ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

Department of Computer Science and Engineering

Second Cycle Degree in Artificial Intelligence

Efficient Distributed Learning with PowerSGD

Supervisors
Dott. Thijs Vogels
Prof. Martin Jaggi
Prof. Zeynep Kiziltan

Candidate
Omar G. Younis

Academic Year 2023-2024

Contents

Acknowledgements

This dissertation owes its completion to the invaluable contributions of numerous people. It is with sincere gratitude that I acknowledge each of them here:

- Thijs Vogels deserves special recognition as one of the smartest people I have had the privilege to meet. He provided friendly supervision throughout the entire project, yet the knowledge I gained from him goes far beyond what is written in this dissertation.
- Martin Jaggi for hosting me, for supporting my work financially, and for creating and coordinating a wonderful environment.
- Zeynep Kiziltan for the comprehensive writing guidance, and for accepting being the local supervisor of this dissertation despite my unusual situation.
- Can Balioglu from Meta for guiding me in contributing to PyTorch and reviewing my code.
- Scott Gray from OpenAI for sharing his extraordinary knowledge of CUDA programming.

Lastly, I extend my heartfelt gratitude to all individuals who have been part of my life. Thank you.

Abstract

Deep learning models are becoming more complex and require a lot of computational resources, which are often only available by combining multiple devices such as GPUs. However, distributing the workload to these devices poses many engineering challenges. One of the main challenges is ensuring that communication between the devices is fast enough, even when scaling on multiple nodes. Our experimental study shows that, in some scenarios, communication becomes the major bottleneck, and adding more devices makes the training slower instead of faster. To address this issue, researchers have proposed compression algorithms that are designed to reduce the size of data that needs to be communicated (usually gradient) while retaining as much information as possible. For the compression algorithm to be effective, it must be sufficiently fast to save time in communication, and the compression must be accurate enough that it doesn't negatively affect training.

Our dissertation focuses on the state-of-the-art gradient compression algorithm, PowerSGD, and how we have improved it in both speed and accuracy. Specifically, we made compression around 20 times faster, making it effective for wider use cases. We also improved the accuracy of the gradient compression without affecting speed, which led to the training process converging twice as fast compared to standard PowerSGD, for the scenario we tested on. To make these improvements available to the community, we contributed to PyTorch 1.11 and published all the code used for the experiments and our improved PowerSGD versions on GitHub[1][2].

[1]https://github.com/younik/powersgd-cuda

[2]https://github.com/younik/async-optim

Chapter 1

Introduction

Recently, there has been a significant shift towards training increasingly large deep learning models, utilizing vast datasets [1, 2]. This trend demands substantial computational resources involving the use of expensive hardware. For instance, the training of GPT-4 reportedly used around 25 000 A100 GPUs, incurring a cost of approximately \$60 million in computational expenses and taking nearly 100 days to complete[1]. Such figures underscore the importance of efficient training methodologies for these massive models.

A key area of research in this context is distributed learning, which explores strategies for training models across multiple devices, predominantly GPUs [3]. Scaling up a model from a single device to many is a complex task, requiring careful consideration of numerous design elements [4]. In this dissertation, we focus on distributed data-parallel, a specific parallelism paradigm, where the batch data is distributed across devices to speed up forward and backward passes. During the training process, it is essential to communicate the gradients between devices to obtain an average of them. he gradient has the same size as the weights, and for large models, billions of floating-point numbers need to be communicated at each training step. As the model scales to many devices, communication becomes slower and often becomes a significant bottleneck, leading to devices idling while waiting for the communication cycle to complete [5]. In such scenarios, the speedup gained by distributing the workload across multiple nodes is lower than the increased communication time resulting from increasing the number of nodes, making the distributed settings ineffective at speeding up training.

To address this challenge, researchers have proposed various solutions, such as compression algorithms [5, 6, 7], which we focus on in this dissertation. These are a category of functions designed to reduce the size of data that needs to be communicated while retaining as much information as possible, thus speeding up the training process by alleviating the communication bottleneck. For these algorithms to be effective, they need

[1]https://www.semianalysis.com/p/gpt-4-architecture-infrastructure

to be fast enough to ensure that the time saved during communication outweighs the time spent in compressing the data. At the same time, the compression process mustn't negatively affect the learning process and performance. For instance, SIGNSGD [5] is one of the simplest compression methods, which communicates only the sign of each floating-point value. Although it typically reduces the data by a factor of 32 and results in faster training, the accuracy of compression is very low, usually hurting the training performances. Another solution studied in the literature is TOP-K [7], which communicates only the higher k floating-point numbers of the gradient. Although it achieves a great approximation accuracy for proper value k, the compression algorithm is usually too slow for most scenarios. The state-of-the-art compression algorithm is POWERSGD [8], which approximates the gradient using low-rank decomposition. It uses the power iteration algorithm to converge to the low-rank approximation, making it faster during compression.

In this dissertation, we improve POWERSGD, in both compression speed and accuracy. Firstly, we have analyzed PyTorch's POWERSGD implementation, finding the orthogonalization process as a major bottleneck during compression. We have studied different orthogonalization techniques and developed a custom CUDA kernel that exploits the typical shape of the matrix to orthogonalize in POWERSGD. We thus improved the PyTorch implementation of POWERSGD, resulting in a 20-fold increase in the speed of the compression process. We have incorporated these changes into the PyTorch codebase, making them available to the library users from version 1.11 onwards.

Moreover, we have made significant improvements to the POWERSGD algorithm, which has enhanced the compression process's accuracy without affecting its speed. During our analysis, we observed that the compression error norm increased while training, indicating that we could improve the compression accuracy further. To address this, we experimented with several algorithmic variants to reduce the compression error. We found two promising solutions. The first uses the local compression error to update the weights, which we named Nesterov error feedback due to its similarity to the Nesterov momentum. The second variant takes advantage of the idle communication channel during forward and backward passes. As a result, we can communicate information without affecting training speed. By communicating part of the error feedback during these stages, we achieved 2x faster training convergence compared to standard POWERSGD. We provide detailed information on the algorithm variations in Section 4.1.2 and have made all implementations and experiments open-source.

1.1 Dissertation organization

We organized the dissertation as follows:

Background: In Chapter 2 we introduce fundamental concepts essential for comprehending our contributions. This includes a concise overview of deep learning, with a particular focus on distributed settings. Additionally, we cover the fundamentals of PyTorch, emphasizing the library's distributed functionalities. To conclude the chapter, we explore two distinct algorithms for matrix orthogonalization, providing a foundational understanding of their application in our research.

Related works: In Chapter 3, we provide a detailed overview of relevant research papers that contextualize our work. This includes other compression algorithms, and theoretical insights on PowerSGD improvements. Additionally, we present a research paper that uses PowerSGD to train a model on 64 GPUs, which provides practical engineering tricks.

Contributions: In Chapter 4, we present the specifics of our contributions, structuring the chapter into two main sections: methods and experimental results. The methods section is dedicated to outlining our enhancements to PowerSGD, and explaining several innovative variants that we have conceptualized. In the experimental results section, we construct a detailed model that illustrates how the timing of various training phases scales with the expansion of node numbers in a typical distributed setting. This is followed by a comprehensive benchmarking of our implementations. To conclude, we analyze our developed variants against the standard PowerSGD algorithm, demonstrating the efficacy and results of our improvements.

Conclusions: In the concluding chapter 5 of the dissertation, we provide a summary of our findings, emphasizing their significance and relevance. Furthermore, we propose several avenues for future research, identifying gaps in the current knowledge that our work has not covered.

Appendix: Finally, in the appendix, we provide a comprehensive introduction to parallel programming using CUDA. Additionally, we delve into the details of PyTorch, explaining its internal mechanisms. The chapter concludes with a detailed guide on integrating custom operations into PyTorch.

Chapter 2

Background

We present here key concepts necessary for understanding our contributions such as distributed deep learning and the PyTorch distributed package.

2.1 Deep learning

Deep learning is a subset of machine learning inspired by the structure and function of the brain. With the goal of learning complex functions, it has revolutionized our approach to complex problems in various fields. For example, deep learning plays a crucial role in guiding drug discovery[9], powering self-driving vehicles[10], assisting in climate prediction models[11], optimizing energy use in power grids[12], and improving quality assurance processes in manufacturing lines[13][14].

At the core of machine learning is the concept of learning from data. This approach involves training algorithms on vast datasets, allowing them to learn and make predictions or decisions based on patterns they discern. Thus, unlike traditional programming, where rules are explicitly set by humans, machine learning algorithms develop their own rules by analyzing and interpreting data, making it applicable to complex problems where hand-crafting rules are unfeasible.

Deep learning specifically uses artificial neural networks to process the data. We will formalize neural networks in section 2.1.3. This chapter aims to introduce the fundamental concepts behind deep learning, starting from the basics of learning and delving into gradient-based optimization for deep learning.

2.1.1 What is learning?

We formalize here the concept of learning in the context of machine learning. We start by defining the concepts of hypothesis space.

Hypothesis space: the set of functions that the learning algorithm can choose from to solve a given problem. Mathematically, this set can be denoted as $\mathcal{H}$, where each function $h \in \mathcal{H}, h : X \to Y$ maps inputs in the set X to outputs in the set Y, depending on the data and goal. For example, in image classifications, the inputs are the images while the outputs are the labels.

Data distribution: refers to the theoretical probability distribution of the data. In the case of the classification task, it is the probability distribution $P(X, Y)$, where X represents the features or inputs and Y represents the outputs or labels.

Loss function: A function $L : Y \times Y \to \mathbb{R}$ that provide a distance measure in the Y space.

With these definitions, we can define learning as the process of finding the $h \in \mathcal{H}$ that approximates the true distribution of data as closely as possible. Formally, the goal is to find h^*:

$$h^* = \operatorname*{argmin}_{h \in \mathcal{H}} \mathbb{E}_{x,y \sim P} \left[L(h(x), y) \right] \tag{2.1}$$

However, $P(X, Y)$ is unknown, but we have at our disposal a dataset $D \sim P(X, Y)$. Thus a **learning algorithm** is a function $A : D \to \mathcal{H}$. The algorithm takes the dataset D as input and outputs a hypothesis $h \in \mathcal{H}$ that ideally minimizes the expected value of equation 2.1.

One may think of minimizing the sum of the losses on the dataset:

$$h^* = \operatorname*{argmin}_{h \in \mathcal{H}} \frac{1}{|D|} \sum_{x,y \in D} L(h(x), y) \tag{2.2}$$

however, when $\mathcal{H}$ is a sufficiently broad set, this can *overfit* the dataset with the result of an h function that doesn't generalize well, i.e. have poor performances on new data, even with perfect performances on D. For this reason, we usually divide D in two different datasets, one used for the learning algorithm and one to have an unbiased estimation of the expected performances of our hypothesis. Moreover, to avoid the problem of overfitting the dataset, we should restrict the set $\mathcal{H}$ using various techniques that are out of the scope of this dissertation.

In summary, learning in AI is about finding a function within a hypothesis space that best approximates the true data distribution, as measured by some loss function. This is achieved through a learning algorithm, which adjusts the model in response to the data it is exposed to, aiming for good generalization while avoiding overfitting or underfitting.

2.1.2 Gradient-based learning

Gradient-based optimization[15] is the key in training many machine learning algorithms. It refers to a set of techniques that use gradient information to minimize or maximize an objective function, typically the loss function in the context of machine learning. As a learning algorithm, the process aims to find a good performing $h \in \mathcal{H}$ on a given task.

For simplicity, from this chapter we will restrict ourselves to the case $\mathcal{H}$ is a parameterized function $h(x, \theta)$. This is indeed an infinite set of functions mapping from X to Y. To make it clearer the role of x as input and θ as parameter, we will write $h(x, \theta)$ as $h_\theta(x)$.

The gradient of a function at a point is a vector that points in the direction of the steepest ascent. In machine learning, we are interested in minimizing the loss, thus in the steepest descent, which can be proved to be the negative of the gradient. The gradient is calculated with respect to the parameters of the model, and it provides the direction to adjust the parameters θ to reduce the loss.

$$\nabla_\theta \mathbb{E}_{x,y\sim P}\left[L(h_\theta(x), y)\right] = \mathbb{E}_{x,y\sim P}\left[\nabla_h L(h_\theta(x), y)\nabla_\theta h_\theta(x)\right] \tag{2.3}$$

As done for equation 2.2, we can estimate the gradient averaging the gradient of the losses on the training dataset. While the basic gradient descent algorithm updates parameters using the entire dataset (batch gradient descent), this can be computationally expensive for large datasets. Two popular variants address this issue:

Stochastic Gradient Descent (SGD): Instead of using the entire dataset, SGD updates parameters using a single data point at a time for estimating the gradient. This introduces more variance in the parameter updates but can lead to faster convergence on large datasets.

Mini-batch Gradient Descent: This approach strikes a balance between batch and stochastic gradient descent by using a small, randomly selected subset of the data (a mini-batch) for each update. It is the most commonly used variant in practice.

With an estimation of the gradient, the basic idea is straightforward: iteratively adjust the parameters in the direction that most reduces the loss. The updates to the parameters at each iteration are governed by the equation:

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \nabla_\theta J(\theta), \tag{2.4}$$

where η is the learning rate, and $\nabla_\theta J(\theta)$ is the estimation of the gradient of equation 2.3.

The learning rate η is a crucial hyperparameter. A learning rate that is too high can cause the algorithm to overshoot the minimum and possibly diverge, while a rate that is too low can lead to slow convergence.

Gradient-based optimization faces several challenges, such as avoiding local minima, dealing with the vanishing or exploding gradient problem, and choosing an appropriate learning rate. Advanced techniques such as momentum, adaptive learning rates (e.g., Adam, RMSprop), and second-order methods (e.g., Newton's method) have been developed to address these challenges, making the optimization process more effective and efficient. Even if often used in practice, these methods are out of scope of this short introduction on gradient-based optimization.

2.1.3 Artificial neural networks

Deep learning[16] uses artificial neural network as a function space. They are composed of interconnected units called neurons, which are organized into layers. We explained here, specifically the feedforward neural network architecture, which for simplicity doesn't contemplate for recursive connection. Thus, each layer processes information and passes it on to the next layer. We have three main types of layers:

Input Layer (Layer 1): This layer takes the raw input data. Each neuron in this layer corresponds to a feature in the input data. Thus, the number of neurons in this layer is fixed by the type of data.

Hidden Layers (Layers 2 to N-1): These layers perform the bulk of the computation in a neural network. Each neuron in a hidden layer receives inputs from the previous layer, applies a transformation, and passes the result to the next layer. The number of hidden layers and the number of neurons in each layer can be chosen arbitrary. Increasing it increase the size of the hypothesis space $\mathcal{H}$.

Output Layer (Layer N): The final layer produces the network's output. Again, the number of neurons for this layer depends by the task.

Specifically, let a_1 be the input of the network, thus the value of the first layer. Every hidden layer i, process the input in this way:

$$z_i = W_i \cdot a_{i-1} + b_i \qquad \forall i = 2, ..., N \tag{2.5}$$

$$a_i = \sigma(z_i) \qquad \forall i = 2, ..., N-1 \tag{2.6}$$

Where W_i and b_i are, respectively, a matrix and a vector of parameters and σ is a non-linearity function which must be differentiable almost everywhere. The output layer process the a_{N-1} only linearly with equation 2.5.

There is a lot more to discuss regarding artificial neural networks, and numerous books have been dedicated to this topic. For the purpose of this dissertation, we will concentrate on what is important to compression algorithm. Therefore, in the next section, we will provide a more detailed explanation of the gradient of neural networks.

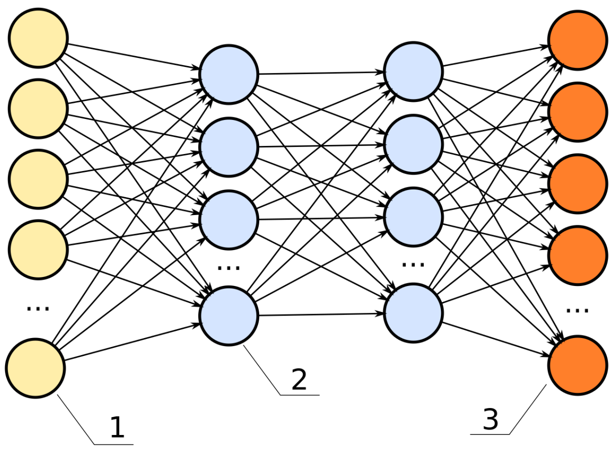

Figure 2.1. Representation of a neural network. In yellow, the neurons of the input layer (1), in orange the output layer (3), and in blue the hidden layers (2). The network is fully connected, i.e. each neuron is connected to every neuron of the next layer.

2.1.4 Computing the gradient of a neural network

To apply gradient descent algorithm on artificial neural networks, we need to compute the gradient of the loss with respect to the parameters. Thus, given the chain rule of gradient, as showed in equation 2.3, we need to compute the gradient of the neural network output with respect to its parameters $\nabla_\theta h_\theta(x)$. Specifically, the gradient of the output of each layer with respect ot its parameters is:

$$\frac{\partial a_i}{\partial \theta_i} = \frac{\partial a_i}{\partial z_i}\frac{\partial z_i}{\partial \theta_i} \qquad \forall i = 2, ..., N-1 \tag{2.7}$$

$$\frac{\partial a_i}{\partial z_i} = \sigma'(z_i) \qquad \forall i = 2, ..., N-1 \tag{2.8}$$

$$\frac{\partial z_i}{\partial \theta_i} = \left[\frac{\partial z_i}{\partial W_i}, \frac{\partial z_i}{\partial b_i}\right] = [a_{i-1},\ 1] \qquad \forall i = 2, ..., N \tag{2.9}$$

Notice that for the output layer, there is no activation function, thus we only need $\frac{\partial z_N}{\partial \theta_N}$. However, we are interested in $\frac{\partial z_N}{\partial \theta_i}$, for every layer i, since we need to compute the gradient over the output. Again, we can use chain rule to derive it:

$$\frac{\partial z_N}{\partial \theta_i} = \frac{\partial z_N}{\partial \theta_N}\left(\frac{\partial \theta_N}{\partial a_{N-1}}\frac{\partial a_{N-1}}{\partial z_{N-1}}\frac{\partial z_{N-1}}{\partial \theta_{N-1}}\right) ... \left(\frac{\partial \theta_{i+1}}{\partial a_i}\frac{\partial a_i}{\partial z_i}\frac{\partial z_i}{\partial \theta_i}\right) \qquad \forall i = 2, ..., N-1 \tag{2.10}$$

As we can see from equation 2.10, to compute $\frac{\partial z_N}{\partial \theta_i}$, we can reuse computation of $\frac{\partial z_N}{\partial \theta_{i+1}}$. Thus, to compute the gradients efficiently with respect to all parameters, we may start from computing the gradient of the last layer N, proceed with $N - 1$ reusing the previous computation and move backwards until the first layer. For this reason, this algorithm is known as **backpropagation**[17].

In general, neural networks can be more intricate than the fully connected explained here. Thus, computing the gradient may become very hard; we want an automatic way for doing it. Nowadays, several programming library exists that automate the process. At their core, they use a computational graph, an essential concept in deep learning. At its core, a computational graph is a representation of mathematical operations involved in the neural network. Each node in the graph represents an operation or variable, while edges depict the dependency between these operations. Every atomic operation has a gradient function associated; thus with the gradient rules we can compute the gradient of the whole computational graph. We call this process automatic differentiation or *AutoGrad*[18].

2.1.5 Training phases

We aim here to summarize the phases of training a neural network.

Forward propagation: in forward propagation, random sampled input data is fed into the neural network. Each neuron computes a weighted sum of its inputs and applies an activation function to produce an output. This process continues layer by layer until the final output is generated.

Loss computation: the network's performance is evaluated by a loss function, which quantifies the difference between the predicted output and the actual target values. Common loss functions include mean squared error for regression tasks and cross-entropy for classification tasks.

Backward propagation: we then compute the gradient of the loss function with respect to each weight, by using the chain rule of calculus and propagating backward through the network. The gradient identifies how changes to weights impact the loss.

Weight updating: Finally, the weights are updated to minimize the loss. This is typically done using optimization algorithms like stochastic gradient descent. The learning rate, a hyperparameter, determines the size of the weight updates.

All these steps are looped several times and constitute the learning process.

2.2 Distributed learning

With the increasing size of datasets and complexity of models, the field of deep learning has become more computational demanding. To satisfy the computational needs, we need to efficiently exploit multiple devices, such as GPUs. Distributed learning is a paradigm that involves dividing the learning process across multiple computational nodes. This approach is essential for handling large-scale data and complex models that are beyond the capacity of a single machine. In fact, distributed learning primarily arise from two challenges: memory and time. As models and datasets grow larger, it becomes impractical to store them on a single machine. Similarly, processing datasets with models becomes slower as we scale them up. Distributed learning addresses these challenges by leveraging the power of multiple machines, enabling the handling of larger datasets and models.

2.2.1 Data parallelism

Data parallelism is a distributed learning strategy where the input data is partitioned across multiple machines. Each machine holds a slice of the batch sampled from the dataset and a complete copy of the model. During the backward pass phase, each device computes gradients based on its subset of data. These gradients are then aggregated across all machines to update the model. Averaging the gradients produce the gradient of the average losses (as in single device scenario), for linearity of the gradient operator. This approach is particularly effective for large datasets as it allows parallel processing of data, speeding up the training process.

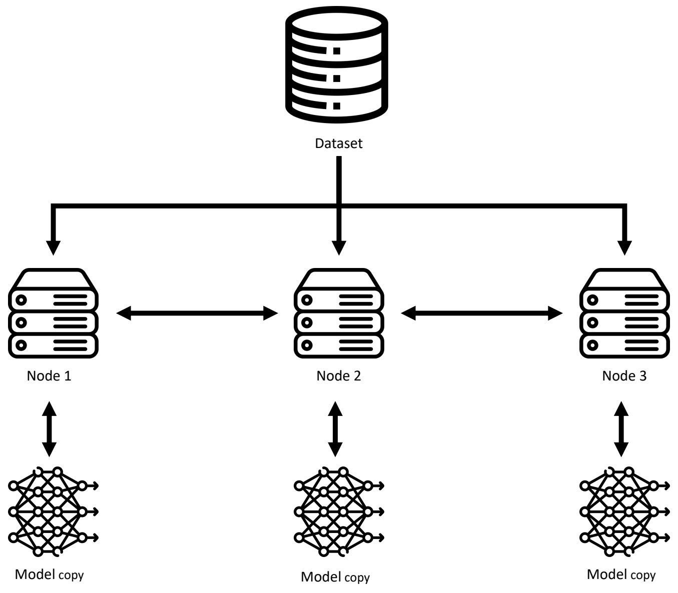

Figure 2.2. Illustration of the distributed learning scenario. *Icons from flaticon.com*

2.2.2 Model parallelism

In contrast to data parallelism, model parallelism involves splitting the model itself across multiple nodes. Each node is responsible for a portion of the model, processing different layers or parts of the neural network. This approach is useful for very large models that cannot fit into the memory of a single machine. We have two primary methods:

Horizontal parallelism: This involves dividing each tensor into several slices. Rather than storing the entire tensor on a single node, each fragment (or shard) of the tensor is allocated to a specific node. Each shard is processed independently and concurrently across different nodes. The outcomes from each shard are then synchronized at the end of the process. This method became more popular with big transformers architectures, as multi-head attention provide a natural way to shard the weights.

Vertical parallelism: In this approach, the model is split along its vertical axis, which corresponds to its layers. The division is such that only one or a few layers of the model are assigned to each node. This setup allows each node to process different stages of the pipeline concurrently, handling a smaller portion of the batch. This is called 'vertical parallelism' because the splitting is based on the layers of the model. Since every layer needs the output of the preceding one as input, this method doesn't fully exploit parallelism of the nodes . However, more advanced techniques are used to increase throughput of this method.

2.2.3 Synchronous vs. asynchronous methods

In distributed learning, synchronization refers to how the data or model updates are shared across the machines. We divide the methods in two categories: synchronous and asynchronous.

In **synchronous** distributed learning, all machines must complete their portion of the task before any model update is performed. This is the most common scenario as it ensures consistency in the model updates but can lead to idle time, as all machines must wait for the slowest one.

Asynchronous methods allow machines to update the model independently without waiting for others. This approach can speed up the training process but may lead to stability issues and problems when updates are made based on outdated information. However, this may speed up considerably training in settings where nodes has different update speed. This family of approaches are particularly prominent in reinforcement learning, with algorithm such A3C[19] and IMPALA[20].

2.2.4 Communication primitives

We explain here basics operations that are used to communicate data across different nodes.

Broadcast involves a single source node sending data to all other nodes in the network. In distributed learning, broadcasting is essential when sharing updated model parameters or algorithms from a central server to all worker nodes, ensuring each node operates with the most current information.

Gather is the process where data is collected from all nodes in the network and brought together at a single receiver node. In distributed learning, gathering is often used to accumulate information (like trajectories in reinforcement learning) from various worker nodes to a central server for model updating. The **AllGather** variant ensures that all nodes obtain the information at the end of the process.

Reduce is the communication process where data from all nodes is combined into a single one through a specific operation (like summing or averaging). In the context of distributed learning, reducing is important for aggregating gradients or error terms across nodes. The **AllReduce** variant ensures that all nodes obtain the aggregate data at the end of the process.

Scatter is the opposite of gather. In this method, distinct pieces of data are distributed from one source node to multiple receiver nodes. In distributed learning, scattering is used to distribute subsets of a large dataset or model to different worker nodes for parallel processing.

2.2.5 Federated learning

Federated learning[21] is a specific instance of distributed learning that is worth to mention for its interesting applications. It differs from other distributed learning approaches for its focus on privacy. In this settings, we have multiple nodes controlled by different actors, each with a different dataset that must remain private and not shared with others. However, the final goal is still to train a global model that can leverage the joint datasets. This makes federated learning a challenging tasks as it has to deal with non identically independently distributed and unbalanced data while preventing data leakage during the learning process and ensuring the robustness of the aggregated model against potential adversarial attacks. Some promising applications include:

Healthcare: Federated learning allows hospitals and research institutions to collaborate on developing predictive models without sharing sensitive data about their patients.

Finance: Banks and financial institutions can use federated learning to detect fraud and manage risk while maintaining the confidentiality of client data.

Smartphones: Companies like Google use federated learning to train vision and language models without uploading personal data to their servers.

2.2.6 Compression algorithms

As introduced earlier in this section, training a model in a distributed setting requires to communicate the gradient to the other nodes multiple times. The gradient of a model has the same size of the model, thus big models requires a large bandwidth to share the gradients. This can be a bottleneck in training loops, slowing down the whole learning process. When this is the case, a compression algorithm can be used.

Formally, a compression algorithm is a function $C : X \to Y$, where X is the space of the data that we want to compress and Y is the resulting space. We also define a decompression function associated with C, $D_C : Y \to X$. To be useful, compression and decompression algorithms should satisfy the following conditions:

Quality of reconstruction: The compression data must Introduce a metric to evaluate the quality of compression. More formally, the reconstruction error must be below a threshold, depending on the applications:

$$||x - D_C(C(x))|| < \epsilon \tag{2.11}$$

Efficiency: The algorithm should be efficient in terms of computation. Specifically, if the time spent to compress the data is larger than the time saved to communicate, the compression algorithm becomes ineffective.

Unbiasedness of reconstruction: For many applications we need the reconstruction to be unbiased. More formally:

$$\mathbb{E}_P \left[D_C(C(x))\right] = x \tag{2.12}$$

where P is the probability distribution that generated x.

2.2.7 PowerSGD

We present now a specific compression algorithm named PowerSGD[8] that will be the focal point of this dissertation.

The main idea of PowerSGD is to compress the gradient using low-rank approximation. More formally, for a matrix A of size (n, m), the k-rank approximator, is the matrix A_k of size same size such that:

$$A_k = \underset{X}{\operatorname{argmin}} ||A - X||_F, \textit{ with rank}(X) = k \tag{2.13}$$

where $||.||_F$ is the Frobenius norm.

A common method to achieve low-rank approximation is using singular value decomposition to decompose the matrix A as follows:

$$A = U\Sigma V^T \tag{2.14}$$

Where:

- $U \in \mathbb{R}^{nxn}$ is an orthogonal matrix containing the left singular vectors.
- $\Sigma \in \mathbb{R}^{nxm}$ is a diagonal matrix with singular values $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r$.
- $V \in \mathbb{R}^{mxm}$ is an orthogonal matrix containing the right singular vectors.

To approximate A, we take the first k largest singular values and the corresponding singular vectors:

$$A_k = U_k \Sigma_k V_k^T \tag{2.15}$$

Where:

- U_k contains the first k columns of U.
- Σ_k is a diagonal matrix with the top k singular values.
- V_K^T contains the first k rows of V^T.

However, singular value decomposition can be very slow to compute for big matrices, as its computational complexity is in $O(\min(nm^2, n^2m))$.

PowerIteration

The power iteration algorithm is an algorithm used to approximate the dominant eigenvalue and the corresponding eigenvector of a matrix. It's particularly useful when dealing with large matrices where methods like singular value decomposition are not feasible, since it has time complexity in $O(n + m)$. Moreover, it can better exploit parallelization in hardware for linear algebra operations like GPUs.

Assumptions: The matrix A has a dominant eigenvalue, i.e., there exists an eigenvalue λ_1 such that $|\lambda_1| > |\lambda_i|$ for all $i \neq 1$. Moreover, the initial guess for the eigenvector $\mathbf{q}_0$ is not orthogonal to the dominant eigenvector.

Algorithm Steps:

1. Start with a normalized random vector $\mathbf{q}_0 \in \mathbb{R}^m$.
2. For some iterations $i = 1, 2, 3, \ldots$:

    2a Compute $\mathbf{p}_i = A\mathbf{q}_{i-1}$.

2b Normalize $\mathbf{p}_i$: $\mathbf{p}_i = \frac{\mathbf{p}_i}{\|\mathbf{p}_i\|}$.

2c Compute $\mathbf{q}_i = A^T \mathbf{p}_i$

2d Normalize $\mathbf{q}_i$: $\mathbf{q}_i = \frac{\mathbf{q}_i}{\|\mathbf{q}_i\|}$.

3. Return $\mathbf{q} = \mathbf{q}_i$ as the approximation of first eigenvector of A, and $\mathbf{p} = \mathbf{p}_i$ as the approximation of first eigenvector of A^T.

The rate of convergence depends on the ratio $\frac{|\lambda_2|}{|\lambda_1|}$, where λ_1 and λ_2 are, respectively, the first and the second dominant eigenvalue. Moreover, notice that to estimate the second eigenvector, it is sufficient the first eigenvector component from the matrix and repeat the algorithm with the new matrix, i.e. $A^{(2)} = A - \lambda_1(\mathbf{p} \cdot \mathbf{q}^T)$, where λ_1 can be estimated as $\lambda_1 = \|A\mathbf{q}\|$.

PowerSGD

The key idea of POWERSGD is to exploit PowerIteration to efficiently aggregate data (usually gradients) in distributed learning. Specifically, let's have L nodes, each of them having a local matrix $A_0, A_1, ..., A_L$. Our goal is to approximate the sum (or the mean): $A = A_0 + A_1 + ... + A_L$. We present here two variants of POWERSGD.

Variant 1

1. Start with a random vector $\mathbf{q}_0 \in \mathbb{R}^m$, via *broadcast*, each node obtains this vector.

2. Each node initializes a local approximation of A, $\tilde{A} = \mathbf{0}$

3. For some iterations $i = 1, \dots, k$:

   3a Each node normalizes $\mathbf{q}_i$: $\mathbf{q}_i = \frac{\mathbf{q}_i}{\|\mathbf{q}_i\|}$.

   3b Each node l computes $\mathbf{p}_i^{(l)} = A_l \mathbf{q}_{i-1}$.

   3c Using *AllReduce*, every node get $\mathbf{p}_i = \mathbf{p}_i^{(0)} + ... + \mathbf{p}_i^{(L)}$.

   3d Each node normalizes $\mathbf{p}_i$: $\mathbf{p}_i = \frac{\mathbf{p}_i}{\|\mathbf{p}_i\|}$.

   3e Each node l computes $\mathbf{q}_i^{(l)} = A_l^T \mathbf{p}_i$

   3f Using *AllReduce*, every node get $\mathbf{q}_i = \mathbf{q}_i^{(0)} + ... + \mathbf{q}_i^{(L)}$.

   3g Each node adds the current approximation to the local buffer: $\tilde{A} = \tilde{A} + \mathbf{p}_i \cdot \mathbf{q}_i^T$.

   3h Each node l removes the current approximation from the matrix to compress: $A_l = A_l - \mathbf{p}_i \cdot \mathbf{q}_i^T$.

4. Return $\tilde{A}$ as the approximation of $A_0 + A_1 + ... + A_L$.

Variant 2

1. Start with a random matrix $\mathbf{Q} \in \mathbb{R}^{m \times k}$, via *broadcast*, each node obtains this vector.
2. Each node orthogonalize the matrix $\mathbf{Q}$.
3. Each node l computes $\mathbf{P}^{(l)} = A_l\mathbf{Q}$.
4. Using *AllReduce*, every node get $\mathbf{P} = \mathbf{P}^{(0)} + ... + \mathbf{P}^{(L)}$.
5. Each node orthogonalize $\mathbf{P}$.
6. Each node l computes $\mathbf{Q}^{(l)} = A_l^T\mathbf{P}$.
7. Using *AllReduce*, every node get $\mathbf{Q} = \mathbf{Q}^{(0)} + ... + \mathbf{Q}^{(L)}$.
8. Return $\mathbf{P} \cdot \mathbf{Q}^T$ as the approximation of $A_0 + A_1 + ... + A_L$.

In both algorithms k adjust the precision of the compression at the cost of slowing the algorithm. This compression algorithm reduce the communication data, since every nodes sends only $n \times k + m \times k$ numbers instead of $n \times m$ and n, m are typically very large numbers, while k is a small constant.

Both variants work for linearity of matrix multiplication. For example, in *Variant 1*, at step *2c*:

$$\mathbf{p}_i = \mathbf{p}_i^{(0)} + ...\mathbf{p}_i^{(L)} = A_0\mathbf{q}_{i-1} + ...+ = A_L\mathbf{q}_{i-1} = (A_0 + ... + A_L)\mathbf{q}_{i-1} \tag{2.16}$$

making the algorithm equivalent to a *PowerIteration* step on the matrix $A_0 + ... + A_L$.

Warm start

To enhance the accuracy of the PowerSGD approximation, the algorithm *warm-start* the initial value of $\mathbf{Q}$, i.e. it uses the last $\mathbf{Q}$ value from previous iteration to initialize the current one. Remember that the compression algorithm is employed repeatedly during training, at each backward propagation phase, as detailed in section 2.1.5. The rationale behind this method is based on the premise that gradients change gradually throughout the training process. Therefore, the eigenvectors from one training step to the next are likely to have only minor differences. Consequently, we can effectively use the $\mathbf{Q}$ value from the preceding step as the starting point for the subsequent step. If the gradient has changed little compared to previous step, starting with the last $\mathbf{Q}$ is similar to continuing from where the *PowerIteration* algorithm left off in the previous step, leading to a more precise approximation.

Error feedback

The PowerSGD compression algorithm is biased:

$$\mathbb{E}\left[\mathcal{D}(\mathcal{C}(A_0 + ... + A_L))\right] \neq A_0 + ... + A_L \tag{2.17}$$

where $\mathcal{C}$ and $\mathcal{D}$ represent, respectively, the compression and decompression operators. For this reason, error feedback is used in PowerSGD, resulting in the following algorithm:

Compression with error feedback

1. Each node l initialize the error e_l as zero.
2. At every iteration $i = 1, 2, \ldots$ of the training loop:

   2a Each node l adds the error to the gradient to sum: $A_l = A_l + e_l$.

   2b PowerSGD is used, obtaining $\tilde{A}$.

   2c The error is updated as $e_l = A_l - \tilde{A}$.

The use of error-feedback makes PowerSGD unbiased leading to better optimization outcomes when using PowerSGD to compress gradients in a learning algorithm.

Analysis

We report here some analysis from the original paper showing the effectiveness of PowerSGD. Specifically, we show in Table 1.1 a comparison of test accuracy between standard SGD and PowerSGD. In Table 1.2, we show the effectiveness of *warm-start* comparing PowerSGD compression algorithm to SVD low-rank approximation. For both experiments, the authors used a ResNet18 on the CIFAR10 dataset. On the same task, we show the effectiveness of error feedback on the final accuracy in figure 2.3.

Table 2.1. PowerSGD (variant 2) test accuracy compared to plain SGD learning algorithm. PowerSGD with $k = 2$ matches SGD on test accuracy.

Algorithm	Accuracy	Data
SGD	94.3%	1023 MB
Rank-1 PowerSGD	93.6%	4 MB
Rank-2 PowerSGD	94.4%	8 MB

Table 2.2. SVD rank-2 approximation vs. PowerSGD. Warm-start improves test accuracy, even matching the performance of SVD approximation.

Algorithm	Accuracy
Best approximation	94.4%
Warm start (default)	94.4%
Without warm start	94.0%

2.3 PyTorch

PyTorch[22] is a powerful, open-source machine learning library for Python. It is widely adopted for its flexibility, ease of use, and as a valuable tool in both research and development of deep learning models. PyTorch stands out for its dynamic computation graph and efficient memory usage. Moreover, PyTorch provides seamless GPU

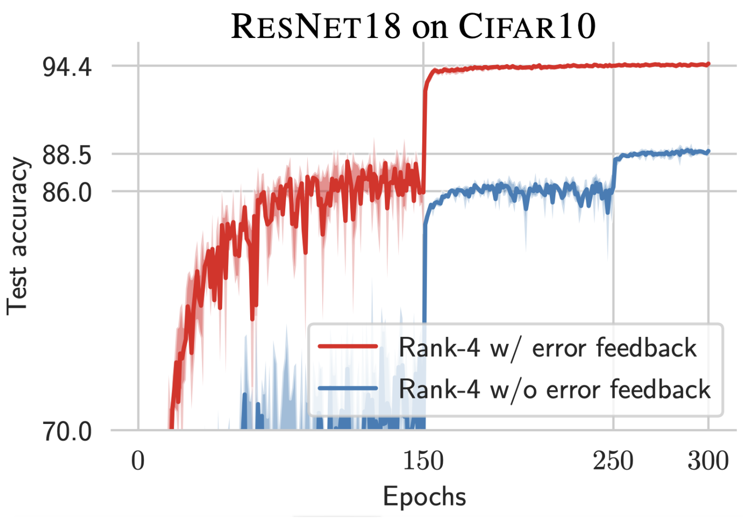

Figure 2.3. The comparison between PowerSGD with error feedback and without it reveals significant differences. Although rank-4 PowerSGD with error feedback reaches the same test accuracy as full-precision SGD, PowerSGD without error feedback fails to achieve comparable accuracy. Notably, both experiments employed the identical learning rate, which was specifically optimized for full-precision SGD.

integration, enabling faster computations and model training.

The fundamental unit of PyTorch is the tensor. Tensors in PyTorch are array-like data structures that are able to run on multiple devices, notably CPUs and GPUs. Running tensor on GPU accelerates the computation of many linear algebra operations, as GPU are specialized hardware for that. Tensors can store data of various types and support a wide range of operations, making them ideal for neural networks.

Another important concept is the automatic differentiation capabilities. PyTorch provides automatic differentiation for all operations on tensors, making gradient computation straightforward. Importantly the computational graph is built at runtime, letting the computation be dynamic. PyTorch keeps track of all operations on tensors for which gradients are to be calculated.

Installing PyTorch

To install PyTorch we can use the Python package manager:

```
pip install torch
```

Tensor creation

We show here how to create new tensors specifying the values to fill and generating them randomly.
After importing :

```
import torch
```

To create a tensor from a list:

```
tensor = torch.tensor([1, 2, 3])
```

To create a 2×3 tensor filled with zeros:

```
zeros = torch.zeros(2, 3)
```

To create a 2×3 tensor filled with random numbers

```
random_tensor = torch.rand(20, 30)
```

Access elements of tensors

We can access element to the tensors using indexing syntax:

```
random_tensor[0, 2]  # access element at row 0 and column 2
random_tensor[0]  # access first row
random_tensor[:, 0]  # access first column
random_tensor[1:6, 0:2]  # slice a submatrix
```

Operations on tensors

We can use the basic arithmetic operations on tensor, like element-wise multiplication:

```
result = tensor * tensor
```

Or linear algebra operations like the dot product:

```
result = torch.dot(tensor, random_tensor)
```

Moreover tensor can be reshaped:

```
reshaped = random_tensor.view(30, 20)
```

However, notice that the resulting tensor point to the same data of the original tensor in memory. This means that changing the data *inplace* will affect also the other tensor data.

2.3.1 Autograd: automatic differentiation

PyTorch's provides automatic differentiation for all operations on tensors. In this way, computing the gradient for tensors is straightforward:

```
# Create a tensor and enable autograd
tensor = torch.tensor([1., 2., 3.], requires_grad=True)

# Perform some operations
y = torch.dot(tensor, tensor)

# Compute gradients
y.backward()

# Gradients are stored in the '.grad' attribute
print(tensor.grad)
```

2.3.2 Building neural networks

PyTorch provides a module to help create and train neural networks:

```
import torch.nn as nn
import torch.nn.functional as F

class SimpleNet(nn.Module):
    def __init__(self):
        super(SimpleNet, self).__init__()
        self.fc1 = nn.Linear(256, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

2.3.3 Training the network

Training a network typically involves a forward pass, loss computation, backward pass, and updating the model parameters, as explained in 2.1.5. PyTorch ease updating the model parameters providing some common optimizers, like SGD:

```
# Example of a training loop
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
loss_function = nn.MSELoss()
model = SimpleNet()
train_loader = torch.utils.data.DataLoader(...)
```

```
for data, target in train_loader:
    optimizer.zero_grad()
    output = model(data)
    loss = loss_function(output, target)
    loss.backward()
    optimizer.step()
```

2.3.4 Distributed package

PyTorch provides a distributed package, which is essential for distributed training of models across multiple nodes. Moreover, it abstracts many of the complexities of distributed computing, making it accessible to developers who may not be experts in parallel computing.
We highlight some important features:

Distributed data parallel: This allows the distribution of data across multiple nodes, ensuring that each node processes a subset of the data. This unlock data parallelism resulting in reduction in training time as the workload is shared across multiple processing units.

Model parallelism: Apart from data parallelism, PyTorch also supports model parallelism. In fact, the model can be split across multiple nodes, allowing for the training of very large models.

Multiple backend support: PyTorch's distributed package supports and abstract multiple backends like *NCCL*, *Gloo*, and *MPI*. This variety allows users to choose the most suitable backend based on their hardware and specific needs.

Collective operations: PyTorch's distributed package supports the common collective primitives listed in section 2.2.4 as easy-to-use functions.

Communication hooks: Finally, PyTorch provides algorithms for compressing data and an easy API to compress gradients while training a model.

We show here a simple example on using PyTorch distributed package. This code should be run in every nodes, with the correct number vlaue for *rank* (node id) and *world size*:

```
import torch

torch.distributed.init_process_group("nccl", rank=..., world_size=...)

tensor = torch.rand(2, 3)
```

```
result = torch.distributed.all_reduce(tensor)
# now result contains the sum of tensor among all nodes

model = SimpleNet()
model = nn.parallel.DistributedDataParallel(model)
# now the model can be trained in the distributed setting using the same code as s
```

2.3.5 PyTorch's PowerSGD

PowerSGD is implemented in the PyTorch's distributed package using the second variant algorithm. Thanks to the library abstraction, using PowerSGD for training is straightforward:

```
import torch
import torch.nn as nn
import torch.distributed.algorithms.ddp_comm_hooks.powerSGD_hook

torch.distributed.init_process_group("nccl", rank=..., world_size=...)

model = SimpleNet()
model = nn.parallel.DistributedDataParallel(model)
state = powerSGD_hook.PowerSGDState(
    process_group=None,
    matrix_approximation_rank=1,
    start_powerSGD_iter=100,
)
model.register_comm_hook(state, powerSGD_hook.powerSGD_hook)
# now the model can be trained seamlessly using PowerSGD!
```

We outline some key features of PyTorch implementation:

- The implementation uses the Gram-Schmidt method for orthogonalizing matrices.
- The algorithm selectively compresses matrices, acting only on those which meet a pre-defined threshold for minimum compression rate.
- The implementation doesn't compress gradients during the initial iterations, as these are crucial and sensitive to the final performance.
- By default, the implementation compresses the gradient of each layer separately. However, an alternative approach is also available: it flattens and stacks the network's gradients into a single matrix before compression. Although this method may be faster, it does not leverage specific gradient properties, potentially leading to a decrease in accuracy.

2.4 Orthogonalization algorithms

2.4.1 Gram-Schmidt

The Gram-Schmidt algorithm is a method used in linear algebra for orthonormalizing a set of vectors in a space. We focus on the application of the method to orthogonalize a matrix.

Specifically, in order to orthogonalize the matrix A:

1. For every row a_i of matrix A:

   1a For every previous row a_j (if any), subtract the projection: $a_i = a_i - (a_i \cdot a_j)a_j$

   1b Normalize $a_i = \frac{a_i}{\|a_i\|}$

   At the end of the process, the matrix A will be orthogonalized:

$$AA^T = I \tag{2.18}$$

Notice that the algorithm is inherently sequential, as each step of the loop depends on the outcomes of the previous steps.

2.4.2 QR decomposition

QR factorization decomposes a given matrix A into the product of two matrices, Q and R:

$$A = QR \tag{2.19}$$

where Q is an orthogonal matrix and R is an upper triangular matrix.

Householder reflection

A Householder reflection is a linear transformation that reflects a vector with respect to a hyperplane. The transformation can be represented by a Householder matrix H, which is defined as:

$$H = I - 2vv^T \tag{2.20}$$

where I is the identity matrix, and v is a unit vector orthogonal to the reflection hyperplane.

Construction of the householder matrix

1. Given a vector x in the matrix we wish to transform, and a target vector e, v is chosen as $x - \lambda e$, where λ is the norm of x if x and e are not collinear, and $-\lambda$ otherwise. This choice of v ensures that the reflection sends x to a scalar multiple of e.

2. Normalize v so that $v^T v = 1$. This is important for the matrix H to be orthogonal.

Application in QR decomposition

1. Start with the matrix A that we want to decompose.
2. For each column i of A:
   - 2a Construct a Householder matrix H_i that zeroes out all elements below the diagonal.
   - 2b Apply H_i to A to get a new matrix A_i, i.e., $A_i = H_i A_{i-1}$ where $A_0 = A$.
   - 2c After applying this process to each column, the matrix A is transformed into an upper triangular matrix R.
3. The product of the Householder matrices H_i gives the orthogonal matrix Q, i.e., $Q = H_1 H_2 ... H_n$.
4. The final QR decomposition is $A = QR$, where Q is orthogonal, and R is upper triangular.

Chapter 3

Related work

We present in this chapter some works that are related this dissertation.

3.1 Compression algorithms

We present here the most popular compression algorithms studied in the literature. We also provide a visual comparison of these algorithms in Figure 3.1.

SignSGD[5]: this method focuses on compressing data based on the sign (positive or negative) of each data element. It is straightforward and fast, making it time-efficient for large gradients. However, it is a biased compressor. Moreover, the quality of compression can be low depending on the gradient.

SignSGD + norm[6]: this approach extends the SignSGD algorithm by also considering the norm (magnitude) of the whole gradient. Thus, during reconstruction, the matrix of signs is multiplied by the L_1-norm. In this way, this algorithm is more informative than the basic SignSGD, leading to better data representation.

Top-K algorithm[7]: this algorithm selects the top-K elements based on the magnitude for compression, ensuring that the most significant elements are retained, often leading to high-quality data reconstruction. However, it is a biased compression algorithm.

Random K algorithm[7]: It is the natural unbiased counterpart of the top-K algorithm. Instead of selecting the top K elements, it randomly selects K elements. However, the randomness can lead to the omission of significant data elements, affecting the quality of compressed data.

Random block algorithm[7]: This method is very similar to the previous one. However, it is more efficient as it accesses a contiguous block of memory.

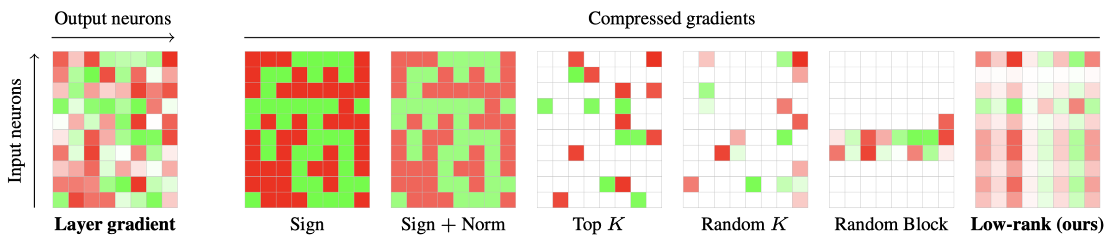

Figure 3.1. Representation of different compression algorithms.
Image from PowerSGD paper by Vogels et al.

PowerSGD[8]: introduced by Vogels et al., this compression algorithm approximates the gradient with a low-rank approximation. This approach is explained in detail in Section 2.2.7 and it generally provides a more precise gradient estimation.

3.2 The gradient is limited in rank

Guy Gur-Ari et al. explore the dynamics of gradient descent in deep learning, demonstrating that during the training of large-scale deep learning models, the gradient converges rapidly to a very small subspace[23]. This subspace is defined by a few top eigenvectors, which interestingly corresponds to the number of classes in the dataset, for the case of classification task. The paper argues that gradient descent predominantly occurs within this small subspace.

Within this subspace, the gradient does not exhibit any special properties and appears randomly oriented with respect to the eigenvector basis. Furthermore, they observe that the top eigenvectors do not significantly mix with the bulk eigenvectors, even over extended periods of training. This implies that the top subspace remains relatively stable during training.

The findings suggest that PowerSGD is effective in approximating the gradient for two main reasons. Firstly, the gradient typically exists within a limited subspace, allowing it to be accurately approximated with a low-rank representation. Secondly, the stability of this subspace during training justifies the effectiveness of the *warm-start* technique. Since with this technique, we begin with an initial estimation of the eigenvectors, and the subspace doesn't change during training, this initial estimation remains relevant and useful throughout the process. However, this also suggests that the error may accumulate on the subspace orthogonal to the approximation, which can hurt learning. This motivates the study on error buffer we perform in Section 4.1.2.

3.3 Error feedback

Karimireddy et al. study the error feedback technique that we explained in Section 2.2.7[24]. This general technique is used to address the limitations of biased compression algorithms, specifically SIGNSGD in the paper. SIGNSGD has issues with convergence and generalization due to the biased nature of the sign compression operator. By incorporating the error made by the compression operator into the next step, the paper overcomes these issues. The authors prove that adding error feedback to SIGNSGD achieves the same rate of convergence as standard SGD without additional assumptions. Extensive experiments confirm that error feedback improves both convergence and generalization performance. The use of error-feedback and its properties motivate the POWERSGD variants we introduce in Section 4.1.2.

3.4 DALL-E: practical use of PowerSGD

The paper introducing DALL-E model[25] from OpenAI uses POWERSGD to train the big model in their distributed setting.

They customize POWERSGD to improve efficiency, which inspired the work for this thesis. We highlight here some practical techniques they used:

Same buffer for error and gradient: In the paper, the gradient is accumulated directly in the error buffer of POWERSGD, resulting in a significant memory saving (remember that the gradient size, thus the error buffer size, is the same as the model size). Since the gradient buffer is usually zeroed in training loops, this is not done in the PyTorch implementation, where two different buffers are used, one for the gradient and one for the error.

Parallelism with backward propagation: As we discussed in the section 2.10, backward propagation calculates the gradient sequentially, starting from the last layer and moving to the first. To speed up the training process, the POWERSGD compression technique is employed as soon as the gradient for a particular layer becomes available. This approach is particularly beneficial in scenarios involving model parallelism, as it optimizes the use of resources. In fact, even when the entire backward pass has not been completed, some GPUs may remain underutilized or idle. Moreover, also communication happens per layer, as soon as compression is done, without waiting to send the whole gradient.

Orthogonalization algorithm: Unlike PyTorch, which employs the Gram-Schmidt algorithm for orthogonalization, they opted for the Householder algorithm. This choice yields two significant benefits. Firstly, the Gram-Schmidt process is inherently sequential and thus slower, especially when compared to the Householder algorithm on hardware capable of parallelizing tasks, such as GPUs. Secondly, the Householder algorithm boasts greater stability, thereby reducing numerical problems. To further mitigate this issue, they incorporated an additional step of adding ϵI to the matrix being orthogonalized, where ϵ represents a small value and I is the identity matrix.

Chapter 4

Contributions

We present in this chapter the main contributions of our work.

4.1 Novel approaches

We start by defining the different algorithmic modification we tried and we analyze them in the next chapter.

4.1.1 Optimizing PowerSGD implementation

We optimized PyTorch's PowerSGD implementation by changing the orthogonalization technique from Gram-Schmidt to QR factorization. This is based on the reasonable hypothesis that a device capable of parallel operations, such as a GPU, is available. However, since the QR factorization doesn't support some data types, for these cases the orthogonalization method falls back to Gram-Schmidt. Moreover, as detailed in Section 4.2.2, our analysis shows that Gram-Schmidt is still faster for orthogonalizing a matrix with 2 columns or rows. Since this is a common case for PowerSGD when doing 2-rank approximation, we also treat this specific case using the Gram-Schmidt algorithm.

These updates have been integrated into PyTorch starting from version 1.11.

Orthogonalization kernel

To further improve performances, we develop a custom CUDA kernel for orthogonalization using the Householder method. This kernel is specifically tailored for PowerSGD, thus using assumptions of having as input a skewed matrix with m rows and r columns, with $r << m$.

We published the implementation with benchmarking scripts on GitHub[1].
We highlight here some features of the custom CUDA kernel:

- We devide the algorithm in two kernels: REFLECTIONS and Q__LOOP. The first one computes the reflection vectors v, and the second one computes the final orthogonalization result Q.
- Both kernels have a dynamic number of threads per block depending on the number of columns r. Since the number of blocks needed must be known at compile-time, we compile 3 versions with 256, 512, and 1024 blocks and we select the one to run at runtime.
- Each block of the REFLECTIONS kernel computes one reflection vector v.
- Each block of the Q__LOOP kernel computes one row of the final matrix Q using the reflection vectors v computed by the previous kernel.
- To calculate the reflection vector v_j, it is essential that row j is first updated using all preceding reflection vectors v_i, where $i < j$. Therefore, a synchronization method is required between blocks within the REFLECTIONS kernel. We have developed a specialized barrier system for each block to maximize throughput and guarantee the correctness of the result.
- To improve the efficiency of the reduction operation within a block, we ensure that threads in the same *warp* access to contiguous elements, even when the number of threads is less than the number of elements.

4.1.2 PowerSGD variants

We introduce here several variants of PowerSGD that we have conceptualized, implemented, and studied. These variants are inspired by the hypothesis that error feedback might accumulate and become outdated as training progresses. This led us to question: does error feedback indeed accumulate and become obsolete over the course of training? If so, how does this impact performance? Can we develop better algorithms to mitigate this potential issue?

All variants are implemented in PyTorch; we publish their implementation on GitHub[2].

Decaying error feedback

These variants address the issue of obsolete gradient by applying a decay factor λ. Thus, for a user-defined $\lambda \in [0, 1]$, the algorithm is updated as follows:

[1]https://github.com/younik/powersgd-cuda

[2]https://github.com/younik/async-optim

1. Each node l, initialize as zero the error e_l.

2. At every iteration $i = 1, 2, \ldots$ of the training loop:

   2a Each node l adds the error to the gradient to sum: $A_l = A_l + \lambda e_l$.

   2b PowerSGD is used, obtaining $\tilde{A}$.

   2c The error is updated as $e_l = A_l - \tilde{A}$.

Nesterov error feedback

The basic idea behind this variant is to allow different nodes to temporarily diverge on the model before computing the gradient. More specifically, before computing the gradient, each node l moves the local weights of the model towards the local error of the previous iteration:

1. User defines λ, each node l initialize as zero the error e_l.

2. At every iteration $i = 1, 2, \ldots$ of the training loop:

   2a Move towards the negative error feedback: $w_l = w - \lambda e_l$

   2b Apply forward and backward passes using the set of weights w_l

   2c Restore original weights: $w = w_l + \lambda e_l$ (w equal for each node)

   2d Average gradient using PowerSGD, obtaining new e_l

   2e Update weights with the gradient

Notice that this method computes the gradient over a different set of weights for each node. This may be an interesting behavior to study as can potentially enhance generalization, steering the final model toward convergence at a flat minimum. It is widely recognized that flat minima tend to offer superior generalization capabilities, see for example Hochreiter et al. [26].

Asynchronous communication

We also implement some variants with the scope of exploiting the communication channel while it is idle during the other phases of training. In fact, communication can happen in parallel while the nodes are busy with other computation tasks. Thus, in this variant, after each node receives the compressed version of the gradient, it also communicates the uncompressed error feedback buffer. Since no compression is applied to the error feedback, its transmission over the network requires a considerable amount of time. However, we do not wait for the communication to end until the next iteration, where we need error feedback. Also, notice that in this case, the error buffer will be the same for all nodes. Therefore, it could be integrated after communication with *PowerSGD*. However, to eliminate the need for an additional buffer, this step is not undertaken. In summary, the algorithm executes the following steps:

1. Each node l, initialize as zero the error e_l.
2. At every iteration $i = 1, 2, \ldots$ of the training loop:

   2a If $i \neq 1$, wait for the error buffer e_l to be ready.

   2b Each node l adds the error to the gradient to sum: $A_l = A_l + e_l$.

   2c PowerSGD is used, obtaining $\tilde{A}$.

   2d The error is updated as $e_l = A_l - \tilde{A}$.

   2e *AllReduce* on e_l is started as asynchronous operation

We also developed a similar variant where we refined the process by transmitting only a portion of the error feedback instead of the entire buffer. To minimize delays during step 2a, the size of the slice is dynamically adjusted. Specifically, if the error buffer is not ready yet, we halve the size of the slice for the subsequent iteration. Moreover, at every iteration, we shift the slice window to ensure communication of the entire error buffer.

4.2 Experimental results

We proceed here to present experimental results.

For our experiments, we utilized the EPFL ICCluster with a single NVIDIA V100 GPU, except for experiments that required multiple nodes. For those, we utilized the Google Cloud Platform, using a V100 per node. For reproducibility purposes, we have published all of our scripts on GitHub.[3].

4.2.1 Timing orthogonalization methods

We start by evaluating various orthogonalization algorithms, a crucial component of the second variant of PowerSGD, which is the implementation of PyTorch. In particular, we assess the performance of the Gram-Schmidt algorithm, PyTorch's QR factorization, and our custom orthogonalization kernel, detailed in Section 4.1.1. We conducted tests across a range of plausible sizes for *PowerSGD*, varying the number of rows from 2 to 128 and the number of columns from 256 to 4096, exploring all powers of two within these ranges. The findings of these evaluations are presented in Table 4.2.1, while the benchmarking scripts are available on GitHub[3].

The custom CUDA kernel we developed significantly outperforms the traditional Gram-Schmidt method, being up to 27 times faster. Moreover, when compared to PyTorch's QR implementation, our kernel demonstrates an astounding improvement, operating

[3]https://github.com/younik/powersgd-cuda

up to 300 times more rapidly. Luckily, starting with version 1.9, PyTorch has enhanced the efficiency of their QR implementation, aligning it with the speed of our custom kernel, for big matrices. Consequently, to accelerate the PowerSGD compression process, we transitioned from the Gram-Schmidt method to QR factorization. In Figure 4.1, we present a comparison of the timings for these two distinct methods, observing how they vary with changes in rank. QR factorization method demonstrates superior scalability with increasing rank, resulting in a training step that is four times faster when the rank is set to 16.

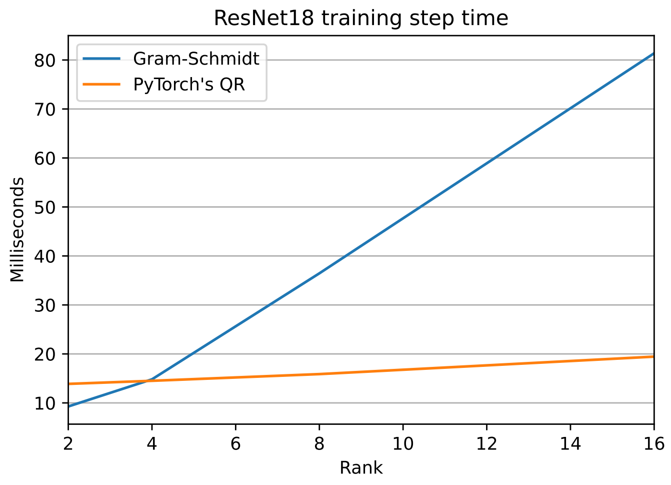

Figure 4.1. Training step duration in milliseconds for ResNet18 on CIFAR10 using PowerSGD with two orthogonalization techniques: Gram-Schmidt method and PyTorch's QR factorization (after v1.9). Training is conducted on a single, thus excluding communication overheads.

4.2.2 Timing training phases

We conducted timing analyses for various phases of training in a distributed environment. This involved altering both the batch size and the number of nodes used. For these tests, we specifically employed a VGG19 model and utilized the Google Cloud Platform, which was equipped with V100 GPUs.

We report in the following our findings.

Forward pass: The forward pass refers to the process of computing the output of the neural network from a given batch of inputs. In this phase, there is no need for inter-node communication, which means that the duration of the forward pass remains consistent regardless of the number of nodes involved. However, when implementing data parallelism, the input batch is typically divided among the available nodes. Consequently, as the number of nodes increases, the batch size allocated to each node decreases. We report timings for batch size of 128, 256, 512, and 1024 in figure 4.2, on the left. Evidently, the timing for the step function grows linearly with the batch size.

Backward pass: The backward pass is a phase in which backpropagation occurs, leading to the computation of local gradients. As for the forward pass, this stage does not involve communication between nodes, resulting in a time complexity that remains constant relative to the number of nodes. However, the duration varies linearly in relation to the batch size. In Figure 4.2, we present the outcomes of our experiments, which were conducted using the same batch sizes as employed in the forward pass timings.

Update phase: This phase involves updating the weights, a process typically in-

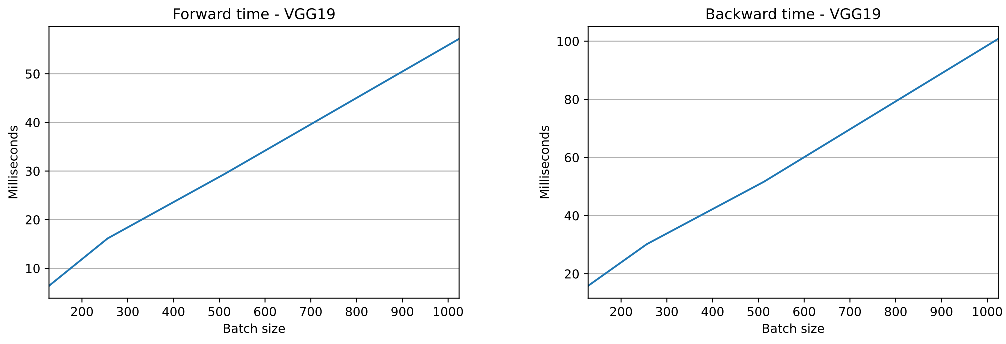

Figure 4.2. *Left:* Forward phase duration (in milliseconds) as a function of varying batch sizes (128, 256, 512, 1024). *Right:* Backward phase duration (in milliseconds) for the same range of batch sizes. In both scenarios, the timing exhibits a linear increase with respect to the batch size.

variant to the number of nodes and batch size. However, its time depends on the chosen type of optimizer.

Communication: The communication phase is the stage where gradients are transmitted across the network. In algorithms such as PowerSGD, this phase intertwines communication with compression. Consequently, both aspects – communication and compression – are considered when measuring the duration of this phase. The size of the gradients remains constant regardless of batch size variations; therefore, the time taken for this phase is primarily influenced by changes in the number of nodes involved. As in reduction operations the nodes aggregate data following a tree structure, we expect the the communication time to scale logarithmically with respect to the number of nodes. This is consistent with the results that we report in Figure 4.3.

Finally, in Figure 4.4, we present a comparative analysis of the time taken for a complete training loop by SGD and PowerSGD. This comparison is made by varying the number of nodes while maintaining a total batch size of 1024 on a VGG19 architecture.

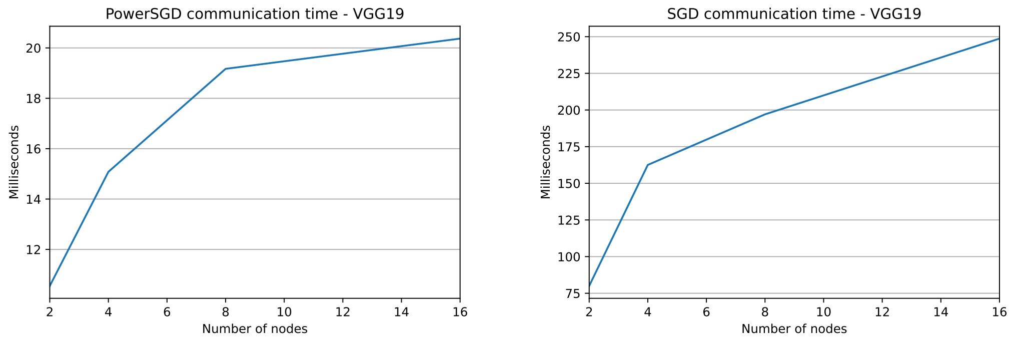

Figure 4.3. *Left:* Compression and communication time (in milliseconds) for PowerSGD while varying the number of nodes (2, 4, 8, 16). *Right:* Communication time (in milliseconds) for communicating the whole gradient without compression for the same range of nodes. In both scenarios, the timing exhibits a logarithmic increase with respect to the number of nodes. However, notice that PowerSGD is around 10x faster than standard SGD without compression.

Remarkably, PowerSGD demonstrates superior speed efficiency, outperforming even with as few as two nodes. Surprisingly, the use of a greater number of nodes with data parallelism in standard SGD results in a deceleration of the training process. This slowdown is attributed primarily to the time consumed in communication, despite the forward and backward passes benefit of the distributed workload.

4.2.3 Variants benchmark

We proceed now to study the PowerSGD variants that we introduced in Section 4.1.2. Their implementations along with the training scripts are available on GitHub[4].

Firstly, we investigate whether the error buffer accumulates over time and potentially becomes outdated, affecting the training process. For this scope, we observe the dynamics of the error buffer while training a ResNet18 model on the CIFAR10 dataset. For this analysis, we display the norm of the error buffer as it evolves throughout the training period. This norm is a critical measure, indicating the magnitude of the error buffer at various stages of training. The results are shown in Figure 4.5 along with a graph depicting the loss during the same training period. Interestingly, we observe a pattern where the norm of the error buffer initially increases as the training progresses. This increase continues until the model reaches a point of convergence and finally decreases after. This decrease is attributed to the fact that the gradient approaches zero as the model converges. This phenomenon raises important questions about the impact of the error buffer in neural network training.

[4]https://github.com/younik/async-optim

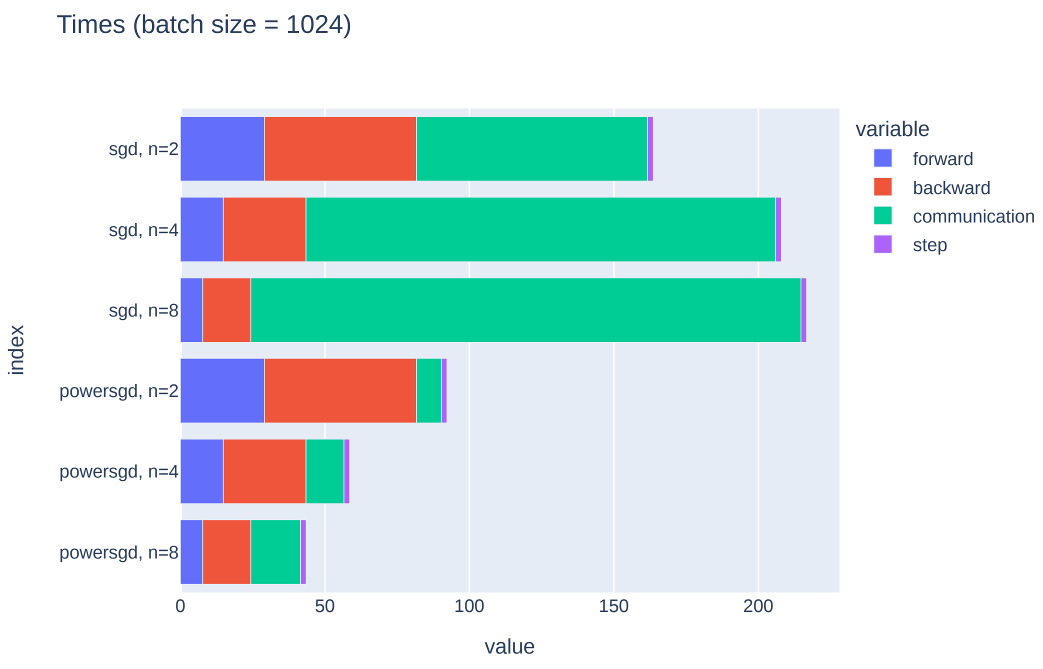

Figure 4.4. Comparison of the duration of various phases in the training loop between PowerSGD and SGD when applied to a VGG19 model with a batch size of 1024. We evaluate the performance across different numbers of nodes, specifically 2, 4, and 8. It is observed that, unlike PowerSGD, SGD does not gain any significant advantages from distributed data-parallel settings. Furthermore, PowerSGD consistently outperforms SGD by achieving faster training times in all three tested scenarios.

Decaying error feedback

Given that error feedback seems to accumulate during training, we test the variants were we decay it accordingly to λ, as explained in Section 4.1.2. As the method is invariant to the number of nodes, we test on a single node compressing the gradient as in the distributed setting. We focus on evaluating the performance of ResNet18 during its training phase on the CIFAR10 dataset, particularly examining the impact of varying the hyperparameter λ within the range of 0 to 1. For each distinct value of λ, we tune the learning rate to optimize the training process. We plot the results in Figure 4.6. Our findings reveal a gradual and consistent improvement in the model's performance as λ increases from 0 to 1. Notably, the model achieves its best results at $\lambda = 1$, which clearly indicates the positive influence of error feedback on the model's performance. This trend underscores the significance of error feedback in enhancing the learning efficiency.

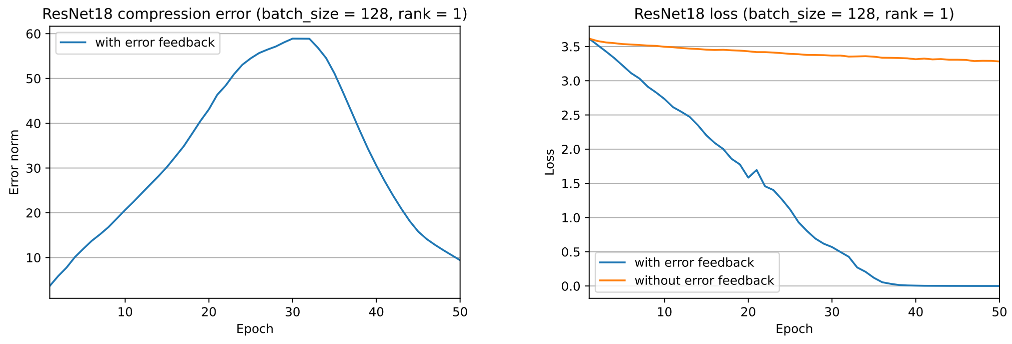

Figure 4.5. *Does error buffer become outdated?* We investigate if an error buffer accumulates and potentially becomes outdated training a ResNet18 on CIFAR10. On the left, we show the norm of the error buffer during training. On the right, is the loss during training. The norm of the error buffer increases during training until we converge, and then it finally starts decreasing since the gradient is almost zero.

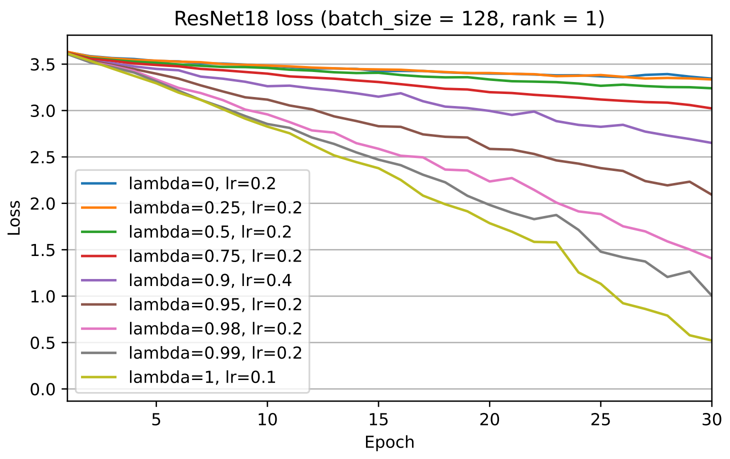

Figure 4.6. Loss of ResNet18 during training on CIFAR10. We test different values for λ, from 0 to 1 and we tune the learning rate for each of them. The results show a smooth transition from $\lambda = 0$ to $\lambda = 1$, where the latter achieves better results. This shows that error feedback improves performance.

Nesterov error feedback

We continue by analyzing the performances of the *Nesterov error feedback* variant introduced in Section 4.1.2. Our analysis focuses on the accuracy of a VGG19 network applied to the CIFAR10 dataset, utilizing a total batch size of 1024 across 4 nodes. As each node computes the gradient with different weight values, the method is sensible to the number of nodes. We vary the hyperparameters λ and tune the learning rate for each of them. We compare the method with plain *PowerSGD* and show the results in Figure 4.7. The *Nesterov error feedback* variant with $\lambda = 0.05$ outperforms

PowerSGD during training; however, they both converge around the same value of accuracy. Moreover, the *Nesterov error feedback* variant can be unstable with high values of λ.

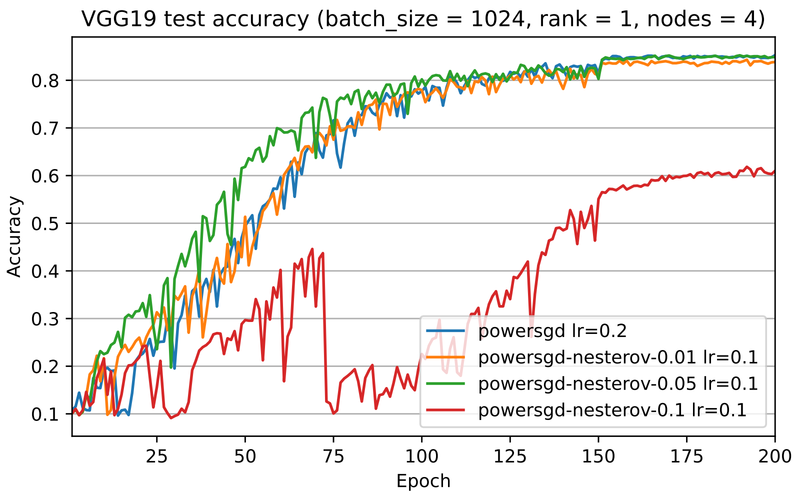

Figure 4.7. *Nesterov error feedback* variant analysis using a VGG19 on CIFAR10, using a total batch size of 1024 across 4 nodes. We plot the test accuracy over the training epochs. The *Nesterov error feedback* variant with $\lambda = 0.05$ outperforms PowerSGD; however, it can be unstable depending on λ.

Asynchronous communication

We now compare different algorithms: SGD, Asynchronous SGD, PowerSGD, and two unique asynchronous variants of PowerSGD. These variants are distinguished by their error transmission methods; the first variant transmits the entire error buffer, while the second sends only a fraction (one-quarter) of it. With Asynchronous SGD, we refer to a variant of SGD, where the algorithm starts the next forward and backward passes while the gradient is communicated over the network, resulting in a one-step delay of weight updates. To evaluate their performance, the test accuracy over time for the VGG19 neural network architecture was plotted in Figure 4.8 using the CIFAR10 dataset, with a total batch size of 1024 across 4 nodes.

The study reveals that PowerSGD demonstrated a significant edge over traditional SGD when the training time is taken into account. This shows that PowerSGD is more efficient in reaching higher accuracy in a shorter period. Among the PowerSGD variants, the adaptive asynchronous PowerSGD stood out, showing notable improvements over the standard PowerSGD model. This indicates that reducing the error is important to enhance learning efficiency. On the other hand, the non-adaptive variant of PowerSGD, which communicates the entire error buffer, shows performance on par with the standard SGD. This outcome could be attributed to the overhead caused by the transmission of the full error buffer.

These findings highlight the potential of asynchronous methods in improving neural network training, especially in distributed environments. They also underscore the importance of balancing communication overhead with computational efficiency to optimize the performance of training.

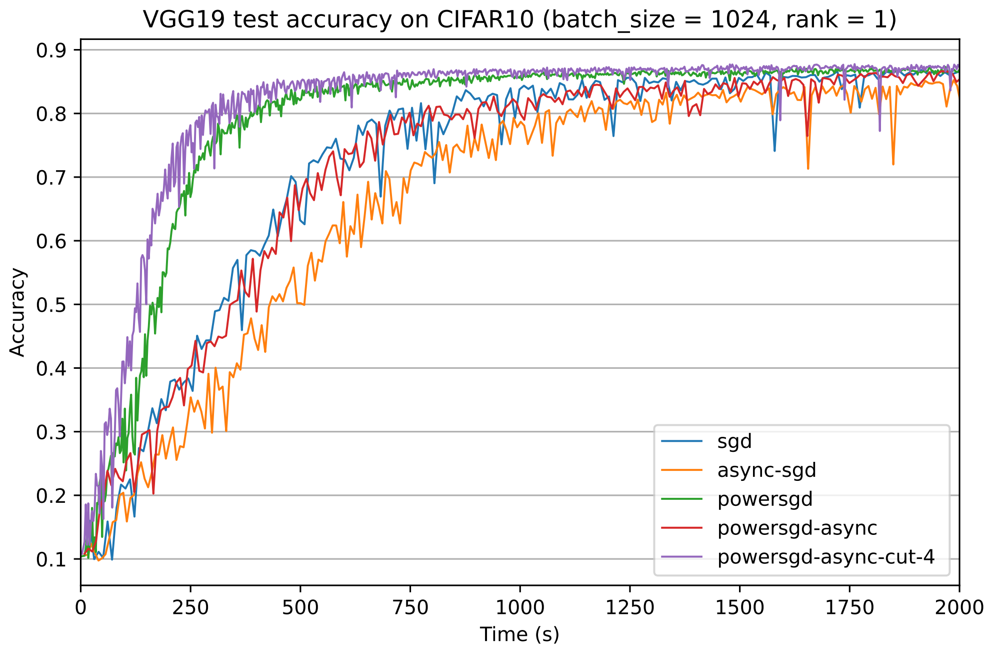

Figure 4.8. Comparison of SGD, Asynchronous SGD, PowerSGD, and two asynchronous variants of PowerSGD. The first variant transmits the full error, while the second sends only a quarter of it. We plotted the test accuracy over time for VGG19 on the CIFAR10 dataset, utilizing 4 nodes. The results showed that PowerSGD significantly outperforms SGD when training time is considered. Among the variants, the adaptive asynchronous PowerSGD showed further improvements over the standard PowerSGD. In contrast, the non-adaptive variant, which communicates the entire error buffer, displayed performance comparable to SGD, as its process is hampered by full error communication.

4.2.4 Discussion of results

We conducted a comprehensive analysis to determine how training phase durations scale within a typical setup, such as the Google Cloud Platform. This investigation enables the evaluation of the efficacy of gradient compression algorithms across various contexts and facilitates the identification of the break-even point for model scaling.

Our study uncovered that, contrary to expectations, the training speed of a VGG19 model does not improve significantly in a distributed setting. However, when employing PowerSGD for gradient compression, we observed notable benefits from scaling the model across multiple devices. Moreover, the efficiency of PowerSGD directly influences the break-even point, with faster compression leading to a higher break-even threshold.

Thus, we analyzed the efficiency of PowerSGD, identifying the orthogonalization process as a primary bottleneck. To address this, we conducted a comparative analysis between two orthogonalization techniques: the Gram-Schmidt algorithm and the QR factorization. Our experimental results unveiled a notable advantage of QR factorization over Gram-Schmidt orthogonalization when a SIMD hardware is available and the approximation rank is at least 4. With these findings, we modify the PyTorch codebase to use the faster orthogonalization technique depending on the setting, making the PowerSGD implementation around 20 times faster since version 1.11.

Furthermore, we've identified instances where error feedback accumulates, indicating potential for improving compression accuracy. This observation motivated the development and examination of novel variants of PowerSGD. Notably, the Nesterov error feedback variant has demonstrated superior performance in terms of test metrics during training. However, its sensitivity to the hyperparameter λ can result in training instability. Remarkably, our asynchronous variations, which involve parallel communication of the error buffer alongside other training phases, have proven effective in mitigating error buffer accumulation without sacrificing time efficiency. In our experiments, we observed that training with asynchronous communication converges about twice as fast as without it in comparison to standard PowerSGD.

shape	Gram-Schmidt	torch.qr	custom	GS/custom	torch/custom
(2, 256)	124	11087	36	3	308
(2, 512)	128	11176	39	3	287
(2, 1024)	120	10923	35	3	312
(2, 2048)	126	12328	52	2	237
(2, 4096)	132	11774	50	3	235
(4, 256)	302	11671	52	6	224
(4, 512)	301	10497	51	6	206
(4, 1024)	346	11793	54	6	218
(4, 2048)	466	12098	51	9	237
(4, 4096)	461	12448	51	9	244
(8, 256)	938	12351	51	18	242
(8, 512)	954	12177	56	17	217
(8, 1024)	669	11954	51	13	234
(8, 2048)	650	12204	62	10	197
(8, 4096)	807	12500	78	10	160
(16, 256)	1366	12214	67	20	182
(16, 512)	1321	11780	73	18	161
(16, 1024)	1320	12503	89	15	140
(16, 2048)	1332	12832	113	12	114
(16, 4096)	1337	13296	144	9	92
(32, 256)	3342	14109	123	27	115
(32, 512)	2892	11872	135	21	88
(32, 1024)	2703	15903	167	16	95
(32, 2048)	2727	14239	216	13	66
(32, 4096)	2986	18132	278	11	65
(64, 256)	5991	14093	234	26	60
(64, 512)	5438	11562	259	21	45
(64, 1024)	6224	13836	322	19	43
(64, 2048)	5440	14284	429	13	33
(64, 4096)	6446	12940	562	11	23
(128, 256)	11948	15023	460	26	33
(128, 512)	11719	16406	517	23	32
(128, 1024)	11729	18203	673	17	27
(128, 2048)	11657	17731	911	13	19
(128, 4096)	12248	27551	1262	10	22

Table 4.1. Orthogonalization times in microseconds for different shapes and algorithms. The Gram-Schimdt method is written in PyTorch and was the default choice for PowerSGD. torch.qr refers to the QR function in PyTorch before the 1.9 release and custom is our CUDA kernel. Our algorithm is up to 27 times faster than the Gram-Schmidt method and up to 300 times faster than PyTorch's QR implementation.

Chapter 5

Conclusions

We conclude the dissertation with a summary and possible extensions of our work.

5.1 Summary

In this dissertation, we have presented a comprehensive study of the PowerSGD compression algorithm, a critical component in the distributed training of deep learning models. Our work has not only contributed to the theoretical understanding of this compression algorithms in distributed settings but has also yielded practical improvements in training efficiency, as demonstrated through our experimental results.

After a brief introduction in Chapter 1, we present important concepts in machine learning in Chapter 2, focusing on distributed learning and the engineering challenges that come with it. In particular, we introduced different types of training parallelism, how to implement them in PyTorch, and the role of compression algorithms distributed training. In Chapter 3 we contextualize our work, introducing other compression algorithms and a practical work that uses PowerSGD[25] to scale training of big models.

Finally, in Chapter 4, we improve PowerSGD on several aspects. . After contextualizing our work in Chapter 3, we improve the popular PowerSGD algorithm, making it more efficient and accurate. Specifically, our contributions can be summarized as follows:

PyTorch 1.11 contribution: we successfully optimized PyTorch's PowerSGD implementation, transitioning from the Gram-Schmidt to QR factorization method. This change, coupled with the development of a custom CUDA kernel for orthogonalization, resulted in a remarkable increase in compression speed, making PowerSGD around 20 times faster.

Enhancement of PowerSGD accuracy: we conceptualized and implemented several innovative variants of PowerSGD that try to deal with the accumulation of the error buffer. These include decaying error feedback, Nesterov error feedback, and asynchronous communication methods, each addressing specific challenges in the training process. Notably, the latter two methods demonstrated superior performance, making training convergence twice as fast compared to the standard PowerSGD.

Distributed setting analysis: We conducted a study where we examined different distributed settings by simulating various scenarios with different numbers of nodes, batch sizes, and network latencies. This comprehensive approach helped us gain insights into the ideal conditions for scaling up the number of nodes while also identifying situations where scaling may not be advantageous. Notably, training a network like VGG19 does not benefit from distributed settings and becomes slower when increasing the number of nodes on the Google Cloud Platform. However, by using the optimized version of PowerSGD, we can scale the model and take advantage of utilizing multiple devices.

In conclusion, our contributions to optimizing the PowerSGD algorithm represent a meaningful step forward in the pursuit of more efficient and scalable deep learning training methodologies. As the demand for larger and more complex models continues to grow, the importance of such advancements becomes more evident. The integration of our optimizations into the PyTorch codebase aims to have a direct impact on the broader deep learning community. We hope that our work will not only benefit current practitioners in the field but also inspire further research in this vital area of deep learning.

5.2 Future work

We highlight here few directions of improvements over our work.

Save error in gradient memory: To enhance the memory efficiency of PowerSGD, a possible strategy involves eliminating the error buffer and storing the error directly in the gradient memory. Implementing this approach can be done with custom coding. However, integrating it into PyTorch demands careful API design because the standard practice in PyTorch training loops involves calling the optimizer's zero_grad, which resets the gradient memory.

Parallelize communication: In the feedforward and backpropagation phases, the communication channel remains idle, which inspired the asynchronous methods detailed in Section 4.1.2. Conducting an exhaustive analysis of the advantages and limitations of these methods could inform potential modifications to PyTorch. However, overlapping different phases in PyTorch presents challenges. For instance, during backpropagation, it is feasible to start transmitting the gradients of each layer as soon as they are computed. Yet, in practice, the communication process commences only after the completion of backpropagation. This suggests an area for improvement in PyTorch efficiency.

Study PowerSGD effect: PowerSGD modifies the learning dynamics by projecting the gradient into a lower-dimensional space. This alteration raises several research questions. For instance, it prompts an investigation into the specific impacts on the learning dynamics. Additionally, there is a need to explore whether this dimensional reduction can function as a regularizer, potentially enhancing the model's generalization capabilities.

Bibliography

[1] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems* (H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, eds.), vol. 33, pp. 1877–1901, Curran Associates, Inc., 2020.

[2] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," *arXiv preprint arXiv:2001.08361*, 2020.

[3] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. a. Ranzato, A. Senior, P. Tucker, K. Yang, Q. Le, and A. Ng, "Large scale distributed deep networks," in *Advances in Neural Information Processing Systems* (F. Pereira, C. Burges, L. Bottou, and K. Weinberger, eds.), vol. 25, Curran Associates, Inc., 2012.

[4] F. Iandola, M. Moskewicz, K. Ashraf, and K. Keutzer, "Firecaffe: Near-linear acceleration of deep neural network training on compute clusters," pp. 2592–2600, 06 2016.

[5] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu, "1-bit stochastic gradient descent and application to data-parallel distributed training of speech dnns," in *Interspeech 2014*, September 2014.

[6] J. Bernstein, Y.-X. Wang, K. Azizzadenesheli, and A. Anandkumar, "signsgd: compressed optimisation for non-convex problems," in *International Conference on Machine Learning*, 2018.

[7] N. Strom, "Scalable distributed dnn training using commodity gpu cloud computing," in *Interspeech*, 2015.

[8] T. Vogels, S. P. Karimireddy, and M. Jaggi, "PowerSGD: Practical Low-Rank Gradient Compression for Distributed Optimization," in *NeurIPS 2019 - Advances in Neural Information Processing Systems*, 2019.

[9] H. Askr, E. Elgeldawi, H. Aboul Ella, Y. A. M. M. Elshaier, M. M. Gomaa, and A. E. Hassanien, "Deep learning in drug discovery: an integrative review and future challenges.," *Artif Intell Rev*, vol. 56, no. 7, pp. 5975–6037, 2023.

[10] Q. Rao and J. Frtunikj, "Deep learning for self-driving cars: Chances and challenges," in *2018 IEEE/ACM 1st International Workshop on Software Engineering for AI in Autonomous Systems (SEFAIAS)*, pp. 35–38, 2018.

[11] R. Lam, A. Sanchez-Gonzalez, M. Willson, P. Wirnsberger, M. Fortunato, F. Alet, S. Ravuri, T. Ewalds, Z. Eaton-Rosen, W. Hu, A. Merose, S. Hoyer, G. Holland, O. Vinyals, J. Stott, A. Pritzel, S. Mohamed, and P. Battaglia, "Learning skillful medium-range global weather forecasting," *Science*, vol. 382, no. 6677, pp. 1416–1421, 2023.

[12] S. Akhtar, M. Adeel, M. Iqbal, A. Namoun, A. Tufail, and K.-H. Kim, "Deep learning methods utilization in electric power systems," *Energy Reports*, vol. 10, pp. 2138–2151, 2023.

[13] N. Banús, I. Boada, P. Xiberta, P. Toldrà, and N. Bustins, "Deep learning for the quality control of thermoforming food packages," *Scientific Reports*, vol. 11, no. 1, p. 21887, 2021.

[14] Z. Kang, C. Catal, and B. Tekinerdogan, "Machine learning applications in production lines: A systematic literature review," *Computers & Industrial Engineering*, vol. 149, p. 106773, 2020.

[15] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[16] I. J. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016. `http://www.deeplearningbook.org`.

[17] Hecht-Nielsen, "Theory of the backpropagation neural network," in *International 1989 Joint Conference on Neural Networks*, pp. 593–605 vol.1, 1989.

[18] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: a survey," *Journal of Machine Learning Research*, vol. 18, no. 153, pp. 1–43, 2018.

[19] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *Proceedings of The 33rd International Conference on Machine Learning* (M. F. Balcan and K. Q. Weinberger, eds.), vol. 48 of *Proceedings of Machine Learning Research*, (New York, New York, USA), pp. 1928–1937, PMLR, 20–22 Jun 2016.

[20] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning, *et al.*, "Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures," in *International conference on machine learning*, pp. 1407–1416, PMLR, 2018.

[21] H. B. McMahan, E. Moore, D. Ramage, and B. A. y Arcas, "Federated learning of deep networks using model averaging," *CoRR*, vol. abs/1602.05629, 2016.

[22] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019.

[23] G. Gur-Ari, D. A. Roberts, and E. Dyer, "Gradient descent happens in a tiny subspace," *CoRR*, vol. abs/1812.04754, 2018.

[24] S. P. Karimireddy, Q. Rebjock, S. U. Stich, and M. Jaggi, "Error feedback fixes SignSGD and other gradient compression schemes," in *ICML - Proceedings of the 36th International Conference on Machine Learning*, pp. 3252–3261, 2019.

[25] A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever, "Zero-shot text-to-image generation," in *Proceedings of the 38th International Conference on Machine Learning* (M. Meila and T. Zhang, eds.), vol. 139 of *Proceedings of Machine Learning Research*, pp. 8821–8831, PMLR, 18–24 Jul 2021.

[26] S. Hochreiter and J. Schmidhuber, "Flat minima," *Neural computation*, vol. 9, no. 1, pp. 1–42, 1997.

Appendix

Appendix A

CUDA basics

A.1 Introduction

CUDA (Compute Unified Device Architecture) is a parallel computing platform and application programming interface model created by NVIDIA. It allows software developers to use NVIDIA GPUs for general-purpose processing. The CUDA platform is designed to work with programming languages such as C++. This enables developers to accelerate compute-intensive applications by using the power of GPUs for parallelizable tasks. CUDA provides a range of tools and libraries that simplify developing software that runs on NVIDIA GPUs.

CUDA code is typically written in `.cu` files, which are a mix of C++ code and CUDA-specific extensions. The CUDA kernels, i.e. the functions that run on the GPU, are denoted by the `__global__` keyword. CUDA kernels are launched from the host code with a specific execution configuration that defines the grid and block dimensions. So, let's define these important concepts:

Grid: A grid is the collection of blocks that execute a kernel. It's an abstraction to organize CUDA threads in the GPU. Each block within the grid can be executed independently and can contain a different number of threads.

Block: A block is a group of threads that execute the same kernel and share the same memory space. Threads within a block can synchronize their execution and share data using shared memory. The number of threads per block is a key performance-tuning parameter in CUDA programming.

Warps: A warp is the basic unit of execution in an NVIDIA GPU. It consists of a group of 32 threads that are executed in lockstep, i.e. at the same clock time. When a CUDA program is run, the blocks are divided into warps, which are scheduled and executed independently. The warp execution model exploits thread-level parallelism and is fundamental to achieving high performance in CUDA applications.

The execution model of CUDA is designed to achieve massive parallelism, as thousands of threads can run in parallel. NVIDIA GPUs are built around a scalable array of multithreaded Streaming Multiprocessors (SMs). When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to SMs with available execution capacity. The threads of a block execute concurrently on one SM, and multiple blocks can execute concurrently on one SM. As blocks terminate, new blocks are launched on the vacated SMs.

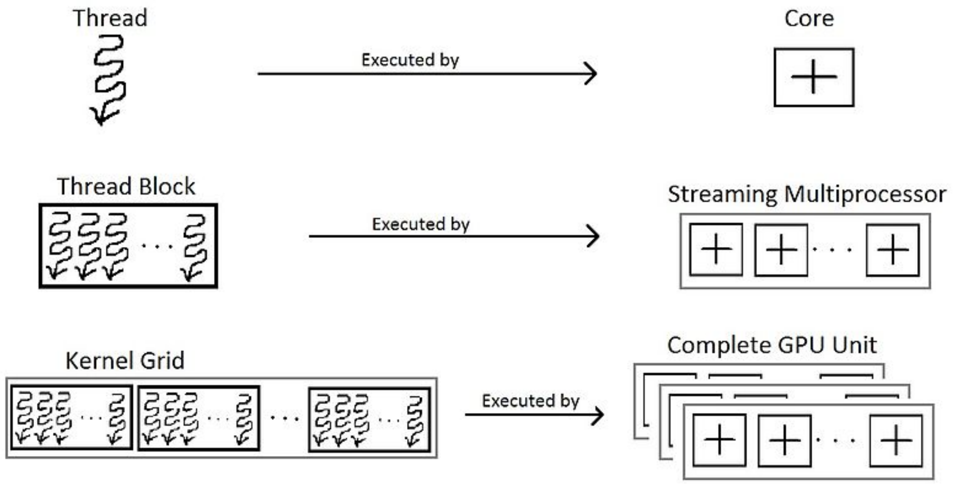

Figure A.1. Diagram illustrating the hierarchy of GPU execution models: individual threads executed by cores, thread blocks by a streaming multiprocessor, and kernel grids by the complete GPU unit. *Image from WikiMedia.*

To illustrate the CUDA syntax, let's examine the following example:

```
#include <cuda_runtime.h>
#include <stdio.h>

// CUDA kernel
__global__ void setToZeroKernel(int *array) {
    // each thread has its own value for blockDim, blockIdx,
    // and threadIdx, each possibly 3-dimensional with x, y, z
    // In this case blockDim.x == 0, blockIdx.x == 0,
    // threadIdx.x spans from 0 to 255
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    array[i] = 0;
}

int main() {
    const int numElements = 256;

    // Allocate device memory
    int *d_array = NULL;
    size_t size = numElements * sizeof(int);
    cudaMalloc((void **)&d_array, size);

    // Launch the kernel
    setToZeroKernel<<<1, numElements>>>(d_array);

    // Free device memory
    cudaFree(d_array);

    return 0;
}
```

A.2 Performance considerations

When developing applications using CUDA, it's crucial to consider various factors that can significantly impact the performance of your program. We list here some key performance considerations:

Memory hierarchy: Understand the different memory types and their access speeds:

- Global Memory: Accessible by all threads, with relatively high access latency.
- Shared Memory: On-chip and much faster than global memory, accessible by all threads within the same block.
- Registers: Fastest memory, private to each thread.

Thus, optimize memory usage by utilizing faster memory wherever possible.

Memory coalescing: Ensure that memory accesses are coalesced. This means that consecutive threads should access consecutive memory addresses to maximize throughput.

Reduce memory transfer overheads: Minimize data transfers between the host and the device, as these can be costly.

Maximize occupancy: Aim for high occupancy to ensure a large number of warps are active. This helps in hiding latency and improves utilization of the GPU's resources.

Minimize memory usage: Be aware of the limitations in the number of registers and shared memory size. The number of threads that can run simultaneously is limited by the amount of registers in the GPU. Excessive usage of these resources can reduce occupancy.

Optimize block size: Choose an appropriate block size for kernel launches. A balance must be struck between having enough threads to achieve full occupancy and not having so many that they contend for resources.

Minimize warp divergence: Ensure that threads within a warp do not diverge significantly, for example taking two different branches of an `if-else` statement. This will make some warp idling while others are executing different instructions.

Balance workload: Distribute the workload evenly across threads.

A.3 Reduction pattern

To provide an educational illustration, we will develop a reduction kernel. This type of function is common when we need to aggregate data across multiple threads. A classic use case for this is calculating the dot product.

Let's start by defining the kernel structure and initializing the shared memory used for reduction:

```
__global__ void dotProductKernel(float *a, float *b, float *c, int N) {
    __shared__ float cache[threadIdx.x];
}
```

Then, let's incorporate the code that calculates the product of corresponding elements on a per-thread basis:

```
int tid = threadIdx.x + blockIdx.x * blockDim.x;

float temp = 0;
while (tid < N) {
    temp += a[tid] * b[tid];
    tid += blockDim.x * gridDim.x;
}
```

This allows each thread to operate on multiple elements if the vector is larger than the total number of threads. Moreover, it allows to launch of the kernel with multiple blocks.

Now, let's store intermediate results in the shared memory and synchronize threads to ensure all computations are done before reduction.

```
cache[cacheIndex] = temp;
__syncthreads();
```

Now, we can perform the block-wise reduction:

```
int i = blockDim.x / 2;
while (i != 0) {
    if (threadIdx.x < i)
        cache[threadIdx.x] += cache[threadIdx.x + i];
    __syncthreads();
    i /= 2;
}
```

And, finally, the block-wise reduction using atomic operation:

```
if (cacheIndex == 0)
    atomicAdd(c, cache[0]);
```

While the code can be optimized further, for example using warp-level primitives, this code provides a starting point for understanding the reduction pattern. To enhance computational efficiency, we can tailor the number of threads per block and the corresponding number of blocks to the specific hardware and vector dimensions in use. This optimization ensures the most effective allocation of computational resources, leading to improved performance in processing tasks.

A.4 QR factorization implementation

We comment now on optimizations on our QR factorization code.

First, to write a custom dot product function relying on `cub` library for reduction:

```
#include <cub/cub.cuh>

template <int BLOCK_THREADS, typename scalar_t>
__device__  __forceinline__
scalar_t dot(scalar_t *a, scalar_t *b, uint length){
    typedef cub::BlockReduce<scalar_t, BLOCK_THREADS,
    cub::BLOCK_REDUCE_RAKING_COMMUTATIVE_ONLY> BlockReduce;
    __shared__ typename BlockReduce::TempStorage tmp_storage;

    int tx = threadIdx.x;
    uint unroll = ceil( (float)length / (float)BLOCK_THREADS );
    uint idx = (tx & -32u)*unroll + (tx & 31);

    scalar_t local_prod = 0;
    for (uint i = 0; i < unroll; ++i){
        local_prod+= (idx<length)? a[idx]*b[idx] : (scalar_t)0;
        idx += 32;
    }

    scalar_t reduce = BlockReduce(tmp_storage).Sum(local_prod);

    __shared__ scalar_t dot;
    if (tx == 0)
        dot = reduce;
    __syncthreads();

    return dot;
}
```

Notice how we compute the local product of each thread: in order to optimize memory access, we make sure warp access to a contiguous memory, as shown in Figure A.2.

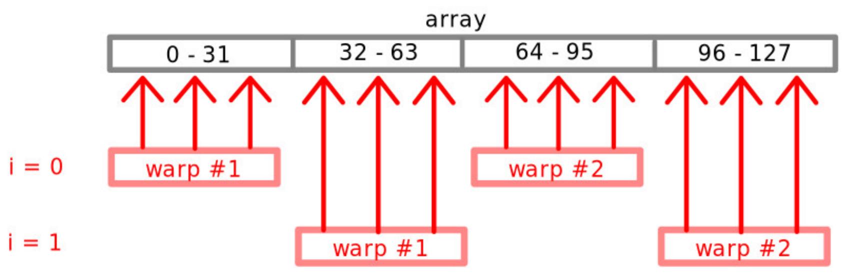

Figure A.2. Illustration of the *warp-unrolling* technique to optimize memory access.

Moreover, as we needed to synchronize different blocks while constructing the R matrix, we developed a custom barrier mechanism:

```
    __device__ __forceinline__
    void wait_barrier(int* barrier, int target){
        if (threadIdx.x == 0){
            int counter;
            do {
                asm volatile (
                "ld.relaxed.gpu.global.s32 %0, [%1];" :
                "=r"(counter): "l"(barrier) );
            }
            while (counter < target);
        }
        __syncthreads();
    }
```

```
__device__ __forceinline__
void set_barrier(int* barrier, int value){
    if(threadIdx.x == 0)
        asm volatile ("st.global.cg.s32 [%0], %1;" ::
        "l"(barrier), "r"(value));
}
```

The `wait_barrier` function waits until the value of the barrier is greater or equal to the target. The synchronization logic is executed only by the first thread in the block, which continuously reads the value of the barrier variable using the assembly instruction `ld.relaxed.gpu.global.s32`.

The `set_barrier` is used to set the value of the barrier variable. Again, only the first thread in the block performs the action, which is writing the given value into the barrier variable using the assembly instruction `st.global.cg.s32`.

Appendix B

PyTorch internals

B.1 Repository structure

This chapter delves into the structure of PyTorch's repository, focusing on the `aten`, `c10`, and `torch` directories, each of which plays a crucial role in the library's functionality.

B.1.1 torch

The `torch` directory is the face of PyTorch that Python users interact with. It provides higher-level functions, neural network layers, and optimization algorithms. Key subdirectories include:

nn: This subdirectory is crucial for building neural network models in PyTorch. It contains predefined layers like convolutional, recurrent, linear, and dropout layers, which are the building blocks of most neural network architectures. Beyond layers, `nn` also offers a collection of activation functions (e.g., ReLU, Sigmoid), loss functions (e.g., CrossEntropyLoss, MSELoss), and other utilities necessary for constructing and managing neural networks.

optim: This directory contains various optimization algorithms like SGD, Adam, RMSprop, which are used to update the weights of the network during training. The common interface allows users to easily switch between different optimization strategies.

autograd: Autograd is a core feature of PyTorch that provides automatic differentiation for all operations on tensors. This is essential for implementing the backpropagation algorithm. It keeps track of the computation graph, and when a backward pass is triggered, it automatically computes the gradients for each tensor involved in the computation, facilitating the optimization process. This feature enables dynamic computation graphs, meaning the graph can change from iteration to iteration, offering flexibility in model design.

utils: This subdirectory offers a variety of utility functions that support different aspects of using PyTorch. It includes data loading and preprocessing utilities, which are essential for feeding data into neural networks. It also provides functions for model serialization and deserialization, which are useful for saving and loading models. Other utilities include helper functions for tensor operations, model visualization, and more, enhancing the ease of use of PyTorch.

jit: Just In Time (JIT) compilation in PyTorch refers to a dynamic runtime process that significantly optimizes the execution of PyTorch models. The traditional execution of Python code involves interpreting one instruction at a time, which can be inefficient. The JIT compiler in PyTorch tackles this by translating portions of the model into a more efficient, low-level machine code before execution. This process involves two primary components: *tracing* and *scripting*. Tracing is used on models where the control flow (like loops and conditionals) depends on the input data, and it records the operations performed when the model is run with sample data. Scripting, on the other hand, converts Python code, including control flow, into TorchScript, a static graph representation that provides performance improvements through optimized execution. This JIT compilation process is crucial for real-time applications and high-performance computing tasks where speed and efficiency are paramount.

distributed: The distributed module provides functionalities for distributed training. It includes different backends for communications, like NCCL and MPI, and offers tools for both data parallelism and model parallelism. The core implementations of these functionalities are primarily written in C++ and are located in the `torch/csrc/distributed/c10d` directory. Within this directory, the C++ codebases are seamlessly integrated with Python, allowing for a user-friendly interface. The subdirectory `algorithms` provides implementations of compression algorithms, including PowerSGD.

B.1.2 ATen

ATen is the core tensor library in PyTorch. It stands for "A Tensor Library" and provides the foundational data structures and operations for tensors. ATen provides an abstraction layer for the underlying computational device, whether it's a CPU or GPU, facilitating backend-specific implementations of tensor operations. This approach optimizes performance across a range of hardware platforms.

The Tensor class is the most fundamental in ATen, representing a multi-dimensional array. It's the core data structure used for all mathematical operations. This class allows for creating tensors, manipulating their shapes, and performing many operations. PyTorch employs reference counting as a sophisticated mechanism to efficiently handle and manipulate tensors. When operations like `view()` are invoked on a tensor, PyTorch doesn't immediately allocate new memory for the resultant tensor. Instead, it creates a new tensor object that shares the same underlying data storage as the original tensor, thereby increasing its reference count. When a Tensor is no longer needed and is garbage collected, the reference count of its storage decreases. The underlying storage is freed only when the reference count drops to zero, indicating no more tensors are referencing that particular block of memory. This shared storage mechanism is efficient, as it avoids unnecessary data duplication, reducing memory usage and enhancing performance. However, since multiple tensors can share the same data, modifications to the data through one tensor view are reflected in all other views, requiring careful consideration while manipulating tensors to avoid unintended side effects.

B.2 Operation bindings

In PyTorch, operation bindings refer to the process of linking high-level Python code to low-level C++ implementations. This binding ensures efficient execution of tensor operations. This is handled by the Dispatcher which acts as an abstraction layer, allowing users to write code without worrying about the underlying hardware. Depending on the device type (CPU, GPU) and data type (float, int), it dynamically binds Python functions to their C++ counterparts.

Functions are registered with the Dispatcher, along with their metadata (such as the device and data type they support). When a PyTorch function is called in Python, the call is directed to the Dispatcher. The Dispatcher looks up the registered functions and selects the appropriate implementation based on the current context.

B.2.1 Adding operations to PyTorch

Let's suppose you wrote a custom tensor operation. How to add the operation to PyTorch, to be usable in Python?

Method 1: Runtime extension loading

A simple way to use a custom C++ operation in PyTorch is to load the module using `torch.utils.cpp_extension.load`. After writing the custom C++ code, we can load it with the following code:

```
from torch.utils.cpp_extension import load

my_extension = load(
    name='example_extension',
    sources=['path/to/my_extension.cpp'],
    extra_cflags=['-O2'],
)
```

where `extra_cflags` gives the additional flags to pass to the C++ compiler and `sources` the path to our custom C++ code. After running the code above, the function defined in our C++ module is now usable:

```
my_extension.function_name(...)
```

where `function_name` is the name of the function defined in `my_extension.cpp`.

Method 2: Adding to native functions Integrating an operation as a native function within PyTorch makes it accessible directly under the torch namespace. Additionally, this operation is compiled as part of the PyTorch installation process. However, this necessitates modifications to the PyTorch codebase and requires compiling the library from the source. For those looking to contribute new tensor operations to PyTorch, this is the standard procedure to follow.

The first thing to do is declare your function in `native_functions.yaml`. This file serves as the central registry for native functions in ATen. The entry should follow this format:

```
- func: func_name(ArgType arg0[=default], ...) -> Return
  python_module: linalg
  variants: function, method
  dispatch:
     CPU: func_cpu
     CUDA: func_cuda
```

where:
- `func` is the function signature with name, arguments, and return type. Common types include ‘Tensor‘, ‘int[]‘, ‘float‘, ‘bool‘, and ‘str‘. Optional arguments and default values are also supported. The signature must match the signature declared in the C++ file.
- `python_module` defines in which Python module the function will be available. If not specified, the function will be accessible from torch. - `variants` specifies if the function is a Tensor method, function, or both.
- `dispatch` maps backends (e.g., CPU, CUDA) to specific implementation functions.

If the function is not automatically differentiable, i.e. it doesn’t rely only on other differentiable torch functions, you need to write a corresponding backward function and add an entry in `tools/autograd/derivatives.yaml`.

After doing these modifications and placing our code in the appropriate folder, we can build PyTorch from the source; Python bindings are automatically generated.