COMET: A Novel Memory-Efficient Deep Learning Training Framework by Using Error-Bounded Lossy Compression

Sian Jin
Washington State University
Pullman, WA, USA
sian.jin@wsu.edu

Chengming Zhang
Washington State University
Pullman, WA, USA
chengming.zhang@wsu.edu

Xintong Jiang
McGill University
Montréal, QC, Canada
xintong.jiang@mail.mcgill.ca

Yunhe Feng
University of Washington
Seattle, WA, USA
yunhe@uw.edu

Hui Guan
University of Massachusetts
Amherst, MA, USA
huiguan@cs.umass.edu

Guanpeng Li
University of Iowa
Iowa City, IA, USA
guanpeng-li@uiowa.edu

Shuaiwen Leon Song
University of Sydney
Sydney, NSW, Australia
shuaiwen.song@sydney.edu.au

Dingwen Tao
Washington State University
Pullman, WA, USA
dingwen.tao@wsu.edu

ABSTRACT

Deep neural networks (DNNs) are becoming increasingly deeper, wider, and non-linear due to the growing demands on prediction accuracy and analysis quality. Training wide and deep neural networks require large amounts of storage resources such as memory because the intermediate activation data must be saved in the memory during forward propagation and then restored for backward propagation. However, state-of-the-art accelerators such as GPUs are only equipped with very limited memory capacities due to hardware design constraints, which significantly limits the maximum batch size and hence performance speedup when training large-scale DNNs. Traditional memory saving techniques either suffer from performance overhead or are constrained by limited interconnect bandwidth or specific interconnect technology.

In this paper, we propose a novel memory-efficient CNN training framework (called COMET) that leverages error-bounded lossy compression to significantly reduce the memory requirement for training in order to allow training larger models or to accelerate training. Our framework purposely adopts error-bounded lossy compression with a strict error-controlling mechanism. Specifically, we perform a theoretical analysis on the compression error propagation from the altered activation data to the gradients, and empirically investigate the impact of altered gradients over the training process. Based on these analyses, we optimize the error-bounded lossy compression and propose an adaptive error-bound control scheme for activation data compression. Experiments demonstrate that our proposed framework can significantly reduce the training memory consumption by up to 13.5× over the baseline training and 1.8× over another state-of-the-art compression-based framework, respectively, with little or no accuracy loss.

PVLDB Reference Format:
Sian Jin, Chengming Zhang, Xintong Jiang, Yunhe Feng, Hui Guan, Guanpeng Li, Shuaiwen Leon Song, and Dingwen Tao. COMET: A Novel Memory-Efficient Deep Learning Training Framework by Using Error-Bounded Lossy Compression. PVLDB, 15(4): 886 - 899, 2022.
doi:10.14778/3503585.3503597

Proceedings of the VLDB Endowment, Vol. 15, No. 4 ISSN 2150-8097.
doi:10.14778/3503585.3503597

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/jinsian/COMET.

1 INTRODUCTION

Deep neural networks (DNNs) have rapidly evolved to the state-of-the-art technique for many artificial intelligence (AI) tasks in various science and technology domains, including image and vision recognition [47], recommendation systems [57], and natural language processing (NLP) [7]. DNNs contain millions of parameters in an unparalleled representation, which is efficient for modeling complex nonlinearities. Many works [19, 28, 51] have suggested that using either deeper or wider DNNs is an effective way to improve analysis quality and in fact, many recent DNNs have gone significantly deeper and/or wider [25, 58]. Most of such wide and deep neural networks contain a large portion of convolutional layers, also known as convolutional neural networks (CNNs). For instance, EfficientNet-B7 increases the number of convolutional layers from 31 to 109 and doubles the layer width compared to the base EfficientNet-B0 for higher accuracy (i.e., top-1 accuracy of 84.3% compared to 77.1% on the ImageNet dataset) [52].

In this paper, we explore a *general memory-driven approach for enabling efficient deep learning training*. Specifically, our goal is to drastically reduce the memory requirement for training in order to enlarge the limit of maximum batch size for training speedup. When training a DNN model, the intermediate activation data (i.e., the input of all the neurons) is typically saved in the memory during forward propagation, and then restored during backpropagation to calculate gradients and update weights accordingly [20]. However, taking into account the deep and wide layers in the current large-scale nonlinear DNNs, storing these activation data from all the layers requires large memory spaces which are not available in state-of-the-art training accelerators such as GPUs. For instance, in recent climate research [29], training DeepLabv3+ neural network with 32 images per batch requires about 170 GB memory, which is about 2× as large as the memory capacity supported by the latest NVIDIA GPU A100. Furthermore, modern DNN model design trades off memory requirement for higher accuracy. For example, Gpipe [21] increases the memory requirement by more than 4× for achieving a top-1 accuracy improvement of 5% from Inception-V4.

Evolving in recent years, on the one hand, model-parallel [3] techniques that distribute the model into multiple nodes can reduce the memory consumption of each node but introduce high

communication overheads; on the other hand, data-parallel techniques [45] replicate the model in every node but distribute the training data to different nodes, thereby suffering from high memory consumption to fully utilize the computational power. Several techniques such as recomputation, migration, and lossless compression of activation data have been proposed to address the memory consumption challenge for training large-to-large-scale DNNs. For example, GeePS [8] and vDNN [42] have developed data migration techniques for transferring the intermediate data from GPU to CPU to alleviate the memory burden. However, the performance of data migration approaches is limited by the specific intra-node interconnect technology (e.g., PCIe and NVLinks [14]) and its available bandwidth. Some other approaches are proposed to recompute the activation data [5, 15], but they often incur large performance degradation, especially for computationally intensive layers such as convolutional layers. Moreover, memory compression approaches based on lossless compression of activation data [49] suffer from the limited compression ratio (e.g., only around 2:1 for most floating-point data). Alternatively, recent works [6, 13] proposed to develop compression offloading accelerators for reducing the activation data size before transferring it to the CPU DRAM. However, adding a new dedicated hardware component to the existing GPU architecture requires tremendous industry efforts and is not ready for immediate deployment. This design may not be general enough to accommodate future DNN models and accelerator architectures.

To tackle these challenges, we propose a memory-efficient deep neural network training framework (called COMET, lossy Compression Optimized Memory-Efficient Training) by compressing the activation data using adaptive error-bounded lossy compression. Compared to lossy compression approaches such as JPEG [56] and JPEG2000 [54], error-bounded lossy compression can provide more strict control over the errors that occurred to the floating-point activation data. Also, compared to lossless compression such as GZIP [9] and Zstd [62], it can offer a much higher compression ratio to gain higher memory consumption reduction and performance improvement. The key insights explored in this work include: (i) the impact of compression errors that occurred in the activation data on the gradients and the entire CNN training process under the strict error-controlling lossy compression can be theoretically and experimentally analyzed, and (ii) the validation accuracy can be well maintained based on an adaptive fine-grained control over error-bounded lossy compression (i.e., compression error). To the best of our knowledge, this is the first work *to investigate the lossy compression error impact during CNN training and leverage this analysis to significantly reduce the memory consumption for training large CNNs while maintaining high validation accuracy*. In summary, this paper makes the following contributions:

- We propose a novel memory-efficient CNN training framework via dynamically compressing the intermediate activation data through error-bounded lossy compression.
- We provide a thorough analysis of the impact of compression error propagation during DNN training from both theoretical and empirical perspectives.
- We propose an adaptive scheme to adaptively configure the error-bounded lossy compression based on a series of current training status data.

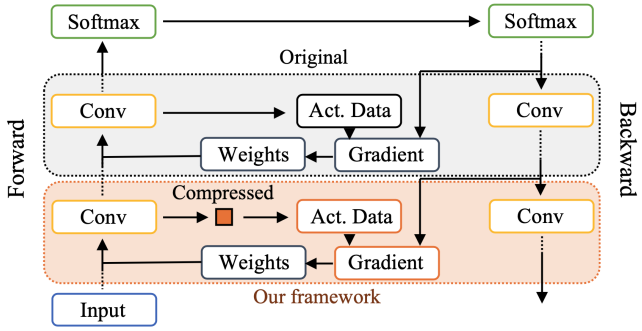


Figure 1: An example data flow of one iteration in CNN training with our COMET framework.

- We propose an improved SZ error-bounded lossy compression to further handle compressing continuous zeros in the intermediate activation data, which can avoid the significant alteration (vanish or explosion) of gradients.
- We evaluate our proposed training framework on four widely-adopted DNN models (AlexNet, VGG-16, ResNet-18, ResNet-50) with the ImageNet-2012 dataset and compare it against state-of-the-art solutions. Experimental results show that our design can reduce the memory consumption by up to 13.5× and 1.8× compared to the original training framework and the state-of-the-art method, respectively, under the same batch size. COMET can improve the end-to-end training performance by leveraging the saved memory for some models (e.g., about 2× training performance improvement on AlexNet).

The rest of this paper is organized as follows. In Section 2, we discuss the background and motivation of our research. In Section 3, we describe an overview of our proposed COMET framework. In Section 4, we present our theoretical support of error impact on validation accuracy from compressed activation data during training. In Section 5, we present the evaluation results of our proposed COMET from the perspectives of parameter selection, memory reduction ability, and performance. In Section 6, we conclude our work and discuss our future work.

2 BACKGROUND AND MOTIVATION

In this section, we first present the background information on large-scale DNN training (i.e., some related work on memory reduction techniques for training) and error-bounded lossy compression for floating-point data. We then discuss the motivation of this work and our research challenges.

2.1 Training Large-Scale DNNs

Training deep and wide neural networks has become increasingly challenging. While many state-of-the-art deep learning frameworks such as TensorFlow [1] and PyTorch [40] can provide high training throughput by leveraging the massive parallelism on general-purpose accelerators such as GPUs, one of the most common bottlenecks remains to be the high memory consumption during the training process, especially considering the limited on-chip memory available on modern DNN accelerators. This is mainly due to the ever-increasing size of the activation data computed in the training

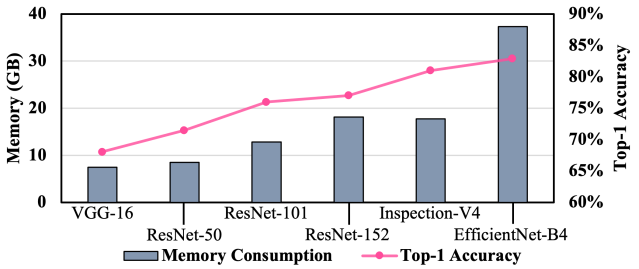


Figure 2: Memory consumption and top-1 accuracy of different state-of-the-art neural networks (batch size = 128).

process. Training a neural network involves many training epochs to update and learn the model weights. Each iteration includes a forward and backward propagation, as shown in Figure 1. The intermediate activation data (the output from each neuron) generated by every layer are commonly kept in the memory until the backpropagation reaches this layer again. Several works [5, 8, 13, 15, 42] have pointed out the large gap between the time when the activation data is generated in the forward propagation and the time when the activation data is reused in the backpropagation, especially when training very deep neural networks.

Figure 2 shows the memory consumption of various neural networks. For these CNNs, comparing with the model/weight size, the size of activation data is much larger since the convolution kernels are relatively small compared to the activation tensors. In addition, training models with enormous batch size over multiple nodes can significantly reduce the training time [16, 41, 61] while reducing the memory consumption can enlarge the maximum batch size capability of a single node for an overall lower cost. In summary, we are facing two main challenges due to the high memory consumption in today's deep learning training: (1) it is challenging to scale up the training process under a limited GPU memory capacity, and (2) a limited batch size leads to low training performance.

In recent years, several works have been proposed to reduce the memory consumption for DNN training, including activation data recomputation [5, 15], migration [8, 42], and compression [13]. Recomputation takes advantage of the layers with low computational cost, such as the pooling layer. Specifically, it frees those layers' activation data and recomputes them based on their prior layers during the backpropagation on demand. This method can reduce unnecessary memory costs, but it is only applicable for the layers of limited types to achieve low performance overhead. For example, compute-intensive convolutional layers that are often hard to recomputed dwarf the efforts of such a method.

Another type of methods are proposed around data migration [8, 42], which sends the activation data from the accelerator to the CPU host when generated, and then loads it back from the host when needed. However, the performance of data migration heavily depends on the interconnect bandwidth available between the host and the accelerator(s), and the intra-node interconnect technology applied. For example, NVLink [14] technology is currently limited to high-end NVIDIA AI nodes (e.g., DGX series) and IBM power series. This paper targets to develop a general technique that can be applied to all types of HPC and datacenter systems.

Finally, data compression is another efficient approach to reduce memory consumption, especially for conserving the memory bandwidth [12, 13, 30]. The basic idea using data compression here is to compress the activation data when generated, hold the compressed data in the memory, and decompress it when needed. However, using lossless compression [6] can only provide a relatively low memory reduction ratio (i.e., compression ratio), e.g., typically lower than 2×. Some other studies such as JPEG-ACT [13] leverages the similarity between activation tensors and images for vision recognition tasks and apply a modified JPEG lossy compressor to activation data. But it suffers from two main drawbacks: First, it introduces uncontrollable compression errors to activation data. Eventually, it could lose control of the overall training accuracy since JPEG is mainly designed for images and is an integer-based lossy compression. Second, the JPEG-based solution [13] needs support from a dedicated hardware component to be added to GPU hardware, and it cannot be directly deployed to today's systems.

We note that all three methods above are orthogonal to each other, which means they could be deployed together to maximize the memory reduction and training performance. Thus, in this paper, we mainly focus on designing an efficient data compression, more specifically a lossy-compression-based solution, to achieve the memory reduction ratio beyond the state-of-the-art approach on CNN models. In addition, since convolutional layers are the most difficult type of layers for efficient recomputation, our solution focuses on convolutional layers to provide high compression ratios with minimum performance overheads and accuracy losses. We also note that COMET can further improve the training performance by combining with model parallelism techniques such as Cerebro [39], which is designed for efficiently training multiple model configurations to select the best model configuration.

2.2 Lossy Compression for Floating-Point Data

Floating-point data compression has been studied for decades. Lossy compression can compress data with little information loss in the reconstructed data. Compared to lossless compression, lossy compression can provide a much higher compression ratio while still maintaining useful information for scientific or visualized discoveries. Lossy compressors offer different compression modes to control compression error or compression ratio, such as error-bounded mode. The error-bounded mode requires users to set an error bound, such as absolute error bound and point-wise relative error bound. The compressor ensures that the differences between the original and reconstructed data do not exceed the user-set error bound.

In recent years, a new generation of lossy compressors for scientific data have been proposed and developed, such as SZ [10, 32, 53] and ZFP [34]. Unlike traditional lossy compressors such as JPEG [56] which are designed for images (in integers), SZ and ZFP are designed to compress floating-point data and can provide a strict error-controlling scheme based on user's requirements. In this work, we choose SZ instead of ZFP because the GPU version of SZ—cuSZ [55] [1] —provides a higher compression ratio than ZFP and offers the absolute error bound mode that the GPU version of ZFP does not support (but necessary for our error control). Specifically,

[1] Compared to CPU SZ, cuSZ can provide much higher compression and decompression speed on GPUs and can also be tuned to avoid the CPU-GPU data transfer overheads.

SZ is a prediction-based error-bounded lossy compressor for scientific data. SZ has three main steps: (1) predict each data point's value based on its neighboring points by using an adaptive, best-fit prediction method; (2) quantize the difference between the real value and predicted value based on the user-set error bound; and (3) apply a customized Huffman coding and lossless compression to achieve a higher ratio. We note that a recent work [25] proposed to use the new generation of lossy compressors to compress DNN weights, thereby significantly reducing model storage overhead and transmission time. However, this work only focuses on compressing the DNN model itself instead of compressing the activation data to reduce memory consumption.

2.3 Research Goals and Challenges

This is the first known work that explores whether the new generation of lossy compression techniques, which have been widely adopted to help scientific applications gain significant compression ratio with precise error control, can significantly reduce the high memory consumption of the common gradient-descent-based training scenarios (e.g., CNNs with forward and backward propagation). We focus on compressing the activation data of convolutional layers, because (1) convolutional layers dominate the size of activation data and cannot be easily recomputed [58], and (2) non-convolutional layers (e.g., pool layers or fully-connected layers) can be easily recomputed for memory reduction due to their low computation complexity.

Note that convolutional layers also dominate the computational time during training process, which benefits us for apply compression with low overhead. Similar to many previous studies [13, 24, 42, 58], our research goal is to develop an efficient and generic strategy to achieve a high reduction in memory consumption for CNN training. Our work can increase the batch size limit and convergence speed or enable training on the hardware with lower memory capacity for the same CNN model.

To achieve this goal, there are several critical challenges to be addressed. First, because of the prediction-based mechanism of SZ lossy compressor, when compressing continuous zeros in between the data, SZ cannot guarantee the decompressed array to remain the same continuous zeros but decompress them into a continuous value that is within the user-defined error bound to zero. This characteristic can cause non-negligible side-effects when implementing our proposed COMET. Thus, we must propose a modified version of SZ lossy compression to overcome this issue for our use case. Second, since we plan to use an error-bounded lossy compressor, a strictly controlled compression error would be introduced to the activation data. In order to maintain the training accuracy curve with a minimum impact to the performance and final model accuracy, *we must understand how the introduced error would propagate through the whole training process.* In other words, we must theoretically and/or experimentally analyze the error propagation, which is challenging. To the best of our knowledge, there is no prior investigation on this. Third, once we understand the connection between the controlled error and training accuracy, how to balance the compression ratio and accuracy degradation in a fine granularity is also challenging. In other words, a more aggressive compression can provide a higher compression ratio but also introduces more errors to activation data, which may significantly degrade the final model accuracy or training performance (cannot converge). Thus, we must find a balance to offer as high a compression ratio as possible to different layers across different iterations while maintaining minimal impact to the accuracy.

3 DESIGN METHODOLOGY

In this section, we describe the overall design of our proposed lossy compression supported CNN training framework COMET and analyze the performance overhead.

Our proposed memory-efficient framework COMET is shown in Figure 3. We iteratively repeat the process shown in the figure for each convolutional layer in every iteration. COMET mainly includes four phases, shown in Figure 3 from left to right: (1) parameter collection of current training status for adaptive compression, (2) gradient assessment to determine the maximum acceptable gradient error, (3) estimation of compression configuration (e.g, absolute error bound), and (4) compression/decompression of activation data with our modified cuSZ. Note that the analysis of our error-bound control scheme for lossy compression of activation data that supports the COMET design will be presented in Section 4.

3.1 Parameter Collection

First, we collect the parameters of the current training status for the following adjustment of lossy compression configurations. COMET mainly collects two types of parameters: (1) offline parameters in CNN architecture, and (2) semi-online parameters including activation data samples, gradient, and momentum.

First of all, we collect multiple static parameters including batch size, activation data size of each convolutional layer, and the size of its output layer. We need these parameters because they affect the number of elements considered into each value in the gradient and hence affect the standard deviation σ in its normal error distribution, which will further impact the validation accuracy curve during training if introduced excessive error too. It would also help the framework collects corresponding semi-online parameters.

For the semi-online parameters, we collect the sparsity of activation data and its average gradient of the loss in backpropagation to estimate how the compression error would propagate from the activation data to the gradient. For the gradient, we compute the average value of its momentum. Note that in many DNN training frameworks such as Caffe [23] and TensorFlow [1], momentum is naturally supported and activated, so it can be easily accessed. The data collection phase is shown as the dashed thin arrows in Figure 3.

Moreover, an active factor W needs to be set at the beginning of training process to adjust the overall activeness in COMET. W is used to determine the activeness of our parameter extraction. We only extract semi-online parameters every W iterations to reduce the computation overhead and improve the overall training performance. Based on our experiment, these parameters vary relatively slowly during training (i.e., the model would not change dramatically with reasonable learning rates in a short time). Thus, we only need to estimate the error impact in a fixed iteration interval in COMET. In this paper, we set W to 1000 as default, which

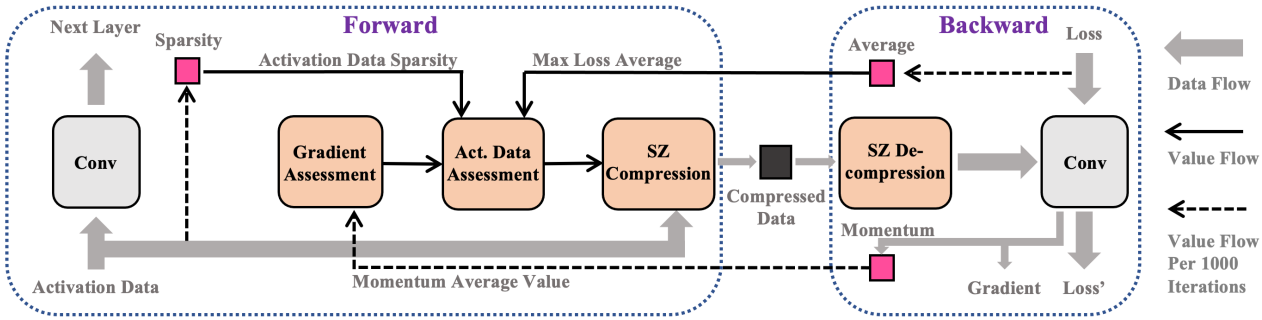


Figure 3: Overview of our proposed memory-efficient CNN training framework COMET.

provides high accuracy and low overhead in our evaluation. However, COMET would reduce W by half if it determines that the maximum error-bound change of all layers between W exceeds 2×, and reset W to default when the error-bound settles to reduce the optimization time. We note that this technique is only effective for models that evolve rapidly during training. Also, note that we still use decompressed data during training to collect these parameters, and the collected parameters from one period of iterations only affect during the following optimization computation.

3.2 Gradient Assessment

Next, we estimate the limit of the gradient error that would result in little or no accuracy loss to the validation accuracy curve during training, as shown in Figure 3. Even with the help of the offset from momentum, we still want to keep the gradient of each iteration as close as possible to the original one. Based on our analysis (will be discussed in Section 4.3), we need to determine the acceptable standard deviation σ in the gradient error distribution that minimizes the impact on the overall validation accuracy curve during training. We use 1% as the acceptable error rate based on our empirical study (will be shown in Section 5.2), i.e., the σ in the momentum error model needs to be:

$$\sigma = 0.01 M_{Average}, \tag{1}$$

where $M_{Average}$ is the average value of the momentum. Note that here we use the average value instead of the modulus length of the momentum because we focus on each individual value of the gradient and the average value is more representative. The average value of the momentum can be considered as the average value of the gradient over a short time period based on the following equation:

$$M_t = \alpha G_{t-1} + \beta G_t, \tag{2}$$

where M_t is the momentum and G_t is the gradient at iteration t. We monitor the momentum by using the API provided by training framework (e.g., `MomentumOptimizer` in TensorFlow) and calculate its average value using simple matrix operations. Similarly, based on our experiment, the average value of gradient does not tend to vary dramatically in a short time period during training.

3.3 Activation Assessment

After that, we dynamically configure the lossy compression for activation data based on the gradient assessment in the previous phase and the collected parameters as shown in Figure 3. Based on our analysis (to be performed in Section 4.2), we need σ (from gradient error model), R (sparsity of activation data), $\bar{L}$ (average value of current loss), and N (batch size) to determine the acceptable error bound for compressing the activation data at the current layer in order to satisfy the gradient error limit proposed in the previous phase. We simplify our estimator as below:

$$eb = \frac{\sigma}{a\bar{L}\sqrt{NR}}, \tag{3}$$

where eb is the absolute error bound for activation data with SZ lossy compressor, σ describes the acceptable error distribution in the gradient, a is the empirical coefficient, $\bar{L}$ is the average value of the current layer's loss, N is the batch size, and R is the sparsity ratio of activation data. Note that our technique can be applied to any non-momentum-based training, which only needs to monitor the "hidden" momentum which can be derived by gradient via simple matrix operations.

3.4 Optimized Adaptive Compression

In the last phase, we deploy the lossy compression with our optimized configuration to the corresponding convolutional layers. We also monitor the compression ratio for analysis. Note that we compress the activation data of each convolutional layer right after its forward pass. We then decompress the compressed activation data in the backpropagation when needed. Also, we only perform the adaptive analysis in every W iterations from Section 3.1 to minimize the analysis overhead. For Pooling, Normalization, and Activation layers, we use the recomputation technique to reduce their memory consumption since they take a large proportion of the memory consumption (~ 60% in the four tested models) with little recomputation overhead. The other layers do not consume a noticeable amount of memory, so they are processed as is.

At the beginning of each training, the training dataset is unknown, and there is no collected semi-online parameter. COMET trains the model with original batch size to guarantee the memory constraint for the first W iterations (negligible to the entire training process) to collect parameters. After that, COMET starts to dynamically adjust the batch size based on the previous compression ratio and control the remaining memory space under the following optimizations: (1) choosing the batch size of 2^k to stabilize the performance, where $k > 0$ is an integer; (2) defining a maximum batch size to avoid unnecessary scaling for smaller models; and (3)

reserving 5% of the total available GPU memory when calculating the capable batch size to avoid overflow caused by unexpectedly low compression ratio. For the extremely low compression ratio case, the compressed data will be evacuated to CPU with a certain communication overhead. However, in our evaluation, this rarely happens with the 5% reserved memory space and thus results in a negligible overhead to the system. We also note that the optimized batch size is almost constant thanks to the batch size setting of 2^k, and the compression ratio is relatively stable on our test models. We will try to provide a dynamic batch size in future work.

Through analysis in Section 4.2 and evaluation in Section 5.2, we identify that the current version of SZ algorithm cannot always reconstruct continuous zeros as an exact zero but introduce a small shift (within the error bound) to those continuous zeros. which can eventually cause gradient explosion and an untrainable model (showed in Figure 10). Thus, we need to force zeros in activation data to remain unchanged in the compression algorithm to maximize the performance of COMET, as discussed in Section 2.3. To solve this issue, we propose an improved version of cuSZ algorithm [55] to handle the case of compressing continuous zeros. Specifically, we add a filter to the decompression process to re-zero those values within the error bound. Every reconstructed value that has the distance to zero within the error bound would be decompressed as zero. Another solution is to leverage those values into zeros during the first quantization step of cuSZ's dual-quant mechanism when compressing to grantee the reconstructed values remain zeros. Compared to the second solution, re-zero those values during decompression means we can still use these non-zero reconstructed values for their following points' prediction instead of using zero during compression, which can ensure our compression ratio would not be affected. By doing so, we will inevitably flush some small values into zeros, but they only take a small proportion and contribute little when calculating the gradient. Thus, it is not worth adding extra overhead to distinguish these decompressed zeros from real zeros.

4 COMPRESSION ERROR IMPACT ANALYSIS

In this section, we present the analytical support of our proposed training framework—analyzing compression error propagation (1) from activation data to gradient and (2) from gradient to validation accuracy curve during training for convolutional layers.

4.1 Modeling Compression Error

cuSZ [55] is a prediction-based error-bounded lossy compressor for floating-point data on GPUs. It first uses a dual-quantization technique to quantize the floating-point input data based on user-set error bound. Then, it applies Lorenzo-based predictor [22] to efficiently predict the value of each data point based on its neighboring points. After that, a quantization code (integer) is generated for each value. Finally, a customized Huffman coding is applied to all the quantization codes. Similar to the original SZ, the error introduced to the input data after decompression usually forms a uniform distribution. This is mainly because of the linear-scaling quantization technique adopted. We refer readers to [35] for more details about the error distribution of SZ from a statistical perspective.

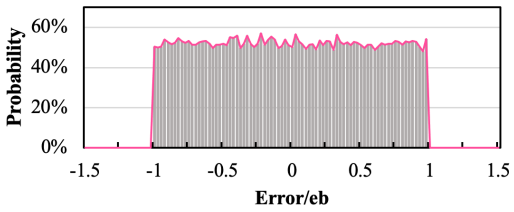


Figure 4: An example of compression error distribution of activation data compressed by cuSZ error-bounded lossy compression with absolute error bound $eb = 10^{-4}$.

Figure 4 illustrates an example of compression error distribution when compressing/decompressing the activation data (generated by the 5th convolutional layer of AlexNet [28] with the ImageNet dataset) by cuSZ. Note that we plot the error distribution every 50 iterations, and observe that all the error distributions are quite similar and follow uniform distribution, which is consistent with the conclusion drawn in the prior work [35]. In fact, based on the evaluation in Section 5, COMET compresses the activation data with a compression ratio less than 20×, where the error distribution is uniform in theory with SZ compression [26]. Thus, we propose to use the uniformly distributed error model to perform analysis and an error-injection-based approach to demonstrate the effectiveness of our theoretical derivation in this section:

$$e \sim U[-eb, +eb] = \begin{cases} \frac{1}{2eb}, & -eb \le x \le eb, \\ 0, & \text{otherwise}, \end{cases} \tag{4}$$

where eb is the user-defined absolute error bound for SZ lossy compressor. Note that for the purpose of our preformed analysis, we inject the error, rather than actually compressing activation data, to demonstrate how uniformly distributed error propagates from the activation data to the gradient and then to the whole training process. We will use actual compression/decompression in our following evaluation.

4.2 Modeling Error Impact on Gradient

Next, we theoretically derive how error propagates from activation data to gradient and provide experimental proof based on statistical analysis using error injection.

As aforementioned (shown in Figure 1), the compressed activation data needs to be decompressed when the backpropagation reaches the corresponding layer. During the backpropagation, each layer computes the gradient to update the weights and the gradient of the loss to be propagated to the previous layer (backpropagation). As shown in Figure 5, on the one hand, the gradient of the loss of activation data for the previous layer only depends on the current layer's gradient of the loss and weight. On the other hand, the gradient depends not only on the gradient of the loss of the current layer but also on the activation data. We note that errors introduced in activation data do not pass across layers along with loss function. In conclusion, in order to understand the impact of compressing the activation data, we must first understand how compression error introduced to the activation data would propagate to the gradient.

In the forward pass, multiple kernels perform convolutions on the input activation data. As shown in Figure 6a, the kernel is performed on the activation data (marked in brown) and generates

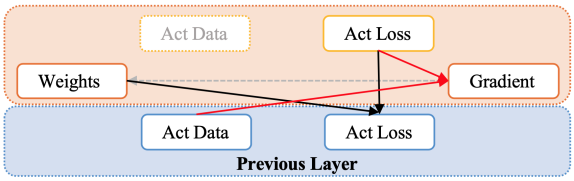


Figure 5: Data dependencies in one convolutional layer during backpropagation.

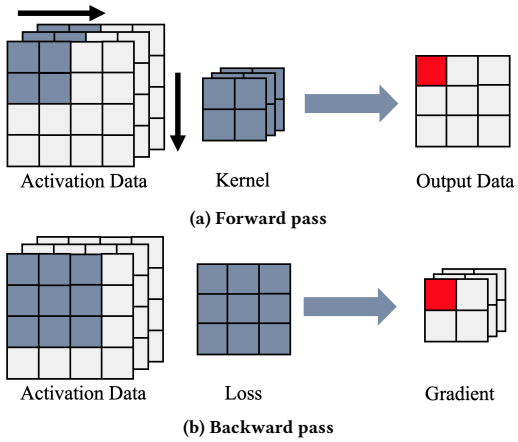


Figure 6: Forward and backward computation in a sample convolutional layer. Kernel size is 2×2, stride is 1, input channel count is 3, and output channel count is 1. Batch size is on another dimension and shown as 1.

the output value as shown in red. Similar to the forward pass, the backward pass reverses the computation, where the parameter's gradient is computed based on the gradient of the loss (with the same dimension of the output data in the forward pass) and the original data in the kernel, as shown in Figure 6b. Similarly, the activation data and the gradient of the loss calculate the gradient value (as shown in red). More specifically, this value is computed as

$$G_{k,i} = \sum_{i=0}^{n} A_{k,i'} \times L_i, \tag{5}$$

where G is the gradient, A is the activation data, L is the gradient of the loss, k is the current channel, i is the value index of the channel, n is the number of values in the gradient of the loss matrix, i' is the corresponding index of activation data to the gradient of the loss matrix. Note that for simplicity, we ignore all i on the right side of Equation (5). Note that here, if the number of output channels is greater than 1, which is true for most convolutional layers, the same process will be used for multiple kernels, as shown in Figure 6b, and the above formula still holds in this case.

Based on our analysis in Section 4.1, the error introduced to the activation data is uniformly distributed. Thus, in the backpropagation, we can have

$$\begin{aligned} G'_{k,i} &= \sum_{i=0}^{n} A'_{k,i'} \times L_i, \\ A'_{k,j} &= A_{k,j} + e, e \sim U[-eb, +eb], \end{aligned} \tag{6}$$

where A' is the decompressed activation data, G' is the gradient altered by the compression error, e is the error, and eb is the user-set absolute error bound. After a simple transform, we can have

$$\begin{aligned} G'_{k,i} &= \sum_{i=0}^{n} A_{k,i'} \times L_i + \sum_{i=0}^{n} e_i \times L_i = G_{k,i} + E, \\ E &= \sum_{i=0}^{n} e_i \times L_i, \end{aligned} \tag{7}$$

where E is the gradient error.

Although it is not possible to calculate or predict the exact value of every element E, we can predict its distribution based on our previous assumption. We also note that the batch size of a typical neural network is usually relatively large during the training process, such as 256, since a larger batch size results in higher training performance in general. As a result, the final gradient for updating weights can be computed as follows.

$$\begin{aligned} G'_{final} &= Average(G'_0, G'_1, ..., G'_N), \\ E_{final} &= Average(E_0, E_1, ..., E_N) \\ &= \sum_{i=0}^{n} \sum_{j=0}^{N} e_{j,i} \times L_{j,i}, \\ e &\sim U[-eb, +eb], \end{aligned} \tag{8}$$

where N is the batch size. Note that all es are independently and uniformly distributed as discussed in Section 4.1. Although L can be related to each other in the same batch, they are still independently distributed across different batches. According to Central Limit Theorem [50], the sum of a series of independent random variables with the same distribution follows a normal distribution, which means the error distribution of the gradient can be expected to be normally distributed. We identify that the distribution of the gradient of the loss L for one input (i.e., one image) is highly concentrated in zero, where the highest value in the gradient of the loss is usually much larger than the average of L. Thus, we can simplify Equation 8 to

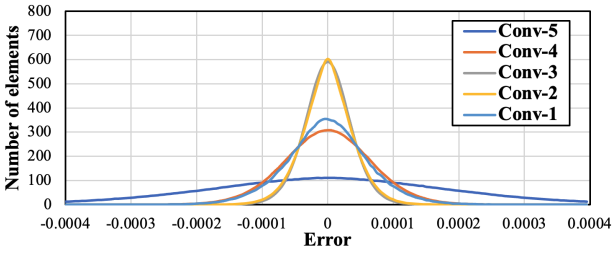
$$\begin{aligned} E_{final} &\approx \sum_{i=0}^{N} e_i \times L_{max,i}, \\ e &\sim U[-eb, +eb], \end{aligned} \tag{9}$$

where L_{max} is the maximum value in the gradient of the loss for each input alone. To reduce the complexity, we can greatly improve the performance with fewer parameters that need to be collected.

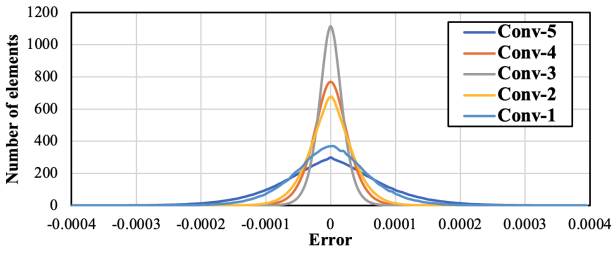
Next, we inject error (to simulate the compression error) to the activation data in convolutional layers based on our error model discussed in Section 4.1. Then, we collect the error of the gradient in the backpropagation.

Figure 7a illustrates the normalized error distribution of gradients collected from different layers, all of which follow the normal distribution as expected. In fact, by calculating the percentage of the area within $\pm\sigma$ of each curve, we can get a value close to 68.2%, which confirms our theoretical derivation. Then, we need to figure out how to predict σ before compression in order to calculate the desired and acceptable error bound for each layer's activation data.

First, we note that σ is highly related to the number of elements that are combined together. In general, more elements result in larger σ, and vice versa. A 2× increase of elements results in $\sqrt{2}\times$ increase of σ, which means that more uncertainties have been added to the system. Second, σ is also related to the value scale, in this case, the average of the gradient of the loss L at the current layer. Note that it is not necessary to compute the average value of the gradient of the loss in every iteration. Instead, we can compute it every W iterations to reduce the computation overhead, since this



(a) Original zeros have been compressed



(b) Original zeros remain zero

Figure 7: Distributions of gradient error when injecting modeled compression error to activation data. Note that the number of elements is normalized to ensure different layers under the same scale. Data is collected every 100 iterations.

value is relatively stable in a fixed period of time. Based on these two parameters, we can estimate σ by the following equation:

$$\sigma \approx a\bar{L}\sqrt{N}eb, \tag{10}$$

where L is the gradient of the loss matrix, N is the batch size, and a is an empirical coefficient. Note that this coefficient a is unchanged for different neural networks because it is essentially a simplified value of the previous equation.

Finally, we note that there is a notable fraction of activation data to be zeros, however, our above analysis so far does not cover it. When compressing a series of continuous zeros, the original cuSZ may change them into a continuous small value within the user-defined error bound instead of exactly zero. Because the reconstructed continuous small values are the same value due to the Lorenzo-based predictor in SZ, this can cause the gradient computation to have nonnegligible offset that eventually can cause gradient exploding problems. To solve this issue, we propose an improved version of cuSZ to enhance the compression on continuous zeros as discussed in Section 3. This means no error would be introduced when facing continuous zeros. Moreover, in some cases, the activation data may contain many zeros due to the activation function layer (such as the ReLU layer) before the current convolutional layer. If this happens, the activation data of convolutional layer can be quickly recomputed through the activation function instead of being saved, which will essentially erase the negative values to zeros. Since lossy compression such as cuSZ is unlikely to change the sign of the activation data value, these data will remain at zero.

Figure 7b shows the distribution of the gradient error after we inject the error into the activation data (maintaining zeros unchanged). Compared with Figure 7a, we can observe a decrease of σ, but it still holds a normal distribution. This decrease is partly because of the reduction of the number of elements in Equation 8, since those zeros do not have any error. In these cases, we can revise the prediction of σ accordingly by the following equation:

$$\sigma' = \sigma\sqrt{R}, \tag{11}$$

where R is the ratio of non-zero elements percentage in the activation data. Again, in practice, we do not need to compute this ratio every iteration but every W iterations, since this ratio is relatively stable in a fixed period of time.

4.3 Error Impact Analysis

Finally, we discuss the error propagated from gradient to overall validation accuracy curve during training using an experimental analysis. Our goal is to identify the maximum acceptable gradient error that would cause little or no validation accuracy loss. According to our performed analysis in Section 4.2, the gradient error can be modeled as a normally distributed error.

In this subsection, we follow the same strategy used in the last subsection to inject error to the gradient that follows our error model and perform the analysis and evaluation. It is worth noting that similar to many existing studies (e.g., CNN model pruning [18], compression [25], mixed-precision training [37]), our hypothesis is that *the accuracy loss caused by the errors added to a given convolutional layer is not noticeably amplified by its following layers.* Other existing study [59] also points out that adding noise to the training data can even provide a regularization effect that can help improve the training performance from overfitting. Our introduced error is slightly different from purely adding noise to training data, but rather it can be considered as noise with a uniform distribution on the nonzero activation data, which only affects the gradient computation during the backpropagation phase and precludes any error propagation across layers. This might degrade the training performance due to the alternated gradient update. However, as pointed out by Lin *et al.* [33], gradient can be considerably approximated until the training performance is affected. In addition, such noise can potentially help training get rid of local minima, especially for models with rough loss surfaces, while providing similar convergence speeds for models with smooth loss surfaces [31].

Momentum has been widely adopted in most neural network training [11], which can be used for alleviating the impact of the gradient error [44]. Actually, in order to update the weights, it is based not only on the gradient computed from the current iteration but also on the momentum. In other words, both the gradient and the momentum (with the same dimension as the weights and gradient) take up a portion of the updated data for weights. Thus, it is critical to maintaining an accurate momentum vector (similar to the error-free one) to guide the weight update. While thanks to the normally distributed gradient error, which is centralized and symmetric in the original direction, the momentum error is relatively low compared to the gradient error. Therefore, this does help the training towards the correct direction—even a few iterations may generate the undesired gradient, they can be offset quickly through the momentum based on the estimated gradient error. Similarly, other optimization algorithms such as Adagrad and Adam [44] also follow a similar principle and can be benefited from COMET.

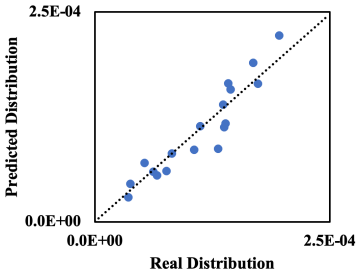


Figure 8: Comparison of standard deviation σ of gradient error (caused by compression error introduced to activation data) from measured distribution and from predicted distribution (based on our theoretical analysis).

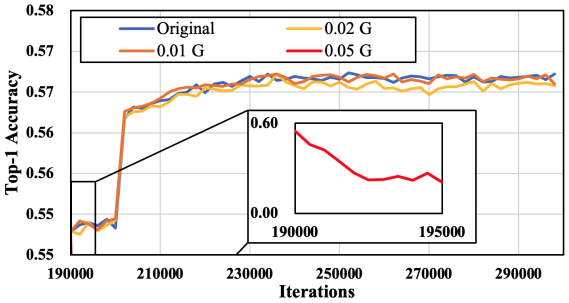


Figure 9: Validation accuracy training curve started from itration 190,000. Learning rate updated at iteration 200,000 Evaluation on different fractions of error introduced to the gradient. G represents for average value of gradient with coefficient value used in COMET.

5 EXPERIMENTAL EVALUATION

In this section, we evaluate COMET from four aspects, including (1) evaluation of compression error impact on gradient, (2) evaluation of error propagation from gradient to training curve, (3) comparison between COMET and the state-of-the-art method, and (4) performance evaluation with multiple state-of-the-art GPUs. We manually set the batch size of COMET for observation and variable control purposes unless specified.

5.1 Experimental Setup

Our evaluation is conducted with Caffe [23] and Tensorflow 1.15 [1]. We choose Caffe for a single-node experiment due to its easy-to-modify architecture and choose Tensorflow for multi-node evaluation due to its being widely used in the research. Our experiment platform is the Longhorn system [36] at TACC and the Bridge-2 system [4] at PSC, of which each GPU node is equipped with 4/8 NVIDIA Tesla V100 GPUs [17] per node. Our evaluation dataset is the ImageNet-2012 [28] and Stanford Dogs Dataset [27]. The CNN models used for image classification include AlexNet [28], VGG-16 [47], ResNet-18, ResNet-50 [19], and EfficientNet [52].

5.2 Error Impact Evaluation

First, we evaluate our proposed theoretical analysis in Section 4.2. Based on Equations 10 and 11, we can estimate σ which stands for how an error is distributed in the gradient. After implementing our estimation, we identify that coefficient a in Equation 10 is 0.32 based on our experiment. This is reasonable because if we consider the extreme condition that the batch size $N = 1$, the error distribution in the gradient will be the same as the SZ lossy compression to uniformly distribute and result in $a = 1/3$.

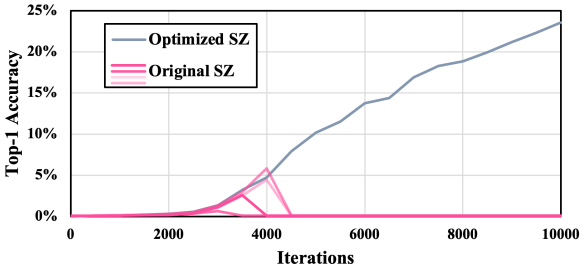


Figure 10: Validation accuracy training curve at the start of training. Comparison of validation accuracy (during training) using original SZ and our optimized SZ in COMET on AlexNet (batch size = 256). The evaluation of the original SZ is conducted four times (with four curves).

We also evaluate our estimation on different layers of AlexNet and VGG-16 using the batch size of 256, as shown in Figure 8. We can clearly observe that the coefficient and how our estimated value aligns with the actual error distribution. This means that we can not only estimate the error propagation but also determine the error bound based on a given acceptable σ error distribution.

Next, we evaluate the error impact from gradient to the overall training process in terms of validation accuracy, as discussed in Section 4.3. Since we target to cause little or no accuracy loss, we focus on the iterations close to the end of training in this evaluation, since the accuracy is harder to be increased when the training is close to the end. To reduce the training time and find an empirical solution for this specific analysis, we pre-train the model without COMET first and save the snapshot every epoch. Then, we perform our evaluation of error impact analysis using those snapshots from different iterations to demonstrate the effectiveness of COMET.

Figure 9 shows our experiment with AlexNet starting from the iteration of 190,000 with a batch size of 256. Here smaller coefficient value means less error introduced to the gradient but potentially lower compression ratio. On the one hand, we can observe from the zoomed-in subfigure that $\sigma = 0.05$ would result in an unacceptable error loss that cannot be eventually recovered. On the other hand, $\sigma = 0.02$ can provide better accuracy and a higher compression ratio, but it does affect the accuracy a bit in some cases. Thus, considering that our goal is a general solution for convolutional layers, we eventually choose $\sigma = 0.01$ as default in COMET; in other words, the target σ is 1% of the average of gradient. In fact, we evaluate 0.1%~5% for σ and choose the best one (i.e., 1%) to minimize the accuracy impact.

Last but not the least, we also evaluate the effectiveness of our proposed optimized SZ, by adding a filter to the decompression process to re-zero values that are closer to zero than the defined error bound in order to improve the performance for the reconstruction of contiguous zeros. As shown in Figure 10, we can observe that without our modification to the current version of SZ, the training process cannot even sustains through 5,000 iterations. This is because the original SZ would usually reconstruct continuous zeros to a continuous small value by design of the Lorenzo predictor (i.e.,

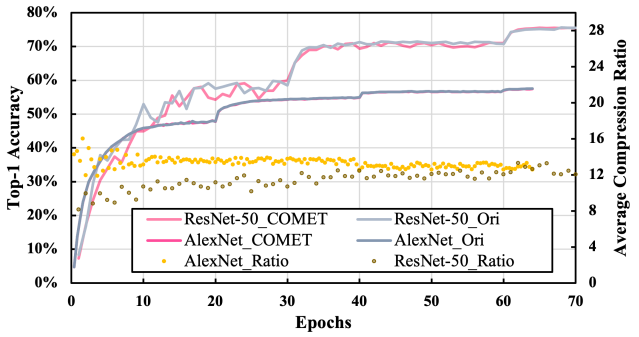


Figure 11: Comparison of validation accuracy between baseline training and COMET (batch size = 256). Lines represent the validation accuracy during training; dots represent the compression ratio of COMET on secondary y-axis.

only ensuring the reconstructed values to be within the pre-defined error bound to the original values) and can cause gradient update to shift from original that eventually leads to gradient explosion. Note that we perform the experiment with the original SZ four times, leading to crashes in four different iterations due to their different training initial states and the uncertainty of gradient explosion. Figure 10 illustrates that our optimized SZ can solve this issue and provide stable performance during the training process.

5.3 Memory Reduction Evaluation

We test COMET on various popular CNNs and evaluate its memory reduction capability. We use the original training approach of each model without memory reduction techniques (i.e., recomputation, migration, and compression) as the baseline. Figure 11 illustrates the result with AlexNet and ResNet-50. The black and red lines are the validation accuracy curves of the baseline training and COMET. We can observe that these two curves are very close to each other, meaning COMET does not noticeably affect the validation accuracy. We also illustrate the change of compression ratio to iteration in yellow dots. In the early stage of training, the compression ratio can be slightly unstable because of the relatively large change to the model itself. Note that the compression ratio will change slightly when the learning rate changes because the learning rate only matters when updating the gradient to the weights. Other than that, for some layers, although the average maximum loss of each input should be decreased and result in a higher error bound, the corresponding activation data value is actually increased. Thus, the compression ratio would not increase even with a higher error bound. Moreover, we evaluate a static strategy for comparison: we estimate the error bound only once at the beginning of training (i.e., at iteration 1, 100, and 200) and keep using this error bound through the whole training process. This static strategy leads to a significant accuracy drop (i.e., top-1 accuracy of only 30.6%, 36.7%, and 31.0% on AlexNet at 10 epochs, respectively) compared to the baseline training (i.e., 45.9% at 10 epochs), which proves the necessity of adaptively configuring error bound.

Table 1 shows the compression ratio of convolutional layers and the overall peak memory usage that COMET can provide at the batch size of 128. "Max batch" means the maximum batch size that the baseline and COMET can run with on a single GPU with 16 GB memory. There is almost no accuracy loss or only little, with up to 0.31%. Thanks to our careful control of compression error and thorough analysis and modeling of error impact. COMET can deliver a promising compression ratio without heavy efforts of fine-tuning any parameter for different models. Overall, COMET can provide up to 13.5× compression ratio with little or no validation accuracy loss.

Table 1: Comparison of validation accuracy between baseline training and COMET; comparison of compression ratio between JPEG-ACT and COMET.

Neural Nets		Top-1 Accuracy	Peak Mem.	Max Batch	Conv. Act. Size	COMET	JPEG-ACT
AlexNet	b.	57.41%	2.17 GB	512	407 MB		
	c.	57.42%	0.85 GB	2048	**30 MB**	13.5×	-
VGG-16	b.	68.05%	17.29 GB	64	6.91 GB		
	c.	68.02%	5.04 GB	256	**0.62 GB**	11.1 ×	-
ResNet-18	b.	67.57%	5.16 GB	256	1.71 GB		
	c.	67.43%	1.37 GB	1024	**0.16 GB**	10.7 ×	7.3 ×
ResNet-50	b.	75.55%	15.57 GB	128	5.14 GB		
	c.	75.51%	4.40 GB	512	**0.46 GB**	11.0 ×	6.0 ×

b.= baseline, c.= compressed

Compared to recomputation-based memory reduction solution [5, 15], COMET can provide a high compression ratio to activation data of convolutional layers that cannot be reduced with acceptable overhead by recomputation. Compared to the migration-based solution, COMET provides a higher compression ratio on our evaluated models without the constrain of CPU-GPU communication (e.g., COMET provides 11.0× compression ratio compared to 2.1× by Layrub [24] on ResNet-50). COMET is also more flexible in comparison: if users decide not to increase the batch size (although it is a common optimization), COMET can still help train the same model with much lower memory requirement, which otherwise cannot be trained with the baseline training, enabling a scaled-up training solution; it can also save more precious shared-memory space for other co-running workloads via container [2] or GPU multi-instance technologies [38]. Compared with the lossless-compression-based solution [43], which reduces the memory usage by only 1~2×, COMET outperforms it by over 9×; compared with the current state-of-the-art JPEG-based solution [13], which uses an hardware-implemented image-based compressor to provide up to 7× compression ratios, COMET outperforms it by 1.5× and 1.8× on ResNet-18 and ResNet-50, respectively, shown in Table 1. Note all methods mentioned above including our proposed COMET are orthogonal to each other and can be deployed together to maximize the compression ratio and training performance.

Moreover, other than training from scratch, COMET is also capable of fine tuning from an existing model. We use EfficientNet-B0 [52] for demonstration. The model was pre-trained on ImageNet dataset and is evaluated by fine tuning on the Stanford Dogs dataset that contains 12,000 images for training and 8,580 images for testing in 120 categories. We set all layers "trainable" and use a relatively small learning rate of 2^{-5} to perform COMET on all activation data. Note that compressing all convolutional layers (i.e., introducing compression error to all convolutional layers) during the fine-tuning stage with a small learning rate is more challenging to COMET compared to the partial-layer fine-tuning approach. Similar to Figure 11, Figure 12 shows that the validation accuracy curve

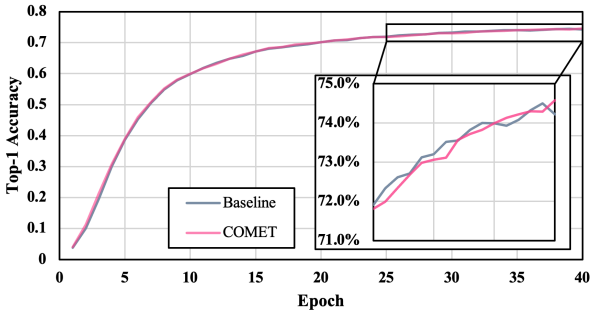


Figure 12: Comparison of validation accuracy between baseline training and COMET on EfficientNet-B0.

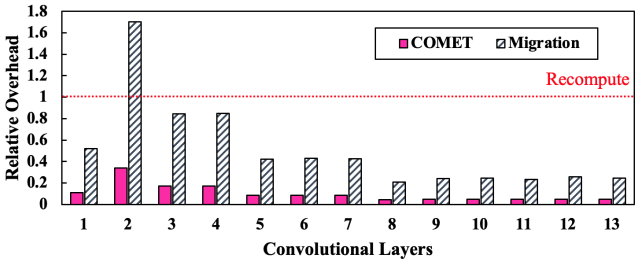


Figure 13: Overhead comparison between migration, recomputation, and COMET on VGG-16 (batch size = 128). Time is normalized to the computation time of given convolutional layer, which is also the recompute overhead.

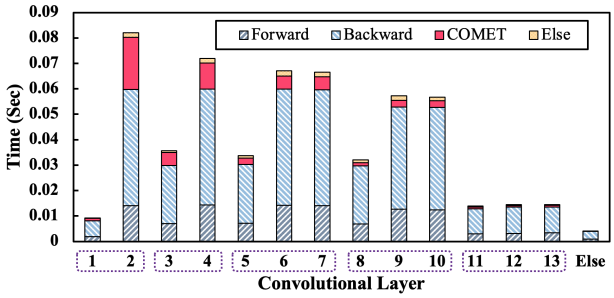


Figure 14: Time breakdown of one training iteration with COMET on VGG-16 (batch size = 128).

of the baseline and COMET are aligned with each other, meaning that COMET successfully provides a high compression ratio with minimal validation accuracy loss during the training process.

5.4 Performance Evaluation and Analysis

As aforementioned, COMET features a theoretical analysis to estimate error propagation and to provide an adaptive configuration for activation data compression based on easy-to-collect parameters. Note that it is almost impossible to find adaptive configurations by trial-and-error method for DNN training because of its extremely long training period, not to mention it requires a large number of traverses for high configuration precision. As a result, only by using the theoretical analysis as the backbone of COMET can we avoid the trial-and-error method to select adaptive solutions.

Regarding the performance overhead of COMET, it needs to extract usable parameters and compute the compression configuration every 1,000 iterations, while the amortized overhead is

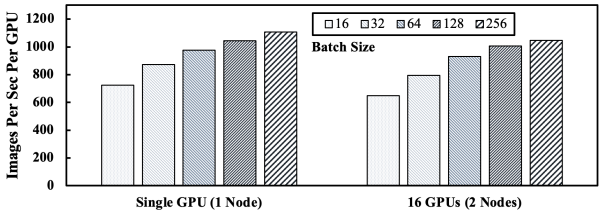


Figure 15: Training throughputs with different batch sizes on ResNet-50. Throughput per GPU slightly decreases with more GPUs due to increasing communication overhead for gradient updates.

almost negligible; on the other hand, thanks to its high working efficiency, cuSZ can provide an extremely high compression and decompression speed on GPU [55]. We modify cuSZ by adding a filter that changes all the values under the error bound to zeros, as discussed in Section 3.4. This helps us to keep zeros in the activation data unchanged while only causing little overhead to the framework. Overall, COMET introduces about 17% overhead the training process while keeping the same training batch size for our experimented models. In comparison, the state-of-the-art recomputation-based solution [15] cannot support convolutional layers without the significantly large overhead, and the image-based compression solution [13] is based on hardware implementation and simulation. Moreover, the state-of-the-art migration solution Layrub achieves a memory reduction of 2.4× on average but with a higher performance overhead of 24.1% [24] on K40M GPU. Figure 13 shows the overhead comparison between the three techniques on convolutional layers. Note that the migration takes an even larger overhead than claimed because the computational power improvement from K40M to V100 reduces the computation time while both systems still bottleneck by similar communication bandwidth. Overall, compared to data migration, COMET can provide a comparable compression ratio while introducing fairly less overhead; both outperform recomputation on convolutional layers.

Figure 14 demonstrates the training time breakdown of COMET on VGG-16. We can observe that for most layers, COMET only introduces a little overhead in comparison to the forward and backward propagation time. Specifically, COMET introduces an overall overhead of 11.5% with the same batch size. Note that for each group of convolutional layers (framed in purple dashed line in Figure 14), the first convolutional layer (i.e., layer 1, 3, 5, 8, and 11) has less overhead compared to the other layers in the group due to its smaller size of activation data. We also acknowledge that COMET may introduce higher overheads than expected to the networks that contain convolutional layers with many 1×1 kernels. This is because 1×1 kernels take little time to compute but require a relatively high overhead to compress and decompress. Calculating such layers is very efficient, compared with the GPU (de)compression on similar sizes of activation data. Thus, COMET is more suitable for the CNNs composed of larger convolution kernels than 1×1.

COMET introduces relatively small overhead to the training process while can greatly reduce the memory utilization and allow larger and wider neural networks to be trained with limited GPU memory. Moreover, the saved memory can also be further

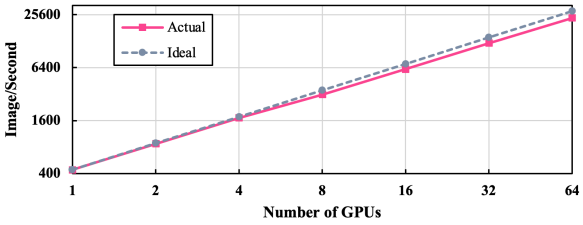


Figure 16: Multi-GPU performance of COMET framework with ResNet-50. Y-axis represents the overall throughput. "Ideal" represents the ideally linear performance scalability.

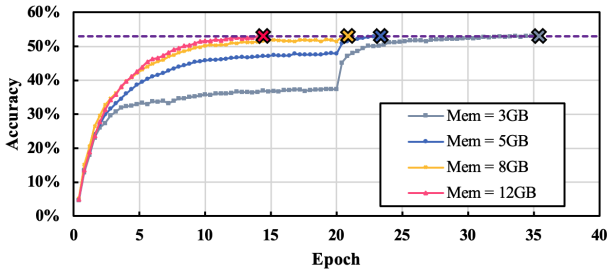


Figure 17: Validation accuracy curve of COMET under different GPU memory constraint on AlexNet. By compressing activation data and increasing batch size, COMET can improve the end-to-end training performance.

utilized for a larger batch size, which improves the overall performance. Figure 15 shows the improvement of training throughout (i.e., images per second per GPU) with increasing batch size on both single-/multi-GPU cases. Specifically, COMET provides a training throughout improvement of up to 1.27× and 1.30× on ResNet-50 with 1 GPU and 8 GPUs, respectively. Furthermore, this performance improvement can offset the overheads of COMET (e.g., error-bound estimation and activation data compression). For example, the overall overhead can be reduced from 11.5% to -7% on VGG-16 by utilizing the saved memory to increase the batch size from 32 to 256. It means that we can still improve the overall training performance by fully utilizing the GPU computational performance despite the compression overhead.

In addition, we also evaluate the performance of our framework scaled from a single GPU (from a single node) to 64 GPUs (from 8 nodes), as shown in Figure 16. It illustrates that COMET is highly scalable thanks to no extra communication cost introduced. Note that the small performance degradation compared to the ideal speedup is due to the training framework itself (e.g., the communication overhead of AllReduce for gradient update).

Finally, COMET improves the end-to-end training performance by faster convergence speed due to larger batch size [60]. This is because a larger batch size involved more training data per iteration which leads to a more precise gradient direction. Figure 17 shows that the convergence speed of COMET is faster with a larger batch size on AlexNet. For example, it takes 14.6 epochs with the batch size of 1024 (COMET at Mem = 12 GB) while 21.3 epochs with the batch size of 512 (COMET at Mem = 8 GB) to train AlexNet to the same top-1 accuracy of 53.0%. The baseline training under 12 GB memory can only use the batch size of 512, which is significantly slower than COMET. As a result, we can achieve over 2× speedup by using 8× larger batch size for AlexNet. Note that the convergence speeds of AlexNet at the batch sizes of 512 and 1024 are similar to each other, as both of them reach the scalability limit, meaning that larger batch sizes cannot further improve the training performance.

It is worth noting that prior studies [16, 46] showed that the scaling limit is considerably large for many deep neural networks. For example, Goyal *et al.* work [16] provides tuning insights to train ResNet-50 at an enormous batch size of 8k without scaling bottleneck. Shallue *et al.* [46] points out that the scaling bottleneck can be more significant for deep convolutional models. Thus, increasing batch size can reduce the overall training time with the same amount of compute resources and significantly increase the training performance on various models and datasets [13, 24, 41, 48]. Considering that the memory consumption is relatively large for datasets like ImageNet, COMET can help the training reach the maximum batch size that can achieve the minimum time-to-solution.

Overall, we identify that the performance improvement of COMET is threefold: (1) higher throughput per GPU unit due to higher resource utilization (thanks to larger batch size), (2) higher convergence speed (thanks to larger batch size within the scaling limit), and (3) a new capability of training larger models with limited memory space. We acknowledge that not all three benefits can be achieved at the same time for a given model. For simple models such as AlexNet, enlarging the batch size to increase the convergence speed only happens at a relatively small batch size (i.e, 2^{10}).

6 CONCLUSION AND FUTURE WORK

In this paper, we propose a novel memory-efficient deep learning training framework. We utilize the SZ error-bounded lossy compressor to reduce the memory consumption of convolutional layers. We develop an error propagation model and prove its accuracy. We evaluate our proposed framework on several popular CNNs with the ImageNet dataset. The result shows that our framework significantly reduces the memory usage by up to 13.5× with little or no accuracy loss. Compared with the state-of-the-art compression-based approach, our framework can provide a memory reduction improvement of up to 1.8×. By leveraging the saved memory, COMET can improve the end-to-end training performance (e.g., about 2× on AlexNet). We plan to integrate data migration and recomputation methods into COMET for higher performance and more memory reduction. We will also explore the applicability of COMET to other types of layers and models such as transformer. Moreover, we will further reduce the (de)compression overhead of COMET by overlapping compression with operations such as convolution.

ACKNOWLEDGMENTS

This material is based upon work supported by National Science Foundation under Grant No. OAC-2034169 and OAC-2042084. The work was also partially supported by Australian Research Council Discovery Project DP210101984 and Facebook Faculty Award. This work used the Bridges-2 system at the Pittsburgh Supercomputing Center (PSC) under the Extreme Science and Engineering Discovery Environment (XSEDE). The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing access to the Longhorn system that has contributed to the research results reported within this paper.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).

[2] Marcelo Amaral, Jordà Polo, David Carrera, Seetharami Seelam, and Malgorzata Steinder. 2017. Topology-aware gpu scheduling for learning workloads in cloud environments. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[3] Tal Ben-Nun and Torsten Hoefler. 2019. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)* 52, 4 (2019), 1–43.

[4] Bridge-2 system. 2020. https://www.psc.edu/resources/bridges-2/. (Accessed on 12/22/2021).

[5] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174* (2016).

[6] Esha Choukse, Michael B Sullivan, Mike O'Connor, Mattan Erez, Jeff Pool, David Nellans, and Stephen W Keckler. 2020. Buddy compression: Enabling larger memory for deep learning and HPC workloads on gpus. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 926–939.

[7] Ronan Collobert and Jason Weston. 2008. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th International Conference on Machine Learning*. ACM, 160–167.

[8] Henggang Cui, Hao Zhang, Gregory R Ganger, Phillip B Gibbons, and Eric P Xing. 2016. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 4.

[9] Peter Deutsch. 1996. *GZIP file format specification version 4.3.* Technical Report.

[10] Sheng Di and Franck Cappello. 2016. Fast error-bounded lossy HPC data compression with SZ. In *2016 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 730–739.

[11] Yinpeng Dong, Fangzhou Liao, Tianyu Pang, Hang Su, Jun Zhu, Xiaolin Hu, and Jianguo Li. 2018. Boosting adversarial attacks with momentum. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 9185–9193.

[12] Magnus Ekman and Per Stenstrom. 2005. A robust main-memory compression scheme. In *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 74–85.

[13] R David Evans, Lufei Liu, and Tor M Aamodt. 2020. JPEG-ACT: Accelerating Deep Learning via Transform-based Lossy Compression. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 860–873.

[14] Denis Foley and John Danskin. 2017. Ultra-performance Pascal GPU and NVLink interconnect. *IEEE Micro* 37, 2 (2017), 7–17.

[15] Aidan N Gomez, Mengye Ren, Raquel Urtasun, and Roger B Grosse. 2017. The reversible residual network: Backpropagation without storing activations. In *Advances in neural information processing systems*. 2214–2224.

[16] 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).

[17] NVIDIA V100 TENSOR CORE GPU. 2020. https://www.nvidia.com/en-us/data-center/v100/. (Accessed on 12/22/2021).

[18] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[19] 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.

[20] Robert Hecht-Nielsen. 1992. Theory of the backpropagation neural network. In *Neural networks for perception*. Elsevier, 65–93.

[21] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in neural information processing systems*. 103–112.

[22] Lawrence Ibarria, Peter Lindstrom, Jarek Rossignac, and Andrzej Szymczak. 2003. Out-of-core compression and decompression of large n-dimensional scalar fields. In *Computer Graphics Forum*, Vol. 22. Wiley Online Library, 343–348.

[23] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 675–678.

[24] Hai Jin, Bo Liu, Wenbin Jiang, Yang Ma, Xuanhua Shi, Bingsheng He, and Shaofeng Zhao. 2018. Layer-centric memory reuse and data migration for extreme-scale deep learning on many-core architectures. *ACM Transactions on Architecture and Code Optimization (TACO)* 15, 3 (2018), 1–26.

[25] Sian Jin, Sheng Di, Xin Liang, Jiannan Tian, Dingwen Tao, and Franck Cappello. 2019. Deepsz: A novel framework to compress deep neural networks by using error-bounded lossy compression. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*. 159–170.

[26] Sian Jin, Guanpeng Li, Shuaiwen Leon Song, and Dingwen Tao. 2021. A novel memory-efficient deep learning training framework via error-bounded lossy compression. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 485–487.

[27] Aditya Khosla, Nityananda Jayadevaprakash, Bangpeng Yao, and Li Fei-Fei. 2011. Novel Dataset for Fine-Grained Image Categorization. In *First Workshop on Fine-Grained Visual Categorization, IEEE Conference on Computer Vision and Pattern Recognition*. Colorado Springs, CO.

[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[29] Thorsten Kurth, Sean Treichler, Joshua Romero, Mayur Mudigonda, Nathan Luehr, Everett Phillips, Ankur Mahesh, Michael Matheson, Jack Deslippe, Massimiliano Fatica, et al. 2018. Exascale deep learning for climate analytics. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 649–660.

[30] Sohan Lal, Jan Lucas, and Ben Juurlink. 2017. Eˆ 2MC: Entropy Encoding Based Memory Compression for GPUs. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1119–1128.

[31] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. 2017. Visualizing the loss landscape of neural nets. *arXiv preprint arXiv:1712.09913* (2017).

[32] Xin Liang, Sheng Di, Dingwen Tao, Sihuan Li, Shaomeng Li, Hanqi Guo, Zizhong Chen, and Franck Cappello. 2018. Error-Controlled Lossy Compression Optimized for High Compression Ratios of Scientific Datasets. (2018).

[33] 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).

[34] Peter Lindstrom. 2014. Fixed-rate compressed floating-point arrays. *IEEE Transactions on Visualization and Computer Graphics* 20, 12 (2014), 2674–2683.

[35] Peter Lindstrom. 2017. *Error distributions of lossy floating-point compressors.* Technical Report. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States).

[36] Longhorn subsystem. 2020. https://www.tacc.utexas.edu/systems/longhorn. (Accessed on 12/22/2021).

[37] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. 2017. Mixed precision training. *arXiv preprint arXiv:1710.03740* (2017).

[38] MIG. 2020. NVIDIA Multi-Instance GPU. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/. (Accessed on 12/22/2021).

[39] Supun Nakandala, Yuhao Zhang, and Arun Kumar. 2020. Cerebro: A data system for optimized deep learning model selection. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2159–2173.

[40] 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.

[41] J Gregory Pauloski, Zhao Zhang, Lei Huang, Weijia Xu, and Ian T Foster. 2020. Convolutional neural network training with distributed K-FAC. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[42] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. 2016. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 18.

[43] Minsoo Rhu, Mike O'Connor, Niladrish Chatterjee, Jeff Pool, Youngeun Kwon, and Stephen W Keckler. 2018. Compressing DMA engine: Leveraging activation sparsity for training deep neural networks. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 78–91.

[44] Sebastian Ruder. 2016. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747* (2016).

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

[46] Christopher J Shallue, Jaehoon Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E Dahl. 2018. Measuring the effects of data parallelism on neural network training. *arXiv preprint arXiv:1811.03600* (2018).

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

[48] Leslie N Smith and Nicholay Topin. 2019. Super-convergence: Very fast training of neural networks using large learning rates. In *Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications*, Vol. 11006. International Society for Optics and Photonics, 1100612.

[49] Seung Woo Son, Zhengzhang Chen, William Hendrix, Ankit Agrawal, Wei-keng Liao, and Alok Choudhary. 2014. Data compression for the exascale computing era-survey. *Supercomputing Frontiers and Innovations* 1, 2 (2014), 76–88.

[50] Sum of normally distributed random variables. [n. d.]. https://en.wikipedia.org/wiki/Sum_of_normally_distributed_random_variables. (Accessed on 12/22/2021).

[51] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*. 1–9.

[52] Mingxing Tan and Quoc Le. 2019. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*. PMLR, 6105–6114.

[53] Dingwen Tao, Sheng Di, Zizhong Chen, and Franck Cappello. 2017. Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization. In *2017 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 1129–1139.

[54] David Taubman and Michael Marcellin. 2012. *JPEG2000 image compression fundamentals, standards and practice: image compression fundamentals, standards and practice*. Vol. 642. Springer Science & Business Media.

[55] Jiannan Tian, Sheng Di, Kai Zhao, Cody Rivera, Megan Hickman Fulp, Robert Underwood, Sian Jin, Xin Liang, Jon Calhoun, Dingwen Tao, and Franck Cappello. 2020. cuSZ: An Efficient GPU-Based Error-Bounded Lossy Compression Framework for Scientific Data. (2020), 3–15.

[56] Gregory K Wallace. 1992. The JPEG still picture compression standard. *IEEE Transactions on Consumer Electronics* 38, 1 (1992), xviii–xxxiv.

[57] Hao Wang, Naiyan Wang, and Dit-Yan Yeung. 2015. Collaborative deep learning for recommender systems. In *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*. ACM, 1235–1244.

[58] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: dynamic GPU memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 41–53.

[59] Xue Ying. 2019. An overview of overfitting and its solutions. In *Journal of Physics: Conference Series*, Vol. 1168. IOP Publishing, 022022.

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

[61] Yang You, Jonathan Hseu, Chris Ying, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. 2019. Large-batch training for LSTM and beyond. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–16.

[62] Zstandard. 2020. http://facebook.github.io/zstd/. (Accessed on 12/22/2021).