Saving Memory via Residual Reduction for DNN Training with Compressed Communication

Xinjue Zheng, Zhangqiang Ming, Yuchong Hu(✉), Chenxuan Yao, Wenxiang Zhou, Rui Wang, Xun Chen, and Dan Feng

Huazhong University of Science and Technology, Wuhan, China
{zhengxinjue,zqming,yuchonghu,deadfffool,wxzhoucs,akane,xunchen,dfeng}@hust.edu.cn

Abstract. Deep neural network (DNN) training systems suffer from communication bottlenecks among workers for gradient synchronization. Gradient compression reduces this overhead but impacts model accuracy, prompting the use of residuals to compensate for the loss. However, we observe that these residuals consume significant GPU memory but fortunately can be reduced with tiny accuracy impact. We propose ResiReduce, a memory-saving mechanism that reuses residuals across similar layers and applies strategic compression within specific layers. Experiments on local and cloud clusters show that ResiReduce can reduce the memory footprint of the model states by up to 15.7% while preserving the model accuracy and training throughput.

1 Introduction

Deep neural networks (DNNs) have seen rapid development in natural language processing (NLP) [1,5] and computer vision (CV) [8,20]. As the dataset in today's DNN training is continuously growing, DNN training systems often enable each node (or *worker*) to have a subset of the dataset to compute the model updates (i.e., *gradients*) based on its own data, and then exchange the gradients across workers through synchronization strategies to update the model. Nevertheless, the gradient communication caused by exchanging gradients often takes up a significant portion of the overall training time (as high as 76.8%) [4], which has become one of the major bottlenecks in DNN training [21,23,25].

One common way to address the above communication bottleneck is to adopt *compressed communication* (or called *gradient compression*) to reduce communication overhead [2,11,18], at the expense of losing accuracy (due to compressed gradients). To ensure accuracy, *residuals* that represent the difference between the original and compressed gradient are often stored in memory to compensate the loss, also known as *error feedback* [13,18,24], which has been applied in many DNN training systems (e.g., Horovod [19], Hipress [4], and Grace [25]).

However, we observe that residuals often incur significant memory overhead; for example, the residuals account for 23.56% of the VGG-19 model state in terms of the GPU memory consumption (see Observation #1 in Sect. 3). Fortunately, we also observe that, via a tradeoff curve between the residual memory overhead

W. E. Nagel et al. (Eds.): Euro-Par 2025, LNCS 15901, pp. 221–235, 2026.
https://doi.org/10.1007/978-3-031-99857-7_16

and accuracy, residuals can be reduced moderately in memory at the expense of only a tiny accuracy loss (see Observation #2 in Sect. 3).

In this paper, motivated by the above two observations, we propose a **resi**dual **reduc**tion based memory-optimizing mechanism, called ResiReduce, whose main idea is to save memory by trading a tiny sacrifice in accuracy for a significant reduction in residuals. Specifically, ResiReduce performs residual reduction in two ways that i) first reuse residuals on adjacent DNN layers that have an identical structure, and ii) further compress some DNN layers inside themselves carefully with rare accuracy loss. Our contributions includes:

- We conduct measurement analysis and give two observations to show that residuals cost a lot of memory overhead but they can be reduced with rarely affecting the accuracy. We propose ResiReduce which is the first to leverage residual reduction to save GPU memory for DNN training (Sect. 3).
- We design ResiReduce composed of i) two inter-layer residual reduction schemes (a straightforward one and an improved one) based on the similarity of adjacent residuals and ii) two intra-layer residual reduction schemes (a naive one and an improved one) by an L1-norm based strategic compression (Sect. 4).
- We implement ResiReduce atop Horovod [19] and Pytorch [14] (opensourced at https://github.com/YuchongHu/ResiReduce), and train six typical DNN models on a local cluster as well as a Cloud cluster. Experimental results show that ResiReduce can reduce the memory footprint of the model states by up to 15.7%, without damaging the accuracy and throughput (Sect. 5).

2 Background and Related Work

2.1 Basics of DNN Training

A DNN model is basically composed of multiple *layers*, which contains thousands or even millions of *parameters*. To reach the model convergence (i.e., the model training process reaches a stable state), the training process 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.

Existing DNN training systems typically distribute the training tasks across multiple *nodes* (i.e., GPUs) and employ parallelization methods to accelerate the training and handle the large-scale datasets and models [16,19]. During each iteration, the dataset is divided into multiple subsets (*mini-batches*) and allocated to various nodes. Each node computes gradients based on its mini-batch, and then synchronizes gradients with other nodes to collectively update the model parameters. The formula for updating model parameters is denoted as: $x_{t+1}^i = x_t^i - \eta \frac{1}{N} \sum_{i=1}^{N} G_t^i$, where x_t^i and G_t^i represent the model parameters and the gradients of the i-th node at the t-th iteration, respectively, η determines the step size at each iteration (called *learning rate*), and N is the number of nodes. Here, $\frac{1}{N} \sum_{i=1}^{N} G_t^i$ represents the gradient synchronization across multiple nodes.

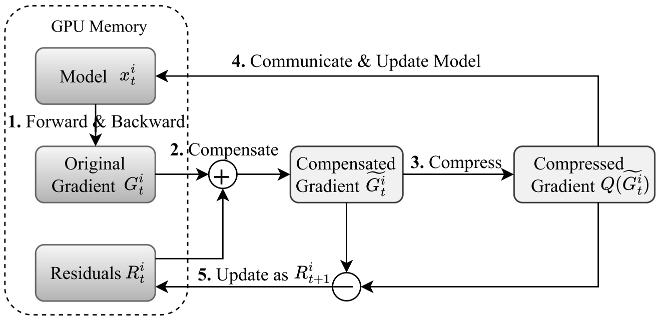

GPU Memory
Model x_t^i
1. Forward & Backward
Original Gradient G_t^i
2. Compensate
Compensated Gradient $\widetilde{G_t^i}$
3. Compress
Compressed Gradient $Q(\widetilde{G_t^i})$
4. Communicate & Update Model
Residuals R_t^i
5. Update as R_{t+1}^i

Fig. 1. Residual in Error Feedback.

2.2 Compressed Communication in DNN Training

Gradient Compression. Gradient communication between nodes during DNN training often causes significant communication overhead (accounting for as high as 76.8% of the training time [4]), which severely limits the DNN training performance. Thus, many distributed DNN training systems employ gradient compression to reduce the amount of data transmitted during gradient synchronization, which can be generally categorized into two major categories: sparsification and quantization [23], where the former [2,11] selects a subset of gradient elements to produce a sparse vector for communication, and the latter [18] lowers the number of bits of each gradient element to reduce communication volume. Since only the compressed gradient need to be transmitted, the model update equation is described as: $x_{t+1}^i = x_t^i - \eta \frac{1}{N} Q^{-1}(\sum_{i=1}^N Q(G_t^i))$, where Q denotes gradient compression (e.g., Top-k [2]), and Q^{-1} denotes gradient decompression, which reconstructs the shape and size of the original gradient.

Residual. Due to the lossy nature of compression, gradient compression can lead to a decrease in model accuracy [11,21]. Consequently, *Residual*, which is generally known as *error feedback*, is widely applied by many gradient compression methods to ensure the accuracy of the model training [2,13,18].

As shown in Fig. 1, an iteration of training with residuals is conducted with five steps. (Step 1) <u>Gradient Calculation</u>: The original gradient G_t^i is calculated through forward and backward propagation. (Step 2) <u>Gradient Compensation</u>: The residual R_t^i is accumulated with original gradient G_t^i to compensate the loss of information caused by gradient compression. The formula of compensating the gradient is: $\widetilde{G_t^i} = R_t^i + G_t^i$, where $\widetilde{G_t^i}$ represents the compensated gradient. (Step 3) <u>Gradient Compression</u>: The compensated gradients are then compressed to $Q(\widetilde{G_t^i})$. (Step 4) <u>Model update</u>: The compressed gradients are aggregated through communication and then update the parameters. (Step 5) <u>Residual Update</u>: The residuals of next iteration are updated by the difference of the compressed gradient $Q(\widetilde{G_t^i})$ and the original gradient G_t^i. The formula of updating the residual is: $R_{t+1}^i = \widetilde{G_t^i} - Q(\widetilde{G_t^i})$.

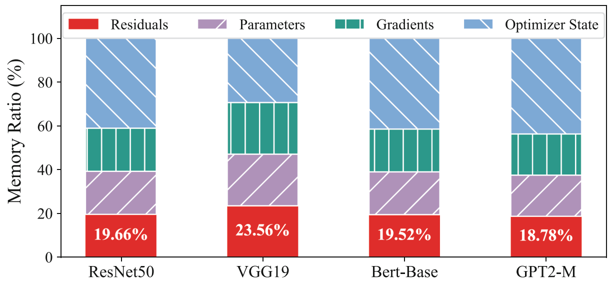


Fig. 2. Observation #1: Residual ratio of the model states.

2.3 Memory Optimization Techniques for DNN Training

Chen et al. [6] first introduced activation recomputation to recompute activations as necessary instead of storing them. To mitigate the computational overhead, Vijay et al. [9] proposed selective activation recomputation to recalculate only parts of activation data. ZeRO-DP [16] is proposed to partitions the model state instead of duplicating it to reduce redundancy. StrongHold [22] extends the maximum trainable model size by dynamically offloading data from GPU to CPU. Different from the above studies, this paper is the first to save memory by reducing residuals for distributed DNN training systems, which is orthogonal to and complements existing memory optimization techniques.

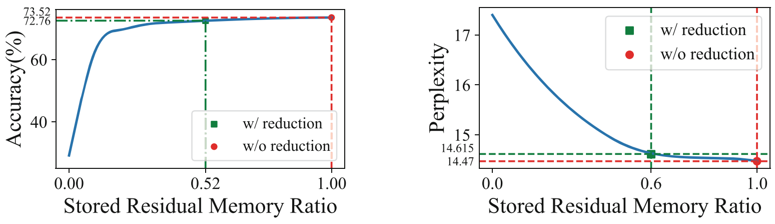
3 Motivation

We conduct measurement analysis to observe how residuals have an impact on the memory overhead and the accuracy.

Measurement Setting. We evaluate four typical training tasks (specified in Sect. 5.2) on a local cluster with 4 NVIDIA V100 GPUs connected by a 25Gbps Ethernet, including ResNet-50 [8] and VGG-19 [20] on Cifar-100 [10], BERT-base [7] on SQuAD [17], and GPT2-M [15] on WikiText-103 [12]. We apply Top-k [2] with residual as a standard compressed communication technique.

Observation #1. Residual Occupies a Significant Amount of Memory Space. Many studies [9,22] show that the *model states*, which include model parameters, gradients, residuals and optimizer states, often dominate the memory overhead when training large DNNs (e.g., the model states can account for 87.5% of the GPU memory footprint [22]). Thus, we measure the proportion of residuals of the model states to see if the residual memory overhead is significant. Figure 2 shows that residuals account for approximately 20% of the model-state memory consumption. For example, the memory footprint of residuals occupies 23.56% of the VGG-19 model states, which leads non-negligible memory overhead as the model states dominate the total memory consumption.

Observation #2. Residual Memory Can be Reduced with a Tiny Accuracy Loss. Observation #1 indicates that residuals incur significant memory



(a) Tradeoff of ResNet-50. (b) Tradeoff of GPT2-M.

Fig. 3. Observation #2: Tradeoff between residual memory and accuracy.

overhead in return for compensating the accuracy loss, so it is natural to ask if we can reduce the residual memory footprint without damaging the accuracy. To this end, we apply a layer-wise random sparsification [21] to the entire residuals in a way that sets a random subset of elements to zero to simulate residual reduction (i.e., only a portion of the residuals stored). In this way, we measure a tradeoff between the stored residual memory ratio (100% means all residuals are stored in memory) and the accuracy (i.e., Top-1 accuracy for ResNet-50 and perplexity for GPT-2). Note that the lower the perplexity is, the higher accuracy we have.

Interestingly, we observe that only storing around half of residuals in memory results in a very small loss in accuracy. As shown in Fig. 3a, storing 52% and 100% of the residuals in memory can achieve the accuracy of 72.76% and 73.52%, respectively. Similarly, in Fig. 3b, storing 60% and 100% of the residuals can achieve the perplexity of 14.61 and 14.47, respectively.

Main Idea. Observation #1 shows the memory overhead of residuals is significant. Observation #2 implies that moderately reducing residuals only slightly affects the training accuracy. These observations motivate us to propose ResiReduce, whose main idea is to leverage residual reduction to save memory without damaging the accuracy. While ResiReduce can help reduce the memory consumption of residuals, how to design ResiReduce in details still remains unexplored, which will be specified in Sect. 4.

4 Design

4.1 Insight and Design Goals

Based on the measurement setting in Sect. 3, we evaluate the probability density function (PDF) of residual values (i.e., R_t^i in Sect. 2.2) and find that multiple adjacent residual layers which have the same layer type often have similar density distribution. Note that different residual layers have different types, e.g., ResNet-50 has four types of residuals: Conv2, Conv3, Conv4 and Conv5 [8]. Specifically, Fig. 4a shows that six adjacent residual layers (Layers 0–5) that belong to the same "Conv4" layer type in ResNet-50 have almost overlapped distribution curves. Similarly, as shown in Fig. 4b, 4c and 4d, homogeneous triple

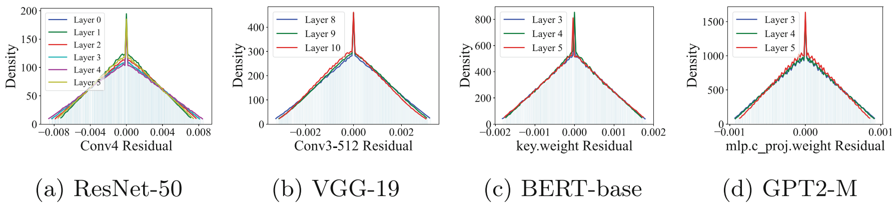


Fig. 4. Similar distribution of residuals of the same type.

adjacent residual layers in VGG-19, BERT-base, and GPT2-M exhibit near-overlapping distribution curves, demonstrating that both CNN and Transformer architectures inherently exhibit analogous distribution characteristics in adjacent residual layers.

Based on the insight above, we introduce a new concept called *reusable groups*, each consisting of multiple adjacent residual layers that belong to the same residual type, such that a reusable group can just *reuse* a single residual to replace all residuals within the group.

We design ResiReduce with the following goals based on *reusable groups*:

- **Inter-layer residual reduction:** ResiReduce reuses an average residual to replace residuals of multiple layers within each reusable group, and further proposes a weighted averaging method to improve the training accuracy (Sect. 4.2).
- **Intra-layer residual reduction:** ResiReduce additionally reduces residuals existing inside some specific layers via a simple dimension-wise compression method, and further proposes a two-dimension compression method to improve the training accuracy (Sect. 4.3).

4.2 Inter-layer Residual Reduction

ResiReduce-a: Average residual reusing of ResiReduce. A straightforward method to utilize the reusable group (Sect. 4.1) is to *average* the residuals within each reusable group, replacing all residuals with the average residual, called ResiReduce-a, making each reusable group only stores its average residual in memory, such that the residual memory overhead can be largely reduced.

Figure 5 illustrates ResiReduce-a with two reusable groups. Specifically, ResiReduce-a has four steps. (Step 1) Grouping: First, the DNN model is divided into m reusable groups based on their different layer types, with each group containing α_j $(1 \leq j \leq m)$ residual layers. (Step 2) All residual calculating: The α_j residuals of the j^{th} reusable group $(1 \leq j \leq m)$ are calculated based on Step 5 in Sect. 2.2. (Step 3) Average residual calculating: The average residual (denoted by ar_j) of the j^{th} reusable group is calculated by averaging the α_j residuals within the group, and then it is stored in GPU memory to represent all the α_j residuals within the j^{th} reusable group, meaning that the original α_j residuals do not need to be stored. (Step 4) Gradient compensation: For the j^{th} reusable group, its ar_j is reused to compensate all the α_j original gradients in the group.

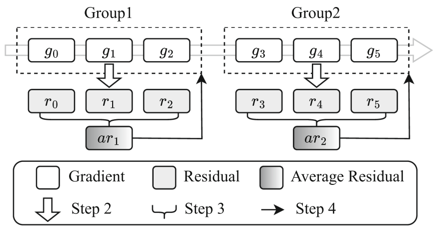


Fig. 5. An example of ResiReduce-a.

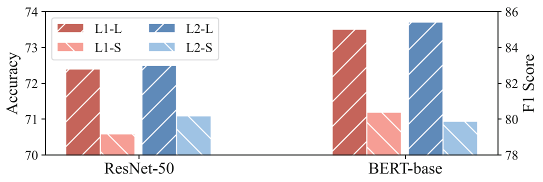


Fig. 6. The effect of significance of accuracy. Ln-L/S denotes storing residuals with the largest L-n norm (e.g., L1-L) or smallest L-n norm (e.g., L1-S).

ResiReduce-w: Weighted Average Reusing of ResiReduce. Although ResiReduce-a can significantly reduce the residual memory overhead, we find that its simple averaging operation does not consider the fact that different residuals within the same group often have different significance of accuracy.

To confirm the above fact, we evaluate different significance using the L-n norms (L1 and L2 norms) for measurement, similar to [24,26]. We observe that in the same group, the layers with the largest L1 or L2 norms have higher accuracy than those with the smallest L1 or L2 norms. Specifically, Fig. 6 shows that residuals with the largest L1 or L2 norms improve the accuracy by 1.5–5.5% compared to those with the smallest ones, indicating that the residuals within each reusable group have different significance of accuracy.

Therefore, we propose ResiReduce-w, which reuses a weighted average residual (denoted by war) instead of the simple average one ar in ResiReduce-a, where the different weights indicate the different significance of accuracy of different residuals within the group. Specifically, the formula of calculating the weight of the i^{th} residual within each reusable group is: $w_i = \phi\frac{\|r_i\|_1}{n_i} + (1-\phi)\frac{\|r_i\|_2}{\sqrt{n_i}}$, where the L1 and L2 norms jointly represent the significance of the residual r_i, $\|r\|_1$ and $\|r\|_2$ refer to the L1 and L2 norms of the residual respectively, n_i denotes the number of elements in the residual and the parameter ϕ represents the significance coefficient (which we set as 0.5 by default). Note that the division between $\|r\|_n$ and $\sqrt[n]{n_i}$ is to keep them in the same order of magnitude. Finally, we can obtain the weighted average residual of the j^{th} group is: $war_j = \frac{\sum_{i=1}^{\alpha_j} w_i r_i}{\sum_{i=1}^{\alpha_j} w_i}$.

4.3 Intra-layer Residual Reduction

Limitation of ResiReduce-w: We note that ResiReduce-w only reduces the residual memory for multiple residuals of the reusable groups. However, we find

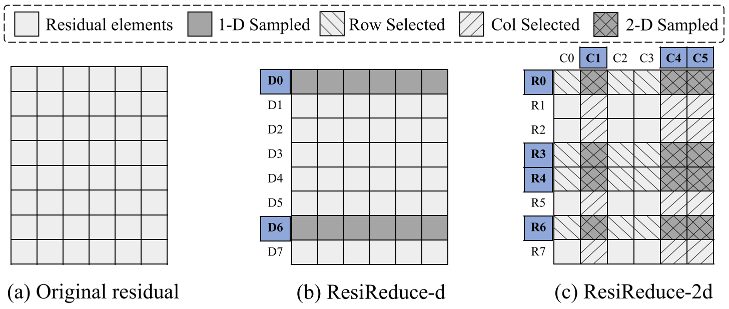


Fig. 7. Examples of ResiReduce-d and ResiReduce-2d.

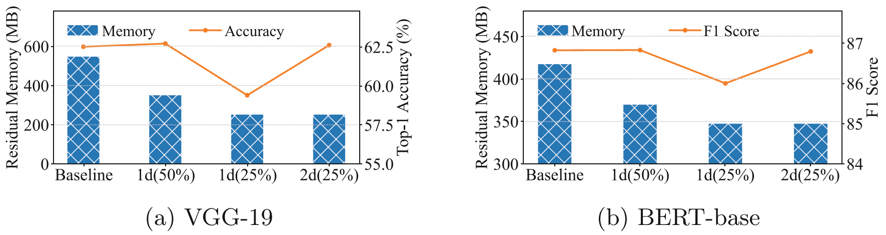


Fig. 8. Comparison between 1d and 2d at different compression ratios. 1d (50%) represents applying ResiReduce-d to achieve 50% compression.

that for some *specific layers*, each of them belongs to only one layer type (i.e., its reusable group has only one residual), so we cannot perform inter-layer reduction. For example, for the fully connected layers in VGG-19, each of them has a unique type [20], and thus can be considered one specific layer, which cannot be reduced by ResiReduce-w to save more memory space.

ResiReduce-d: Dimension-Wise Compression of ResiReduce. To handle the above limitation, a naive method is to first find the specific layers and represent them as a two-dimension tensor [24], and then individually perform dimension-wise compression on the residual for each of the specific layers, called ResiReduce-d. Specifically, the dimension-wise compression first calculates all L1 norms of all row vectors of the two-dimension tensor, and then samples the rows with the largest L1 norms, which reduces the residual itself. As illustrated in Fig. 7, the original residual corresponding to one specific layers is represented as an 8*6 tensor (Fig. 7a), and we perform L1-norm (same as [24]) to compress the specific layer (Fig. 7b) with a compression ratio of 25%; that is, we sample two out of eight rows ($2/8 = 25\%$).

ResiReduce-2d: Two-Dimension Compression of ResiReduce. We note that ResiReduce-d only samples some rows for compression and thus results in losing dimension information (i.e., other rows are not selected). Unfortunately, the above residual dimension loss may have a negative impact on the accuracy, which has been found in [13].

To verify the impact of residual dimension loss on accuracy, we evaluate the accuracy of ResiReduce-d under different residual compression ratios. We use the non-compression scheme as baseline. We consider the following specific layers: FC layers in VGG-19 [20] and WPE and WTE layers in BERT-base [7]. Figure 8 shows that the accuracy decreases as the compression ratio increases when applying ResiReduce-d. For example, as illustrated in Fig. 8a, we compress the specific layers in VGG-19 by ResiReduce-d, and the accuracy at a 25% compression ratio is 3.2% less than that at a 50% compression ratio. Figure 8b shows a similar observation. We see that the dimension loss of residuals of the specific layers has a non-negligible negative impact on accuracy.

To address the above issue, we propose ResiReduce-2d which enhances ResiReduce-d by considering the column dimension information besides the row dimension information. Specifically, we first select both row vectors and column vectors that have the largest L1 norms, and then we sample the *overlapping* elements of the above selected elements.

As illustrated in Fig. 7c, ResiReduce-2d selects four rows R_0, R_3, R_4, R_6 and three columns C_1, C_4, C_5, and finally samples 12 overlapping elements from the above selected rows and columns, which also achieves the same 25% compression ratio as ResiReduce-d in Fig. 7b, while the former introduces more dimension information, which can help improve the accuracy. As illustrated in Fig. 8, we observe that compared to ResiReduce-d at a 25% compression ratio, the accuracy of ResiReduce-2d is much higher while maintaining the compression ratio 25%.

5 Evaluation

5.1 Implementation

We prototype our system atop Horovod [19] and Pytorch [14], establishing a memory-efficient distributed DNN training framework with the inter-layer residual reusing module (i.e., ResiReduce-w) and the intra-layer residual compression module (i.e., ResiReduce-2d). We implement the gradient compression library, including Top-k [2], DGC [11] and QSGD [3] according to their open source code.

Inter-layer Residual Reusing Module. We implement the inter-layer residual reusing module with a `memory.update` function and a `memory.compensate` function, similar to grace [25]. In the `memory.update` function, we first traverse the entire DNN layer, grouping and numbering the layers of the same type. Next, we use a `dictionary` to store the group number corresponding to each layer of the model, such that the layers of the same type have the same group number. We allocate a separate buffer for each group to store the residual. In the `memory.compensate` function, we compensate the original gradient of each node with the reused residual for each grouped layer at each iteration.

Intra-layer Residual Compression Module. We implement the intra-layer residual compression module with a dimension-wise compression function, called `dwcompress`. The `dwcompress` takes the original gradient matrix as input, and uses Pytorch's `torch.topk()` function [14] as the basic compressor to select the

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

Tasks	Models	Model Size	Dataset
Image Classification	ResNet-50 [8]	97.5 MB	Cifar-100 [10]
	VGG-19 [20]	548.1 MB	Cifar-100 [10]
Natural Language Processing	BERT-base [7]	420 MB	SQuAD [17]
	BERT-large [7]	1.34 GB	SQuAD [17]
	GPT2-M [5]	1.4 GB	WikiText-103 [12]
	GPT2-L [5]	3.25 GB	WikiText-103 [12]

top-k largest dimensions from the row and column dimensions for compensating the original gradient. The `dwcompress` further reduces the indices and values of the residual gradients, thus reducing the GPU memory overhead.

5.2 Experimental Setup

Testbeds. We conduct experiments on two testbeds: 1) a cloud cluster of 16 nodes, each of which is equipped with 4 NVIDIA A100 GPUs (80 GB), 2 EPYC 7543 CPUs, and 200 Gbps InfiniBand; 2) a local cluster of 4 nodes, each of which is equipped with 2 NVIDIA V100S GPUs (32 GB), 2 XEON Silver 4214 CPUs, and 25 Gbps Ethernet. Both clusters run Ubuntu 20.04, with software libraries including CUDA-12.0, NCCL-2.8.3, PyTorch-1.13.1 and Horovod-0.28.1.

Workloads. We evaluate six widely-used DNN tasks on three datasets, which are listed in Table 1. We set training parameters such as batch size and learning rate on different models similar to [4,13,25].

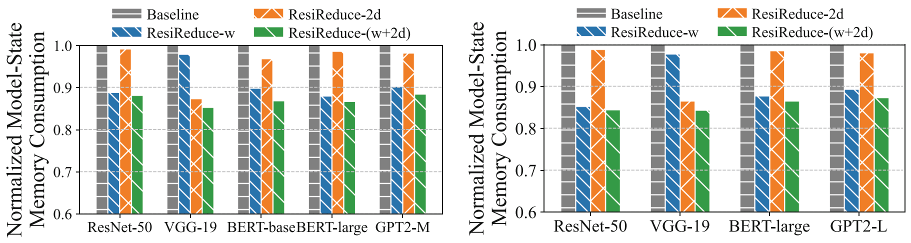
Baseline and ResiReduce Setting. We use Top-k with a default density of 0.01 [2] for gradient compression. We set the compression scheme with storing the whole residual in GPU memory as the Baseline scheme. We name our proposed schemes in Sects. 4.2 and 4.3 as ResiReduce-w and ResiReduce-2d, respectively. ResiReduce-(w+2d) represents our proposed final scheme that combines the above two schemes. In addition, in Experiments 2 and 5, we also consider a non-compression scheme which does not apply gradient compression and thus has no accuracy loss (i.e., the best accuracy).

Metrics. We mainly evaluate three metrics: 1) the normalized model-state memory consumption, which refers to the memory consumption of the model states of all schemes normalized to that of the Baseline scheme; 2) the training accuracy, including Top-1 accuracy (acc) for ResNet-50 and VGG-19, F1 score (F1) for BERT, and perplexity (ppl) for GPT2; and 3) the training throughput, measured by the number of trained samples (images for image classification and tokens for natural language processing) per second [4].

5.3 Experiments

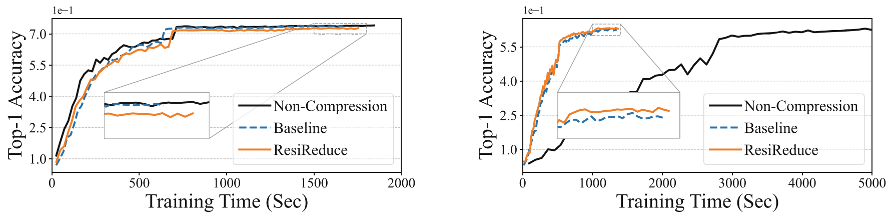
Experiment 1 (Memory consumption): We evaluate the normalized model-state memory consumption of ResiReduce and Baseline on five training tasks in our local cluster. Figure 9a shows that ResiReduce can reduce the memory footprint of the model states by up to 13.3%. For CV tasks (ResNet-50 and VGG-19), compared to Baseline, the normalized model-state memory consumption is reduced by 11.9% and 14.8%. For NLP tasks (BERT-base, BERT-large and GPT2-M), compared to Baseline, the normalized model-state memory consumption of ResiReduce is reduced by 13.2%, 13.3%, and 11.6%, respectively.

We also evaluate the individual performance gains under the different settings for ResiReduce. We see that inter-layer residual reduction (e.g., ResiReduce-w) provides the main contribution for saving memory due to our insight (see Sect. 4.1); intra-layer residual reduction (e.g., ResiReduce-2d) enhances the memory-saving benefits, especially for VGG-19 which contains large-size specific layers (see Sect. 4.1).

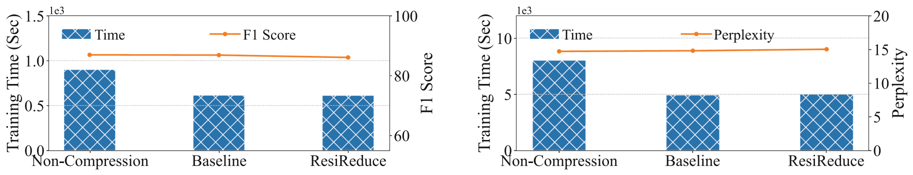


(a) Local cluster with 8 GPUs. (b) Large-scale cluster with 64 GPUs.

Fig. 9. Experiment 1: Normalized memory consumption of the model states



(a) ResNet-50 on Cifar-100 (b) VGG-19 on Cifar-100



(c) BERT-base on SQuAD (d) GPT2-M on WikiText-103

Fig. 10. Experiment 2: Convergence performance and training time.

Table 2. Experiment 2: The training accuracy on the large-scale cluster.

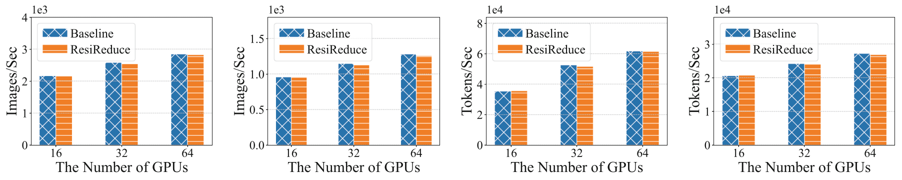
Methods	ResNet-50	VGG-19	BERT-large	GPT2-L
Baseline	72.6%	60.5%	88.73	16.62
ResiReduce	72.9%	61.9%	88.64	16.80

Scaling to More GPUs: We evaluate the memory consumption of ResiReduce and Baseline for four training tasks on a large-scale cloud cluster with 64 GPUs. Figure 9b shows that ResiReduce can reduce the memory footprint of model states by up to 15.7%. Specifically, compared to Baseline, ResiReduce reduces the normalized model-state memory consumption by 15.6%, 15.7%, 13.4% and 12.7% on ResNet-50, VGG-19, BERT-large and GPT2-L, respectively. Thus, ResiReduce maintains its benefits for memory saving in large-scale clusters.

Experiment 2 (Training Accuracy): We evaluate the training time and accuracy of ResiReduce, Baseline and non-compression (the best accuracy) on our local cluster. Figure 10 shows that ResiReduce can preserve the accuracy compared to Baseline. For example, Fig. 10c for BERT-base shows ResiReduce achieves F1-scores of 86.03, which is also close to Baseline (86.82) and non-compression (86.99). This means that ResiReduce does not reduce training accuracy. In addition, the training time of ResiReduce is close to that of Baseline, indicating that ResiReduce has nearly the same benefit as Baseline from gradient compression.

Scaling to More GPUs: We compare the training accuracy of ResiReduce and Baseline on the 64-GPU cluster. Table 2 shows that ResiReduce has a very close accuracy to Baseline. For example, Baseline and ResiReduce achieve 72.6% and 72.9% accuracy for ResNet-50, 16.62 and 16.80 (perplexity) for GPT2-L, respectively. Thus, ResiReduce preserves model accuracy in large-scale clusters.

Experiment 3 (Training Throughput): We evaluate the training throughput of ResiReduce and Baseline using four training tasks on a large-scale cluster with 64 GPUs. Figure 11 shows that the training throughput achieved by ResiReduce is nearly identical to that of Baseline when the number of GPUs increased from 16 to 64. Specifically, ResiReduce exhibits throughput reductions of 0.72%, 1.84%, 0.48% and 1.23% compared to Baseline when training ResNet-50, VGG-19, BERT-large and GPT2-L with 64 GPUs. Therefore, ResiReduce has a slight



(a) ResNet-50 (b) VGG-19 (c) BERT-large (d) GPT2-L

Fig. 11. Experiment 3: The training throughput on the large-scale cluster.

impact on model training throughput in large-scale scenarios. Note that ResiReduce exhibits a slightly more pronounced throughput reduction in VGG-19 compared to other models, primarily due to the higher proportion of specific layer parameters in VGG-19 (see Sect. 4.3).

Experiment 4 (Impact of Different Gradient Compression Methods): We evaluate the training accuracy of ResiReduce under two classic gradient compression methods: QSGD [3] for quantization and DGC [11] for sparsification. Table 3 shows the training accuracy of ResiReduce and Baseline on four models with QSGD and DGC. We see that ResiReduce performs consistently well on different gradient compression methods. For example, when training VGG-19 with DGC, ResiReduce achieves 0.16% higher accuracy compared to Baseline. Therefore, ResiReduce integrates well with any quantization and sparsification gradient compression methods. Note that we do not consider the memory consumption under different compression methods, since they have the same residual memory reduction due to the fact that the size of residuals only depends on the model size regardless of the compression details [24].

Experiment 5 (Impact of Different Gradient Compression Ratios): We evaluate the training accuracy of ResiReduce and non-compression scheme on four models using different compression ratios. Table 4 shows that compared with the non-compression scheme, ResiReduce has rare accuracy loss. Specifically, for compression ratio of 0.1, 0.05 and 0.01, the accuracy loss is 0.17%–0.68%, 0.24%–1.35%, 1.10%–1.76%, respectively. This ensures that ResiReduce has a negligible accuracy loss under different compression rates.

Table 3. Experiment 4: Accuracy across different compression methods.

Methods	ResNet-50	VGG-19	BERT-base	GPT2-M
Baseline-QSGD	72.5%	64.2%	86.68	15.16
ResiReduce-QSGD	72.4%	64.7%	87.07	14.99
Baseline-DGC	71.7%	63.0%	86.88	14.81
ResiReduce-DGC	71.5%	63.1%	86.01	14.96

Table 4. Experiment 5: Accuracy across different compression ratios.

Methods	ResNet-50	VGG-19	BERT-base	GPT2-M
Non-Compression	74.0%	64.2%	86.99	14.69
ResiReduce-0.1	73.5%	64.0%	86.84	14.72
ResiReduce-0.05	73.0%	63.5%	86.78	14.78
ResiReduce-0.01	72.7%	63.2%	86.03	14.87

6 Conclusion

In this paper, we propose ResiReduce, a memory-saving mechanism designed to address the significant memory overhead of residuals in DNN training. Our analysis reveals that residuals consume a significant portion of the model states memory but can be reduced with tiny accuracy impact. By leveraging inter-layer and intra-layer residual reduction techniques, our results show that ResiReduce reduces memory footprint of the model states by up to 15.7% while maintaining model accuracy and training throughput.

Acknowledgments. This work was supported in part by the National Natural Science Foundation of China (NSFC) No.62272185, Shenzhen Science and Technology Program + JCYJ 20220530161006015, and Key Laboratory of Information Storage System Ministry of Education of China.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Achiam, J., et al.: GPT-4 technical report. arXiv preprint arXiv:2303.08774 (2023)
2. Aji, A.F., et al.: Sparse communication for distributed gradient descent. arXiv preprint arXiv:1704.05021 (2017)
3. Alistarh, D., et al.: QSGD: communication-efficient SGD via gradient quantization and encoding. Adv. Neural Inf. Process. Syst. **30** (2017)
4. Bai, Y., et al.: Gradient compression supercharged high-performance data parallel DNN training. In: Proceedings of ACM SOSP, pp. 359–375 (2021)
5. Brown, T., et al.: Language models are few-shot learners. Adv. Neural. Inf. Process. Syst. **33**, 1877–1901 (2020)
6. Chen, T., et al.: Training deep nets with sublinear memory cost. arXiv preprint arXiv:1604.06174 (2016)
7. Devlin, J., et al.: BERT: pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018)
8. He, K., et al.: Deep residual learning for image recognition. In: CVPR, pp. 770–778 (2016)
9. Korthikanti, V.A., et al.: Reducing activation recomputation in large transformer models. Proc. Mach. Learn. Syst. **5** (2023)
10. Krizhevsky, A., et al.: Learning multiple layers of features from tiny images. Master's thesis, University of Tront (2009)
11. Lin, Y., et al.: Deep gradient compression: reducing the communication bandwidth for distributed training. arXiv preprint arXiv:1712.01887 (2017)
12. Merity, S., et al.: Pointer sentinel mixture models. arXiv preprint arXiv:1609.07843 (2016)
13. Ming, Z., et al.: ADTopk: all-dimension top-k compression for high-performance data-parallel DNN training. In: Proceedings HPDC, pp. 135–147 (2024)
14. Paszke, A., et al.: PyTorch: an imperative style, high-performance deep learning library. Adv. Neural Inf. Process. Syst. 8026–8037 (2019)
15. Radford, A., et al.: Language models are unsupervised multitask learners. OpenAI blog **1**(8), 9 (2019)

16. Rajbhandari, S., et al.: Zero: Memory optimizations toward training trillion parameter models. In: SC20, pp. 1–16. IEEE (2020)
17. Rajpurkar, P., et al.: Know what you don't know: unanswerable questions for squad. arXiv preprint arXiv:1806.03822 (2018)
18. Seide, F., et al.: 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech DNNs. In: INTERSPEECH (2014)
19. Sergeev, A., et al.: Horovod: fast and easy distributed deep learning in TensorFlow. arXiv preprint arXiv:1802.05799 (2018)
20. Simonyan, K., et al.: Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014)
21. Stich, S.U., et al.: Sparsified SGD with memory. Adv. Neural. Inf. Process. Syst. **31**, 4452–4463 (2018)
22. Sun, X., et al.: Stronghold: fast and affordable billion-scale deep learning model training. In: SC22, pp. 1–17. IEEE (2022)
23. Tang, Z., et al.: Communication-efficient distributed deep learning: a comprehensive survey. arXiv preprint arXiv:2003.06307 (2023)
24. Wu, D., et al.: BIRD: a lightweight and adaptive compressor for communication-efficient distributed learning using tensor-wise bi-random sampling. In: ICCD, pp. 605–613. IEEE (2023)
25. Xu, H., et al.: GRACE: a compressed communication framework for distributed machine learning. In: ICDCS, pp. 561–572. IEEE (2021)
26. Zhang, Z., et al.: MIPD: an adaptive gradient sparsification framework for distributed DNNs training. IEEE Trans. Parallel Distrib. Syst. **33**(11), 3053–3066 (2022)