Training Recommender Systems at Scale: Communication-Efficient Model and Data Parallelism

Vipul Gupta
vipul_gupta@berkeley.edu
University of California, Berkeley
Berkeley, CA, USA

Dhruv Choudhary, Peter Tang,
Xiaohan Wei, Xing Wang,
Yuzhen Huang, Arun Kejariwal
Facebook Inc.
Menlo Park, CA, USA

Kannan Ramchandran,
Michael W. Mahoney
University of California, Berkeley
Berkeley, CA, USA

ABSTRACT

In this paper, we consider hybrid parallelism—a paradigm that employs both Data Parallelism (DP) and Model Parallelism (MP)—to scale distributed training of large recommendation models. We propose a compression framework called Dynamic Communication Thresholding (DCT) for communication-efficient hybrid training. DCT filters the entities to be communicated across the network through a simple hard-thresholding function, allowing only the most relevant information to pass through. For communication efficient DP, DCT compresses the parameter gradients sent to the parameter server during model synchronization. The threshold is updated only once every few thousand iterations to reduce the computational overhead of compression. For communication efficient MP, DCT incorporates a novel technique to compress the activations and gradients sent across the network during the forward and backward propagation, respectively. This is done by identifying and updating only the most relevant neurons of the neural network for each training sample in the data. We evaluate DCT on publicly available natural language processing and recommender models and datasets, as well as recommendation systems used in production at Facebook. DCT reduces communication by at least 100× and 20× during DP and MP, respectively. The algorithm has been deployed in production, and it improves end-to-end training time for a state-of-the-art industrial recommender model by 37%, without any loss in performance.

CCS CONCEPTS

• **Computing methodologies** → **Distributed algorithms**; *Neural networks*; • **Information systems** → Recommender systems.

KEYWORDS

Neural networks; Recommender systems; Distributed training; Hybrid parallelism

ACM Reference Format:
Vipul Gupta, Dhruv Choudhary, Peter Tang, Xiaohan Wei, Xing Wang, Yuzhen Huang, Arun Kejariwal, and Kannan Ramchandran, Michael W. Mahoney. 2021. Training Recommender Systems at Scale: Communication-Efficient Model and Data Parallelism. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '21), August 14–18, 2021, Virtual Event, Singapore.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3447548.3467080

KDD '21, August 14–18, 2021, Virtual Event, Singapore.

ACM ISBN 978-1-4503-8332-5/21/08…$15.00
https://doi.org/10.1145/3447548.3467080

1 INTRODUCTION

Data Parallelism (DP), in which each (of many) trainers stores a replica of the entire model, is a popular parallelization paradigm for the training of very large Deep Neural Networks (DNNs) [12, 38]. At the beginning of each training iteration, each worker processes a subset of entire training data with a predetermined batch size, and then each worker synchronizes the model parameters at the end of the iteration. DP has experienced widespread deployment for state-of-the-art industrial applications, but it is now facing two major challenges. The first challenge is that large batch size is needed to exploit fully the ever-increasing compute power of training nodes. This turns out to be difficult. Both theoretical and empirical evidence suggests that going beyond a certain batch size for training DNNs results in loss in generalization performance (e.g., see [6, 21, 26, 35, 41, 45, 46, 54]). Despite active research on restoring generalization performance when the batch size is large [13, 23, 25, 30, 39, 57, 69, 70], these methods either are specific to certain models and/or datasets, require extensive hyperparameter tuning, or can at best increase the maximum batch size by a small factor. The second challenge is replicating an entire DNN model on each worker, which is becoming an increasingly infeasible proposition. This is due to increasing model complexity and parameters in domains such as, but not limited to, natural language processing and recommendation systems (e.g., see [14, 27, 49, 55]), coupled with the saturation of single machine memory and compute power due to trends such as the ending of Moore's law [4, 15].

For these reasons, Model Parallelism (MP) has gained significant traction, both from the industry and the research community, as an alternative parallelization paradigm [9, 17, 24, 28, 36, 48]. In its purest form, the entire network during MP is partitioned into a number of sub-networks equal to the number of workers. While this form can accommodate a larger network than DP, it fails to capitalize on the largest batch size that is allowable before generalization performance degrades.

Hybrid Parallelism (HP)—that employs both DP and MP—is a natural next step, an idea that was arguably first introduced in [12], and more recently exploited further for large-scale DNN training [18, 19, 31, 32, 47, 51]. An illustration of hybrid training that uses MP to distribute the model across workers and DP to process multiple batches of training data at once is provided in Fig. 1. Here, each

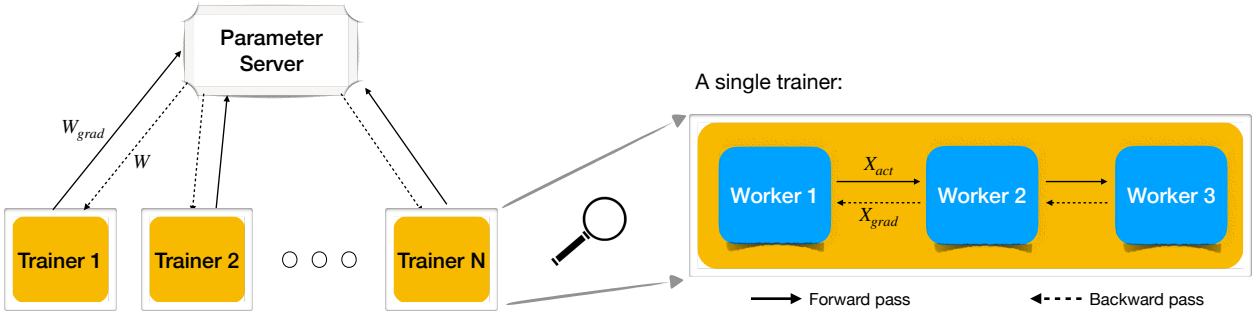


Figure 1: Distributed DNN training with hybrid training which uses both DP (left) and MP (right) for greater parallelization gains. During DP, multiple trainers process several mini-batches of data in parallel. During MP, one copy of the model is processed by one trainer which in turn is comprised of multiple workers.

partition of the network for MP is replicated in a group of workers, each processing the entire batch for that sub-network in question. Currently, hybrid training is employed in training a subset of large-scale recommendation models in production at Facebook.

The scaling of model size and batch size by HP has now progressed to the next bottleneck: communication bandwidth [48]. This bottleneck exists in two crucial places. First, for MP, activation values and gradient information need to be communicated from one sub-network to the next during forward and backward propagation. Second, for DP, gradients of the same sub-network but for different sub-batches need to be communicated, regardless of the exact operations that follow. This depends on the specific communication protocol (centralized versus decentralized reduction) or the algorithm (synchronous versus asynchronous updates). To compound the problem, increasing the batch size to fully exploit DP increases the communication of activations and gradients in MP, the sizes of which are directly proportional to the batch size. Additionally, in the asynchronous training, increasing batch size exacerbates the stale gradient problem due to an increase in the time interval between a worker receiving the model and sending the gradient [10]. In short, the benefits of communication reduction are many.

Dynamic Communication Thresholding. We propose a Dynamic Communication Thresholding (DCT) framework for communication efficient training for HP. DCT incorporates two algorithms, DCT-DP and DCT-MP, to alleviate communication congestion for DP and MP, respectively. Our algorithms filter the entities to be communicated through a simple hard-thresholding function, eliminating the need to pass many of them over the communication fabric. We propose practical methods to compute the thresholds to reduce the computational overhead of compression. Our thresholding technique is versatile, as it applies to different communication primitives in DP for the gradients, to different pipelining methods in MP (e.g., GPipe [28], PipeDream [48]), and to different applications such as recommendation systems and natural language processing models. While thresholding communication introduces errors, we apply (previously known) error compensation technique as well as a model consistency adjustment method (we developed) to mitigate the effect of the error in compression. Consequently, despite significant communication thresholding, model accuracy does not degrade, and in fact it often improves.

We apply DCT to large-scale state-of-the-art recommendation models in production with real-world datasets as well as publicly available models and datasets. We observe that the communication costs are reduced by factors of up to 20× for MP and 100× for DP. Further, end-to-end training time for large-scale training is cut by as much as 37% for industry-scale models in production. Further, applying DCT reduces the network utilization from 94.2% to 49.3% and increases the overall CPU utilization from 48.7% to 91.1%, shifting the bottleneck of model training from communication to computation in such systems.

Related Work. Due to the use of large clusters with powerful machines to train complex DNNs (e.g. BERT-Large [14] with 340M parameters), the distributed training workloads are becoming increasingly communication bound. For this reason, numerous compression schemes have been proposed in the past several years for the data parallel setting (see [67] for a comprehensive survey). These compression schemes come in various forms, such as the following: (i) Quantization, where the number of bits per entry of the communicated vector is reduced (e.g., [2, 34, 65]); (ii) Sparsification, where only a few entries of the communicated vector are sent (e.g., [1, 3, 40, 59, 60, 64]); (iii) Statistical techniques such as Randomized Sketching (e.g., [29, 33]); and (iv) Low-rank approximation, which decomposes the vector into low-rank components before communication (e.g., [11, 62, 63, 71]).

When it comes to performance on real-world systems, many of these existing schemes have one or more of the following shortcomings. (i) Focus is mostly on a theoretical analysis of schemes based on restricted assumptions, such as convexity and synchronous SGD. (ii) The empirical evaluation ignores the cost of compression and decompression which, in many cases, deprives them of any savings due to communication. (iii) Comparison of convergence with respect to baseline is reported, while the number of epochs (or iterations) and the actual training time is ignored. For instance, in Fig. 1 in [67], the authors compare the compression scheme in [33] with a baseline without compression. They observe that, although the convergence with respect to the number of epochs is unaffected due to compression, it takes almost twice the time for training to converge, rendering the scheme worse than no compression. We

also observed in our experiments that for sparsification using top-K sparsity [3, 59], the overhead of copying and sorting the large vectors ends up taking more time than the gains obtained due to communication reduction. (See Fig. 7 in Sec. 3.3 for details.)

In this paper, **we propose practical schemes for communication reduction during DP, and we show performance improvements in terms of the end-to-end DNN training times**, with performance similar to, or in some cases better than, the baseline algorithms as implemented in industry. For the MP case, existing works target the scheduling of communication of entities across the network to improve the efficiency of training DNNs [37, 53]. However, to the best of our knowledge, **this is the first work that targets communication reduction for MP by compressing the entities (i.e., activations and gradients) that are sent across the network**. As such, it can be applied on top of existing training efficiency schemes, such as communication scheduling [37, 53] and Pipelining [24, 28, 48, 68] for MP. As illustrated in Fig. 1 (right), communication is a major bottleneck for MP-based training since the activations are communicated from (say) worker 1 to worker 2 during the forward pass and the gradients are then communicated from worker 2 to worker 1 during the backward pass (similar communication happens between workers 2 and 3). However, we further observed that naively applying compression schemes, such as sparsification, quantization and sketching, to the activations and gradients either do not achieve high enough compression rates to be practical, or the degradation in model performance is beyond an acceptable level. (See Appendix A for details on such negative results.)

In the next section, we describe our algorithms for communication efficiency during parallelization, for both the MP and DP primitives of the DNN training. In particular, we discuss DCT-DP (in Section 2.1) and explain our gradient reduction technique for DP that requires minimal computational overhead for compression; and then we discuss DCT-MP (in Section 2.2), a flexible thresholding framework with theoretical support for our design. Then, Section 3 reports our findings from a diverse set of experiments and demonstrates the advantages of using DCT-DP and DCT-MP for training large-scale models for both publicly available and production models.

2 COMMUNICATION-EFFICIENT TRAINING WITH HYBRID PARALLELISM

We start, in Section 2.1, by proposing a Dynamic Communication Thresholding (DCT) technique for DP (DCT-DP). DCT-DP is inspired by existing theoretical works such as [59] and [3]. It sparsifies the gradient in each iteration before sending it over the wire, and it intelligently chooses the threshold for sparsification to reduce the computational overhead introduced due to compression and decompression. Then, in Section 2.2, we propose DCT-MP, a novel thresholding scheme for sparsification of activations and gradients during forward and backward passes, respectively, to reduce communication during MP.

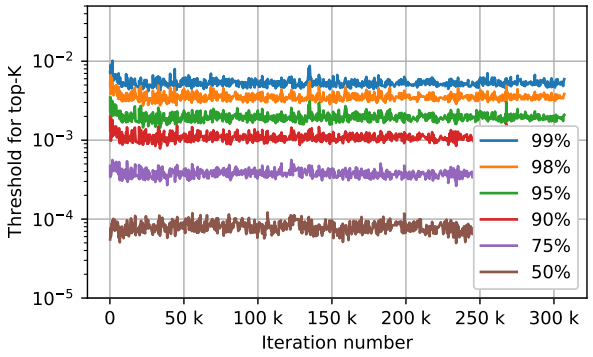


Figure 2: Top-K threshold for various levels of sparsity during the gradient compression for DCT-DP. We see that the top-K thresholds, for different sparsity levels, do not deviate much from the mean. Thus, updating the threshold only every $L(>1)$ iterations can help reduce the overhead of sorting to find the top-K threshold.

2.1 DCT-DP: Reducing communication for Data Parallelism

During DP, as illustrated in Fig. 1 (left), we compress the gradient, W_{grad}, from trainers to the parameter server to improve the communication bottleneck. Our compression algorithm, DCT-DP, is inspired by previous works which focus on data-parallel training for alleviating communication bottlenecks, and in particular the works of [3, 59], where error feedback is employed along with sparsification to correct the error in gradient direction due to compression. Such schemes find a top-K threshold by sorting the gradient vector, and they use the threshold to sparsify the gradients by keeping only the top-K entries. However, they focus on proving theoretical convergence guarantees, and they do not show improvements in end-to-end times for training neural networks.

In our experiments, we observed that the overhead of allocating memory to copy the gradient (with its size easily scaling into the millions) and sorting the resultant copy to find the top-K threshold in each iteration is sufficiently expensive that it deprives any improvements in end-to-end training time in real-world systems (see Sec. 3.3 for details). Hence, such gradient compression schemes, in their most basic form, cannot be employed directly to obtain promised gains in training efficiency. However, we take advantage of the following observation to reduce the overhead introduced due to compression.

In Fig. 2, we plot the top-K thresholds for various levels of sparsity for the Deep Learning Recommendation Model (DLRM) [49] with the Criteo Ad Kaggle Dataset for one of the Fully Connected (FC) layers (see Sec. 3.1 for details on the training process). We see that the threshold value increases as the sparsity increases, which is expected. More importantly, we note that given a sparsity factor, the threshold value does not vary much across iterations. For example, for 95% sparsity, the threshold deviates by at most 26% around its running mean. Thus, even for reasonably large compression factors, updating the threshold every iteration is excessive.

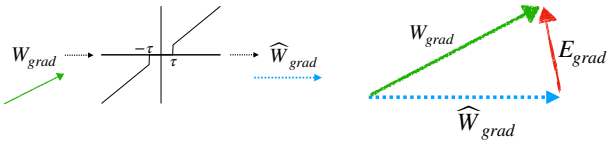


Figure 3: A illustration of DCT-DP. First, $W_{grad} \in \mathbb{R}^N$ (which already incorporates error from the previous iteration) is compressed using a threshold τ to obtain the sparse vector $\widehat{W}_{grad}$. Then, the error is calculated as $E_{grad} = W_{grad} - \widehat{W}_{grad}$ to be used in the next iteration to correct the error in gradient direction.

Algorithm 1 DCT-DP: Communication-Efficient Data Parallelism

Input: Sparsity factor η ($0 < \eta \leq 1$), Threshold life-span L, Iteration number k, Gradient of the DNN layer $W_{grad} \in \mathbb{R}^N$, Error $E_{grad} \in \mathbb{R}^N$, and Threshold τ (from iteration $k-1$)
Error Feedback: $W_{grad} = W_{grad} + E_{grad}$
if L divides k **then**
  $[w_1, w_2, \cdots, w_N] = Sort(|W_{grad}|)$
  Assign $\tau = w_{\lfloor N \times \eta \rfloor}$
else
  Use τ from iteration $k-1$
end if
Compute mask $M = \mathbb{I}(|W_{grad}| \geq \tau)$
Compute compressed gradient $\widehat{W}_{grad} = W_{grad} \odot M$
Compute error $E_{grad} = W_{grad} - \widehat{W}_{grad}$
Send $\widehat{W}_{grad}$ to the parameter server which updates the model

Inspired by this observation, we update the threshold only once every L iterations (where L is generally in thousands) while compressing the gradient of the parameters, W_{grad}, for each DNN layer. We refer to L as the threshold life-span. As we observe in our experiments (see Sec. 3.3), we can compress the gradients by as much as 99% sparsity with $L = 1000$ for each layer using top-K sparsification and error correction without any loss in performance. Our algorithm is illustrated in Fig. 3 and detailed steps are provided in Algorithm 1. Throughout this paper, the function $\mathbb{I}(\cdot)$ denotes the indicator function, and the symbols $\lfloor \cdot \rfloor$ and $\odot$ denote the integer floor and element-wise product of two matrices, respectively.

Note that each trainer consists of multiple workers, and each worker compresses the gradients layer-wise using sparsification before communication (see Fig. 1 for an illustration, where each trainer consists of 3 workers). This is unlike existing works (e.g. Ivkin et al. [29], Stich et al. [59]) where the gradient vectors of all the model parameters are combined and compressed together. However, the theoretical guarantees on the convergence of the algorithm still holds and can be trivially extended to our case. This is because, for any threshold $\tau > 0$, the compressed gradient satisfies the contraction property (Definition 2.1 in Stich et al. [59]). Hence, DCT-DP satisfies the same rate of convergence as Stochastic Gradient Descent (SGD) without compression (see Theorem 2.4 in Stich et al. [59]).

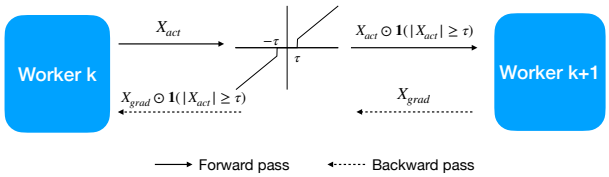


Figure 4: A illustration of DCT-MP. During the forward pass, we sparsify and compress the activations, say X_{act}, corresponding to one data sample, using the mask, $\mathbb{I}(|X_{act}| \geq \tau)$, is generated based on the threshold τ. During the backward pass, the same mask is used to compress the gradients and selectively train neurons.

2.2 DCT-MP: Reducing communication for Model Parallelism

Training of large-scale DNNs is often regarded with pessimism due to its associated training latency (multiple days/weeks). However, training such large-scale models can be a "blessing in disguise" from a communication-efficiency point of view. For such models, with billions of parameters in each layer, only a few of the neurons are activated during the forward pass, potentially allowing us to compress these activations by a factor of 20× or more with no loss in model performance. This idea of training only a subset of neurons every iteration based their activation values stems from several existing observations [8, 43, 44]. In fact, in works such as dropout [58] and adaptive dropout [5], the authors have shown that selective sparsification can improve the generalization performance due to implicit regularization [42]. With such a scheme, we also observe gains in generalization performance on top of communication efficiency (see experiments in Section 3).

Motivated by this, we propose a sparsification scheme where the neurons compete with each other in every iteration during DNN training, and the ones with the largest (absolute) value of activations are selected. Thus, for a given training sample, DCT-MP selects only a few neurons (say ~5%) during the forward pass that are generally sufficient to represent the entire information for that training sample. We next describe DCT-MP in more detail.

Algorithm. Let the mini-batch size be B and the number of output features before the model split be d. Thus, the activation and gradient matrices (X_{act} and X_{grad}, respectively) lie in $\mathbb{R}^{B \times d}$. Based on the idea that each example activates only a subset of neurons, we select a fraction, say η, of largest entries according to their absolute value in each row. Thus, for the i-th row of X_{act}, say $X_{act,i}$, we select a threshold τ_i which is greater than $d \times \eta$ values in $X_{act,i}$, and the mask is thus calculated for the i-th data sample as $\mathbb{I}(X_{act,i} \geq \tau_i)$. The same mask is then used to compress the entities $X_{act,i}$ and $X_{grad,i}$ during forward and backward passes, respectively, for all $i \in \{1, 2, \cdots, B\}$. Thus, the training for each mini-batch happens only on the relevant neurons corresponding to each sample in the training data. In Fig. 4, we illustrate the compression using DCT-MP when the mini-batch size is one. Detailed steps for a general mini-batch size B are provided in Algorithm 2.

DCT-MP Promotes Sparsity in Model Activations. In Fig. 5, we plot the mean, $\frac{1}{B}\sum_{i=1}^{B} \tau_i$, of threshold vector $\tau = [\tau_1, \tau_2, \cdots, \tau_B]$ with respect to the number of iterations for the DLRM model with

Algorithm 2 DCT-MP: Communication-Efficient Model Parallelism

Input: Sparsity factor η ($0 < \eta \leq 1$),
Forward Pass:
Input: Activation matrix $X_{act} = [X_{act,i}]_{i=1}^{B} \in \mathbb{R}^{B \times d}$
Define the mask, $M = [\ ]$
for $i = 1$ **to** B **do**
  $[x_1, x_2, \cdots, x_d] = Sort(|X_{act,i}|)$
  Define $\tau_i = x_{\lfloor d \times \eta \rfloor}$
  $m_i = \mathbb{I}(|X_{act,i}| \geq \tau_i)$
  $M = [M;\ m_i]$
end for
Compute the sparse matrix $X_{act} \odot M$
Send $X_{act} \odot M$ and the mask M across the network
Backward Pass:
Input: Gradient matrix $X_{grad} \in \mathbb{R}^{B \times d}$
Compute the sparse matrix $X_{grad} \odot M$
Send $X_{grad} \odot M$ across the network

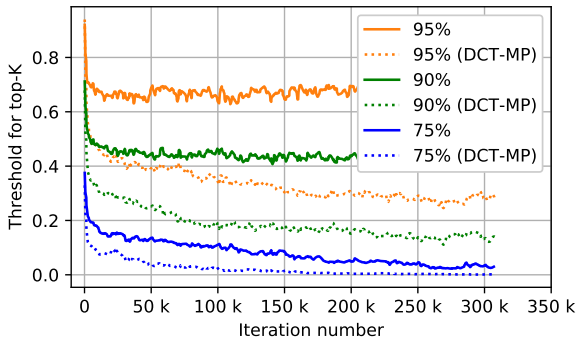


Figure 5: Top-K threshold for various levels of sparsity for the cases when compression using DCT-MP is applied and when it is not applied. The top-K thresholds decrease significantly when DCT-MP is applied. Thus, DCT-MP induces sparsity in neuron activations. This is possibly the reason for its improved generalization performance.

the Criteo Ad Kaggle Dataset. The threshold is calculated for activations after one of the fully connected layers (see Sec. 3.1 for details on the experimental setup). The mean of the threshold is calculated for different sparsity levels (75%, 90% and 95%) for the two cases when sparsification using DCT-MP is applied (dotted lines) and when it is not applied (solid lines). Thus, the solid lines correspond to a single training run where we are simply measuring the mean of top-K threshold values without actually sparsifying the activations sent across the wire. The dotted lines with different sparsification levels correspond to different training runs where the stated sparsification is actually applied to the activations (and gradients) that are sent across the wire.

We observe that, as the training progresses, the top-K thresholds decrease significantly faster for the case when DCT-MP is applied. A decrease in the top-K threshold corresponds to the activations getting sparser (maybe approximately) as the training progresses. Thus, DCT-MP induces sparsity in activations while training, which is exploited for communication efficiency. An important advantage of such sparsity-inducing regularization is the improved generalization performance of the model, as shown in our experiments in Sec. 3. Our conjectured explanation for why sparsity helps in improving the generalization error is based on the performance of existing popular schemes. This includes dropout (see Fig. 8, Srivastava et al. [58]) and Rectified Linear Units (ReLU) (see Fig. 3, Glorot et al. [20]), which themselves introduce sparsity in model activations, as well as implementations of implicit sparsity based methods in scalable algorithms for graph analysis [16, 56].

Analysis of DCT-MP. To provide further insights into DCT-MP, we prove that the stochastic gradient obtained with Algorithm 2 is equal, in expectation, to the stochastic gradient obtained in a network without any communication thresholding. More details of this unbiased estimation, including a formal statement of the theorem and its proof, are provided in Appendix B.

Comparision with Dropout. Dropout and DCT-MP are similar in essence as they both selectively train neurons. However, the two schemes are different: both in the goals they try to achieve, and in the mechanisms they use. Furthermore, they can be used complementarily. Here are the main differences between the two schemes. First, Dropout drops neurons randomly, while DCT-MP keeps only the most relevant neurons for each training sample. Second, for Dropout, going beyond 50% sparsity results in accuracy loss, but DCT-MP achieves up to 95% sparsification. Third, Dropout is applied to every parameter layer, but DCT-MP is applied only to the layers before the model split.

3 EMPIRICAL RESULTS

In this section, we investigate DCT-MP and DCT-DP for three different experimental setups. In subsections 3.1 and 3.2, we evaluate the performance of DCT-MP on the Deep Learning Recommendation Model (DLRM) and a Natural Language Processing (NLP) model, respectively, for different levels of compression and different number of MP workers. The models and datasets are publicly available. We show that high compression factors can be obtained (up to ~95%) with DCT-MP along with small improvements in model performance.

We further evaluate DCT-DP on the DLRM model in subsection 3.1 and see no loss in performance with up to 98% sparsity. Finally, we evaluate the performance of DCT-DP and DCT-MP on large-scale recommendation models that are trained with hybrid parallelism in production systems. We show that the deployed algorithm reduces the training time by 37% for such production-scale models without any performance loss.

Further, in all our experiments, we tried to show at least one negative result that would provide insights into the scalability of DCT. For instance, as the number of workers for MP (i.e., model splits) increases, the compression factor with DCT-MP decreases (e.g., Tables 1, 3, 4 and 5).

3.1 Experiments on the DLRM Model

Experimental Setup. For these experiments, we use the DLRM model from [49]. In this model, the dense features are first processed by a Multilayer Perceptron (MLP) with four layers, where each

layer contains a Fully Connected (FC) layer followed by a Rectified Linear Unit (ReLU). Then, there is a feature interaction between the processed dense and sparse features, which goes through a second MLP with four layers (the last layer has Sigmoid instead of ReLU as the non-linearity) to produce the final output. In our experiments, the embedding dimension for sparse features was kept at 16, and the output dimensions of the four FC layers in the first MLP are 512, 256, 64 and 16, respectively. Similarly, for the second MLP, the output dimensions for the fours FC layers are 512, 256, 128 and 1, respectively.[1] Training and testing sets comprise of 6 days and one day, respectively, of the Criteo Ad Kaggle dataset.[2]

Fig. 6 provides an illustration of MP with the DLRM model. The shaded area in blue shows a sample partition for MP. In our simulations, we consider up to two splittings of the DLRM model. The first split is after two layers in the first MLP, and the second split is after two layers in the second MLP. Our goal is to reduce communication across different workers (both during the forward and backward passes). This is a typical setup in MP Training where workers 1, 2, and 3 can be the different pipes of a single trainer (e.g., see Huang et al. [28]). For all our experiments, the data shuffle remains constant across different training runs.

In Fig. 6, we mark the three entities that are sent across the network which we compress to alleviate communication costs in distributed DNN training. X_{act} and X_{grad} are the activation and gradient matrices sent across the network during the forward pass and backward passes, respectively. The third entity that can be compressed is the parameter gradient (shown as W_{grad}) that is sent from Workers 1, 2, and 3 to the parameter server. This keeps a central copy of weights and updates it regularly through the gradients received from different workers.

Table 1: DCT-MP on the DLRM model: Train and Test Loss and Accuracy for multiple sparsity ratios (denoted by η) and different settings for MP.

η	MP Workers	Train Loss	Train Acc (%)	Test Loss	Test Acc (%)
0%	–	0.4477	79.23	0.4538	78.78
75%	2	0.4473	79.29	0.4532	78.81
90%	2	0.4472	79.28	0.4530	78.81
95%	2	0.4473	79.24	**0.4534**	78.80
98%	2	0.4505	79.07	0.4562	78.61
75%	3	0.4482	79.19	0.4536	78.79
90%	3	0.4479	79.24	**0.4537**	78.78
95%	3	0.4495	79.18	0.4546	78.72

In Table 1, we show the cross-entropy loss [22] and accuracy with the DLRM model on the training and testing data samples. A sparsity factor (η) of 0% denotes the baseline with no compression. We consider two settings for MP: one split (that is, 2 MP workers); and two splits (or three workers for MP).

MP with two workers (one split). In rows 2-5 in Table 1, we consider one split in the model (or MP with two workers) in the first MLP after two layers. We see that even with 95% sparsity (that is, 20× compression) on X_{act} (and X_{grad}) sent across the network, we are able to perform better than baseline (with no compression), both in terms of train and test loss (highlighted in bold cases). However, we see a tangible loss in performance when the sparsity is further increased to 98%.

MP with three workers (two splits). In rows 6-8 in Table 1, we consider MP with 3 workers, where the two model splits are in the first and second MLP, as shown in Fig. 6. Note that, in the case of two splits, compressing the entities that are sent across the network by up to 90% does not affect the test accuracy, and it is still better than the baseline with no compression. However, increasing the sparsity factor to 95% is too ambitious for the two split case, and it increases the test loss by 0.18%. Further increasing the number of splits results in a greater performance loss, and the performance is worse than baseline for even 75% sparsity.

Remark 1. *We emphasize that for all the experiments in this paper, the location of splits for MP were not tuned as hyperparameters. Instead, we inserted splits after randomly chosen FC layers, or after the ReLU following the FC layer if it exists. The advantage of inserting a split after ReLU layers is that the activation matrix is* 50% *sparse on average, resulting in higher compression rates for DCT-MP.*

Table 2: DCT-DP on the DLRM model: Train and Test Loss and Accuracy for various levels of sparsity.

Sparsity Factor	Train Loss	Train Acc (%)	Test Loss	Test Acc (%)
Baseline	0.4477	79.23	0.4538	78.78
75%	0.4478	79.23	0.4534	78.81
90%	0.4478	79.22	0.4536	78.79
95%	0.4479	79.25	0.4538	78.79
98%	0.4478	79.23	**0.4537**	78.80
99.5%	0.4482	79.20	0.4547	78.75

DP with the DLRM Model. In Table 2, we illustrate the performance of DCT-DP on DLRM by compressing the gradients of the parameters of all the 8 FC layers while they are sent across the wire to the parameter server. The parameter server then updates the model parameters using the compressed gradient. We use error feedback [34] to compensate for the error in gradient compression by feeding it back to the gradients in the next iteration. In general, DCT-DP compression enjoy higher compression rates due to the use of error compensation schemes and the fact that error in one layer does not propagate to the other layers, unlike in the case of MP compression. Compression up to 98% sparsity does not show any loss in performance. However, further compressing to 99.5% sparsity increases the test loss by 0.20%.

Communication-efficient Hybrid Training. Next, we apply compression to W_{grad} for the 8 FC layers (in the DP case) and to X_{act} (and X_{grad}) for two splits (in the MP case) and present our results in Table 3. We see that compression up to 90% sparsity (both during DP and MP) does not affect the performance, but the test loss increases by 0.22% when the sparsity factor is increased to 95%.

[1] See the Criteo Kaggle benchmark for further details on the training process: https://github.com/facebookresearch/dlrm

[2] https://labs.criteo.com/2014/02/kaggle-display-advertising-challenge-dataset/

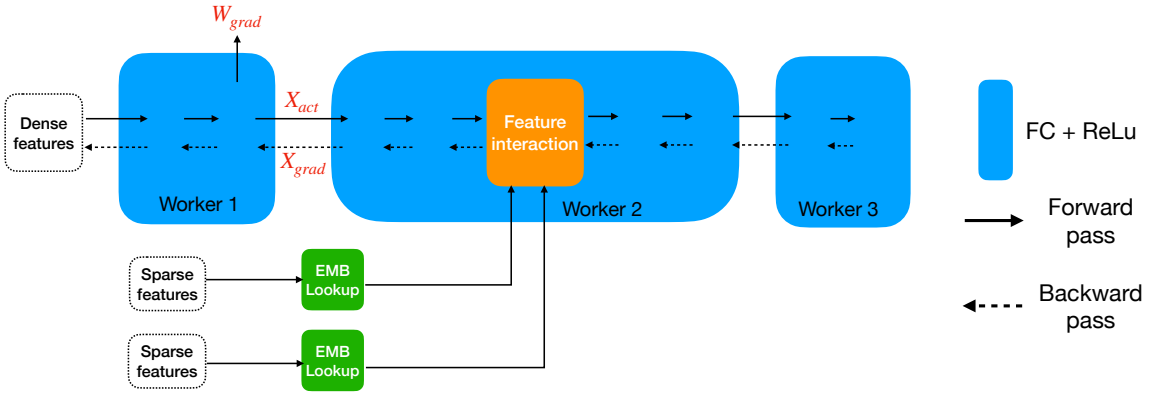


Figure 6: A illustration of model parallelism with DLRM. The entities that are sent across the network are shown in red. X_{act} and X_{grad} are communicated during MP, and W_{grad} is communicated during DP. The shaded area in blue represents a sample model partitioning for MP. In this case, three workers are working on one copy of the model during MP and comprise a trainer.

Table 3: Compression using DCT-DP and DCT-MP on the DLRM model: Train and Test Loss and Accuracy with two MP splits (that is, three workers for MP).

Sparsity Factor	Train		Test	
	Loss	Acc (%)	Loss	Acc (%)
Baseline	0.4477	79.23	0.4538	78.78
75%	0.4480	79.23	0.4535	78.81
90%	0.4481	79.26	**0.4537**	78.78
95%	0.4492	79.19	0.4548	78.70

3.2 Experiments on a Translation Model

For our experiments with DCT-MP, we next consider the Transformer translation model as an application of NLP using DNNs. We train over the IWSLT'14 German to English dataset [7]. The setup and hyperparameters were directly borrowed from the fairseq NLP Library [50]. The model used was borrowed from [61], where both encoder and decoder have 6 layers, each of which uses a fully connected Feed-Forward Network (FFN) with input and output dimensionality of 512 and inner layer dimensionality of 1024.[3] We report the training and testing losses and the BLEU scores after 50 epochs of training.

Our results with DCT-MP on the translation model are described in Table 4. We consider three training scenarios: Two MP workers (with one split), Three MP workers (with two splits), and Five MP workers (with 4 splits). For the case with one split, we inserted the DCT-MP operator after the ReLu operator in the FFN of the fifth encoder layer. For the two splits case, we additionally inserted the DCT-MP operator after the ReLu operator in the FFN of the fifth encoder layer. We further added two splits after the ReLu operator in the third FFN in both the encoder and decoder layers for the four splits case. For each scenario, we show the best performing sparsity factor in bold.

[3]For further details on the translation model, dataset preprocessing and the hyperparameters used, see https://github.com/pytorch/fairseq/tree/master/examples/translation

We emphasize that no hyperparameter tuning was performed in choosing the splits, and we observed in our experiments that using DCT-MP after an FC Layer or a ReLu layer improves the generalization performance, possibly due to (implicitly) added regularization (as illustrated in Fig. 5). Note that we can add more MP splits for the NLP model compared to the DLRM model since the model is significantly deeper (and thus less susceptible to changes in outputs of a few layers) with larger FC layers (thus allowing for greater sparsity). This shows that DCT-MP is more beneficial for wider and/or deeper models (that is, typical setups where MP is used).

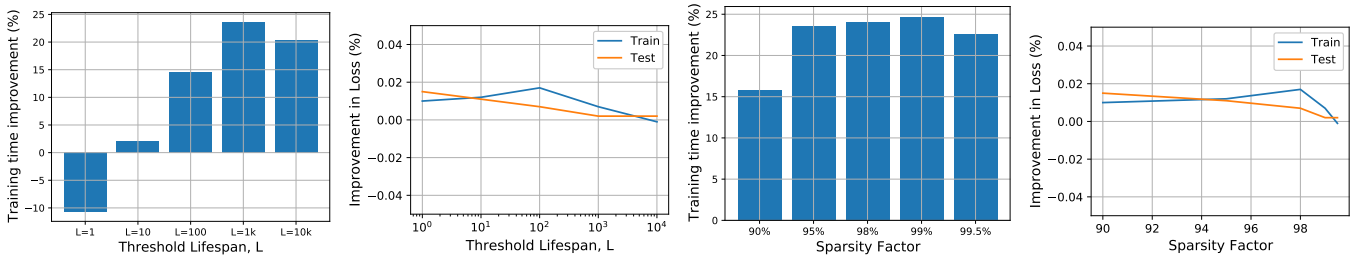
Table 4: DCT-MP on a translation model with IWSLT'14 dataset: Train and Test Losses and BLEU scores for various levels of sparsity and different splits for MP.

Sparsity Factor	MP Workers	Train Loss	Test Loss	BLEU Score
Baseline	–	3.150	3.883	35.17
90%	2	3.159	3.879	35.23
95%	2	3.157	**3.882**	35.18
90%	3	3.151	3.881	35.22
95%	3	3.148	**3.882**	35.19
90%	5	3.157	**3.882**	35.20
95%	5	3.188	3.890	35.15

In this subsection, we do not consider DCT-DP since similar schemes have been evaluated for NLP models in existing works such as [67] and [1]. In the next subsection, we evaluate DCT-MP and DCT-DP on large-scale recommendation models for end-to-end training times and overall model performance.

3.3 Large-Scale Recommendation System

We present our results for a real-world large scale recommendation system that employs HP for parallelization on click-through rate prediction task. We employ DCT-MP and DCT-DP to reduce the network bandwidth usage in these systems.



(a) Training time improvements for different values of L. **(b) Train and test error improvements for different values of L.** **(c) Training time improvements for various sparsity factors.** **(d) Train and test error for various sparsity factors.**

Figure 7: DCT-DP on Large-Scale Recommendation Models. Figures (a) and (b) show the training time and loss improvements, respectively, over baseline for different values of the threshold life-span, L, for a sparsity level of 95%. Figures (c) and (d) show the same statistics for various levels of sparsity for $L = 1000$.

Experimental Setup. We leverage a distributed data-parallel asynchronous training system with multiple trainers to train a recommendation model. Each trainer in the DP setup may consist of one or more workers that use MP (see Fig. 1 for an illustration). Typically, the model is split into 10 or more parts and fine-grained parallelism is employed for high throughput. Hence, the worker machines suffer from very high communication cost for both MP and DP. The batch sizes are usually in the range of 100-1000, but they are employed with hogwild threads (see Recht et al. [52]) to increase the throughput of the system, further exacerbating the communication cost problem. The recommendation model considered in this section takes multiple days to train with general-purpose CPU machines. All the workers and parameter servers run on Intel 18-core 2GHz processors with 12.5Gbit Ethernet. The hardware configurations are identical and consistent across all the experiments. We train using 7B training examples and evaluate the model on 0.5B examples. For quantifying model performance, we report the cross-entropy loss from the classification task. We compare relative cross-entropy loss and end-to-end training times of the proposed techniques with respect to a baseline model without communication compression.

DCT-DP with Large-Scale Recommendation Model. Figure 7 shows the results of applying DCT-DP on the large-scale recommendation model. In Figure 7a, we plot the improvements in end-to-end training times when DCT-MP is applied to compress the parameter gradients, W_{grad}, that are sent to the parameter server. Here, we keep the sparsity level constant at 95% and vary the threshold life-span L (the interval after which the top-K threshold is updated). We note that compression with $L = 1$ takes 11% more time than the baseline with no compression. This is due to the cost of the copy-and-sort routine which computes the top-K threshold.[4] Increasing L to 1000 trains the model 23% faster and further increasing it to 10000 does not provide any additional gain. Figure 7b illustrates that for different values of L, the train and test losses are within 0.01% of the baseline performance.

Fig. 7c shows the improvement in training time for various levels of sparsity when the threshold life span is kept constant at $L = 1000$. We observe the general trend that when the sparsity is increased, the training time improves. Overall, we are able to compress the gradients to sparsity factors of up to 99.5% without any loss in train and test performance (as noted from Fig. 7d). However, we do not see significant improvements in training time beyond the sparsity level of 95%, possibly because the message size is small enough to not hurt bandwidth usage, and the only cost remaining is the fixed latency cost associated with sending any message, irrespective of its size.

Remark 2. *We observe that error feedback works very well in this asynchronous data-parallel training paradigm with a larger number of hogwild threads. Note that this should not be expected since existing works prove convergence guarantees only for the synchronous SGD settings. An implementation detail that helped was sharing the error feedback buffer between the multiple threads. But this can lead to a fast growing magnitude of error in the buffer leading to stale updates. To avoid this, we drain the error feedback buffer stochastically every 1 million iterations.*

DCT-MP with Large-Scale Recommendation Model. We employ DCT-MP to compress the entities sent through the network during MP for communication efficiency. DCT-MP is applied across the 12 splits of the model after the ReLU layer. Our results are summarized in Table 5. We show improvement in training and test losses[5] in columns 2 and 3, respectively, and the improvements in end-to-end training times in column 4 for various levels of sparsity. We observe that the training performance slightly degrades with DCT-MP on large-scale models. However, the test performance improves up to sparsity levels of 90%, with a 14% improvement in end-to-end training time. Increasing the sparsity level to 95% degrades the test performance by 0.121%. Note that we can further improve the performance of DCT-MP by identifying the layers whose activations are sensitive to sparsification and avoiding compressing them during DCT-MP (or changing the location of the split). However, such selectivity in choosing layers for DCT-MP is beyond the scope of this paper.

[4]Note that $L = 1$ represents the scheme proposed in popular works such as [59] and [3]. Thus, naively implementing existing schemes for top-K sparsification might not always yield expected gains in production. However, as we observe later, simply updating L every thousand iterations can improve the training time by 25% without any loss in performance.

[5]Positive numbers imply better performance.

Table 5: DCT-MP on a large-scale recommender model

Sparsity Factor	Loss Improvement (%) Train	Test	Time Gain (%)
Baseline	0.000%	0.000%	0.000%
75%	-0.006%	0.023%	7.04%
90%	-0.021%	**0.016%**	**13.95%**
95%	-0.070%	-0.121%	14.43%

Communication-Efficient Hybrid training. Next, we apply both DCT-DP and DCT-MP for communication reduction during hybrid training of a large-scale recommendation model. Inspired by our previous results, we chose the sparsity levels as 90% and 99% for DCT-MP and DCT-DP (with $L = 1000$), respectively. We observe a 37.1% reduction in end-to-end training time, with train and test loss within 0.01% of the baseline model that does no compression.

Further, before applying DCT, we observed that the network utilization was high (94.2%) and the CPU utilization was low (48.7%), implying that communication is a bottleneck. However, after applying DCT, CPU utilization increased to 91.1% and network utilization decreased to 49.3%, implying that DCT shifted the bottleneck from communication to computation in production models.

4 CONCLUSIONS

Inspired by the fact that communication is increasingly becoming the bottleneck for large-scale training, we proposed two practical algorithms, DCT-DP and DCT-MP, to reduce the communication bottleneck during data and model parallelism, respectively, for fast training of DNN models. DCT-DP and DCT-MP improve end-to-end training time by sparsifying the matrices to be sent across the wire by appropriately selecting a sparsification threshold. We empirically evaluated the proposed algorithms on publicly-available as well as industry-scale models and datasets. We show a reduction in communication for MP and DP by up to 20× and 100×, respectively, without any loss in performance. Further, the end-to-end training time reduces by 37% in production models. Further, our algorithms reduce the network bandwidth utilization by half and almost double the CPU utilization, shifting the training bottleneck from communication to computation.

REFERENCES

[1] Alham Fikri Aji and Kenneth Heafield. 2017. Sparse Communication for Distributed Gradient Descent. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. 440–445.

[2] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2017. QSGD: Communication-efficient SGD via gradient quantization and encoding. In *Advances in Neural Information Processing Systems*. 1709–1720.

[3] Dan Alistarh, Torsten Hoefler, Mikael Johansson, Nikola Konstantinov, Sarit Khirirat, and Cédric Renggli. 2018. The convergence of sparsified gradient methods. In *Advances in Neural Information Processing Systems*. 5973–5983.

[4] Akhil Arunkumar, Evgeny Bolotin, Benjamin Cho, Ugljesa Milic, Eiman Ebrahimi, Oreste Villa, Aamer Jaleel, Carole-Jean Wu, and David Nellans. 2017. MCM-GPU: Multi-chip-module GPUs for continued performance scalability. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 320–332.

[5] Jimmy Ba and Brendan Frey. 2013. Adaptive dropout for training deep neural networks. In *Advances in neural information processing systems*. 3084–3092.

[6] 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.

[7] Mauro Cettolo, Christian Girardi, and Marcello Federico. 2012. Wit3: Web inventory of transcribed and translated talks. In *Conference of european association for machine translation*. 261–268.

[8] Beidi Chen, Tharun Medini, James Farwell, Sameh Gobriel, Charlie Tai, and Anshumali Shrivastava. 2019. Slide: In defense of smart algorithms over hardware acceleration for large-scale deep learning systems. *preprint arXiv:1903.03129* (2019).

[9] Chi-Chung Chen, Chia-Lin Yang, and Hsiang-Yun Cheng. 2018. Efficient and robust parallel dnn training through model parallelism on multi-gpu platform. *arXiv preprint arXiv:1809.02839* (2018).

[10] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. 2016. Revisiting distributed synchronous SGD. *preprint arXiv:1604.00981* (2016).

[11] Minsik Cho, Vinod Muthusamy, Brad Nemanich, and Ruchir Puri. 2019. GradZip: Gradient Compression using Alternating Matrix Factorization for Large-scale Deep Learning.

[12] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. 2012. Large scale distributed deep networks. In *Advances in neural information processing systems*. 1223–1231.

[13] Aditya Devarakonda, Maxim Naumov, and Michael Garland. 2017. AdaBatch: Adaptive Batch Sizes for Training Deep Neural Networks. *CoRR* abs/1712.02029 (2017). arXiv:1712.02029 http://arxiv.org/abs/1712.02029

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

[15] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *2011 38th Annual international symposium on computer architecture (ISCA)*. IEEE, 365–376.

[16] K. Fountoulakis, D. F. Gleich, and M. W. Mahoney. 2017. An optimization approach to locally-biased graph algorithms. *Proc. IEEE* 105, 2 (2017), 256–272.

[17] Jinkun Geng, Dan Li, and Shuai Wang. 2019. ElasticPipe: An Efficient and Dynamic Model-Parallel Solution to DNN Training. In *Proceedings of the 10th Workshop on Scientific Cloud Computing* (Phoenix, AZ, USA) *(ScienceCloud '19)*. Association for Computing Machinery, New York, NY, USA, 5–9.

[18] Jinkun Geng, Dan Li, and Shuai Wang. 2019. Horizontal or vertical? a hybrid approach to large-scale distributed machine learning. In *Proceedings of the 10th Workshop on Scientific Cloud Computing*. 1–4.

[19] Amir Gholami, Ariful Azad, Peter Jin, Kurt Keutzer, and Aydin Buluc. 2018. Integrated model, batch, and domain parallelism in training neural networks. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*. 77–86.

[20] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. 2011. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. 315–323.

[21] Noah Golmant, Nikita Vemuri, Zhewei Yao, Vladimir Feinberg, Amir Gholami, Kai Rothauge, Michael W Mahoney, and Joseph Gonzalez. 2018. On the computational inefficiency of large batch sizes for stochastic gradient descent. *arXiv preprint arXiv:1811.12941* (2018).

[22] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning*. Vol. 1.

[23] 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. *preprint arXiv:1706.02677* (2017).

[24] Lei Guan, Wotao Yin, Dongsheng Li, and Xicheng Lu. 2019. XPipe: Efficient Pipeline Model Parallelism for Multi-GPU DNN Training. *arXiv preprint arXiv:1911.04610* (2019).

[25] Vipul Gupta, Santiago Akle Serrano, and Dennis DeCoste. 2020. Stochastic Weight Averaging in Parallel: Large-Batch Training That Generalizes Well. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rygFWAEFwS

[26] Elad Hoffer, Itay Hubara, and Daniel Soudry. 2017. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. In *NIPS*.

[27] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4700–4708.

[28] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Advances in Neural Information Processing Systems 32*. 103–112.

[29] Nikita Ivkin, Daniel Rothchild, Enayat Ullah, Vladimir Braverman, Ion Stoica, and Raman Arora. 2019. Communication-efficient distributed sgd with sketching. In *Advances in Neural Information Processing Systems*. 13144–13154.

[30] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, et al. 2018. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *arXiv preprint arXiv:1807.11205* (2018).

[31] Zhihao Jia, Sina Lin, Charles R Qi, and Alex Aiken. 2018. Exploring hidden dimensions in parallelizing convolutional neural networks. *arXiv preprint arXiv:1802.04924* (2018).

[32] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond data and model parallelism for deep neural networks. (2019).

[33] Jiawei Jiang, Fangcheng Fu, Tong Yang, and Bin Cui. 2018. SketchML: Accelerating distributed machine learning with data sketches. In *Proceedings of the 2018 International Conference on Management of Data*. 1269–1284.

[34] Sai Praneeth Karimireddy, Quentin Rebjock, Sebastian Stich, and Martin Jaggi. 2019. Error Feedback Fixes SignSGD and other Gradient Compression Schemes. In *International Conference on Machine Learning*. 3252–3261.

[35] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. 2016. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836* (2016).

[36] Jin Kyu Kim, Qirong Ho, Seunghak Lee, Xun Zheng, Wei Dai, Garth A. Gibson, and Eric P. Xing. 2016. STRADS: A Distributed Framework for Scheduled Model Parallel Machine Learning *(EuroSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 5, 16 pages.

[37] Seunghak Lee, Jin Kyu Kim, Xun Zheng, Qirong Ho, Garth A Gibson, and Eric P Xing. 2014. On model parallelization and scheduling strategies for distributed machine learning. In *Advances in neural inf. processing systems*. 2834–2842.

[38] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 583–598.

[39] Tao Lin, Sebastian U. Stich, Kumar Kshitij Patel, and Martin Jaggi. 2020. Don't Use Large Mini-batches, Use Local SGD. In *International Conference on Learning Representations*. https://openreview.net/forum?id=B1eyO1BFPr

[40] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and Bill Dally. 2018. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. In *International Conference on Learning Representations*.

[41] Siyuan Ma, Raef Bassily, and Mikhail Belkin. 2018. The power of interpolation: Understanding the effectiveness of SGD in modern over-parametrized learning. In *International Conference on Machine Learning*. PMLR, 3325–3334.

[42] M. W. Mahoney. 2012. Approximate Computation and Implicit Regularization for Very Large-scale Data Analysis. In *Proceedings of the 31st ACM Symposium on Principles of Database Systems*. 143–154.

[43] Alireza Makhzani and Brendan Frey. 2013. K-sparse autoencoders. *arXiv preprint arXiv:1312.5663* (2013).

[44] Alireza Makhzani and Brendan J Frey. 2015. Winner-take-all autoencoders. In *Advances in neural information processing systems*. 2791–2799.

[45] Charles H. Martin and Michael W. Mahoney. 2018. *Implicit Self-Regularization in Deep Neural Networks: Evidence from Random Matrix Theory and Implications for Learning*. Technical Report Preprint: arXiv:1810.01075.

[46] Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. 2018. An empirical model of large-batch training. *arXiv preprint arXiv:1812.06162* (2018).

[47] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V Le, and Jeff Dean. 2018. A hierarchical model for device placement. In *International Conference on Learning Representations*.

[48] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 1–15.

[49] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. 2019. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091* (2019).

[50] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. 2019. fairseq: A Fast, Extensible Toolkit for Sequence Modeling. In *Proceedings of NAACL-HLT 2019: Demonstrations*.

[51] Jay H Park, Gyeongchan Yun, Chang M Yi, Nguyen T Nguyen, Seungmin Lee, Jaesik Choi, Sam H Noh, and Young-ri Choi. 2020. HetPipe: Enabling Large DNN Training on (Whimpy) Heterogeneous GPU Clusters through Integration of Pipelined Model Parallelism and Data Parallelism. *arXiv preprint arXiv:2005.14038* (2020).

[52] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*. 693–701.

[53] Haidong Rong, Yangzihao Wang, Feihu Zhou, Junjie Zhai, Haiyang Wu, Rui Lan, Fan Li, Han Zhang, Yuekui Yang, Zhenyu Guo, et al. 2020. Distributed Equivalent Substitution Training for Large-Scale Recommender Systems. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*. 911–920.

[54] Christopher J Shallue, Jaehoon Lee, Joe 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).

[55] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538* (2017).

[56] J. Shun, F. Roosta-Khorasani, K. Fountoulakis, and M. W. Mahoney. 2016. Parallel Local Graph Clustering. *Proc. of the VLDB Endowment* 9, 12 (2016), 1041–1052.

[57] Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V Le. 2017. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489* (2017).

[58] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research* 15, 1 (2014), 1929–1958.

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

[60] Haobo Sun, Yingxia Shao, Jiawei Jiang, Bin Cui, Kai Lei, Yu Xu, and Jiang Wang. 2019. Sparse gradient compression for distributed SGD. In *International Conference on Database Systems for Advanced Applications*. Springer, 139–155.

[61] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[62] Thijs Vogels, Sai Praneeth Karimireddy, and Martin Jaggi. 2019. PowerSGD: Practical low-rank gradient compression for distributed optimization. In *Advances in Neural Information Processing Systems*. 14259–14268.

[63] Hongyi Wang, Scott Sievert, Shengchao Liu, Zachary Charles, Dimitris Papailiopoulos, and Stephen Wright. 2018. Atomo: Communication-efficient learning via atomic sparsification. In *Advances in Neural Information Processing Systems*. 9850–9861.

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

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

[66] David P. Woodruff. 2014. Sketching As a Tool for Numerical Linear Algebra. *Found. Trends Theor. Comput. Sci.* 10 (2014), 1–157.

[67] Hang Xu, Chen-Yu Ho, Ahmed M Abdelmoniem, Aritra Dutta, El Houcine Bergou, Konstantinos Karatsenidis, Marco Canini, and Panos Kalnis. 2020. *Compressed Communication for Distributed Deep Learning: Survey and Quantitative Evaluation*. Technical Report.

[68] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Ré, Christopher R Aberger, and Christopher De Sa. 2019. PipeMare: Asynchronous Pipeline Parallel DNN Training. *arXiv preprint arXiv:1910.05124* (2019).

[69] Zhewei Yao, Amir Gholami, Qi Lei, Kurt Keutzer, and Michael W Mahoney. 2018. Hessian-based analysis of large batch training and robustness to adversaries. *arXiv preprint arXiv:1802.08241* (2018).

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

[71] Mingchao Yu, Zhifeng Lin, Krishna Narra, Songze Li, Youjie Li, Nam Sung Kim, Alexander Schwing, Murali Annavaram, and Salman Avestimehr. 2018. Gradiveq: Vector quantization for bandwidth-efficient gradient aggregation in distributed CNN training. In *Advances in Neural Information Processing Systems*. 5123–5133.

A SCHEMES THAT DO NOT WORK

We saw in Sec. 3 that activations during the forward pass and gradients during the backward pass can be compressed by large factors (up to 20×) using DCT-MP. This is due to selecting and training only the most relevant neurons corresponding to a given training sample. In this section, we present some negative results with other methods to compress the activation and gradient matrices during the forward and backward passes, respectively.

Gaussian Sketching for Activation Compression. Here, we use a Gaussian sketching scheme to compress the activations going forward. In Randomized Numerical Linear Algebra (RandNLA), the idea of sketching is to represent a large matrix by a smaller proxy that can be further used for matrix operations such as matrix multiplication, least squares regression, and low-rank approximation [66]. The sketched version of a matrix A is given by $A \times S$, where S is a random sketching matrix (e.g., all entries of S are sampled i.i.d. from an appropriately scaled Gaussian distribution).

In Table 6, we compress the activations during the forward pass using Gaussian sketching. Unlike the DCT-MP algorithm, we do not compress the gradients during the backward pass. The aim is to identify if a low-rank structure exists in the activation matrix that can be used to compress the activation matrix in general.

Table 6: Compressing the activation matrix during MP using Gaussian sketching does not yield good results.

Compression Factor	Train Loss	Train Acc (%)	Test Loss	Test Acc (%)
Baseline	0.4477	79.23	0.4538	78.78
50%	0.4569	78.72	0.4618	78.37
75%	0.4610	78.53	0.4656	78.12
90%	0.4685	77.95	0.4721	77.78

As seen in Table 6, sketching techniques directly borrowed from RandNLA do not perform as well. This is likely because such schemes were designed to cater to operations such as low-rank approximation, where the matrices to be compressed are generally well-approximated by low-rank matrices. For instance, Gaussian sketching has seen success in approximate least squares regression and low-rank matrix approximation [66]. This suggests that the activation matrix for DNNs, in general, does not reside in a subspace that is sufficiently low-rank to be meaningfully used for compression.

Top-K Thresholding for Gradient Compression. We saw in Sec. 3 that the parameter gradients (illustrated as W_{grad} in Fig. 1) can be compressed to high factors with any loss in accuracy when used with appropriate error compensation. However, the same is not true for the gradients with respect to hidden neurons (illustrated as X_{grad} in Fig. 1) that are sent across the network during the backward pass in MP. This can be seen from our results in Table 7, where we apply gradient compression using top-K thresholding with error feedback. Further, we observed that training without error feedback can cause divergence.

Our hypothesis on why compressing the gradients of the hidden neurons by top-K thresholding does not yield good results is due to the propagation of error to the initial layers. Consider the following example to illustrate this. Consider the following deep network, where we have several vector-valued functions $A(\cdot), B(\cdot), C(\cdot), \cdots, L(\cdot)$ composed in a chain, that is $A \to B \to C \to \cdots \to K \to L$. Algebraically, the loss looks like $L(A) = L(K(\cdots C(B(A)) \cdots))$. Then, the gradient of the loss with respect to A is given by the multiplication of the Jacobians, that is, $J_L(A) = J_L(K) \times \cdots \times J_C(B) \times J_B(A)$. (Here, $J_L(A)$ denotes the gradient of L with respect to A.) If we change any of the Jacobians in between (that is, compress the gradient X_{grad} with respect to hidden neurons), then the error is propagated all the way to the initial layers of the network. Even adding error feedback to the compression process does not recover the lost accuracy.

Table 7: Compressing the gradient matrix during backward pass in MP using top-K sparsification does not yield good results.

Compression Factor	Train Loss	Train Acc (%)	Test Loss	Test Acc (%)
Baseline	0.4477	79.23	0.4538	78.78
50%	0.4495	79.07	0.4561	78.62
75%	0.4516	78.95	0.4588	78.48
90%	0.4701	77.76	0.4789	77.13

B ANALYSIS OF DCT-MP (ALGORITHM 2)

Unlike compression of gradients, where the error introduced can be effectively corrected, compressing activations introduces errors that are propagated to the downstream of the network. If the thresholds $\{\tau_i\}_{i=1}^B$ are fixed for all training iterations, then Algorithm 2 is simply performing SGD of a changed network, one in which the additional layer $\tilde{X}_{act,i} = \mathbb{I}(|X_{act,i}| \geq \tau_i)$ inserted right after $\{X_{act,i}\}_{i=1}^B$ are obtained. Extra analysis is needed when $\{\tau_i\}$ are dynamic. Consider a particular SGD iteration k, and further annotate the thresholds as $\{\tau_i^{(k)}\}$ to indicate their dependence on the stochastic mini-batch being chosen at iteration k. Let $\bar{\tau}^{(k)} = \mathbb{E}_i(\tau_i^{(k)})$ be the average of these thresholds over the entire dataset. Denote by $\bar{L}$ the loss function of the network with this batch-independent threshold $\tilde{X}_{act,i} = \mathbb{I}(|X_{act,i}| \geq \bar{\tau}^{(k)})$ inserted, and let L_k be the loss function of this dynamically thresholded network with batch k. Standard SGD assumptions say that for a randomly chosen batch k, $\mathbb{E}_i(\partial L_i/\partial \boldsymbol{\theta}) = \partial L/\partial \boldsymbol{\theta}$, with $\boldsymbol{\theta}$ being the network parameters. However, since each L_i is related to a different threshold, it is unclear what the L should be on the right hand side of this expression. In the following theorem, we show that $\mathbb{E}_i(\partial L_i/\partial \boldsymbol{\theta}) = \partial \bar{L}/\partial \boldsymbol{\theta}$, where L_i is batch-i loss function of the *dynamically* thresholded network in Algorithm 2.

Theorem 1. *Consider a 2-worker MP network where the activations from Worker 1 to Worker 2 are thresholded as in Algorithm 2. Let L_i be the associated loss function for data point i, where $i = 1, 2, \ldots, N$, and N be the total number of training samples. Let $\bar{\tau} = \mathbb{E}_i(\tau_i)$, the entire data set at a particular training iteration j (the subscript j in τ_i and $\bar{\tau}$ is omitted for simplicity). Let $\bar{L}_i$ be the loss function corresponding to the threshold $\bar{\tau}$. If*

$$\mathbb{E}_i[X_{act,i} \odot \mathbb{I}(|X_{act,i}| \geq \tau_i)] = \mathbb{E}_i[X_{act,i} \odot \mathbb{I}(|X_{act,i}| \geq \bar{\tau})], \quad (1)$$

then, up to first order

$$\mathbb{E}_i\left(\frac{\partial L_i}{\partial \boldsymbol{\theta}}\right) = \mathbb{E}_i\left(\frac{\partial \bar{L}_i}{\partial \boldsymbol{\theta}}\right) = \frac{\partial \bar{L}}{\partial \boldsymbol{\theta}}$$

where $\boldsymbol{\theta}$ is the parameter of the network.

Remark 3. *The theorem statement implies that the training step on the dynamically thresholded network is equivalent to that of training the network with just one threshold $\bar{\tau}$. This happens long as $\bar{\tau}$ and τ_i are sufficiently close and the mean of activations around τ_i is the same as their mean around $\bar{\tau}$ (formalized by assumption (1)).*

Proof. To establish the proof, let the dynamically thresholded network be represented as follows, starting with a random data point $(X_i^{(0)}, y_i)$ (where $X_i^{(0)}$ is the input and y_i the corresponding label):

$$\begin{aligned}
X_i^{(k)} &= \mathcal{N}^{(k)}(\boldsymbol{\theta}^{(k)}, X_i^{(k-1)}), \quad k = 1, 2, \ldots, m \\
X_i^{(m+1)} &= X_i^{(m)} \cdot \mathbb{I}(|X_i^{(m)}| \geq \tau_i) \\
X_i^{(k)} &= \mathcal{N}^{(k)}(\boldsymbol{\theta}^{(k)}, X_i^{(k-1)}), \quad k = m+2, m+3, \ldots, K \\
L_i &= \ell(X_i^{(K)}, y_i).
\end{aligned}$$

Here, the MP split was inserted after the k-th layer, and the activation function for the k-th layer with parameters $\boldsymbol{\theta}^{\boldsymbol{k}}$ is represented as $\mathcal{N}^{(k)}(\boldsymbol{\theta}^{\boldsymbol{k}}, \cdot)$. For an activation layer $\mathcal{N}^{(k)}$, $\boldsymbol{\theta}^{(k)}$ is simply an empty set. For the network with a static threshold $\bar{\tau}$:

$$\begin{aligned}
\bar{X}_i^{(m+1)} &= X_i^{(m)} \cdot \mathbb{I}(|X_i^{(m)}| \geq \tau) \\
\bar{X}_i^{(k)} &= \mathcal{N}^{(k)}(\boldsymbol{\theta}^{(k)}, \bar{X}_i^{(k-1)}), \quad k = m+2, m+3, \ldots, K \\
\bar{L}_i &= \ell(\bar{X}_i^{(K)}, y_i).
\end{aligned}$$

By assumption (1), we have $\mathbb{E}_i(X_i^{(m+1)}) = \mathbb{E}_i(\bar{X}_i^{(m+1)})$. Therefore, by Taylor's expansion, we have

$$\mathbb{E}_i(X_i^{(k)}) = \mathbb{E}_i(\bar{X}_i^{(k)}), \quad k = m+2, m+3, \ldots, K,$$

up to first order, by expanding each $\mathcal{N}^{(k)}(\boldsymbol{\theta}^{(k)}, \bar{X}_i)$ to first order. For example, for a linear layer, we have

$$\begin{aligned}
\mathbb{E}_i(X_i^{(k)}) &= \mathbb{E}_i(\mathcal{N}(\boldsymbol{\theta}^{(k)}, X_i^{(k)})) \\
&= \mathbb{E}_i(W^{(k)} X_i^{(k)} + b^{(k)}) \\
&= W^{(k)} \mathbb{E}_i(X_i^{(k)}) + b^{(k)} \\
&= \bar{X}_i^{k+1}.
\end{aligned}$$

Similarly, for a general activation function $\sigma(\cdot)$, we have

$$\begin{aligned}
&\mathbb{E}_i(\sigma(X_i^{(k)})) = \mathbb{E}_i(\sigma(\bar{X}_i^{(k)} + (X_i^{(k)} - \bar{X}_i^{(k)}))) \\
&= \mathbb{E}_i(\sigma(\bar{X}_i^{(k)}) + \sigma'(\bar{X}_i^{(k)}) \cdot (X_i^{(k)} - \bar{X}_i^{(k)})) + \text{second order terms} \\
&= \mathbb{E}_i(\sigma(\bar{X}_i^{(k)})) + \sigma'(\bar{X}_i^{(k)}) \mathbb{E}_i(X_i^{(k)} - \bar{X}_i^{(k)}) \\
&= \mathbb{E}_i(\sigma(\bar{X}_i^{(k)})) = \mathbb{E}_i(\bar{X}_i^{(k+1)}),
\end{aligned}$$

where we have ignored the second-order terms beyond the second step. Let $\Delta_i^{(k)} = X_i^{(k)} - \bar{X}_i^{(k)}$, $E_i(\Delta_i^{(k)}) = \mathbf{0}$. Consider the gradient $\frac{\partial L_i}{\partial \boldsymbol{\theta}}$ as a function of $\Delta_i^{(k)}$:

$$\frac{\partial L_i}{\partial \boldsymbol{\theta}}(\Delta_i^{m+1}, \Delta_i^{(m+2)}, \ldots, \Delta_i^{(K)}) = \frac{\partial \bar{L}_i}{\partial \boldsymbol{\theta}} + \sum_{k=m+1}^{K} \left(\frac{\partial^2 \bar{L}_i}{\partial \Delta^{(m)} \partial \boldsymbol{\theta}}\right) \cdot \Delta_i^{(k)}.$$

Thus

$$\mathbb{E}_i\left(\frac{\partial L_i}{\partial \boldsymbol{\theta}}\right) = \mathbb{E}_i\left(\frac{\partial \bar{L}_i}{\partial \boldsymbol{\theta}}\right),$$

which proves the desired result. □