Communication-Efficient Distributed Training of Deep Neural Networks: An Algorithms and Systems Perspective

by

Ilia Markov

September, 2024

A thesis submitted to the
Graduate School
of the
Institute of Science and Technology Austria
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Committee in charge:

Matthew Robinson, Chair
Dan Alistarh
Lefteris Kokoris Kogias
Matrin Jaggi
Tim Harris

The thesis of Ilia Markov, titled *Communication-Efficient Distributed Training of Deep Neural Networks: An Algorithms and Systems Perspective*, is approved by:

Supervisor: Dan Alistarh, ISTA, Klosterneuburg, Austria

Signature: _____

Committee Member: Lefteris Kokoris Kogias, ISTA, Klosterneuburg, Austria

Signature: _____

Committee Member: Matrin Jaggi, EPFL, Lausanne, Switzerland

Signature: _____

Committee Member: Tim Harris, Microsoft Research, Cambridge, United Kingdom

Signature: _____

Defense Chair: Matthew Robinson, ISTA, Klosterneuburg, Austria

Signature: _____

Signed page is on file

ISTA Thesis, ISSN: 2663-337X

I hereby declare that this thesis is my own work and that it does not contain other people's work without this being so stated; this thesis does not contain my previous work without this being stated, and the bibliography contains all the literature that I used in writing the dissertation.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee, and that this thesis has not been submitted for a higher degree to any other university or institution.

I certify that any republication of materials presented in this thesis has been approved by the relevant publishers and co-authors.

Signature: _____

Ilia Markov
September, 2024

Signed page is on file

Abstract

Deep learning is essential in numerous applications nowadays, with many recent advancements made possible by training very large models. Despite their broad applicability, training neural networks is often time-intensive, and it is usually impractical to manage large models and datasets on a single machine. To address these issues, distributed deep learning training has become increasingly important. However, distributed training requires synchronization among nodes, and the mini-batch stochastic gradient descent algorithm places a significant load on network connections. A possible solution to tackle the synchronization bottleneck is to reduce a message size by lossy compression.

In this thesis, we investigate systems and algorithmic approaches to communication compression during training. From the systems perspective, we demonstrate that a common approach of expensive hardware overprovisioning can be replaced through a thorough system design. We introduce a framework that introduces efficient software support for compressed communication in machine learning applications, applicable to both multi-GPU single-node training and larger-scale multi-node training. Our framework integrates with popular ML frameworks, providing up to 3× speedups for multi-GPU nodes based on commodity hardware and order-of-magnitude improvements in the multi-node setting, with negligible impact on accuracy.

Also, we consider an application of our framework to different communication schemes, such as Fully Sharded Data Parallel. We provide strong convergence guarantees for the compression in such a setup. Empirical validation shows that our method preserves model accuracy for GPT-family models with up to 1.3 billion parameters, while completely removing the communication bottlenecks of non-compressed alternatives, providing up to 2.2× speedups end-to-end.

From the algorithmic side, we propose a general framework that dynamically adjusts the degree of compression across a model's layers during training. This approach enhances overall compression and results in significant speedups without compromising accuracy. Our algorithm utilizes an adaptive algorithm that automatically selects the optimal compression parameters for model layers, ensuring the best compression ratio while adhering to an error constraint. Our method is effective across all existing families of compression methods. It achieves up to 2.5× faster training and up to a 5× improvement in compression compared to efficient implementations of current approaches. Additionally, LGreCo can complement existing adaptive algorithms.

Acknowledgements

I would like to thank my supervisor, Dan for leading me in my research, while also allowing me to to explore my research interests. I am grateful for his support and for believing in me throughout this journey. I also thank Lefteris, Martin and Tim for being part of my thesis committee, and I thank Matthew for being my defense chair.

I also wish to convey my appreciation to all the friends I made at ISTA and in Vienna. The memories and adventures we shared greatly enriched my PhD experience. Notable thanks to Ivan, Misha, Ted, Jeto, Jenni, Katja and Yuri for many enjoyable moments over the past five years. Additionally, the final year of my PhD would have been far less vibrant and engaging without the presence of Yaroslav and Anastasia. I would like to offer my deepest appreciation to a special person in my life, Ksenia, whose constant support and compassionate understanding have been invaluable.

Most importantly, I am extremely grateful to my parents, Marina and Valery. Their firm belief in me and their endless sacrifices have provided me with the foundation and strength to pursue my dreams. I am forever indebted to them for their love, knowledge, and patience. Thank you for always being there, for every milestone, and for every challenge I have faced!

Granting sources. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 805223 ScaleML). Furthermore, the candidate acknowledges the support from the Scientific Service Units (SSU) of ISTA through resources provided by Scientific Computing (SciComp).

About the Author

Ilia Markov completed his undergraduate degree in Computer Security at the Moscow Institute of Physics and Technology. He joined ISTA in September 2019, where he was supervised by Dan Alistarh. His main research interests are related to efficiency in deep learning. During his PhD, he has explored topics related to systems in machine learning. His work, which was published in several international conferences, focused on developing practical systems targeting the efficiency of scalable training and application of deep learning models.

List of Collaborators and Publications

This thesis is based on the following first-author papers of Ilia Markov. We provide a summary of the contributions of each author, referred to by their initials.

1. Project CGX: Algorithmic and System Support for Scalable Deep Learning on a Budget. Ilia Markov, Hamidreza Ramezanikebrya, Dan Alistarh. ACM/IFIP International Middleware Conference, 2022

    - Chapter 3 is based on this work.
    - IM worked on the implementation of the main algorithm and CUDA kernel.
    - HR worked on integration of quantization kernel into NCCL library.
    - IM performed the main experiments validating the accuracy recovery and speedup of the suggested system for various models and problems.
    - Throughout the project, DA supervised IM and HR, coordinated regular progress meetings and suggested related work.
    - All authors contributed to the writing of the final manuscript.

2. L-GreCo: An Efficient and General Framework for Layerwise-Adaptive Gradient Compression. Ilia Markov, Kaveh Alim, Elias Frantar, Dan Alistarh. Conference on Machine Learning and Systems (MlSys), 2024

    - Chapter 4 is based on this work.
    - IM developed the simulation framework to test the accuracy recovery and integrated the idea into communication framework
    - KA integrated the idea into PowerSGD implementation and conducted the experiments.
    - DA supervised IM and KA throughout the project, suggesting new experiments.
    - EF helped with debugging and integration of Dynamic Programming algorithm into the framework.
    - All authors contributed to the writing of the final manuscript.

3. Quantized Distributed Training of Large Models with Convergence Guarantees. Ilia Markov, Adrian Vladu, Qi Guo, Dan Alistarh. International Conference on Machine Learning, 2023

    - Chapter 5 is based on this work.
    - IM integrated of the weight compression under FSDP workload into the CGX framework.

- AV developed the theory and analysis of the problem covered in the paper.
- QG conducted a part of the experiments.
- DA initially proposed the idea to IM, suggested the codebase to start the work from, supervised IM, QG and VA throughout the entire project.

The candidate has also co-authored in the following publications which are not included in the thesis:

- QUIK: Towards End-to-End 4-Bit Inference on Generative Large Language Models. Saleh Ashkboos, Ilia Markov, Elias Frantar, Tingxuan Zhong, Xincheng Wang, Jie Ren, Torsten Hoefler, Dan Alistarh. Under submission, ACL Rolling review.
- Elastic Consistency: A Practical Consistency Model for Distributed Stochastic Gradient Descent. Giorgi Nadiradze, Ilia Markov, Bapi Chatterjee, Vyacheslav Kungurtsev, Dan Alistarh. Association for the Advancement of Artificial Intelligence Conference, 2021.
- NUQSGD: Provably Communication-efficient Data-parallel SGD via Nonuniform Quantization. Ali Ramezani-Kebrya, Fartash Faghri, Ilia Markov, Vitalii Aksenov, Dan Alistarh, Daniel Roy. Journal of Machine Learning Research, 2021.
- Adaptive Gradient Quantization for Data-Parallel SGD. Fartash Faghri, Iman Tabrizian, Ilia Markov, Dan Alistarh, Daniel M. Roy, Ali Ramezani-Kebrya. Conference on Neural Information Processing Systems (NeurIPS), 2020

Table of Contents

List of Figures

List of Tables

List of Algorithms

CHAPTER 1

Introduction

Deep Neural Networks have made remarkable advancements in the last few years. The models find widespread applications across various domains, including computer vision [HZRS16, DBK+20], natural language processing [VSP+17, DCLT18], as well as applications in natural sciences [JEP+21]. Neural networks can also assist in image generation [RPG+21], information search or analysis [BMR+20, LC17]. Yet, highily-adaptive models require a large scale of models and datasets. In turn, training large models on massive data can be time- and energy-consuming.

In this thesis, we focus on reducing the training time for the distributed training. Fast training enhances the productivity of machine learning researchers and practitioners. Also, it reduces the time between the data collection and deployment of the final model. It is often impractical to handle large models and datasets on a single machine. The common strategy to tackle these problems is to distribute the compute on multiple resources. The primary approach in Deep Learning, which was also used in all aforementioned successes, is Data Parallelism [DCM+12]. Multiple devices process different partitions of the data in parallel and compute model updates.

Scaling the number of devices allows to increase the batch size which allows to speed up the training. However, this improvement comes with two problems. The first is related to the optimization algorithm, Stochastic Gradient Descent [RM51] (SGD), used in most of Deep Neural Network training approaches. Parallelism helps SGD reduce the variance of model updates by computing on larger portions (batches) of training data. However, it was found [GDG+17] that training with large batch sizes results in worse accuracy on data the model has never seen during the training. The second problem is that distributed training introduces synchronization requirements among compute devices. The aforementioned SGD represents a communication-intensive pattern. It contains computation-intensive phases and communication-intensive synchronization of model updates interleaved with one another. Communication is used to exchange the model parameters or the gradients between the devices. This significantly strains the network both in terms of bandwidth and latency. As hardware accelerators speed development continues to outpace network bandwidth and models keep growing in size, the computation-to-communication ratio increasingly shifts towards communication, i.e. advances in computation term are more prominent than communication improvements. For example, BERT [DCLT18] model published in 2018 contains 340 million parameters, whereas Nemotron-4 [WDD+24] released in 2024 consists of 340 billion parameters showing the parameters count increase of 1000 times in 6 years. From the hardware perspective, the interconnect bandwidth provided by major cloud providers between cloud instances has only

improved by 10 times over the past decade. As a result, network performance can substantially impact the training time for large Deep Neural Networks (DNNs).

In this thesis, we focus on system and algorithmic approaches that tackle communication bottlenecks in Data Parallel training. One option is to use a specific hardware [SCH+21] to circumvent the bandwidth problems. This requires additional efforts to deploy and maintain the hardware, which can be excessive for most practitioners. Another solution is to reduce communication by either reducing the message size or reducing the number of sent messages. A number of messages can be reduced by taking local steps [Sti18], breaking the global synchronization into peer-to-peer synchronization [LZZL18], or by the aforementioned approach with batch size increase.

At a high level, this thesis is concerned with methods for lossy compression. In this approach attention has to be paid to the following things: 1. Applying lossy compression affects the final model accuracy, so the compression parameters have to be carefully chosen. 2. The compression costs should be significantly lower than the communication gains so that applying compression makes sense in practice. We address these challenges and propose systems and algorithmic solutions to improve communication efficiency at Distributed Data Parallel training.

This thesis investigates the bottlenecks in distributed training of Deep Neural Networks and presents a method for creating a system that uses efficient lossy compression in this context. Additionally, we have implemented a framework that supports various communication compression techniques, enabling us to match the performance of state-of-the-art systems without affecting the final model's accuracy.

The remainder of the thesis is organized in the following way:

- In Chapter 2, we present a necessary background on Neural networks and distributed training concepts, and overview common approaches to tackle communication problems.
- In Chapter 3, we introduce a system supporting efficient gradient compression. We show that the costly hardware overprovisioning approach can be replaced by algorithmic and system design, and propose a framework called CGX, which provides efficient software support for compressed communication in ML applications, for both multi-GPU single-node training, as well as larger-scale multi-node training. CGX is based on two technical advances: At the system level, it relies on a re-developed communication stack for ML frameworks, which provides flexible, highly-efficient support for compressed communication. At the application level, it provides seamless, parameter-free integration with popular frameworks, so that end-users do not have to modify training recipes or significant training code. This chapter is based on our published work [MRA22], appearing in Middleware'22.
- In Chapter 4, we investigate the idea of adapting compression parameters across model layers and training time. This chapter, based on our work [MAFA24], published in MlSys'22, presents a general framework for dynamically adapting the degree of compression across the model's layers during training, improving overall compression while leading to substantial speedups, without sacrificing accuracy. The framework, called L-GreCo, is based on an adaptive algorithm that automatically picks the optimal compression parameters for model layers, guaranteeing the best compression ratio, while satisfying an error constraint. Extensive experiments over image classification and language modeling tasks show that L-GreCo is effective across all existing families

of compression methods and achieves up to 2.5× training speedup and up to 5× compression improvement over efficient implementations of existing approaches, and can even complement existing adaptive algorithms.

- In Chapter 5, which is based on our ICML'23 paper [MVGA23], we consider a variation of Data Parallel training, namely fully-sharded data parallel (FSDP) training. We present QSDP, a variant of FSDP supporting both gradient and weight quantization. QSDP is simple to implement and has essentially no overheads. We validate this approach by training GPT-family models with up to 1.3 billion parameters on a multi-node cluster. Experiments show that QSDP preserves model accuracy, while completely removing the communication bottlenecks of FSDP, providing end-to-end speedups of up to 2.2x.
- Finally, in Chapter 6, we review our methods and results and explore potential new research directions for the future.

CHAPTER 2

Background

In this chapter, we present the key concepts, algorithms, and systems for distributed deep learning which we will refer to in the subsequent chapters. In the first part, we cover the main concepts related to neural networks, training process, and optimization. Next, we discuss distributed training. In the second part, we describe various approaches to scale the training, and communication patterns in Data Parallel training and highlight the communication bottlenecks in consumer-grade GPUs and compression methods that are commonly used to tackle them.

2.1 Basics of Neural Network Training

Deep Learning is a subfield of Machine Learning and Artificial Intelligence. This section explores the fundamental components and processes involved in Deep Learning training as a comprehension of these mechanisms is essential for enhancing training efficiency.

2.1.1 Neural Networks

Neural Network represents a set of multiple layers of basic processing units called artificial neurons. They take inputs, process them, and produce an output. Each neuron uses an aggregation function to compute a single combined value from its weighted inputs. Then each neuron applies an activation function to its input to produce the output. The connections between neurons are called weights, and the model training involves adjusting these weights to minimize the difference between the model predictions and true outcomes.

2.1.2 Training process

The training of the model is iterative and performed in steps. Each step comprises of two phases: forward and backward passes.

Forward pass. During the forward pass the data is passed through the network, layer by layer, from input to output. Each neuron processes the input using weights and an activation function and then passes the result to the next layer. The final layer's output is compared to the expected result and an error or loss value is computed.

Loss function. The loss value represents a measure of how far the network's output is from the actual target value. The choice of the loss function depends on the type of task the model is designed to solve. Common loss functions include Mean Squared Error (MSE) for regression tasks and Cross-Entropy Loss [Goo52] for classification tasks.

Backward pass. The backward pass or backpropagation is the process of computing the appropriate update for each layer in the network. It involves calculating the gradient of the loss function with respect to each weight using the chain rule, and then updating the weights in the opposite direction of the gradient so that it minimizes the model loss in subsequent iterations.

2.1.3 Optimization algorithms

One of the common gradient-based optimization algorithms is **Gradient Descent**. It updates weights by moving in the direction of the negative gradient. While Gradient Descent has good convergence rates, it also comes with quite a high computational cost as it requires computing the full gradient. Given the size of the dataset and the sizes of the modern neural networks, the computation of the full gradient at each iteration becomes infeasible.

A possible solution to circumvent the costs of the Gradient Descent approach is to use an estimation of the full gradient. **Stochastic Gradient Descent** [RM51], variation of gradient descent, suggests using only a subset (mini-batch) of the data to calculate the gradient. Although Stochastic Gradient Descent converges more slowly in terms of iterations compared to Gradient Descent, its lower cost per iteration makes it suitable for large datasets and models.

While Stochastic Gradient Descent and its variations are among the most popular algorithms for optimizing neural networks, several other algorithms have been introduced, such as Adagrad [DHS11] and Adam [KB14], which calculate individual learning rates for each parameter in the model.

2.1.4 Major hyper-parameters

The essential part of the training process is the hyper-parameters that affect both the efficiency of the training and final model accuracy. *Learning Rate*: The rate at which the model is updated. Too high learning rate value can cause instability, and too low can lead to slow convergence. *Mini-Batch Size*: Number of samples processed before updating the model. The large batch size allows to improve the training speedup but it decreases the generalization qualities of the final model.

2.2 Distributed Deep Learning Training

Training deep learning models at the modern scale of model and dataset sizes is highly computationally intensive and time-consuming. A possible way to tackle these problems is to distribute the training onto multiple compute devices. There are several approaches to distributed the compute of the training process.

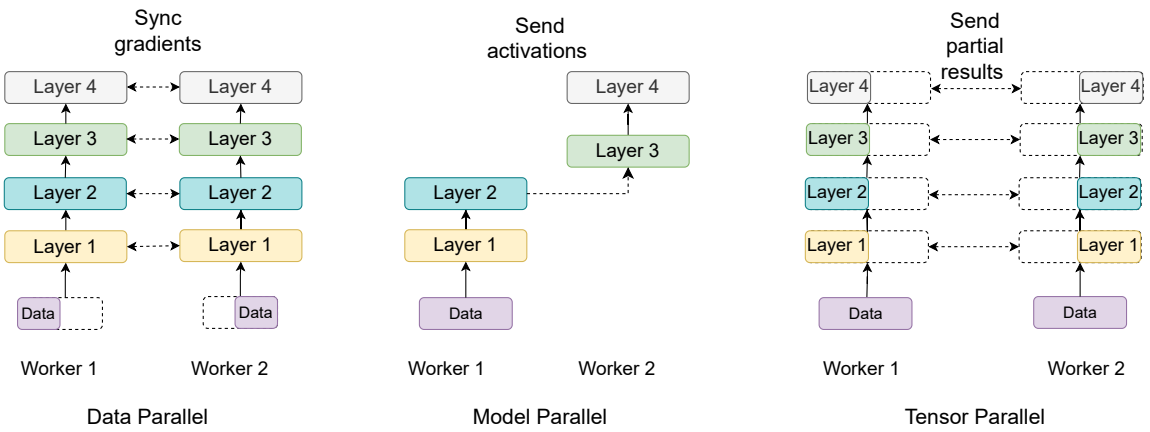


Figure 2.1: Types of distributed training. Solid arrow lines represent communication between layers within a device and dashed arrow lines show the communication between devices.

2.2.1 Communication

Data Parallel

Data parallel distributed training is a popular method for training deep learning models on large datasets. It involves splitting the data across multiple devices (workers), such as GPUs or CPUs, while maintaining a copy of the entire model on each device. Each device is assigned a data partition so that each device processes a unique portion of the data. At each step, all workers perform forward-backward iterations on local data resulting in the local gradients. The locally computed gradients are aggregated and used to update the model parameters. Data parallel is characterized by repeated gradient communication among devices which synchronize the model updates. The communication volume increases with both the model size and the number of workers, leading to significant network traffic.

A variation of the Data Parallel approach is Fully-Sharded Data Parallel (FSDP) technique, introduced in [RRRH20b]. This method addresses the challenges of training large models by efficiently utilizing memory. FSDP incorporates data parallelism by allowing each shard to process a different subset of the input data but it avoids keeping the entire copy of the model on each device. In FSDP, model parameters are divided (sharded) across multiple devices. Each device holds only a part of the full model's parameters. During forward-backward the weights of a layer that is about to execute an operation are collected on each device and vanilla Data Parallel operation is performed.

Model Parallelism

In Model Parallelism, the model is split into multiple parts that are distributed across the devices. Each device computes the input for its assigned model part and propagates it to the subsequent device. The common approach for model parallelism optimization is pipelining. The method suggests overlapping computations between one layer and the next as data becomes ready. Pipelining is used to overlap forward evaluation, backpropagation, and weight updates. This approach is commonly implemented [CKF11, SPP+19, AAB+15] and boosts utilization by reducing periods of processor inactivity. The traffic pattern in such distributed training type generally shows a sequential nature as data moves through each stage of the pipeline. The amount of data exchanged between stages is proportional to the output sizes of the layers at

the ends of the model partitions. It is typically smaller than the entire model because users are able to choose how to partition the model to reduce communication between stages.

Tensor Parallelism

The third partitioning strategy for DNN training is tensor parallelism. This strategy divides the work inside a layer and distributes the operations of a neural network across multiple devices, such as GPUs. This approach is especially useful for training very large models that do not fit into the memory of a single device. The main idea is to split the tensors - the data and model parameters - involved in the computations across multiple devices then perform operations locally and synchronize the results. Note, that unlike in the Data Parallel approach, in Tensor Parallel the input can be split not only in batch dimension and we synchronize the outputs of each layer not model updates. The typical example of Tensor Parallel is a distribution of matrix-matrix multiplication. We split the input matrices into smaller submatrices and distribute them across multiple devices or processors. Each processor then performs the matrix multiplication using its assigned partitions. To get the final result matrix, we must combine the partial results from each processor communicating and aggregating the local results.

In practice, large-scale deep learning typically employs a hybrid approach that integrates all or some of the described distributed training types. This combination harnesses the advantages of the methods to efficiently train large neural networks on extensive datasets. Figure 2.1 illustrates the major distributed training techniques implemented in the modern Deep Learning frameworks.

This thesis focuses on the Data Parallel approaches - the ones that are mostly affected by communication overhead in practice.

Communication primitives in Data Parallel

The original Data Parallel technique features storing a copy of a model on each device and aggregation, typically summation, of model updates at each training iteration. Basically, we need to compute the sum of local vectors and store the result on each device. Modern implementations of this procedure include Parameter Server [LNC+18, LAP+14] approach and various types of AllReduce collective primitive.

Parameter server Devices in this approach are split into two groups: workers that perform the training computation and Parameter servers that are responsible for the model updates aggregation (one device may serve two roles). In this method, the workers compute the model updates and send them to parameter servers. These servers aggregate the updates to calculate and distribute the new model parameters. This approach can accommodate a varying number of workers and parameter servers, making it adaptable to different hardware configurations and workloads. Sharding the model parameters across several Parameter Servers helps balance the load and prevent bottlenecks, improving the overall efficiency of the training process. This approach allows to avoid explicit synchronization between workers, however, it does not completely exclude the global barrier - each worker still has to wait till all other workers compute the update and send it to the Parameter Server.

Next, we describe various implementations of AllReduce primitive. The primitive gets an input vector and a list of workers and at the end of the execution, each worker has an aggregated input vectors from all workers.

Full mesh approach is a straight-forward one-round algorithm in which every node sends a vector to other participants and sums the received vectors. Basically, it represents AllGather primitive with summing the resulting matrix along the 0-th dimension. The communication complexity of this algorithm per node is dN^2 (where d - size of the input, N - number of workers), the latency term for Full Mesh is α (the communication with other servers can be done in parallel).

Ring-Allreduce is arguably the most bandwidth-optimal algorithm. It is implemented in all popular communication libraries (NCCL, Gloo) used in distributed Deep Learning frameworks. Similar to SRA, Ring uses the division of the initial vector into chunks. But communication is done in a ring-shaped topology. The algorithm consists of 2 phases, each phase has $N-1$ rounds. In the first phase, each node p sends chunk p to the neighbor on the right in the ring topology and receives a chunk from its left neighbor. Then it sums the received chunk and its chunk $p-1$ and sends the result to the right neighbor receiving chunk $p-2$. The receive-sum-send repeats $N-1$ times and each node has a different portion of the resulting vector. In the second phase, the nodes want to broadcast the resulting chunks, i.e. perform AllGather collective, and, according to the algorithm, they do it within the ring-shaped topology. They follow the same communication pattern as in the first round but instead of summing local chunk chunks with the received ones each node overwrites local values. The communication complexity is $O(dN)$, the latency term is higher than in the previous algorithms and amounts to $2\alpha(N-1)$, as long as communication can not be parallelized due to the algorithm design.

Rabenseifner's AllReduce [Rab04] consists of two rounds. First, each process divides its vector into N subarrays, which we refer to as "chunks". In the first round, each node p receives the p-th chunk of the initial vector (we refer to it as chunk p) from all nodes and aggregates it. For example, the first node accumulates the elements of the input vector from 0 to $\lceil \frac{d}{N} \rceil$, second node broadcasts the range from $\lceil \frac{d}{N} \rceil$ to $2\lceil \frac{d}{N} \rceil$, etc. In other words, each node performs a Scatter-Reduce collective. In the second round, each node broadcasts the aggregated chunk to all other nodes, i.e. AllGather collective. The communication complexity of the algorithm is $O(dN)$, and the latency term is 2α.

Tree-AllReduce essentially represents a hierarchical parameter-server. It uses a dendritic data aggregation pattern where each node collects data from its child nodes and then forwards the aggregated data to its parent node. This traffic pattern follows the tree structure, typically a binary or n-ary tree. The depth of the tree determines the latency. The communication is done in $2log(N)$ rounds and two phases. The communication complexity is $O(dN)$, latency term is $2\alpha \log N$.

Hierarchical Approach is specifically designed for multi-tier architectures and initially performs AllReduce within nodes, such as multi-GPU systems, typically utilizing higher-speed links. It then aggregates data between nodes during the inter-node AllReduce. They may combine several aforementioned algorithms, in the sense that it uses different algorithms for inter- and intra-node communication. Hierarchical combinations can face performance challenges when there is a significant difference between intra-node and inter-node communication speeds or when the hierarchy becomes excessively deep.

Fully Sharded Data Parallel

Fully Sharded Data Parallel (FSDP) is an advanced distributed training technique designed to efficiently scale the training of large deep learning models across multiple GPUs. The key idea (see Figure 2.2) is that both the training data and the model parameters are split among

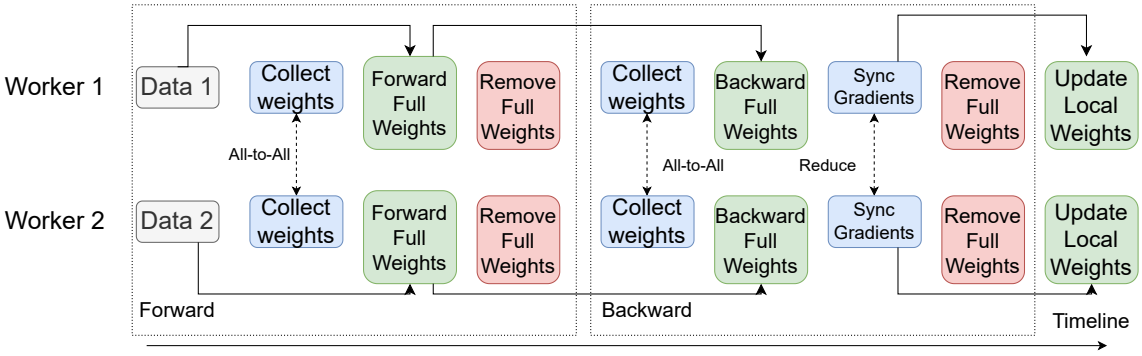


Figure 2.2: Scheme of Fully Sharded Data Parallel technique. Each worker keeps only a split of the weights. Before the forward pass of a layer, all workers collect the missing weights of the layer. Then the original Data Parallel forward pass with full weights is executed which is followed by the removal of the received weights. At the backward pass, the weights are gathered again and local gradients are computed and synchronized. Afterwards, the collected weights are removed and the remaining weight split is updated with the synchronized gradients.

the N nodes. That is, only a $1/N$ partition of the parameters of each layer is stored at a node. Then, during both the forward and backward passes, nodes operate synchronously layer-by-layer. They use all-to-all communication to gather the full weights for the current layer before performing the forward or backward operation. Once the operation is complete, nodes can discard the received weight partitions for the current layer and proceed to the next layer. The major advantage of the method is memory usage reduction. However, FSDP faces challenges in terms of communication: since every forward and backward pass includes all-to-all weight exchanges, the network bandwidth may suffer from massive communication.

2.2.2 Communication bottleneck

Since the communication of the gradients in the Data Parallel approach involves large amounts of data, comparable to the size of the DNN, the communication becomes the bottleneck [MRA22, PZC+19]. Specifically, Luo et al. [LNC+18] state that computation speed and neural network sizes grow faster than network bandwidth. Hence, modern compute devices (typically, GPUs) encounter long idle times while waiting for communication. This leads to longer training time and inefficient utilization of computational resources.

The shift of the performance bottleneck from computation to communication is due to two factors. First, advancements in GPUs and other compute accelerators, like the NVIDIA H100 [NVI22], provide significant performance enhancements — 20× for floating point calculations and 40× for mixed precision calculations—compared to V100, released in 2017. Moreover, introducing new supported data types, e.g. fp8, increases the inference speed by another 2× for mixed precision computations. This rate of improvement outpaces the advancements in network bandwidth, which took 8 years to achieve a 10× increase in Ethernet speeds, going from 10 Gbps to 100 Gbps.

Secondly, the training workload is undergoing a shift in the communication-to-computation ratio, which is intensified by the trend toward larger neural networks. However, the impact of this shift varies depending on the specific application. Popular deep learning frameworks support partial overlaps of communication and computation phases, allowing communication to begin as soon as the first partial results of backpropagation are available. The effectiveness

of this technique depends on the structure of the model, with limited benefits for DNNs with large initial layers, as there is less opportunity to overlap communication with computation in these cases.

Model size, params count	10 Gbits	50 Gbits	100 Gbits	Ideal
125M	2.6	2.0	1.9	1.1
350M	14.0	6.8	6.0	3.2
1B	51.0	26.2	23.5	10.8

Table 2.1: Average step time (in seconds, lower is better) in the training of Large Language models using Fully Sharded Data Parallel at different inter-node communication bandwidths. "Ideal" stands for the training without communication.

To quantitatively validate the bottleneck, we performed a benchmark of 3 language models trained using Fully Sharded Data Parallel in a cluster of 4 nodes, each includes 8 NVIDIA V100 GPUs. In order to verify the training performance for different communication bandwidths (we chose 10, 50, and 100 Gbits) we artificially reduced input-output bandwidth on each node, using the UNIX `tc` tool [TC01]. We can see (Table. 2.1) that training of a large model is highly affected by communication bandwidth, the communication in this case contains 400% of the computation ("Ideal" column in the table). The typical size of the modern language models could be by an order of magnitude larger which implies that communication bottleneck in that case will become a significant challenge.

2.2.3 Communication compression

To address the problem of communication bottleneck, lossy gradient compression was proposed. The DNN training usually applies versions of the Stochastic Gradient Descent (SGD) [RM51]. Given the training stochasticity, intuitively, the process should converge despite the noise introduced by the lossy compression. The existing compression methods can be divided into biased and unbiased.

Unbiased methods. A compression operator $Q(\mathbf{v})$ over a vector $\mathbf{v}$ is called *unbiased* if: $\mathbb{E}[Q(\mathbf{v})] = \mathbf{v}$. Examples of unbiased operators include quantization schemes, such as QSGD [AGL+17], TernGrad [WXY+17], and some versions of sparcification, e.g. rand-K [SCJ18]. Quantization reduces the precision of each element in the gradient vector, for example, casts each 32-bits float to one byte. Sparsification methods pick a subset of the gradient components, resulting in a sparse vector. Intuitively, sparsification might profit from bucketing as well, but in practice, it works better when compressing gradients per layer or even the full network-level gradient vector.

QSGD is a codebook-based quantization scheme. It maps vector components into predefined values. This method quantizes each component of the gradient via randomized rounding to a uniformly distributed set of values. The randomized rounding allows preserving the expected value of the stochastic gradient. Formally, the method can be defined as follows. For any $\mathbf{v} \in \mathbb{R}^d$, with $\mathbf{v} \neq \mathbf{0}$, $Q_s(v_i) = \|\mathbf{v}\|_2 \cdot sign(v_i) \cdot q(v_i, s)$. $q(v_i, s)$ is a random variable that is defined the following way. Let $0 \leq l \leq s-1$ be an integer such that $|v_i|/\|\mathbf{v}\| \in [l/s, (l+1)/s]$, s is a size of the codebook. Then,

$$q(v_i, s) = \begin{cases} l/s, \text{with probability } 1 - p(|v_i|/\|\mathbf{v}\|, s) \\ (l+1)/s, \text{otherwise} \end{cases}$$

Algorithm 2.1: Training with error feedback

input : Local copy of model $\mathbf{x}$, Number of nodes N, $e_0 = \mathbf{0}$
output : The trained model $\hat{x}$

On each node i **for** *each iteration* t **do**
  Calculate stochastic gradient $g_t^i(\mathbf{x})$;
  $p_t^i = g_t^i(\mathbf{x}) + e_t^i$; `// Apply error feedback`
  $c_t^i = Q(p_t^i)$; `// Compress the vector`
  $e_{t+1}^i = p_t^i - c_t^i$; `// Update error feedback`
  Perform AllReduce operation decompressing vectors when needed;
  Result of AllReduce: $p_t = \frac{1}{N}\sum_{j=1}^{N} p_t^j$;
  Apply p_t to $\mathbf{x}$;
end

where $p(a, s) = as - l$ for any $a \in [0, 1]$.

The major advantages of QSGD are efficient implementation of the algorithm, absence of training hyperparameters tuning in practice, and lack of additional tools to compensate the loss. The disadvantages include constraints on compression ratio (the experiments show that the accuracy degrades in general when the compression ratio exceeds ~ 8 and, in theory, one cannot quantize the element to less than 1 bit), non-associativity of the compression operator with summation (i.e. the order of compression and summation operations matters). Another downside of the QSGD is that when it is applied to the entire gradient vector it leads to a convergence degradation, due to scaling issues. The common way to address this is to split the vector into subarrays, called buckets, and apply a compression technique independently to each bucket. This approach increases the compressed size of the vector because we have to keep meta-information for each bucket and slows downs the compression, but helps to recover the full accuracy.

TernGrad encodes gradient components into three values {-1, 0, 1} scaled by the infinite norm of the gradient. First, given a stochastic gradient g, the method encodes values to bits, the bit values are selected with probability: $P(b_i = 1|g[i]) = |g_i| / \|g\|_\infty$. Formally, TernGrad is defined as: $Q(g_i) = \|g\|_\infty \cdot sign(g_i) \cdot b_i$. The main disadvantage of the algorithm is the necessity of tuning hyperparameters such as learning rate schedule and weight decay. When the standard parameters are used, the algorithm has a substantial accuracy loss.

Rand-K is a fixed-dimension sparsification method. A set of k values is randomly selected out of d possible ones and passed along with the corresponding indices, all other values are ignored. By design, Rand-k is biased but can be made unbiased by multiplying g with a factor d/k. In comparison to the previous methods, Rand-K has higher limits of the compression ratio. However, training with applied Rand-K does not converge or has a high accuracy loss in practice with comparable to the QSGD compression ratio using the standard hyperparameters.

Biased methods. Basically, a compression operator $Q(\mathbf{v})$ is called *biased* if $\mathbb{E}[Q(\mathbf{v})] \neq \mathbf{v}$. In general, such compression operators need compensation techniques that aggregate the error introduced by the compression. An example of such a technique is error feedback (see Algorithm.2.1). Applying such compensation methods lets us recover accuracy and improve convergence rate for both unbiased and biased methods [KRSJ19]. A possible drawback of the error feedback is a memory overhead - the size of error feedback buffers is comparable to the model's size. Examples of biased methods include quantizations, sparsifications, and low-rank methods.

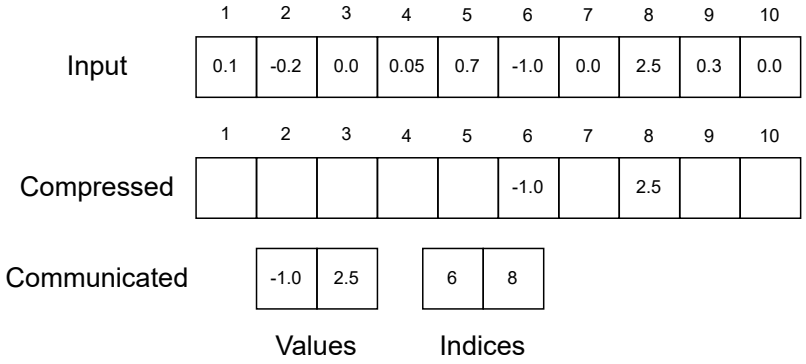


Figure 2.3: Example Top-k compression: 20% of the gradient components and corresponding indices are sent.

One-bit SGD [SFD+14] represents a quantization method that encodes all elements that are less than the constant threshold τ to 0, and all others to 1. In decoding, zeros transform to the mean of negative values, and ones to the mean of positive values. The approach can be easily efficiently implemented, but it requires memory compensation mechanisms, and hyperparameters tuning and the compression operator is not associative.

Top-K [RAA+19, Sti18] is a fixed-dimension sparsification method. It selects the elements from the vector that belong to the k largest values of the vector (in absolute value) and saves their indices and values (see Figure.2.3). This method has unlimited perspectives of the compression ratio value. Although similar to QSGD, it loses accuracy with standard parameters starting from a certain compression ratio threshold and, as a biased compression it requires additional error compensation techniques. Another downside of the method is an expensive estimation of the largest values which significantly slows down the compression.

A different approach to gradient compression is to treat the gradients as multidimensional tensors, not as vectors. The idea of the low-rank factorization methods is to decompose the gradient matrix $G \in \mathbb{R}^{m \times n}$ into 2 rank-r matrices $P \in \mathbb{R}^{m \times r}$ and $Q \in \mathbb{R}^{r \times n}$. In order to reduce communication complexity r is chosen much smaller than m and n.

There are several methods favoring this approach. ATOMO [WSL+18] uses singular value decomposition (SVD) to find P and Q matrices. But in the case of large models SVD of gradient matrices becomes too compute-intensive to use it in DNN training. PowerSGD [VKJ19] uses a generalized power iteration algorithm to calculate P and Q matrices. Among the other low-rank factorization methods PowerSGD is the fastest one, to the best of our knowledge. The advantages of this technique are that it has high compression rates(up to $100 - 1000\times$) and that the compression operator is associative. One of the disadvantages is high latency overhead because it compresses and communicates layers separately and does two communication rounds per layer. Also, in order to recover the accuracy the method introduces several memory compensation buffers which increases the memory footprint.

CHAPTER 3

Systems support for efficient gradient compression

3.1 Preface

As previously mentioned in Chapter 2, distributed training in Data Parallel setup suffers from communication bandwidth limitations. Lossy gradient compression is a widely used approach to address this issue. Although numerous methods and techniques have been proposed, only a few are practically applicable. Furthermore, integrating these methods into real-world applications demands substantial engineering effort, and the resulting performance improvements may fall short of expectations.

In this Chapter, which follows [MRA22] with minor changes, we discuss the communication problems of distributed training and suggest an algorithmic and systems design approach as an alternative to hardware overprovisioning approach. We present a gradient compression framework, called CGX (for **C**ompressed **G**radient E**x**change), that provides significant speedups for multi-GPU nodes based on commodity hardware, and order-of-magnitude improvements in the multi-node setting, with negligible impact on accuracy.

3.2 Introduction

Distributed scalability of Deep Learning still presents non-trivial challenges, and the last decade has seen a tremendous amount of work on distributed paradigms, algorithms, and implementations to address them [LAP+14, CSAK14, ABC+16, PZC+19, JZL+20].

Specifically, two key scaling challenges behind are reducing the *synchronization costs* among computing nodes [JWG+19, JZL+20, PZC+19, LMXG21], and minimizing the *communication costs* which arise naturally due to the high bandwidth requirements of all-to-all transmission of model updates (gradients) between nodes. In this chapter, we focus mainly on mitigating the *bandwidth cost* of gradient transmission in DNN training, which is an increasingly common bottleneck, correlated to the soaring parameter counts of modern machine learning models.

There are two main strategies for addressing bandwidth bottlenecks. The *industrial approach* involves bandwidth over-provisioning, exemplified by the 30-fold increase in inter-GPU bandwidth in NVIDIA-enabled cloud-grade multi-GPU servers from the 2015 Kepler generation to

the post-2018 Ampere generation, aided by the customized GPU-centric NCCL communication library [Nvi18]. However, this approach incurs substantial hardware and development costs, leading to significantly higher expenses for end-users. For instance, there is a nearly tenfold cost difference between cloud-grade, over-provisioned multi-GPU servers like NVIDIA DGX systems [GLD+17] and commodity workstations with consumer-grade GPUs, which are popular due to their lower costs and comparable single-GPU performance, but have notable scalability performance gaps.

The alternative algorithmic approach leverages the fact that stochastic gradient descent (SGD) [RM51] can converge with compressed gradients. Methods such as gradient quantization, sparsification, and gradient decomposition can theoretically reduce bandwidth costs by up to two orders of magnitude without compromising accuracy. However, practical implementation of these methods faces significant challenges.

The first challenge is that of **parametrization and integration**: approaches such as gradient sparsification or decomposition often require non-trivial parameter and implementation changes to the training process, e.g. [LHM+17, RAA+19, VKJ19], to support compression. This would require practitioners to revisit their entire training setup and tune additional hyper-parameters, in order to achieve compression while recovering accuracy. A second challenge is that of **efficient system support** for communication-compression, as it often requires significant changes to lower levels of the software stack, such as supporting compressed or sparse data types. Despite research in this direction [XHA+21, BLZ+21, GJY+21], the question of general and efficient system support for communication-compression is still open: currently, only one such approach, PowerSGD decomposition [VKJ19], is supported natively by one popular framework, PyTorch [PGM+19].

Contributions. In this chapter, we introduce a communication framework called CGX, which addresses these challenges, and allows for *parameter-free, seamless integration* of communication-compression into data-parallel DNN training workflows, with up to order-of-magnitude speedups for data-parallel DNN training.

At the application level, CGX starts from an investigation of the feasibility of *parameter-free compression*: specifically, we implement and test all existing algorithmic approaches, and identify a variant of quantization-based compression that converges to *full accuracy* for many popular models, under *fixed, universal settings of parameters*, without modifying to the original training recipes. At the system level, we investigate how gradient compression can be seamlessly and efficiently integrated with modern ML frameworks. Specifically, we revisit the entire communication stack of modern ML frameworks with compression in mind, from a new point-to-point communication mechanism which supports compressed types, to compression-aware reductions, and finally a communication engine which interfaces with ML frameworks, supporting compression at the tensor/layer level.

The existence of a parameter-free compression technique which recovers accuracy, combined with the ability of CGX to customize the compression level per layer motivates a new *layer-wise adaptive compression problem*. The idea is that we can customize the way model gradients are compressed in *layer-wise* fashion, so that the overall compression error is close to a given accurate baseline, but maximizing the bandwidth gains: for instance, one can apply more aggressive compression to layers that are larger, but less "sensitive" in terms of accuracy. While prior work has already considered techniques which globally adapt the degree of compression during training, e.g. [AWL+21, MAEAC21], this is the first instance of this problem to jointly considers both error and compression constraints at the fine-grained *per-layer* level. Our

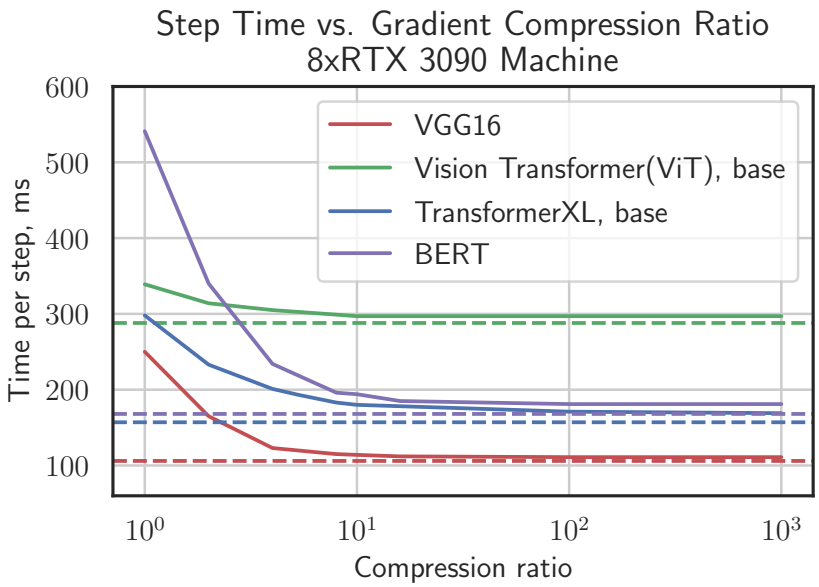


Figure 3.1: Compression vs. average step time for different models, when using all GPUs on an 8x RTX-3090 machine (Table 3.2). Dotted lines denote the throughput at perfect scalability for each model. Throughput nears ideal as we decrease transmission size, suggesting that bandwidth is the main bottleneck. See Section 3.3.1 for details.

experimental results show that layer-wise adaptive compression can bring significant additional gains.

To justify our design choices, we contrast our design against the first implementation of quantized collectives in NCCL, which we call QNCCL, which we contribute as a separate artifact, showing clear performance and usability improvements in favor of the CGX design. In addition, CGX does not require significant user-code or training pipeline changes, as we provide turn-key integrations with popular ML frameworks such as Pytorch(Section 3.5.1) and Tensorflow(Appendix A.2).

Experimental Validation. From the practical perspective, our work is motivated by the experimental data in Figure 3.1, showing that *bandwidth congestion* is the key scalability bottleneck on single-node, multi-GPU commodity servers, which have emerged as a popular training approach [Lam21, Gen21, Lea21]. The same phenomenon occurs generally in multi-node data-parallel training settings, for a wide range of current and emerging training workloads, from image classification using classical convolutional neural networks (CNNs), to Transformer-based models for both language modeling [VSP+17, DYY+19] and image classification [PVU+18, DBK+20].

We validate our system experimentally in both single-node and multi-node settings, across all of the above standard training tasks. We compare servers using commodity NVIDIA GPUs (RTX series) against cloud-grade NVIDIA servers from the Volta and Ampere architectures. (See Table 2 for details.) First, we find that, once communication bottlenecks are eliminated from “commodity" machines using CGX, they can match or *outperform* cloud-grade server with similar peak performance. Importantly, this can be done with negligible accuracy loss.

For example, we find that, on a commodity 8x RTX 3090 server, CGX can almost *triple* training throughput, reaching up to 90% of the ideal scaling, matching or even outperforming a bandwidth-overprovisioned (and more expensive) DGX-1 system. Our second application

Table 3.1: Server-grade (first 2) vs. consumer-grade NVIDIA GPUs. Throughput obtained using the NVIDIA Deep Learning Examples benchmark [Nvi20]. Throughput for Resnet50 is measured in images/s, for Transformer-XL (T-XL in table) in tokens/s. TDP stands for maximal Thermal design power (in Watts).

GPU type	Arch.	SM	TensorCores	GPU Direct	GPU RAM, GB	TDP	ResNet50	T-XL
A100	Ampere	108	432	Yes	40	250	2470	60K
V100	Volta	80	640	Yes	16	250	1226	37K
A6000	Ampere	84	336	Yes	48	300	566	39K
RTX3090	Ampere	82	328	No	24	350	850	39K
RTX2080 TI	Turing	68	544	No	10	250	484	13K

Table 3.2: Systems characteristics of workstations used in evaluation.

System	GPUs	Inter-GPU link	Inter-GPU bandwidth	GPU RAM	RAM	CPUs
DGX-1	8xV100	NVLink	100 Gbps	128 GB	512 GB	64
A6000	8xA6000	NVLink	100 Gbps	384 GB	1008 GB	128
RTX-3090	8xRTX3090	None (bus)	15 Gbps	192 GB	512 GB	128
RTX-2080	8xRTX2080 TI	None (bus)	15 Gbps	96 GB	256 GB	72

is to *multi-node training*, where we show up to 10x performance gains, enabled in part by our new solution to the adaptive layer-wise compression problem, without accuracy loss or additional parameters.

Our findings imply that hardware bandwidth overprovisioning may not be required for scalability in DNN training, and that highly-customized, hyperparameter-heavy compression techniques are not always necessary to remove bottlenecks. This should be immediately useful to users aiming to scale such workloads on commodity or multi-node hardware, but also more broadly for hardware/software co-design for distributed deep learning.

3.3 Motivation and Prior Work

3.3.1 A Motivating Experiment

The standard computational unit for DNN training is the multi-GPU node, usually in instances with 4–16 GPUs. End-users often rely on consumer-grade GPUs for training, whereas traditionally cloud services mainly employ cloud-grade GPUs, with some notable exceptions, e.g. [Lam21, Lea21, Gen21]. We begin by briefly examining the scalability differences between cloud and commodity GPU servers. As we illustrate in Figures 3.5b and 3.5c, the maximum effective throughput of a cloud-grade 8-GPU DGX-1 server is $> 2\times$ higher than that of a comparable commodity 8xRTX-3090 GPU server, when using the same state-of-the-art software configuration (specifically, Horovod [SB18] on top of the NCCL communication library).

This gap is surprising, considering that the single-GPU performance is similar (see Table 3.1). To examine the specific impact of *gradient transmission / bandwidth cost*, we implemented a synthetic benchmark that reduces bandwidth cost by artificially compressing transmission. Specifically, assuming a buffer of size N to be transmitted, e.g. a layer's gradient, and a target compression ratio $\gamma \geq 1$, we only transmit the first $k = N/\gamma$ elements. The results for the 8x RTX-3090 machine, using all 8 GPUs, are shown in Figure 3.1, where the compression ratio

is varied on the X axis, and we examine its impact on the time to complete an optimization step, shown on the Y axis. The dotted line represents the time per step in the case of ideal (linear) scaling of single-GPU times. We consider Transformer [DYY+19] and BERT-based models [DCLT18] for language modeling tasks, as well as VGG-16 [SZ14] and Vision Transformer (ViT) models for classification on ImageNet.

We therefore observe that *bandwidth cost appears to be the main scalability bottleneck on this machine*. Moreover, recent models (Transformer-XL and ViT) benefit more from compression relative to the classic ResNet50 model, which has fewer parameters. Second, *there are limits to how much compression is required for scalability*, which depend on the model characteristics. An order of magnitude compression appears to be sufficient for significant timing improvements, although Transformer-based architectures can still benefit from compression of up to two orders of magnitude.

Discussion. The reason for this poor scalability is the lack of efficient communication support. Specifically, GPU-to-GPU transmissions on commodity hardware have significantly lower bandwidth, and higher latency, relative to their cloud counterparts. Specifically, in software , the NVIDIA GPUDirect technology should allow GPUs on the same machine to communicate directly, without the need for extra memory copies. Commodity GPUs, such as the RTX 3090, do not support this technology. At the same time, the hardware communication support for NVIDIA GPUs, i.e. NVLink and NVSwitch components, is also not available or severely restricted for commodity GPUs [Har21, nvi21].

3.3.2 Data-Parallel DNN Training

Distribution Strategies and Costs. Training a DNN essentially minimizes a loss function, related to the error of the model on the dataset, via a sequence of optimization steps, each acting on some data samples. To preserve computational efficiency, it is common to perform a *batched* version of this process, by which several samples are processed in a single optimization step, and the sum of gradients is applied.

Data-parallelism is arguably the standard way to scale DNN training, and can be viewed as a variant of batch SGD in which sample gradients are generated in parallel over compute nodes. Specifically, the dataset is partitioned over nodes, each of which maintains a copy of the model, and computes gradients over samples in parallel. Periodically, these gradients are aggregated (e.g., averaged) and the resulting update is applied to all local models.

Several techniques have been proposed to address the synchronization and communication costs inherent to this lock-step averaging procedure. Here, we focus on *communication/bandwidth cost*, and assume that synchronization preserves the synchronous ordering of gradient iterations, although our techniques are also compatible with all existing scheduling strategies, e.g. [JWG+19, PZC+19, JZL+20, YLM+20].

Batch Scaling. An orthogonal scaling approach is increasing the batch size at each node. This requires careful hyper-parameter tuning for accuracy preservation, e.g. [GDG+17, YLR+19], although recipes for large batch scaling are known for many popular models. We consider scalability in both 1) *the large-batch setting*, where we adopt the best-known hyperparameter recipes to preserve accuracy, and 2) *the small-batch setting*, corresponding to datasets or models for which large-batch scaling parameters are unavailable or unknown.

3.3.3 Communication Compression Methods

The basic idea behind communication-compression methods is to reduce the bandwidth overhead of the gradient exchange at each step by performing lossy compression. Our presentation assumes that a generic mechanism allowing for all-to-all communication among the nodes is available. (We discuss our implementation choices in Sections 3.4 and 3.5.) Roughly, existing schemes can be classified as follows.

Gradient Quantization. Roughly, this approach works by reducing the bit-width of the transmitted updates [SFD+14]. One of the first compression approaches [AGL+17] observed that *stochastic* quantization of the gradient values is sufficient to guarantee convergence. Their method, called QSGD, is a codebook compression method which quantizes each component of the gradient via randomized rounding to a uniformly distributed grid. Formally, for any non-zero vector $\vec{v}$, given a codebook size s and $\vec{v} \in \mathbb{R}^d, Q_s(v_i) = \|\vec{v}\|_2 \cdot sign(v_i) \cdot q(v_i, s)$. The stochastic quantization function $q(v_i, s)$ essentially maps the component's value v_i to an integer quantization level, as follows. Let $0 \leq \ell \leq s-1$ be an integer such that $|v_i|/\|\vec{v}\| \in [\ell/s, (\ell+1)/s]$. That is, ℓ is the lower endpoint of the quantization interval corresponding to the normalized value of v_i. Then,

$$q(v_i, s) = \begin{cases} \ell/s, \text{with probability } 1 - p(|v_i|/\|\vec{v}\|, s), \\ (\ell+1)/s, \text{otherwise} \end{cases}$$

where $p(a, s) = as - \ell$ for any $a \in [0, 1]$. The trade-off is between the higher compression due to using a lower codebook size s, and the increased variance of the gradient estimator, which in turn affects convergence speed. This idea inspired a range of related work [LAK18, RKFM+21, FTM+20] reducing the variance of the compression by improved quantizers. We discuss these schemes further in Section 3.5.

Gradient Sparsification. These methods, e.g. [Str15, DJMVE16, LHM+17, KRSJ19], capitalize on the intuition that many gradient values may be skipped from transmission. The standard approach to sparsification is *magnitude thesholding*, effectively selecting the top K gradient components for transmission, where K is a hyper-parameter. Then, error correction is applied to feed the thresholded gradient components back into the next round's gradient. Variants of this procedure can achieve more than $100\times$ gradient compression while still recovering accuracy [LHM+17]. However, this comes at the price of model-specific hyper-parameter tuning, which may be unreasonable in a deployment setting.

Renggli et al. [RAA+19] proposed efficient sparse collectives, and observed that sparsification methods can be promising in cases where there is high natural redundancy–such as fully-connected or embedding layers–but may be a poor choice for general compression due to the need for hyper-parametrization. Our investigation confirmed their finding.

Gradient Decomposition. This approach treats the gradients as multidimensional tensors, and decomposes the gradient matrix $G \in \mathbb{R}^{m \times n}$ into 2 rank-r matrices $P \in \mathbb{R}^{m \times r}$ and $Q \in \mathbb{R}^{r \times n}$, with r much smaller than m and n. ATOMO [WSL+18] uses singular value decomposition (SVD) to find the matrices P and Q. However, in the case of large models, the SVD of gradient matrices becomes too compute-intensive to be used during training. PowerSGD [VKJ19] uses a generalized power iteration algorithm to calculate the matrices P and Q, and is the fastest currently-known factorization method. To recover accuracy, it applies a combination of error correction techniques. Their results show that these methods can be highly useful in the case of CNNs, yielding high compression ratios (up to $100\times$). However, in

Table 3.3: Compression approaches. Stateful here means that approach requires maintaining of a state of error compensating techniques.

	Compression rate with recovery	Tunable Parameters	Properties	Computational Overhead
Quantization	$\sim$ 8x	Bits, bucket size	Non-associative, stateless	$\leq$ 3%
Sparsification (TopK)	$\sim$ 100x	Sparsity, momentum	Non-associative, stateful, not overlapping with compute	10%
Decomposition (PowerSGD)	$\sim$ 100x	Rank, warm-up	Associative, stateful, incompatible with mixed precision	20%

our experience, recovering accuracy in e.g. Transformers training requires careful tuning, and higher rank values, resulting in lower performance.

Adapting Compression during Training. The idea of adapting the degree of compression during different stages of DNN training has been considered by [AWL+21, GLW+20, CCB+18, CYRW20, MAEAC21]. However, we emphasize the fact that all these references *globally adapt* the amount of gradient compression for the entire model to preserve end accuracy, whereas we investigate mechanisms which adapt compression at the per-layer level. Moreover, to achieve high compression, some existing methods require hyperparameter tuning [CYRW20] or focus on specific architectures [AWL+21]. By contrast, we adapt compression parameters automatically both across layers, and across training iterations.

Efficient Software Support. There has already been significant work on providing system support for compression. Two main challenges are: 1) the introduction of additional hyper-parameters in the training process, and 2) the fact that, since most compression methods are *not associative*, they are not directly supported by standard collective implementations and require algorithm-specific re-implementations. Grubic et al. [GTAZ18] showed that CNNs can withstand 8-bit gradient compression, and provided a simple MPI-based implementation of quantization, while Dutta et al. [DBA+20] examined the implementation gap, showing that frameworks should support both global and per-layer compression. Renggli et al. [RAA+19] and Fei et al. [FHS+21] provided efficient support for sparse reductions, while the GRACE framework [XHA+21], Bagua [GJY+21] and HiPress [BLZ+21] frameworks provided efficient implementations of communication-compression methods. We compare against these frameworks in Section 3.7.

We differ from this prior work in two major directions. At the application level, we focus on *seamless, parameter-free integration* with existing data-parallel training pipelines: thus, we investigate compression techniques which allow accuracy recovery *without additional hyper-parameter tuning*. This is not the case with prior frameworks, which leave the choice of compression parameters to the user. Second, at the system level, we seek to maximize speedup by rewriting components of the communication stack to support compression, provide an adaptive layer-wise compression solution which maximizes speedup.

Recent work by [AWVP] investigated the practical potential of gradient compression methods in cloud-grade settings. They provide analytical and empirical evidence suggesting that gradient compression methods can only provide marginal speedups in distributed data-parallel training of DNNs in such bandwidth-overprovisioned settings.

However, the generality of their results is restricted by the following factors: 1) they only consider a limited subset of compression methods and possible implementations: for instance,

their compressed implementations strictly follow the NCCL API, which, as we illustrate via our QNCCL implementation, means that the compression methods were used inefficiently and with accuracy loss; 2) they focus on cloud-grade bandwidth-overprovisioned systems, and therefore their findings do not apply to the popular setting of commodity servers. These two factors, as well as additional implementation differences, explain the difference between their conclusions and the ones from this work.

3.4 Goals and Challenges

The results in Section 3.3.1 suggest that bandwidth can be a key bottleneck when attempting to scale DNN training on commodity GPUs, while the discussion in Section 3.3.3 outlines non-trivial trade-offs when implementing these techniques for general models. We therefore outline our key goals:

1. **Accuracy Recovery:** Similar to MLPerf [MRC+20], we set our accuracy loss threshold at $< 1\%$ relative to the main metric of the full-precision baseline (e.g. Top-1 classification accuracy), although in most of the tasks we present the accuracy loss is practically negligible.

2. **Hyperparameter-Freedom:** Second, we wish to enable scalable data-parallel DNN training in the absence of any model or task information, recovering accuracy under *standard (uncompressed) hyper-parameters*.

3. **Eliminating Bandwidth Bottlenecks:** Third, we aim to mitigate or even completely eliminate bandwidth constraints. Since not all target models are equally communication-bottlenecked, this allows us some flexibility with respect to how much compression to apply depending on the model and application.

4. **Simple Interface:** Finally, the integration with the underlying training framework should be seamless.

State of the art. We executed implementations of the compression methods described in Section 3.3.3 on a range of modern tasks and models. Our findings are summarized in Table 3.3, and discussed in detail below.

We found that no existing approach fully satisfies all the above requirements. For instance, *quantization-based methods* are known recover accuracy on CNNs when using 8-bit compression [GTAZ18], meeting Goals 1 and 2. However, this amount of compression is not sufficient to remove the bandwidth bottlenecks for modern Transformer-class models (Goal 3); moreover, the parameters of [GTAZ18] do not allow full accuracy recovery on Transformers.

Second, examining *gradient sparsification* methods, we notice that they can ensure high compression (Goal 3); however, they require complex hyperparameter tuning for accuracy recovery in the high-compression regime [LHM+17], breaking either Goal 1 or Goal 2. Conversely, as also noted by [RAA+19], these methods can recover accuracy under medium density (e.g. 20%), but in that case their performance is similar to quantization approaches. This family of methods has the additional cost of having to maintain state (the error buffer) and being less amenable to computation-communication overlap, since the selection operation is applied over the entire gradient.

Finally, *decomposition* methods have been shown to yield compression ratios of up to $100\times$ in the case of CNNs, attaining Goal 3. Moreover, with careful tuning of hyper-parameters, PowerSGD is able to recover accuracy for CNNs under generic rank-decomposition values. In addition, this method is *associative*, lending itself to seamless implementation via MPI or NCCL (Goal 4). Unfortunately, however, we found that this method can require high rank values for stable training, especially on Transformers, where there is almost no speedup, and that it is not compatible with reduced-precision (FP16) training, which is used by virtually all frameworks.

3.5 CGX System Design

A typical DNN training framework has three parts, as described in Figure 3.2:

1. **Framework interface** (in Python) with high-level API. It may also include a frontend that unifies the input from the learning framework.

2. **Background thread** collecting inputs, groups them into blocks based on query type and input properties.

3. **Communication engine** performing the query (Allreduce, Broadcast, Allgather). At this stage, the framework typically calls an existing communication library, such as NCCL, Gloo, or an MPI implementation.

A key issue when implementing most compression methods such as quantization or sparsification is that their operations are *non-associative*, and so the aggregation function (sum) must be performed at the lowest level in the above diagram. This means that we cannot integrate the compression into higher levels without a bespoke implementation, which in turn may lead to performance and implementation costs.

3.5.1 The CGX Communication Engine

To efficiently support compression, we implemented our own communication engine, with primitives which support non-associative compression operators. Broadly, there are two approaches to do this. The first is a *native* one, by which one can implement compression-aware Allreduce using communication libraries. Alternatively, one can modify or extend existing communication libraries, such as NCCL, to support compression operators.

The native approach requires deeper integration, but has the advantage that compression is performed “closer" to training, which means that the compression engine has information about the model layers, and their gradients and thus has a richer, more flexible API. The disadvantage is that it has to explicitly interface with the training framework, and users may have to adjust their training pipeline.

The second *low-level* approach is to directly perform compression and de-compression at the primitive/transport level, independently of the user's code and training pipeline. In this case, the framework can only operate with the raw data buffers provided by the upper layers. This loses information about the data it operates with, e.g., layer names, which could be useful for compression operators, but is easier to interface with, and may have lower overheads.

Framework Integration

To investigate this non-trivial dichotomy, we implemented *both* variants. Specifically, our main framework, called CGX, integrates natively with the user's code, and can interface both via Horovod [SB18], a popular distribution wrapper that works with all major ML frameworks, but also separately via framework-specific extensions, such as PyTorch Distributed Data Parallel (DDP). Separately, as an instance of the "low-level" approach, we re-implemented the NCCL communication library to support quantized reduction operations. We call this separate implementation QNCCL, and contrast it to our main approach.

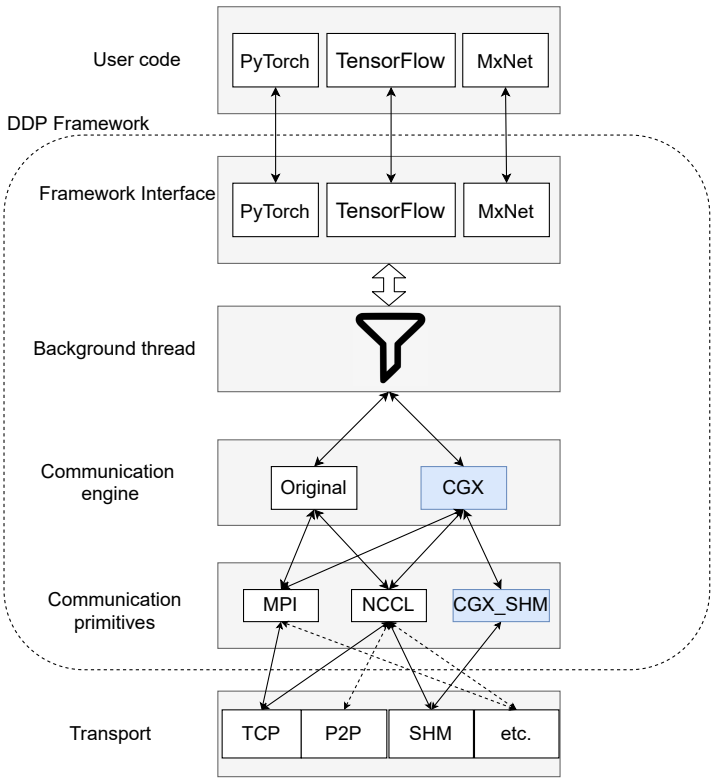


Figure 3.2: Abstract architecture of a Distributed Data Parallel (DDP) framework. CGX components are in blue, and arrows stand for procedure calls. Dashed arrows represent hardware interactions, e.g. P2P transport is supported via GPU NVLinks.

The Native CGX Framework. The main version of CGX uses the Horovod wrapper [SB18] to interface with popular ML frameworks. Specifically, we implemented a communication engine with Allreduce methods supporting compression operators. Next, we added layer filters that split model gradients into logical subsets, which the framework may handle differently: some accuracy-critical subsets are communicated in full precision, while other subsets are compressed and reduced in lower-precision. Empirically, it is known that layers like batch/layer normalization and bias layers are sensitive to gradient compression, while being small. Therefore, we communicate them uncompressed. As a bonus, this avoids calling compression operators for multiple small inputs.

Further, CGX performs compression *per-layer*, and not as a blob of concatenated tensors. This provides the flexibility of exploring heterogeneous compression parameters and avoids mixing gradient values from different layers, which may have different value distributions, leading to large quantization error. We found that such filters can be applied "at line rate" without loss of performance, as most of the computation can be overlapped with the transmission of other layers. CGX's API allows users to choose the compression parameters for specific layers or filter out the group of layers.

Torch DDP Integration. Our compression/communication engine is portable: to illustrate this, we also integrate it separately with the Torch DDP pipeline [PGM+19]. The code is available at **`https://github.com/IST-DASLab/torch_cgx`**. In this case, CGX acts as a Torch extension that implements an additional Torch DDP backend, as a supplement to the built-in NCCL, MPI and Gloo backends. Thus, users only need to import the extension and change the backend at initialization.

We integrated our functionality into the communication engine of the Data Parallel framework. At this level, we no longer have access to the buffer structure, therefore we can not explicitly filter layers. Nevertheless, the user can provide the layout of the model layers (e.g. gradient sizes and shapes). Using this information, we can obtain the offsets of the layers in each buffer provided by `torch.distributed`. An example of using the Torch extension is presented in the Appendix(Listing A.1).

Choosing a Reduction Scheme

The "hottest" operation in distributed data-parallel training is Allreduce, corresponding to the logical gradient averaging. To support non-associative compression operators, we need to choose the reduction algorithm together with the compression operator, to maximize performance and minimize the compression error due to iterative compression-decompression. We considered the following reduction schemes.

Scatter-Reduce-Allgather (SRA) (or Rabenseifner's Allreduce [Rab04]) works in two rounds: a process first divides its vector of dimension d into N subarray ("chunks") each node receives its "chunk" of the initial vector from all other nodes and aggregates it (Scatter-Reduce). Second, it broadcasts the aggregated "chunk" (Allgather). The bandwidth cost is $O(d(N-1))$, the latency term is 2α, corresponding to the two rounds. **Ring-Allreduce** is the bandwidth-optimal algorithm, implemented in most libraries (e.g. NCCL, Gloo). Similar to SRA, it divides the initial vector into chunks, and communication is done in a ring-shaped topology. In the first phase, each node sends a chunk to its "right" neighbor and receives a chunk from its left neighbor. It then sums the received chunk with its local result and sends the result forward, repeating $N-1$ times.

Table 3.4: Throughput of different reduction schemes (items per second).

	ResNet-50	Transformer-XL	ViT
SRA	**2900**	**260k**	**1918**
Ring	2830	236k	1883
Tree	2770	202k	1756

In the second phase, nodes broadcast (Allgather) the resulting chunks on the ring. The bandwidth cost is $O(d(N-1)/N)$, with latency $2\alpha(N-1)$, assuming communication can not be itself parallelized. **Tree-Allreduce** can be seen as a hierarchical parameter-server. Communication is done in $2\log N$ rounds and two phases. The nodes build a tree-like topology, and send their vectors up to the root, summing them along the path, and then propagate back the result. Communication complexity is $O(2d \times \log(N))$, while latency is $2\alpha \log N$.

Discussion. We examined the practicality of these reductions, and found **Scatter-Reduce-Allgather (SRA)** to show the best performance. It also has the key algorithmic advantage of *lower compression error*, due to fewer compression/decompression steps. Thus, we mainly employed this algorithm inside CGX. Table 3.4 illustrates CGX throughput under different reduction schemes, for different tasks, on an 8-GPU server. (See Section 3.7 for the full setup.)

Default Compression Approach

Our framework implements several compression approaches; yet, based on the discussion in Section 3.3.3, we use *gradient quantization* as our main method. The rationale behind our choice is the following. First, as suggested by Figure 3.1, quantization compression by 8-10x should provide sufficient bandwidth reduction to overcome most of the communication bottleneck. Moreover, it can do so *in a generic, parameter-free way*: an independent contribution of our work is that we identify *general parameter values providing 8-10x compression*

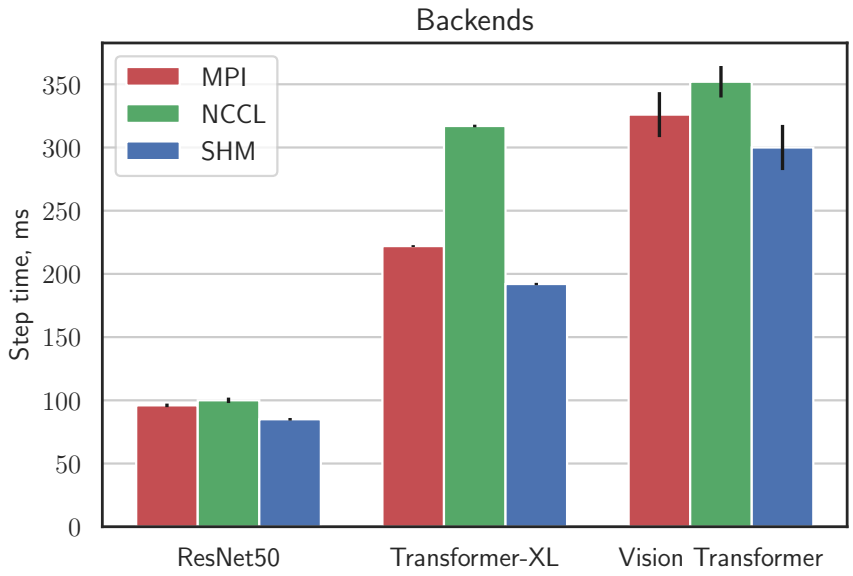


Figure 3.3: Training step times for different communication backends in CGX Communication engine on a single node, 8 RTX3090 GPUs. Lower is better.

without accuracy loss on all the model classes and tasks we tried. We investigate additional performance improvements customized per-layer compression, which can provide an additional performance boost.

3.5.2 Communication Backend

The key question at the lower level of the stack is how to implement the peer-to-peer communication primitives. Here, existing options are GPU-aware MPI implementations, NCCL, or Facebook Gloo. (For instance, GRACE [XHA+21] supports all three options.) To maximize performance, instead of relying on existing implementation, we developed a set of new peer-to-peer communication primitives, that are based on data transfers through UNIX shared memory. We call this communication backend SHM.

SHM works by registering a UNIX shared memory buffer for each pair of GPUs within a node and map it to GPU memory. On send, we move the input buffer to the shared segment and synchronize with the recipient using CUDA IPC primitives. SHM is only supported for a single node, while CGX can use both MPI- and NCCL-based backends in multi-server setups. Moreover, we support heterogeneous communication where the intra node communication uses SHM, MPI, or NCCL as the backend, or performs NCCL-allreduce without compression, while the inter-node communication uses MPI or NCCL. The difference in performances is illustrated in Figure 3.3, showing that SHM outperforms other backends, by up to 33%. The speedup is justified by the lower synchronization between compression and communication, and the single memory transfers via the GPU Communication Engine. Thus, unless otherwise stated, we use SHM for intra-node communication in all our experiments.

3.5.3 Implementation Details

Efficient Quantization. The quantization algorithm sketched in Section 3.3.3 has the following downside: when applied to the entire gradient vector it leads to convergence

degradation, due to scaling issues. A common way to address this is to split the vector into subarrays, called buckets, and apply compression independently to each bucket [AGL+17]. This approach increases the compressed size of the vector because we have to keep scaling meta-information for each bucket and slows down the compression, but helps to recover full accuracy. The bucket size has an impact on both performance and accuracy recovery: larger buckets lead to faster and higher compression, but higher per-element error. Therefore, one has to pick the bucket size appropriate for the chosen bits-width empirically. We found out that 4 bits and 128 bucket size always recovers full accuracy, has reasonable speedup, and can be efficiently implemented, so we use this as a compression baseline in all our experiments.

To achieve low compression overheads, we applied the following optimizations: we use an efficient parallel bucket norm computation algorithm, and, for elementwise compression/decompression, we perform cache-friendly vectorized memory load/stores. Quantization overhead amounts to 1-3% of computational cost in our benchmarks.

Improved Scheduling. CGX also aims to improve the latency term. For this, we perform fine-grained scheduling of gradient synchronization, which is known to lead to improved performance for Parameter Servers [JZL+20]. As part of scheduling optimization, CGX supports user-defined filtering of layers and cross-barrier training. Filtering of small layer modules such as biases or batch norm not only improves convergence but positively affects performance. Such filtering removes the need for extra compression kernel calls without a notable increase in communication costs. Cross-barrier optimization does not provide significant performance in a single node setup, confirming the observations in [JZL+20].

3.5.4 The QNCCL Library

The role of the QNNCL implementation is to contrast our design choices relative to a direct re-implementation of communication compression in the popular NCCL library. To build this low-level variant, we started from vanilla NCCL and replaced Allreduce with implementations that compress every piece of data before its transfer. We leverage the NCCL communication optimizations to avoid costs for additional GPU calls. However, in this case, we lack information about the internal structure of the buffer and have to apply compression parameters uniformly over the entire model. In this case, we also have limitations in terms of the GPU resources imposed by NCCL itself, which lead to additional compression overheads. We examine the performance trade-offs of this approach in the experimental section.

3.6 Layer-wise Adaptive Quantization

One key optimization supported by CGX is *varying compression parameters at the per-layer level*. This is especially well-suited to models such as Transformers which have heterogeneous layer sizes, e.g. due to large embeddings. Synchronization of such layers can be quite expensive, and, since they come early in the model, cannot be overlapped with computation. Yet, these massive layers can support highly-compressed communication. Thus, we investigate *automatic* mechanisms to pick per-layer compression levels.

We focus on the trade-off between two parameters for each layer: the *magnitude of the compression error* and *compressed size of the layer*. Our adaptive algorithm tries to balance these constraints in order to maximize speedup while recovering convergence. We periodically collect gradient statistics and then re-assign bit-widths and bucket-size to each layer. Specifically, we

Table 3.5: Validation results for training with the baseline and CGX optimizations, respectively. ResNet50, VGG and ViT numbers are Top-1% accuracies, Transformer-XL and GPT-2 show perplexity, while BERT shows F1-score.

	ResNet50	VGG16	ViT-base	Transformer-XL-base	GPT-2	BERT
Baseline	75.8 ± 0.2	69.1 ± 0.1	79.2	22.81 ± 0.1	14.1 ± 0.1	93.12 ± 0.05
CGX	75.9 ± 0.2	68.9 ± 0.1	78.6	22.9 ± 0.1	13.9 ± 0.1	93.06 ± 0.05

want to minimize the compressed size of the model gradients while minimizing the ℓ_2-norm of the compression error, which is linked to convergence [KRSJ19].

Problem Definition. We formalize this problem as identifying per-layer bit-widths $b_1, b_2, \ldots, b_L$ for the L layers minimizing the *bandwidth objective* $\sum_{\ell=1}^{L} b_\ell \cdot size(L_\ell)$ across all the b_is, *subject to* the fact that compression error cannot exceed a maximum threshold $\alpha \cdot E_4$. Here, $\alpha > 0$ is a fixed parameter, and E_4 is the error when we compress all layers to 4 bits, for which we know that full recovery occurs.

We emphasize that this formulation is different from the (global) adaptive compression problems considered by prior work [AWL+21, GLW+20, CCB+18, CYRW20, MAEAC21], as they usually consider the problem of adapting the global degree of compression to the various stages of the training process, as opposed to the fine-grained layer-wise bit-width adaptation we consider.

This constrained optimization problem can be approached via standard solvers, and in fact, our first approach has been to use Bayesian optimization. However, we found that this requires an instance-specific tuning, and adds hyper-parameters. We therefore investigate problem-specific heuristics.

A straightforward approach is to simply sort layers by the ratio of gradient magnitude over the layer size. We then assign the lowest bit-width to the first layers in this order, and the highest to the last layers, interpolating linearly in the middle. Experimentally, this approach recovers accuracy and improves over static assignment, but the performance gains are minor.

This observation inspires a *clustering-based* approach, by which we collect layers with similar sensitivity to gradient compression into groups, and assign bit-widths correspondingly. We use a 2D-clustering algorithm [Mac67], where the dimensions are the size of the layer, and the ℓ_2-norm of the top values of the accumulated gradient. We perform clustering to obtain "sensitivity groups," each with its own centroid, and then sort the centroids by their gradient norms. Finally, we linearly map bit-widths and bucket sizes to the layers. The exact procedure is described in Algorithm 3.1. We investigate its practical performance in Section 3.7.3.

Algorithm 3.1: KMEANS-based adaptive compression

Input: Model Layers L_i, accumulated gradients G_i, possible bit-widths $B = \{\beta_1, \beta_2, \ldots, \beta_k\}$
Output: Bit-width assignments $b_\ell \in B$ for each layer ℓ
Initialisation : Compute 2D-representation for each layer ℓ by computing points ($size(L_\ell)$, $norm(G_\ell)$).
1: Obtain (centroids, clusters) = kmeans over data into k clusters
2: Sort centroids based on $norm(C_i) - size(C_i)$ and assign them
3: Assign points (layers) corresponding to each centroid to the corresponding bit width b_ℓ.

3.7 Experimental Validation

3.7.1 Experimental Setting

Infrastructure. Our evaluation uses commodity workstations based on RTX2080 and RTX3090 consumer-grade GPUs, and a cloud-grade EC2 `p3.16xlarge` machine, with 8 V100 GPUs, equivalent to a DGX-1 server. Please see Table 3.2 for complete system characteristics. The scheme of interconnect between GPUs on the 8x RTX3090 machine is shown in Appendix (Figure 3.4). In brief, the 8 GPUs are split into two groups, each assigned to a NUMA node, which are bridged via QPI. Bandwidth measurements via [LSC+18] show that inter-GPU bandwidth varies from 13 to 16 Gbps depending on location. At the same time, we have 1Gbps Allreduce bandwidth for reasonable buffer sizes. Results for RTX2080 are similar, with 1.5Gbps Allreduce bandwidth.

The V100/DGX-1 machine forms a so-called *Backbone Ring* inside a *Hypercube Mesh* [LSC+20], in which GPUs are connected via NVLINK. The DGX-1 has GPU-to-GPU bandwidth of up to 100 Gbps, leading to the same Allreduce bandwidth in our workloads. Performance on our setup is identical to a branded DGX-1 measured via NVIDIA's benchmarks [Nvi20].

Environment and Tasks. Most experiments were run using the PyTorch version of the NVIDIA Training Examples benchmark [Nvi20]. For state-of-the-art model implementations, we used the Pytorch Image Models [Wig19] and the Huggingface Transformers repositories [Hug22]. For the experiments on V100 machine, we used the official NGC PyTorch 20.06-py3 Docker image. We used CUDA 11.1.1, NCCL 2.8.4, and cudnn/8.0.5. We examine three different DNN learning tasks: 1) image classification on ImageNet [DDS+09] ; 2) language modeling on WikiText-103; 3) question-answering on the SQUAD dataset.

Baselines. We use the non-compressed original training recipes as a baseline. We *do not* modify any of the training hyper-parameters. In distributed training, we use either Horovod-NCCL or PyTorch-DDP with NCCL backend. In all our experiments, NCCL showed better performance than OpenMPI or Gloo, so we use it as the default backend. For a fair comparison, we use the CGX extension depending on the baseline framework: for Horovod-NCCL, we use our Horovod extension, and for PyTorch-DDP we apply our Torch distributed backend extension. We also compare our results against ideal linear scalability on the same machine, calculated by training speed on a single device multiplied by the number of devices. We use step time and throughput (items/sec) as the performance metrics. For all performance experiments, we validated that the hyper-parameters used are sufficient to recover training accuracy, across 3 runs with different seeds. All the reported speed numbers are averaged over 300 training iterations after a warm-up of 10 iterations. Unless specifically stated, we *do not* employ the adaptive compression algorithm.

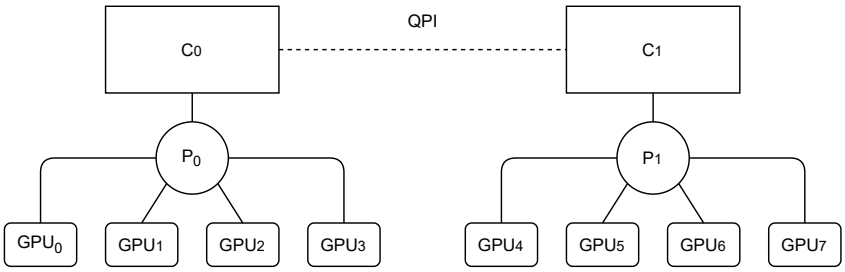
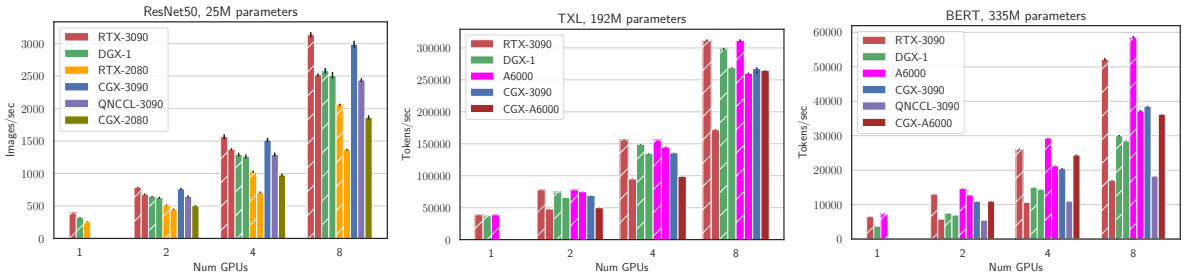


Figure 3.4: PCIe topology for RTX machines.



(a) ResNet50, ImageNet (b) Transformer-XL, WikiText103 (c) BERT, SQUAD

Figure 3.5: Throughput for ResNet50/ImageNet, Transformer-XL (TXL) on WikiText, and BERT on SQUAD. Higher is better. Hatched bars represent ideal scaling. CGX leads to self-speedups of $> 2\times$, and scalability of 80% to 90%. Hatched bars represent ideal scaling.

Table 3.6: Training throughput with CGX, PowerSGD, and GRACE on single machine with 8 RTX3090 GPUs. (Transformer-XL/PowerSGD did not converge, so we only provide throughput numbers.)

	ResNet50	Transformer-XL-base	BERT
Baseline	1900	170k	17.5k
CGX	**2900**	**260k**	**38.7k**
PowerSGD	2600	220k*	38.3k
Grace	1000	30k	14.3k

3.7.2 Experimental results

Accuracy Recovery

We first examine the model accuracies using standard hyper-parameters in end-to-end training experiments. The gradient bit-width used for these experiments is 4 bits. The bucket size was 1024 for CNNs, and 128 for Transformer models, chosen empirically. As stated, we reduce small layers (biases, batch, and layer normalization layers) in full precision. The results of training on the RTX3090 machine with 8 GPUs are presented in the Table 3.5, with the corresponding accuracy parameters. All CGX accuracy results are within the standard 1% error tolerance [MRC+20]; in most cases, accuracy is within seed random variability. We provide detailed training details in terms of batch size and mixed precision in Appendix A.1. Unless otherwise stated, we focus on the following model/task combinations: Transformer-XL on WikiText-103, ResNet50 on ImageNet, and ViT on ImageNet. The parameters are identical to the ones provided above. All experiments were run on 8 GPUs.

Comparison with other algorithmic approaches

PowerSGD Compression. We follow the implementation of [VKJ19], and set the rank to 4 for CNNs and use rank 8 for Transformers, implying up to 100x compression. PowerSGD can not be used in conjunction with FP16 training, as it can lead to divergence in our experiments, so we compare at FP32. But with full-precision gradients training PowerSGD can not achieve baseline accuracy at Transformers pre-training (we tried ranks up to 32). As Table 3.6 shows CGX has superior performance on single node over PowerSGD in spite of lower compression. This is because 1) higher compression shows diminishing returns, 2) CGX has lower compression overhead (Table 3.3), and 3) CGX implements faster reductions.

Sparsification. We implemented the TopK [DJMVE16] algorithm as part of the CGX framework. Usage of the sparcification compression there faces the following issues. In

order to converge under standard parameters, sparcification must be applied upon the entire model, not layer-wise which is impossible due to specifics of the communication frameworks (`torch.distributed`, `Horovod`). In our experiments we did not manage to make topK with error feedback converge with a similar to QSGD compression rate. Moreover, topK with higher compression rates did not show any speedup in comparison to QSGD due to compression saturation on our workstation (see Figure 3.1) and higher topK overhead (see Table 3.3, we used [MAEAC21] topK mechanism).

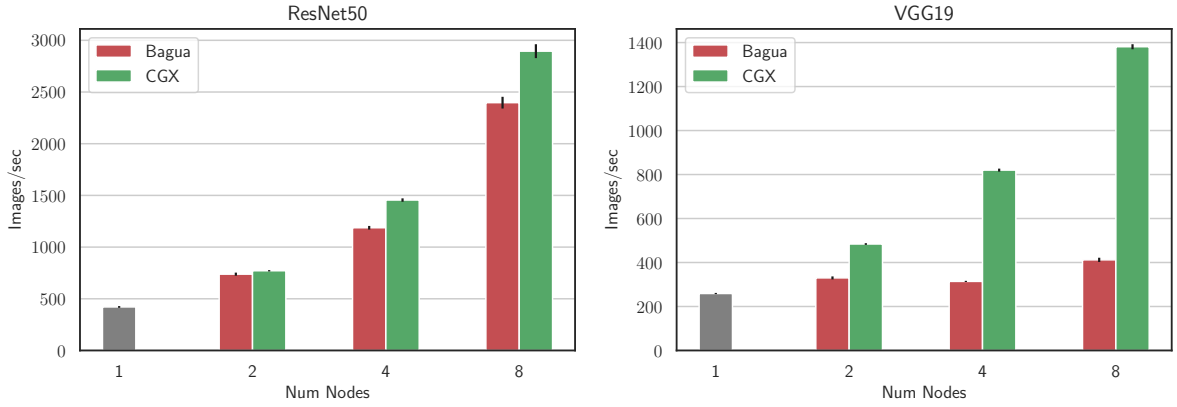
Comparison with other systems

GRACE Comparison. We adapted our benchmarks to also compare to GRACE [XHA+21], which also implements quantization and sparsity compression techniques. We used the same uniform 4-bit compression variant for frameworks, as this recovers accuracy. We used NCCL as the communication backend for GRACE, as it provided the best performance in our setting. We found (see Table 3.6) that CGX outperforms GRACE by more than 3x on average. Our profiling suggests that this occurs because GRACE uses a less effective reduction scheme (NCCL-Allgather vs. optimized Allreduce), less efficient compression (e.g., no bucketing), and transmission (even with 4 bits compression, GRACE communicates in INT8). We also tried GRACE with very high-sparsity TopK compression (0.001), and performance did not improve significantly. This suggests that GRACE's implementation has additional bottlenecks in terms of communication latency.

NCCL and QNCCL Comparison. As shown in Figure 3.1, NCCL has poor scaling on commodity machines, especially from 4 to 8 GPUs, where communication cost is highest. CGX can give $>$ 2x speedup relative to NCCL, reaching 80-90% of linear scaling. This enables the consumer-grade RTX3090 GPU to match or even surpass the throughput of a DGX-1 server. We found that QNCCL partly alleviates the scaling problems of NCCL, and only improves throughput by a limited margin, as it does not benefit from the bespoke communication backend integrated in CGX. An orthogonal issue for QNCCL is the fact that it has higher accuracy degradation: since compression cannot be performed layer-wise (as QNNCL does not have layer information), it cannot perform layer-wise compression. We have been able to recover accuracy within 1% with QNCCL at 4bit compression by reducing bucket size to 128 for all models, but this comes with a further performance reduction.

Bagua and HiPress Comparison. Bagua [GJY+21] and HiPress-CaSync [BLZ+21] are distributed training frameworks, which also support some generic forms of gradient quantization. In multinode experiments on 4x EC2 p3.8xlarge instances with 4 V100 GPUs each, we observed that Bagua and HiPress have similar performance to CGX on the smaller ResNet50 model, and that they are up to 10% slower on the larger VGG19 model. This is since all frameworks use the same NCCL backend for inter-node communication, but CGX uses a faster pattern (SRA vs Ring or Tree for NCCL) than HiPress and has better compression rate than Bagua (which only supports 8 bit quantization). Moreover, HiPress only supports 2 bit quantization, e.g. [WXY+17, Str15]), which does not converge under standard parameters for Transformed-based models.

HiPress unfortunately does not support the newer commodity RTX-3090 GPUs, so we could only compare with Bagua on the Genesis Cloud 8xRTX3090 instance. The results of the comparison are presented in Figure 3.6, showing that CGX provides clearly superior performance, especially for the VGG19 model.



(a) ResNet50 (b) VGG19

Figure 3.6: Scaling throughput in multi-node environment for image classification tasks. Bagua vs CGX.

Comparison with Hardware Bandwidth Overprovisioning.

We now turn to Figure 3.5 where we first observe that, although in terms of single-GPU performance the RTX3090 is comparable to the V100/DGX-1, it has poor multi-GPU scaling for large models when using the standard NCCL setup ($< 50\%$ of linear scaling). The older 2080 GPUs have lower throughput both due to both lower memory, limiting maximum batch size, as well as lower computational power (Fig. 3.5a). Thus, we mainly focus on 3090 GPUs.

If we compare the maximum achievable performance (ideal scaling), CGX achieves similar results to the bandwidth overprovisioning approach, on both the DGX and the A6000 machines. In other words, CGX allows us to get bandwidth-overprovisioning performance via a "middleware" approach, achieving our stated goals. The remaining percentage gaps from perfectly linear scaling are because of 1) latency costs, 2) inefficiencies in our implementation, and 3) remaining communication costs, especially in early layers, which cannot be overlapped with computation. To measure this, we artificially removed the bandwidth bottleneck, by sending only a small number of elements per layer. The results in Table 3.7 show that CGX is close to ideal bandwidth reduction.

Table 3.7: Ideal performance (% of linear scaling) achievable via bandwidth-overprovisioning for different workloads, relative to CGX.

	ResNet50	VGG16	TXL	BERT	ViT
Ideal Perf.	92 %	91 %	95 %	88 %	95 %
CGX Perf.	90%	84 %	87 %	75 %	93 %

Table 3.8: Comparison of adaptive methods. Speedups and compression rates are relative to static bits-width assignment (4 bits). Experiments are run with Transformer-XL base model on 8 RTX3090 GPUs (single node) and 4 nodes with 4xRTX3090 GPUs each (multi-node). Accordion is applied to QSGD with 3 and 4 as compression bounds.

	Compression	Speedup 1-Node	Speedup Multi-Node
KMEANS	**1.47**	**5%**	**40%**
Bayes	1.34	3%	30%
Linear	1.15	2%	13%
Accordion	1.21	3%	15%

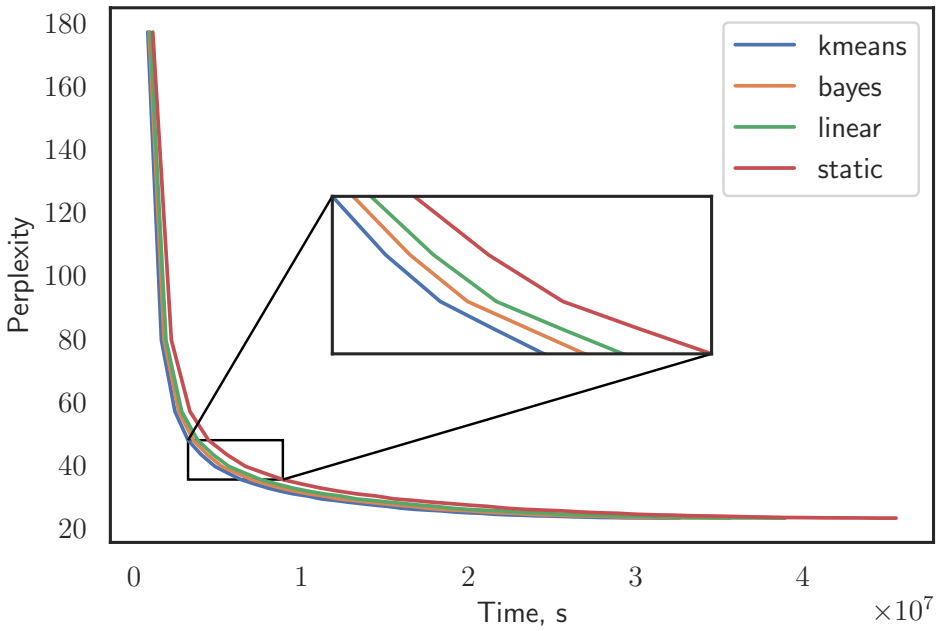
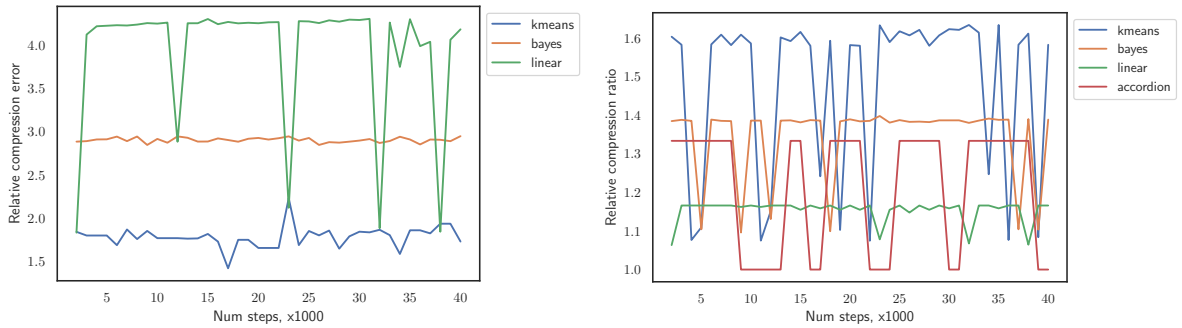


Figure 3.7: Transformer-XL training with adaptive schemes.



(a) Compression error.

(b) Compression ratio.

Figure 3.8: Comparison of adaptive compression approaches. Error and size compression are shown relative to uniform static assignment of compression parameters to 4 bits.

3.7.3 Layer-wise Adaptive Compression

So far, we have provided results for our version of 4bit quantization, which always recovers accuracy. We now examine additional performance savings due to adaptive compression. Across all models, the automated procedure in Section 3.5.1 identifies large layers with low-performance sensitivity (e.g. fully-connected or embedding layers) for lower bit-widths, and has similar total compression error to uniform compression. We illustrate this on Transformer-XL, the model with the most non-uniform layer sizes. We conducted single-node experiments on an 8xRTX3090 machine, and multi-node on four 4xRTX3090 machines. As before, the baseline is 4-bits static compression, which was shown to recover full accuracy. Figure 3.7 represents perplexity against time for different selection mechanisms. Figures 3.8a and 3.8b represent compression error and compression ratio relative to static assignment. Table 3.8 shows that Bayesian optimization shows stable compression error, and good *average* compression. Yet, the kmeans-based method shows the lowest quantization error, best average compression, and highest speedup, as it tends to compress large layers more. Specifically, this can lead to additional improvements in the order of 5% on a single node and up to 40% in multinode

setting, without accuracy loss. This approach can still be improved by taking into account *runtime speedups* instead of absolute compression.

Among existing adaptive schemes, AdaComp [CCB+18] and Accordion [AWL+21] are the only ones which can be adapted to our setting. For comparison, we execute the Transformer-XL model on a language modelling (LM) task. AdaComp basically suggest adaptive scheme for sparsification, with possible further quantization of communicated elements. We first applied AdaComp only for sparsification: however, unfortunately the compression assignment provided by AdaComp did not converge to reasonable accuracy on this task.

Second, we adapted Accordion to our framework with QSGD compression, using Accordion to choose bit-width parameters based on its critical regimes detection approach. We used Accordion with hyperparameter $\eta = 0.5$, as suggested by the authors, and updated the compression parameter every 1k steps of training. As the lower and higher compression levels, we checked (2, 4) and (3, 4). The first pair resulted in significantly lower final accuracy relative to the baseline. The second pair (3,4) recovered the final accuracy, but the compression ratio was inferior to all the other adaptive schemes we investigated, and considerably below our proposed clustering scheme. Please see Figure 3.8b and Table 3.8 for an illustration. For instance, our adaptive scheme resulted in 17% additional multi-node speedup compared to Accordion.

3.7.4 Practical Implications

Multi-node experiments. Next, we examine performance on multi-node training in the cloud. We used 4 4xRTX3090 Genesis instances with 10Gbps intra-node bandwidth and 5 Gbps inter-node bandwidth. Table 3.9 shows that CGX provides up to 10x speedup over the uncompressed baseline.

Table 3.9: Items per second when training with the NCCL and CGX optimizations, respectively, on 4 machines with 4 RTX3090 GPUs each.

	ResNet50	ViT-base	Transformer-XL-base	BERT
Baseline	564	34	32k	1.4k
CGX	2.3k	235	85k	12k

Implications for Cloud Training. Several cloud services provide servers with commodity GPUs [Gen21, Lam21, Lea21]. We therefore compare a standard AWS EC2 4xV100 GPU instance (p3.8xlarge) instance with a 4xRTX 3090 Genesis Cloud instance [Gen21]. We execute the same training benchmark, with and without CGX. The numbers in Table 3.10 show that CGX allows us to obtain almost *twice* higher throughput (training tokens/second) per dollar on the more affordable Genesis instance, for a standard language modelling task (SQuAD) task using an industry-standard BERT model.

Table 3.10: Comparison of training performance for different cloud services (AWS and Genesis) with and without CGX. The training task is BERT-QA and achieves full accuracy.

Instance	Throughput (1K tok./sec)	Price per hour ($)	Tokens/second per $
Genesis + NCCL	4737	6.8	696
AWS + NCCL	**14407**	12.2	1181
Genesis + CGX	**14171**	6.8	**2083**

CHAPTER 4

Layerwise-Adaptive Gradient Compression for Data-Parallel Training

4.1 Preface

As we discussed in the previous chapters data-parallel distributed training of deep neural networks is widespread, but it still experiences communication bottlenecks. To mitigate these bottlenecks, various compression techniques like quantization, sparsification, and low-rank approximation have been developed. However, many implementations remain sub-optimal because they apply compression uniformly across DNN layers, disregarding the layers' differences in parameter count and impact on accuracy.

In this chapter, which follows [MAFA24] with minor changes, we provide a general framework for dynamically adapting the degree of compression across the model's layers during training, improving overall compression while leading to substantial speedups, without sacrificing accuracy. Our framework, called `L-GreCo`, is based on an adaptive algorithm that automatically picks the optimal compression parameters for model layers, guaranteeing the best compression ratio, while satisfying an error constraint.

4.2 Introduction

A popular approach for reducing the cost of gradient communication, which is the main focus of this chapter, is *lossy gradient compression* [SFD+14, AGL+17, DJMVE16, VKJ19], which reduces the number of communicated bits per iteration. Hundreds of such techniques have been proposed, which can be roughly categorized into three method families. The first is *quantization* [SFD+14, AGL+17, WXY+17], which reduces the bit width of the communicated gradients in a variance-aware fashion in order to preserve convergence. The second is *sparsification* [Str15, DJMVE16, LHM+17], reducing the number of gradient components updated at every step, which are chosen via various saliency metrics. The third and most recent approach is *low-rank approximation* [WSL+18, VKJ19], which leverages the low-rank structure of gradient tensors to minimize communication.

In practice, these approaches come with trade-offs in terms of compression versus ease of use. For instance, gradient quantization is easy to implement and deploy, but only provides limited compression before accuracy degradation; sparsification and low-rank approximation

can provide order-of-magnitude compression improvements, but come with additional costs in terms of maintaining error correction and careful hyper-parameter tuning. These trade-offs have been investigated via *adaptive* compression methods [AWL+21, MRA22, FTM+20], which adjust the compression to the error incurred during various phases of DNN training.

Currently, there is still a significant gap between ideal, theoretically-justified compression methods, and their efficient systems implementations. For example, the theory of gradient compression [KRSJ19, NMC+21] suggests that the ideal compression method in terms of convergence is that which minimizes the *norm of the compression error*, i.e., the difference between the global model gradient and the compressed one. (For instance, global TopK selection provides the best sparsification-based compressor for a given parameter K [SDMA+21].) Yet, parameter selection based on a global gradient at each step is not practical from the systems perspective: performing global selection, such as TopK, requires waiting for the whole model gradient to be available; yet fast implementations of data-parallel training transmit each layer's gradients as soon as they are generated, overlapping communication and computation. Moreover, many existing implementations miss significant opportunities for optimization: modern models such as Transformers [VSP+17] are highly heterogeneous in terms of both layer sizes and layer sensitivity to gradient compression (see Figure 4.1), and gradient compression impacts various stages of training differently [ARS18].

This gap between the theory and practice of gradient compression, led some to question the usefulness of this approach [AWVP]. In this chapter, we address this gap and show that, when paired with efficient and adaptive systems support, gradient compression can be a powerful technique for efficient data-parallel training. Specifically, the question we address is the following: Given an arbitrary model and gradient compression technique, is there an efficient way to balance accuracy constraints, such as *layer sensitivities*, and the communication constraints, such as *layer sizes*, dynamically during training in order to maximize speedup, without sacrificing theoretical convergence and practical accuracy?

To address this, we introduce `L-GreCo`, an efficient and general framework for Layer-wise parametrization of GRadiEnt COmpression. At the algorithmic level, L-GreCo is based on a new formalization of the *layer-wise adaptive compression* problem, which identifies per-layer compression parameters, e.g. per-layer sparsity or quantization levels, seeking to maximize compression, under a fixed constraint on the total compression error, which ensures both good theoretical convergence, and negligible accuracy loss. At the system level, `L-GreCo` works by integrating with standard training frameworks, such as `torch.distributed`, to exploit model heterogeneity in terms of both per-layer structure and per-layer sensitivity, determining on-the-fly by how much to compress individual layer gradients in order to maximize compression or end-to-end training times.

We validate `L-GreCo` across *all existing families of compression strategies*: quantization, sparsification, and low-rank compression across a variety of vision and language tasks. `L-GreCo` consistently achieves higher compression rates than existing manual or adaptive strategies [VKJ19, AWL+21], in a black-box fashion, and is particularly effective for modern Transformer models, in both single- and multi-node settings.

We summarize our contributions as follows:

- We introduce a new approach leveraging the heterogeneous structure of DNNs in order to reduce communication overheads while maintaining convergence, guaranteeing optimal layer-wise compression-based by balancing a theoretically-justified error metric with an optimization objective maximizing compression.

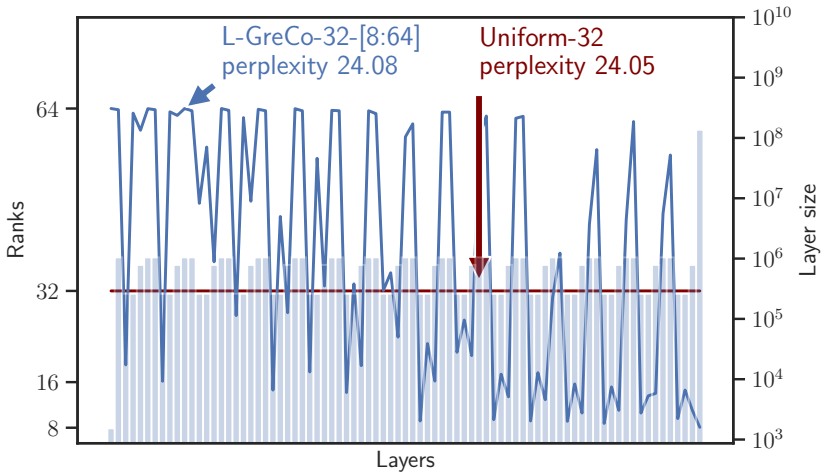


Figure 4.1: Profile of L-GreCo rank choices for PowerSGD compression on Transformer-XL. The red line represents uniform compression, while the blue line represents the L-GreCo profile. Transparent bars show layer sizes. Layers are indexed in the order they are communicated. The annotated number is the final test perplexity (ppl) for the experiment (lower is better). Here, the average compression of L-GreCo is **1.5x higher** than uniform.

- We provide an extensive empirical evaluation on different neural networks (ResNet18, ResNet50, Transformer-XL, Transformer-LM) with different datasets (CIFAR-100, ImageNet, Wikitext-103) showing that L-GreCo reduces communication by up to 5× and achieves speedups up to 2.5× without loss of accuracy or significant tuning, across both single and multi-node deployments.
- In addition, we conduct the first in-depth study of both sensitivity and performance metrics. We show that the theoretically-justified error norm metric is essentially equivalent to more complex metrics based on examining output loss. From the performance perspective, we show that optimization objectives seeking to maximize absolute compression lead to similar results to objectives that minimize transmission time.
- Finally, we show that L-GreCo is compatible with prior adaptive compression schemes; specifically, it can be extended to use information about the different stages of training [AWL+21], leading to further performance improvements.

4.3 Related work

Compression methods. Gradient compression usually employs three strategies: quantization, sparsification, and low-rank decomposition. Quantization methods [SFD+14, AGL+17, WXY+17, LAK18, RKFM+21] use lower precision of each gradient component, reducing the number of transmitted bits. They are easy to implement and work under stable hyper-parameters [AGL+17, XHA+21, MRA22]. However, their compression is limited by the fact that at least one bit per entry must usually be transmitted. Sparsification techniques [Str15, DJMVE16, LHM+17, KRSJ19] circumvent this by identifying *salient components* in the gradient and only transmit such subsets. Finally, gradient decomposition algorithms [VKJ19, WSL+18] use the fact that the layer-wise gradient tensors are known to be well-approximable via low-rank matrices and aim to design light-weight projection approaches that also provide low error. Sparsification and low-rank techniques usually require *error correction* buffers to preserve good convergence, as well as non-trivial hyper-parameter

tuning. As we show experimentally, `L-GreCo` is compatible with all of these approaches and can provide significant additional bandwidth savings for each such strategy without sacrificing model accuracy and without tuning.

Adaptive Compression. The general idea of *adapting* the degree of compression during training has been investigated by AdaComp [CCB+18], which proposes a self-tuning adaptive compression method; yet, their method does not adapt compression parameters per layer and cannot be combined with other compression approaches. [FTM+20] adapts the quantization grid to the gradient distribution; yet, their approach is specifically tuned to quantization, and oblivious to layer heterogeneity.

`Rethink-GS` [SDMA+21] optimize *the total error over steps* for sparsification-based compression and suggest threshold global sparsifiers, which are shown to reach higher compression rates than uniform per-layer compression on small vision tasks (e.g. ResNet18 on CIFAR10/100). However, their approach is restricted to sparsification and leaves unclear how to tune the threshold for large-scale, sensitive models such as Transformers or ImageNet-scale models. In particular, we were unable to obtain good results with this approach on models such as Transformer-XL or Transformer-LM. In Section 4.6.3, we present a comparison with their approach on ResNet18/CIFAR-100, showing that our method yields both higher accuracy and higher compression.

Accordion [AWL+21] adapts the compression parameters of sparsification and low-rank compression based on the critical regimes of training. The algorithm alternates between two compression levels ("low" and "high"), provided by the user and is prone to accuracy loss. Our approach improves upon Accordion in terms of speedup, but also that we can combine our method with Accordion and obtain even higher gains. CGX(Chapter 3) investigated a kmeans-based heuristic, which we show experimentally to be sub-optimal.

To our knowledge, our dynamic programming strategy has not been employed in the context of adaptive gradient compression. Related approaches have been investigated in the context of weight compression for DNNs, see e.g. [WLCC20, ANL+20, FA22, SYM+22]. Yet, there are major differences: (1) the error metrics and speedup objectives are different in the case of weight compression; (2) we execute online at training time, which means that our algorithm has to be extremely efficient and adapt to dynamic inputs.

Recently, Agarwal et al. [AWVP] questioned the utility of gradient compression for distributed training. Yet, their study is limited in the sense that they only consider a limited subset of compression methods and focus on NCCL-type implementations on bandwidth-overprovisioned networks. By contrast, we consider a more flexible system implementation that allows to map layer parameters to different compression levels and show practically that L-GreCocan yield speedups both on commodity single-node GPU servers and on general multi-node systems.

4.4 Problem Formulation

Goals. Assuming we are given a DNN model $\mathcal{M}$ with L layers and a compression technique, we would intuitively like to find a choice of compression parameters c_ℓ, one for each layer $\ell \in \{1, 2, \ldots, L\}$ which would minimize a metric representing damage of the training quality introduced by compression while minimizing the total number of bits transmitted. Yet, this intuitive description leaves open a range of details, such as 1) the notion of layer-wise metric that corresponds to the compression effect for a given set of parameters; 2) the exact problem

formulation, constraining the compression effect or the compression ratio; and 3) an efficient implementation of such an algorithm.

Sensitivity Metrics. Since choosing the right sensitivity metric is key for accuracy recovery, we have investigated two different approaches. First, the sensitivity of a layer to gradient compression can be measured by the impact on the model's loss. To evaluate this, we set up the following experiment: We saved model checkpoints at different stages of the uncompressed training. Then, we conducted multiple short runs (50 steps) with the same data starting from the checkpoint, varying compression parameters. We use the difference of loss with and without compression as the metric. Since we wish to observe the model's reaction to the compression of individual layers, we vary compression parameters for each layer, not compressing other layers' gradients. With that, we collect the differences of loss for each layer and the compression parameter as the metric.

Another approach is based on gradient compression theory, which shows formally that the *squared ℓ_2 compression error* is a good measure of the convergence impact of compression technique [KRSJ19, NMC+21, SDMA+21]. Here, we aggregate gradients for the training, then compress the aggregated gradients for each compression parameter and for each layer individually, and use the magnitude of the error as a metric.

As shown in Section 4.6.4, the two approaches are strongly correlated, and the resulting optimal parameters are close to each other. However, the loss-based approach is less practical, as it requires *offline* evaluation, altering the original training pipeline and additional training time. By contrast, the error-based approach can be used *online* during training and has negligible overheads. Thus, we use the L2 norm of the compression error as the main sensitivity metric.

The Constrained Optimization Problem. Given an error metric, we formalize the layer-wise compression optimization problem as follows. Given a model $\mathcal{M}$ with L layers $\ell \in \{1, 2, \ldots, L\}$ and a compression technique, providing a set of compression choices $\mathcal{C} = \{c^1, c^2, \ldots c^k\}$ for each layer. We emphasize that, for simplicity, we consider a single compression technique and the same compression choices/levels for each layer, but our approach would also work for different techniques being applied to the same model and heterogeneous compression choices.

In this context, our method receives as input an error function $error(\ell, c^j)$, which provides the L2 norm of the compression error at layer ℓ for compression choice c^j, and a function $size(\ell, c^\ell)$ which measures the transmission cost of layer ℓ for choice c^ℓ. In addition, we assume to be given a fixed maximal error threshold $\mathcal{E}_{\max}$ which the algorithm should not violate. Then, we wish to find a layer-wise setting of compression parameters $c_1, \ldots, c_L$ with the objective:

$$\text{minimize } \sum_{\ell=1}^{L} size(\ell, c^\ell) \text{ s.t. } \sum_{\ell=1}^{L} error(\ell, c^\ell) \leq \mathcal{E}_{\max}.$$

Practically, this formulation minimizes the total transmission cost for the gradient tensors under a maximum *additive* constraint on the gradient compression error. One implicit assumption is that the metric $error(\cdot, \cdot)$ is *additive* over layers and that it is possible to obtain a "reference" error upper bound, which does not result in accuracy loss. We will see that this is the case for the error metric we adopt.

The Error Bound. We pick the error bound $\mathcal{E}_{\max}$ to track that of a reference compression approach which is *known not to lose accuracy relative to the baseline*. Here, we leverage the fact that the literature provides parameters that allow reaching full accuracy recovery

for different models and datasets. For instance, for quantization, we track the error of *4-bit quantization*, known to recover for every model [AGL+17, MRA22]. For sparsification, [LHM+17, RAA+19] as well as for matrix decomposition [VKJ19], we use different reference parameters according to their baselines. For details, please refer to Tables 4.2 and 4.3. An interesting consequence of this choice is that, since we guarantee ℓ_2 compression error, which is a small constant factor of the error of these theoretically-justified approaches, we inherit similar convergence guarantees, as per [NMC+21].

4.5 The L-GreCo Framework

Overview. We now describe a general algorithm to solve the constrained optimization problem from the previous section. Our algorithm makes layer-wise decisions in order to balance the magnitude of the compression error and the compressed size of the model. As inputs, our algorithm takes in the uncompressed layer sizes $size(\ell, \perp)$, a set $\mathcal{G}$ of accumulated gradients per layer (which will be used to examine compression error), as well as a fixed error bound $\mathcal{E}_{\max}$. Specifically, at a given decision step, the objective is to find an optimal mapping of each layer ℓ to a compression level c_ℓ, such that the norm of the total compression error, computed over the set of accumulated gradients $\mathcal{G}$ does not surpass $\mathcal{E}_{\max}$, but the total compressed size of the model $\sum_{\ell=1}^{L} size(\ell, c_\ell)$ is minimal for this error bound.

This formulation is reminiscent of the *knapsack problem*: the error is the size of the knapsack, and the compressed size is the value we wish to optimize. In this formulation, the problem would have an efficient optimal algorithm using dynamic programming (DP). However, the squared L_2 error is not discrete, so we cannot directly apply this approach. Instead, we reduce this to a solvable problem by *discretizing* the possible set of error values. Since errors are monotonic and we can use a very fine discretization without significant efficiency loss, it is unlikely that we would miss the optimal solution by a significant amount. For illustration, in our implementation, we use $D = 10000$ as a discretization factor (i.e. steps of size $\mathcal{E}_{\max}/D$).

The Algorithm. The procedure, presented in Algorithm 4.1, works as follows. First, we compute the data needed for the algorithm for all layers and all considered compression parameters (lines 1-10), corresponding to errors and compressions for each possible choice. Then, we execute a dynamic programming algorithm to solve the following problem. We want to compute the minimum total size given total compression error E in the first ℓ layers $\mathcal{C}(\ell, E) = \min_\ell \mathcal{C}(\ell - 1, E - error(\ell, c_\ell)) + size(\ell, c_\ell)$. To achieve this, for each layer we want to consider, we run over all error increments and all possible compression parameters and minimize the total compressed size for the current total compression error (lines 12-22), saving the compression parameter with which we obtain the minimum. Then, in lines 23-27, we find the error increment achieving the lowest total compressed size and reconstruct the compression parameter mapping—obtaining the result.

System Implementation. We integrate `L-GreCo` between the user training code and the communication system responsible for gradient compression and synchronization. We run the above algorithm periodically, e.g., once per training epoch, on a single designated worker; unless otherwise stated, this worker performs all steps. In between runs of the algorithm, we accumulate per-layer gradients in auxiliary buffers. We then build an L_2 error table for each layer, for every compression parameter in the user-provided range, and for the reference compression parameters set. To find the error, we simulate the compression/decompression of each layer with the given compression parameter without applying error feedback and compute the L_2 distance between the original and recovered vectors. Then, we run the DP

Algorithm 4.1: L-GreCo adaptive compression

Input: Model Layers L_i, accumulated gradients G_i, compression parameters $C = \{c^1, c^2, \ldots, c^k\}$, static default compression parameters to improve C_i^d, discretization factor D
Output: Compression assignments $c_\ell \in C$ for each layer ℓ

N = number of layers
Compute $\mathcal{E}_{\max}$ for the default compression parameters C_i^d
Compute discretization step $\mathcal{E}_{\max}/D$.
Costs matrix $N \times |C|$ where position i, j has a value of the size of layer i compressed with compression parameter c^j.
Errors matrix $N \times |C|$ where position i, j has a value of the discretized L_2 of the compression error when the accumulated gradients of layer i are compressed with parameter c^j.
DP matrix $N \times (D+1)$ filled with ∞ values.
PD matrix $N \times (D+1)$.
// Initialization of the cost tables:
for $c \in C$ **do**
    $DP[1][Errors[1][c]] = Costs[1][c]$
    $PD[1][Errors[1][c]] = c$
end for
// Dynamic programming algorithm
for Layer $l_i := 2..N$ **do**
    for $c_i \in C$ **do**
        for $e_i := Errors[l_i][c_i]..D$ **do**
            $t = DP[l_i - 1][e_i - Errors[l_i][c_i]] + Costs[l_i][c_i]$
            if $t < DP[l_i][e_i]$ **then**
                $DP[l_i][e_i] = t$
                $PD[l_i][e_i] = c_i$
            end if
        end for
    end for
end for
$err_{min} = argmin(DP[N])$
// Reconstruction of the optimal parameters
for $l_i = N..1$ **do**
    $result[l_i] = PD[l_i][err_{min}]$
    $err_{min} = err_{min} - Errors[l_i][result[l_i]]$
end for
return $result$

algorithm. This provides us with the optimal compression mapping, which the designated worker broadcasts to the other workers. Then, on each worker, we save the parameters mapping in the communication engine.

Computational and memory costs. The algorithm assumes that we accumulate gradients in additional buffers, occupying the model size memory. The DP algorithm has $O(D|L||C|)$ time complexity and $O(|L|D)$ memory complexity. The actual timings for the algorithm are presented in Table 4.1. The overheads consist of 1. *error computation*, and 2. *running dynamic programming*. Dynamic programming takes a minor fraction of training time, whereas most of the overhead is caused by computation. However, both overheads are negligible compared to the speedups provided by `L-GreCo` (see Figures 4.3 and B.5).

Communication details. In a data-parallel implementation, gradients become available

Table 4.1: Timing overheads for L-GreCo in relation to the total training time. Numbers in brackets represent error computation.

Model	Description	ResNet50
PowerSGD	$0.56\%[0.49\%]$	$0.15\%[0.14\%]$
QSGD	$0.14\%[0.13\%]$	$0.04\%[0.03\%]$
TopK	$0.38\%[0.35\%]$	$0.33\%[0.30\%]$

right after the backward propagation of the corresponding layer, and are grouped into several buffers—called *buckets* in Pytorch)—allowing the overlapping of gradient communication with further computation. Thus, the communication of the first buckets is completely "hidden" by computation, whereas the synchronization of the last bucket becomes a significant part of the timing delay between steps. Thus, the transmission time of different buckets has a different impact on the training speed. Hence, in theory, optimizing for the compression ratio might yield suboptimal results in practice.

Optimizing for Time. To showcase the flexibility of L-GreCo, we augmented our system to allow us to measure the *actual synchronization time* for each gradient bucket, allowing the algorithm to optimize directly for *communication times*. We also reformulated the optimization problem: to minimize the *length of time* between the start of the first bucket synchronization and the end of the last one, rather than the compression ratio. Then, we train a regression model to learn the relation between the transmitted bucket sizes and gradient synchronization time. Thus, we obtained per-bucket coefficients $T(b)$, which we can apply for each layer in a respective bucket. Then we change the objective in the optimization problem(see Formula. 4.4) to:

$$\text{minimize } \sum_{\ell=1}^{L} size(\ell, c^{\ell}) * T(\ell) \text{ s.t. } \sum_{\ell=1}^{L} error(\ell, c^{\ell}) \leq \mathcal{E}_{\max}.$$

4.6 Experimental Validation

We experimentally evaluate L-GreCo across all existing compression strategies: quantization using QSGD, TopK sparsification, and low-rank approximation via PowerSGD.

4.6.1 Experimental setup

Infrastructure. Our evaluation uses commodity workstations with 4 or 8 NVIDIA RTX3090 GPUs. In the multi-node setting, we use 4 cloud instances with 4xRTX3090 GPUs provided by Genesis Cloud. Bandwidth measurements show that inter-GPU bandwidth values lie between 13 to 16 Gbps, and inter-node bandwidth in the cloud is up to 10 Gbps. We used Pytorch 1.10, openmpi/4.1.4, CUDA 11.3, NCCL 2.8.4, and cudnn/8.1.1.

Implementation. We implemented L-GreCo in PyTorch using `torch.distributed` hooks for PowerSGD and leveraging the open-source CGX framework(Chapter 3) for basic quantization and sparsification operations.

Datasets and models. We examine two different DNN learning tasks: 1) image classification on the CIFAR100 [Kri09] and ImageNet [DDS+09] datasets, and 2) language modeling on WikiText-103 [MXBS16]. We used state-of-the-art model implementations and parameters provided by the PyTorch version of the NVIDIA Training Examples benchmark [Nvi20] and the `fairseq` library PyTorch examples [OEB+19]. We used ResNet-18 for CIFAR-100 training with batch size 256, ResNet-50 in the mixed-precision regime for ImageNet with batch

Table 4.2: Accuracy recovery and compression ratios for different compression methods with uniform and adaptive schemes on image classification tasks. The compression ratios measure actual transmission savings. Values in brackets for L-GreCo compression ratios stand for improvements relative to the corresponding uniform compression.

Compression approach	Parameter choice	ResNet18 on CIFAR-100			ResNet50 on ImageNet		
		Default param	Accuracy	Compression ratio	Default param	Accuracy	Compression ratio
Baseline	N/A	-	76.60 ± 0.40	1.0	-	76.88 ± 0.16	1.0
QSGD	uniform	4 bit	76.80 ± 0.40	7.8	4 bit	77.38 ± 0.10	7.7
	L-Greco		76.46 ± 0.21	8.6 [1.10×]		76.77 ± 0.25	11.0 [1.41×]
TopK	uniform	1%	75.73 ± 0.46	48.1	1%	76.85 ± 0.06	45.6
	L-Greco		75.66 ± 0.35	182.0 [3.78×]		77.04 ± 0.27	122.0 [2.67×]
PowerSGD	uniform	rank 4	76.36 ± 0.28	72.2	rank 4	76.50 ± 0.37	66.5
	L-Greco		76.43 ± 0.37	133.9 [1.85×]		76.33 ± 0.27	96.2 [1.44×]

Table 4.3: Accuracy recovery and compression ratios for different compression methods with uniform and adaptive schemes on language modeling tasks. The compression ratios measure actual transmission savings. Values in brackets for L-GreCo compression ratios stand for improvements relative to the corresponding uniform compression.

Compression approach	Compression parameters	TransformerXL on WIKITEXT-103			TransformerLM on WIKITEXT-103		
		Default param	Perplexity	Compression ratio	Default param	Perplexity	Compression ratio
Baseline	N/A	-	23.82 ± 0.10	1.0	-	29.34 ± 0.12	1.0
QSGD	uniform	4 bit	23.82 ± 0.1	7.8	4 bit	29.39 ± 0.10	7.8
	L-Greco		24.11 ± 0.09	9.1 [1.16×]		30.03 ± 0.16	9.9 [1.26×]
TopK	uniform	10%	24.13 ± 0.14	4.9	10%	29.29 ± 0.09	4.9
	L-Greco		24.19 ± 0.13	12.8 [2.61×]		29.08 ± 0.20	25.6 [5.2×]
PowerSGD	uniform	rank 32	24.08 ± 0.12	14.0	rank 32	29.98 ± 0.09	15.0
	L-Greco		24.09 ± 0.15	20.8 [1.48×]		30.19 ± 0.09	26.5 [1.76×]

size 2048, and Transformer-XL and Transformer-LM trained in full-precision for WikiText-103, with batch sizes 256 and 2048, respectively. **All our experiments use the original uncompressed training recipes, without any additional hyperparameter tuning to account for gradient-compressed training.**

Baselines. The first natural baseline is uncompressed training, which sets our accuracy baseline. Matching MLPerf [MRC+20], we set our accuracy threshold to 1% relative to uncompressed training. The second natural baseline is the *best existing manually-generated* gradient compression recipes. By and large, existing methods propose *uniform* per-layer compression to a given threshold, e.g. [AGL+17, WXY+17, RAA+19, VKJ19]. For such baselines, we want to improve compressed size and training speed, possibly also improving final model accuracy. We found that the best choice of compression parameters for uniform per-layer assignment depends on the compression method, dataset, and task. For some experiments, we had to tune the uniform compression parameters to match baseline (non-compressed) results (see Tables 4.2 and 4.3).

Parameter ranges. L-GreCo requires a range of possible compression parameters as input. We have always chosen this range to include the default compression parameters used in the

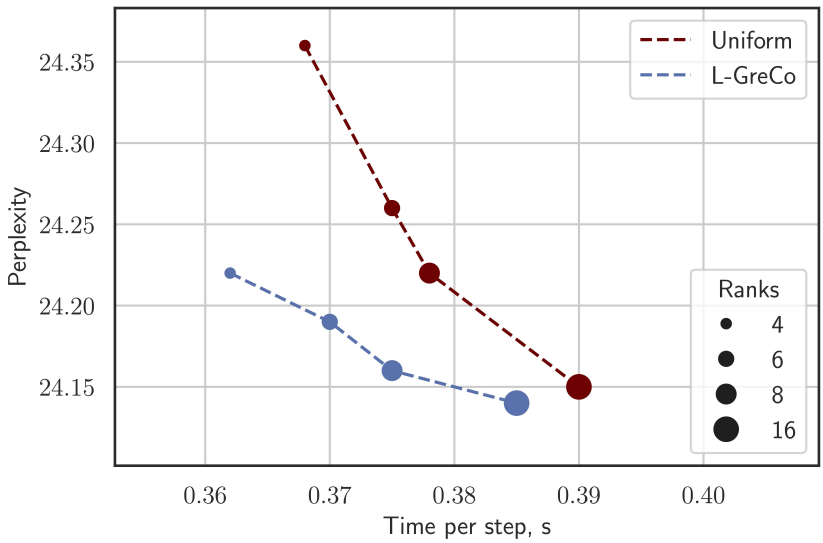


Figure 4.2: Perplexity (lower is better) vs. time per step (smaller is better) for different ranks of PowerSGD compression for the Transformer-XL wikitext-103 task with uniform or `L-GreCo` suggested compression schemes. Single node, 8 RTX3090 GPUs.

literature. Moreover, we left a gap between the default parameter and the highest possible compression parameter (the right range bound) — otherwise, we are limited to matching L_2 error, and a gap between the default parameter and the lowest possible compression (the left range bound)—otherwise, we will not improve compression. We use the following approach for deciding ranges. Assume a default uniform compression parameter D, e.g., 4 quantization bits. For quantization and low-rank methods, the search space was defined as $[D/2, 2 * D]$ with an incremental step of 1. For sparsification, we chose $[D/10, 10 * D]$ with an increment of $D/10$.

The other two input parameters of `L-GreCo` are how frequently the algorithm is run and the warm-up period after which the compression is turned on. The first parameter matches the evaluation period (typically, 1 epoch). As we will see (Figure 4.5a), the compression ratio of the schemes returned by `L-GreCo` is relatively stable, so it does not need a frequent re-adjustment. The warm-up period equals the default learning rate warm-up period.

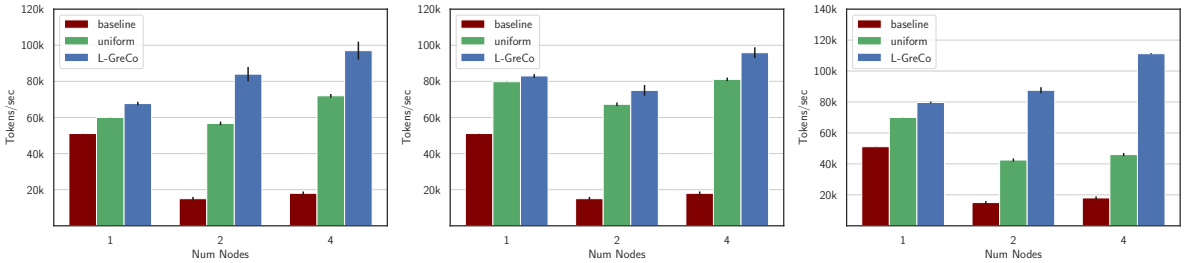
4.6.2 Evaluation results

Accuracy recovery. We first examine model accuracies using standard recipes for end-to-end training. For each experiment, we performed 3 runs with different seeds. We compare `L-GreCo` recovery with the uncompressed (baseline) and the best uniform per-layer compression parameters (uniform). The results are presented in Tables 4.2 and 4.3, including seed variability. The *compression ratio* represents actual transmission cost savings versus the uncompressed baseline. For each compression method and training task, we show the parameter value that provides the highest compression ratio while recovering final accuracy, i.e., further uniform compression leads to *worse* convergence.

Overall, results show that `L-GreCo` stays within the accuracy recovery limit of 1% multiplicative error [MRC+20] for most tasks, often being close to the uniform baseline while consistently increasing the compression ratio across all the tasks and compression techniques. We stress that we did not perform task-specific parameter tuning. The gains are remarkably high for sparsification and low-rank techniques, where the search space and savings potential are higher.

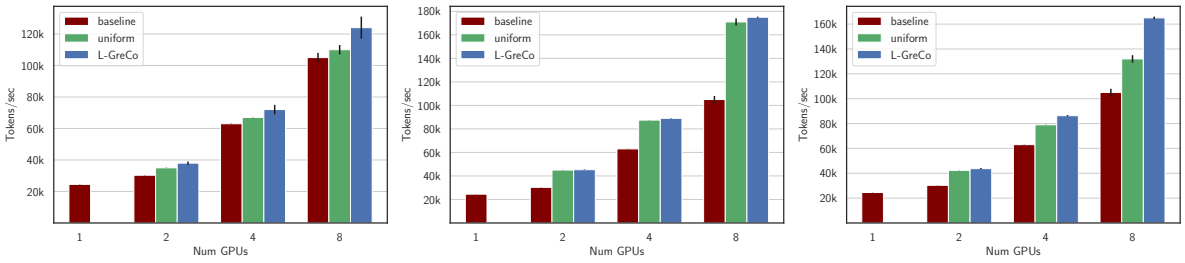
For instance, for Transformer-LM, we obtain **up to 5x higher compression** relative to the uniform baseline, with negligible accuracy impact. At the same time, L-GreCo induces $> 1\%$ multiplicative loss on quantization and low-rank compression for the highly-sensitive Transformer-LM model.[1] This is because our default compression range is too aggressive in this case; this can be easily addressed by adjusting the range—we chose not to do it for consistency.

Further, we varied the uniform default parameters, specifically throttling the target PowerSGD rank for the Transformer-XL/WikiText-103 task. Figure 4.2 shows that `L-GreCo` provides a markedly better trade-off than uniform compression. We can see that L-GreCo provides better perplexity recovery while improving training speed.



(a) PowerSGD (b) QSGD (c) TopK

Figure 4.3: Throughput for Transformer-XL (TXL) on WikiText-103. Multi-node, each node has 4 RTX3090 GPUs.



(a) PowerSGD (b) QSGD (c) TopK

Figure 4.4: Throughput for Transformer-XL (TXL) on WikiText-103. Single node, 8 RTX3090 GPUs.

Speedup results. For end-to-end training speedup improvements, we compare against standard uncompressed training ResNets and Transformers. We consider *weak scaling*, i.e., increase the global batch size while increasing the node count. (Performance improvements are higher for strong scaling.) We begin by examining training throughput results for *multi-node* training of Transformer-XL in Figure 4.3, executed in the cloud environment. (See Appendix Figure B.5 for ResNet50 experiments.) This setting encounters a bandwidth bottleneck even at a lower node count, which is apparent given the poor performance of the uncompressed baseline. Tuned uniform compression partly removes this bottleneck: for instance, uniform PowerSGD/ResNet50 reaches 75% of ideal scaling on 4 nodes.

It is, therefore, surprising that automatic non-uniform compression can still provide significant improvements in this setting: relative to uniform compression, `L-GreCo` gives up to 2.4x speedup.

[1]Specifically, our loss is of at most 0.85 perplexity relative to the uncompressed baseline. For this model, however, even basic FP16 training loses more than 1 point of perplexity vs FP32.

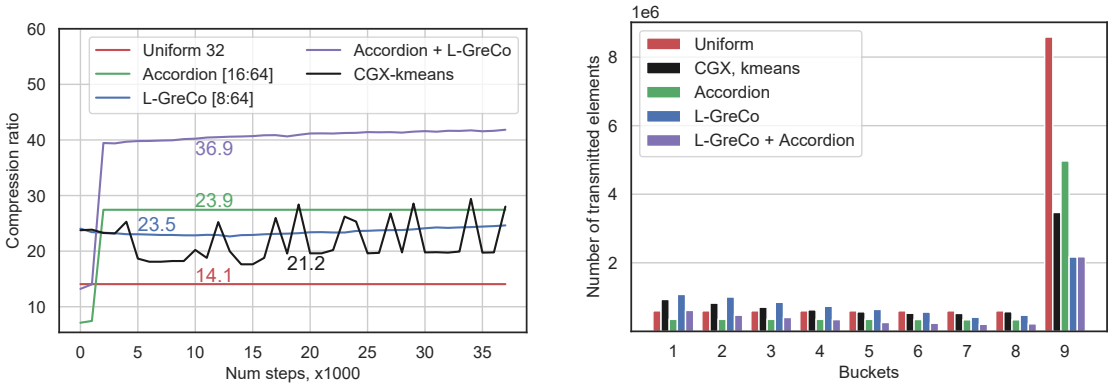
Table 4.4: Comparison of L-GreCo with other adaptive algorithms on ResNet-18/CIFAR-100, TopK compression.

Method	Parameters	Accuracy	Average Density (%)
Uniform	2%	71.8	2.00 (1×)
Uniform	0.1%	70.6	0.1 (20×)
Accordion	min = 0.1%, max = 2%	71.6	0.57 (3.5×)
Rethink-GS	$\lambda = 4.72 \times 10^{-3}$	71.4	0.35(5.7×)
L-GreCo	[0.1%, 10%]	71.7	0.30% (6.7×)

This suggests that non-uniform compression can be an effective strategy in this scenario, especially for layer-heterogeneous models such as Transformers.

We next examine results for single-node scaling from 1 to 8 GPUs, presented in Figure 4.4. For this model, using PowerSGD and TopK, L-GreCo leads to gains up to 25% *end-to-end* speedup compared to uniform, with negligible accuracy difference. For QSGD, the search space is very limited: uniform already uses 4 bits and provides very good scaling. Our adaptive method still provides 2% speedup compared to our well-tuned uniform compression and 50% speedup compared to non-compressed training, reaching $\geq 90\%$ of ideal scaling. (Appendix Figure B.4 presents ResNet50 results, which show lower improvements since training is weakly communication-bound in this setting.)

Overall, we note that L-GreCo provides statistically-significant performance improvements over static uniform compression (especially given heterogeneous models) when applied to all considered compression methods, with negligible impact on accuracy. We explore the compression overheads in Appendix B.5.



(a) Compression ratio over time. (b) Compression per layer group (bucket).

Figure 4.5: Adaptive compression using L-GreCo versus other methods, for PowerSGD compression on Transformer-XL. The left plot shows the dynamics of the compression ratio during training, marking the *average compression ratio*. The right plot presents the transmitted number of elements per bucket averaged over time. Buckets are in communication order.

4.6.3 Comparison with other adaptive methods

So far, we have used uniform compression as our baseline. We now compare L-GreCo with prior works on adaptively choosing compression parameters. We consider Accordion [AWL+21] and CGX(Chapter

Here, since we aim to maximize speedup without dropping accuracy, we tune the compression parameters for each algorithm, the chosen compression method, and training task (without changing the training hyperparameters, e.g., learning rate, weight decay, etc.) so that we get the best timing results with the final model accuracy within the 1% MLPerf standard. The best range of parameters for L-GreCo turns is ranks $[8, 64]$ (see Table 4.5) with default rank 32.

The adaptive compression of CGX is based on kmeans and maps layers into a 2-dimensional space (layer size vs. L_2-error). The algorithm clusters layers into several groups and assigns predefined compression parameters to the layers in the groups. We have implemented this logic with PowerSGD compression. We used rank 32 as default, and the best (in terms of compression) range was from 8 to 64, using 6 layer clusters. The results are shown in the Table 4.5. In short, L-GreCo improves upon the kmeans approach by up to 33%. In Figure 4.5b, we observe that L-GreCo picks parameters such that the largest and the last bucket are compressed the most, whereas the kmeans algorithm chooses worse compression parameters for those layers in some iterations.

Rethink-GS[SDMA+21] suggests a sparcification method, which is technically adaptive hard-threshold sparsification changes the number of transmitted elements based on the gradient distribution. We have run the L-GreCo training of ResNet18/CIFAR-100. We used 1% density as a default parameter for L-GreCo and the search range was $[0.1\%, 10\%]$. For Rethink-GS we used parameter $\lambda = 4.72 \times 10^{-3}$. The setup from [SDMA+21] is presented in Appendix B in Table B.1. The comparison of compression ratios and accuracies for this setup is shown in the Table 4.4.

As a result, L-GreCo improves upon Rethink-GS by 17% in terms of compression ratio ($6.7\times$ vs $5.7\times$) while also improving the final accuracy - 71.7% vs 71.4% (the numbers differ from the ones we show in Table. 4.2 as here we used the setup from [SDMA+21]). We note that our framework did not require *any hyperparameter tuning at all* for this experiment, whereas Rethink-GS requires careful tuning of the hard-threshold λ parameter.

Table 4.5: Comparison of L-GreCo with other adaptive algorithms on Transformer-XL using PowerSGD.

Adaptive algorithm	Param. Range	Ratio	Single node Tokens/s	Multi-node Tokens/s
Uniform	32	14.1	110k	72k
L-GreCo	8 - 64	23.5	**144k**	150k
Accordion	16, 64	23.9	114k	107k
CGX, kmeans	8 - 64	21.6	124k	112k
L-GreCo + Accordion	8 - 128	36.9	138k	**176k**

Global TopK Comparison. The optimization problem of minimizing gradient error magnitude given the desired compression ratio - can be alternatively solved by taking a global TopK in the case of gradient sparsification, which in this case minimizes total error. However, this method has several drawbacks. First, the global TopK requires careful fine-tuning and hyperparameter search in order to converge when low densities are used. L-GreCo, in turn, does not try to minimize the global compression error – it tries to match it to the compression error of the uniform *layer-wise* compression that recovers accuracy. Also, L-GreCo aims to maximize compression, meaning that each layer has a contribution to gradient synchronization. This may not be the case for global TopK: at high sparsities, some layers could have gradient zero for several steps, impacting model quality. The second disadvantage of global TopK is the actual speedup. As we discussed in Section 4.5, in modern data-parallel frameworks,

gradients are synchronized in parallel with computation for the sake of efficiency, hiding communication costs behind computation. However, in global TopK, one has to wait until all layer gradients are produced, then perform compression and communication. In this case, the loss of performance due to non-overlapped communication may be higher than the improvements due to compression. To confirm this, we have implemented the algorithm using `torch.distributed` hooks and ran the RN18/CIFAR100 training. Even on this relatively small model, global TopK is 10% slower than `L-GreCo` when applied with a similar global density, in this case, 0.25%.

Accordion Comparison. `Accordion` adapts compression by detecting critical regimes during training. It accepts two possible compression modes (corresponding to low and high compression) and has a threshold error parameter η. It collects the gradients and periodically decides the parameter to use based on gradient information for each layer. We have implemented Accordion using the `torch.distributed` hook, used for PowerSGD. For the parameter η, we chose the value of 0.5 suggested by the authors and tried to hand-tune the best pair of low and high compression parameters for each model, with which training converges to an accuracy that is within MLPerf bounds. We ran this algorithm on Transformer-XL/Wikitext-103, and found that the best pair of parameters (in terms of training time without losing accuracy) are high compression rank 16 and low compression rank 64.

In Figure 4.5, one can observe the dynamics of the average compression ratio over the training of `Accordion` relative to `L-GreCo`. We notice that `Accordion` chooses a low compression rank for almost all layers during the first period of training and a high compression rank for the rest of the training time, leading to completely bimodal uniform compression. This suggests that Accordion may not really exploit the heterogeneous nature of DNN models. Therefore, the optimizations of `Accordion` and `L-GreCo`, respectively, could be seen as *orthogonal*: `Accordion` focuses on varying the amount of *average compression* during training, whereas `L-GreCo` finds an optimal way of reaching this average level by setting layer-wise targets.

L-GreCo+ Accordion. With this in mind, we combined these two algorithms: We first executed `Accordion` to get the suggested parameters for each layer and used these parameters as the default set of parameters in `L-GreCo`, used to define the maximal error of the DP algorithm (see line 2 in Algorithm 4.1). Thus, `Accordion` determines the model sensitivity to gradient compression at different points in training, while `L-GreCo` finds the best mapping of compression parameters per layer. In Figure 4.5, we see that the resulting combination (`L-GreCo` with range $[8, 128]$ and Accordion with high=16, low=64) provides better compression ratios, without accuracy drop.

We also compare the performance of the two algorithms in isolation (see Table 4.5). We observe that, despite the fact that the theoretical compression ratio suggested by `Accordion` is essentially the same as that of `L-GreCo`, the `Accordion` throughput is less by around 30%. This is explained by the fact that `L-GreCo` compressed the last transmitted layers (buckets) to higher levels, leading to significantly-improved total transmission time. Specifically, in Figure 4.5b, we observe that `L-GreCo` transmits twice fewer elements in the last bucket relative to `Accordion`. Moreover, combining `L-GreCo` with `Accordion` improves the compression ratio by 50%, and training time by up to 66% compared to `Accordion`.

Overall, `L-GreCo` improved practical compression relative to prior techniques. Of note, the highest compression ratio is achieved by the hybrid `Accordion` + `L-GreCo` method, which leverages layer-wise insights in terms of both sensitivity and training dynamics.

4.6.4 Evaluating the loss-based accuracy metric

As discussed in Section 4.4, a key advantage of L-GreCo is that it can use any metric for measuring layer sensitivity to compression. To illustrate this, we investigated a loss-based metric, in which we collect model loss differences between uncompressed training and training with the gradient compression for certain layers, while other layers stay intact and use the loss difference as the sensitivity metric.

To compare the loss-based and error-based approaches, we evaluate the correlation coefficients of the metrics these two approaches provide. We observe (see Appendix B.6 for details) that the metric values from the two approaches have a high correlation and lead to very similar layer-wise compression parameters. Hence, since collecting loss-based metrics requires additional offline training runs, our usage of an error-based metric is justified.

4.6.5 Optimizing specifically for time

Considering that transmitted layer groups/communication buckets have different impacts on training performance, one may notice that compression may not always result in speedup. With this in mind, we have modified `L-GreCo` to *explicitly optimize the expected communication time*, rather than the compression ratio. See the last part of Section 4.5 for a detailed description of this algorithm variant.

We have run the resulting time-aware variant of `L-GreCo` with PowerSGD on Transformer-XL/WikiText-103 training. The linear model built on the timing data we collected (5000 samples - sets of compression ratios per bucket) has a score close to 1, meaning that we managed to predict the communication time using communicated bucket sizes almost perfectly. (The linear regression model was trained by running training for 50 steps with 5 steps of warmup for each set of compression parameters.)

We find that the per-bucket coefficients from the linear model are close to each other (see Appendix B.7for precise numbers). This means that each bucket's impact on communication time is proportional to the bucket size. Also, we noticed that the parameters we obtain with the modified algorithm are close to the parameters from the original `L-GreCo` algorithm. Given that, we figure that in the case of Transformer-XL the original `L-GreCo` is close to optimal in terms of timing as well.

CHAPTER 5

Quantized Sharded Data-Parallel Training with Convergence Guarantees

5.1 Preface

The impressive recent progress of Deep Learning in tasks such as natural language processing and computer vision has been accompanied by massive increases in parameter counts. For instance, large language models (LLMs) from Transformer family, such as GPT [RNS+18], OPT [ZRG+22] and BLOOM [LSW+22] easily count billions of trainable parameters. Training such models can easily exceed the memory capacity of a single computational unit, such as a GPU. This has necessitated new methods to leverage data-parallelism. Among these methods, fully-sharded data parallel (FSDP) training has gained significant popularity, but similar to the original Data Parallel it continues to face scalability bottlenecks. A possible solution - lossy communication compression is challenging as most communication involves the model's weights and direct compression can affect convergence and reduce accuracy. In this chapter, which follows [MVGA23], we introduce QSDP, a variant of FSDP that supports both gradient and weight quantization with theoretical guarantees. It is simple to implement and has essentially no overheads. We prove that a natural modification of SGD achieves convergence even when maintaining only quantized weights.

5.2 Introduction

As a consequence, standard distribution strategies such as *data-parallel training* [Bot10], which require each node to be able to keep all parameters in memory, are no longer directly applicable. Several novel distribution strategies have been proposed to mitigate this challenge, such as *model-parallel training* [SPP+19, RSR+20], *pipeline-parallel training* [HCB+19, HNP+18] and *model sharding* [RRA+21, RRRH20a, RRRH20b, Fai21].

We consider the communication costs of distribution strategies for massive models, and focus on *Fully-Sharded Data-Parallel (FSDP)* distributed training, which is among the most popular and user-friendly approaches to mitigate per-node memory limitations. FSDP is supported natively by Pytorch [PGM+19], Facebook `fairscale` [Fai21], and Microsoft DeepSpeed [RRA+21], where it is known as ZeRO-3.

The main idea behind FSDP is that both the training data *and the model parameters* are partitioned among the P nodes. That is, only a $1/P$ partition of the parameters of each layer is stored at a node. Then, both for the forward and for the backward pass, nodes proceed synchronously layer-by-layer, gathering full weights for the current layer, via *all-to-all communication*, before executing its forward or backward operation. After this operation is complete, nodes can discard the current layer's received weights partitions, and move to the next layer. (Please see Figure 5.1 for an illustration, and Section 5.5.1 for a detailed description.)

The key advantage of this pattern is that it reduces memory usage linearly in P. Thus, it enables running models with billions of parameters on small or medium-sized clusters [Fai21, Mos22b]. At the same time, FSDP faces challenges in terms of *communication efficiency*: since every forward and backward pass relies on all-to-all weight exchanges, FSDP can put massive pressure on the network bandwidth, which becomes a bottleneck.

As we will show, all-to-all communication leads to significant communication bottlenecks when training LLMs on multi-node clusters. Two key challenges to removing this communication bottleneck are that 1) a majority of FSDP's communication is *layer weights*: quantizing them naively loses theoretical convergence, and can easily lead to practical divergence; 2) the FSDP setting poses stringent compute and memory constraints, restricting the set of approaches.

Contribution. We propose the first communication-efficient variant of FSDP, called QSDP, which provides both convergence guarantees, and strong practical performance. QSDP is inspired by a new analysis of SGD convergence with *full quantization of transmitted model state*. That is, we show that a simple modified variant of SGD can allow both weights and gradients to be quantized during training, without additional per-node memory or costly local computation. We find the fact that this is possible with convergence guarantees surprising, since nodes only observe *biased estimators* of the gradients, taken over quantized weights, without any error-correction [KRSJ19]. From the practical perspective, our approach is fast and easy to implement, and completely removes the communication bottlenecks of FSDP, while recovering accuracy for billion-parameter GPT models.

At a high level, the QSDP algorithm simply performs weight and gradient quantization before the corresponding FSDP all-to-all communication steps. While gradient compression can be performed using standard unbiased compressors, e.g. [AGL$^+$17], weight compression is performed using a carefully designed unbiased estimator. Our key contribution is in the analysis: we model the training process as a new instance of sparse recovery [BD08, Fou12], in which 1) the projection step is performed via quantization and not sparsification, and 2) the gradient step is itself quantized. This connection allows us to prove, under analytic assumptions, that QSDP converges towards a minimizer of the loss over the set of lattice points corresponding to the quantization being employed. We believe this is the first instance of such an analysis. The analysis could potentially be improved by further leveraging structure, as it currently requires increasing the sparsity by a quadratic factor involving the "condition number" of the underlying objective, which, although inherent to this type of approach [AS22], may in theory be quite large. However, despite this shortcoming, it paves the way towards a principled algorithm, with excellent practical behavior.

We complement our analysis with an efficient implementation of QSDP in Pytorch [PGM$^+$19], which we validate by training LLMs from the GPT family [RNS$^+$18, ZRG$^+$22] between 125M and 1.3B parameters, on a multi-node multi-GPU environment on Amazon EC2. Our experiments first show that communication bottlenecks can significantly impact standard FSDP in this standard practical setting, and that QSDP essentially removes such bottlenecks

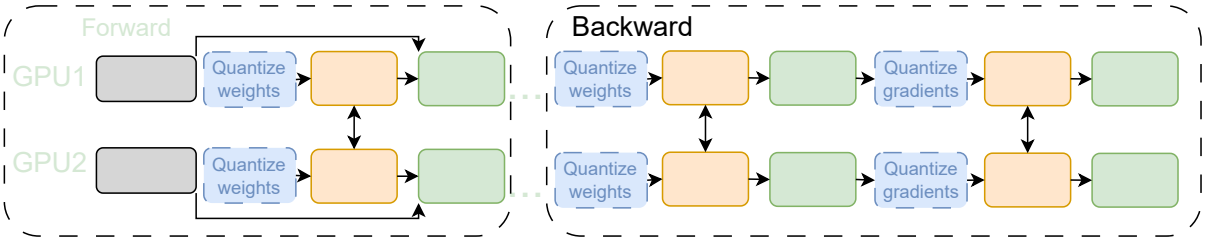


Figure 5.1: Scheme of (Quantized) Fully Sharded Data Parallel algorithm. During forward pass we collect the missing partitions of layer's weights, compute its activations and discard the partitions. At backward pass, we collect the weights again, compute the gradients, synchronize the gradients corresponding to our partition.

without impact on accuracy. For example, QSDP can train GPT-1.3B to essentially the same perplexity up to 2.2x faster than standard FSDP on a 10Gbps network. In addition, we also introduce a "learned" adaptive weight quantization approach which can further reduce bit-width, without significant accuracy impact.

5.3 Related Work

Over the past decade, there has been a massive amount of work on communication-efficient variants of Data-Parallel SGD, e.g. [SFD+14, DJMVE16, AGL+17, VKJ19, TYL+19, WSL+18]. (Please see [BNH19] for a survey.) The vast majority of this work focuses on gradient compression, the main communication cost of SGD, and is thus mostly orthogonal to our work. The massive scale of recent deep models, e.g. [CND+22, BMR+20] has led to significant work on novel distribution strategies [RRA+21, RRRH20a, RRRH20b, Fai21] adapted to the requirements of these models, among which FSDP is a standard approach, e.g. [CND+22]. While there is recent work on further optimizing FSDP, e.g. [JMNC22, MWJ+22], we are the first to investigate and address its communication costs. Our results are part of a broader line of work using different techniques to make the training of massive models amenable to standard infrastructure, e.g. [WYR+22, YHD+22, BBD+22].

Quantized weight exchange during training has been investigated independently in the context of decentralized distributed learning. Tang et al.[TZG+18] present a scheme which supports quantized weight exchange by having each node extrapolate each of its neighbors' model values; yet, this would require unrealistic $\Theta(Pd)$ extra memory in our case. Similarly, other work in this vein [KSJ19, NSD+21, LDS20] either requires additional storage, or would not fit the FSDP algorithm structure. Both our analysis approach and our algorithms' guarantees are different relative to this line of work.

Recently, there has been a surge of interest in *post-training* quantization approaches for large language models, which reduce the deployment costs of already trained models [YAZ+22, DLBZ22, FAHA22, XLS+22]. Our work is complementary, in the sense that we show that quantized weights and gradient representations can be applied *during training*, without accuracy loss, leading to training speedup. On the other hand, these post-training approaches would be too expensive to be executed for compression at training time.

A parallel line of work aims to perform *fully-quantized* training of DNNs [BHHS18, ZGY+20]. One general finding from this line of work is that integrating weight and gradient quantization *into training* is extremely challenging, even when using 8bit precision, from both accuracy and performance perspectives. Specifically, this line of work investigates model modifications via

e.g. parameter tuning and specialized normalization layers, in order to recover accuracy. By contrast, we preserve model structure and do not modify hyper-parameter values, although we only quantize the transmitted state.

5.4 Background and Motivation

5.4.1 Data-Parallel Training

In this classic SGD distribution pattern [Bot10], each node (e.g., GPU) holds a copy of the model, and the data is partitioned among the nodes. Each training step samples a subset of the data called a batch, performs a forward pass over the batch to obtain model predictions, and then performs a backward pass to obtain gradient updates. Finally, nodes communicate their local gradient updates in all-to-all fashion to keep the model in sync.

5.4.2 Gradient Compression

Transmitting gradients is the key communication cost of Data-Parallel SGD, and there has been a tremendous amount of work on addressing the resulting bandwidth bottleneck [SFD+14, DJMVE16, Str15]. (As this area is extremely vast, we refer to [BNH19, LZ+20] for a full overview.) Of these, gradient quantization is a particularly-popular technique, which has the advantage that variants of it can be implemented without additional memory cost. A simple example is the QSGD technique [AGL+17], which is essentially a codebook compression method which maps each gradient value to a point on a uniform grid, via randomized rounding. For this, values are first scaled to the range $[-1, 1]$, and then each scaled coordinate v is mapped to one of the endpoints of its quantization interval $v \in [q_i, q_{i+1}]$ via the following rule:

$$q(v) = \begin{cases} q_i, \text{with probability } \frac{v-q_i}{q_{i+1}-q_i}, \\ q_{i+1}, \text{otherwise.} \end{cases}$$

It is easy to see that this gradient estimator is unbiased with respect to the stochastic quantization, and that its variance can be bounded by the norm of the original gradient. We will revisit this scheme in Sections 5.5.3 and 5.6.

5.4.3 Fully-Sharded Data-Parallel Training

As the name suggests, FSDP starts from the Data-Parallel (DP) approach. The main observation is that nodes do not necessarily need to store the full set of parameters at every stage of training, in particular during the backward pass. Specifically, we use the scarce GPU memory to represent only those layers which are in the forward-backward "working set" at a given moment of time.

Initially, model parameters are partitioned, so that each of the P workers is assigned a distinct $1/P$ fraction of each layer's weights. At each optimization step (see Figure 5.1, ignoring the dashed quantization operations), before the forward pass on a layer, each worker collects the missing partitions from other workers, computes the output activations, discards the received partitions and proceeds to the next layer. For the backward pass, workers again collect all layer weights, compute the gradients, synchronize them, discard the layer weights, and proceed to the next layer. Technically, each optimization step consists of two AllGather

collective operations for weights, and one Reduce-Scatter to sync gradients (full pseudocode in Appendix C.1).

One can easily check that the above approach implements the standard SGD iteration one-to-one, relative to a sequential execution. If we denote by $\boldsymbol{y}_t$ the model's parameter vector used at iteration t, and by $\boldsymbol{g}(\boldsymbol{y}_t)$ the average of the nodes' stochastic gradients at step t, taken at $\boldsymbol{y}_t$, then, for learning rate η, we can model the iteration as

$$\boldsymbol{y}_{t+1} = \boldsymbol{y}_t - \eta \boldsymbol{g}(\boldsymbol{y}_t). \tag{5.1}$$

FSDP with Compression. The natural way to reduce the cost of weight and gradient transmission in the above scheme would be to simply quantize them before transmission. (Please see the full Figure 5.1.) To examine the impact of adding compression on the above SGD iteration, let us consider abstract quantization operators Q^w applied to the weights, and Q^g applied to the gradients. (We will specify these quantization functions precisely in Section 5.5.3, and the exact implementation in Section 5.6.) For iteration $t \geq 0$, let $\boldsymbol{v}_t$ be a "virtual" view of the model weights at the beginning of iteration t, obtained by aggregating all the weights, across all the weight partitions, *in full precision*.

First, notice that, if we apply Q^w before all transmissions, then the algorithm will only observe the *quantized* version of $\boldsymbol{v}_t$, which we denote by $Q^w(\boldsymbol{v}_t)$. Then, we can re-write one iteration of the algorithm as

$$\boldsymbol{v}_{t+1} = Q^w(\boldsymbol{v}_t) - \eta Q^g(\boldsymbol{g}(Q^w(\boldsymbol{v}_t))).$$

This formulation inspires the notation $\boldsymbol{x}_t = Q^w(\boldsymbol{v}_t)$, as the algorithm only "sees" the quantized version of the full-precision weights. Then, we get the following iteration:

$$\boldsymbol{x}_{t+1} = Q^w(\boldsymbol{x}_t - \eta Q^g(\boldsymbol{g}(\boldsymbol{x}_t))), \tag{5.2}$$

which would correspond to an abstractly-quantized version of FSDP. This iteration is the starting point for our analysis in the next section.

5.5 SGD with Quantized Weights and Provable Convergence

The cornerstone of our method consists of a stochastic gradient method that provably converges to a good *quantized* iterate, under reasonable analytic assumptions. One main novelty is that it converges despite the fact that the domain is *non-convex*. At a very high level, it is similar to the iterative hard thresholding (IHT) method, which achieves provable guarantees despite the fact that it seeks a good iterate in the set of vectors of bounded sparsity [BD08]. Throughout this section, we abstract away specifics of the system architecture, since they are not relevant to our analysis. We explain their relationship to the actual implementation in Section 5.6.

5.5.1 Background and Assumptions

The main challenge we face is to obtain quantized solutions to optimization problems that seek to minimize a function $f : \mathbb{R}^n \to \mathbb{R}$:

$$\min_{\boldsymbol{x}\in\mathcal{G}} f(\boldsymbol{x}) \,, \tag{5.3}$$

where the domain $\mathcal{G}$ is a lattice that allows for an efficient communication of its elements. We restrict our attention to shifts of the lattice $\delta\mathbb{Z}^n$ along the direction of the all-ones vector. Formally, $\mathcal{G} = \{\delta\mathbb{Z}^n + r1 : r \in [-\delta/2, \delta/2)\}$.

Overview. Even in the case where f is convex, the non-convex structure of $\mathcal{G}$ makes it incredibly difficult to obtain a good minimizer to f without suffering a large loss. In fact, problems of this form are generally NP-hard. However, we show that when f is reasonably well-conditioned we can obtain strong convergence guarantees. The idea consists of alternating stochastic gradient descent steps with applications of a quantization operator Q^w which projects the new iterate back onto a certain subset of $\mathcal{G}$. Letting $\boldsymbol{g}(\boldsymbol{x}_t)$ be a stochastic gradient, and δ a parameter which determines the coarseness of the quantization grid that we project onto, our update at step $t+1$ has the following form:

$$\boldsymbol{x}_{t+1} = Q^w_\delta\left(\boldsymbol{x}_t - \eta \boldsymbol{g}(\boldsymbol{x}_t)\right) . \tag{5.4}$$

This formulation covers the practical case where the stochastic gradient $\boldsymbol{g}(\boldsymbol{x}_t)$ corresponds to a mini-batch stochastic gradient. Indeed, as in practice f takes the form $f(\boldsymbol{x}) = \frac{1}{Pm}\sum_{i=1}^{P}\sum_{j=1}^{m} f(\boldsymbol{x}; \boldsymbol{y}_j)$, where $S = \{\boldsymbol{y}_1, \ldots, \boldsymbol{y}_m\}$ are data samples, and $f_i(\boldsymbol{x})$ are loss functions at individual nodes, the stochastic gradients obtained via backpropagation takes the form $\frac{1}{|B|}\sum_{j\in B}\nabla f_i(\boldsymbol{x}; \boldsymbol{y}_j)$, where i is a random node, and B is a sampled mini-batch.

Quantization by Random Shift. For weight quantization, we consider the following unbiased stochastic quantization method. To quantize a vector, we first sample a single fixed random scalar r, then shift all the coordinates of the vector by r. For vector encoding, it rounds each coordinate to the nearest neighbor on the quantization grid, and sends the lattice coordinates of the resulting vector, together with the scalar r. For decoding, it takes the lattice point, and undoes the shift on the quantized coordinates. The notable difference between this and more standard quantization methods, e.g.[AGL$^+$17], is that quantization is *dependent* across coordinates. In exchange for losing independence, it allows us to provide stronger guarantees in the context of weight quantization. We define it formally:

Definition 1 (quantization by random shift)**.** *Let $\delta > 0$ be a scalar defining the coarseness of the quantization grid. Let a scalar $r \in [-\delta/2, \delta/2)$, and let the deterministic operator $Q^w_{r,\delta} : \mathbb{R} \to \mathbb{R}$ which rounds to the nearest element in $\delta\mathbb{Z} + r$:*

$$Q^w_{r,\delta}(x) = \delta \cdot \left\lfloor \frac{x - r}{\delta} \right\rceil + r \,.$$

Define the randomized quantization operator $Q^w_\delta : \mathbb{R} \to \mathbb{R}$ via $Q^w_\delta(x) = Q^w_{r,\delta}(x)$, for a random $r \sim$ Unif$([-\delta/2, \delta/2))$. We apply Q^w_δ to vectors, with the meaning that it is dependently applied to each coordinate for a single random shift r.

We use this quantization method to show that, for a well-conditioned loss function, and appropriate grid parameters, the iteration (5.4) converges, under reasonable analytical assumptions,

to a set of weights that are comparable in quality to the best possible quantized weights from a slightly coarser grid. We note that to further reduce communication costs, our method also supports gradient quantization in addition to weight quantization, provided that gradients are quantized using an (arbitrary) unbiased estimator.

Analytical Assumptions. Formally, our analysis uses the following assumptions on f.

1. Unbiased gradient estimators with variance σ: $\mathbb{E}\left[\boldsymbol{g}\left(\boldsymbol{x}\right)|\boldsymbol{x}\right] = \nabla f\left(\boldsymbol{x}\right)$.
2. For $\beta > 0$, the β-smoothness condition: for all $\boldsymbol{x}, \boldsymbol{\Delta}$,
$$f\left(\boldsymbol{x}+\boldsymbol{\Delta}\right) \leq f\left(\boldsymbol{x}\right) + \left\langle \nabla f\left(\boldsymbol{x}\right), \boldsymbol{\Delta}\right\rangle + \frac{\beta}{2}\left\|\boldsymbol{\Delta}\right\|_2^2 .$$
3. For $\alpha > 0$, the Polyak-Łojasiewicz (α-PL) condition:
$$\frac{1}{2}\left\|\nabla f\left(\boldsymbol{x}\right)\right\|_2^2 \geq \alpha\left(f\left(\boldsymbol{x}\right) - f^*\right) ,$$
where $f^* = \min_{\boldsymbol{x}} f\left(\boldsymbol{x}\right)$.

The first two assumptions are standard in stochastic optimization (see e.g. [LSB+19]). The Polyak-Łojasiewicz (PL) condition [KNS16] is common in non-convex optimization, and versions of it are essential in the analysis of DNN training [LZB20, AZLS19]. In words, it states that small gradient norm, i.e. approximate stationarity, implies closeness to optimum in function value.

5.5.2 Main Theoretical Results

We are now ready to state our main analytical result.

Theorem 2. *Let $\alpha, \beta, \delta_\star, \varepsilon > 0$ and $\sigma \geq 0$ be real parameters, and let $\eta = \min\left\{\frac{3}{10}\frac{\varepsilon\alpha}{\sigma^2}, 1\right\}$. Let $f : \mathbb{R}^n \to \mathbb{R}$ be a β-smooth and α-PL function, with access to a stochastic gradient $\boldsymbol{g}\left(\boldsymbol{x}\right)$, i.e. $\mathbb{E}\left[\boldsymbol{g}\left(\boldsymbol{x}\right)|\boldsymbol{x}\right] = \nabla f\left(\boldsymbol{x}\right)$ with bounded variance $\mathbb{E}\left\|\boldsymbol{g}\left(\boldsymbol{x}\right) - \nabla f\left(\boldsymbol{x}\right)\right\|_2^2 \leq \sigma^2$. For each $r \in [-\delta_\star/2, \delta_\star/2)$, let $\boldsymbol{x}^\star_{r,\delta_\star}$ be any minimizer of f over $\delta_\star\mathbb{Z}^n + r1$. Let $\delta = \frac{\eta}{\lceil 16(\beta/\alpha)^2\rceil}\cdot\delta_\star$. Consider the iteration:*
$$\boldsymbol{x}_{t+1} = Q_\delta^w\left(\boldsymbol{x}_t - \frac{\eta}{\beta}\boldsymbol{g}\left(\boldsymbol{x}_t\right)\right) .$$
In $T = \frac{10}{\eta}\cdot\frac{\beta}{\alpha}\ln\frac{f(\boldsymbol{x}_0)-\mathbb{E}f(\boldsymbol{x}^\star_{r,\delta_\star})}{\varepsilon}$ iterations we obtain a point $\boldsymbol{x}_T$ satisfying $\mathbb{E}f\left(\boldsymbol{x}_T\right) - \mathbb{E}f(\boldsymbol{x}^\star_{r,\delta_\star}) \leq \varepsilon$.

Discussion. To understand the convergence of this method, let us establish as a benchmark the expected value $\mathbb{E}f(\boldsymbol{x}^\star_{r,\delta_\star})$ of the best iterate on the lattice $\delta_\star\mathbb{Z}^n + r1$, where the expectation is taken over the shift r. Our method finds a point in a slightly finer grid $\boldsymbol{x}_T \in \delta\mathbb{Z}^n + r'1$, such that in expectation over the randomness in the algorithm, the value of the function is at most ϵ larger than our benchmark. The sacrifice we have to make in exchange for this surprisingly strong convergence is an increase in resolution for the iterates we maintain, which is dependent, among others, on the condition number of f. It appears that without further assumptions, the dependence on condition number can be quite large – unfortunately, even the best-known analyses of "standard" iterative hard thresholding (the sparsity correspondent

to our projection approach) have quadratic dependency in the condition number, as shown in [AS22], so bounds of this type are inherent even for simple examples, like ℓ_2 regression.

Since our method works with stochastic gradients, we can additionally quantize gradients to further reduce communication. In fact, any quantization method that compresses gradients to an unbiased estimator with low variance can be directly plugged into Theorem 2.

We state in the following corollary a generic bound for quantized gradients, which highlights the trade-off between variance and communication for the quantization method.

Corollary 3 (Gradient Quantization)**.** *Let $\alpha, \beta, \delta_\star, \varepsilon, b > 0$ and $\sigma, \sigma_\nabla \geq 0$ be real parameters. Let $f : \mathbb{R}^n \to \mathbb{R}$ be a β-smooth and α-PL function, with access to a stochastic gradient estimator $\boldsymbol{g}(\boldsymbol{x})$, i.e. $\mathbb{E}[\boldsymbol{g}(\boldsymbol{x})|\boldsymbol{x}] = \nabla f(\boldsymbol{x})$ with bounded variance $\mathbb{E}\|\boldsymbol{g}(\boldsymbol{x}) - \nabla f(\boldsymbol{x})\|_2^2 \leq \sigma^2$. Let $Q^g : \mathbb{R}^n \to \mathbb{R}$ be a gradient quantizer which for any stochastic gradient $\boldsymbol{g}(\boldsymbol{x})$ encountered during the execution of the algorithm, ensures:*

1. *unbiased estimator: $\mathbb{E}[Q^g(\boldsymbol{g}(\boldsymbol{x}))|\boldsymbol{g}(\boldsymbol{x})] = \boldsymbol{g}(\boldsymbol{x})$,*
2. *variance: $\mathbb{E}\left[\|Q^g(\boldsymbol{g}(\boldsymbol{x})) - \boldsymbol{g}(\boldsymbol{x})\|_2^2 |\boldsymbol{g}(\boldsymbol{x})\right] \leq \sigma_\nabla^2$,*
3. *requires b bits to communicate $Q^g(\boldsymbol{g}(\boldsymbol{x}))$.*

For each $r \in [-\delta_\star/2, \delta_\star/2)$, let $\boldsymbol{x}^\star_{r,\delta_\star}$ be any minimizer of f over $\delta_\star\mathbb{Z}^n + r \cdot 1$. Let $\eta = \min\left\{\frac{3}{10}\frac{\varepsilon\alpha}{\sigma^2+\sigma_\nabla^2}, 1\right\}$, $\delta = \frac{\eta}{\lceil 16(\beta/\alpha)^2\rceil} \cdot \delta_\star$, and consider the iteration:

$$\boldsymbol{x}_{t+1} = Q_\delta^w\left(\boldsymbol{x}_t - \frac{\eta}{\beta}Q^g\left(\boldsymbol{g}(\boldsymbol{x}_t)\right)\right).$$

In $T = \frac{10}{\eta}\cdot\frac{\beta}{\alpha}\ln\frac{f(\boldsymbol{x}_0)-\mathbb{E}f\left(\boldsymbol{x}^\star_{r,\delta_\star}\right)}{\varepsilon}$ iterations we obtain a point $\boldsymbol{x}_T$ satisfying $\mathbb{E}f(\boldsymbol{x}_T) - \mathbb{E}f(\boldsymbol{x}^\star_{r,\delta_\star}) \leq \varepsilon$. Furthermore, the entire algorithm requires $O\left(b \cdot \frac{\sigma^2+\sigma_\nabla^2}{\varepsilon\alpha}\frac{\beta}{\alpha}\ln\frac{f(\boldsymbol{x}_0)-\mathbb{E}f(\boldsymbol{x}^\star_{r,\delta_\star})}{\varepsilon}\right)$ bits to communicate the quantized gradients.

We notice that since b and σ_∇ are inversely associated, we can establish a trade-off between the number of iterations and the total communication. As a matter of fact, this trade-off kicks in only at the point where the variance of the quantized gradient estimator becomes as large as that of the stochastic gradient, as the number of iterations scales linearly with $\sigma^2 + \sigma_\nabla^2$. For example, given a resolution parameter $\delta_\nabla > 0$, a simple gradient quantization scheme such as the one employed by QSGD, quantizes gradient entries to $\delta_\nabla\mathbb{Z}^n$, while guaranteeing $\sigma_\nabla^2 \leq \delta_\nabla G_{\ell_1}$, where $G_{\ell_1} \geq \|\boldsymbol{g}(\boldsymbol{x})\|_1$, and $b = O(G_{\ell_1}/\delta_\nabla \cdot (\ln n + \ln G_{\ell_1}))$ bits required for communication. See a more detailed discussion in Section C.5.3. By varying δ_∇ we distinguish between the extreme cases, corresponding to the scenarios where the quantized gradients are dense and sparse, respectively. While the total communication does not improve by varying δ_∇, by doing so we are able to reduce the communication performed in each iteration, in practice [AGL+17].

Dense gradients: setting $\delta_\nabla = \sigma^2/G_{\ell_1}$, we obtain exactly the same number of iterations as in the basic case without quantized gradients, but the communication per iteration is reduced to $O(G_{\ell_1}^2/\sigma^2 \cdot (\ln n + \ln G_{\ell_1}))$.

Sparse gradients: setting $\delta_\nabla = G_{\ell_1}$, the number of iterations scales with $\max\left\{\sigma^2, G_{\ell_1}^2\right\}$ rather than σ^2, but the pre-step communication is reduced to $O(\ln n + \ln G_{\ell_1})$ bits.

5.5.3 Analysis Overview

Let us briefly explain the intuition behind our theoretical analyses. We view our iteration as a version of projected gradient descent, where iterates are projected onto the non-convex domain of quantized vectors. In general, when the domain is convex, projections do not hurt convergence. But in our setting the distance to the optimal solution can increase and drastically affect the loss. However, we can show a trade-off between how much this distance increases and the ratio between the target and optimal resolution $\delta/\delta_\star$.

To understand this better, consider a point $\boldsymbol{x}'$ obtained by taking a step $\boldsymbol{x}' = Q_\delta^w(\boldsymbol{x} - \frac{1}{\beta}\nabla f(\boldsymbol{x}))$. Using smoothness, we can verify that this significantly decreases the loss, provided that the quantization operator does not perturb its input by too much in ℓ_2 norm. Formally, using Lemma 7 we see that

$$f(\boldsymbol{x}') \leq f(\boldsymbol{x}) - \frac{1}{2\beta}\|\nabla f(\boldsymbol{x})\|_2^2 + \frac{\beta}{2}\left\|Q_\delta^w\left(\boldsymbol{x} - \frac{1}{\beta}\nabla f(\boldsymbol{x})\right) - \left(\boldsymbol{x} - \frac{1}{\beta}\nabla f(\boldsymbol{x})\right)\right\|_2^2. \tag{5.5}$$

Since compared to a vanilla gradient method, this suffers a reduction in the progress made in a single iteration, we can force this to be significantly smaller so as not to undo more than a fraction of the progress we would ideally make. To do so, we notice that we can change the last term in (5.5) to the current error in function value, and we can make this dependence be arbitrarily small by using a finer resolution δ for our quantization grid. This is captured by the following crucial lemma, which we prove in Section C.5.4.

Lemma 4. *Let $\delta_\star > \delta > 0$, such that $\delta_\star/\delta \in \mathbb{Z}$. Let $\boldsymbol{x} \in \mathbb{R}^n$, and for all $r \in [-\delta_\star/2, \delta_\star/2)$, let an arbitrary $\boldsymbol{x}^\star_{r,\delta_\star} \in \delta_\star\mathbb{Z}^n + r1$. Then*

$$\mathbb{E}\left[\|Q_\delta^w(\boldsymbol{x}) - \boldsymbol{x}\|_2^2\right] \leq \frac{\delta}{\delta_\star}\mathbb{E}_r\left[\left\|\boldsymbol{x}^\star_{r,\delta_\star} - \boldsymbol{x}\right\|_2^2\right].$$

Using Lemma 4, together with the α-PL condition, we can change the extra error term to

$$\frac{\beta}{2}\cdot\frac{\delta}{\delta_\star}\mathbb{E}_r\left[\frac{2}{\alpha}\left(f\left(\boldsymbol{x} - \frac{1}{\beta}\nabla f(\boldsymbol{x})\right) - f(\boldsymbol{x}^\star_{r,\delta_\star})\right)\right],$$

where $\boldsymbol{x}^\star_{r,\delta_\star}$ are picked to be the best minimizers in $\delta_\star\mathbb{Z}^n + r1$. To simplify the exposition and highlight the main ideas, let us assume that $\mathbb{E}_r[f(\boldsymbol{x}^\star_{r,\delta_\star})] = f(\boldsymbol{x}^\star)$. Since by the α-PL condition we know that the gradient norm is large compared to the error in function value, we conclude that

$$\begin{aligned} f(\boldsymbol{x}') - f(\boldsymbol{x}^\star) &\leq f(\boldsymbol{x}) - f(\boldsymbol{x}^\star) - \frac{\alpha}{\beta}\left(f(\boldsymbol{x}) - f(\boldsymbol{x}^\star)\right) \\ &\quad + \frac{\beta}{\alpha}\cdot\frac{\delta}{\delta_\star}\left(f\left(\boldsymbol{x} - \frac{1}{\beta}\nabla f(\boldsymbol{x})\right) - f(\boldsymbol{x}^\star)\right) \\ &\leq \left(f(\boldsymbol{x}) - f(\boldsymbol{x}^\star)\right)\left(1 - \frac{\alpha}{\beta} + \frac{\beta}{\alpha}\frac{\delta}{\delta_\star}\right). \end{aligned}$$

This shows that by setting the $\delta \leq \delta_\star \cdot (\alpha/\beta)^2/2$, in each iteration the error contracts by a $1 - \Theta(\alpha/\beta)$ factor, which allows us to conclude that this algorithm converges linearly to a minimizer. We provide full proofs in Section C.5.

5.6 QSDP Implementation

5.6.1 Overview

We implemented a practical version of the QSDP algorithm described in the previous section, supporting both weight and gradient quantization, in Pytorch [PGM+19] starting from the PyTorch FSDP support. Our implementation uses the CGX framework(Section 3) as a communication backend, to which we added support for quantized AllGather and Reduce-Scatter collectives.

In the original FSDP implementation, layers are packed into groups: weights and gradients of layers in the same group are concatenated before communication. In QSDP, we compress layers separately, filtering out normalization layers and biases, which are communicated in full precision. This filtering is implemented at the level of the CGX communication backend. The quantized AllGather and Reduce-Scatter operations are implemented by leveraging peer-to-peer NVIDIA NCCL primitives. For multi-node (inter-server) communication, we used hierarchical versions of the algorithms to reduce the size of inter-node transmissions.

One important optimization regards the granularity at which quantization is performed. Specifically, applying quantization over large tensors suffers from scaling issues, which results in accuracy degradation. To address this, we perform compression independently into equally-sized "buckets" of fixed size, and compress each bucket independently. This approach sacrifices compression by a negligible amount (as we need to transmit min-max scaling meta-information for each bucket), but helps avoid loss in terms of model quality.

Bucketing (or grouping) on the weights is known to be necessary for good accuracy when quantizing *pre-trained* LLMs [DZ23]. It is also justified theoretically (Theorem 2), as it both reduces compression variance and allows us to explore solutions over finer-grained lattices. We observed experimentally that bucket size 1024 provides a good balance between compression and accuracy and use it as a universal hyper-parameter. In the context of this optimization, we observed that the impact of stochasticity in the quantization becomes minimal.

5.6.2 Learned Weight Quantization

We now describe an additional (optional) optimization, which allows us to further reduce practical bits-width, at little to no accuracy loss. The motivating observation behind this optimization is that the quantization schemes we use for weights and gradients assume *uniform* locations of the quantization levels. Yet, this uniform grid does not take the distribution of values into account. The idea of adapting the locations of quantization levels to the data distribution has already been studied [ZLK+17, FTM+20]. However, existing dynamic-programming approaches ZipML [ZLK+17] have high computational cost (quadratic in the number of data points); thus, we use a fast version of gradient-descent-based optimization over the quantization levels [FTM+20].

The goal of the distribution-aware quantizer in Algorithm 5.1 is to select new locations for a fixed number of quantization points and weight values, so as to minimize the error introduced by quantization. The algorithm runs iteratively across all values, finds the locations of quantization points for each value, and updates the *quantization points* using the gradient descent update rule. We run this heuristic periodically after a warmup period, separately for the weights and gradients of each layer. We save the derived locations of the quantization levels and use them for quantization until the next re-computation.

Algorithm 5.1: Gradient-based Optimization of the Levels

Input: values V, initial levels Q_0, learning rate α.
Output: optimized quantization levels Q.
Normalize values V bucket-wise.
for each value v_i from V **do**
    $q_i = \text{find\_closest}(v, Q_i)$ // Quantize using current level
    $q_i = q_i - \alpha(q_i - v_i)$ // Update chosen quantization level
end for

5.7 Experimental Validation

5.7.1 Experimental setup

Infrastructure. We evaluate QSDP for training GPT-scale LLMs using multiple cloud-grade Amazon EC2 `p3dn.24xlarge` machines, with 8 V100 SXM2 GPUs each. Each GPU has 32GB of memory. The inter-GPU interconnect is provisioned by NVLinks with 200Gbps, while the inter-server bandwidth is 100 Gbps.

Environment and Tasks. We use the official NGC PyTorch 22.05-py3 Docker image with PyTorch 1.12, CUDA 11.6.2, NCCL 2.12, and the MosaicML Composer library (version 0.12), as well as a fork of the CGX communication library (Chapter 3). All experiments were run with MosaicML Large Language Models implementation [Mos22a]. The benchmarks run the pre-training of different versions of GPT-family models [RNS+18, BMR+20], varying the sizes of the models, on the C4 dataset [RSR+20]. Specifically, we examine the accuracy of GPT models with 125M, 350M, and 1.3B parameters. For benchmarks, we use 4 servers with 8 GPUs each. See Appendix C.1 for training details.

Baselines. As a baseline, we use training with default parameters, which is already highly optimized by MosaicML [Mos22a]. We note that directly using INT8 quantization, without bucketing, resulted in very poor accuracy, and therefore we do not use it as a baseline. In terms of communication, the baseline transmits weights in full (FP32) precision and gradients in half (FP16) precision. In QSDP experiments, we *do not* modify any hyperparameters. We convert gradients to full precision before quantization. For all timing experiments, the reported numbers are averaged over 50 training steps after a warm-up of 10 iterations. Our main accuracy measure is *perplexity*, which is known to be a very stringent accuracy measure in this setting, and correlates extremely well with zero-shot performance [DLBZ22].

Accuracy Recovery. We first examine the effect of quantization on model quality, i.e. final model perplexity, in the end-to-end experiments. The default bit-width for weights and gradients quantization is 8 bits, using 1024 bucket size, which we illustrate as W8G8. We communicate normalization layers and biases in full precision. We emphasize that straightforward round-to-nearest or stochastic quantization *does not converge* to reasonable final perplexity in this setup: Naive quantization without bucketing loses more than 2 units of perplexity on GPT-125M, a model on which W8G8 with 1024 bucket size *improves* perplexity.

The accuracy results are presented in Table 5.1. The QSDP final perplexity is almost identical to that of regular training, and QSDP can even *slightly improve* the baseline accuracy. We stress that we did not perform any parameter tuning: quantization parameters are the same across all layers.

End-to-end Speedup. For end-to-end training speedup improvements, we use multi-node

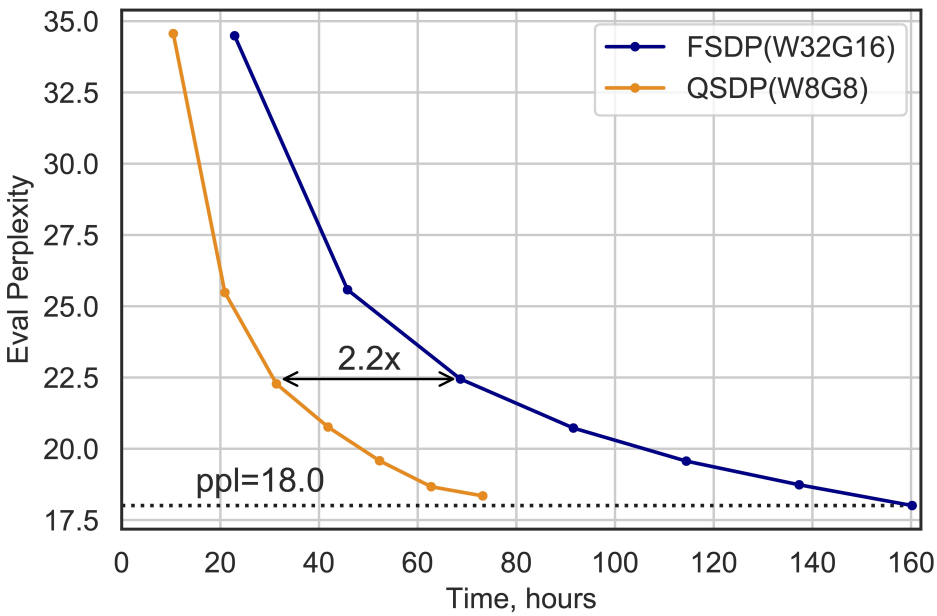


Figure 5.2: Perplexity vs time for standard FSDP (FP32 weights and FP16 gradients) and QSDP (both weights and gradients quantized to 8 bits) for the 1.3B model in the 10Gbps bandwidth setup.

GPT pretraining under standard hyperparameters. We examine speedup for different inter-node bandwidths: 10 Gbits, 50 Gbits, and 100 Gbits. For that, we artificially reduce input-output bandwidth on each node, using the UNIX `tc` tool [TC01]. The results are presented in Figure 5.3. First, please notice that standard FSDP training has a non-trivial bandwidth bottleneck even at 100Gbps bandwidth as we increase model size and that this bandwidth bottleneck can dominate training time on the lower 10Gbps bandwidth. Second, the running time of QSDP is *essentially constant* across all three scenarios, showing that it has essentially removed the bandwidth bottleneck. More precisely, QSDP outperforms the baseline by up to 15% in the 100Gbps scenario (a non-trivial reduction of 12 hours of training time or 1.5k$ of cloud costs[1]), and by 2.25x in the 10Gbps scenario.

Learned quantization. We examined the performance of learned quantization for the small 125M parameters model. We ran the optimization algorithm after 400, 1900, and 3800 training steps, and noticed that optimizing the locations of quantization levels has no effect for bit-widths higher than 6 bits, but leads to noticeable improvements for lower bit-widths. Please see Table 5.3. Learned weight quantization allows us to improve the final model performance for different weight and gradient quantization parameter pairs, reaching perplexities that are close to the baseline. Specifically, using learned quantization results in reaching the highest compression ratio for weights and gradient in training (i.e. 5 and 4 bits respectively) without substantial accuracy impact. We expand upon these experiments in Appendix C.3.

[1]price of 12 hours training on 4 AWS p3dn.24xlarge instances

Table 5.1: Perplexities recoveries for different models end-to-end training using QSDP. Weights and gradients quantized to 8 bits, uniform quantization.

	125M	350M	1.3B
Baseline	35.81	23.94	18.00
QSDP	35.58	23.95	18.34

Table 5.2: Final perplexities of training 125m GPT-2 model with combinations of weights and gradients low-bits uniform (not learned) quantization.

Weights bits \ Gradients bits	6	5	4
6	35.74	36.08	35.84
5	36.01	35.94	36.36
4	37.11	37.38	37.61

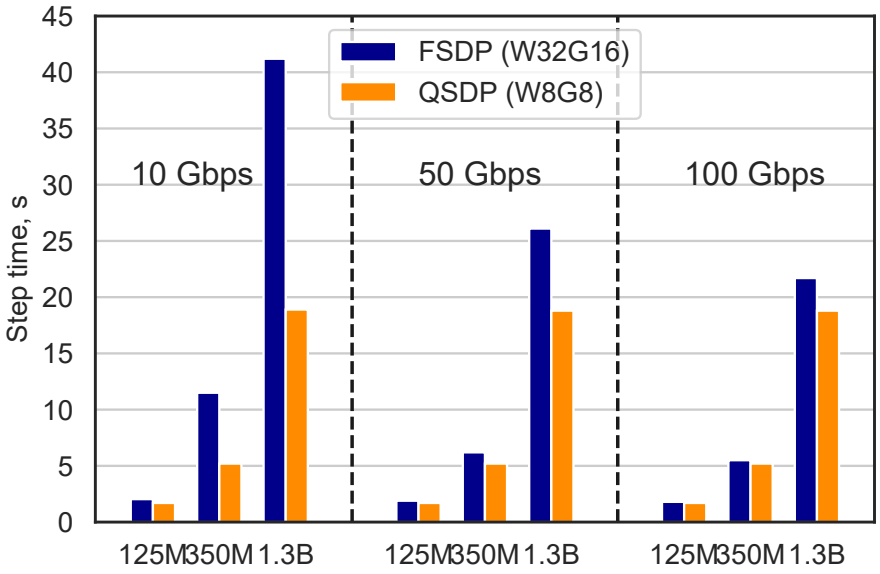


Figure 5.3: Training step time for different models at various inter-node bandwidth with and without QSDP enabled. The fact that QSDP step time is constant across considered bandwidths means that QSDP successfully tackles bandwidth bottlenecks.

Table 5.3: Final perplexities of low-bits quantization of 125m GPT-2 model using the learned quantization levels. Learned quantization in the W6G4 configuration provides lower perplexity than the baseline.

	baseline	w6g4	w5g4	w4g4	w4g32
Uniform	35.81	35.81	36.34	37.61	37.11
Learned		**35.75**	**36.01**	**36.94**	**36.55**

CHAPTER 6

Discussion and Future work

Given that the success of deep learning heavily relies on the scale of the models and the datasets used for training, efficient training is more crucial than ever. Distributed training accelerates the process; however, communication can become a scalability bottleneck in these systems. This thesis explores solutions to this communication bottleneck through lossy compression.

Thesis summary

As the first step, we designed and implemented the system, called CGX, that supports efficient gradient compression. We demonstrated in Chapter 3 that the expensive approach of hardware overprovisioning can be replaced by algorithmic and system design. Our framework offers efficient software support for compressed communication in machine learning applications, applicable to both multi-GPU single-node training and larger-scale multi-node training. CGX utilizes a newly developed communication stack for ML frameworks at the system level and enables seamless, parameter-free integration with popular frameworks, allowing end-users to avoid modifying training recipes or significant portions of training code.

Next, we have explored the concept of adapting compression parameters across different model layers and throughout the training process. Chapter 4 introduces a general framework for dynamically adjusting the degree of compression across a model's layers during training. The algorithm determines the optimal compression parameters so that 1) the total L2 compression error aligns with a target known to maintain the accuracy, and 2) the overall compressed size is minimized for this target. In other words, the approach enhances overall compression and achieves significant speedups without compromising accuracy. The framework, named L-GreCo, employs an adaptive algorithm that automatically selects the optimal compression parameters for each model layer, ensuring the best compression ratio while meeting an error constraint. Overall, we presented a new approach to enhance existing gradient compression methods with virtually no cost in terms of time or accuracy loss.

Moreover, we have examined a variation of Data Parallel training known as fully-sharded data parallel training. We introduce a variant called QSDP, which supports both gradient and weight quantization. QSDP is straightforward to implement and incurs virtually no overhead. The approach is validated by training GPT-family models with up to 1.3 billion parameters on a multi-node cluster. Experiments demonstrate that QSDP maintains model accuracy while eliminating the communication bottlenecks of FSDP, achieving end-to-end speedups of up to 2.2x. Additionally, we have provided a theoretical investigation that resulted in a novel analysis

demonstrating that Stochastic Gradient Descent (SGD) can converge with robust guarantees, even with quantized iterates, provided a well-quantized solution is available. Our findings show that communication compression can be a useful technique within new distribution strategies inspired by large-scale training. We believe we are the first to demonstrate both convergence guarantees and strong practical performance for straightforward weight compression schemes applied during SGD-based training.

Future directions

The research prospects in distributed deep learning discussed in this thesis focus on a systems-algorithmic co-design approach. As deep learning models increase in size and complexity, new challenges related to scalability, performance, and efficiency will require more improvements. In what follows, we highlight potential research avenues for further exploration of these emerging trends.

Integration with other distributed training approaches. In CGX (Chapter 3) and other works we only focus on Data Parallel based training and its extensions. Future work may extend our results to model-parallel or hybrid synchronization setups, e.g. [ZGQ+20, LMXG21]. The Tensor Parallel approach that becomes more popular with the development of Large Language models might require additional systems and algorithmic approaches.

Further research of layer-wise compression. In Chapter 4, we examine the impact of heterogeneous compression on training performance and accuracy, primarily using L_2 compression error as a key metric. However, this approach does not account for inter-layer interactions during training, as we treat the layers independently. Future work could investigate alternative metrics that consider the impact of layers on one another.

Additionally, in Chapter 4, we focus on experiments using individual compression techniques, selecting compression parameters for either quantization, sparsification, or decomposition. However, `L-GreCo` inherently supports hybrid approaches, allowing for different layers to be quantized, decomposed, or sparsified. Implementing this idea would require additional system support due to the diverse backends of the compression techniques, but it could potentially lead to further improvements in speed and accuracy.

Another possible research direction is to further explore the combination of critical training regimes [ARS18] with our work [MAFA24] on Language Models in various types of training, such as pre-training or fine-tuning.

Compression in Fully Sharded Data Parallel. An intriguing extension of QSDP, as discussed in Chapter 5, would be to explore other compression techniques, such as weight pruning or gradient low-rank decomposition. Additionally, integrating `L-GreCo` with QSDP could reveal interesting patterns in layers, not just for gradient but for weight compression as well. This integration might further identify layers that are particularly sensitive to compression, providing valuable insights for post-training weight compression. Furthermore, such an approach could help optimize the balance between compression efficiency and model performance, potentially leading to more refined and effective compression strategies across different layers. This would be particularly useful for enhancing the performance of large-scale deep learning models while maintaining their accuracy.

Bibliography

[AAB+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous systems, 2015.

[ABC+16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 265–283, USA, 2016. USENIX Association.

[AGL+17] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. QSGD: Communication-efficient sgd via gradient quantization and encoding. *Advances in Neural Information Processing Systems*, 30:1709–1720, 2017.

[ANL+20] Yonathan Aflalo, Asaf Noy, Ming Lin, Itamar Friedman, and Lihi Zelnik. Knapsack pruning with inner distillation. *arXiv preprint arXiv:2002.08258*, 2020.

[ARS18] Alessandro Achille, Matteo Rovere, and Stefano Soatto. Critical learning periods in deep networks. In *International Conference on Learning Representations*, 2018.

[AS22] Kyriakos Axiotis and Maxim Sviridenko. Iterative hard thresholding with adaptive regularization: Sparser solutions without sacrificing runtime. In *International Conference on Machine Learning*, pages 1175–1197. PMLR, 2022.

[AWL+21] Saurabh Agarwal, Hongyi Wang, Kangwook Lee, Shivaram Venkataraman, and Dimitris Papailiopoulos. Adaptive gradient communication via critical learning regime identification. In *Proceedings of Machine Learning and Systems*, volume 3, pages 55–80, 2021.

[AWVP] Saurabh Agarwal, Hongyi Wang, Shivaram Venkataraman, and Dimitris Papailiopoulos. In *Proceedings of Machine Learning and Systems*.

[AZLS19] Zeyuan Allen-Zhu, Yuanzhi Li, and Zhao Song. A convergence theory for deep learning via over-parameterization. pages 242–252. PMLR, 2019.

[BBD+22] Alexander Borzunov, Dmitry Baranchuk, Tim Dettmers, Max Ryabinin, Younes Belkada, Artem Chumachenko, Pavel Samygin, and Colin Raffel. Petals: Collaborative inference and fine-tuning of large models. *arXiv preprint arXiv:2209.01188*, 2022.

[BD08] Thomas Blumensath and Mike E Davies. Iterative thresholding for sparse approximations. *Journal of Fourier analysis and Applications*, 14(5-6):629–654, 2008.

[BHHS18] Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. Scalable methods for 8-bit training of neural networks. *Advances in neural information processing systems*, 31, 2018.

[BLZ+21] Youhui Bai, Cheng Li, Quan Zhou, Jun Yi, Ping Gong, Feng Yan, Ruichuan Chen, and Yinlong Xu. Gradient compression supercharged high-performance data parallel dnn training. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 359–375, New York, NY, USA, 2021. Association for Computing Machinery.

[BMR+20] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[BNH19] Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)*, 52(4):1–43, 2019.

[Bot10] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[CCB+18] Chia Yu Chen, Jungwook Choi, Daniel Brand, Ankur Agrawal, Wei Zhang, and Kailash Gopalakrishnan. Adacomp: Adaptive residual gradient compression for data-parallel distributed training. In *32nd AAAI Conference on Artificial Intelligence, AAAI 2018*, pages 2827–2835, 2018.

[CKF11] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS workshop*, 2011.

[CND+22] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

[CSAK14] Trishul M Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, volume 14, pages 571–582, 2014.

[CYRW20] Mengqiang Chen, Zijie Yan, Jiangtao Ren, and Weigang Wu. Standard deviation based adaptive gradient compression for distributed deep learning. In *Proceedings of 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 529–538, 2020.

[DBA+20] Aritra Dutta, El Houcine Bergou, Ahmed M Abdelmoniem, Chen-Yu Ho, Atal Narayan Sahu, Marco Canini, and Panos Kalnis. On the discrepancy between the theoretical analysis and practical implementations of compressed communication for distributed deep learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 3817–3824, 2020.

[DBK+20] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[DCLT18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. 2018.

[DCM+12] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. *Advances in neural information processing systems*, 25, 2012.

[DDS+09] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. IEEE, 2009.

[DG17] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[DJMVE16] Nikoli Dryden, Sam Ade Jacobs, Tim Moon, and Brian Van Essen. Communication quantization for data-parallel training of deep neural networks. In *Proceedings of the Workshop on Machine Learning in High Performance Computing Environments*, pages 1–8. IEEE Press, 2016.

[DLBZ22] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. LLM.int8(): 8-bit matrix multiplication for transformers at scale. *arXiv preprint arXiv:2208.07339*, 2022.

[DYY+19] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. 2019.

[DZ23] Tim Dettmers and Luke Zettlemoyer. The case for 4-bit precision: k-bit inference scaling laws. In *International Conference on Machine Learning*, pages 7750–7774. PMLR, 2023.

[FA22] Elias Frantar and Dan Alistarh. SPDY: Accurate pruning with speedup guarantees. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 6726–6743. PMLR, 17–23 Jul 2022.

[FAHA22] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

[Fai21] FairScale. Fairscale: A general purpose modular pytorch library for high performance and large scale training. `https://github.com/facebookresearch/fairscale`, 2021.

[FHS+21] Jiawei Fei, Chen-Yu Ho, Atal N. Sahu, Marco Canini, and Amedeo Sapio. Efficient sparse collective communication and its application to accelerate distributed deep learning. In *Proceedings of the 35th ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 676–691, 2021.

[Fou12] Simon Foucart. Sparse recovery algorithms: sufficient conditions in terms of restricted isometry constants. In *Approximation Theory XIII: San Antonio 2010*, pages 65–77. Springer, 2012.

[FTM+20] Fartash Faghri, Iman Tabrizian, Ilia Markov, Dan Alistarh, Daniel M Roy, and Ali Ramezani-Kebrya. Adaptive gradient quantization for data-parallel sgd. *Advances in neural information processing systems*, 33:3174–3185, 2020.

[GDG+17] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[Gen21] Genesis. Genesis gpu cloud offering, 2021.

[GJY+21] Shaoduo Gan, Jiawei Jiang, Binhang Yuan, Ce Zhang, Xiangru Lian, Rui Wang, Jianbin Chang, Chengjun Liu, Hongmei Shi, Shengzhuo Zhang, Xianghong Li, Tengxu Sun, Sen Yang, and Ji Liu. Bagua: Scaling up distributed learning with system relaxations. *Proc. VLDB Endow.*, 15(4):804–813, dec 2021.

[GLD+17] Nitin A. Gawande, Joshua B. Landwehr, Jeff A. Daily, Nathan R. Tallent, Abhinav Vishnu, and Darren J. Kerbyson. Scaling deep learning workloads: Nvidia dgx-1/pascal and intel knights landing. pages 399–408, 2017.

[GLW+20] Jinrong Guo, Wantao Liu, Wang Wang, Jizhong Han, Ruixuan Li, Yijun Lu, and Songlin Hu. Accelerating distributed deep learning by adaptive gradient quantization. In *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1603–1607, 2020.

[Goo52] Irving John Good. Rational decisions. *Journal of the Royal Statistical Society: Series B (Methodological)*, 14(1):107–114, 1952.

[GTAZ18] Demjan Grubic, Leo K Tam, Dan Alistarh, and Ce Zhang. Synchronous multi-gpu deep learning with low-precision communication: An experimental study. In *Proceedings of the 21st International Conference on Extending Database Technology*, pages 145–156. OpenProceedings, 2018.

[Har21] William Harmon. Dual nvidia geforce rtx 3090 nvlink performance review, 2021.

[HCB+19] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[HNP+18] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377*, 2018.

[Hug22] Inc Huggingface. Huggingface transformers repository, 2022.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[JEP+21] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin vZídek, Anna Potapenko, et al. Highly accurate protein structure prediction with alphafold. *Nature*, 596(7873):583–589, 2021.

[JMNC22] Youhe Jiang, Xupeng Miao, Xiaonan Nie, and Bin Cui. Osdp: Optimal sharded data parallel for distributed deep learning. *arXiv preprint arXiv:2209.13258*, 2022.

[JWG+19] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. Priority-based parameter propagation for distributed dnn training. 2019.

[JZL+20] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 463–479. USENIX Association, November 2020.

[KB14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[KNS16] Hamed Karimi, Julie Nutini, and Mark Schmidt. Linear convergence of gradient and proximal-gradient methods under the Polyak-Łojasiewicz condition. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 795–811. Springer, 2016.

[Kri09] Alex Krizhevsky. Learning multiple layers of features from tiny images. pages 32–33, 2009.

[KRSJ19] Sai Praneeth Karimireddy, Quentin Rebjock, Sebastian Stich, and Martin Jaggi. Error feedback fixes signsgd and other gradient compression schemes. In *International Conference on Machine Learning*, pages 3252–3261. PMLR, 2019.

[KSJ19] Anastasia Koloskova, Sebastian Stich, and Martin Jaggi. Decentralized stochastic optimization and gossip algorithms with compressed communication. In *International Conference on Machine Learning*, pages 3478–3487. PMLR, 2019.

[LAK18] Hyeontaek Lim, David G Andersen, and Michael Kaminsky. 3lc: Lightweight and effective traffic compression for distributed machine learning. 2018.

[Lam21] LambdaLabs. Lambdalabs gpu cloud offering, 2021.

[LAP+14] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on operating systems design and implementation (OSDI 14)*, pages 583–598, 2014.

[LC17] Victor Lavrenko and W Bruce Croft. Relevance-based language models. In *ACM SIGIR Forum*, volume 51, pages 260–267. ACM New York, NY, USA, 2017.

[LDS20] Yucheng Lu and Christopher De Sa. Moniqua: Modulo quantized communication in decentralized sgd. In *International Conference on Machine Learning*, pages 6415–6425. PMLR, 2020.

[Lea21] LeaderGPU. Leadergpu cloud offering, 2021.

[LHM+17] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. 2017.

[LMXG21] Shijian Li, Oren Mangoubi, Lijie Xu, and Tian Guo. Sync-switch: Hybrid parameter synchronization for distributed deep learning. 2021.

[LNC+18] Liang Luo, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. Parameter hub: a rack-scale parameter server for distributed deep neural network training. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 41–54, 2018.

[LSB+19] Tao Lin, Sebastian U Stich, Luis Barba, Daniil Dmitriev, and Martin Jaggi. Dynamic model pruning with feedback. 2019.

[LSC+18] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Xu Liu, Nathan Tallent, and Kevin Barker. Tartan: Evaluating modern gpu interconnect via a multi-gpu benchmark suite. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 191–202, 2018.

[LSC+20] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):94–110, 2020.

[LSW+22] Hugo Laurençon, Lucile Saulnier, Thomas Wang, Christopher Akiki, Albert Villanova del Moral, Teven Le Scao, Leandro Von Werra, Chenghao Mou, Eduardo González Ponferrada, Huu Nguyen, et al. The bigscience roots corpus: A 1.6 tb composite multilingual dataset. In *Thirty-sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2022.

[LZ+20] Ji Liu, Ce Zhang, et al. Distributed learning systems with first-order methods. *Foundations and Trends® in Databases*, 9(1):1–100, 2020.

[LZB20] Chaoyue Liu, Libin Zhu, and Mikhail Belkin. Toward a theory of optimization for over-parameterized systems of non-linear equations: the lessons of deep learning. *arXiv preprint arXiv:2003.00307*, 2020.

[LZZL18] Xiangru Lian, Wei Zhang, Ce Zhang, and Ji Liu. Asynchronous decentralized parallel stochastic gradient descent. In *International Conference on Machine Learning*, pages 3043–3052. PMLR, 2018.

[Mac67] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of California Press, 1967.

[MAEAC21] Ahmed M Abdelmoniem, Ahmed Elzanaty, Mohamed-Slim Alouini, and Marco Canini. An efficient statistical-based gradient compression technique for distributed training systems. In *Proceedings of Machine Learning and Systems*, volume 3, pages 297–322, 2021.

[MAFA24] Ilia Markov, Kaveh Alim, Elias Frantar, and Dan Alistarh. L-greco: Layerwise-adaptive gradient compression for efficient data-parallel deep learning. *Proceedings of Machine Learning and Systems*, 6:312–324, 2024.

[Mos22a] Mosaicml examples. https://github.com/mosaicml/examples, 2022.

[Mos22b] MosaicML. Mosaic LLMs (part 2): Gpt-3 quality for <$500k, 2022.

[MRA22] Ilia Markov, Hamidreza Ramezanikebrya, and Dan Alistarh. Cgx: adaptive system support for communication-efficient deep learning. In *Proceedings of the 23rd ACM/IFIP International Middleware Conference*, pages 241–254, 2022.

[MRC+20] Peter Mattson, Vijay Janapa Reddi, Christine Cheng, Cody Coleman, Greg Diamos, David Kanter, Paulius Micikevicius, David Patterson, Guenther Schmuelling, Hanlin Tang, et al. Mlperf: An industry standard benchmark suite for machine learning performance. *IEEE Micro*, 40(2):8–16, 2020.

[MVGA23] Ilia Markov, Adrian Vladu, Qi Guo, and Dan Alistarh. Quantized distributed training of large models with convergence guarantees. In *International Conference on Machine Learning*, pages 24020–24044. PMLR, 2023.

[MWJ+22] Xupeng Miao, Yujie Wang, Youhe Jiang, Chunan Shi, Xiaonan Nie, Hailin Zhang, and Bin Cui. Galvatron: Efficient transformer training over multiple gpus using automatic parallelism. *arXiv preprint arXiv:2211.13878*, 2022.

[MXBS16] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

[NMC+21] Giorgi Nadiradze, Ilia Markov, Bapi Chatterjee, Vyacheslav Kungurtsev, and Dan Alistarh. Elastic consistency: A practical consistency model for distributed stochastic gradient descent. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 9037–9045, 2021.

[NSD+21] Giorgi Nadiradze, Amirmojtaba Sabour, Peter Davies, Shigang Li, and Dan Alistarh. Asynchronous decentralized sgd with quantized and local updates. *Advances in Neural Information Processing Systems*, 34:6829–6842, 2021.

[Nvi18] Nvidia. Nvidia collective communications library, 2018.

[Nvi20] Nvidia. Nvidia deep learning examples for tensor cores, 2020.

[nvi21] Nvidia ampere ga102 gpu architecture, 2021.

[NVI22] NVIDIA. Nvidia hopper architecture in-depth. `https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/`, 2022.

[OEB+19] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. fairseq: A fast, extensible toolkit for sequence modeling. In *Proceedings of NAACL-HLT 2019: Demonstrations*, 2019.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32:8026–8037, 2019.

[PVU+18] Niki Parmar, Ashish Vaswani, Jakob Uszkoreit, Lukasz Kaiser, Noam Shazeer, Alexander Ku, and Dustin Tran. Image transformer. In *International Conference on Machine Learning*, pages 4055–4064. PMLR, 2018.

[PZC+19] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 16–29, 2019.

[RAA+19] Cèdric Renggli, Saleh Ashkboos, Mehdi Aghagolzadeh, Dan Alistarh, and Torsten Hoefler. Sparcml: High-performance sparse communication for machine learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2019.

[Rab04] Rolf Rabenseifner. Optimization of collective reduction operations. In *Computational Science-ICCS 2004: 4th International Conference, Kraków, Poland, June 6-9, 2004, Proceedings, Part I 4*, pages 1–9. Springer, 2004.

[RKFM+21] Ali Ramezani-Kebrya, Fartash Faghri, Ilya Markov, Vitalii Aksenov, Dan Alistarh, and Daniel M Roy. NUQSGD: Provably communication-efficient data-parallel sgd via nonuniform quantization. *Journal of Machine Learning Research*, 22(114):1–43, 2021.

[RM51] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[RNS+18] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[RPG+21] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation. In *International conference on machine learning*, pages 8821–8831. Pmlr, 2021.

[RRA+21] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. {ZeRO-Offload}: Democratizing {Billion-Scale} model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 551–564, 2021.

[RRRH20a] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

[RRRH20b] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3505–3506, 2020.

[RSR+20] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J Liu, et al. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21(140):1–67, 2020.

[SB18] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in TensorFlow. 2018.

[SCH+21] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtárik. Scaling distributed machine learning with {In-Network} aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808, 2021.

[SCJ18] Sebastian U Stich, Jean-Baptiste Cordonnier, and Martin Jaggi. Sparsified sgd with memory. *Advances in neural information processing systems*, 31, 2018.

[SDMA+21] Atal Sahu, Aritra Dutta, Ahmed M. Abdelmoniem, Trambak Banerjee, Marco Canini, and Panos Kalnis. Rethinking gradient sparsification as total error minimization. In *Advances in Neural Information Processing Systems*, volume 34, pages 8133–8146. Curran Associates, Inc., 2021.

[SFD+14] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Fifteenth annual conference of the international speech communication association*, 2014.

[SPP+19] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[Sti18] Sebastian U Stich. Local sgd converges fast and communicates little. *arXiv preprint arXiv:1805.09767*, 2018.

[Str15] Nikko Strom. Scalable distributed dnn training using commodity gpu cloud computing. In *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

[SYM+22] Maying Shen, Hongxu Yin, Pavlo Molchanov, Lei Mao, Jianna Liu, and Jose M Alvarez. Structural pruning via latency-saliency knapsack. *arXiv preprint arXiv:2210.06659*, 2022.

[SZ14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. 2014.

[TC01] Unix TC. *tc(8) Linux User's Manual*, December 2001.

[TYL+19] Hanlin Tang, Chen Yu, Xiangru Lian, Tong Zhang, and Ji Liu. Doublesqueeze: Parallel stochastic gradient descent with double-pass error-compensated compression. In *International Conference on Machine Learning*, pages 6155–6165. PMLR, 2019.

[TZG+18] Hanlin Tang, Ce Zhang, Shaoduo Gan, Tong Zhang, and Ji Liu. Decentralization meets quantization. *arXiv preprint arXiv:1803.06443*, 2018.

[VKJ19] Thijs Vogels, Sai Praneeth Karimireddy, and Martin Jaggi. Powersgd: Practical low-rank gradient compression for distributed optimization. *Advances in Neural Information Processing Systems*, 32, 2019.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[WDD+24] Zhilin Wang, Yi Dong, Olivier Delalleau, Jiaqi Zeng, Gerald Shen, Daniel Egert, Jimmy J. Zhang, Makesh Narsimhan Sreedhar, and Oleksii Kuchaiev. Helpsteer2: Open-source dataset for training top-performing reward models, 2024.

[Wig19] Ross Wightman. Pytorch image models. `https://github.com/rwightman/pytorch-image-models`, 2019.

[WLCC20] Yu-Cheng Wu, Chih-Ting Liu, Bo-Ying Chen, and Shao-Yi Chien. Constraint-aware importance estimation for global filter pruning under multiple resource constraints. In *Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, 2020.

[WSL+18] Hongyi Wang, Scott Sievert, Shengchao Liu, Zachary Charles, Dimitris Papailiopoulos, and Stephen Wright. Atomo: Communication-efficient learning via atomic sparsification. *Advances in neural information processing systems*, 31, 2018.

[WXY+17] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Terngrad: Ternary gradients to reduce communication in distributed deep learning. *Advances in neural information processing systems*, 30, 2017.

[WYR+22] Jue Wang, Binhang Yuan, Luka Rimanic, Yongjun He, Tri Dao, Beidi Chen, Christopher Re, and Ce Zhang. Fine-tuning language models over slow networks using activation compression with guarantees. *arXiv preprint arXiv:2206.01299*, 2022.

[XHA+21] Hang Xu, Chen-Yu Ho, Ahmed M. Abdelmoniem, Aritra Dutta, El Houcine Bergou, Konstantinos Karatsenidis, Marco Canini, and Panos Kalnis. GRACE: A Compressed Communication Framework for Distributed Machine Learning. In *Proceedings of ICDCS'21*, Jul 2021.

[XLS+22] Guangxuan Xiao, Ji Lin, Mickael Seznec, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. *arXiv preprint arXiv:2211.10438*, 2022.

[YAZ+22] Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. ZeroQuant: Efficient and affordable post-training quantization for large-scale transformers. *arXiv preprint arXiv:2206.01861*, 2022.

[YHD+22] Binhang Yuan, Yongjun He, Jared Quincy Davis, Tianyi Zhang, Tri Dao, Beidi Chen, Percy Liang, Christopher Re, and Ce Zhang. Decentralized training of foundation models in heterogeneous environments. *arXiv preprint arXiv:2206.01288*, 2022.

[YLM+20] Xiaodong Yi, Ziyue Luo, Chen Meng, Mengdi Wang, Guoping Long, Chuan Wu, Jun Yang, and Wei Lin. Fast training of deep learning models over multiple gpus. In *Proceedings of the 21st International Middleware Conference*, page 105–118, 2020.

[YLR+19] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large batch optimization for deep learning: Training bert in 76 minutes. 2019.

[ZGQ+20] Qihua Zhou, Song Guo, Zhihao Qu, Peng Li, Li Li, Minyi Guo, and Kun Wang. Petrel: Heterogeneity-aware distributed deep learning via hybrid synchronization. *IEEE Transactions on Parallel and Distributed Systems*, 32(5):1030–1043, 2020.

[ZGY+20] Feng Zhu, Ruihao Gong, Fengwei Yu, Xianglong Liu, Yanfei Wang, Zhelong Li, Xiuqi Yang, and Junjie Yan. Towards unified int8 training for convolutional neural network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1969–1979, 2020.

[ZLK+17] Hantian Zhang, Jerry Li, Kaan Kara, Dan Alistarh, Ji Liu, and Ce Zhang. Zipml: Training linear models with end-to-end low precision, and a little bit of deep learning. In *International Conference on Machine Learning*, pages 4035–4043. PMLR, 2017.

[ZRG+22] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

APPENDIX A

Appendix for Chapter 3

A.1 Training Hyperparameters

Following the original recipes, ResNet50, VGG16, and the Vision Transformer (base model) were trained on ImageNet with total batch sizes 256, 256, 576 respectively. ViT was trained in mixed precision level 1 (activations at FP16, weights, and gradients in full precision). The Transformer-XL (base model) experiment was run on WikiText-103 dataset with batch size 256 and second level mixed precision (model, activations, and gradients cast FP16). The GPT-2 model was trained on WikiText-2, batch size 24, level 2 mixed precision. For question-answering we used BERT model on the SQUAD-v1 dataset with batch size 3 per GPU and FP32 training.

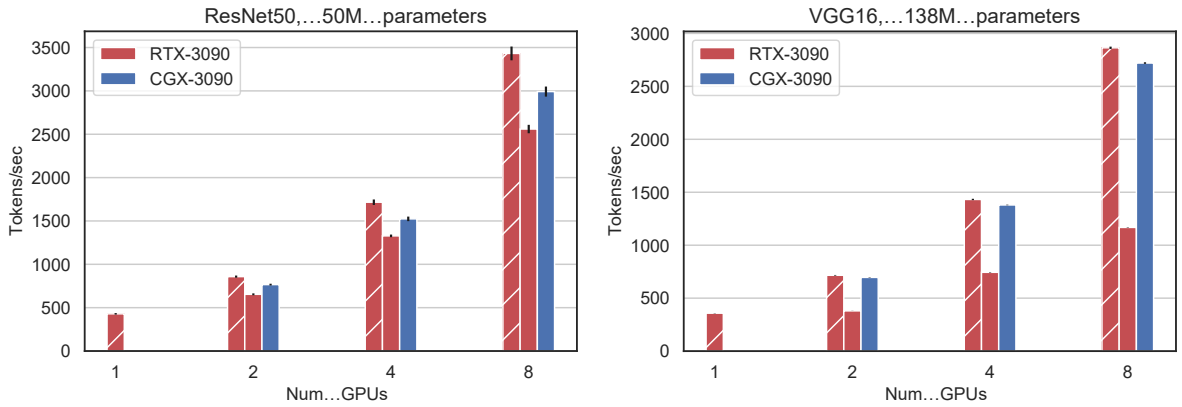


Figure A.1: Throughput for ResNet50, VGG16 /ImageNet with Tensorflow. Higher is better. Hatched bars represent ideal scaling.

A.2 Other frameworks

As an extension of Horovod, CGX also supports not only Pytorch but other Deep Learning frameworks, e.g. Tensorflow. Figure A.1 shows the results of CNN benchmarks based on Tensorflow. As we can see, CGX outperforms NCCL backend by up to 130%.

Listing A.1: Torch extension (torch_cgx) usage example

```
import torch
import torch_cgx

# specify backend with gradient compression
torch.distributed.init_process_group(backend='cgx')
...
#model definition
model = ...
# register model
layers = [(name, p.numel()) for name, p in model.named_parameters()]
torch_cgx.register_model(layers)
# Specify filtering. batch norm and bias modules will be reduced in full p
torch_cgx.exclude_layer("bn")
torch_cgx.exclude_layer("bias")
...
model = DDP(model, device_ids=[local_rank])
```

APPENDIX B

Appendix for Chapter 4

Table B.1: Hyperparameters for ResNet-18/CIFAR-100 from [SDMA+21]

Parameter	Value
Number of workers	8
Optimizer	SGD with momentum
Global batch size	1024
Momentum	0.9
Post warmup LR	1.6
LR decay	/10 at epoch 150 and 250
LR warmup	Linear for 5 epochs, starting from 0.1
Epochs	300
Weight decay	10^{-4}

B.1 Bucket prioritization

Considering the fact that communication buckets have different impacts on training performance, we modified the `L-GreCo` algorithm so that the last buckets in transmission order, corresponding to the earlier layers, were compressed more. This compensates for the compression error caused by picking lower compression parameters for the first buckets, i.e., the last layers. In practical terms, we have added linear priorities to the layers in Algorithm 4.1, multiplying the size of each layer by the index of the bucket the layer is communicated in. The profile of communicated elements per buckets is shown in the Figure B.1. We observe the linear shift of higher compression ratios towards the last buckets. However, bucket prioritization performs worse than original `L-GreCo`. It means that the effect of the first big buckets transmission is higher than the effect of better compression of the last buckets.

B.2 Low-rank error computation

As discussed in Section 4.5, one of the main steps of our algorithm is to compute the error matrix for different possible compression parameters. Table 4.1 suggests that this is the most

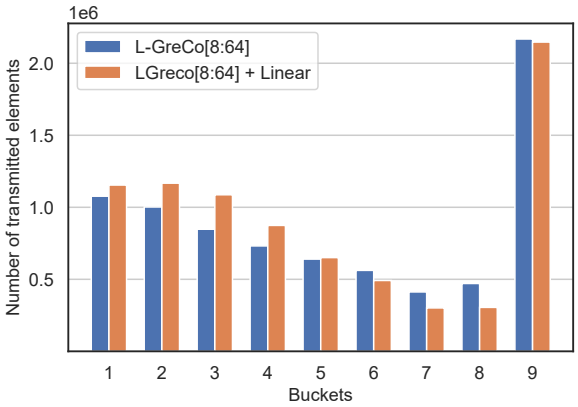


Figure B.1: Communicated elements per bucket for L-GreCo and L-GreCo with linear bucket prioritizing. Transformer-XL with PowerSGD.

time-consuming part of our framework. Specifically for the PowerSGD, we need to compute low-rank errors for a wide range of ranks. There are two possible solutions to do so.

B.2.1 Singular Value Decomposition

The first way to compute errors is to use singular value decomposition and compute singular values for a particular layer, and calculate the approximation error for rank $r < min(m, n)$ by calculating $e_r = \sqrt{\sum_{i=r+1}^{min(m,n)} \sigma_i^2}$, which can be done efficiently for all ranks. Specifically, it is sufficient to compute squared singular values once, and then compute all the errors by a single matrix product. Thus, the bottleneck is computing singular values requiring $O(mn \cdot min(m, n))$ time and $O(n^2 + mn)$ space.

B.2.2 Power Iteration Steps

The second approach is to calculate the approximation error for each rank separately by doing a few power steps (without the communication parts); as PowerSGD[VKJ19] claims, this approach converges to the SVD-suggested matrix. On the practical side, we have observed that applying only 5 power steps is enough to have a small error relative to the optimal low-rank approximation suggested by SVD. This approach needs $O(mnr)$ time and $O((m + n) \cdot r)$ space for calculating rank r approximation error and therefore $O(mnr_{max}^2)$ to compute errors for all $r \in [r_{min}, r_{max}]$.

B.2.3 The Best of Both Worlds

Comparing computational complexity and memory requirements of two methods suggests it is better to use the power method when the rank range is small, e.g., ResNet50 on ImageNet or ResNet18 on Cifar100, and to use the SVD method when the rank range is large, e.g., TransformerXL and TransformerLM on WIKITEXT-103.

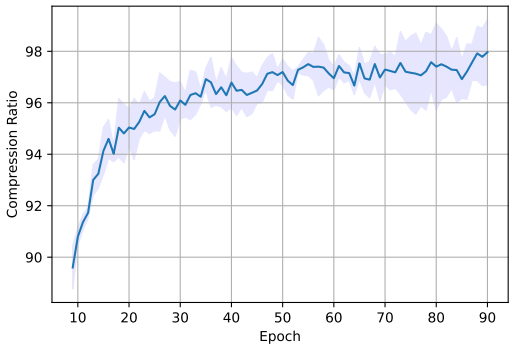


Figure B.2: Compression ratio of the scheme suggested by `L-GreCo` during the training. ResNet50 with PowerSGD.

Table B.2: Hyperparameters for ResNet-18/CIFAR-100

Parameter	Value
Number of workers	8
Optimizer	SGD with momentum
Global batch size	128
Momentum	0.9
Base LR	0.1
LR decay	/10 at epoch 150 and 250
Epochs	200
Weight decay	10^{-4}

Table B.3: Hyperparameters on ResNet-50/ImageNet

Parameter	Value
Number of workers	8
Optimizer	SGD with momentum
Global batch size	2048
Momentum	0.875
LR warmup	Linear for 8 epochs, starting from 0.256
LR schedule	cosine
LR decay	/10 at epoch 150 and 250
Epochs	90
Weight decay	1/32768
Label smoothing	0.1

B.3 Combination of PowerSGD and L-GreCo

We note that in all of our wide-range experiments, the compression ratio when `L-GreCo` is applied to PowerSGD generally increases during the training (see Figure B.2). This also aligns with the intuition behind the results of [AWL+21]. This suggests that in this scenario, `L-GreCo` is able to increase the compression in the less crucial learning periods, e.g., last epochs.

Table B.4: Hyperparameters on Transformer-XL/WikiText-103

Parameter	Value
Number of workers	8
Optimizer	LAMB
Global batch size	256
LR warmup	Linear for 1000 steps
LR schedule	cosine
Number of steps	40k
Weight decay	0.0

Table B.5: Hyperparameters on Transformer-LM/WikiText-103

Parameter	Value
Number of workers	8
Optimizer	Adam
Adam betas	(0.9, 0.98)
Global batch size	2048
LR warmup	Linear for 4000 steps starting from 10^{-7}
LR schedule	inverse sqrt
Number of steps	50k
Weight decay	0.01

B.4 Detailed experimental settings

For all the experiments, we used the standard hyperparameters, datasets, and data preprocessing. The detailed hyperparameters are shown in the tables B.2, B.3, B.4, and B.5. For the experiments with Rethink-GS [SDMA+21] we used hyperparameters presented in the Table B.1.

For preprocessing the images of CIFAR-100 datasets, we follow the standard data augmentation and normalization routines. Random cropping and horizontal random flipping were applied for data augmentation. We also normalized each color with the following mean and standard deviation values for each channel: (0.4914, 0.4822, 0.4465) and (0.2023, 0.1994, 0.2010).

ResNet50 model uses the following data augmentation. We perform random resized crop to 224×224, scale from 8% to 100%, and do a random horizontal flip. Also, we do normalization with means (0.485, 0.456, 0.406) and standard deviations (0.229, 0.224, 0.225).

For wikitext-103 preprocessing, we used the standard preprocessing tools and tokenizers provided by Nvidia Examples [Nvi20] and FairSeq library [OEB+19].

B.5 Profiling.

In order to explore the compression overhead, we run the profiling of the training. The result is presented in Figure. B.3. We compare operation timings for the original(uncompressed) training and training where the gradients are compressed with PowerSGD, rank 32. We can see that relatively expensive compression (PowerSGD is more time-consuming than QSGD and optimized TopK) takes less than 10% of the step time.

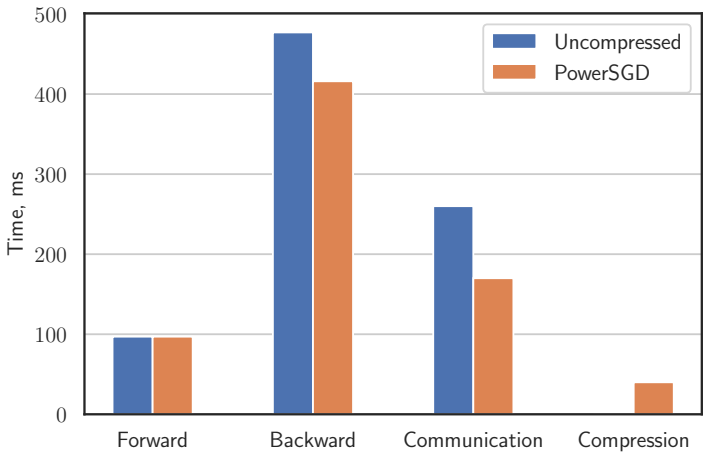
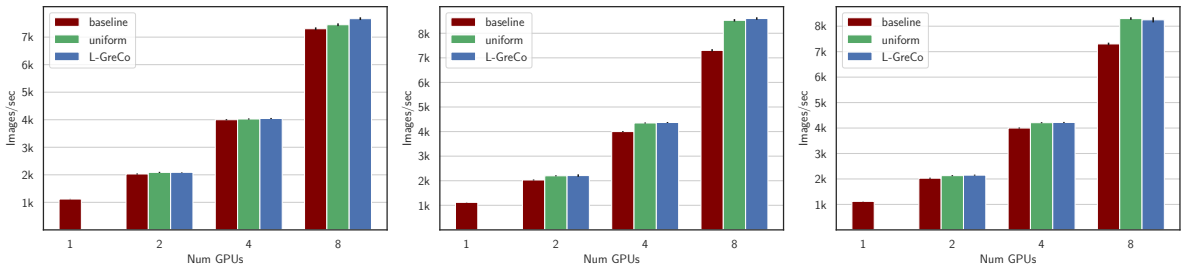
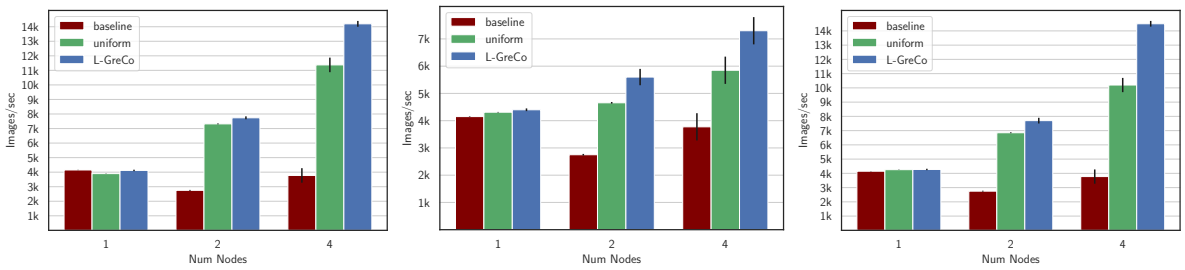


Figure B.3: Profiling of the training without compression vs PowerSGD compression, rank 32. Transformer-XL model on WikiText-103 dataset. Single node, RTX3090 GPUs.



(a) PowerSGD (b) QSGD (c) TopK

Figure B.4: Throughput for ResNet50/ImageNet. Single node, RTX3090 GPUs.

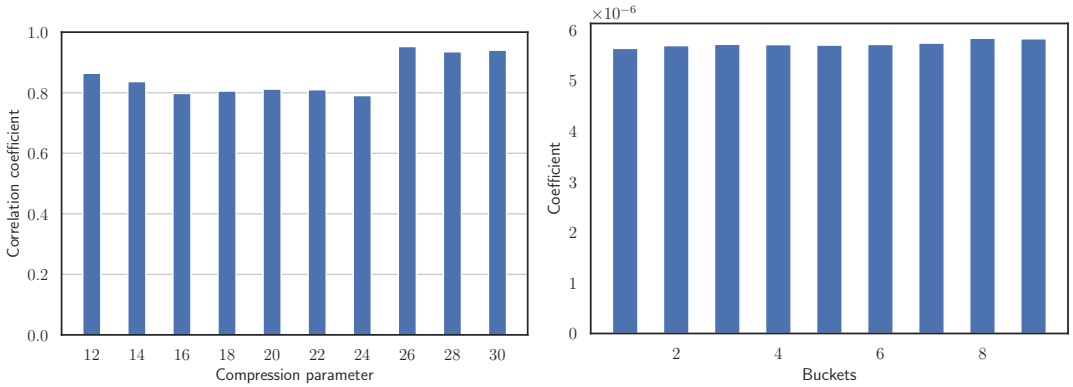


(a) PowerSGD (b) QSGD (c) TopK

Figure B.5: Throughput for ResNet50/ImageNet. Multi-node, each node has 4 RTX3090 GPUs.

B.6 Metrics connection

To verify the connection between loss-based and error-magnitude approaches we have collected the metrics for the same compression parameters with the same model. We started from the same checkpoint, ran for the same number of steps. For error-magnitude we collected the gradients in a buffer, for loss we estimated loss in the end of the experiment. In figure B.6a, we see that the resulting metrics have a high correlation.



(a) Metrics correlation

(b) Coefficients

Figure B.6: Correlation coefficients between metric values (a) for PowerSGD using loss-based and error-magnitude approaches as the sensitivity metrics. Timing coefficients per bucket (b) for timing-based approach. Transformer-XL/WikiText-103 model training with PowerSGD method.

B.7 Timing-based optimization

We have conducted the experiments described in Section 4.6.5 on Transformer-XL/WikiText103 for PowerSGD compression. We collected the per-bucket coefficients in the linear model to see if the impact of a bucket on the training performance is based not only on its size. Figure. B.6b shows that the coefficients are uniform accross the model meaning that the performance is a function of a parameters count in the bucket.

APPENDIX C

Appendix for Chapter 5

C.1 Training details.

For training of GPT-2 models we were using MosaicML [Mos22a] examples. The global batch size for 125M and 350M models was 256, for 1.3B - 512, resulting in 4 gradient accumulations at each iteration. For all models AdamW optimizer was used, the optimizer parameters are presented in the Table C.1. 125M model was trained in 4800 steps, 350M model in 13400 steps, 1.3B model in 14000 steps.

C.2 Network overhead experiments

In order to evaluate the effect on communications in FSDP training we conducted the synthetic experiment which reduces the bandwidth costs in each iteration. Specifically, given the buffer of size N which is about to be communicated, and compression ratio γ we only transmit the first N/γ elements. The results for our setup (4 x 8V100-32G GPUs) at different internode bandwidths is shown in the Figures.C.1, communication weights and gradients are reduced to the same compression ratio. We see that the most effect of compression is reached as expected for the largest 1.3B model and at lowest bandwidth. However, one can get around 80% speedup at high bandwidth when up to 8x compression ratio is applied. Also, we notice that $8\times$ compression almost reaches the ideal scaling for large model and has a evident overhead over the no-communication training in case of the small model. It infers that the large models have a bottleneck in bandwidth component of the communication and the small model has a dominating latency part.

To see the variance of the compression effects on weights and gradients we conducted the similar experiment for different combinations of compression ratio pairs (see C.2). We observe that weight compression gives more performance profits than gradient compression. This can be naturally explained by the fact that weights are communicated more frequently than gradients in FSDP (in this particular experiment weights are communicated 5 times per one gradient exchange) and the amount of transmissions per communication is similar.

The difference between the synthetic experiment and QSDP performance numbers with the same compression ratios can be justified by the performance inefficiency of NCCL point-to-point communication primitives on which QSDP compressed communication is based on -

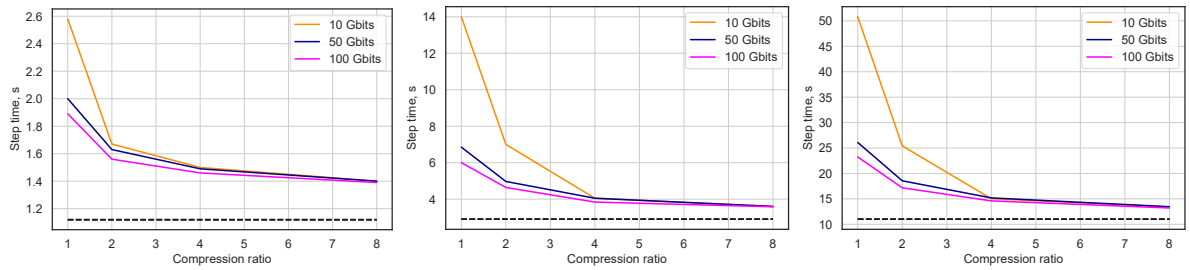
Algorithm C.1: Pseudocode of QSDP for a Fixed Layer

Input: worker p, layer input x_p, worker weight partition w_p.
function ExecuteForwardPass
  $qw_p \leftarrow \text{QuantizeWeights}(w_p)$ // Quantize p's weights
  $qw \leftarrow \text{AllGather}(qw_i \text{ for all } i)$ // Collect quantized weights
  $o_p \leftarrow \text{Layer}(qw, x_p)$ // Compute output for p
  $\text{free}(qw)$ // Discard aggregated layer weights
end function
function ExecuteBackwardPass
  $qw_p \leftarrow \text{QuantizeWeights}(w_p)$ // Quantize p's weights
  $qw \leftarrow \text{AllGather}(qw_i \text{ for all } i)$ // Collect quantized weights
  $g_p \leftarrow \text{Gradient}(qw, o_p)$ // Compute gradient for p
  $\text{free}(qw)$ // Discard aggregated layer weights
  $qg_p \leftarrow \text{QuantizeGradients}(g_p)$ // Quantize p's gradient
  $qg_p \leftarrow \text{ReduceScatter}(qg_i \text{ for each } i)$ // Distribute gradients
  $w_p \leftarrow \text{WeightUpdate}(qg_p, w_p)$ // Update p's weights
  $\text{free}(qg)$ // Discard aggregated gradients
end function

Table C.1: AdamW optimizer parameters.

	125M	350M	1.3B
learning rate	6e-4	3e-4	2e-4
betas	0.9, 0.95	0.9, 0.95	0.9, 0.95
epsilon	1e-8	1e-8	1e-8

the compression overhead in our experiments was verified to be negligible (less than 1% per iteration).



(a) 125M (b) 350M (c) 1.3B

Figure C.1: Compression vs average step time for different models at different inter-node bandwidths with fake compression (weights and gradients have the same compression ratio). Lower is better. The dashed line represents ideal scaling - training without communication.

C.3 Learned quantization

We implemented stochastic gradient descent optimization of quantization levels in PyTorch. We use learning rate 0.01, batch size 1024. We run the learning for each layer larger than 1e5 parameters, for other layers uniform quantization was used. We evaluate the quality of

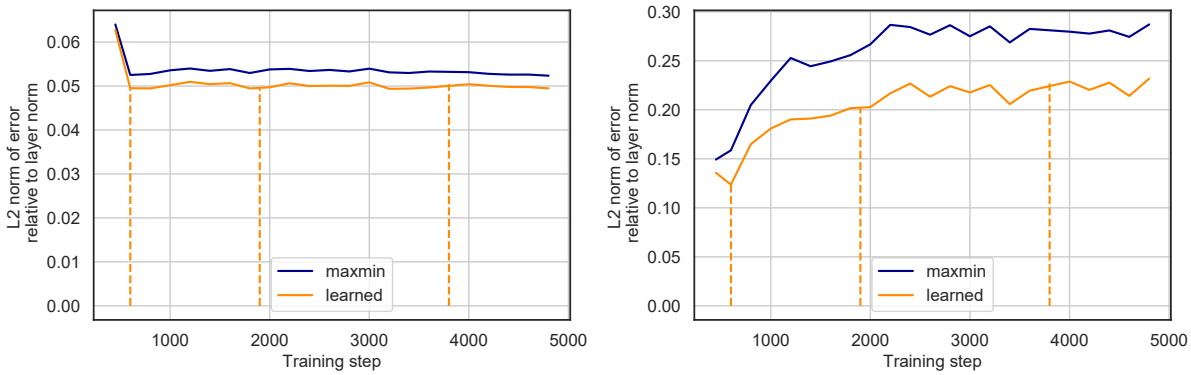
Table C.2: Training step timings (in seconds) for 1.3B model at 100 Gbps bandwidth with various combinations of weights and gradient compression ratio.

Weights ratio \ Gradients ratios	1	2	4	8
1	23.23	21.36	20.62	20.2
2	19.27	17.17	16.26	15.95
4	17.50	15.35	14.6	14.08
8	16.62	14.52	13.66	13.21

quantization levels by comparing L2 of compression error introduced by quantizing a buffer using the levels. We conducted the such evaluation for weights quantized to 5 bits and gradients quantized to 4 bits during the training of GPT 125M model. The results for one of the attention layers and LM head layer are shown in the Figures C.2 and C.3. The dashed vertical lines show the moment of running learning quantization levels algorithm. We see that compression error of learned quantization levels is constantly lower for the learned algorithm, and the lower bits-width (for gradients we use 4 bits quantizaton) the larger the gap between the considered methods. Also, we see that the compression error of the learned quantization only increases in sync with uniform quantization over time. It means that learning algorithm can be run only once, at the start of the training.

Also, we measured overhead of running learning algorithm for GPT 125M with weights quantized to 5 bits, gradients to 4 bits. The overhead of learning algorithm amounts to around 9 minutes, whereas the full training takes lasts 5 hours.

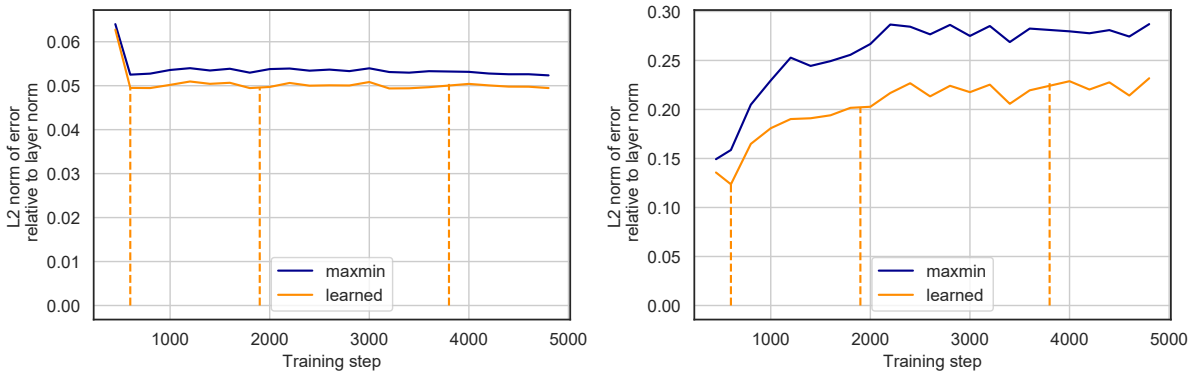
The extra experiments results with low bit-width quantization are shown in the Table. C.3. The number doesn't show full perplexity recovery but they represent the improvements achieved by learned levels algorithm. We can see that with learned quantization levels one can reduce up to 3 units of perplexity.



(a) Weights

(b) Gradients

Figure C.2: Compression error (L2 norm of the error relative to L2 norm of the input) comparison with learned quantization levels for attention layer of 125M model, W5G4 quantization.



(a) Weights

(b) Gradients

Figure C.3: Compression error (L2 norm of the error relative to L2 norm of the input) comparison with learned quantization levels for LM-Head layerof 125M model, W5G4 quantization.

Table C.3: Final perplexities of low-bits quantization of 125m GPT-2 model using the learned quantization levels.

	baseline	w3g32	w2g32	w8g3	w8g2
Uniform	35.81	45.53	57.92	39.91	44.79
Learned		42.31	56.54	37.72	44.65

C.4 Convergence experiments.

Quantization bucket size. To highlight the importance of bucket size choice on practice we trained the 125M model quantizing weights and gradients to 8 bits with larger than default bucket size. The results are shown in Figure C.4. We can see that training with quantization with bucket size 16284 has a remarkable gap in the convergence curve and does not recover full accuracy.

Quantization by Random shift. We have conducted an empirical study of the Theorem 2 to show the impact of the random shift in quantization on the training convergence. We setup a linear regression problem on the YearPredictionMSD dataset from UCI Machine learning repository data [DG17]. We trained the model quantizing weights and gradients of the linear model to 8 bits with bucket size 16 using the original QSGD and QSGD with random shift. We can see in the Figure. C.5 that QSGD with random shift has better convergence than QSGD in the setup with convex problem.

C.5 Convergence Proofs

In this section we provide the convergence analysis for our algorithms.

C.5.1 Overview

We use the notation and assumptions defined in Section 5.5.1. As all of our analyses revolve around bounding the progress made in a single iteration, to simplify notation we will generally use $\boldsymbol{x}$ to denote the current iterate, and $\boldsymbol{x}'$ to denote the iterate obtained after the generic

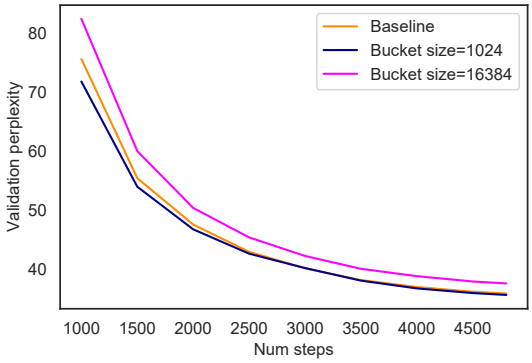


Figure C.4: Validation perplexity vs number of steps in quantized training with default and larger bucket size.

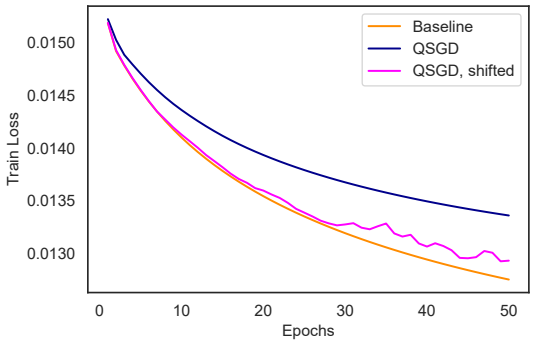


Figure C.5: Training loss vs number of steps for convex problem comparing QSGD and QSGD with random shift.

update:

$$\boldsymbol{x}' = Q_\delta^w\left(\boldsymbol{x} - \frac{\eta}{\beta}\boldsymbol{g}\left(\boldsymbol{x}\right)\right),$$

where β is the smoothness parameter of f. In Section C.5.2 we will first prove convergence for the deterministic method, where we have direct access to the gradients of f. The analysis precisely follows the steps we described in Section 5.5.1. Then, we extend this analysis to the case of stochastic gradients, and provide the full proof for Theorem 2. Finally, in Section C.5.3 we show that given an appropriate gradient quantization method with bounded variance, we can use it on top of our iteration to further reduce the required amount of communication, and thus prove Corollary 3.

Before proceeding, we first formally analyze the quantization method we defined in Section 5.5.1, and show some additional properties that will be useful later.

Lemma 5. *Let $\boldsymbol{v} \in \mathbb{R}^n$, and let $\delta > 0$. Then,*

$$\begin{aligned}
\mathbb{E}\left[Q_\delta^w\left(\boldsymbol{v}\right)\right] &= \boldsymbol{v}\,, \\
\mathbb{E}\left[\left\|Q_\delta^w\left(\boldsymbol{v}\right) - \boldsymbol{v}\right\|_2^2\right] &= \delta^2 \cdot \sum_{i=1}^n \left\{\frac{v_i}{\delta}\right\}\left(1 - \left\{\frac{v_i}{\delta}\right\}\right), \\
\mathbb{E}\left[\left\|Q_{r,\delta}^w\left(\boldsymbol{v}\right) - r1\right\|_0\right] &\leq \left\|\boldsymbol{v}\right\|_1 / \delta\,.
\end{aligned}$$

Since the proofs are technical, we defer them to Section C.5.4. The most important feature of this quantization scheme is captured by Lemma 4, which is crucial for our convergence proof. We first restate it, and prove it formally in Section C.5.4.

Lemma 4. *Let $\delta_\star > \delta > 0$, such that $\delta_\star/\delta \in \mathbb{Z}$. Let $\boldsymbol{x} \in \mathbb{R}^n$, and for all $r \in [-\delta_\star/2, \delta_\star/2)$, let an arbitrary $\boldsymbol{x}^\star_{r,\delta_\star} \in \delta_\star\mathbb{Z}^n + r1$. Then*

$$\mathbb{E}\left[\left\|Q_\delta^w\left(\boldsymbol{x}\right) - \boldsymbol{x}\right\|_2^2\right] \leq \frac{\delta}{\delta_\star}\mathbb{E}_r\left[\left\|\boldsymbol{x}^\star_{r,\delta_\star} - \boldsymbol{x}\right\|_2^2\right].$$

The proof crucially relies on the fact that $\delta/\delta_\star \in \mathbb{Z}$, and is rooted in the following inequality:

Lemma 6. *Let $y \in \mathbb{R}$ and $k \in \mathbb{Z}$. Then*

$$\left(1 - \left\{y\right\}\right)\left\{y\right\} \leq k\left(1 - \left\{\frac{y}{k}\right\}\right)\left\{\frac{y}{k}\right\}.$$

Proof. It suffices to consider $y \in [0, k]$, as both $\{y\}(1-\{y\})$ and $\{y/k\}(1-\{y/k\})$ are periodic within this interval. The function $\{y/k\}(1-\{y/k\})$ is a quadratic which is monotonically increasing over $[0, k/2]$ and symmetric around $k/2$. As $(1-\{y\})\{y\}$ is periodic on intervals of length 1 it suffices to show that $(1-\{y\})\{y\} \leq k\left(1-\left\{\frac{y}{k}\right\}\right)\left\{\frac{y}{k}\right\}$ on the interval $[0,1]$. At this point we can drop the fractional part, and simply need to compare two quadratics over $[0,1]$. Equivalently we need to show that $k(1-y/k)y/k \geq y(1-y)$ over $[0,1]$, which after simplifying both sides is equivalent to $y^2(1-1/k) \geq 0$ over this interval, which is true. □

Finally, we provide some basic optimization inequalities, which will allow us to prove our theorems.

Optimization Basics. The first Lemma bounds the change in function value using smoothness, while the latter upper bounds the ℓ_2 distance to optimality using the error in function value. We provide the proofs in Sections C.5.4 and C.5.4.

Lemma 7. *Let $f : \mathbb{R}^n \to \mathbb{R}$ be a β-smooth function. Then for any $\boldsymbol{\Delta} \in \mathbb{R}^n$,*

$$f(\boldsymbol{x}+\boldsymbol{\Delta}) \leq f(\boldsymbol{x}) + (1-\eta)\left\langle \nabla f(\boldsymbol{x}), \boldsymbol{\Delta}\right\rangle - \frac{\eta^2}{2\beta}\left\|\nabla f(\boldsymbol{x})\right\|_2^2 + \frac{\beta}{2}\left\|\frac{\eta}{\beta}\nabla f(\boldsymbol{x}) + \boldsymbol{\Delta}\right\|_2^2 .$$

Lemma 8. *If $f : \mathbb{R}^n \to \mathbb{R}$ satisfies the α-PL condition, then for all $\boldsymbol{x} \in \mathbb{R}^n$,*

$$f(\boldsymbol{x}) - f^\star \geq \frac{\alpha}{2}\left\|\boldsymbol{x}-\boldsymbol{x}^\star\right\|_2^2 ,$$

where $\boldsymbol{x}^\star \in \arg\min_{\boldsymbol{x}} f(\boldsymbol{x})$.

We are now ready to prove the main theorems in this paper.

C.5.2 SGD with weight quantization

We first prove the stepping lemma for our quantized gradient method, in the case where full gradients are available. The steps are essentially the same we described in Section 5.5.1.

Lemma 9. *Let $f(\boldsymbol{x}) : \mathbb{R}^n \to \mathbb{R}$ be a β-smooth and α-PL function. For each $r \in [-\delta_\star/2, \delta_\star/2]$, let $\boldsymbol{x}^\star_{r,\delta_\star}$ be any minimizer of f over $\delta_\star\mathbb{Z}+r$. Let $\delta = \frac{\delta_\star}{\lceil 4(\beta/\alpha)^2\rceil}$. Then letting $\boldsymbol{x}' = Q^w_\delta\left(\boldsymbol{x} - \frac{1}{\beta}\nabla f(\boldsymbol{x})\right)$, one has that in expectation over the random bits used by the quantization operator:*

$$\mathbb{E}f(\boldsymbol{x}') - \mathbb{E}f\left(\boldsymbol{x}^\star_{r,\delta_\star}\right) \leq \left(1-\frac{\alpha}{2\beta}\right)\left(\mathbb{E}f(\boldsymbol{x}) - \mathbb{E}f\left(\boldsymbol{x}^\star_{r,\delta_\star}\right)\right) .$$

Proof. Letting $\boldsymbol{\Delta} = \boldsymbol{x}' - \boldsymbol{x}$, we write:

$$\left\|\frac{1}{\beta}\nabla f(\boldsymbol{x}) + \boldsymbol{\Delta}\right\|_2^2 = \left\|(\boldsymbol{x}+\boldsymbol{\Delta}) - \left(\boldsymbol{x} - \frac{1}{\beta}\nabla f(\boldsymbol{x})\right)\right\|_2^2 = \left\|Q^w_\delta\left(\boldsymbol{x} - \frac{1}{\beta}\nabla f(\boldsymbol{x})\right) - \left(\boldsymbol{x} - \frac{1}{\beta}\nabla f(\boldsymbol{x})\right)\right\|_2^2 .$$

Also using Lemma 4, we have that for any $\boldsymbol{x}^\star \in \arg\min_{\boldsymbol{x}} f(\boldsymbol{x})$,

$$\begin{aligned}\mathbb{E}\left\|Q^w_\delta\left(\boldsymbol{x} - \frac{1}{\beta}\nabla f(\boldsymbol{x})\right) - \left(\boldsymbol{x} - \frac{1}{\beta}\nabla f(\boldsymbol{x})\right)\right\|_2^2 &\leq \frac{\delta}{\delta_\star}\mathbb{E}_r\left[\left\|\boldsymbol{x} - \frac{1}{\beta}\nabla f(\boldsymbol{x}) - \boldsymbol{x}^\star_{r,\delta_\star}\right\|_2^2\right] \\ &\leq 2\frac{\delta}{\delta_\star}\left(\mathbb{E}_r\left[\left\|\boldsymbol{x}^\star_{r,\delta_\star} - \boldsymbol{x}^\star\right\|_2^2\right] + \left\|\boldsymbol{x} - \frac{1}{\beta}\nabla f(\boldsymbol{x}) - \boldsymbol{x}^\star\right\|_2^2\right) .\end{aligned}$$

Using the α-PL condition we upper bound distance from $\boldsymbol{x}^\star$ with function value i.e.

$$\left\|\boldsymbol{x}^\star_{r,\delta_\star} - \boldsymbol{x}^\star\right\|_2^2 \leq \frac{2}{\alpha}\cdot\left(f\left(\boldsymbol{x}^\star_{r,\delta_\star}\right) - f\left(\boldsymbol{x}^\star\right)\right),$$
$$\left\|\boldsymbol{x} - \frac{1}{\beta}\nabla f\left(\boldsymbol{x}\right) - \boldsymbol{x}^\star\right\|_2^2 \leq \frac{2}{\alpha}\left(f\left(\boldsymbol{x} - \frac{1}{\beta}\nabla f\left(\boldsymbol{x}\right)\right) - f\left(\boldsymbol{x}^\star\right)\right).$$

Combining these with Lemma 7 for $\eta = 1$ we conclude that

$$\begin{aligned}\mathbb{E}f\left(\boldsymbol{x}'\right) &\leq f\left(\boldsymbol{x}\right) - \frac{1}{2\beta}\left\|\nabla f\left(\boldsymbol{x}\right)\right\|_2^2 \\ &\quad + 2\frac{\delta}{\delta_\star}\cdot\frac{\beta}{\alpha}\left(f\left(\boldsymbol{x} - \frac{1}{\beta}\nabla f\left(\boldsymbol{x}\right)\right) - f\left(\boldsymbol{x}^\star\right)\right) + 2\frac{\delta}{\delta_\star}\cdot\frac{\beta}{\alpha}\cdot\mathbb{E}_r\left[f\left(\boldsymbol{x}^\star_{r,\delta_\star}\right) - f\left(\boldsymbol{x}^\star\right)\right].\end{aligned}$$

Again, using the PL condition we lower bound $\frac{1}{2}\left\|\nabla f\left(\boldsymbol{x}\right)\right\|_2^2 \geq \alpha\left(f\left(\boldsymbol{x}\right) - f\left(\boldsymbol{x}^\star\right)\right)$, which gives

$$\begin{aligned}&\mathbb{E}\left[f\left(\boldsymbol{x}'\right) - f\left(\boldsymbol{x}^\star\right)\right] \\ &\leq \left(1 - \frac{\alpha}{\beta}\right)\left(f\left(\boldsymbol{x}\right) - f\left(\boldsymbol{x}^\star\right)\right) + 2\frac{\delta}{\delta_\star}\cdot\frac{\beta}{\alpha}\left(f\left(\boldsymbol{x} - \frac{1}{\beta}\nabla f\left(\boldsymbol{x}\right)\right) - f\left(\boldsymbol{x}^\star\right)\right) + 2\frac{\delta}{\delta_\star}\cdot\frac{\beta}{\alpha}\cdot\mathbb{E}_r\left[f\left(\boldsymbol{x}^\star_{r,\delta_\star}\right) - f\left(\boldsymbol{x}^\star\right)\right] \\ &\leq \left(1 - \frac{\alpha}{\beta} + 2\frac{\delta}{\delta_\star}\cdot\frac{\beta}{\alpha}\right)\left(f\left(\boldsymbol{x}\right) - f\left(\boldsymbol{x}^\star\right)\right) + 2\frac{\delta}{\delta_\star}\cdot\frac{\beta}{\alpha}\cdot\mathbb{E}_r\left[f\left(\boldsymbol{x}^\star_{r,\delta_\star}\right) - f\left(\boldsymbol{x}^\star\right)\right].\end{aligned}$$

Equivalently we obtain

$$\begin{aligned}\mathbb{E}f\left(\boldsymbol{x}'\right) - \mathbb{E}f\left(\boldsymbol{x}^\star_{r,\delta_\star}\right) &\leq \left(1 - \frac{\alpha}{\beta} + 2\frac{\delta}{\delta_\star}\cdot\frac{\beta}{\alpha}\right)\left(f\left(\boldsymbol{x}\right) - \mathbb{E}f\left(\boldsymbol{x}^\star_{r,\delta_\star}\right)\right) + 2\frac{\delta}{\delta_\star}\cdot\frac{\beta}{\alpha}\cdot\mathbb{E}\left[f\left(\boldsymbol{x}^\star_{r,\delta_\star}\right) - f\left(\boldsymbol{x}^\star\right)\right] \\ &\quad + f\left(\boldsymbol{x}^\star\right) - \mathbb{E}f\left(\boldsymbol{x}^\star_{r,\delta_\star}\right) \\ &= \left(1 - \frac{\alpha}{\beta} + 2\frac{\delta}{\delta_\star}\cdot\frac{\beta}{\alpha}\right)\left(f\left(\boldsymbol{x}\right) - \mathbb{E}f\left(\boldsymbol{x}^\star_{r,\delta_\star}\right)\right) + \left(2\frac{\delta}{\delta_\star}\cdot\frac{\beta}{\alpha} - \frac{\alpha}{\beta} + 2\frac{\delta}{\delta_\star}\cdot\frac{\beta}{\alpha}\right)\cdot\mathbb{E}\left[f\left(\boldsymbol{x}^\star_{r,\delta_\star}\right) - f\left(\boldsymbol{x}^\star\right)\right] \\ &= \left(1 - \frac{\alpha}{\beta} + 2\frac{\delta}{\delta_\star}\cdot\frac{\beta}{\alpha}\right)\left(f\left(\boldsymbol{x}\right) - \mathbb{E}f\left(\boldsymbol{x}^\star_{r,\delta_\star}\right)\right) + \left(4\frac{\delta}{\delta_\star}\cdot\frac{\beta}{\alpha} - \frac{\alpha}{\beta}\right)\cdot\mathbb{E}\left[f\left(\boldsymbol{x}^\star_{r,\delta_\star}\right) - f\left(\boldsymbol{x}^\star\right)\right].\end{aligned}$$

Since we set $\delta/\delta_\star = \frac{1}{\left\lceil 4(\beta/\alpha)^2\right\rceil}$, the second term is non-positive. Therefore in this case we have

$$\mathbb{E}f\left(\boldsymbol{x}'\right) - \mathbb{E}f\left(\boldsymbol{x}^\star_{r,\delta_\star}\right) \leq \left(1 - \frac{\alpha}{2\beta}\right)\left(\mathbb{E}f\left(\boldsymbol{x}\right) - \mathbb{E}f\left(\boldsymbol{x}^\star_{r,\delta_\star}\right)\right),$$

which concludes the proof. □

We now generalize the proof of Lemma 9 to the case where only stochastic gradients are available. The proof is essentially the same, the main difference being that we isolate terms involving the difference between the stochastic and the true gradient, which we bound separately using our variance bound.

Lemma 10. *Let $f : \mathbb{R}^n \to \mathbb{R}$ be a β-smooth and α-PL function. For each $r \in [-\delta_\star/2, \delta_\star/2]$, let $\boldsymbol{x}^\star_{r,\delta_\star}$ be any minimizer of f over $\delta_\star\mathbb{Z}+r$. Let $\delta = \frac{\eta}{\left\lceil 16(\beta/\alpha)^2\right\rceil}\cdot\delta_\star$. Let $\boldsymbol{x}' = Q_\delta^w\left(\boldsymbol{x} - \frac{\eta}{\beta}\boldsymbol{g}\left(\boldsymbol{x}\right)\right)$, where $\boldsymbol{g}\left(\boldsymbol{x}\right)$ is an unbiased estimator for $\nabla f\left(\boldsymbol{x}\right)$ i.e. $\mathbb{E}\left[\boldsymbol{g}\left(\boldsymbol{x}\right)|\boldsymbol{x}\right] = \nabla f\left(\boldsymbol{x}\right)$, and $0 < \eta \leq 1$ is a step size parameter. Furthermore assume that the variance of $\boldsymbol{g}\left(\boldsymbol{x}\right)$ is bounded*

$\mathbb{E}\left\|\boldsymbol{g}\left(\boldsymbol{x}\right)-\nabla f\left(\boldsymbol{x}\right)\right\|_2^2 \leq \sigma^2$, *for a real parameter* $\sigma > 0$. *Then, for* $r \sim$ *Unif*$([-\delta_\star/2, \delta_\star/2))$, *in expectation over the gradient stochasticity:*

$$\mathbb{E}\left[f\left(\boldsymbol{x}'\right)|\boldsymbol{x}\right]-\mathbb{E}f\left(\boldsymbol{x}^\star_{r,\delta_\star}\right) \leq \left(1-\frac{3}{4}\eta\frac{\alpha}{\beta}\right)\left(f\left(\boldsymbol{x}\right)-\mathbb{E}f\left(\boldsymbol{x}^\star_{r,\delta_\star}\right)\right)+\frac{5}{4}\frac{\eta^2}{\beta}\sigma^2\,.$$

Proof. We follow the analysis from Lemma 9, while moving the stochastic gradients into expressions that involve the stochastic variance. Letting $\delta = \boldsymbol{x}' - \boldsymbol{x}$, we write:

$$\begin{aligned}\left\|\frac{\eta}{\beta}\nabla f\left(\boldsymbol{x}\right)+\boldsymbol{\Delta}\right\|_2^2 &= \left\|\left(\boldsymbol{x}+\boldsymbol{\Delta}\right)-\left(\boldsymbol{x}-\frac{\eta}{\beta}\nabla f\left(\boldsymbol{x}\right)\right)\right\|_2^2 = \left\|Q^w_{r,\delta}\left(\boldsymbol{x}-\frac{\eta}{\beta}\boldsymbol{g}\left(\boldsymbol{x}\right)\right)-\left(\boldsymbol{x}-\frac{\eta}{\beta}\nabla f\left(\boldsymbol{x}\right)\right)\right\|_2^2 \\ &\leq 2\left\|Q^w_{r,\delta}\left(\boldsymbol{x}-\frac{\eta}{\beta}\boldsymbol{g}\left(\boldsymbol{x}\right)\right)-\left(\boldsymbol{x}-\frac{\eta}{\beta}\boldsymbol{g}\left(\boldsymbol{x}\right)\right)\right\|_2^2+2\left\|\frac{\eta}{\beta}\left(\boldsymbol{g}\left(\boldsymbol{x}\right)-\nabla f\left(\boldsymbol{x}\right)\right)\right\|_2^2,\end{aligned}$$

where we used the inequality $\left\|\boldsymbol{a}+\boldsymbol{b}\right\|_2^2 \leq 2\left\|\boldsymbol{a}\right\|_2^2+2\left\|\boldsymbol{b}\right\|_2^2$. Also using Lemma 4, we have that for any $\boldsymbol{x}^\star \in \arg\min_{\boldsymbol{x}} f\left(\boldsymbol{x}\right)$,

$$\begin{aligned}&\mathbb{E}\left\|Q^w_{r,\delta}\left(\boldsymbol{x}-\frac{\eta}{\beta}\boldsymbol{g}\left(\boldsymbol{x}\right)\right)-\left(\boldsymbol{x}-\frac{\eta}{\beta}\boldsymbol{g}\left(\boldsymbol{x}\right)\right)\right\|_2^2 \\ &\leq \frac{\delta}{\delta_\star}\mathbb{E}\left[\left\|\boldsymbol{x}-\frac{\eta}{\beta}\boldsymbol{g}\left(\boldsymbol{x}\right)-\boldsymbol{x}^\star_{r,\delta_\star}\right\|_2^2\right] \\ &\leq 2\frac{\delta}{\delta_\star}\left(\mathbb{E}\left[\left\|\boldsymbol{x}^\star_{r,\delta_\star}-\boldsymbol{x}^\star\right\|_2^2\right]+\left\|\boldsymbol{x}-\frac{\eta}{\beta}\boldsymbol{g}\left(\boldsymbol{x}\right)-\boldsymbol{x}^\star\right\|_2^2\right) \\ &\leq 2\frac{\delta}{\delta_\star}\left(\mathbb{E}\left[\left\|\boldsymbol{x}^\star_{r,\delta_\star}-\boldsymbol{x}^\star\right\|_2^2\right]+2\left\|\boldsymbol{x}-\frac{\eta}{\beta}\nabla f\left(\boldsymbol{x}\right)-\boldsymbol{x}^\star\right\|_2^2+2\left\|\frac{\eta}{\beta}\left(\boldsymbol{g}\left(\boldsymbol{x}\right)-\nabla f\left(\boldsymbol{x}\right)\right)\right\|_2^2\right)\,.\end{aligned}$$

Using the α-PL condition we upper bound distance from $\boldsymbol{x}^\star$ with function value i.e.

$$\begin{aligned}\left\|\boldsymbol{x}^\star_{r,\delta_\star}-\boldsymbol{x}^\star\right\|_2^2 &\leq \frac{2}{\alpha}\cdot\left(f\left(\boldsymbol{x}^\star_{r,\delta_\star}\right)-f\left(\boldsymbol{x}^\star\right)\right)\,, \\ \left\|\boldsymbol{x}-\frac{\eta}{\beta}\nabla f\left(\boldsymbol{x}\right)-\boldsymbol{x}^\star\right\|_2^2 &\leq \frac{2}{\alpha}\left(f\left(\boldsymbol{x}-\frac{\eta}{\beta}\nabla f\left(\boldsymbol{x}\right)\right)-f\left(\boldsymbol{x}^\star\right)\right)\,.\end{aligned}$$

Combining these with Lemma 7 we conclude that in expectation over the random shift:

$$\begin{aligned}
f\left(\boldsymbol{x}'\right) &\leq f\left(\boldsymbol{x}\right) + \left(1-\eta\right)\left\langle \nabla f\left(\boldsymbol{x}\right), \boldsymbol{\Delta}\right\rangle - \frac{\eta^2}{2\beta}\left\|\nabla f\left(\boldsymbol{x}\right)\right\|_2^2 \\
&+ \frac{\beta}{2}\cdot\left(2\left\|Q_{r,\delta}^{w}\left(\boldsymbol{x}-\frac{\eta}{\beta}\boldsymbol{g}\left(\boldsymbol{x}\right)\right)-\left(\boldsymbol{x}-\frac{\eta}{\beta}\nabla f_i\left(\boldsymbol{x}\right)\right)\right\|_2^2 + 2\left\|\frac{\eta}{\beta}\left(\boldsymbol{g}\left(\boldsymbol{x}\right)-\nabla f\left(\boldsymbol{x}\right)\right)\right\|_2^2\right) \\
&\leq f\left(\boldsymbol{x}\right) + \left(1-\eta\right)\left\langle \nabla f\left(\boldsymbol{x}\right), \boldsymbol{\Delta}\right\rangle - \frac{\eta^2}{2\beta}\left\|\nabla f\left(\boldsymbol{x}\right)\right\|_2^2 \\
&+ 2\beta\frac{\delta}{\delta_\star}\cdot\left(\left\|\boldsymbol{x}_{r,\delta_\star}^{\star}-\boldsymbol{x}^{\star}\right\|_2^2 + 2\left\|x-\frac{\eta}{\beta}\nabla f\left(x\right)-\boldsymbol{x}^{\star}\right\|_2^2 + 2\left\|\frac{\eta}{\beta}\left(\boldsymbol{g}\left(\boldsymbol{x}\right)-\nabla f\left(\boldsymbol{x}\right)\right)\right\|_2^2\right) \\
&+ \frac{\eta^2}{\beta}\left\|\boldsymbol{g}\left(\boldsymbol{x}\right)-\nabla f\left(\boldsymbol{x}\right)\right\|_2^2 \\
&\leq f\left(\boldsymbol{x}\right) + \left(1-\eta\right)\left\langle \nabla f\left(\boldsymbol{x}\right), \boldsymbol{\Delta}\right\rangle - \frac{\eta^2}{2\beta}\left\|\nabla f\left(\boldsymbol{x}\right)\right\|_2^2 \\
&+ 4\frac{\delta}{\delta_\star}\frac{\beta}{\alpha}\left(f\left(\boldsymbol{x}-\frac{\eta}{\beta}\nabla f\left(\boldsymbol{x}\right)\right)-f\left(\boldsymbol{x}^{\star}\right)\right) + 4\frac{\delta}{\delta_\star}\frac{\beta}{\alpha}\left(f\left(\boldsymbol{x}_{r,\delta_\star}^{\star}\right)-f\left(\boldsymbol{x}^{\star}\right)\right) \\
&+ \frac{\eta^2}{\beta}\left(1+4\frac{\delta}{\delta_\star}\right)\left\|\boldsymbol{g}\left(\boldsymbol{x}\right)-\nabla f\left(\boldsymbol{x}\right)\right\|_2^2 .
\end{aligned}$$

Again, using the PL condition we lower bound $\frac{1}{2}\left\|\nabla f\left(\boldsymbol{x}\right)\right\|_2^2 \geq \alpha\left(f\left(\boldsymbol{x}\right)-f\left(\boldsymbol{x}^{\star}\right)\right)$, which gives that in expectation over the random shift:

$$\begin{aligned}
f\left(\boldsymbol{x}'\right)-f\left(\boldsymbol{x}^{\star}\right) &\leq \left(1-\eta^2\frac{\alpha}{\beta}\right)\left(f\left(\boldsymbol{x}\right)-f\left(\boldsymbol{x}^{\star}\right)\right) + \left(1-\eta\right)\left\langle \nabla f\left(\boldsymbol{x}\right), \boldsymbol{\Delta}\right\rangle \\
&+ 4\frac{\delta}{\delta_\star}\frac{\beta}{\alpha}\left(f\left(\boldsymbol{x}-\frac{\eta}{\beta}\nabla f\left(\boldsymbol{x}\right)\right)-f\left(\boldsymbol{x}^{\star}\right)\right) + 4\frac{\delta}{\delta_\star}\frac{\beta}{\alpha}\left(f\left(\boldsymbol{x}_{r,\delta_\star}^{\star}\right)-f\left(\boldsymbol{x}^{\star}\right)\right) \\
&+ \frac{\eta^2}{\beta}\left(1+4\frac{\delta}{\delta_\star}\right)\left\|\boldsymbol{g}\left(\boldsymbol{x}\right)-\nabla f\left(\boldsymbol{x}\right)\right\|_2^2 \\
&\leq \left(1-\eta^2\frac{\alpha}{\beta}+4\frac{\delta}{\delta_\star}\frac{\beta}{\alpha}\right)\left(f\left(\boldsymbol{x}\right)-f\left(\boldsymbol{x}^{\star}\right)\right) + 4\frac{\delta}{\delta_\star}\frac{\beta}{\alpha}\left(f\left(\boldsymbol{x}_{r,\delta_\star}^{\star}\right)-f\left(\boldsymbol{x}^{\star}\right)\right) \\
&+ \left(1-\eta\right)\left\langle \nabla f\left(\boldsymbol{x}\right), \boldsymbol{\Delta}\right\rangle + \frac{\eta^2}{\beta}\left(1+4\frac{\delta}{\delta_\star}\right)\left\|\boldsymbol{g}\left(\boldsymbol{x}\right)-\nabla f\left(\boldsymbol{x}\right)\right\|_2^2 .
\end{aligned}$$

and equivalently, in expectation over the random shift:

$$\begin{aligned}
\mathbb{E}\left[f\left(\boldsymbol{x}'\right)-f\left(\boldsymbol{x}_{r,\delta_\star}^{\star}\right)\right] &\leq \left(1-\eta^2\frac{\alpha}{\beta}+4\frac{\delta}{\delta_\star}\frac{\beta}{\alpha}\right)\left(f\left(\boldsymbol{x}\right)-\mathbb{E}f\left(\boldsymbol{x}_{r,\delta_\star}^{\star}\right)\right) + \left(8\frac{\delta}{\delta_\star}\frac{\beta}{\alpha}-\eta^2\frac{\alpha}{\beta}\right)\left(\mathbb{E}f\left(\boldsymbol{x}_{r,\delta_\star}^{\star}\right)-f\left(\boldsymbol{x}^{\star}\right)\right) \\
&+ \left(1-\eta\right)\left\langle \nabla f\left(\boldsymbol{x}\right), \mathbb{E}\left[\boldsymbol{\Delta}\right]\right\rangle + \frac{\eta^2}{\beta}\left(1+4\frac{\delta}{\delta_\star}\right)\left\|\boldsymbol{g}\left(\boldsymbol{x}\right)-\nabla f\left(\boldsymbol{x}\right)\right\|_2^2 .
\end{aligned}$$

At this point we use Lemma 5 to write

$$\begin{aligned}
\mathbb{E}\left[\boldsymbol{\Delta}\right] &= \mathbb{E}\left[Q_{r,\delta}^{w}\left(\boldsymbol{x}-\frac{\eta}{\beta}\boldsymbol{g}\left(\boldsymbol{x}\right)\right)-x\right] \\
&= -\frac{\eta}{\beta}\boldsymbol{g}\left(\boldsymbol{x}\right) ,
\end{aligned}$$

and thus

$$(1-\eta)\left\langle \nabla f\left(\boldsymbol{x}\right),\mathbb{E}\left[\boldsymbol{\Delta}\right]\right\rangle =-\frac{\eta}{\beta}\left(1-\eta\right)\left\langle \nabla f\left(\boldsymbol{x}\right),\boldsymbol{g}\left(\boldsymbol{x}\right)\right\rangle .$$

Therefore, after taking expectation over both the random shift and gradient stochasticity we obtain:

$$\begin{aligned}
&\mathbb{E}\left[f\left(\boldsymbol{x}'\right)-f\left(\boldsymbol{x}_{r,\delta_\star}^\star\right)|\boldsymbol{x}\right]\\
&\leq\left(1-\eta^2\frac{\alpha}{\beta}+4\frac{\delta}{\delta_\star}\frac{\beta}{\alpha}\right)\left(f\left(\boldsymbol{x}\right)-\mathbb{E}f\left(\boldsymbol{x}_{r,\delta_\star}^\star\right)\right)+\left(8\frac{\delta}{\delta_\star}\frac{\beta}{\alpha}-\eta^2\frac{\alpha}{\beta}\right)\left(\mathbb{E}f\left(\boldsymbol{x}_{r,\delta_\star}^\star\right)-f\left(\boldsymbol{x}^\star\right)\right)\\
&-\frac{\eta}{\beta}\left(1-\eta\right)\left\|\nabla f\left(\boldsymbol{x}\right)\right\|_2^2+\frac{\eta^2}{\beta}\left(1+4\frac{\delta}{\delta_\star}\right)\sigma^2\\
&\leq\left(1-\eta^2\frac{\alpha}{\beta}+4\frac{\delta}{\delta_\star}\frac{\beta}{\alpha}\right)\left(f\left(\boldsymbol{x}\right)-\mathbb{E}f\left(\boldsymbol{x}_{r,\delta_\star}^\star\right)\right)+\left(8\frac{\delta}{\delta_\star}\frac{\beta}{\alpha}-\eta^2\frac{\alpha}{\beta}\right)\left(\mathbb{E}f\left(\boldsymbol{x}_{r,\delta_\star}^\star\right)-f\left(\boldsymbol{x}^\star\right)\right)\\
&-2\eta\left(1-\eta\right)\frac{\alpha}{\beta}\left(f\left(\boldsymbol{x}\right)-f\left(\boldsymbol{x}^\star\right)\right)+\frac{\eta^2}{\beta}\left(1+4\frac{\delta}{\delta_\star}\right)\sigma^2\\
&=\left(1-\eta^2\frac{\alpha}{\beta}-2\eta\left(1-\eta\right)\frac{\alpha}{\beta}+4\frac{\delta}{\delta_\star}\frac{\beta}{\alpha}\right)\left(f\left(\boldsymbol{x}\right)-\mathbb{E}f\left(\boldsymbol{x}_{r,\delta_\star}^\star\right)\right)+\left(8\frac{\delta}{\delta_\star}\frac{\beta}{\alpha}-\eta^2\frac{\alpha}{\beta}-2\eta\left(1-\eta\right)\frac{\alpha}{\beta}\right)\left(\mathbb{E}f\left(\boldsymbol{x}_{r,\delta_\star}^\star\right)\right.\\
&+\frac{\eta^2}{\beta}\left(1+4\frac{\delta}{\delta_\star}\right)\sigma^2.
\end{aligned}$$

Since we set $\delta/\delta_\star=\frac{\eta}{\left\lceil 16(\beta/\alpha)^2\right\rceil}$, the second term is non-positive. Therefore in this case we have

$$\begin{aligned}
\mathbb{E}\left[f\left(\boldsymbol{x}'\right)-\mathbb{E}f\left(\boldsymbol{x}_{r,\delta_\star}^\star\right)|\boldsymbol{x}\right]&\leq\left(1-\eta^2\frac{\alpha}{\beta}-2\eta\left(1-\eta\right)\frac{\alpha}{\beta}+\frac{\eta}{4}\frac{\alpha}{\beta}\right)\left(f\left(\boldsymbol{x}\right)-\mathbb{E}f\left(\boldsymbol{x}_{r,\delta_\star}^\star\right)\right)+\frac{\eta^2}{\beta}\left(1+\frac{\eta}{4}\left(\frac{\alpha}{\beta}\right)^2\right)\\
&\leq\left(1-\frac{7}{4}\eta\frac{\alpha}{\beta}+\eta^2\frac{\alpha}{\beta}\right)\left(f\left(\boldsymbol{x}\right)-\mathbb{E}f\left(\boldsymbol{x}_{r,\delta_\star}^\star\right)\right)+\frac{\eta^2}{\beta}\left(1+\frac{\eta}{4}\left(\frac{\alpha}{\beta}\right)^2\right)\sigma^2\\
&\leq\left(1-\frac{3}{4}\eta\frac{\alpha}{\beta}\right)\left(f\left(\boldsymbol{x}\right)-\mathbb{E}f\left(\boldsymbol{x}_{r,\delta_\star}^\star\right)\right)+\frac{5}{4}\frac{\eta^2}{\beta}\sigma^2,
\end{aligned}$$

as long as $\eta\leq 1$. This concludes the proof. □

Using Lemma 10 the proof of Theorem 2 follows very easily.

Theorem 2. *Let $\alpha,\beta,\delta_\star,\varepsilon>0$ and $\sigma\geq 0$ be real parameters, and let $\eta=\min\left\{\frac{3}{10}\frac{\varepsilon\alpha}{\sigma^2},1\right\}$. Let $f:\mathbb{R}^n\to\mathbb{R}$ be a β-smooth and α-PL function, with access to a stochastic gradient $\boldsymbol{g}\left(\boldsymbol{x}\right)$, i.e. $\mathbb{E}\left[\boldsymbol{g}\left(\boldsymbol{x}\right)|\boldsymbol{x}\right]=\nabla f\left(\boldsymbol{x}\right)$ with bounded variance $\mathbb{E}\left\|\boldsymbol{g}\left(\boldsymbol{x}\right)-\nabla f\left(\boldsymbol{x}\right)\right\|_2^2\leq\sigma^2$. For each $r\in[-\delta_\star/2,\delta_\star/2)$, let $\boldsymbol{x}_{r,\delta_\star}^\star$ be any minimizer of f over $\delta_\star\mathbb{Z}^n+r1$. Let $\delta=\frac{\eta}{\left\lceil 16(\beta/\alpha)^2\right\rceil}\cdot\delta_\star$. Consider the iteration:*

$$\boldsymbol{x}_{t+1}=Q_\delta^w\left(\boldsymbol{x}_t-\frac{\eta}{\beta}\boldsymbol{g}\left(\boldsymbol{x}_t\right)\right).$$

In $T=\frac{10}{\eta}\cdot\frac{\beta}{\alpha}\ln\frac{f(\boldsymbol{x}_0)-\mathbb{E}f(\boldsymbol{x}_{r,\delta_\star}^\star)}{\varepsilon}$ iterations we obtain a point $\boldsymbol{x}_T$ satisfying $\mathbb{E}f\left(\boldsymbol{x}_T\right)-\mathbb{E}f(\boldsymbol{x}_{r,\delta_\star}^\star)\leq\varepsilon$.

Proof. Plugging in Lemma 10 and applying it for $T = \frac{10}{\eta}\frac{\beta}{\alpha}\ln\frac{f(\boldsymbol{x}_0)-\mathbb{E}f\left(\boldsymbol{x}^\star_{r,\delta_\star}\right)}{\varepsilon}$ we obtain:

$$\mathbb{E}f\left(\boldsymbol{x}_T\right)-\mathbb{E}f\left(\boldsymbol{x}^\star_{r,\delta_\star}\right)\leq\frac{\varepsilon}{2}+\frac{5}{4}\frac{\eta^2}{\beta}\sigma^2\cdot\sum_{k=0}^{T-1}\left(1-\frac{3}{4}\eta\frac{\alpha}{\beta}\right)^k\leq\frac{\varepsilon}{2}+\frac{5}{4}\frac{\eta^2}{\beta}\sigma^2\cdot\frac{4}{3}\frac{1}{\eta}\frac{\beta}{\alpha}=\frac{\varepsilon}{2}+\frac{5}{3}\frac{\eta}{\alpha}\sigma^2\,.$$

Since we set $\eta=\min\left\{\frac{3}{10}\frac{\varepsilon\alpha}{\sigma^2},1\right\}$, the entire quantity is at most ε, which concludes the proof. □

C.5.3 Reducing Communication by Quantizing Gradients

The approach described in Section C.5.2 maintains quantized weights, but communicating gradients may still be expensive. In this section we show that any reasonable quantization method for gradients can be used to reduce communication, while paying in exchange an increased variance. This trade-off is inherent, as the reduction in the number of bits requires injecting randomness, so as the entropy of the output is not smaller than that of the original message to be communicated.

To do so we use any gradient quantization method Q^g, as long as it is an unbiased estimator for the input it takes, and has bounded variance. Our formal requirements for Q^g are the following.

Definition 11. *We say that a gradient quantization operator Q^g is a (σ_∇, b)-unbiased quantizer if it:*

1. *is an unbiased estimator: $\mathbb{E}\left[Q^g(\boldsymbol{g}\left(\boldsymbol{x}\right))|\boldsymbol{g}\left(\boldsymbol{x}\right)\right]=\boldsymbol{g}\left(\boldsymbol{x}\right)$,*
2. *has bounded variance on the stochastic gradients: $\mathbb{E}\left[\|Q^g(\boldsymbol{g}\left(\boldsymbol{x}\right))-\boldsymbol{g}\left(\boldsymbol{x}\right)\|_2^2|\boldsymbol{g}\left(\boldsymbol{x}\right)\right]\leq\sigma_\nabla^2$,*
3. *requires b bits to communicate $Q^g(\boldsymbol{g}\left(\boldsymbol{x}\right))$.*

By Lemma 5, these requirements are automatically satisfied by our shift-and-round quantization operator Q^w, and we can show that σ_∇ and b are determined by the ℓ_1 norm of $\boldsymbol{g}\left(\boldsymbol{x}\right)$.

Standard Quantization Schemes and Their Communication Cost. Another standard gradient quantization scheme can be obtained by independently rounding each coordinate to one of the neighboring points on the quantization grid, with an appropriate probability. An identical scheme has been previously used in other related works on gradient quantization [AGL+17].

Definition 12 (quantization by flipping a coin)**.** *Let $\delta>0$ be a scalar defining the coarseness of the quantization grid. Let the operator $Q_\delta:\mathbb{R}\to\delta\mathbb{Z}$ defined as*

$$Q_\delta\left(x\right)=\begin{cases}\delta\left\lfloor\frac{x}{\delta}\right\rfloor & \text{with probability } 1-\left(\frac{x}{\delta}-\left\lfloor\frac{x}{\delta}\right\rfloor\right)\\ \delta\left(\left\lfloor\frac{x}{\delta}\right\rfloor+1\right) & \text{with probability } \frac{x}{\delta}-\left\lfloor\frac{x}{\delta}\right\rfloor\end{cases}$$

where $\delta>0$. We apply Q_δ to vectors, with the meaning that it is independently applied to each coordinate.

It is fairly easy to prove that this satisfies very similar properties to those proved for Q^w in Lemma 5, which we quickly prove in Section C.5.4. We notice an important difference

between these two quantization methods. While Q independently quantizes each coordinate, the quantization in Q^w is done dependently across coordinates, and the output is always a vector in $\delta\mathbb{Z}^n + r1$, for a randomly sampled scalar r. Although morally they are quite similar (in fact, the shift after rounding could just as well be ignored, and still have an unbiased extimator), it is important if we want to relate the quality of the final solution to the best set of weights from a reasonably chosen grid. This difference is apparent when trying to provide bounds of the type of Lemma 4, bu this attempt falls through in the case of the Q operator.

As we can naively relate the communication cost of a quantized gradient to its sparsity, it is important to discuss quantitative bounds. In both cases, the sparsity bound depends on the ℓ_1 norm of the quantized vector, and its easy to see that it is tight. By comparison, the bound from [AGL+17] is provided in terms of the ℓ_2 norm of the vector, but pays an additional $\sqrt{n}$ factor, which is suboptimal when the input is analytically sparse. For Q^w and Q, we see that the variance introduced by quantizing a generic vector $\boldsymbol{v}$ is bounded by $\delta \|\boldsymbol{v}\|_1$, while its sparsity is $\|\boldsymbol{v}\|_1 / \delta$. Hence a naive encoding of this quantized gradient requires at most $O\left(\frac{\|\boldsymbol{v}\|_1}{\delta}\left(\ln n + \ln \|\boldsymbol{v}\|_1\right)\right)$ bits of communication.

SGD with Weight and Gradient Quantization. For gradient quantization operators that are unbiased estimators, we can use them as stochastic gradients inside the scheme we derived in Theorem 2. To do so we crucially use the following identity involving conditional variance:

Lemma 13 (Law of total variance)**.** *Given random variables X and Y, one has that*

$$\mathsf{Var}[Y] = \mathbb{E}\left[\mathsf{Var}[Y|X]\right] + \mathsf{Var}\left[\mathbb{E}\left[Y|X\right]\right] .$$

Corollary 14. *Consider a stochastic gradient estimator $\boldsymbol{g}(\boldsymbol{x})$ such that $\mathbb{E}\left[\boldsymbol{g}(\boldsymbol{x})|\boldsymbol{x}\right] = \nabla f(\boldsymbol{x})$ and $\mathbb{E}\left[\|\boldsymbol{g}(\boldsymbol{x}) - \nabla f(\boldsymbol{x})\|_2^2 |\boldsymbol{x}\right] \leq \sigma^2$. Consider (σ_∇, b)-unbiased quantizer (Definition 11). Then*

$$\mathbb{E}\left[Q_\delta\left(\boldsymbol{g}(\boldsymbol{x})\right)|\boldsymbol{x}\right] = \nabla f(\boldsymbol{x}) ,$$

i.e. it is an unbiased estimator for the gradient, and

$$\mathbb{E}\left[\|Q_\delta\left(\boldsymbol{g}(\boldsymbol{x})\right) - \nabla f(\boldsymbol{x})\|_2^2\right] \leq \sigma_\nabla^2 + \sigma^2 .$$

Proof. The fact that the quantized gradient is an unbiased estimator for $\nabla f(\boldsymbol{x})$ follows from the law of total expectation, as

$$\mathbb{E}\left[Q_\delta\left(\boldsymbol{g}(\boldsymbol{x})\right)|\boldsymbol{x}\right] = \mathbb{E}\left[\mathbb{E}\left[Q_\delta\left(\boldsymbol{g}(\boldsymbol{x})\right)|\boldsymbol{x}, \boldsymbol{g}(\boldsymbol{x})\right]\right] = \mathbb{E}\left[\mathbb{E}\left[\boldsymbol{g}(\boldsymbol{x})|\boldsymbol{x}\right]\right] = \nabla f(\boldsymbol{x}) .$$

For the variance, we use Lemma 13 to write:

$$\begin{aligned}
\mathbb{E}\left[\|Q_\delta\left(\boldsymbol{g}(\boldsymbol{x})\right) - \nabla f(\boldsymbol{x})\|_2^2\right] &= \mathsf{Var}\left[Q_\delta\left(\boldsymbol{g}(\boldsymbol{x})\right)\right] \\
&= \mathbb{E}\left[\mathsf{Var}\left[Q_\delta\left(\boldsymbol{g}(\boldsymbol{x})\right)|\boldsymbol{g}(\boldsymbol{x})\right]\right] + \mathsf{Var}\left[\mathbb{E}\left[Q_\delta\left(\boldsymbol{g}(\boldsymbol{x})\right)|\boldsymbol{g}(\boldsymbol{x})\right]\right] \\
&\leq \sigma_\nabla^2 + \mathsf{Var}\left[\boldsymbol{g}(\boldsymbol{x})\right] \\
&= \sigma_\nabla^2 + \sigma^2 .
\end{aligned}$$

□

Finally, combining Theorem 2 with Corollary 14, we obtain the final result from Corollary 3.

C.5.4 Deferred Proofs

Proof of Lemma 5

Proof. For both the mean and variance computation, it suffices to prove these bounds for the scalar operator.

We note that by definition $Q^w_{r,\delta}(x) - r = Q^w_{0,\delta}(x-r) = \delta \cdot Q^w_{0,1}\left(\frac{x-r}{\delta}\right) := \delta \cdot \left\lfloor \frac{x-r}{\delta} \right\rceil$. Also let $\{x\} = x - \lfloor x \rfloor$ denote the fractional part of x. We can easily verify that for any scalar $0 \le z < 1$, we have

$$\mathbb{E}_{u \sim \mathsf{Unif}([-1/2,1/2))} \left[\lfloor z + u \rceil \right] = z \,. \tag{C.1}$$

This is because $\lfloor z + u \rceil = 1$ if and only if $z + u \ge 1/2$ i.e. $u \ge \frac{1}{2} - z$, which happens with probability z. Now we can express the expectation of $Q^w_{r,\epsilon}(x)$ as follows:

$$\begin{aligned}
\mathbb{E}\left[Q^w_\delta(x)\right] &= \mathbb{E}_r\left[Q^w_{0,\delta}(x-r) + r\right] \\
&= \mathbb{E}_r\left[Q^w_{0,\delta}(x-r)\right] + \mathbb{E}_r[r] \\
&= \mathbb{E}_r\left[Q^w_{0,\delta}\left(\delta\left\lfloor\frac{x}{\delta}\right\rfloor + \delta\left\{\frac{x}{\delta}\right\} - r\right)\right] + \mathbb{E}_r[r] \\
&= \delta\left\lfloor\frac{x}{\delta}\right\rfloor + \mathbb{E}_r\left[Q^w_{0,\epsilon}\left(\delta\left\{\frac{x}{\delta}\right\} - r\right)\right] + \mathbb{E}_r[r] \\
&= \delta\left\lfloor\frac{x}{\delta}\right\rfloor + \mathbb{E}_r\left[\delta \cdot Q^w_{0,1}\left(\left\{\frac{x}{\delta}\right\} - \frac{r}{\delta}\right)\right] + \mathbb{E}_r[r] \,.
\end{aligned}$$

In the last line we used the fact that $Q^w_{0,\delta}(y) = \delta \cdot Q^w_{0,1}\left(\frac{y}{\delta}\right)$. Now we reparameterize by using $u := r/\delta$, where $r \sim \mathsf{Unif}([-1/2, 1/2))$. This allows to write the term in the middle as:

$$\mathbb{E}_r\left[\delta \cdot Q^w_{0,1}\left(\left\{\frac{x}{\delta}\right\} - \frac{r}{\delta}\right)\right] = \delta \cdot \mathbb{E}_{u \sim \mathsf{Unif}([-1/2,1/2))}\left[Q^w_{0,1}\left(\left\{\frac{x}{\delta}\right\} - u\right)\right] = \delta \cdot \left\{\frac{x}{\delta}\right\},$$

were we used (C.1). Plugging back in we obtain that

$$\mathbb{E}\left[Q^w_{r,\delta}(x)\right] = \delta\left\lfloor\frac{x}{\delta}\right\rfloor + \delta \cdot \left\{\frac{x}{\delta}\right\} + 0 = x \,.$$

Next we compute the scalar variance:

$$\begin{aligned}
\mathbb{E}\left[\left(Q^w_\delta(x) - x\right)^2\right] &= \mathbb{E}_r\left[\left(Q^w_{0,\delta}(x-r) - x\right)^2\right] \\
&= \mathbb{E}_r\left[\left(\delta\left\lfloor\frac{x}{\delta}\right\rfloor + \delta \cdot Q^w_{0,1}\left(\left\{\frac{x}{\delta}\right\} - \frac{r}{\delta}\right) - x\right)^2\right] \\
&= \mathbb{E}_r\left[\delta^2 \cdot \left(\left\lfloor\frac{x}{\delta}\right\rfloor + Q^w_{0,1}\left(\left\{\frac{x}{\delta}\right\} - \frac{r}{\delta}\right) - \frac{x}{\delta}\right)^2\right] \\
&= \mathbb{E}_{u \sim \mathsf{Unif}([-1/2,1/2))}\left[\delta^2 \cdot \left(\left\lfloor\frac{x}{\delta}\right\rfloor + Q^w_{0,1}\left(\left\{\frac{x}{\delta}\right\} - u\right) - \frac{x}{\delta}\right)^2\right] \\
&= \delta^2 \cdot \mathbb{E}_{u \sim \mathsf{Unif}([-1/2,1/2))}\left[\left(Q^w_{0,1}\left(\left\{\frac{x}{\delta}\right\} - u\right) - \left\{\frac{x}{\delta}\right\}\right)^2\right] .
\end{aligned}$$

Now we use the fact that for any scalar $0 \le z < 1$ one has that

$$\mathbb{E}_{u \sim \mathsf{Unif}([-1/2,1/2))}\left[\left(\lfloor z + u \rceil - z\right)^2\right] = z^2 \,.$$

This follows from the fact that $\lfloor z+u \rceil = 1$ iff $u \geq 1/2 - z$, which happens with probability z, and makes the expectation equal to

$$\int_{-1/2}^{1/2-z} z^2 du + \int_{1/2-z}^{1/2} (1-z)^2 du = z(1-z) ,$$

which leads us to

$$\mathbb{E}\left[\left(Q_\delta^w(x) - x\right)^2\right] = \delta^2 \cdot \left\{\frac{x}{\delta}\right\}\left(1 - \left\{\frac{x}{\delta}\right\}\right) ,$$

which gives us what we needed.

Finally, for the sparsity bound, let us understand when a single scalar gets rounded to zero (before shifting back by r). We have that for $x \in \mathbb{R}$,

$$\begin{aligned}
&\mathbb{P}\left[Q_{r,\delta}^w(x) - r = 0\right] = \mathbb{P}\left[Q_{r,1}^w\left(\frac{x}{\delta}\right) - r = 0\right] = \begin{cases} \int_{-1/2}^{1/2} \mathbf{1}_{\lfloor \frac{x}{\delta} - r \rceil = 0} dr, & |x| < \delta, \\ 0, & \delta \leq |x| , \end{cases} \\
&= \begin{cases} \int_{-1/2}^{1/2} \mathbf{1}_{-\frac{1}{2} \leq \frac{x}{\delta} - r \leq \frac{1}{2}} dr, & |x| < \delta, \\ 0, & \delta \leq |x| , \end{cases} = \begin{cases} \int_{-1/2}^{1/2} \mathbf{1}_{\frac{x}{\delta} - \frac{1}{2} \leq r \leq \frac{x}{\delta} + \frac{1}{2}} dr, & \left|\frac{x}{\delta}\right| < 1, \\ 0, & 1 \leq \left|\frac{x}{\delta}\right| , \end{cases} \\
&= \min\left\{\frac{x}{\delta} + \frac{1}{2}, \frac{1}{2}\right\} - \max\left\{\frac{x}{\delta} - \frac{1}{2}, -\frac{1}{2}\right\} = \frac{1}{2} + \min\left\{\frac{x}{\delta}, 0\right\} - \left(-\frac{1}{2} + \max\left\{\frac{x}{\delta}, 0\right\}\right) \\
&= 1 + \min\left\{\frac{x}{\delta}, 0\right\} - \max\left\{\frac{x}{\delta}, 0\right\} = 1 - \left|\frac{x}{\delta}\right| .
\end{aligned}$$

which shows that

$$\mathbb{E}\left[\left\|Q_\delta^g(\boldsymbol{v})\right\|_0\right] = \sum_{i=1}^n \left(1 - \mathbb{P}\left[Q_\delta^g(v_i) = 0\right]\right) = \sum_{i=1}^n \begin{cases} \left|\frac{v_i}{\delta}\right| , & |v_i| < \delta, \\ 1, & \delta \leq |v_i| , \end{cases} \leq \|\boldsymbol{v}\|_1 / \delta .$$

This concludes the proof. □

Proof of Lemma 4

Proof. It suffices to prove this coordinate-wise. From Lemma 5 we have that for any $x \in \mathbb{R}$,

$$\mathbb{E}\left[\left(Q_\delta^w(x) - x\right)^2\right] = \delta^2 \left(1 - \left\{\frac{x}{\delta}\right\}\right)\left\{\frac{x}{\delta}\right\}$$

and similarly for $\delta_\star$. Let $k = \delta_\star / \delta$. Then

$$\mathbb{E}\left[\left(Q_{\delta_\star}^w(x) - x\right)^2\right] = k^2\delta^2 \left(1 - \left\{\frac{x/\delta}{k}\right\}\right)\left\{\frac{x/\delta}{k}\right\}$$

Applying the inequality from Lemma 6, we conclude that

$$\mathbb{E}\left[\left(Q_\delta^w(x) - x\right)^2\right] = \delta^2 \left(1 - \left\{\frac{x}{\delta}\right\}\right)\left\{\frac{x}{\delta}\right\} \leq \delta^2 \cdot k \left(1 - \left\{\frac{x}{k\delta}\right\}\right)\left\{\frac{x}{k\delta}\right\} = \frac{1}{k}\mathbb{E}\left[\left(Q_{\delta_\star}^w(x) - x\right)^2\right] .$$

Applying this bound to all coordinates we obtain

$$\mathbb{E}\left[\left\|Q_\delta^w(\boldsymbol{x}) - \boldsymbol{x}\right\|_2^2\right] \leq \frac{\delta}{\delta_\star}\mathbb{E}\left[\left\|Q_{r,\delta_\star}^w(\boldsymbol{x}) - \boldsymbol{x}\right\|_2^2\right] .$$

Also since $Q_{r,\delta_\star}^w$ rounds to the nearest point in $\delta_\star\mathbb{Z} + r$, clearly $\left\|Q_{r,\delta_\star}^w(\boldsymbol{x}) - \boldsymbol{x}\right\|_2^2 \leq \left\|\boldsymbol{x}_{r,\delta_\star}^\star - \boldsymbol{x}\right\|_2^2$ for all r. Taking expectations on both sides and combining with the previous inequality concludes the proof. □

Proof of Lemma 7

Proof. Using smoothness we have

$$\begin{aligned} f(\boldsymbol{x}+\boldsymbol{\Delta}) &\leq f(\boldsymbol{x}) + \langle \nabla f(\boldsymbol{x}), \boldsymbol{\Delta} \rangle + \frac{\beta}{2} \|\boldsymbol{\Delta}\|_2^2 \\ &= f(\boldsymbol{x}) + (1-\eta) \langle \nabla f(\boldsymbol{x}), \boldsymbol{\Delta} \rangle - \frac{\eta^2}{2\beta} \|\nabla f(\boldsymbol{x})\|_2^2 + \left(\frac{\eta^2}{2\beta} \|\nabla f(\boldsymbol{x})\|_2^2 + \eta \langle \nabla f(\boldsymbol{x}), \boldsymbol{\Delta} \rangle + \frac{\beta}{2} \|\boldsymbol{\Delta}\|_2^2 \right) \\ &= f(\boldsymbol{x}) + (1-\eta) \langle \nabla f(\boldsymbol{x}), \boldsymbol{\Delta} \rangle - \frac{\eta^2}{2\beta} \|\nabla f(\boldsymbol{x})\|_2^2 + \frac{\beta}{2} \left\| \frac{\eta}{\beta} \nabla f(\boldsymbol{x}) + \boldsymbol{\Delta} \right\|_2^2 . \end{aligned}$$

□

Proof of Lemma 8

The proof is standard and can be found in literature, such as [KNS16]. However, for completeness we reproduce it here.

Proof. Let $g(\boldsymbol{x}) = \sqrt{f(\boldsymbol{x}) - f^*}$ for which we have

$$\nabla g(\boldsymbol{x}) = \frac{1}{2\sqrt{f(\boldsymbol{x}) - f^*}} \nabla f(\boldsymbol{x}) \ .$$

Using the α-PL condition we have

$$\|\nabla g(\boldsymbol{x})\|^2 = \frac{1}{4(f(\boldsymbol{x}) - f^*)} \cdot \|\nabla f(\boldsymbol{x})\|^2 \geq \frac{1}{2(f(\boldsymbol{x}) - f^*)} \cdot \alpha \cdot (f(\boldsymbol{x}) - f^*) = \frac{\alpha}{2} \ .$$

Now starting at some $\boldsymbol{x}_0$, we consider the dynamic $\dot{\boldsymbol{x}} = -\nabla g(\boldsymbol{x})$. We see that this always decreases function value until it reaches some $\boldsymbol{x}_T$ for which $\nabla g(\boldsymbol{x}_T) = 0$ and hence by the PL inequality, $\boldsymbol{x}_T$ is a minimizer i.e. $f(\boldsymbol{x}_T) = f^*$. Now we can write

$$\begin{aligned} g(\boldsymbol{x}_T) &= g(\boldsymbol{x}_0) + \int_0^T \langle \nabla g(\boldsymbol{x}_t), \dot{\boldsymbol{x}}_t \rangle \, dt = g(\boldsymbol{x}_0) + \int_0^T \langle \nabla g(\boldsymbol{x}_t), -\nabla g(\boldsymbol{x}_t) \rangle \, dt \\ &= g(\boldsymbol{x}_0) - \int_0^T \|\nabla g(\boldsymbol{x}_t)\|^2 \, dt \ . \end{aligned}$$

Thus

$$g(\boldsymbol{x}_0) - g(\boldsymbol{x}_T) = \int_0^T \|\nabla g(\boldsymbol{x}_t)\|^2 \, dt \geq \sqrt{\frac{\alpha}{2}} \cdot \int_0^T \|\nabla g(\boldsymbol{x}_t)\| \, dt = \sqrt{\frac{\alpha}{2}} \cdot \int_0^T \|\dot{\boldsymbol{x}}_t\| \, dt \ ,$$

where we used our lower bound on the norm of $\nabla g(\boldsymbol{x})$. Finally, we use the fact that the last integral lower bounds the total movement of $\boldsymbol{x}$ as it moves from $\boldsymbol{x}_0$ to $\boldsymbol{x}_T$. Thus

$$\int_0^T \|\dot{\boldsymbol{x}}_t\| \, dt \geq \|\boldsymbol{x}_0 - \boldsymbol{x}_T\| \ ,$$

so

$$g(\boldsymbol{x}_0) - g(\boldsymbol{x}_T) \geq \sqrt{\frac{\alpha}{2}} \|\boldsymbol{x}_0 - \boldsymbol{x}_T\| \ ,$$

which enables us to conclude that

$$f(\boldsymbol{x}_0) - f^* \geq \frac{\alpha}{2} \|\boldsymbol{x}_0 - \boldsymbol{x}_T\|^2 \ ,$$

where $\boldsymbol{x}_T$ is some global minimizer of f. This concludes the proof. □

Bound for Quantization by Coin Flip

Lemma 15. *Let $\boldsymbol{v} \in \mathbb{R}^n$, and let $\delta > 0$, and let Q_δ be the quantization operator from Definition 12. Then,*

$$\begin{aligned}
\mathbb{E}\left[Q_\delta\left(\boldsymbol{v}\right)\right] &= \boldsymbol{v}\,, \\
\mathbb{E}\left[\left\|Q_\delta\left(\boldsymbol{v}\right) - \boldsymbol{v}\right\|_2^2\right] &= \delta^2 \cdot \sum_{i=1}^{n} \left\{\frac{v_i}{\delta}\right\}\left(1 - \left\{\frac{v_i}{\delta}\right\}\right)\,, \\
\mathbb{E}\left[\left\|Q_\delta\left(\boldsymbol{v}\right)\right\|_0\right] &\leq \left\|\boldsymbol{v}\right\|_1 / \delta\,.
\end{aligned}$$

Proof. For the expectation and variance, it suffices to prove that these bound holds coordinate-wise. Let $x \in \mathbb{R}$, and write $x = \delta\left(\left\lfloor \frac{x}{\delta}\right\rfloor + \left\{\frac{x}{\delta}\right\}\right)$ so that

$$\begin{aligned}
\mathbb{E}\left[Q_\delta\left(x\right)\right] &= \mathbb{E}\left[Q_\delta\left(\delta\left(\left\lfloor \frac{x}{\delta}\right\rfloor + \left\{\frac{x}{\delta}\right\}\right)\right)\right] \\
&= \delta\left\lfloor \frac{x}{\delta}\right\rfloor + \mathbb{E}\left[Q_\delta\left(\delta\left\{\frac{x}{\delta}\right\}\right)\right] \\
&= \delta\left\lfloor \frac{x}{\delta}\right\rfloor + \mathbb{E}\left[Q_\delta\left(\delta\left\{\frac{x}{\delta}\right\}\right)\right] \\
&= \delta\left\lfloor \frac{x}{\delta}\right\rfloor + \delta \cdot \left\{\frac{x}{\delta}\right\} \\
&= x\,.
\end{aligned}$$

Similarly we write the variance as:

$$\begin{aligned}
\mathbb{E}\left[\left(Q_\delta\left(x\right) - x\right)^2\right] &= \mathbb{E}\left[\left(Q_\delta\left(\delta\left\{\frac{x}{\delta}\right\}\right) - \delta\left\{\frac{x}{\delta}\right\}\right)^2\right] \\
&= \left(1 - \left\{\frac{x}{\delta}\right\}\right)\left(\delta\left\{\frac{x}{\delta}\right\}\right)^2 + \left\{\frac{x}{\delta}\right\} \cdot \left(\delta - \delta\left\{\frac{x}{\delta}\right\}\right)^2 \\
&= \delta^2\left(\left(1 - \left\{\frac{x}{\delta}\right\}\right)\left\{\frac{x}{\delta}\right\}^2 + \left\{\frac{x}{\delta}\right\} \cdot \left(1 - \left\{\frac{x}{\delta}\right\}\right)^2\right) \\
&= \delta^2\left(1 - \left\{\frac{x}{\delta}\right\}\right)\left\{\frac{x}{\delta}\right\}\,,
\end{aligned}$$

For the sparsity bound, we need to understand when a single scalar gets rounded to zero. We have that for $x \in \mathbb{R}$,

$$\mathbb{P}\left[Q_\delta\left(x\right) = 0\right] = \begin{cases} 1 - \left|\frac{x}{\delta}\right|, & |x| < \delta, \\ 0, & \delta \leq |x|\,, \end{cases}$$

which shows that

$$\begin{aligned}
\mathbb{E}\left[\left\|Q_\delta\left(\boldsymbol{v}\right)\right\|_0\right] &= \sum_{i=1}^{n}\left(1 - \mathbb{P}\left[Q_\delta\left(v_i\right) = 0\right]\right) \\
&= \sum_{i=1}^{n} \begin{cases} \left|\frac{v_i}{\delta}\right|, & |v_i| < \delta, \\ 1, & \delta \leq |v_i|\,, \end{cases} \\
&\leq \left\|\boldsymbol{v}\right\|_1 / \delta\,.
\end{aligned}$$

□