ADTopk: All-Dimension Top-k Compression for High-Performance Data-Parallel DNN Training

Zhangqiang Ming
Huazhong University of Science and Technology
Wuhan, Hubei, China
zqming@hust.edu.cn

Yuchong Hu*
Huazhong University of Science and Technology
Wuhan, Hubei, China
Shenzhen Research Institute of Huazhong University of Science and Technology
Shenzhen, Guangdong, China
yuchonghu@hust.edu.cn

Wenxiang Zhou
Huazhong University of Science and Technology
Wuhan, Hubei, China
wxzhoucs@hust.edu.cn

Xinjue Zheng
Huazhong University of Science and Technology
Wuhan, Hubei, China
zhengxinjue@hust.edu.cn

Chenxuan Yao
Huazhong University of Science and Technology
Wuhan, Hubei, China
deadfffool@hust.edu.cn

Dan Feng
Huazhong University of Science and Technology
Wuhan, Hubei, China
dfeng@hust.edu.cn

ABSTRACT

Data-parallel deep neural networks (DNN) training systems deployed across nodes have been widely used in various domains, while the system performance is often bottlenecked by the communication overhead among workers for synchronizing gradients. *Top-k* sparsification compression is the de facto approach to alleviate the communication bottleneck, which truncates the gradient to its largest k elements before sending it to other nodes.

However, we observe that the traditional Top-k still has performance issues: i) the gradient at each layer of a DNN is typically represented as a tensor of multiple dimensions, and the largest k elements selected by the traditional Top-k are centered in only some of all dimensions and hence the training may miss many dimensions (we call *dimension missing*), which leads to low convergence performance; ii) the traditional Top-k performs the selection by globally sorting the gradient elements in each layer (we call *single global sorting*), which leads to a low GPU core parallelism and hence a low training throughput.

In this paper, we propose an all-dimension Top-k sparsification scheme, called ADTopk, which selects the largest k elements from *all* dimensions of the gradient tensor in each layer, meaning that each dimension must provide some elements, so as to avoid the dimension missing. Further, ADTopk enables each dimension to perform sorting *locally* within the elements of the dimension, and thus all dimensions can perform multiple local sortings independently and parallelly, instead of a single global sorting for the entire gradient tensor in each layer. On top of ADTopk, we further propose an interleaving compression scheme and an efficient threshold estimation algorithm so as to enhance the performance of ADTopk. We build a sparsification compression data-parallel DNN training framework and implement a compression library containing state-of-the-art sparsification algorithms. Experiments on a local cluster and Alibaba Cloud show that compared with state-of-the-art sparsification compression methods, ADTopk improves the training throughput by 12.28%-293.49% with approximately the same convergence accuracy as the non-compression baseline.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning**; **Parallel computing methodologies**.

KEYWORDS

High Performance, Data-Parallel DNN Training, Gradient Sparsification Compression

ACM Reference Format:
Zhangqiang Ming, Yuchong Hu, Wenxiang Zhou, Xinjue Zheng, Chenxuan Yao, and Dan Feng. 2024. ADTopk: All-Dimension Top-k Compression for High-Performance Data-Parallel DNN Training. In *International Symposium on High-Performance Parallel and Distributed Computing (HPDC '24), June 3–7, 2024, Pisa, Italy.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3625549.3658678

*Corresponding author: Yuchong Hu, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, Hubei, China.

HPDC '24, June 3–7, 2024, Pisa, Italy

ACM ISBN 979-8-4007-0413-0/24/06…$15.00
https://doi.org/10.1145/3625549.3658678

1 INTRODUCTION

Deep neural network (DNN) models have developed significantly in computer vision and natural language processing in recent years [6, 10, 15, 40]. To efficiently train DNN models over the continuously growing datasets, high-performance data-parallel DNN training systems are deployed across nodes to accelerate the training process of DNN models [5, 26, 33]. In a typical setting, each node (or *worker*)

has a full copy of model parameters and a subset of the input data. For each training iteration, each worker first computes the gradients based on its own data, and then exchanges parameter updates (i.e., *gradients*) through gradient synchronization strategies, and thus updates the model parameters.

Despite widely used, data-parallel DNN training systems have to face the communication problem: frequent gradient exchanges among multiple workers make communication take up a significant portion of the overall training time (as high as 90%) [3, 5, 10, 43], which has become one of the major bottlenecks in high-performance data-parallel DNN training systems [20, 27, 31]. Many recent works propose gradient sparsification compression techniques [2, 11, 12, 19, 22, 38, 39, 41, 49] to alleviate the communication problem, among which the Top-k sparsification compression approach is proved to be one of the most effective compression methods [30]. The Top-k sparsification can save up to 90% of the gradient exchanges by only sending the largest k elements of the original gradient to reduce the transmission overhead with a limited impact on the model convergence [2, 19, 32, 50].

However, we find that the gradient at each layer of a DNN is typically represented as a tensor of multiple dimensions [7], where the dimensions are represented as channels or attention heads of the DNN (see §2.1 for more details), while most existing studies on Top-k and its variants seldom consider the impact of dimensions on the model. In this paper, we observe that there exist two challenges when considering dimensions in the traditional Top-k sparsification compression (see §3.1 for more details):

Challenge #1 (Dimension missing). Our observation shows that the traditional Top-k sparsification compression is likely to select the gradient tensor's elements which are concentrated in some of all dimensions, while some other dimensions are often missing (called *dimension missing*). Dimension missing of the gradient tensor causes the DNN model to lose the opportunity to learn more representations from the missing dimensions [21, 35, 43], which may reduce the convergence performance of the model [1].

Challenge #2 (Single global sorting). Our observation also shows that the traditional Top-k typically selects the k largest elements of the gradient at each layer by a *single global sorting* operation on its elements, which may entail multiple *global synchronizations* across GPU cores which causes a low GPU core parallelism (analyzed in §3.1), thus limiting the training throughput [9].

In this paper, to address the above two challenges, we propose an **A**ll-**D**imension **Top-k** sparsification compression scheme, called ADTopk. For Challenge #1, ADTopk selects the largest k elements of the gradient tensor from all dimensions, each of which must provide some elements to avoid dimension missing. For Challenge #2, ADTopk allows each dimension to perform a local sorting within the elements of the dimension, and thus all dimensions can perform multiple local sortings in parallel to replace the single global sorting. Further, we also propose an interleaving compression scheme and an efficient threshold estimation algorithm to enhance the performance of ADTopk. Our contributions include:

- We conduct measurement analysis and give two observations to show the traditional Top-k sparsification's two challenges: dimension missing and single global sorting. We propose ADTopk which performs sparsification compression on all dimensions of the gradient at each layer rather than only some dimensions in the traditional Top-k sparsification to address the above two challenges (§3).
- We design ADTopk, which leverages its all dimension based gradient selection and realizes two schemes: matrix-based sparsification and multiple local sorting, to deal with the challenges of dimension missing and single global sorting, respectively. Atop ADTopk, we also propose an interleaving compression scheme that accelerates model convergence and an efficient threshold estimation algorithm that reduces the compression overhead (§4).
- We implement a high-performance data-parallel DNN training framework with a compression library containing our ADTopk and state-of-the-art gradient sparsification compression methods (DGC [19], Gaussiank [36], Redsync [12], OkTopk [18], and SIDCo [22]). We also implement communication libraries (Allreduce, Allgather, and our proposed AllgatherFast) to support different sparsification communication schemes (§5). ADTopk is opensourced at https://github.com/YuchongHu/ADTopk.
- We train six typical DNN models on a local cluster as well as in Alibaba Cloud. Experimental results show that compared with state-of-the-art sparsification methods, ADTopk increases the training throughput by 12.28%-293.49%, while maintaining almost the same convergence performance of the non-compression approach (§6).

2 BACKGROUND AND RELATED WORK

In this section, we introduce the basic concepts of data-parallel DNN training (§2.1), discuss some state-of-the-art sparsification compression methods and analyze their shortcomings (§2.2).

2.1 Data-Parallel DNN Training

DNN model training. A DNN model basically contains multiple neural network *layers*, each with many model *parameters*. To reach the model convergence (i.e., the model training process reaches a stable state), DNN training iterates over a dataset many times (i.e., *epochs*), each of which contains multiple *iterations*. In each iteration, the model runs forward and backward propagation to generate a *gradient* for each layer so as to update the model parameters.

Data-Parallel DNN training. Data-parallel DNN training is often applied to multiple workers (*nodes*) to speed up the training of DNN models [3, 33, 42], and enables each worker to have a complete copy of the model. In each iteration, the dataset is split into *mini-batches* and assigned to different workers, each worker uses the assigned mini-batch to compute the gradient of each layer, which is then synchronized with other workers to collectively update the global model parameters. We denote the parameter updates of the model during data-parallel DNN training as:

$$x_{i,t+1} = x_{i,t} - \eta \frac{1}{K} \sum_{i=1}^{K} G_{i,t}. \tag{1}$$

where $x_{i,t}$ and $G_{i,t}$ are the neural network parameter and the gradient of the i-th worker in the t-th iteration, respectively, η determines the step size at each iteration (called *learning rate*), and K is the number of workers. Gradient accumulation and averaging ($\frac{1}{K}\sum_{i=1}^{K} G_{i,t}$) across multiple workers is often implemented using a standard communication library (e.g., Horovod [33]). In this paper, we focus on data-parallel distributed DNN training on top of Horovod.

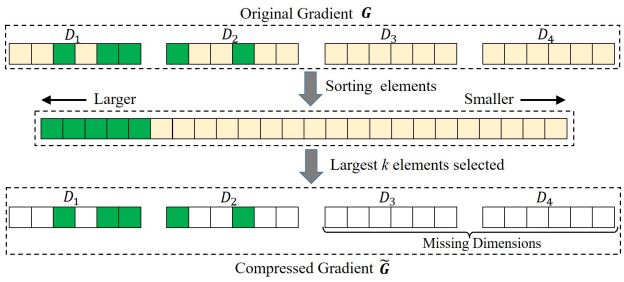


Figure 1: Example of the traditional Top-k sparsification. Note that some dimensions (e.g., D_3 and D_4) are missing, which will be analyzed in §3.1.

Importance of a gradient's dimensions. A gradient $G_{i,t}$ at each layer of a DNN is typically represented as a multi-dimensional tensor [7]. Basically, different dimensions of the gradient seek different features during training; for example, in computer vision, different dimensions determined by different *channels* have their own importance [21]. Therefore, it is natural that more dimensions of the gradient allow the model to capture more features, achieving a higher convergence accuracy; for example, for natural language processing in translation, more dimensions determined by more *attention heads* have lower per-word perplexities (ppl) [43]. In this paper, the importance of a gradient's dimensions helps motivate the main idea of our proposed ADTopk (§3.2).

Performance bottleneck in data-parallel DNN Training. Modern deep learning models scale rapidly from millions to billions of parameters [6, 10], and each worker in the data-parallel DNN training system frequently communicates with each other to exchange gradients to update the model parameters. Thus, the increasing communication overhead becomes the main performance bottleneck, and a common way that decreases the communication overhead is to apply gradient compression techniques. However, a naive gradient compression often yields very low convergence performance, so many existing studies have focused on high-performance data-parallel DNN training systems [5, 29, 45], which aim to ensure higher convergence accuracy for distributed DNN training while significantly improving the training throughput.

2.2 Gradient Compression

Basics. Many gradient compression algorithms have been proposed to alleviate the communication bottleneck in high-performance data-parallel DNN training systems by reducing the transmitted data during gradient synchronization. The gradient compression techniques usually have three categories: sparsification [30], quantization [53], and low-rank [44].

Sparsification is a key approach to lower the communication volume, which leverages the sparsity of the gradient tensor and selects a subset of its all elements (in terms of the absolute value) for communication. Many recent studies [2, 13, 19, 29, 30, 36, 39, 49, 52] focus on how to improve the performance of sparsification compression, and we aim for sparsification with improved accuracy and training throughput in this paper.

Top-k sparsification compression. For different gradient sparsification compression techniques, the *Top-k* sparsification compression (or Top-k for short) has been proven to be one of the most effective compression methods [30]. The traditional Top-k process selects k elements with the largest absolute value (or called *magnitude*) out of all elements of the gradient while zero-ing out the rest. As shown in Figure 1, an original gradient G, represented as a 4-dimensional tensor, is composed of four dimensions D_1, D_2, D_3 and D_4, each with six elements. After Top-k with $k = 5$, we see that the original gradient tensor G with 24 elements are reduced to a compressed gradient tensor $\tilde{G}$ with $k = 5$ elements which have the largest magnitudes. However, Top-k is a lossy compression that impairs the training model's convergence accuracy. Although the error feedback method mitigates the loss of accuracy, the traditional Top-k may cause the gradient to lose some important information (e.g., dimensions of the gradient), making the convergence accuracy unrecoverable [1, 5, 46].

The traditional Top-k in Pytorch [26] often uses Radix Sort [34] to select the largest k elements. Radix Sort is a linear sorting algorithm that sorts elements by processing them digit by digit in a way that distributes the elements into *buckets* based on each digit's value; by *repeatedly* sorting the elements by their significant digits, from the least significant to the most significant, Radix Sort achieves the final sorted order. In this way, the traditional Top-k needs to perform Radix Sort on the dimensions of the gradient repeatedly and globally, and hence has to entail many synchronizations of multiple buckets across GPU cores [9], which incurs a large sparsification compression overhead.

To reduce the sparsification overhead, many state-of-the-art studies [12, 18, 19, 22, 30, 36] focus on the *threshold*-based Top-k sparsification, which only communicates the elements of the gradient with magnitudes greater than or equal to a fixed given threshold in each iteration. In contrast to the traditional Top-k, which globally sorts all elements of the gradient tensor, the threshold-based sparsification method requires only one traversal. These state-of-the-art threshold-based sparsification methods are specified as follows:

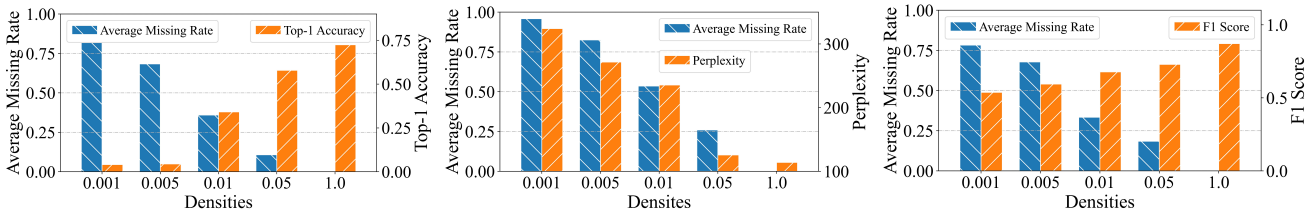
DGC [19]: DGC samples only 0.1% to 1% of the elements of gradient to estimate the threshold, but becomes inefficient if the number of gradient elements is large, since it requires to invoke selection twice on the subsets of the original gradient.

Gaussiank [36]: Gaussiank regards the gradient at each iteration as a normal distribution and estimates the threshold by exploiting the percent point function, but how to accurately adjust the threshold is difficult and hence incurs additional computational overhead, since Gaussiank does not fit an exact normal distribution.

Redsync [12]: Redsync estimates the threshold by moving the ratio between the maximum and mean elements of the gradient, but the threshold estimated in this way is often inaccurate and needs to be adjusted frequently.

SIDCo [22]: SIDCo is similar to Gaussiank and periodically estimates threshold by fitting some sparsity-inducing distributions, but its multi-stage fitting may cause additional computational overhead.

OkTopk [18]: OkTopk, which is the state-of-the-art gradient sparsification, estimates one threshold for all neural network layer gradients at the end of the backward propagation, and re-estimates it in a fixed iteration. Note that OkTopk leverages a global sparsification while most of the other sparsification methods (including the above four state-of-the-art ones and our proposed ADTopk) are *layer-wised* sparsification.



(a) ResNet-50 [15] on Cifar-100 [17] (b) LSTM [23] on Wikitext-2 [24] (c) BERT-base [10] on SQuAD [28]

Figure 2: The average dimension missing rate and different convergence accuracy (i.e., Top-1 accuracy on Cifar-100, Perplexity on WikiText-2, and F1 score on SQuAD) for three training tasks using the traditional Top-k.

Note that the existing studies on Top-k and its variants seldom consider the impact of gradient tensor's dimensions on model training performance, but they are important for the model's convergence accuracy (see §2.1). In this paper, we are the first to manipulate dimensions to optimize the model's convergence accuracy as well as the training throughput, which will be analyzed in §3.

3 OBSERVATIONS AND MOTIVATION

In this section, we give two key observations that show challenges of the traditional Top-k sparsification (§3.1), and then describe the motivation of our main idea to address the challenges (§3.2).

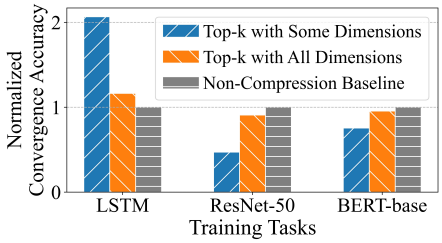


Figure 3: Comparison between Top-k with some dimensions (traditional) and Top-k with all dimensions in terms of baseline-based normalized convergence accuracy (i.e., Top-1 accuracy on ResNet-50, Perplexity on LSTM, and F1 score on BERT-base).

3.1 Observations

We conduct measurement analysis to give two following observations. Note that the compression method proposed in this paper aims to deal with the layer-wise gradient tensor, and thus our observations and experiments also target at the gradient tensor at each layer of the DNN.

Measurement setting. We evaluate three typical training tasks (specified in §6.1) on a local cluster with 8 NVIDIA V100 GPUs connected by a 25Gbps Ethernet, including ResNet-50 [15] on Cifar-100 [17], LSTM [23] on Wikitext-2 [24], and BERT-base [10] on SQuAD [28]. We consider different compression rates (i.e, *densities*) ranging from 0.001 to 0.05 [30], while $density = 1.0$ indicates the non-compression scheme as the baseline in this paper.

Observation #1. The traditional Top-k causes dimension missing which leads to a low convergence accuracy. Figure 2 shows the impact of gradient tensor's dimensions on the model's convergence accuracy. We define the *average dimension missing rate* of the gradient tensor for all layers of the DNN as the number of the missing dimensions divided by the total number of all dimensions of the gradient tensor.

First, we observe that the largest k elements of the gradient selected the traditional Top-k are only centered in some of all dimensions, while many other dimensions are often missing, which we call *dimension missing*. For example, Figure 2(b) shows that, for LSTM on WikiText-2, when the density ranges from 0.001 to 0.05, the average dimension missing rate ranges from 96% to 26%, which indicates many dimensions of the gradient are missing after compression, and more dimensions are missed severe as the density decreases.

Second, we observe that a high average dimension missing rate leads to a low convergence accuracy. For example, when the density is 0.001, we see that (i) Figure 2(a) shows that, for ResNet-50 on Cifar-100, the Top-1 accuracy is only 4%; (ii) Figure 2(b) shows that, for LSTM on WikiText-2, the Perplexity (which measures how uncertain the model is when it tries to predict the next word in a sequence, and a lower Perplexity indicates a higher predictive accuracy) is 324, whose convergence accuracy decreases significantly compared to the baseline (Perplexity=112); and (iii) Figure 2(c) shows that, for BERT-base on SQuAD, the F1 score (which indicates a single metric that balances both precision and recall) is only 50%, much lower than the baseline (87%).

Third, we observe that a lower average dimension missing rate can obtain a higher convergence accuracy. For example, in Figure 2 (a), when the density changes from 0.01 to 0.05, the average dimension missing rate decreases from 36% to 11%, while the Top-1 accuracy of ResNet-50 increases from 34% to 58%. The same observations exists for LSTM and BERT-base.

In addition, as shown in Figure 3, to further verify the effect of the gradient tensor's dimensions on convergence accuracy, we keep the density constant (e.g., density = 0.01) and compare the normalized convergence accuracy between Top-k with some dimensions (traditional) and Top-k with all dimensions on three training tasks. Note that we also consider the non-compression baseline as the normalized convergence accuracy (i.e., its value is 1). We can see that for LSTM, the traditional Top-k has more than 50% dimension loss at density=0.01, and its perplexity is about 2.1× that of the non-compression baseline. On the contrary, the Top-k with all dimensions has a perplexity close to the baseline. ResNet-50 and BERT-base also have the same conclusion.

Therefore, we find that *the traditional Top-k often incurs missing dimensions of the gradient tensor and thus impairs the convergence accuracy of model training, and reducing missing dimensions can improve the convergence accuracy significantly.*

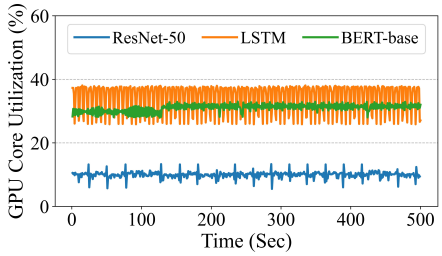


Figure 4: GPU core utilization of the traditional Top-k.

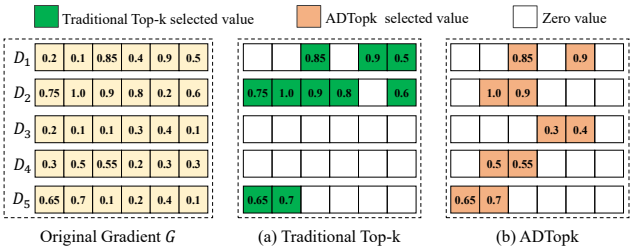


Figure 5: The gradient sparsification process for the traditional Top-k and ADTopk**.**

Observation #2. The traditional Top-k incurs a single global sorting which limits the training throughput. The traditional Top-k compression method (e.g., the `torch.topk` function) based on Pytorch typically uses Radix Sort to select the largest k elements [26, 34], which requires flattening the gradient tensor into a single vector and performing a global sorting, and thus may cause low parallelism in the GPU cores. Figure 4 shows how the traditional Top-k impacts the GPU cores parallelism. We monitor the *GPU core utilization* (i.e., the percentage of GPU core usage) and use the *DCGMG* tool to monitor the streaming multiprocessor (SM) occupancy as the GPU core utilization, which is usually used to indicate the degree of GPU core parallelism [25]. For all three different models, we set the density to 0.01 [30].

We observe that the traditional Top-k has only 13%, 31%, and 35% of GPU core utilization at the peak, respectively, which implies the low GPU core parallelism. The reason is that the traditional Top-k has to globally sort elements of the gradient at each layer of a DNN and then select the k largest ones, which forms a *single global sorting* of all elements within the gradient that needs to synchronize many sorted results across GPU cores, thereby degrading the parallelism. Further, based on the study [9], the low GPU core parallelism will make the compression process inefficient and hence lower the training throughput.

Therefore, we find that *the traditional Top-k has to perform a single global sorting which often incurs a low GPU core parallelism, thus limiting the training throughput.*

3.2 Motivation

Observation #1 shows the dimension missing challenge of the traditional Top-k and also implies the way: the more dimensions Top-k can maintain, the higher convergence accuracy the model can achieve. The reason behind is that each dimension (even if the one with a small magnitude) of the gradient at each layer of a DNN can provide its distinct feature learning to increase the expressiveness and convergence accuracy of the DNN training, which has been confirmed in existing studies [21, 35, 43].

Therefore, we propose ADTopk, whose above main idea is to make Top-k select the largest k elements from all dimensions of the gradient tensor, rather than only some dimensions as in the traditional Top-k; that is, each dimension of the gradient tensor needs to contribute some elements in parameter updating for the best comprehensive feature learning.

Interestingly, based on the main idea of ADTopk, we find that it can perform multiple sorting processes across all dimensions of the gradient independently (i.e., each dimension of the gradient tensor can locally sort its entire elements individually), which provides additional potential for better parallelism to address the challenge shown in Observation #2.

While ADTopk can help address the two challenges, how to design ADTopk in details still remains unexplored, which will be specified in §4.

4 THE DESIGN OF ADTOPK

In this section, we design ADTopk based on the main idea in §3.2 with the design goals as follows:

- **Improving convergence accuracy and throughput:** ADTopk avoids dimension missing via a matrix-based sparsification scheme to improve the convergence accuracy, and increases the GPU core parallelism via a multiple local sorting scheme to improve the training throughput (§4.1).
- **Accelerating convergence:** ADTopk extends to ADTopk-i that employs an interleaved compression scheme to combine ADTopk and the traditional Top-k to speed up the convergence (§4.2).
- **Reducing sparsification overhead:** ADTopk-i extends to ADTopk-t that employs an efficient threshold estimation scheme to reduce the compression overhead (§4.3).

4.1 ADTopk: All-Dimension Top-k Sparsification

To realize the all-dimension sparsification structure of ADTopk, we design two schemes based on Top-k: *matrix-based sparsification* and *multiple local sortings*, which aim to address the two challenges given in §3.1.

Matrix-based sparsification. To realize the all-dimension sparsification, ADTopk treats the original gradient tensor G at each layer of a DNN as an $M \times N$ *matrix*, different from the traditional Top-k that flattens the gradient tensor into a one-dimensional *vector* [7]. Based on the $M \times N$ matrix, ADTopk selects the largest k elements of G in three steps: (i) ADTopk splits matrix G uniformly into M dimensions (i.e., each of which has N elements), denoted by $G_1, G_2, \cdots, G_M$; (ii) ADTopk independently performs a Top-$\frac{k}{M}$ sparsification for each dimension: $\tilde{G}_i$ =Top-$\frac{k}{M}(G_i)$, which means each dimension of the gradient tensor selects the $\frac{k}{M}$ largest elements; and (iii) ADTopk obtains the compressed gradient $\tilde{G} = [\tilde{G}_1, ..., \tilde{G}_M]$. Note that M is determined by the neural network layer.

Figure 5 illustrates the process of selecting elements of the gradient tensor for the traditional Top-k and ADTopk. The original gradient tensor matrix G includes five dimensions (from D_1 to D_5), each of which has six elements (to describe the G easily, we take the absolute values of all elements). We see that:

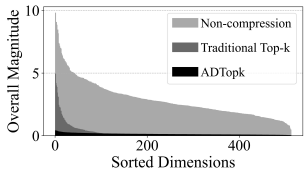


Figure 6: Overall magnitudes.

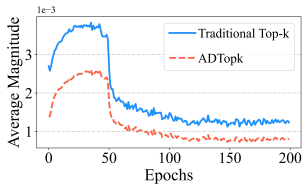


Figure 7: Average magnitude.

The traditional Top-k: Figure 5(a) shows that among all 30 elements of the original gradient tensor G, it selects the ten largest elements which are only centered in three dimensions D_1, D_2, and D_5, while D_3 and D_4 are ignored after compression.

ADTopk: Figure 5(b) shows that it selects two largest elements for each dimension, which avoids the dimension missing of the gradient tensor. It allows each dimension to participate in parameter updating to learn richer features, thus improving the convergence accuracy of the model. Note that the number of elements of the gradient tensor selected by the traditional Top-k and ADTopk remain the same (i.e., both select ten gradient elements).

Multiple Local Sortings. As stated in §2.2, the traditional Top-k performs a global sorting on the elements of the gradient tensor organized in buckets, which entails many synchronizations of multiple buckets (scale with k) across GPU cores [9], thereby degrading the GPU core parallelism, especially for a large k. Thus, many studies show that the traditional Top-k sparsification is only efficient for a small k [30].

As stated in §3.2, instead of the traditional Top-k that is based on a single global sorting, ADTopk can provide potential to enable multiple sortings to work across all dimensions of the gradient tensor in parallel, satisfying that each sorting only sorts the elements of its local dimension. Specifically, for each dimension, ADTopk only performs Top-$\frac{k}{M}$ sparsification, where $\frac{k}{M}$ is much smaller than k for fast sorting [30]; for all M dimension of the gradient tensor, ADTopk performs M sorting processes in parallel to further increase the training throughput [9].

4.2 ADTopk-i: Interleaving Compression in ADTopk

Limitation of ADTopk**.** We first analyze the limitation of ADTopk by measurement analysis, where the measurement trains ResNet-50 [15] on Cifar-100 [17] with density =0.01, on a local cluster with 8 NVIDIA V100 GPUs connected by a 25Gbps Ethernet.

Figure 6 shows the overall magnitudes of each dimension's elements of a gradient tensor for three schemes (non-compression, traditional Top-k, and ADTopk), where the number of the dimension of the gradient tensor is 512. For each of 512 dimensions, we compute the sum of all its elements' magnitudes, and sort all the dimensions' sums by size. We see that the traditional Top-k can select elements that are likely to have large magnitudes, while ADTopk will select the elements with magnitudes which are not as large as in the traditional Top-k. Figure 7 further shows that the average of the overall magnitude of each dimension's elements over 200 epochs for the traditional Top-k and ADTopk. The average magnitude for ADTopk is always smaller than that for the traditional Top-k, especially in the early epochs.

Algorithm 1: ADTopk-i Algorithm

Input: Stochastic gradient $G_{i,t}$ at worker i and iteration $t(t > 0)$, gradient selection ratio k.
Output: Compressed gradient $\tilde{G}_{i,t}$.

1 **Initialization:** $isAllDimension = True$;
2 **Function** `InterleavingTopk`$(G_{i,t}, k)$**:**
3 **if** $isAllDimension$ **then**
4 $values, indices =$ `ADTopk`$(G_{i,t}, k)$;
5 $isAllDimension = False$;
6 **else**
7 $values, indices =$ `TradTopk`$(G_{i,t}, k)$;
8 $isAllDimension = True$;
9 **end**
10 $\tilde{G}_{i,t} = [values, indices]$;
11 **return** $\tilde{G}_{i,t}$

Therefore, we find that ADTopk cannot select elements whose magnitudes are as large as the traditional Top-k. As shown in the definition of parameter updating in Equation (1), when the gradient magnitude is smaller, the parameter updating is weaker, and thus leads to a slower convergence speed.

Design idea: Although ADTopk benefits the convergence accuracy, the convergence speed becomes slower than the traditional Top-k. The main reason is that the ADTopk requires each dimension of the gradient tensor to provide its $\frac{k}{M}$ largest elements and some dimensions who still have large-magnitude elements cannot provide any more, leading to a slow parameter update or falling into a local optimum. In fact, the larger magnitude values can contribute more to the model convergence [16, 41, 48], so the traditional Top-k can help accelerate the convergence of the model.

Therefore, we find that there exists a *trade-off* between convergence accuracy and convergence speed for ADTopk (higher accuracy) and the traditional Top-k (faster convergence), which motivates us to design an interleaving sparsification compression scheme which combines both ADTopk and the traditional Top-k.

Design details: Algorithm 1 describes the details of ADTopk with interleaving compression (called ADTopk-i). Specifically, ADTopk-i has four steps: (Step 1) ADTopk-i determines whether to perform ADTopk based on *isAllDimension*, which is initialized to *True* (Line 1); (Step 2) When *isAllDimension* is *True*, ADTopk-i performs ADTopk (we call `ADTopk`) to select more dimensions of the gradient at a layer in the current iteration and sets *isAllDimension* to *False* (Lines 3-5); (Step 3) When *isAllDimension* is *False*, ADTopk-i performs the traditional Top-k (we call `TradTopk`) to select more large-magnitude elements at the same layer in the next iteration and sets *isAllDimension* to *True* (Lines 6-9); (Step 4) ADTopk repeats Step 2 and Step 3 until finishing all iterations, and finally returns the compressed gradient $\tilde{G}_{i,t}$ (denoted as values and indices) for synchronization (Lines 10-11).

Note that Algorithm 1 emphasizes the interleaving compression scheme of performing `ADTopk` and `TradTopk` between iterations, i.e., if `ADTopk` is performed in the current iteration, `TradTopk` will be performed in the next iteration. All layers of a DNN within an iteration perform the same compression, either `ADTopk` or `TradTopk`.

Algorithm 2: ADTopk-t Algorithm

Input: Thresholds re-estimation period τ, threshold update bound ϵ, warmup training bound σ, reuse threshold thr_reuse.
Output: Compressed gradient $\tilde{G}_{i,t}$.

1 **Function** `ThresholdTopk`$(G_{i,t}, k)$**:**
2 **if** $t < \sigma$ **then**
3 $values, indices =$ `TradTopk`$(G_{i,t}, k)$;
4 $thr\_reuse = |values|.min()$;
5 **else if** $t > T * 3/4$ ***and*** $thr\_reuse < 10 * \epsilon$ **then**
6 $values, indices =$ `ADTopk`$(G_{i,t}, k)$;
7 **else if** $(t-1)\ \%\ \tau == 0$ **then**
8 $values, indices =$ `TradTopk`$(G_{i,t}, k)$;
9 $thr = |values|.min()$;
10 $\epsilon = thr/10$;
11 **if** $|thr - thr\_reuse| < \epsilon$ **then**
12 $\tau = \tau + 1$;
13 **else if** $|thr - thr\_reuse| > 2 * \epsilon$ **then**
14 $\tau = \tau - 1$;
15 **end**
16 $thr\_reuse = thr$;
17 **else**
18 $mask = |G_{i,t}| > thr\_reuse$;
19 $values, indices = G_{i,t} \odot mask$;
20 **end**
21 $\tilde{G}_{i,t} = [values, indices]$;
22 **return** $\tilde{G}_{i,t}$

4.3 ADTopk-t: Threshold Estimation in ADTopk

Limitation of ADTopk-i**:** ADTopk-i is combination of ADTopk and the traditional Top-k to trade off the convergence accuracy and the convergence speed. However, ADTopk-i spends half of the epochs in performing the traditional Top-k and limits the training throughput, which may not an optimal.

Design idea: As stated in §3.1, the traditional Top-k significantly reduces the convergence accuracy, so we aim to reduce the number of the traditional Top-k in ADTopk-i while ensuring a fast convergence. Surprisingly, Figure 7 shows that the average magnitudes for the traditional Top-k and ADTopk are similar and small when the model converges during the late epochs (about from the 50^{th} epoch), which indicates that the k-th largest element of both the traditional Top-k and ADTopk have very close magnitude value from that time. Therefore, we hope to improve the convergence speed by using interleaving compression in the early epochs of training. In the late epochs of training, the magnitude of both ADTopk and traditional Top-k is small and similar, and the parameter update is slow, so more ADTopk is used to alleviate the low training throughput caused by traditional Top-k.

This motivates us to set the k-th largest element value as a *threshold*, which helps ADTopk-i to determine the time (e.g., the 50^{th} epoch) we can start replacing the traditional Top-k with ADTopk, with a slight magnitude loss. In addition, inspired by OkTopk [18], since the gradient distribution at the late epochs remains almost unchanged over multiple iterations, ADTopk can also periodically estimate the accurate threshold and reuse it in the following iterations within a period.

Therefore, to further reduce the sparsification overhead, we design an efficient threshold-estimation scheme for ADTopk, which utilizes the periodically estimated threshold to trigger the replacement of the traditional Top-k with ADTopk.

Design details: Algorithm 2 describes the details of ADTopk with threshold estimation (called ADTopk-t). Here, we define thr_reuse as the threshold for reusing over τ iterations. We define the threshold update bound (denoted by ϵ) to adaptively update the threshold re-estimation period τ, which is set to 5 by default. Since the training model is unstable in early iterations [14], we define the warm-up training bound (denoted by σ), and empirically we set it to 15.

Specifically, ADTopk-t has six steps: (Step 1) ADTopk-t first performs the traditional Top-k function *TradTopk* in the early iterations to select the largest k values and indicates, as well as to obtain the reuse threshold value thr_reuse (Lines 1-4); (Step 2) ADTopk-t performs the ADTopk function `ADTopk` in late iterations when the threshold thr_reuse change is small (Lines 5-6); (Step 3) ADTopk-t re-estimates the accurate threshold after each τ iteration and reuses the threshold for a specific layer in the next $\tau - 1$ iterations (Lines 7-9); (Step 4) When the difference between the latest thr and the previously estimated threshold thr is less than ϵ or exceeds $2 * \epsilon$, we update τ (Lines 10-15); (Step 5) We leverage the estimated threshold to approximate the traditional Top-k elements selection (Lines 16-20); (Step 6) We finally return the compressed gradient $\tilde{G}_{i,t}$ (denoted as values and indices) for synchronized communication across workers (Lines 21-22). The time overhead of threshold estimation is negligible, for the larger value of τ makes the threshold update less frequent. The threshold selection is very GPU friendly with a time complexity of only $O(N)$.

4.4 Discussion

Compression overhead: ADTopk enables all-dimension and layer-wise sparsification of the gradient tensor, but as the size of the DNN increases, the overhead generated by the layer-wise compression becomes larger, which may impair the training performance. It is because each sparsification compression includes a fixed overhead to launch and execute kernels in CUDA (called startup overhead), which is not negligible even for small tensor sizes [47]. One popular way to address the compression overhead issue is to merge multiple gradient tensors into the buffer [37], so we can use the similar way that allows ADTopk to compress the gradient in the buffer, so as to reduce the compression startup overhead.

Scalability: Although ADTopk outperforms the traditional Top-k in increasing the convergence accuracy and training throughput, it still suffers from the same scalability issues as the traditional Top-k. The main reason is that Top-k builds up the gradient from aggregated elements of multiple workers (called *gradient build-up*), such that the number of aggregated gradient elements in the communication becomes larger than that selected by each worker [7, 55]. To improve scalability, ADTopk can leverage similarity in the gradient distribution among workers to select only one worker's Top-k indices in each iteration, as proposed in [7], to reduce the communication overhead during gradient build-up to improve scalability.

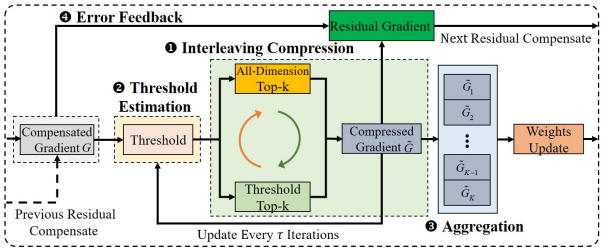


Figure 8: The workflow of the ADTopk.

Table 1: Tasks, models and datasets used for evaluation.

Tasks	Models	Datasets	Model Size
Image Classification	ResNet-50 [15]	Cifar-100 [17]	98MB
	ResNet-101 [15]	ImageNet [8]	170MB
	VGG-16 [40]	Cifar-100 [17]	528MB
	VGG-19 [40]	ImageNet [8]	548MB
Natural Language Processing	LSTM [23]	WikiText-2 [24]	328MB
	BERT-base [10]	SQuAD [28]	420MB

5 IMPLEMENTATION

We prototype our system atop Horovod [33], a popular distributed DNN training framework. Based on Horovod, we implement ADTopk, which mainly consists of four main modules, an interleaving compression module (i.e., ADTopk-i), a threshold estimation sparsification module (i.e., ADTopk-t), a communication and aggregation module and a residual gradient error feedback module.

We implement a gradient sparsification compression library based on PyTorch [26], including DGC [19], Gaussiank [36], Redsync [12], OkTopk [18], SIDCo [22], and our ADTopk. We also implement three collective communication primitives: `Allreduce`, `Allgather` and `AllgatherFast`, based on the NCCL communication library to achieve efficient collective communication.

Our prototype system has about 6.4K SLoCs in Python. Figure 8 shows the workflow of our ADTopk. The specific implementation of each module is as follows:

Interleaving compression module. We implement two compression APIs: `ADTopk` and `TradTopk`. The `ADTopk` takes the original gradient matrix as input, and uses `torch.topk` function [26] as the basic compressor to select the k largest elements from all input dimensions. The number of elements selected on each dimension is k divided by the number of input dimensions, which is at least one. The `TradTopk` first flattens the original gradient into a single one-dimensional tensor, and then it leverages `torch.topk` to select the k largest elements (including values and indices) in the gradient tensor. Note that we use threshold-based sparsification in ADTopk-t to replace the `TradTopk`, called `ThresholdTopk`.

We also implement a `decompress` function, which restores the compressed elements into a tensor of the size of the original gradient, then uses the `scatter` function [26] to fill missing elements with zeros, and reshapes the tensor to the original form. To preserve the form of the original gradient for decompression, we add a specific parameter `ctx` to store the original gradient tensor's shape and number of elements.

Threshold estimation sparsification module. In this module, we implement a `ThresholdTopk` function to perform the ADTopk function in later iterations depending on the threshold and replace the `TradTopk`. The `ThresholdTopk` function consists of a `monitor` and an `updater`. The `monitor` is designed to monitor changes in the threshold and iteration. The `updater` periodically updates the threshold. Specifically, The `updater` uses `TradTopk` to calculate the k-th largest element as the threshold after every τ iteration while also dynamically adapting the value of τ based on the threshold size. Note that the `updater` uses `TradTopk` to sort the gradient values and returns the k-th largest value as the accurate threshold. The `monitor` employs the threshold size as a crucial parameter to use `ADTopk` often in late iterations.

Communication and aggregation module. To enable efficient communication and aggregation of the compressed gradient, we rewrite the core component `DistributedOptimizer` of Horovod. In `DistributedOptimizer`, we provide three collective communication primitives APIs for practitioners, including `Allreduce`, `Allgather`, and `AllgatherFast`, specified as follows:

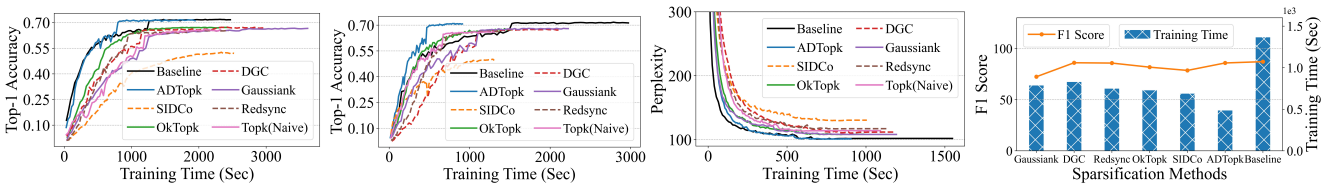
`Allreduce`: We use the `allreduce_async_` function in Horovod to design `Allreduce` collective communication primitive to execute the communication and aggregation in the non-compression baseline. `Allreduce` is the most efficient operation, but it is only suitable for the non-compression scenario. The main limitations in the underlying implementation are that it does not support *sparse* gradient [13] since it requires the same data type and dimension in the form of the compressed gradients.

`Allgather`: We leverage `allgather_async_` function in Horovod to design `Allgather` primitive to execute the communication and aggregation in ADTopk and other state-of-the-art sparsification compression methods. It does not perform any aggregation, only supports input gradients of different forms, and is well suited for sparse sparsification when different nodes select gradient elements at non-overlapping indices. On the backend, these are implemented by several alternate libraries: OpenMPI and NVIDIA NCCL.

`AllgatherFast`: We also implement another alternative primitive called `AllgatherFast`, which speeds up the communication of `Allgather` by eliminating the gradient split step. In the gradient aggregation, we use the `synchronize` function to enforce synchronization of gradient communication tasks for each parameter and decompress the aggregated gradients based on the communication type and compressor. We use `Average` for the reduction operation of aggregation.

Note that, in the gradient communication, we first check if the gradient compression and error feedback are required, and then based on the communication type defined in the parameters, we call the specific communication API (e.g., `Allreduce` for non-compression) to perform the gradient communication.

Residual gradient error feedback module. In this module, we implement an error feedback API, similar to grace [51], including a `memory.compensate` function that compensates at each iteration the node-local generated gradient with the previous iteration's error feedback and a `memory.update` function that calculates the difference between the compensated gradient and the compressed gradient to update the residual and store it in memory.



(a) ResNet-50 on Cifar-100 (b) VGG-16 on Cifar-100 (c) LSTM on WikiText-2 (d) BERT-base on SQuAD

Figure 9: Experiment 1: Convergence accuracy of DNN training tasks with PCIe-only GPU machines and 25Gbps cross-machine Ethernet.

6 EVALUATION

6.1 Experiment Setup

Testbeds. We carried out our experiments on two testbeds: 1) 16 Alibaba cloud ECS instances with 64GPU connected by a 32Gbps Ethernet, each instance has a 12vCPU (92GiB) and 4 NVIDIA V100S GPU (32GB) with NVLink; 2) a local GPU cluster of 8 nodes. Each node is equipped with one NVIDIA V100S GPU (32 GB), two 12-core Intel XEON Silver 4214 CPUs, and is connected by a 25Gbps Ethernet. ECS instances and local nodes run Ubuntu 20.04, with software libraries including CUDA-11.6, OpenMPI-4.0.3, NCCL-2.8.3, PyTorch-1.13.1, and Horovod-0.27.0.

Workloads. We evaluate six widely-used DNN tasks (four from image classification and two from natural language processing) for six models on four datasets, which are listed in Table 1. We set batch sizes and learning rates across different models similar to the studies [5, 10, 13, 23]. Note that to utilize GPU resources efficiently, we set the batch size to be constant and moderate, which does not increase with the number of workers since a too large batch size will cause convergence problems [31].

Baseline and comparative approaches. We use a compressed communication framework, Horovod [33], with PyTorch [26] and NCCL Ring-Allreduce as a non-compression baseline approach, called *Baseline*. We compare ADTopk with five state-of-the-art sparsification compression approaches with density = 0.01, including DGC [19], Gaussiank [36], Redsync [12], OkTopk [18], SIDCo [22], and Traditional Top-k (also called Topk(Naive)). We name our proposed approaches in §4.1-§4.3 as ADTopk, ADTopk-i and ADTopk-t, respectively. Note that the above approaches have been highly optimized in our implementations (see §5).

Performance metrics. We report two performance metrics in our experiments: the convergence accuracy and the training throughput. The test accuracy achieved by the converged model. We use the Top-1 accuracy for image classification models, Perplexity (ppl) for LSTM and F1 score for BERT-base as evaluation metrics, respectively. We evaluate the training throughput based on the number of trained samples (images for image classification and sequences for natural language processing) per second [5].

6.2 Experiments

Experiment 1 (Convergence accuracy): We evaluate the convergence accuracy of ADTopk and state-of-the-art methods (density=0.01 with error feedback) using real-world training tasks listed in Table 1 on a local cluster with 8 PCIe-based GPUs. To provide a fair comparison of all methods, we have the same settings for ADTopk and other state-of-the-arts.

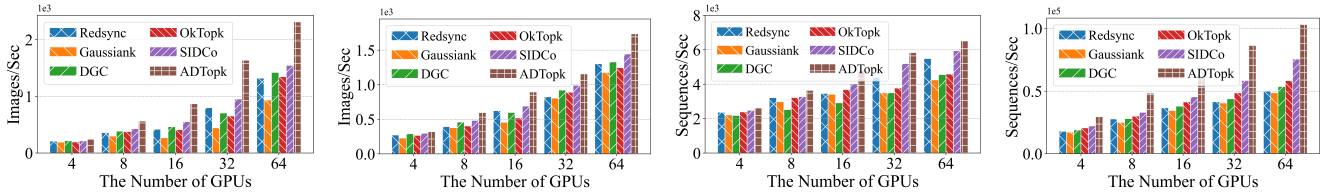
Figure 9 shows that the convergence accuracy achieved by ADTopk is very close to the Baseline (non-compression) and higher than the other state-of-the-arts on four different training tasks. Note that we train the same number of epochs for different compression methods on each training task to obtain training time and convergence accuracy. We see that:

Image classification: Figure 9(a) and (b) for ResNet-50 and VGG-16 on Cifar-100, respectively, show that the Top-1 accuracy of ADTopk achieves 71.41% and 70.69%, which are slightly lower than that of Baseline (i.e., 71.73% and 71.38%), but larger than other state-of-the-art sparsification methods. We can see that SIDCo, OkTopk, DGC, Gaussiank, Redsync, and Topk(Naive) only achieve 52.20%, 67.18%, 66.93%, 66.54%, 65.20%, and 65.43% Top-1 accuracy on ResNet-50, respectively, and 49.84%, 67.45%, 67.37%, 67.94%, 66.31%, and 65.96% Top-1 accuracy on VGG-16.

Natural language processing: Figure 9(c) on WikiText-2 for LSTM shows that ADTopk maintains the same convergence accuracy as non-compression Baseline. For example, Baseline, ADTopk have an almost identical Perplexity of 101, lower than SIDCo (130), OkTopk (108), DGC (111), Gaussiank (108), Redsync (117), and Topk(Naive) (114), which indicates that ADTopk improves convergence performance by preserving more dimensions of the LSTM (e.g., hidden size). Note that the smaller the Perplexity, the higher prediction accuracy of LSTM training Wikitext-2.

Figure 9(d) shows that ADTopk has the same converged F1 score (86%) as non-compression Baseline, DGC, and Redsync. The reason is that BERT-base is usually well pre-trained on a large dataset (like Wikipedia) and hence it only needs some small tuning for various tasks to obtain a high accuracy. On the contrary, Gaussiank only achieves a lower F1 score (72%), probably because the gradient distribution of BERT-base differs significantly from the Gaussian distribution, thus estimating inaccurate thresholds leading to poorer performance.

Discussion. Overall, ADTopk achieves convergence accuracy close to the Baseline, which confirms our main idea of retaining the gradient's all dimensions during sparsification, thus improving the convergence accuracy. In addition, we can see that ADTopk also has faster convergence (shorter training time) than other state-of-the-art sparsification compression methods. For example, the training time of ADTopk is less than other state-of-the-arts by 1.12×-1.74× and 1.32×-2.44× in Figure 9(a) and (b), respectively. ADTopk reduces the training time on LSTM and BERT-base by 1.21×-1.28× and 1.41×-1.70×, respectively. The reason is that ADTopk's independent sparsification of each dimension improves GPU core parallelism and thus reduces the training time. Note the sudden change in convergence accuracy in Figure 9, which is because we use the



(a) ResNet-101 on ImageNet (b) VGG-19 on ImageNet (c) LSTM on WikiText-2 (d) BERT-base on SQuAD

Figure 10: Experiment 2: Throughput of DNN training tasks with NVLink-based GPU machines and 32Gbps cross-machine Ethernet.

same warm-up learning rate and learning rate decay as in the non-compression Baseline.

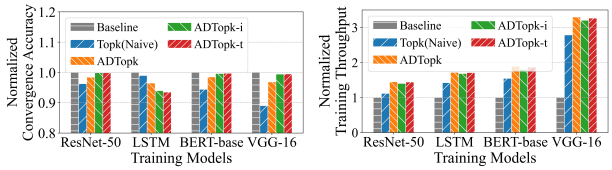
Experiment 2 (Training throughput): We evaluate the training throughput of ADTopk and state-of-the-art methods using real-world training tasks listed in Table 1 on an Alibaba Cloud cluster with 64 NVLink-based GPUs. To provide a fair comparison of similar methods, we have the same settings for ADTopk and other state-of-the-art sparsification compression methods. Figure 10 shows that the training throughput achieved by ADTopk is much higher than the other state-of-the-arts. We see that:

Image classification: Figure 10(a) and (b) for ResNet-101 and VGG-19 on ImageNet, respectively, show that the training throughput of ADTopk outperforms other sparsification compression methods by 14.21%-268.11% and 22.15%-240.02%, respectively, versus different GPU cluster sizes. Unlike the limited training throughput of the latest open-sourced versions of state-of-the-art compression methods, ADTopk improves the training throughput on multiple training tasks. For example, with 64 GPUs, for ResNet-101 on ImageNet, ADTopk outperforms SIDCo, OkTopk, Gaussiank, DGC and Redsync by 49.51%, 71.69%, 147.47%, 62.72%, and 75.15%, respectively. For VGG-19 on ImageNet, ADTopk outperforms SIDCo, OkTopk, Gaussiank, DGC, and Redsync by 20.08%, 39.37%, 47.45%, 30.5% and 33.05%, respectively.

For ResNet-101 on ImageNet, we find that the training throughput of OkTopk, Gaussiank, DGC, and Redsync are all lower than ADTopk, which is becoming increasingly prominent with the number of GPU nodes. The reason is that Gaussiank and Redsync need to adjust the thresholds frequently, which causes a large extra time overhead; DGC needs to perform two Top-k operations, which also incurs more time overhead. It implies that when the GPU cluster size expands, the communication overhead of the communication-intensive models increases. Nevertheless, ADTopk always performs the best no matter how many GPU nodes, which confirms the decent GPU core parallelism of ADTopk.

Natural language processing: For LSTM on WikiText-2, Figure 10(c) shows that ADTopk improves training throughput by 12.28%-105.02% versus different GPU cluster sizes. Specifically, the training throughput of ADTopk outperforms SIDCo, OkTopk, Gaussiank, DGC, and Redsync by 20.39%, 45.64%, 53.51%, 43.11%, and 18.86%, respectively. We train WikiText-2 using the standard LSTM model, which has fewer parameters and gradients overall (only 11 layers), but ADTopk still has a higher throughput by increasing the parallelism of the GPU cores during compression.

Figure 10 (d) for BERT-base on SQuAD, shows that ADTopk improves training throughput by 22.08%-293.49% versus different GPU cluster sizes. Specifically, ADTopk outperforms SIDCo, OkTopk, Gaussiank, DGC, and Redsync by 47.80%, 109.54%, 132.65%, 110.97%, and 126.86%, respectively. BERT-base is a computation-intensive language model, and ADTopk still maintains a high training throughput.



(a) Normalized accuracy (b) Normalized throughput

Figure 11: Experiment 3: Normalized convergence accuracy and training throughput for various optimizations of ADTopk**.**

Discussion. We can see that the training throughput for Gaussiank, DGC, and Redsync does not improve much with the number of GPUs. It is because frequent calculation of thresholds by workers in dynamic threshold adjustment mechanisms can incur additional computational overhead, which can negatively affect training efficiency. OkTopk realizes global compression that merges the gradients of all layers into a single buffer, but waiting on the global gradient prevents opportunities to overlap communication with compute, which inflates the impact of network bottlenecks. ADTopk's training throughout still scales with the number of GPUs. The reason is that our ADTopk realizes layer-wise gradient compression, which avoids the long waiting time of global compression, while ADTopk improves the parallelism of GPU cores by realizing all-dimensional compression within a single gradient tensor, thus increasing the training throughput.

In addition, we find that these sparsification compression methods also suffer from problems such as gradient build-up or inappropriate threshold, which hinder the scalability of the distributed training because the communication traffic increases as the cluster scales out[4, 7, 54–56]. We leave scalable gradient sparsification as future work to further optimize ADTopk in our distributed DNN training system.

Experiment 3 (Effectiveness of various optimizations): To verify the effectiveness of our proposed method, we evaluate the individual performance gains of the various compression optimizations introduced for ADTopk. We illustrate the convergence accuracy and the training throughput when enabling optimization one by one for four training tasks, as shown in Figure 11.

Impact of ADTopk (§4.1). First, we evaluate the case where only all-dimension Top-k is used, called ADTopk. As shown in Figure 11(a)

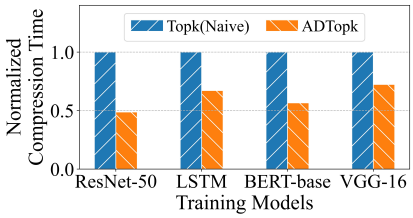
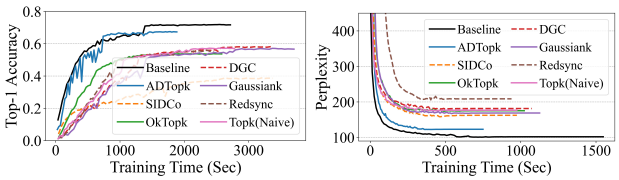


Figure 12: Experiment 4: Normalized layer-wise gradient sparsification compression time of ADTopk on four training models.



(a) ResNet-50 on Cifar-100 (b) LSTM on WikiText-2

Figure 13: Experiment 5: Convergence accuracy of the two DNN training tasks without error feedback.

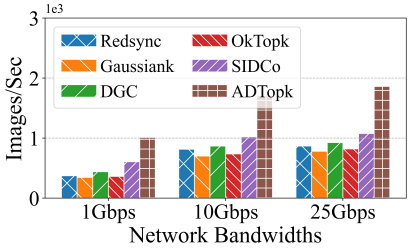


Figure 14: Experiment 6: Training performance comparison using different network bandwidths.

and (b), compared to the Topk(Naive), ADTopk improves the convergence accuracy by 1.51%, 2.63%, 5.24%, and 2.35%, respectively, and increases the training throughput by 24.0%, 5.8%, 8.29%, and 5.13%, respectively. On the one hand, ADTopk selects gradients from each dimension of the gradient, which increases the proportion of small-magnitude dimension selection, such that more dimensions can be covered to enrich the features extracted by the model, thus improving the convergence accuracy of the model. On the other hand, selecting gradients within each dimension independently increases the GPU core parallelism and improves the training throughput.

Impact of ADTopk-i (§4.2). Next, we evaluate the interleaving compression scheme in the ADTopk, called ADTopk-i. As shown in Figure 11(a), the ADTopk-i improves the convergence accuracy after reintroducing the Topk(Naive) in ADTopk. Compared to ADTopk, ADTopk-i further improves the convergence accuracy of ResNet-50, LSTM, BERT-base and VGG-16 by 2.89%, 2.6%, 1.5%, and 1.9%, respectively. This is because ADTopk only uses all-dimension sparsification and hence selects a smaller magnitude of gradients (i.e., not the largest k elements of the entire gradient tensor) at each iteration, leading to slower parameter updates and trapping in local minima; in contrast, ADTopk-i compensates for this limitation by reintroducing the Topk(Naive) to form interleaving compression. Therefore, the ADTopk-i takes the advantages of both ADTopk and the Topk(Naive).

Impact of ADTopk-t (§4.3). Finally, we evaluate the threshold estimation in ADTopk, called ADTopk-t. As shown in Figure 11(b), we see that, compared to ADTopk-i, ADTopk-t further improves the training throughput of ResNet-50, LSTM, BERT-base and VGG-16 by 3.0%, 1.2%, 3.1%, and 1.8%, respectively, while achieving almost the same convergence accuracy as ADTopk-i in Figure 11(a). This is because when we reintroduce the Topk(Naive) in the interleaving compression scheme ADTopk-i, ADTopk-i become more computation-intensive that incurs a large compression overhead due to the involvement of the Topk(Naive). In contrast, ADTopk-t alleviates the influence of the Topk(Naive) with the help of the efficiently-estimated threshold to replace the Topk(Naive) with ADTopk as far as possible in the late iterations.

Experiment 4 (Layer-wise gradient sparsification compression time): As shown in Figure 12, to show that ADTopk can increase the parallelism of GPU cores to reduce the gradient compression time, we calculate the average layer-wise compression time of ADTopk on the four training models, respectively, and normalize ADTopk based on the Topk(Naive) compression time. We can see that the average compression time of ADTopk is smaller than Topk(Naive) on all four training tasks. The sparsification compression time of ADTopk is reduced by 51.23%, 32.85%, 43.41%, and 27.65% on ResNet-50, LSTM, BERT-base and VGG-16, respectively. ADTopk's compression time on ResNet-50 and BERT-base is reduced more because they contain more layers, and ADTopk's increased parallelism of GPU cores enables faster compression, thus reducing the compression time.

Experiment 5 (Impacts of error feedback): Experiment 1 shows the results with error feedback, and Figure 13 shows the convergence accuracy of the two training tasks without it. For ResNet-50 on Cifar-100, Figure 13(a) shows that the Top-1 accuracy of ADTopk achieves 67.42%, which is slightly lower than that of non-compression Baseline (i.e., 71.21%), but much larger than other state-of-the-art sparsification compression methods. We can see that SIDCo, OkTopk, DGC, Gaussiank, Redsync, and Topk(Naive) only achieve 38.64%, 53.75%, 58.06%, 56.30%, 55.60%, and 57.14% Top-1 accuracy on ResNet-50, respectively. For LSTM on WikiText-2, Figure 13(b) shows that the perplexity of ADTopk is 123, which is close to the Baseline (112) and much lower than SIDCo (162), OkTopk (175), DGC (181), Gaussiank (169), Redsync (208), and Topk(Naive) (177). Overall, the state-of-the-art sparsification methods yield very low convergence accuracy without using error feedback due to dimension missing. Nevertheless, at the same density, our ADTopk still achieves higher convergence accuracy by its all-dimensional gradient elements selection scheme.

Experiment 6 (Impacts of network bandwidths): Figure 14 compares the training throughput of ADTopk and other state-of-the-art sparsification compression methods for training ResNet-50 models using different network bandwidths. We use 1Gbps, 10Gbps, and 25Gbps Ethernet as network configurations for an 8-node local GPU cluster, respectively. We can see that the training throughput of ADTopk is higher than other state-of-the-art gradient sparsification compression methods under all three network environments. Therefore, this implies that ADTopk can achieve near-optimal training throughput even in a low bandwidth network environment (e.g., 1Gbps).

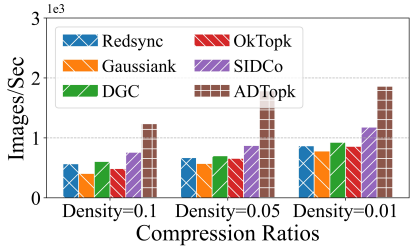


Figure 15: Experiment 7: Training performance comparison using different compression ratios.

Experiment 7 (Impacts of compression ratios): Figure 15 compares the training throughput of ADTopk and other state-of-the-art sparsification compression methods on ResNet-50 using different compression ratios (i.e., densities) with the same configurations as in Experiment 1. We can see that ADTopk outperforms Redsync, Gaussiank, DGC, OkTopk, and SIDCo by 62.60%-203.77%, 104.40%-211.16%, and 57.82%-138.36% when the density is reduced from 0.1 to 0.05 and 0.01, respectively. It implies that ADTopk still achieves higher training throughput by its decent parallelism, even if the density is reduced to a certain level (e.g., density=0.01) which often makes the communication no longer a bottleneck in data-parallel distributed DNN training systems. Note that in practice, smaller densities may impair convergence.

7 CONCLUSION

ADTopk is a high-performance gradient sparsification compression that addresses the challenges of dimension missing and single global sorting caused by the traditional Top-k. ADTopk proposes an all-dimension-based elements selection, which improves convergence accuracy by retaining more dimension information of the gradient tensor and improves the training throughput by increasing the parallelism of GPU cores. ADTopk also proposes an interleaving compression scheme to accelerate the model convergence as well as an efficient threshold estimation scheme. While maintaining almost the same convergence accuracy as the non-compression baseline, ADTopk improves the training throughput by 12.28%-293.49% compared to the state-of-the-art sparsification compression methods for six popular DNN training tasks on a local cluster and in a public cloud, respectively.

ACKNOWLEDGMENTS

This work was supported in part by Shenzhen Science and Technology Program+JCYJ 20220530161006015 and Key Laboratory of Information Storage System Ministry of Education of China.

REFERENCES

[1] Saurabh Agarwal, Hongyi Wang, Kangwook Lee, Shivaram Venkataraman, and Dimitris Papailiopoulos. 2021. Adaptive gradient communication via critical learning regime identification. In *Proceedings of Machine Learning and Systems*. 55–80.

[2] Alham Fikri Aji and Kenneth Heafield. 2017. Sparse communication for distributed gradient descent. *arXiv preprint arXiv:1704.05021* (2017).

[3] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2017. QSGD: Communication-efficient SGD via gradient quantization and encoding. In *Advances in Neural Information Processing Systems*. 8024–8035.

[4] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. 2022. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 472–487.

[5] Youhui Bai, Cheng Li, Quan Zhou, Jun Yi, Ping Gong, Feng Yan, Ruichuan Chen, and Yinlong Xu. 2021. Gradient compression supercharged high-performance data parallel dnn training. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 359–375.

[6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*. 1877–1901.

[7] Chia-Yu Chen, Jiamin Ni, Songtao Lu, Xiaodong Cui, Pin-Yu Chen, Xiao Sun, Naigang Wang, Swagath Venkataramani, Vijayalakshmi Srinivasan, Wei Zhang, et al. 2020. ScaleCom: Scalable Sparsified Gradient Compression for Communication-Efficient Distributed Training. In *Advances in Neural Information Processing Systems*. 13551–13563.

[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] Tim Dettmers, Mike Lewis, Sam Shleifer, and Luke Zettlemoyer. 2022. 8-bit Optimizers via Block-wise Quantization. In *International Conference on Learning Representations*.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[11] Fartash Faghri, Iman Tabrizian, Ilia Markov, Dan Alistarh, Daniel M Roy, and Ali Ramezani-Kebrya. 2020. Adaptive gradient quantization for data-parallel sgd. In *Advances in Neural Information Processing Systems*. 3174–3185.

[12] Jiarui Fang, Haohuan Fu, Guangwen Yang, and Cho-Jui Hsieh. 2019. RedSync: reducing synchronization bandwidth for distributed deep learning training system. *J. Parallel and Distrib. Comput.* (2019), 30–39.

[13] Jiawei Fei, Chen-Yu Ho, Atal N Sahu, Marco Canini, and Amedeo Sapio. 2021. Efficient sparse collective communication and its application to accelerate distributed deep learning. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 676–691.

[14] 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).

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[16] Peng Jiang and Gagan Agrawal. 2018. A Linear Speedup Analysis of Distributed Deep Learning with Sparse and Quantized Communication. In *Advances in Neural Information Processing Systems*, Vol. 31. Curran Associates, Inc., 2530–2541.

[17] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. *Master's thesis, University of Tront* (2009).

[18] Shigang Li and Torsten Hoefler. 2022. Near-optimal sparse allreduce for distributed deep learning. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 135–149.

[19] 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).

[20] Liang Luo, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. 2018. Parameter hub: a rack-scale parameter server for distributed deep neural network training. In *Proceedings of the ACM Symposium on Cloud Computing*. 41–54.

[21] Xu Luo, Jing Xu, and Zenglin Xu. 2022. Channel importance matters in few-shot image classification. In *International conference on machine learning*. PMLR, 14542–14559.

[22] Ahmed M Abdelmoniem, Ahmed Elzanaty, Mohamed-Slim Alouini, and Marco Canini. 2021. An efficient statistical-based gradient compression technique for distributed training systems. In *Proceedings of Machine Learning and Systems*. 297–322.

[23] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. 2017. Regularizing and optimizing LSTM language models. *arXiv preprint arXiv:1708.02182* (2017).

[24] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843* (2016).

[25] NVIDIA. 2023. The Developer Tools Documentation of NVIDIA Nsight Systems. https://docs.nvidia.com/nsight-systems. Online accessed on Sept-2023.

[26] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*. 8026–8037.

[27] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. 2019. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 16–29.

[28] Pranav Rajpurkar, Robin Jia, and Percy Liang. 2018. Know what you don't know: Unanswerable questions for SQuAD. *arXiv preprint arXiv:1806.03822* (2018).

[29] Cèdric Renggli, Saleh Ashkboos, Mehdi Aghagolzadeh, Dan Alistarh, and Torsten Hoefler. 2019. SparCML: High-performance sparse communication for machine learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[30] Atal Sahu, Aritra Dutta, Ahmed M Abdelmoniem, Trambak Banerjee, Marco Canini, and Panos Kalnis. 2021. Rethinking gradient sparsification as total error minimization. In *Advances in Neural Information Processing Systems*. 8133–8146.

[31] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan RK Ports, and Peter Richtárik. 2019. Scaling distributed machine learning with in-network aggregation. *arXiv preprint arXiv:1903.06701* (2019).

[32] 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*.

[33] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[34] Anil Shanbhag, Holger Pirk, and Samuel Madden. 2018. Efficient top-k query processing on massively parallel hardware. In *Proceedings of the 2018 International Conference on Management of Data*. 1557–1570.

[35] Wenqi Shao, Shitao Tang, Xingang Pan, Ping Tan, Xiaogang Wang, and Ping Luo. 2020. Channel equilibrium networks for learning deep representation. In *International Conference on Machine Learning*. PMLR, 8645–8654.

[36] Shaohuai Shi, Xiaowen Chu, Ka Chun Cheung, and Simon See. 2019. Understanding top-k sparsification in distributed deep learning. *arXiv preprint arXiv:1911.08772* (2019).

[37] Shaohuai Shi, Xiaowen Chu, and Bo Li. 2021. Exploiting simultaneous communications to accelerate data parallel distributed deep learning. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE, 1–10.

[38] Shaohuai Shi, Zhenheng Tang, Qiang Wang, Kaiyong Zhao, and Xiaowen Chu. 2019. Layer-wise adaptive gradient sparsification for distributed deep learning with convergence guarantees. *arXiv preprint arXiv:1911.08727* (2019).

[39] Shaohuai Shi, Qiang Wang, Kaiyong Zhao, Zhenheng Tang, Yuxin Wang, Xiang Huang, and Xiaowen Chu. 2019. A distributed synchronous SGD algorithm with global top-k sparsification for low bandwidth networks. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2238–2247.

[40] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[41] Sebastian U Stich, Jean-Baptiste Cordonnier, and Martin Jaggi. 2018. Sparsified SGD with memory. In *Advances in Neural Information Processing Systems*. 4452–4463.

[42] Hanlin Tang, Shaoduo Gan, Ce Zhang, Tong Zhang, and Ji Liu. 2018. Communication compression for decentralized training. In *Advances in Neural Information Processing Systems*. 7663–7673.

[43] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*. 6000–6010.

[44] Hongyi Wang, Saurabh Agarwal, and Dimitris Papailiopoulos. 2021. Pufferfish: Communication-efficient models at no extra cost. In *Proceedings of Machine Learning and Systems*. 365–386.

[45] Linnan Wang, Wei Wu, Junyu Zhang, Hang Liu, George Bosilca, Maurice Herlihy, and Rodrigo Fonseca. 2020. FFT-based gradient sparsification for the distributed training of deep neural networks. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*. 113–124.

[46] Zhuang Wang, Haibin Lin, Yibo Zhu, and T. S. Eugene Ng. 2023. Hi-Speed DNN Training with Espresso: Unleashing the Full Potential of Gradient Compression with Near-Optimal Usage Strategies. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys '23)*. 867–882.

[47] Zhuang Wang, Xinyu Crystal Wu, Zhaozhuo Xu, and TS Eugene Ng. 2023. CUPCAKE: ACOMPRESSION OPTIMIZER FOR SCALABLE COMMUNICATION-EFFICIENT DISTRIBUTED TRAINING. In *Proceedings of the Sixth Conference on Machine Learning and Systems (MLSys' 23)*. Proceedings of the Sixth Conference on Machine Learning and Systems (MLSys' 23).

[48] Jianqiao Wangni, Jialei Wang, Ji Liu, and Tong Zhang. 2018. Gradient sparsification for communication-efficient distributed optimization. In *Advances in Neural Information Processing Systems*. 1306–1316.

[49] Donglei Wu, Weihao Yang, Cai Deng, Xiangyu Zou, Shiyi Li, and Wen Xia. 2023. BIRD: A Lightweight and Adaptive Compressor for Communication-Efficient Distributed Learning Using Tensor-wise Bi-Random Sampling. In *2023 IEEE 41st International Conference on Computer Design (ICCD)*. IEEE, 605–613.

[50] Jiaxiang Wu, Weidong Huang, Junzhou Huang, and Tong Zhang. 2018. Error compensated quantized SGD and its applications to large-scale distributed optimization. In *International Conference on Machine Learning*. PMLR, 5325–5333.

[51] Hang Xu, Chen-Yu Ho, Ahmed M Abdelmoniem, Aritra Dutta, El Houcine Bergou, Konstantinos Karatsenidis, Marco Canini, and Panos Kalnis. 2021. GRACE: A compressed communication framework for distributed machine learning. In *2021 IEEE 41st international conference on distributed computing systems (ICDCS)*. IEEE, 561–572.

[52] Hang Xu, Kelly Kostopoulou, Aritra Dutta, Xin Li, Alexandros Ntoulas, and Panos Kalnis. 2021. Deepreduce: A sparse-tensor communication framework for federated deep learning. In *Advances in Neural Information Processing Systems*. 21150–21163.

[53] Guangfeng Yan, Shao-Lun Huang, Tian Lan, and Linqi Song. 2021. DQ-SGD: Dynamic quantization in SGD for communication-efficient distributed learning. In *2021 IEEE 18th International Conference on Mobile Ad Hoc and Smart Systems (MASS)*. IEEE, 136–144.

[54] Daegun Yoon and Sangyoon Oh. 2023. DEFT: Exploiting Gradient Norm Difference between Model Layers for Scalable Gradient Sparsification. In *Proceedings of the 52nd International Conference on Parallel Processing*. 746–755.

[55] Daegun Yoon and Sangyoon Oh. 2023. MiCRO: Near-Zero Cost Gradient Sparsification for Scaling and Accelerating Distributed DNN Training. *arXiv preprint arXiv:2310.00967* (2023).

[56] Daegun Yoon and Sangyoon Oh. 2024. Preserving Near-Optimal Gradient Sparsification Cost for Scalable Distributed Deep Learning. *arXiv preprint arXiv:2402.13781* (2024).