| > thr$
      $G_{k,t+1}[j] \leftarrow G_{k,t+1}[j] \odot Mask$
    end for
  end if
  Send $encode(G_{k,t+1})$ to the worker k
  $v_{k,t+1} \leftarrow v_{k,\text{prev}(k)} - G_{k,t+1}$
  $\text{prev}(k) \leftarrow t + 1$
end while

overhead of the downward communication. We validated the effectiveness of secondary compression in the low bandwidth experiment.

4.3 Sparsification Aware Momentum

Momentum [20] is commonly used in deep training, which is known to offer a significant optimization boost. Momentum for SSGD training can be calculated as follows:

$$u_t = mu_{t-1} + \sum_{k=1}^{N} \left(\eta \nabla_{k,t}\right), \ \theta_{t+1} = \theta_t - u_t \tag{7}$$

u_t is the velocity. On parameter server based momentum ASGD (MASGD), the sever collect gradients and update momentum u_t [4]:

$$u_t = mu_{t-1} + \eta \nabla_{k,t}, \ \theta_{t+1} = \theta_t - u_t \tag{8}$$

With gradients sparsification as in Algorithm 1, it is further changed to be sparse momentum ASGD (SMASGD):

$$r_{k,t} = r_{k,t-1} + \eta * \nabla_{k,t}, \ \ u_t = mu_{t-1} + sparsify\left(r_{k,t}\right) \tag{9}$$

We store the remaining gradients $unsparsify\left(\nabla_{k,t}\right)$ locally:

$$r_{k,t} = unsparsify\left(r_{k,t}\right), \ \ \theta_{t+1} = \theta_t - u_t \tag{10}$$

4.3.1 Momentum Disappearing. In Eq.(10), remaining gradients r at workers **will not** participate in the momentum update in Eq. (9), since workers have not sent them yet. This results in disappearance of momentum and consequently, loss of convergence. $u^{(i)}_t$ denotes the $i-th$ position of a flattened velocity u_t. Ignoring interference from other nodes, after T iterations, $u^{(i)}$ of Eq.(8) becomes:

$$u^{(i)}_{t+T} = \eta \left[\cdots + m^{T-2}\nabla^{(i)}_{t+2} + m^{T-1}\nabla^{(i)}_{t+1}\right] + m^T u^{(i)}_t \tag{11}$$

In SMASGD, Eq.(9), $u^{(i)}$ is sent at t and $t+T$, and accumulated locally during sparse interval T:

$$u^{(i)}_{t+T} = \eta \left[\cdots + \nabla^{(i)}_{t+2} + \nabla^{(i)}_{t+1}\right] + m^T u^{(i)}_t \tag{12}$$

The disappearance of the factor m^{T-1} in Eq.(12), compared to Eq.(11), leads to loss of historical information. Momentum uses hyper parameter m to control the proportion of new information on each update ($0 < m < 1$). In Eq.(12), as T increases, m^T becomes smaller and smaller until it approaches 0:

$$u^{(i)}_{t+T} = \eta \left[\cdots + \nabla^{(i)}_{t+2} + \nabla^{(i)}_{t+1}\right] \tag{13}$$

As Eq.(13) no longer has the factor m to accumulates momentum, momentum terms of Eq.(13) will disappear. $u^{(i)}_{t+T}$ in Eq.(13) is more like a accumulation of updates, or, enlarged batchsize, rather than momentum. Could we use momentum in sparse scenario? This motivates our new design of the momentum, i.e., SAMomentum.

4.3.2 SAMomentum. Sparsification Aware Momentum (SAMomentum) is a novel momentum designed for gradient sparsification scenario. Algorithm 3 is the detailed implementation of DGS. DGS accumulates SAMomentum locally at each worker instead of collecting it at the server, and rescales remaining momentum in u:

$$\begin{aligned} u_{k,t} &= mu_{k,\text{prev}(k)} + \eta \nabla_{k,t} \\ &\quad + unsparsify\left(mu_{k,\text{prev}(k)} + \eta \nabla_{k,t}\right) * \left(\frac{1}{m} - 1\right) \end{aligned} \tag{14a}$$

$$g_{k,t} = sparsify\left(mu_{k,\text{prev}(k)} + \eta \nabla_{k,t}\right) \tag{14b}$$

$$\theta_{t+1} = \theta_t - g_{k,t} \tag{14c}$$

After each iteration, $u^{(i)}_{k,c}$ in Eq.(14) becomes:

$$u^{(i)}_{k,c} = \begin{cases} mu^{(i)}_{k,c-1} + \eta \nabla^{(i)}_{k,c} & > thr \\ \left(mu^{(i)}_{k,c-1} + \eta \nabla^{(i)}_{k,c}\right) * \frac{1}{m} & \leq thr \end{cases} \tag{15}$$

c is the c-th iteration of the worker. As shown in Algorithm 3, if $u^{(i)}_{k,c}$ is larger than the threshold thr, then the worker will send it immediately. The idea behind Eq.(14) is that after each iteration, the remaining momentum values ($< thr$) are more important than values just sent ($> thr$), so it is reasonable to magnify the unsent gradient. Next, we will explain why the magnification of is $\frac{1}{m}$ times. The change of velocity $u^{(i)}_k$ equals $\eta \sum_{i=1}^{T} \nabla^{(i)}_{k,c+i}$:

$$\begin{aligned} u^{(i)}_{k,c+T} &= mu^{(i)}_{k,c+T-1} + \eta \nabla^{(i)}_{k,c+T} \\ &= m\left(\left(mu^{(i)}_{k,c+T-2} + \eta \nabla^{(i)}_{k,c+T-1}\right) * \frac{1}{m}\right) + \eta \nabla^{(i)}_{k,c+T} \\ &= mu^{(i)}_{k,c+T-2} + \eta \nabla^{(i)}_{k,c+T-1} + \eta \nabla^{(i)}_{k,c+T} \\ &= \cdots \\ &= mu^{(i)}_{k,c} + \eta \sum_{i=1}^{T} \nabla^{(i)}_{k,c+i} \end{aligned} \tag{16}$$

T is the length of the sparse update interval between two iterations at which the $u^{(i)}_{k,t}$ is sent. If $T = 1$, then Eq.(16) equals Eq.(11), which indicates that SAMomentum equals the dense momentum at this situation. If $T > 1$, Eq.(16) can be treated as a large momentum update. Different from Eq.(13), as SAMomentum only multiplies $u^{(i)}_{k,c}$ by m once in the interval T, so momentum will not disappear in SAMomentum. DGC[18] employs momentum correction to restore

damaged sparse momentum SSGD, while our DGS employs SAMomentum to turn sparsification into a large momentum update, which outperforms DGC as shown in experimental results. In summary, the equivalence between SAMomentum and large batchsize MSGD is proven in the next section.

Algorithm 3 DGS on worker k

Require: Dataset $\mathcal{X}$
Require: Initial parameters $\theta_0 = \{theta[0], ..., theta[J]\}$
Require: optimization function SGD
Require: $encode()$ function and $decode()$ function
$\theta_{k,0} = \theta_0$
$u_k \leftarrow \{0, ..., 0\}$
for $t = 0, 1, ...$ **do**
  Sample data x from $\mathcal{X}$
  $\nabla_{k,t} \leftarrow Backward(x, \theta_{k,\text{prev}(k)})$
  $u_{k,t} \leftarrow mu_{k,\text{prev}(k)} + \eta\nabla_{k,t}$
  for $j = 0, ..., J$ **do**
    $thr \leftarrow R\%$ of $\left|u_{k,t}[j]\right|$
    $Mask \leftarrow \left|u_{k,t}[j]\right| > thr$
    $g_{k,t}[j] \leftarrow u_{k,t}[j] \odot Mask$
    $u_{k,t}[j] \leftarrow u_{k,t}[j] + \left(\frac{1}{m} - 1\right) u_{k,t}[j] \odot \neg Mask$
  end for
  Send $encode(g_{k,t})$ to the server
  Receive $G_{k,t+1}$ from the server
  $\theta_{k,t+1} \leftarrow SGD\left(\theta_{k,\text{prev}(k)}, decode(G_{k,t+1})\right)$
end for

4.4 Equivalence between SAMomentum and Enlarged Batch Size

SAMomentum can be considered as vanilla momentum SGD (MSGD) increasing batch size and learning rate by T times. With increasing batch size and learning rate, vanilla MSGD becomes:

$$\begin{aligned} u_{k,c+T}^{(i)} &= mu_{k,c}^{(i)} + T\eta * \frac{1}{T}\left(\nabla_{k,c+1}^{(i)} + \cdots + \nabla_{k,c+T}^{(i)}\right) \\ &= mu_{k,c}^{(i)} + \eta \sum_{i=1}^{T} \nabla_{k,c+i}^{(i)} \end{aligned} \tag{17}$$

For every single parameter of weight θ, Eq. (16) is equivalent to (17). The underlying idea of (14) is that, SAMomentum adaptively enlarges the batch size for every single parameter. In other words, we turn the sparsification into the magnification of batch size from parameter perspective, treating all gradients of sparse interval as a gradient.

Recent research [10, 32] attempted to enlarge the batch size of the entire model for efficient training, which makes it possible to train DNNs with large batch size without significant loss of accuracy. We also enlarge the batch size in distributed training, but our approach is in the parameter level rather than model level.

In SSGD and ASGD, each parameter of the local model has a consistent and fixed update interval and batch size. However, sparsification techniques like Gradient Dropping introduced different update intervals to each parameter, since workers only send part of gradients in each iteration. This change makes each parameter have their own asynchronous update interval. Therefore, SAMomentum adjusts the batch size of each parameter according to its update interval. Previous studies on gradients sparsification always keep tracking of remaining gradients, so as to avoid information loss. Different from previous works, with SAMomentum, DGS does not need to accumulate remaining gradients (the $r_{k,t}$ in Algorithm 1) anymore. Such an advantage comes from the equivalence to enlarged-batch-size transformation, because increasing batch size does not cause truncation of gradients.

5 EXPERIMENTAL EVALUATION

We conduct extensive experiments using a 36-GPU cluster (half of them are virtual GPU). The evaluation contains several parts: (a) testing the accuracy with 4 workers via two classical deep learning datasets: Cifar10 and ImageNet; (b) scaling DGS up to 32 workers and examining its scalability and generalization ability; (c) converge performance of DGS with low bandwidth; (d) measuring performance in terms of speedup and memory usage. We compare DGS with four other similar and representative approaches: single node SGD with vanilla momentum (MSGD), asynchronous SGD (ASGD), Gradient Dropping (GD) [1], and deep gradient compression (DGC) [18]. ASGD is vanilla asynchronous SGD without gradient sparsification. Please notice that, since our proposed momentum SAMomentum is included in DGS, we let DGC employ momentum correction and momentum factor masking to correct momentum.

However, Gradient Dropping and DGC are originally designed based on SSGD, and do not work in asynchronous training. For comparison purpose, we implement an asynchronous version of Gradient Dropping and DGC by adding model difference based compression as in our DGS, and they are denoted as GD-async and DGC-async respectively, in experiments.

Experimental results show DGS addressed both the bottleneck of asynchronous training and convergence degradation well.

5.1 Dataset and Models

Cifar10. consists of 50,000 training images and 10,000 validation images in 10 classes [16]. The baseline training using vanilla MSGD with a momentum of 0.7. The momentum coefficient of our approach and DGC-async is also 0.7. All experiments decrease the learning rate by the factor of 10 at epoch 30 and 40 out of 50 epochs. To simplify comparisons, we do not include other training tricks for improving accuracy.

ImageNet. dataset [23], known as ILSVRC2012, consists of RGB images, each of which is labeled as one of 1000 classes. Images are partitioned into 1.28 million training images and 50,000 validation images. The major hyper parameters of ResNet-18 on ImageNet are set as follows: batch size at each worker is 256; number of total epochs is 90; momentum is 0.7 (for single and 4 workers) or 0.45 (for 16 workers); initial learning rate is 0.1, which decays by a factor of 10 on epochs 30 and 60.

5.2 Experiments Setup

Hardware. the distributed environment is established via a 32-GPU cluster with eight machines, each of which is equipped with

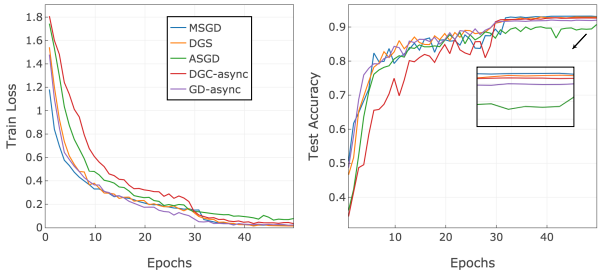

Figure 2: Learning curve of ResNet-18 on Cifar10 with 4 workers

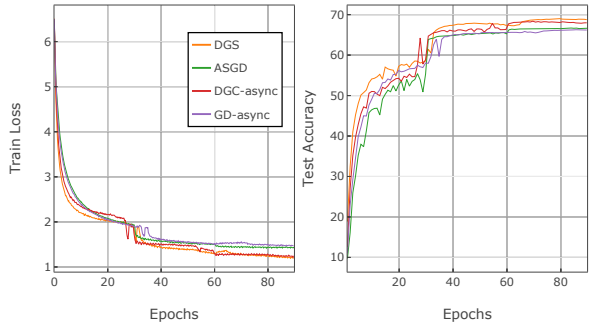

Figure 3: Learning curve of ResNet-18 on Imagenet with 4 workers

2 Intel(R) Xeon(R) Gold 6132 CPUs and 4 NVIDIA Tesla V100 SXM2 GPUs. Each GPU is used as a worker node. The workers and the server communicate via a 10 Gbps Ethernet LAN.

Software. All GPU machines are installed with Linux 3.10.0, NVIDIA GPU driver 410 and CUDA 10. We implement all the algorithms via PyTorch 1.2, a popular distributed deep learning framework with great flexibility. The parameter server is implemented using PyTorch distributed API with gloo backend. Furthermore, as the baseline approach, vanilla MSGD is run with a single node, and others are all executed asynchronously based on our model difference compression as in Algorithm 1 and Algorithm 2. Therefore, the results of them may be different from their synchronous experimental results reported in literature.

5.3 Accuracy and Analysis

We firstly examine our approach on the Cifar10 dataset. Figure 2 is the Top-1 accuracy and training loss of ResNet-18 on Cifar10 with 4 workers. The gradient sparsity of DGC-async, GD-async, and DGS is 99%. The learning curve of GD-async (purple) and ASGD (green) is worse than MSGD (blue) due to gradient staleness. With momentum correction, the learning curve of DGC-async (red) converges slightly slower, but the accuracy is much closer to the baseline. DGS always outperforms the other three approaches, and its convergence performance is very close to single-node MSGD.

As shown in Figure 3, when scaling to the large-scale dataset, DGS exhibits more significant advantage. Moreover, the accuracy curve of DGS converges smoothly and stably, which is obviously better than other approaches, especially DGC. Table 2 shows the detailed accuracy results. ResNet-18 on ImageNet converges very well using our distributed approach with 4 workers.

Table 2: Results of ResNet-18 Trained on Cifar10 and ImageNet

Dataset	Training Method	Workers in total	Top-1 Accuracy
Cifar10	MSGD	1	93.08%
	ASGD	4	90.74%
	GD-async		92.01%
	DGC-async		92.64%
	DGS		92.91%
ImageNet	MSGD	1	69.4%
	ASGD	4	66.68%
	GD-async		66.26%
	DGC-async		68.37%
	DGS		69.0%

Table 3: ResNet-18 Trained on Cifar10 Dataset

Workers in total	Batchsize per worker	Training Method	Top-1 Accuracy	
1	256	MSGD	93.08%	-
		ASGD	91.54%	-1.54%
		GD-async	92.15%	-0.93%
		DGC-async	92.75%	-0.33%
		DGS	**92.97%**	**-0.11%**
4	128	ASGD	90.7%	-2.38%
		GD-async	92.01%	-1.07%
		DGC-async	92.64%	-0.44%
		DGS	**92.91%**	**-0.17%**
8	64	ASGD	90.46%	-2.62%
		GD-async	91.81%	-1.27%
		DGC-async	92.37%	-0.71%
		DGS	**93.32%**	**+0.24%**
16	32	ASGD	90.53%	-3.01%
		GD-async	91.43%	-1.65%
		DGC-async	92.28%	-0.80%
		DGS	**92.98%**	**-0.10%**
32	16	ASGD	88.36%	-4.71%
		GD-async	91%	-2.08%
		DGC-async	91.86%	-1.22%
		DGS	**92.69%**	**-0.39%**

Table 4: ResNet-18 Trained on ImageNet Dataset

Workers in total	Batchsize per iteration	Training Method	Top-1 Accuracy	
1	256	MSGD	69.40%	-
4		ASGD	66.68%	-2.72%
		GD-async	66.26%	-3.14%
		DGC-async	68.37%	-1.03%
		DGS	**69.00%**	**-0.40%**
16		ASGD	66.25%	-3.15%
		GD-async	66.19%	-3.21%
		DGC-async	67.62%	-1.78%
		DGS	**68.25%**	**-1.15%**

5.4 Scalability and Generalization Ability

During the whole experimental process, we scale DGS, ASGD, GD-async, and DGC-async up to 32 workers on Cifar10 and ImageNet. Experiments all have the same hyper-parameter settings, except for the decrease of batch size for Cifar10 in case of small numbers of iterations.

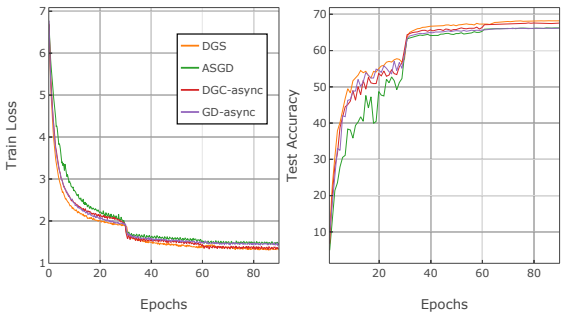

Figure 4: Learning curve of ResNet-18 on Imagenet with 16 workers

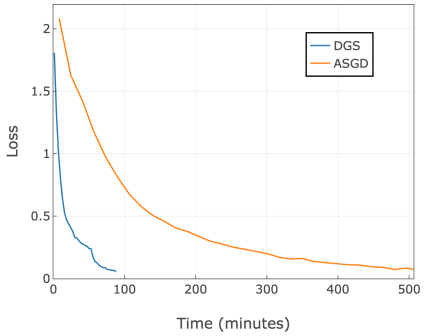

Figure 5: Time vs Training Loss on 8 workers with 1Gbps Ethernet

In Table 3, we can observe that, the test accuracy of other approach decreases as the number of workers increases. This is because, with more nodes, the more staleness asynchrony brings. However, the test accuracy of our approach in 4, 8 and 16 workers is 92.91%, 93.32%, and 92.98% respectively. Therefore, our approach has better converge performance and even defeats the staleness brought by asynchronous in distributed scenarios. Compare to other approaches on 32 workers, our approach achieves the best accuracy, and the accuracy only drops a little (-0.39%) due to a large number of workers. At the same time, the convergence performance of other methods is greatly reduced: ASGD drops to 88.36% (-4.71%), GD-async drops to 91% (-2.08%) and DGC-async drops to 91.86% (-1.22%).

ImageNet results (Table 4) demonstrate the scalability of DGS on large dataset. Due to the limited time and computing resources, we only conducted experiments on 4 workers and 16 workers (Figure 4). Experiment results above show that DGS scales very well and outperforms all other approaches when the number of workers increases.

Furthermore, we have got amazing results by changing hyper-parameters. Since asynchrony introduces momentum to the SGD update [19], we reduce the momentum from 0.7 to 0.3 in the experiments of 32 workers. Surprisingly, the test accuracy increases to 93.7%. This indicates that, the accuracy of ResNet-18 is fully maintained while using DGS.

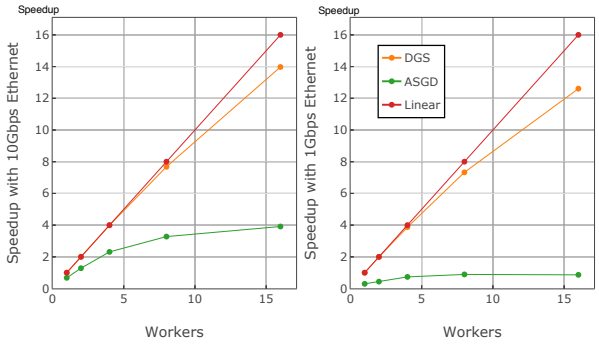

Figure 6: Speedups for DGS and ASGD on ImageNet with 10Gbps and 1Gbps Ethenet

5.5 The Effect of Dual-way Sparsification with Low bandwidth

Figure 5 shows training loss against wall-clock time when training ResNet-18 on Cifar10 with 8 workers. To simulate low bandwidth environments, we reduce network bandwidth from 10Gbps to 1Gbps. The compress ratio of the secondary compression is set to be 99%. Comparing ASGD and DGS in Figure 5, we can find that DGS benefits a lot from Dual-way Sparsification. Our approach completes the training for 88 minutes, while ASGD takes 506 minutes, resulting in a speedup of 5.7×.

5.6 System Performance

5.6.1 Speedup. Figure 6 shows the training speedup with different network bandwidth values. In this evaluation we did not take the data IO time into account. As the number of workers increases, the acceleration of ASGD decreases to nearly zero due to the bottleneck of communication. In contrast, DGS achieves nearly linear speedup with 10Gbps. With 1Gbps network, ASGD only achieves 1× speedup with 16 workers, while DGS achieves 12.6× speedup, which proves the the superiority of our DGS under low bandwidth.

5.6.2 Memory Usage. Since $v_{k,t}$ is used to track model difference in our design, the additional memory usage at server equals NumOfWorkers × ParameterMemOfModel. In our experiments, one NVIDIA Tesla V100 16GB card of server can support more than 300 ResNet-18[1] workers. At the worker side, DGS replace the vanilla momentum and local gradient accumulation [18] with SAMometum, which reduces memory usage of ParameterMemOfModel for each worker. Thus, DGS moves part of memory usage from worker to server, with the unchanged overall memory usage.

5.7 Summary

The Effect of Dual-way Gradient Sparsification. Since we implement GD-async by adding model difference based compression (downward communication) and GD itself already sparsifies the upward communication, the only difference between GD-async and ASGD is that, GD-async has dual-way gradient sparsification. In

[1]The parameter memory of ResNet-18 is 46 MB, according to: https://github.com/albanie/convnet-burden

Table 5: Techniques in DGS

	Gradient Sparsification	Momentum	Momentum Correction	Remaining Gradients Accumulation
ASGD	N	N	N	N
GD	GD	N	N	Y
DGC	GD	vanilla momentum	Y	Y
GD-async / DGS without SAMomentum	Model Difference Ttracking based Dual-way Gradient Sparsification	N	N	Y
DGC-async	Model Difference Ttracking based Dual-way Gradient Sparsification	vanilla momentum	Y	Y
DGS	Model Difference Ttracking based Dual-way Gradient Sparsification	SAMomentum	N	N

other words, GD-async combines ASGD and dual-way gradient sparsification. Consequently, comparing GD-async and ASGD shows the effect of Dual-way Gradient Sparsification. Experiment results of GD-async and ASGD explicitly illustrate that, dual-way gradient sparsification brings gradient sparsification for ASGD while fully preserving the convergence.

The Effect of Sparsification Aware Momentum. We can consider GD-async as the combination of ASGD and dual-way gradient sparsification, so it is obvious that, DGS (with SAMomentum) is the combination of ASGD, dual-way gradient sparsification, and SAMomentum (shown in Table 5). Likewise, DGC-async can be viewed as a combination of ASGD, dual-way gradient sparsification, momentum correction, and momentum factor masking. Thus, comparing with GD-async, DGS converges much better than GD-async, which clearly demonstrates the effectiveness of SAMomentum. Furthermore, the comparison between DGS and DGC-async proves that, SAMomentum is superior to the momentum correction [18], a popular and SOTA method of gradient sparsification.

6 CONCLUSION AND FUTURE WORK

Recent studies mainly focus on solving the bottleneck of gradients exchange. Issues here are: First, the downward communication of asynchronous training is a model exchange, which is unsuitable for gradient sparsification and could cause a serious communication bottleneck. Second, information loss in gradient sparsification leads to degradation of convergence performance. Our major contribution lies in these two points. We propose a model difference tracking mechanism to enable sparsification for downward communications from parameter server to the workers, and also design the momtemum SAMomtemum to bring significant optimization boost. DGS enables large-scale asynchronous distributed training with inexpensive, commodity networking infrastructure.

In future, the combination of DGS and other compression approaches (e.g. TernGrad [31], randomly coordinates dropping [30]) can be considered. Also, the new momentum SAMomentum is a general design and can be used to design new synchronization training approaches.

ACKNOWLEDGMENTS

This research is partially supported by National Natural Science Foundation of China (U1801266, U1811461, U1711263), and Guangdong Natural Science Foundation of China (2018B030312002) and Guangdong Key Fields R&D Plan of China (2019B020228001).

REFERENCES

[1] Alham Fikri Aji and Kenneth Heafield. 2017. Sparse communication for distributed gradient descent. *arXiv preprint arXiv:1704.05021* (2017).
[2] Dan Alistarh, Torsten Hoefler, Mikael Johansson, Nikola Konstantinov, Sarit Khirirat, and Cédric Renggli. 2018. The convergence of sparsified gradient methods. In *Advances in Neural Information Processing Systems*. 5973–5983.
[3] 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).
[4] Saar Barkai, Ido Hakimi, and Assaf Schuster. 2019. Gap Aware Mitigation of Gradient Staleness. *arXiv preprint arXiv:1909.10802* (2019).
[5] Chia-Yu Chen, Jungwook Choi, Daniel Brand, Ankur Agrawal, Wei Zhang, and Kailash Gopalakrishnan. 2018. Adacomp: Adaptive residual gradient compression for data-parallel distributed training. In *Thirty-Second AAAI Conference on Artificial Intelligence*.
[6] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. 2013. Deep learning with COTS HPC systems. In *International conference on machine learning*. 1337–1345.
[7] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. 2012. Large scale distributed deep networks. In *Advances in neural information processing systems*. 1223–1231.
[8] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.
[9] Nikoli Dryden, Tim Moon, Sam Ade Jacobs, and Brian Van Essen. 2016. Communication quantization for data-parallel training of deep neural networks. In *2016 2nd Workshop on Machine Learning in HPC Environments (MLHPC)*. IEEE, 1–8.
[10] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677* (2017).
[11] Suyog Gupta, Wei Zhang, and Fei Wang. 2016. Model accuracy and runtime tradeoff in distributed deep learning: A systematic study. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*. IEEE, 171–180.
[12] Elad Hoffer, Itay Hubara, and Daniel Soudry. 2017. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. In *Advances in Neural Information Processing Systems*. 1731–1741.
[13] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, et al. 2018. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *arXiv preprint arXiv:1807.11205* (2018).
[14] Janis Keuper and Franz-Josef Pfreundt. 2015. Asynchronous parallel stochastic gradient descent: A numeric core for scalable distributed machine learning algorithms. In *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*. ACM, 1.
[15] Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997* (2014).
[16] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. *Learning multiple layers of features from tiny images*. Technical Report. Citeseer.
[17] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 583–598.
[18] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. 2017. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887* (2017).
[19] Ioannis Mitliagkas, Ce Zhang, Stefan Hadjis, and Christopher Ré. 2016. Asynchrony begets momentum, with an application to deep learning. In *2016 54th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, 997–1004.
[20] Boris T Polyak and Anatoli B Juditsky. 1992. Acceleration of stochastic approximation by averaging. *SIAM Journal on Control and Optimization* 30, 4 (1992), 838–855.
[21] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*. 693–701.
[22] Cèdric Renggli, Dan Alistarh, Torsten Hoefler, and Mehdi Aghagolzadeh. 2018. Sparcml: High-performance sparse communication for machine learning. *arXiv preprint arXiv:1802.08021* (2018).
[23] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. 2015. Imagenet large scale visual recognition challenge. *International journal of computer vision* 115, 3 (2015), 211–252.
[24] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 2014. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Fifteenth Annual Conference of the International Speech Communication*

Association.

[25] Sebastian U Stich, Jean-Baptiste Cordonnier, and Martin Jaggi. 2018. Sparsified SGD with memory. In *Advances in Neural Information Processing Systems*. 4447–4458.

[26] Nikko Strom. 2015. Scalable distributed DNN training using commodity GPU cloud computing. In *Sixteenth Annual Conference of the International Speech Communication Association*.

[27] Hanlin Tang, Xiangru Lian, Tong Zhang, and Ji Liu. 2019. DoubleSqueeze: Parallel Stochastic Gradient Descent with Double-Pass Error-Compensated Compression. *arXiv preprint arXiv:1905.05957* (2019).

[28] John Tsitsiklis, Dimitri Bertsekas, and Michael Athans. 1986. Distributed asynchronous deterministic and stochastic gradient optimization algorithms. *IEEE transactions on automatic control* 31, 9 (1986), 803–812.

[29] Weiran Wang and Nathan Srebro. 2017. Stochastic nonconvex optimization with large minibatches. *arXiv preprint arXiv:1709.08728* (2017).

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

[31] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2017. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Advances in neural information processing systems*. 1509–1519.

[32] Yang You, Igor Gitman, and Boris Ginsburg. 2017. Scaling sgd batch size to 32k for imagenet training. *arXiv preprint arXiv:1708.03888* 6 (2017).

[33] Shanshan Zhang, Ce Zhang, Zhao You, Rong Zheng, and Bo Xu. 2013. Asynchronous stochastic gradient descent for DNN training. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 6660–6663.