Toward Robust and Communication Efficient Distributed Machine Learning

by

Hongyi Wang

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2021

Date of final oral examination: 06/17/2021

The dissertation is approved by the following members of the Final Oral Committee:

Dimitris Papailiopoulos, Assistant Professor, Electrical and Computer Engineering
Shivaram Venkataraman, Assistant Professor, Computer Science
Kangwook Lee, Assistant Professor, Electrical and Computer Engineering
Stephen Wright, Professor, Computer Science
Theodoros Rekatsinas, Assistant Professor, Computer Science

To my family and friends.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Dimitris Papailiopoulos. He has been an exemplar of a determined researcher and encouraging mentor. I deeply appreciate the opportunities he created, the enjoyable research environment he cultivated, and the guidance he kindly offered during my Ph.D. journey. I can still remember Dimitris walked me through the derivation of SGD step by step during our very first meeting. During the first two years of my Ph.D., Dimitris did an amazing job guiding me to delve into the machine learning and systems (MLSys) research area. Later on, Dimitris encourages me a lot in generating my own research agenda and becoming an independent researcher. In my humble opinion, Dimitris is a great advisor who really cares about the students' interests and encourages them to delve deep.

I am also grateful to Professor Shivaram Venkataraman, Professor Stephen Wright, Professor Kangwook Lee, and Professor Theodoros Rekatsinas for helping with completing this dissertation and for serving on my committee. Their doors are always open for me. I would like to especially thank Dimitris, Shivaram, and Kangwook for being the readers of this dissertation.

It has been a pleasure to be part of several labs and institutions at the University of Wisconsin - Madison, including MADLab, IFDS, the MLOPT Research Group, and the Visual Computing Lab. Within these groups and institutions, I have had the fortune of working with many outstanding researchers, including but not limited to: Saurabh Agarwal, Jordan Black, Lingjiao Chen, Zachary Charles, Michael Gleicher, Paraschos Koutris, Shengchao Liu, Alliot Nagle, Shashank Rajput, Daniel Rakita, Harrison Rosenberg, Kartik Sreenivasan, Guru Subramani, Scott Sievert, Daewon Seo, Jy-yong Sohn, Harit Vishwakarma, Yuzhe Xiao, Michael Zinn, and Jinman Zhao. I would like to especially thank Zachary Charles and Lingjiao Chen for their friendship and indispensable help, which mean a lot to me.

I have also had the opportunity to work with several excellent collaborators beyond Madison, including Leshang Chen, Susan Davidson, Edgar Dobriban, Kannan Ramchandran, Yuekai Sun, Jianyu Wang, Dong Yin. I want to also thank

all my collaborators of the FedML project, especially Chaoyang He and Songze Li.

Apart from research in academia, I have also spent time in the industry. During the summer of 2019, I interned at IBM Research at Cambridge, Massachusetts hosted by Mikhail Yurochkin and Yasaman Khazaeni. Misha and Yasaman took good care of me and inspired my research a lot. I had three wonderful and productive months and delved really deep into federated learning at IBM Research. More importantly, I made a few friends at Cambridge. I would like to especially thank Zhongshen Zeng and Ren Wang with whom I had a lot of fun time exploring many great places and restaurants in Boston and Cambridge. I also enjoyed hanging out a lot in Boston with my friends Xin Luo and Xikun Sun.

During the summer of 2020, I interned at the DeepSpeed team of Microsoft. I was very honored to be hosted by Minjia Zhang and Yuxiong He. Although the internship was completely virtual, I have learned a lot in MLSys from Minjia and Yuxiong. I would like to especially thank Minjia for being an inspiring and encouraging mentor. I enjoyed all the one-on-one meetings with Minjia.

I would like to also thank SONY for supporting the Pufferfish project. I want to especially thank Yoshiki Tanaka, Hisahiro Suganuma, Pongsakorn U-chupala, Yuji Nishimaki, and Tomoki Sato from the SONY team for invaluable discussions and feedback.

Apart from research and studies, I had a lot of fun with friends at Madison during the past five years. I want to thank Jiefeng Chen, Qixiu Cao, Tuan Dinh, Hao Fu, Shuo Han, Oliver Liu, Vishnu Lokhande, Yuzhe Ma, Zhiwei Fan, Xu Ji, Xin Jin, Yutian Tao, Xiaomin Zhang, Ankit Pensia, Muni Sreenivas Pydi, Rong Pan, Zeping Ren, Huawei Wang, Yijing Zeng, Jinnian Zhang, and Xuezhou Zhang for their friendship.

Finally, I would like to thank my parents Jiansong Wang and Wenyan Cai who are constant sources of support and inspiration. I want also to thank my lovely girlfriend Xinle Jia. This dissertation is possible only as a result of their love, support, and encouragement.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Distributed machine learning (ML) is emerging as its own field at the heart of MLSys due to the exploding scale of modern deep learning models and the enormous amounts of data. However, distributed ML suffers from low communication efficiency and is vulnerable to adversarial attacks. This dissertation focuses on improving the communication efficiency and robustness of distributed ML for two popular use cases, *i.e.*, centralized distributed ML and federated learning (FL).

The first part of this dissertation focuses on communication efficiency. For centralized distributed ML, we start by presenting Atomo, a general framework to compress the gradients via atomic sparsification. Improving Atomo, we present Pufferfish, which bypasses the need for gradient compression via integrating it into model training. Pufferfish trains the factorized low-rank model starting from its full-rank counterpart, which achieves both high communication and computation efficiency without the need of using any gradient compression. For FL, we propose FedMA, which uses matched averaging in a layer-wise manner instead of one-shot coordinate-wise averaging for model aggregation. FedMA effectively reduces the number of FL rounds needed for the global model to converge.

The second part of this dissertation focuses on robustness. In the centralized setting, we present Draco, which leverages algorithmic redundancy to achieve Byzantine resilience with black-box convergence guarantees. To improve Draco, we present Detox, which combines robust aggregation with algorithmic redundancy. Detox can be used in tandem with any robust aggregation methods and enhances their Byzantine resilience and scalability. For FL, we demonstrate its vulnerability to training-time backdoors. We establish that robustness to backdoors implies model robustness to adversarial examples, a major open problem in itself. Furthermore, detecting the presence of a backdoor in an FL model is unlikely. We couple our results with *edge-case backdoors*, which forces a model to misclassify on seemingly easy inputs that are however unlikely to be part of the training or test data. We demonstrate that edge-case backdoors can lead to unsavory failures and may have serious repercussions on fairness and bypass all existing defense mechanisms.

Part I

Preliminaries

1 INTRODUCTION

Modern ML has recently ushered a plethora of advances in data science and engineering. Many of these advances are made possible through *de facto* deep neural networks that have hundreds of billions of trainable parameters (Devlin et al., 2019; Brown et al., 2020; Jumper et al., 2021). Training these massive-scale ML models to SotA accuracy is a computationally formidable task. For instance, the initial training stage of AlphaFold2 takes approximately one week, and the fine-tuning stage takes approximately 4 more days on 128 TPUv3 cores (Jumper et al., 2021).

To speed up training, centralized distributed training has become the *de facto* architectural choice for deploying learning algorithms [1] (Dean et al., 2012; Li et al., 2014; Abadi et al., 2016; Sergeev and Del Balso, 2018). Several ML frameworks such as MXNet, TensorFlow, PyTorch, Horovod, and DeepSpeed (Chen et al., 2015; Abadi et al., 2016; Paszke et al., 2017; Sergeev and Del Balso, 2018; Rasley et al., 2020) come with distributed implementations for popular ML tasks, *e.g.,* computer vision and natural language processing.

FL, as an alternative distributed ML scheme to centralized distributed ML, has emerged as a popular paradigm to collaboratively train a shared model while keeping all the training data on decentralized mobile devices/organizations (*e.g.,* medical or financial) for protecting data privacy better as clients' data is usually private, *e.g.,* medical patient data and student educational data (McMahan et al., 2017a; Kairouz et al., 2019; Wang et al., 2021b). Several current applications of FL include text prediction in mobile device messaging (Yang et al., 2018; Hard et al., 2018; Ramaswamy et al., 2019; Chen et al., 2019; Yuan et al., 2020), speech recognition (Sim et al., 2019), face recognition for device access ((Apple); Vincent), and maintaining decentralized predictive models across health organizations (Brisimi et al., 2018; Xu and Wang, 2019; Rieke et al., 2020). Compared to centralized distributed ML, the participating clients' data in FL are usually unbalanced and non-IID. The typical FL paradigm involves two stages: (i) clients train models with their local datasets

[1] This dissertation focuses on centralized data-parallel distributed training.

independently, and (ii) the data center gathers the locally trained models and aggregates them to obtain a shared global model. One of the standard aggregation methods is FedAvg where parameters of local models are averaged coordinate-wise with weights proportional to sizes of the client datasets (McMahan et al., 2017a).

Unfortunately, both centralized distributed ML and FL algorithms suffer from low communication efficiency and robustness issues, which we discuss in detail next.

1.1 The communication efficiency and robustness challenges for distributed ML

Communication efficiency challenges. The distributed speedup gains promised by centralized distributed training vanish when scaling out to beyond a few tens of compute nodes. Several studies on distributed ML have consistently observed that there is a tremendous gap between ideal and realizable distributed speedup gains when using more than tens of compute nodes to train a deep representation (Dean et al., 2012; Seide et al., 2014; Qi et al., 2016; Grubic et al., 2018; Narayanan et al., 2019). This in turn causes a drastic reduction in the pace of scientific discovery and the development, testing, and development of modern deep learning models.

It is now widely understood that *communication bottlenecks* are the main source of this speedups saturation phenomenon (Dean et al., 2012; Seide et al., 2014; Qi et al., 2016; Grubic et al., 2018; Narayanan et al., 2019). These bottlenecks arise due to the size of the messages exchanged among the distributed cluster, *i.e.,* the gradient shares the same size with the model. Moreover, the communication cost arises due to the high frequency of information exchange in the training process, *e.g.,* standard distributed SGD requires to incur one `all-reduce` operation per iteration. Communication bottlenecks become even more pronounced in the recently studied field of FL (Konečnỳ et al., 2016; McMahan et al., 2016; Smith et al., 2017; Kairouz et al., 2019; Wang et al., 2020b, 2021b), where edge devices (*e.g.,* mobile devices, sensors networks powered by embedded systems, and etc.) perform decentralized

training but suffer from low bandwidth during uplink.

For FL, the communication overhead can be even more pronounced as the participating devices are usually unreliable mobile or embedded devices with limited communication bandwidth (McMahan et al., 2017a; Kairouz et al., 2019; Wang et al., 2021b).

Robustness challenges. A recent line of studies reported that centralized distributed ML algorithms are not robust enough. Due to increasingly common malicious attacks, hardware, and software errors (Castro et al., 1999; Kotla et al., 2007; Blanchard et al., 2017a; Chen et al., 2017c), improving the Byzantine resilience of distributed ML algorithms has become increasingly important. Unfortunately, even a single Byzantine compute node in a distributed setup can introduce arbitrary bias and inaccuracies to the end model (Blanchard et al., 2017a; Chen et al., 2017c; Guerraoui et al., 2018).

Across most FL settings, it is assumed that there is no single, central authority that owns or verifies the training data or user hardware, and it has been argued by many recent studies that FL lends itself to new adversarial attacks during decentralized model training (Biggio et al., 2012; Chen et al., 2017b; Blanchard et al., 2017a; Chen et al., 2017c; Koh and Liang, 2017; Liu et al., 2017; Xie et al., 2018b; Bagdasaryan et al., 2018; Bhagoji et al., 2018; Kairouz et al., 2019; Lamport et al., 2019; Xie et al., 2019b; Xie, 2019; Baruch et al., 2019). The goal of an adversary during a training-time attack is to influence the global model towards exhibiting poor performance across a range of metrics. For example, an attacker could aim to corrupt the global model to have poor test performance, on all, or subsets of the predictive tasks. Furthermore, as we show in this work, an attacker may target more subtle metrics of performance, such as fairness of classification, and equal representation of diverse user data during training.

Initiated by the work of (Bagdasaryan et al., 2018), a line of recent literature presents ways to insert backdoors during FL. The goal of a backdoor attack is to corrupt the global FL model into a targeted misprediction on a specific subtask, *e.g.,* by forcing an image classifier to misclassify green cars as frogs (Bagdasaryan

et al., 2018). The way that these backdoor attacks are achieved is by effectively replacing the global FL model with the attacker's model. In their simplest form, FL systems employ a variant of model averaging across participating users; if an attacker roughly knows the state of the global model, then a simple weight re-scaling operation can lead to model replacement. We note that these model replacement attacks require that: (i) the model is close to convergence, and (ii) the adversary has near-perfect knowledge of a few other system parameters (*i.e.,* number of users, data set size, etc.).

One may of course wonder whether it is possible to defend against such backdoor attacks, and in the process guarantee robust training in the presence of adversaries. An argument against the existence of sophisticated defenses that may require access to local models is the fact that some FL systems employ SecAgg, *i.e.,* a secure version of model averaging (Bonawitz et al., 2017). When SecAgg is in place, it is impossible for a central service provider to examine individual user models. However, it is important to note that even in the absence of SecAgg, the service provider is limited in its capacity to determine which model updates are malicious, as this may violate privacy or fairness constraints (Kairouz et al., 2019).

Follow-up work by (Sun et al., 2019) examines simple defense mechanisms that do not require examining local models, and questions the effectiveness of model-replacement backdoors of Bagdasaryan et al. (Bagdasaryan et al., 2018). Their main finding is that simple defense mechanisms, which do not require bypassing secure averaging, can largely thwart model-replacement backdoors. Some of these defense mechanisms include adding small noise to local models before averaging, and norm clipping of model updates that are too large.

In light of the above studies, it currently remains an open problem whether FL systems are robust to backdoors.

This dissertation takes a first step toward improving the communication efficiency and robustness of centralized distributed ML and FL algorithms. We now introduce the solutions in this dissertation.

1.2 Solutions to improve the communication efficiency of distributed ML

1.2.1 ATOMO and PUFFERFISH for centralized setting

A popular approach to tackle the communication overhead in centralized distributed ML is via *gradient compression*, either reducing the precision of the gradients (Seide et al., 2014; De Sa et al., 2015b; Zhou et al., 2016; Konečnỳ et al., 2016; Alistarh et al., 2017; Wen et al., 2017; Zhang et al., 2017b; De Sa et al., 2017, 2018; Bernstein et al., 2018a) or gradient sparsification (Strom, 2015; Mania et al., 2015; Suresh et al., 2016; Konečnỳ and Richtárik, 2016; Leblond et al., 2016; Aji and Heafield, 2017; Lin et al., 2017; Chen et al., 2017a; Renggli et al., 2018; Tsuzuku et al., 2018; Vogels et al., 2019).

In this dissertation, we start by introducing a new gradient sparsification scheme via *atomic decomposition* in Chapter 3. An atomic decomposition represents a vector as a linear combination of simple building blocks in an inner product space. We show that stochastic gradient sparsification and quantization are facets of a general approach that sparsifies a gradient in any possible atomic decomposition, including its entry-wise or singular value decomposition, its Fourier decomposition, and more. With this in mind, we present ATOMO, a general framework for atomic sparsification of stochastic gradients. ATOMO sets up and optimally solves a meta-optimization that minimizes the variance of the sparsified gradient, subject to the constraints that it is sparse on the atomic basis, and also is an unbiased estimator of the input. We show that 1-bit QSGD and TernGrad are in fact special cases of ATOMO, and each is optimal (in terms of variance and sparsity), in different parameter regimes (Alistarh et al., 2017; Wen et al., 2017). Then, we argue that for some neural network applications, viewing the gradient as a concatenation of matrices (each corresponding to a layer), and applying atomic sparsification to their SVD is meaningful and well-motivated by the fact that these matrices are "nearly" low-rank, *e.g.,* see Fig. 1.1. We show that ATOMO on the SVD of each layer's gradient can lead to less variance, and faster training, for the same communication budget as that of QSGD or TernGrad. We

present extensive experiments showing that using ATOMO with SVD sparsification, can lead to up to 2× faster training time (including the time to compute the SVD) compared to QSGD.

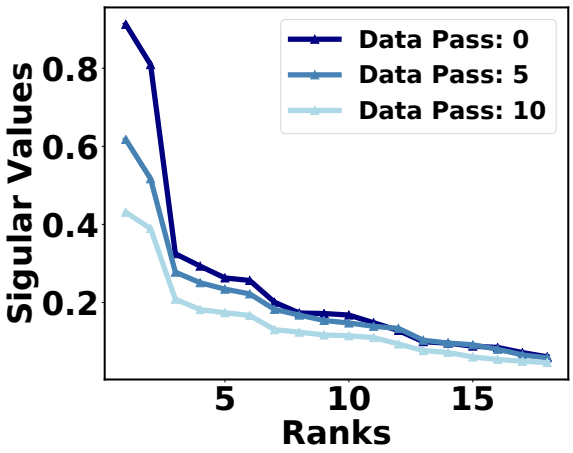


Figure 1.1: The singular values of a convolution layer's gradient, for ResNet-18 while training on CIFAR-10. The gradient of a layer can be seen as a matrix, once we vectorize and appropriately stack the convolutional filters. For all presented data passes, there is a sharp decay in singular values, with the top 3 standing out.

ATOMO achieves high gradient compression ratios and preserves final model accuracy. However, like most of the popular gradient compression methods, ATOMO suffers from some of the following drawbacks: (i) The computation cost for factorizing the gradients using SVD at each iteration is high especially for large-scale models. (ii) ATOMO does not fully utilize the gradients. By adopting the "*error feedback*" scheme (Seide et al., 2014; Stich et al., 2018; Karimireddy et al., 2019), ATOMO can utilize stale gradients aggregated in memory for future iterations, but it requires storing additional information proportional to the model size. (iii) Significant implementation efforts are required to incorporate ATOMO within high-efficiency distributed training APIs in current deep learning frameworks *e.g.*, `DistributedDataParallel` (DDP) in PyTorch (Li et al., 2020a).

Due to the above shortcomings, it is of interest to explore the feasibility of incorporating elements of the gradient compression step into the model architecture

itself. If this is feasible, then communication efficiency can be attained at no extra cost. As the next step towards communication-efficient distributed ML in this dissertation, we propose PUFFERFISH in Chapter 4, which bypasses the gradient compression step via training low-rank, pre-factorized deep networks, starting from full-rank counterparts. We observe that training low-rank models from scratch incurs non-trivial accuracy loss. To mitigate that loss, instead of starting from a low-rank network, we initialize a full-rank counterpart. We train for a small fraction, *e.g.*, 10% of total number epochs, with the full-rank network, and then convert to a low-rank counterpart. To obtain such a low-rank model we apply SVD on each of the layers. After the SVD step, we use the remaining 90% of the training epochs to fine-tune this low-rank model.

PUFFERFISH bears similarities to the "*Lottery Ticket Hypothesis*" (LTH), in that we find "winning tickets" within full-rank/dense models, but without the additional burden of "winning the lottery" (Frankle and Carbin, 2018). Winning tickets seem to be in abundance once we seek models that are sparse in their spectral domain.

PUFFERFISH takes any deep neural network architecture and finds a pre-factorized low-rank representation. PUFFERFISH then trains the pre-factorized low-rank network to achieve both computation and communication efficiency, instead of explicitly compressing gradients. PUFFERFISH supports several types of architectures including fully connected (FC), convolutional neural nets (CNNs), LSTMs, and Transformer networks (Vaswani et al., 2017). As PUFFERFISH manipulates the model architectures instead of their gradients, it is directly compatible with all SotA distributed training frameworks, *e.g.*, PyTorch DDP and BytePS (Li et al., 2020a; Jiang et al., 2020).

We further observe that direct training of those pre-factorized low-rank deep networks leads to non-trivial accuracy loss, especially for large-scale machine learning tasks, *e.g.*, ImageNet (Deng et al., 2009). We develop two techniques for mitigating this accuracy loss: (i) a *hybrid architecture* and (ii) *vanilla warm-up training*. The effectiveness of these two techniques is justified via extensive experiments.

We provide experimental results over real distributed systems and large-scale vision and language processing tasks. We compare PUFFERFISH against a wide range of standard baselines: (i) communication-efficient distributed training methods, *e.g.*,

SIGNUM and POWERSGD (Bernstein et al., 2018a; Vogels et al., 2019); (ii) structured pruning methods, *e.g.,* the *Early Bird Ticket* (EB Train) (You et al., 2019); and model sparsification method, *e.g.,* the iterative pruning algorithm in LTH (Frankle and Carbin, 2018). Our experimental results indicate that PUFFERFISH achieves better model training efficiency compared to POWERSGD, SIGNUM, and LTH. PUFFERFISH also leads to a smaller and more accurate model compared to EB Train. We further show that the performance of PUFFERFISH remains stable under *mixed-precision training*.

1.2.2 Communication-efficient FL via FedMA

One shortcoming of FedAvg is coordinate-wise averaging of weights may have drastic detrimental effects on the performance of the averaged model and adds significantly to the communication burden. This issue arises due to the permutation invariance of neural network (NN) parameters, *i.e.,* for any given NN, there are many variants of it that only differ in the ordering of parameters. Probabilistic Federated Neural Matching (PFNM) addresses this problem by matching the neurons of client NNs before averaging them (Yurochkin et al., 2019b). PFNM further utilizes Bayesian nonparametric methods to adapt to global model size and to heterogeneity in the data. As a result, PFNM has better performance and communication efficiency than FedAvg. Unfortunately, the method only works with simple architectures (*e.g.,* fully connected feedforward networks).

To improve the communication efficiency of FL, we demonstrate how PFNM can be applied to CNNs and LSTMs, but we find that it only gives very minor improvements over weight averaging. To address this issue, in this dissertation, we present Federated Matched Averaging (FedMA) in Chapter 5, a new layers-wise FL algorithm for modern CNNs and LSTMs that appeal to Bayesian non-parametric methods to adapt to heterogeneity in the data. We show empirically that FedMA not only reduces the communication burden, but also outperforms SotA FL algorithms in terms of convergence speed and accuracy.

1.3 Contribution of this dissertation on robust distributed ML

1.3.1 Byzantine-resilient distributed ML via DRACO and DETOX

A recent line of work studies the Byzantine-resilience under a synchronous training setup, where compute nodes evaluate gradient updates and ship them to a parameter server (PS) which stores and updates the global model (Blanchard et al., 2017a; Chen et al., 2017c; Guerraoui et al., 2018). Many of the aforementioned work use median-based aggregation methods, including geometric median (GM) instead of averaging in order to make the training process more robust. The advantage of median-based approaches is they can be robust to up to a constant fraction of the compute nodes being Byzantine nodes (Chen et al., 2017c). However, in applications using the SotA deep neural networks, the computation costs of the median-based aggregation methods can dwarf the cost of computing a batch of gradients, rendering it impractical. Furthermore, proofs of convergence for such systems require restrictive assumptions such as convexity, and need to be re-tailored to each different training algorithm. A scalable distributed training framework that is robust against adversaries and can be applied to a large family of training algorithms (*e.g.,* mini-batch SGD, GD, coordinate descent, SVRG, etc.) remains an open problem.

In this dissertation, we present methods that enhance Byzantine-resilience of centralized distributed ML with provable guarantees. We firstly introduce DRACO in Chapter 6, a general distributed training framework that is robust against adversarial nodes and worst-case compute errors. We show that DRACO can resist any s adversarial compute nodes (as long as s does not exceed half number of compute nodes in a cluster) during training and returns a model that is identical to the one trained in the adversary-free setup. This allows DRACO to come with "black-box" convergence guarantees, *i.e.,* proofs of convergence in the adversary-free setup carry through to the adversarial setup with no modification, unlike prior median-based approaches (Blanchard et al., 2017a; Chen et al., 2017c; Guerraoui

et al., 2018). Moreover, the median computation may dominate the overall training time. In DRACO, most of the computational effort is carried through by the compute nodes. This key factor allows our framework to offer up to orders of magnitude faster convergence in real distributed setups. In standard adversary-free distributed computation setups, during each distributed round, each of the p compute nodes processes $\frac{B}{p}$ gradients and ships their sum to the PS. In DRACO, each compute node processes $\frac{rB}{p}$ gradients and sends a linear combination of those to the PS. Thus DRACO incurs a computational redundancy ratio of r. While this may seem sub-optimal, we show that under a worst-case adversarial setup, it is information-theoretically impossible to design a system that obtains identical models to the adversary-free setup with less redundancy. Upon receiving the p gradient sums, the PS uses a "decoding" function to remove the effect of the adversarial nodes and reconstruct the original desired sum of the B gradients.

With redundancy ratio r, we show that DRACO can tolerate up to $(r-1)/2$ adversaries, which is information theoretically tight. See Fig. 1.2 for a toy example of DRACO's functionality.

We present two encoding and decoding techniques for DRACO. The encoding schemes are based on the fractional repetition code and cyclic repetition code presented in (Tandon et al., 2017; Raviv et al., 2017). In contrast to previous work on stragglers and gradient codes (Tandon et al., 2017; Raviv et al., 2017; Charles et al., 2017), our decoders are tailored to the adversarial setting and use different methods. Our decoding schemes utilize an efficient majority vote decoder and a novel Fourier decoding technique.

Compared to median-based techniques that can tolerate approximately a constant fraction of "average case" adversaries, DRACO's $(r-1)/2$ bound on the number of "worst-case" adversaries may be significantly smaller. However, in realistic regimes where only a constant number of nodes are malicious, DRACO is significantly faster.

Our extensive experimental results show that DRACO is up to orders of magnitude faster compared to GM-based approaches across a range of large-scale neural networks and always converges to the correct adversary-free model, while in some

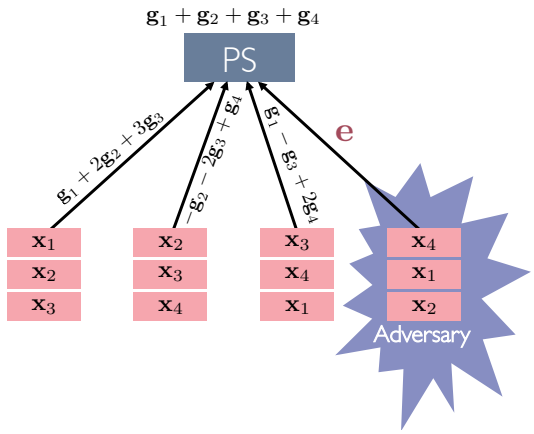


Figure 1.2: The high level idea behind DRACO's algorithmic redundancy. Suppose we have 4 data points $\mathbf{x}_1, \ldots, \mathbf{x}_4$, and let $\mathbf{g}_i$ be the gradient of the model with respect to data point $\mathbf{x}_i$. Instead of having each compute node i evaluate a single gradient $\mathbf{g}_i$, DRACO assigns each node redundant gradients. In this example, the replication ratio is 3, and the PS can recover the sum of the gradients from any 2 of the encoded gradient updates. Thus, the PS can still recover the sum of gradients in the presence of an adversary. This can be done through a majority vote on all 6 pairs of encoded gradient updates. This intuitive idea does not scale to a large number of compute nodes. DRACO implements a more systematic and efficient encoding and decoding mechanism that scales to any number of machines.

cases median-based approaches do not converge.

DRACO offers black-box gradient recovery guarantees. Unfortunately, DRACO may, in the worst case, require each node to compute $\Omega(s)$ times more gradients, where s is the number of Byzantine nodes. This overhead is prohibitive in settings with large numbers of Byzantine nodes.

To improve the computation efficiency of DRACO on the compute node side, we present DETOX in Chapter 7, a Byzantine-resilient distributed training framework that first uses computational redundancy to filter out almost all Byzantine gradients, and then performs a hierarchical robust aggregation method. DETOX is scalable, flexible, and is designed to be used on top of any robust aggregation method to obtain improved robustness and efficiency. A high-level description of the hierarchical nature of DETOX is given in Fig. 1.3.

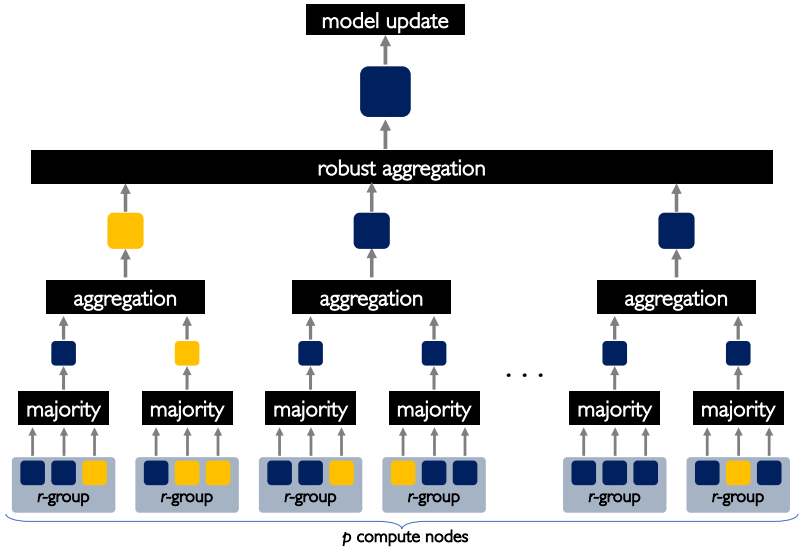


Figure 1.3: DETOX is a hierarchical scheme for Byzantine gradient aggregation. In its first step, the PS partitions the compute nodes in groups and assigns each node to a group with the same batch of data. After the nodes compute gradients with respect to this batch, the PS takes a majority vote of their outputs. This filters out a large fraction of the Byzantine gradients. In the second step, the PS partitions the filtered gradients in large groups, and applies a given aggregation method to each group. In the last step, the PS applies a robust aggregation method (*e.g.*, geometric median) to the previous outputs. The final output is used to perform a gradient update step.

DETOX proceeds in three steps. First the PS partitions the compute nodes in groups of r to compute the same gradients. While this step requires redundant computation at the node level, it will eventually allow for much faster computation at the PS level, as well as improved robustness. After all compute nodes send their gradients to the PS, the PS takes the majority vote of each group of gradients. We show that by setting r to be logarithmic in the number of compute nodes, after the majority vote step only a constant number of Byzantine gradients are still present, even if the number of Byzantine nodes is a *constant fraction* of the total number of compute nodes. DETOX then performs a hierarchical robust aggregation in two steps: First, it partitions the filtered gradients in a small number of groups, and

aggregates them using simple techniques such as averaging. Second, it applies any robust aggregator (*e.g.,* geometric median (Chen et al., 2017c; Yin et al., 2018a), Bulyan (Guerraoui et al., 2018), Multi-Krum (Damaskinos et al., 2019), etc.) to the averaged gradients to further minimize the effect of any remaining traces of the original Byzantine gradients.

We prove that Detox can obtain *orders of magnitude* improved robustness guarantees compared to its competitors, and can achieve this at a nearly linear complexity in the number of compute nodes p, unlike methods like Bulyan (Guerraoui et al., 2018) that require complexity that is quadratic in p.

We extensively test our method in real distributed setups and large-scale settings, showing that by combining Detox with previously proposed Byzantine robust methods, such as Multi-Krum, Bulyan, and coordinate-wise median, we increase the robustness and reduce the overall runtime of the algorithm. Moreover, we show that under strong Byzantine attacks, Detox can lead to almost a 40% increase in accuracy over vanilla implementations of Byzantine-robust aggregation.

1.3.2 The robustness of FL

In this dissertation, we show evidence the FL is vulnerable to training time backdoor attacks in Chapter 8. Defense mechanisms as presented in (Sun et al., 2019), along with more intricate ones based on robust aggregation (Blanchard et al., 2017a), can be circumvented by appropriately designed backdoors. Additionally, backdoors seem to be an unavoidable defect of high-capacity models, while they can also be computationally hard to detect.

We first theoretically establish that if a model is vulnerable to inference-time attacks in the form of adversarial examples (Goodfellow et al., 2014; Papernot et al., 2016; Moosavi-Dezfooli et al., 2016; Carlini and Wagner, 2017; Athalye et al., 2018), then, under mild conditions, the same model will be vulnerable to backdoor training-time attacks. If these backdoors are crafted properly (*i.e.,* targeting low probability, or *edge-case* samples), then they can also be hard to detect.

Based on cues from our theory, and inspired by the work of (Bagdasaryan

et al., 2018), we introduce a new class of backdoor attacks that are resistant to current defenses and can lead to unsavory classification outputs and affect the fairness properties of the learned classifiers. We refer to these attacks as *edge-case backdoors*. Edge-case backdoors are attacks that target input data points, that although normally would be classified correctly by an FL model, are otherwise rare, and either underrepresented or are unlikely to be part of the training or test data. See Figure 1.4 for examples.

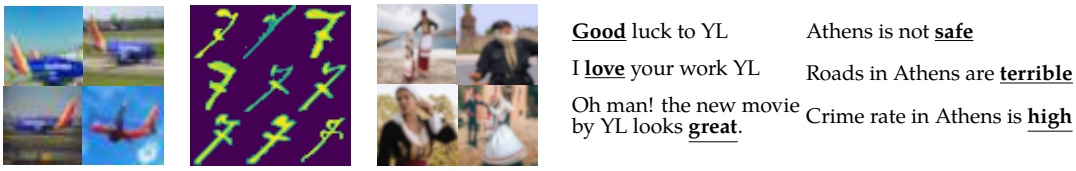


Figure 1.4: Illustration of tasks and edge-case examples for our backdoors. Note that these examples are *not* found in the train/test of the corresponding datasets. (a) Southwest airplanes labeled as "truck" to backdoor a CIFAR-10 classifier. (b) Images of "7" from the ARDIS dataset labeled as "1" to backdoor an MNIST classifier. (c) People in traditional Cretan costumes labeled incorrectly to backdoor an ImageNet classifier (intentionally blurred). (d) Positive tweets on the director Yorgos Lanthimos (YL) labeled as "negative" to backdoor a sentiment classifier. (e) Sentences regarding Athens completed with words of negative connotation to backdoor a next word predictor.

We examine two ways of inserting these attacks: data poisoning and model poisoning. In the data poisoning (*i.e.*, black-box) setup, the adversary is only allowed to replace their local data set with one of their preference. Similar to (Gu et al., 2017; Bagdasaryan et al., 2018; Koh et al., 2018), in this case, a mixture of clean and backdoor data points is inserted in the attacker's data set; the backdoor data points target a specific class, and use a preferred target label. In the model poisoning (*i.e.*, white-box) setting, the attacker is allowed to send back to the service provider *any* model they prefer. This is the setup that (Bagdasaryan et al., 2018; Bhagoji et al., 2018) focus on. In (Bhagoji et al., 2018) the authors take an adversarial perspective

during training, and replace the local attackers metric with one that targets a specific subtask, and resort to using proximal based methods to approximate these tasks. In this dissertation, we employ a similar but algorithmically different approach. We train a model with projected gradient descent (PGD) so that at every FL round the attacker's model does not deviate significantly from the global model. The effect of the PGD attack, also suggested in (Sun et al., 2019) as stronger than vanilla model replacement, exhibits an increased resistance against a range of defense mechanisms.

We show across a suite of prediction tasks (image classification, OCR, sentiment analysis, and text prediction), data sets (CIFAR10/ImageNet /EMNIST/Reddit/Sentiment140), and models (VGG-9/VGG-11/LeNet /LSTMs) that our edge-case attacks can be hard-wired in FL models, as long as 0.5–1% of the total number of edge users are adversarial. We further show that these attacks are robust to defense mechanisms based on differential privacy (DP) (McMahan et al., 2017b; Sun et al., 2019), norm clipping (Sun et al., 2019), and robust aggregators such as Krum and Multi-Krum (Blanchard et al., 2017a). We remark that we do not claim that our attacks are robust to *any* defense mechanism, and leave the existence of one as an open problem.

The implication of edge-case backdoors. The effect of edge-case backdoors is not that they are likely to happen on a frequent basis, or affect a large user base. Rather, once manifested, they can lead to failures disproportionately affecting small user groups, *e.g.,* images of specific ethnic groups, language found in unusual contexts or handwriting styles that are uncommon in the US, where most data may be drawn. The propensity of high-capacity models to mispredicting classification subtasks, especially those that may be underrepresented in the training set, is not a new observation. For example, several recent reports indicate that neural networks can mispredict inputs of underrepresented minority individuals by attaching offensive labels (Simonite, 2018). Failures involving edge-case inputs have also been a point of grave concern for the safety of autonomous vehicles (Krafcik, 2018; Hawkins, 2019).

This dissertation indicates that edge-case failures of this manner can, unfortunately, be hard-wired through backdoors to FL models. Moreover, as we show, attempts to filter out potential attackers inserting these backdoors, have the adverse effect of also filtering out users that simply contain diverse enough data sets, presenting an unexplored fairness and robustness trade-off, which was suggested in (Kairouz et al., 2019). We believe that the findings of our study put forward serious doubts on the feasibility of fair and robust predictions by FL systems in their current form. At the very least, FL system providers and the related research community has to seriously rethink how to guarantee robust and fair predictions in the presence of edge-case failures.

1.4 Organization

The rest of the dissertation is organized as follows. We discuss improving the communication efficiency of centralized distributed training in Chapter 3 and Chapter 4 by presenting the design, analysis, and experimental evaluation of the Atomo and Pufferfish frameworks. Chapter 3 is based on joint work with Scott Sievert, Zachary Charles, Shengchao Liu, Stephen Wright, and Dimitris Papailiopoulos. And this work was published in 32nd Conference on Neural Information Processing Systems (NeurIPS 2018) (Wang et al., 2018). Chapter 4 is based on joint work with Saurabh Agarwal, and Dimitris Papailiopoulos. And this work was published in Proceedings of the 4th MLSys Conference (MLSys 2021) (Wang et al., 2021a). We discuss the design and analysis of the FedMA algorithm on improving the communication efficiency of FL in Chapter 5. Chapter 5 is based on joint work with Mikhail Yurochkin, Yuekai Sun, Dimitris Papailiopoulos, and Yasaman Khazaeni. And this work was published in 8-th International Conference on Learning Representations (ICLR 2020) with an oral presentation (Wang et al., 2020b).

The Byzantine-resilience challenge in centralized distributed training is addressed in Chapter 6 and Chapter 7 where we present two frameworks, *i.e.,* Draco and Detox that leverage algorithmic redundancy to provide strong Byzantine resilience. Chapter 6 is based on joint work with Lingjiao Chen, Zachary Charles,

and Dimitris Papailiopoulos. And this work was published in Proceedings of the 35th International Conference on Machine Learning (ICML 2018) (Chen et al., 2018). Chapter 7 is based on joint work with Shashank Rajput, Zachary Charles, and Dimitris Papailiopoulos. And this work was published in 33rd Conference on Neural Information Processing Systems (NeurIPS 2019) (Rajput et al., 2019). The robustness of FL is discussed in Chapter 8. Chapter 8 is based on joint work with Kartik Sreenivasan, Shashank Rajput, Harit Vishwakarma, Saurabh Agarwal, Jy-yong Sohn, Kangwook Lee, Dimitris Papailiopoulos. And this work was published in 34th Conference on Neural Information Processing Systems (NeurIPS 2020) (Wang et al., 2020a).

2.1 Communication-efficient distributed ML

2.1.1 Centralized communication-efficient distributed ML

Communication-efficient distributed mean estimation has been widely investigated in (Konečnỳ and Richtárik, 2018) and (Suresh et al., 2017). They both note, as this dissertation does, that variance (or equivalently the mean squared error) controls important quantities such as convergence, and they seek to find a low-communication vector averaging scheme that minimizes it. This dissertation differs in two key aspects. First, we derive a closed-form solution to the variance minimization problem for all input gradients. Second, Atomo applies to any atomic decomposition, which allows us to compare entry-wise against singular value sparsification for matrices. Using this, we derive explicit conditions for which SVD sparsification leads to lower variance for the same sparsity budget.

The idea of viewing gradient sparsification through a meta-optimization lens was also used in (Wangni et al., 2018). Atomo differs in two key ways. First, (Wangni et al., 2018) consider the problem of minimizing the sparsity of a gradient for a fixed variance, while we consider the reverse problem, that is, minimizing the variance subject to a sparsity budget. The second more important difference is that while (Wangni et al., 2018) focuses on entry-wise sparsification, we consider a general problem where we sparsify according to any atomic decomposition. For instance, our approach directly applies to sparsifying the singular values of a matrix, which gives rise to faster training algorithms.

Finally, low-rank factorizations and sketches of the gradients when viewed as matrices were proposed in (Xue et al., 2013; Sainath et al., 2013; Jaderberg et al., 2014; Wiesler et al., 2014; Konečnỳ et al., 2016); arguably most of these methods (with the exception of (Konečnỳ et al., 2016)) aimed to address the high flops required during inference by using low-rank models. Though they did not directly aim to reduce communication, this arises as a useful side effect.

To reduce the communication cost in distributed training, the related literature has developed several methods for gradient compression. Some of the methods use quantization over the gradient elements (Seide et al., 2014; Alistarh et al., 2017; Wen et al., 2017; Lin et al., 2017; Luo et al., 2017; Bernstein et al., 2018a; Tang et al., 2019; Wu et al., 2018). Other methods study sparsifying the gradients in the element-wise or spectral domains (Lin et al., 2017; Wang et al., 2018; Stich et al., 2018; Vogels et al., 2019). It has also been widely observed that adopting the "*error feedback*" scheme is generally helpful for gradient compression methods to achieve better final model accuracy (Stich et al., 2018; Wu et al., 2018; Karimireddy et al., 2019; Vogels et al., 2019). Compared to the previously proposed gradient compression methods, Pufferfish merges the gradient compression into model training, thus achieves high communication efficiency at no extra cost.

Partially initialized by *deep compression* (Han et al., 2015), a lot of research proposes to remove the redundant weights in the trained neural networks. The trained neural networks can be compressed via model weight pruning (Li et al., 2016; Wen et al., 2016; Hu et al., 2016; Zhu and Gupta, 2017; He et al., 2017; Yang et al., 2017a; Liu et al., 2018; Yu et al., 2018b,a), quantization (Rastegari et al., 2016; Zhu et al., 2016; Hubara et al., 2016; Wu et al., 2016; Hubara et al., 2017; Zhou et al., 2017), and low-rank factorization (Xue et al., 2013; Sainath et al., 2013; Jaderberg et al., 2014; Wiesler et al., 2014; Konečnỳ et al., 2016). Different model compression, Pufferfish proposes to train the factorized networks, which achieves better overall training time, rather than compressing the model after fully training it.

For smaller model size and higher efficiency (Iandola et al., 2016; Chen et al., 2016b; Zhang et al., 2018; Tan and Le, 2019; Howard et al., 2017; Chollet, 2017; Lan et al., 2019; Touvron et al., 2020; Waleffe and Rekatsinas, 2020) propose to re-design the neural networks. The most related low-rank efficient training framework to Pufferfish is the one proposed in (Ioannou et al., 2015), where a pre-factorized network is trained from scratch. However, we demonstrate that training the factorized network from scratch leads to non-trivial accuracy loss. In Pufferfish, we propose to warm up the low-rank model via factorizing a partially trained full-rank model. Our extensive experiments indicate that Pufferfish achieves significantly higher

accuracy compared to training the factorized network from scratch. Moreover, (Ioannou et al., 2015) only studies low-rank factorizations for convolutional layers, whereas Pufferfish supports FC, CNN, LSTM, and Transformer layers.

2.1.2 Communication efficient federated learning

FL allows participating clients to train a joint model without uploading their local data (mostly private) to the data center to protect user data privacy (McMahan et al., 2017a; Kairouz et al., 2019). To conduct FL, FedAvg is the first algorithm that was introduced, which aims at enhancing data privacy and communication efficiency (McMahan et al., 2017a). (Li et al., 2020b) provides rigorous theoretical analysis on FedAvg. Extending FedAvg, FedProx adds a proximal term to the clients' objective functions, thereby limiting the impact of local updates by keeping them close to the global model (Sahu et al., 2018). On the data center side, FedProx conducts the same coordinate-wise model weights averaging for model aggregation. Agnostic Federated Learning (AFL), as another variant of FedAvg, optimizes a centralized distribution that is a mixture of the client distributions (Mohri et al., 2019). Compared to FedAvg and FedProx, PFNM replaces the data center side coordinate-wise averaging by matched averaging and proposes a Bayesian nonparametric approach to solve the neural matching problem across multiple model updates, which improves the model aggregation performance for several ML tasks (Yurochkin et al., 2019b). PFNM only supports fully connected neural networks, FedMA extends the neural matching scheme of PFNM to convolution and LSTM neural networks. Moreover, the one-shot neural matching scheme in PFNM performs not well on deep neural networks, *e.g.,* VGG-9. FedMA improves PFNM by introducing a novel layer-wise neural matching scheme. Our results indicate that FedMA significantly outperforms PFNM on deep neural networks.

Cross-device FL is one of the most important application scenarios of FL where the participating clients are usually edge devices, *e.g.,* mobile phones and embedded system devices (Bonawitz et al., 2019). As the bandwidth and hardware resources are usually limited, the communication cost becomes the main bottleneck

in FL (McMahan et al., 2017a; Kairouz et al., 2019; Wang et al., 2021b). (Konečnỳ et al., 2016) provides a thorough study on improving the communication efficiency of FL via quantization, sparsification, and factorization of the local model updates. Compared to compressing the trained local model weights directly, FedMA improves the communication efficiency of FL by significantly reducing the number of FL rounds required for the global model to converge.

2.2 Robust distributed ML

2.2.1 Centralized robust distributed ML

The large-scale nature of modern machine learning has spurred a great deal of novel research on distributed and parallel training algorithms and systems (Recht et al., 2011; Dean et al., 2012; Jaggi et al., 2014; Liu et al., 2014; Mania et al., 2015; Chen et al., 2016a; Alistarh et al., 2017). Much of this work focuses on developing and analyzing efficient distributed training algorithms. This work shares ideas with FL, in which training is distributed among a large number of compute nodes without centralized training data (Konečnỳ et al., 2015, 2016; Bonawitz et al., 2016).

Synchronous training can suffer from straggler nodes (Zaharia et al., 2008), where a few compute nodes are significantly slower than average. While early work on straggler mitigation used techniques such as job replication (Shah et al., 2016), more recent work has employed coding theory to speed up distributed machine learning systems (Li et al., 2015; Dutta et al., 2016, 2017; Yang et al., 2017b; Lee et al., 2018a; Reisizadeh et al., 2019). One notable technique is *gradient coding*, a straggler mitigation method proposed in (Tandon et al., 2017), which uses codes to speed up synchronous distributed first-order methods (Cotter et al., 2011; Raviv et al., 2017; Charles et al., 2017). Our work builds on and extends this work to the adversarial setup. Mitigating adversaries can often be more difficult than mitigating stragglers since we have no knowledge as to which nodes are the adversaries.

The topic of Byzantine fault tolerance has been extensively studied since the early 80s by Lamport et al. (Lamport et al., 1982), and deals with worst-case, and/or adver-

sarial failures, *e.g.,* system crashes, power outages, software bugs, and adversarial agents. In the context of distributed optimization, these failures are manifested through a subset of compute nodes returning to the master flawed or adversarial updates. It is now well understood that first-order methods, such as gradient descent or mini-batch SGD, are not robust to Byzantine errors; even a single erroneous update can introduce arbitrary errors to the optimization variables.

Byzantine-tolerant ML has been extensively studied in recent years (Blanchard et al., 2017a; Chen et al., 2017c; Xie et al., 2018b; El-Mhamdi et al., 2019; Xie et al., 2019a; El-Mhamdi and Guerraoui, 2019), establishing that while average-based gradient methods are susceptible to adversarial nodes, median-based update methods can in some cases achieve better convergence, while being robust to some attacks. Although theoretical guarantees are provided in many works, the proposed algorithms in many cases only ensure a weak form of resilience against Byzantine failures, and often fail against strong Byzantine attacks (Guerraoui et al., 2018). A stronger form of Byzantine resilience is desirable for most distributed machine learning applications. Bulyan (Guerraoui et al., 2018) guarantees strong Byzantine resilience. Bulyan requires heavy computation overhead on the PS end.

We note that (Alistarh et al., 2018) presents an alternative approach that does not fit easily under either category, but requires convexity of the underlying loss function. Finally, (Bernstein et al., 2018a) examines the robustness of signSGD with a majority vote aggregation, but studies a restricted Byzantine failure setup that only allows for a blind multiplicative adversary.

The idea of using redundancy to guard against failures in computational systems has existed for decades. Von Neumann used redundancy and majority vote operations in boolean circuits to achieve accurate computations in the presence of noise with high probability (Von Neumann, 1956). These results were further extended in work such as (Pippenger, 1988) to understand how susceptible a boolean circuit is to randomly occurring failures. Draco and Detox can be seen as an application of the aforementioned concepts to the context of distributed training in the face of adversity. Inspired by the solutions of this dissertation, a recent redundancy-based method *ByShield* leverages the properties of bipartite expander graphs for

the assignment of tasks to workers to achieve Byzantine resilience.

2.2.2 Robustness of federated learning

Some recent work considers training time attacks for FL, identifying its weaknesses (Cao et al., 2019; Zhang et al., 2019; Hong et al., 2020).

Data poisoning has been extensively studied for traditional ML pipelines and is closely related to backdoors. It usually relies on modifying training data to influence predictions at inference time (Rubinstein et al., 2009; Huang et al., 2011; Chen et al., 2017b; Gu et al., 2017; Liu et al., 2017; Steinhardt et al., 2017; Mahloujifar et al., 2018). Trigger-based attacks such as those proposed by (Gu et al., 2017) have also been shown to be effective and readily extend to the FL setting. However, these require both training-time and inference-time access to input data in order to insert the pixel-pattern trigger that the poisoned model is trained to identify. We focus on trigger-less attacks in this work, but (Bagdasaryan et al., 2018) show that model replacement extends to the trigger-based setting as well.

Typical defenses against data poisoning attacks involve *Data Sanitization* (Cretu et al., 2008) which uses outlier detection (Hodge and Austin, 2004), however (Koh et al., 2018) shows that these defenses can be overcome.

Machine teaching is closely related to data poisoning (Zhu et al., 2018), and is the process by which one designs training data to drive a learning algorithm to a target model. Although it is typically used to speed up training (Zhu, 2013; Lessard et al., 2018; Ma et al., 2018), it can also be used to force the learner into a model with backdoors (Alfeld et al., 2016; Mei and Zhu, 2015, 2017).

Model replacement attacks are related to *Byzantine gradient attacks* (Blanchard et al., 2017a), mostly studied in the context of centralized, distributed learning. Defense mechanisms for distributed Byzantine ML draw ideas from robust estimation (Chen et al., 2017d; Su and Vaidya, 2016b,a; Blanchard et al., 2017a; Yin et al., 2019, 2018a; Damaskinos et al., 2019; Xie et al., 2018b; Alistarh et al., 2018), coding theory (Chen et al., 2018; Sohn et al., 2019) or a mixture of the two (Rajput et al., 2019).

Part II

Communication-efficient Distributed ML

3 ATOMO COMMUNICATION-EFFICIENT LEARNING VIA ATOMIC SPARSIFICATION

Distributed model training suffers from communication overheads due to frequent gradient updates transmitted between compute nodes. To mitigate these overheads, several studies propose the use of sparsified stochastic gradients. We argue that these are facets of a general sparsification method that can operate on any possible *atomic decomposition*. Notable examples include element-wise, singular value, and Fourier decompositions. In this chapter, we present ATOMO, a general framework for atomic sparsification of stochastic gradients. Given a gradient, an atomic decomposition, and a sparsity budget, ATOMO gives a random unbiased sparsification of the atoms minimizing variance. We show that recent methods such as QSGD and TernGrad are special cases of ATOMO and that sparsifiying the singular value decomposition of neural networks gradients, rather than their coordinates, can lead to significantly faster distributed training.

3.1 Problem setup

In machine learning, we often wish to find a model w minimizing the *empirical risk*

$$f(w) = \frac{1}{n}\sum_{i=1}^{n} \ell(w; x_i) \tag{3.1}$$

where $x_i \in \mathbb{R}^d$ is the i-th data point. One way to approximately minimize $f(w)$ is by using stochastic gradient methods that operate as follows:

$$w_{k+1} = w_k - \gamma \widehat{g}(w_k)$$

where w_0 is some initial model, γ is the step size, and $\widehat{g}(w)$ is a stochastic gradient of $f(w)$, *i.e.*, it is an unbiased estimate of the true gradient $g(w) = \nabla f(w)$. Mini-batch SGD, one of the most common algorithms for distributed training, computes $\widehat{g}$

as an average of B gradients, each evaluated on randomly sampled data from the training set. Mini-batch SGD is easily parallelized in the parameter server (PS) setup, where a PS stores the global model, and P compute nodes split the effort of computing the B gradients. Once the PS receives these gradients, it applies them to the model, and sends it back to the compute nodes.

To prove convergence bounds for stochastic-gradient based methods, we usually require $\widehat{g}(w)$ to be an unbiased estimator of the full-batch gradient, and to have small second moment $\mathbb{E}\|\widehat{g}(w)\|^2$, as this controls the speed of convergence. To see this, suppose w^* is a critical point of f, then we have

$$\begin{aligned}&\mathbb{E}[\|w_{k+1} - w^*\|_2^2]\\ =&\mathbb{E}[\|w_k - w^*\|_2^2] - \underbrace{\left(2\gamma\langle\nabla f(w_k), w_k - w_*\rangle - \gamma^2\mathbb{E}[\|\widehat{g}(w_k)\|_2^2]\right)}_{\text{progress at step } t}.\end{aligned}$$

Thus, the progress of the algorithm at a single step is, in expectation, controlled by the term $\mathbb{E}[\|\widehat{g}(w_k)\|]_2^2$; the smaller it is, the bigger the progress. This is a well-known fact in optimization, and most convergence bounds for stochastic-gradient based methods, including mini-batch, involve upper bounds on $\mathbb{E}[\|\widehat{g}(w_k)\|]_2^2$ in convex and nonconvex settings (Cotter et al., 2011; Recht et al., 2011; Ghadimi and Lan, 2013; Bubeck et al., 2015; De Sa et al., 2015a; Reddi et al., 2016; Karimi et al., 2016; De et al., 2016; Yin et al., 2018c). In short, recent results on low-communication variants of SGD design unbiased quantized or sparse gradients, and try to minimize their variance (Alistarh et al., 2017; Konečnỳ and Richtárik, 2018; Wangni et al., 2018). Note that when we require the estimate to be unbiased, minimizing the variance is equivalent to minimizing the second moment.

Since variance is a proxy for speed of convergence, in the context of communication-efficient stochastic gradient methods, one can ask: *What is the smallest possible variance of an unbiased stochastic gradient that can be represented with* k *bits?* Note that under the unbiased assumption, minimizing variance is equivalent to minimizing the second moment of the random vector. This meta-optimization can be cast as

the following meta-optimization:

$$\begin{aligned} &\min_{\widehat{g}} \mathbb{E}\|\widehat{g}(w)\|^2 \\ \text{s.t. } &\mathbb{E}[\widehat{g}(w)] = g(w) \\ &\widehat{g}(w) \text{ can be expressed with } k \text{ bits} \end{aligned}$$

Here, the expectation is taken over the randomness of $\widehat{g}$. We are interested in designing a stochastic approximation $\widehat{g}$ that "solves" this optimization. However, it seems difficult to design a formal, tractable version of the last constraint. In the next section, we replace this with a simpler constraint that instead requires that $\widehat{g}(w)$ is sparse with respect to a given atomic decomposition.

3.2 Atomo: atomic decomposition and sparsification

Let $(V, \langle \cdot, \cdot \rangle)$ be an inner product space over $\mathbb{R}$ and let $\|\cdot\|$ denote the induced norm on V. In what follows, you may think of g as a stochastic gradient of the function we wish to optimize. An *atomic decomposition* of g is any decomposition of the form $g = \sum_{a \in \mathcal{A}} \lambda_a a$ for some set of atoms $\mathcal{A} \subseteq V$. Intuitively, $\mathcal{A}$ consists of simple building blocks. We will assume that for all $a \in \mathcal{A}$, $\|a\| = 1$, as this can be achieved by a positive rescaling of the λ_a.

An example of an atomic decomposition is the entry-wise decomposition $g = \sum_i g_i e_i$ where $\{e_i\}_{i=1}^n$ is the standard basis. More generally, any orthonormal basis of V gives rise to a unique atomic decomposition of any $g \in V$. While we focus on finite-dimensional vectors, one could use Fourier and wavelet decompositions in this framework for infinite-dimensional spaces. When considering matrices, the singular value decomposition gives an atomic decomposition in the set of rank-1 matrices. More general atomic decompositions have found uses in a variety of situations, including solving linear inverse problems (Chandrasekaran et al., 2012).

We are interested in finding an approximation to g with fewer atoms. Our primary motivation is that this reduces communication costs, as we only need to

send atoms with non-zero weights. We can use whichever decomposition is most amenable for sparsification. For instance, if X is a low rank matrix, then its singular value decomposition is naturally sparse, so we can save communication costs by sparsifying its singular value decomposition instead of its entries.

Suppose $\mathcal{A} = \{a_i\}_{i=1}^n$ and we have an atomic decomposition $g = \sum_{i=1}^n \lambda_i a_i$. We wish to find an unbiased estimator $\widehat{g}$ of g that is sparse in these atoms, and with small variance. Since $\widehat{g}$ is unbiased, minimizing its variance is equivalent to minimizing $\mathbb{E}[\|\widehat{g}\|^2]$. We use the following estimator:

$$\widehat{g} = \sum_{i=1}^n \frac{\lambda_i t_i}{p_i} a_i \tag{3.2}$$

where $t_i \sim \text{Bernoulli}(p_i)$, for $0 < p_i \leqslant 1$. We refer to this sparsification scheme as *atomic sparsification*. Note that the t_i's are independent. Recall that we assumed above that $\|a_i\|^2 = 1$ for all a_i. We have the following lemma about $\widehat{g}$.

Lemma 3.1. *If* $g = \sum_{i=1}^n \lambda_i a_i$ *is an atomic decomposition then* $\mathbb{E}[\widehat{g}] = g$ *and*

$$\mathbb{E}[\|\widehat{g}\|^2] = \sum_{i=1}^n \frac{\lambda_i^2}{p_i} + \sum_{i \neq j} \lambda_i \lambda_j \langle a_i, a_j \rangle.$$

Let $\lambda = [\lambda_1, \ldots, \lambda_n]^T$, $p = [p_1, \ldots, p_n]^T$. In order to ensure that this estimator is sparse, we fix some *sparsity budget* s. That is, we require $\sum_i p_i = s$; note that this a *sparsity on average* constraint. We wish to minimize $\mathbb{E}[\|\widehat{g}\|^2]$ subject to this constraint. By Lemma 3.1, this is equivalent to

$$\min_p \sum_{i=1}^n \frac{\lambda_i^2}{p_i} \quad \text{subject to } 0 < p_i \leqslant 1, \quad \sum_{i=1}^n p_i = s. \tag{3.3}$$

An equivalent form of this optimization problem was previously presented in (Konečnỳ and Richtárik, 2018) (Section 6.1). The authors considered this problem for entry-wise sparsification and found a closed-form solution for $s \leqslant \|\lambda\|_1 / \|\lambda\|_\infty$. We give a version of their result but extend this to a closed-form solution for all s.

A similar optimization problem was given in (Wangni et al., 2018), which instead minimizes sparsity subject to a variance constraint.

Algorithm 1 ATOMO probabilities

Input : $\lambda \in \mathbb{R}^n$ with $|\lambda_1| \geqslant \dots |\lambda_n|$; sparsity budget s such that $0 < s \leqslant n$.
Output: $p \in \mathbb{R}^n$ solving equation 3.3.

```
i = 1;
while i ⩽ n do
    if |λ_i| s ⩽ Σ_{j=i}^{n} |λ_i| then
        for k = i, ..., n do
            p_k = |λ_k| s (Σ_{j=i}^{n} |λ_i|)^{-1};
        end
        i = n + 1;
    else
        p_i = 1, s = s − 1;
        i = i + 1;
    end
end
```

We will show that the Algorithm 1 produces a probability vector $p \in \mathbb{R}^n$ solving equation 3.3 for $0 < s \leqslant n$. While we show in Appendix 3.8 that this result can be derived using the KKT conditions, we use an alternative method that focuses on a relaxation of equation 3.3 in order to better understand the structure of the problem. This approach has the added benefit of shedding light on what variance is achieved by solving equation 3.3.

Note that equation 3.3 has a non-empty feasible set only for $0 < s \leqslant n$. Define $f(p) := \sum_{i=1}^{n} \lambda_i^2 / p_i$. To understand how to solve equation 3.3, we first consider the following relaxation:

$$\min_p \sum_{i=1}^{n} \frac{\lambda_i^2}{p_i} \quad \text{subject to } 0 < p_i, \quad \sum_{i=1}^{n} p_i = s. \tag{3.4}$$

We have the following lemma about the solutions to equation 3.4, first shown in

(Konečnỳ and Richtárik, 2018).

Lemma 3.2 ((Konečnỳ and Richtárik, 2018)). *Any feasible vector* p *to equation 3.4 satisfies* $f(p) \geqslant \frac{1}{s}\|\lambda\|_1^2$. *This is achieved iff*

$$p_i = \frac{|\lambda_i| s}{\|\lambda\|_1}. \tag{3.5}$$

Lemma 3.2 implies that if we ignore the constraint that $p_i \leqslant 1$, then the optimal p is achieved by setting $p_i = |\lambda_i| s / \|\lambda\|_1$. If the quantity in the right-hand side is greater than 1, this does not give us an actual probability. This leads to the following definition.

Definition 3.3. *An atomic decomposition* $g = \sum_{i=1}^{n} \lambda_i a_i$ *is* s*-unbalanced at entry* i *if* $|\lambda_i| s > \|\lambda\|_1$.

Fix the atomic decomposition of g. If there are no s-unbalanced entries then we say that the g is s-balanced. We have the following lemma which guarantees that g is s-balanced for s not too large.

Lemma 3.4. *An atomic decomposition* $g = \sum_{i=1}^{n} \lambda_i a_i$ *is* s*-balanced iff* $s \leqslant \|\lambda\|_1 / \|\lambda\|_\infty$.

Lemma 3.2 gives us the optimal way to sparsify s-balanced vectors, since the p that is optimal for equation 3.4 is feasible for equation 3.3. Moreover, the iff condition in Lemma 3.2 implies that the optimal assignment of the p_i are between 0 and 1 iff ν is s-balanced. Suppose now that g is s-unbalanced at entry j. We cannot assign p_j as in equation 3.5. We will show that setting $p_j = 1$ is optimal in this setting. This comes from the following lemma.

Lemma 3.5. *Suppose that* g *is* s*-unbalanced at entry* j *and that* q *is feasible in equation 3.3. Then* $\exists p$ *that is feasible in equation 3.3 such that* $f(p) \leqslant f(q)$ *and* $p_j = 1$.

Lemmas 3.2 and 3.5 imply the following theorem about solutions to equation 3.3.

Theorem 3.6. *Suppose we sparsify* g *as in equation 3.2 with sparsity budget* s.

1. *If* g *is* s*-balanced, then*

$$\mathbb{E}[\|\widehat{g}\|^2] \geqslant \frac{1}{s}\|\lambda\|_1^2 + \sum_{i \neq j} \lambda_i \lambda_j \langle a_i, a_j \rangle$$

   with equality if and only if $p_i = |\lambda_i| s / \|\lambda\|_1$.

2. *If* g *is* s*-unbalanced, then*

$$\mathbb{E}[\|\widehat{g}\|^2] > \frac{1}{s}\|\lambda\|_1^2 + \sum_{i \neq j} \lambda_i \lambda_j \langle a_i, a_j \rangle$$

   and is minimized by p *with* $p_j = 1$ *where* $j = \arg\max_{i=1,\ldots,n} |\lambda_i|$.

This theorem implies that Algorithm 1 produces a vector $p \in \mathbb{R}^n$ solving equation 3.3. Note that due to the sorting requirement in the input, the algorithm requires $O(n \log n)$ operations. As we discuss in Appendix 3.8, we could instead do this in $O(sn)$ operations by, instead of sorting and iterating through the values in order, simply selecting the next unvisited index i maximizing $|\lambda_i|$ and performing the same test/updates. As we show in Appendix 3.8, we need to select at most s indices before the if statement in Algorithm 1 holds. Whether to sort or do selection depends on the size of s relative to $\log n$.

3.3 Relation to QSGD and TernGrad

In this section, we will discuss how ATOMO is related to two recent quantization schemes, 1-bit QSGD (Alistarh et al., 2017) and TernGrad (Wen et al., 2017). We will show that in certain cases, these schemes are versions of the ATOMO for a specific sparsity budget s. Both schemes use the entry-wise atomic decomposition.

3.3.1 1-bit QSGD

QSGD takes as input $g \in \mathbb{R}^n$ and $b \geqslant 1$. This b governs the number of quantization buckets. When $b = 1$, this is referred to as 1-bit QSGD. 1-bit QSGD produces a

random vector $Q(g)$ defined by

$$Q(g)_i = \|g\|_2 \operatorname{sign}(g_i)\zeta_i.$$

Here, the $\zeta_i \sim \text{Bernoulli}(|g_i|/\|g\|_2)$ are independent random variables. A straightforward computation shows that $Q(g)$ can be defined equivalently by

$$Q(g)_i = \frac{g_i t_i}{|g_i|/\|g\|_2} \tag{3.6}$$

where $t_i \sim \text{Bernoulli}(|g_i|/\|g\|_2)$. Therefore, 1-bit QSGD exactly uses the atomic sparsification framework in equation 3.2 with $p_i = |g_i|/\|g\|_2$. The total sparsity budget is therefore given by

$$s = \sum_{i=1}^{n} p_i = \frac{\|g\|_1}{\|g\|_2}.$$

By Lemma 3.4 any g is s-balanced for this s. Therefore, Theorem 3.6 implies that the optimal way to assign p_i with this given s is $p_i = |g_i|/\|g\|_2$. Since this agrees with equation 3.6, this implies that 1-bit QSGD performs variance-optimal entry-wise sparsification for sparsity budget $s = \|g\|_1/\|g\|_2$.

3.3.2 TernGrad

Similarly, TernGrad takes as input $g \in \mathbb{R}^n$, and produces a sparsified version $T(g)$ given by

$$T(g)_i = \|g\|_\infty \operatorname{sign}(g_i)\zeta_i$$

where $\zeta_i \sim \text{Bernoulli}(|g_i|/\|g\|_\infty)$. A straightforward computation shows that $T(g)$ can be defined equivalently by

$$T(g)_i = \frac{g_i t_i}{|g_i|/\|g\|_\infty} \tag{3.7}$$

where $t_i \sim \text{Bernoulli}(|g_i|/\|g\|_\infty)$. Therefore, TernGrad exactly uses the atomic sparsification framework in equation 3.2 with $p_i = |g_i|/\|g\|_\infty$. The total sparsity budget is given by

$$s = \sum_{i=1}^{n} p_i = \frac{\|g\|_1}{\|g\|_\infty}.$$

By Lemma 3.4, any g is s-balanced for this s. Therefore, Theorem 3.6 implies that the optimal way to assign p_i with this given s is $p_i = |g_i|/\|g\|_\infty$. This agrees with equation 3.7. Therefore, TernGrad performs variance-optimal entry-wise sparsification for sparsity budget $s = \|g\|_1/\|g\|_\infty$.

3.3.3 ℓ_q-quantization

We can generalize both of these with the following quantization method, which we refer to as ℓ_q-*quantization*. Fix $q \in (0, \infty]$. Given $g \in \mathbb{R}^n$, we define the ℓ_q-quantization of g, denoted $L_q(g)$ by

$$L_q(v)_i = \|g\|_q \operatorname{sign}(g_i)\zeta_i$$

where $\zeta_i \sim \text{Bernoulli}(|g_i|/\|g\|_q)$. Note that for all i, $|g_i| \leqslant \|g\|_\infty \leqslant \|g\|_q$, so this does give us a valid probability. We can define $L_q(v)$ equivalently by

$$L_q(g)_i = \frac{g_i t_i}{|g_i|/\|g\|_q} \tag{3.8}$$

where $t_i \sim \text{Bernoulli}(|g_i|/\|g\|_q)$. Therefore, ℓ_q-quantization exactly uses the atomic sparsification framework in equation 3.2 with $p_i = |g_i|/\|g\|_q$. The total sparsity budget is therefore

$$s = \sum_{i=1}^{n} p_i = \frac{\|g\|_1}{\|g\|_q}.$$

By Lemma 3.4, the optimal way to assign p_i with this given s is $p_i = |g_i|/\|g\|_q$. Since this agrees with equation 3.8, Theorem 3.6 implies the following theorem.

Theorem 3.7. *ℓ_q-quantization performs atomic sparsification in the standard basis with $p_i = |g_i|/\|g\|_q$. This solves equation 3.3 for $s = \|g\|_1/\|g\|_q$ and satisfies $\mathbb{E}[\|L_q(g)\|_2^2] = \|g\|_1\|g\|_q$.*

In particular, for $q = 2$ we get 1-bit QSGD while for $q = \infty$, we get TernGrad. This shows strong connections between quantization and sparsification methods. Note that as q increases, the sparsity budget for ℓ_q-quantization increases while the variance decreases. The choice of whether to set $q = 2$, $q = \infty$, or q to something else is therefore dependent on the total expected sparsity one is willing to tolerate, and can be tuned for different distributed scenarios.

3.4 Spectral-Atomo: sparsifying the singular value decomposition

In this section we compare different methods for matrix sparsification. The first uses Atomo on the entry-wise decomposition of a matrix, and the second uses Atomo on the singular value decomposition (SVD) of a matrix. We refer to this second approach as Spectral-Atomo. We show that under concrete conditions, Spectral-Atomo incurs less variance than sparsifying entry-wise. We present these conditions and connect them to the equivalence of certain matrix norms.

Notation. For a rank r matrix X, denote its singular value decomposition by

$$X = \sum_{i=1}^{r} \sigma_i u_i v_i^T.$$

Let $\sigma = [\sigma_1, \ldots, \sigma_r]^T$. Taking the ℓ_p norm of σ gives a norm on X, referred to as the Schatten p-norm. For $p = 1$, we get the spectral norm $\|\cdot\|_*$, for $p = 2$ we get the Frobenius norm $\|\cdot\|_F$, and for $p = \infty$, we get the spectral norm $\|\cdot\|_2$. We define

the $\ell_{p,q}$ norm of a matrix X by

$$\|X\|_{p,q} = \left(\sum_{j=1}^{m} (\sum_{i=1}^{n} |X_{i,j}|^p)^{q/p} \right)^{1/q}.$$

When $p = q = \infty$, we define this to be $\|X\|_{max}$ where $\|X\|_{max} := \max_{i,j} |X_{i,j}|$.

Comparing matrix sparsification methods. Suppose that V is the vector space of real $n \times m$ matrices. Given $X \in V$, there are two standard atomic decompositions of X. The first is the entry-wise decomposition

$$X = \sum_{i,j} X_{i,j} e_i e_j^T.$$

The second is the singular value decomposition

$$X = \sum_{i=1}^{r} \sigma_i u_i v_i^T.$$

If r is small, it may be more efficient to communicate the $r(n+m)$ entries of the SVD, rather than the nm entries of the matrix. Let $\widehat{X}$ and $\widehat{X}_\sigma$ denote the random variables in equation 3.2 corresponding to the entry-wise decomposition and singular value decomposition of X, respectively. We wish to compare these two sparsifications.

In Table 3.1, we compare the communication cost and second moment of these two methods. The communication cost is the expected number of non-zero elements (real numbers) that need to be communicated. For $\widehat{X}$, a sparsity budget of s corresponds to s non-zero entries we need to communicate. For $\widehat{X}_\sigma$, a sparsity budget of s gives a communication cost of $s(n+m)$ due to the singular vectors. We compare the optimal second moment from Theorem 3.6.

To compare the second moment of these two methods under the same communication cost, we s and suppose X is s-balanced entry-wise. By Theorem 3.6 and

Table 3.1: Communication cost versus second moment of singular value sparsification and vectorized matrix sparsification of a $n \times m$ matrix.

	Communication Cost	Second Moment
Entry-wise	s	$\frac{1}{s}\|X\|_{1,1}^2$
SVD	$s(n+m)$	$\frac{1}{s}\|X\|_*^2$

Lemma 3.4, the second moment in Table 3.1 is achieved iff

$$s \leqslant \frac{\|X\|_{1,1}}{\|X\|_{\max}}.$$

To achieve the same communication cost with $\widehat{X}_\sigma$, we take a sparsity budget of $s' = s/(n+m)$. By Theorem 3.6 and Lemma 3.4, the second moment in Table 3.1 is achieved iff

$$s' = \frac{s}{n+m} \leqslant \frac{\|X\|_*}{\|X\|_2}.$$

This leads to the following theorem.

Theorem 3.8. *Suppose* $X \in \mathbb{R}^{n\times m}$ *and*

$$s \leqslant \min\left\{\frac{\|X\|_{1,1}}{\|X\|_{\max}}, (n+m)\frac{\|X\|_*}{\|X\|_2}\right\}.$$

Then $\widehat{X}_\sigma$ *with sparsity* $s' = s/(n+m)$ *incurs the same communication cost as* $\widehat{X}$ *with sparsity* s*, and* $\mathbb{E}[\|\widehat{X}_\sigma\|^2] \leqslant \mathbb{E}[\|\widehat{X}\|^2]$ *if and only if*

$$(n+m)\|X\|_*^2 \leqslant \|X\|_{1,1}^2.$$

To better understand this condition, we will make use of the following well-

known fact concerning the equivalence of the $\ell_{1,1}$ and spectral norms.

Lemma 3.9. *For any* $n \times m$ *matrix* X *over* $\mathbb{R}$, $\frac{1}{\sqrt{nm}}\|X\|_{1,1} \leqslant \|X\|_* \leqslant \|X\|_{1,1}$.

For expository purposes, we give a proof of this fact in Appendix 3.9 and show that these bounds are the best possible. In other words, there are matrices for which both of the inequalities are equalities. If the first inequality is tight, then $\mathbb{E}[\|\widehat{X}_\sigma\|^2] \leqslant \mathbb{E}[\|\widehat{X}\|^2]$, while if the second is tight then $\mathbb{E}[\|\widehat{X}_\sigma\|^2] \geqslant \mathbb{E}[\|\widehat{X}\|^2]$. As we show in the next section, using singular value sparsification can translate in to significantly reduced distributed training time.

3.5 Experiments

We present an empirical study of Spectral-Atomo and compare it to the recently proposed QSGD (Alistarh et al., 2017), and TernGrad (Wen et al., 2017), on a different neural network models and data sets, under real distributed environments. Our main findings are as follows:

- We observe that spectral-Atomo provides a useful alternative to entry-wise sparsification methods, it reduces communication compared to vanilla mini-batch SGD, and can reduce training time compared to QSGD and TernGrad by up to a factor of 2× and 3× respectively. For instance, on VGG11-BN trained on CIFAR-10, spectral-Atomo with sparsity budget 3 achieves 3.96× speedup over vanilla SGD, while 4-bit QSGD achieves 1.68× on a cluster of 16, g2.2xlarge instances. Both Atomo and QSGD greatly outperform TernGrad as well.
- We observe that spectral-Atomo in distributed settings leads to models with negligible accuracy loss when combined with parameter tuning.

Implementation and setup. We compare spectral-Atomo [1] with different sparsity budgets to b-bit QSGD across a distributed cluster with a parameter server (PS),

[1] code available at: `https://github.com/hwang595/ATOMO`

implemented in mpi4py (Dalcin et al., 2011a) and PyTorch (Paszke et al., 2017) and deployed on multiple types of instances in Amazon EC2 (*e.g.,* m5.4xlarge, m5.2xlarge, and g2.2xlarge), both PS and compute nodes are of the same type of instance. The PS implementation is standard, with a few important modifications. At the most basic level, it receives gradients from the compute nodes and broadcasts the updated model once a batch has been received.

In our experiments, we use data augmentation (random crops, and flips), and tuned the step-size for every different setup. Momentum and regularization terms are switched off to make the hyperparamter search tractable and the results more legible. Tuning the step sizes for this distributed network for three different datasets and eight different coding schemes can be computationally intensive. As such, we only used small networks so that multiple networks could fit into GPU memory. To emulate the effect of larger networks, we use synchronous message communication, instead of asynchronous.

Each compute node evaluates gradients sampled from its partition of data. Gradients are then sparsified through QSGD or spectral-Atomo, and then are sent back to the PS. Note that spectral-Atomo transmits the weighted singular vectors sampled from the true gradient of a layer. The PS then combines these, and updates the model with the average gradient. Our entire experimental pipeline is implemented in PyTorch (Paszke et al., 2017) with mpi4py (Dalcin et al., 2011a), and deployed on either g2.2xlarge, m5.2xlarge and m5.4xlarge instances in Amazon AWS EC2. We conducted our experiments on various models, datasets, learning tasks, and neural network models as detailed in Table 3.2.

Scalability. We study the scalability of these sparsification methods on clusters of different sizes. We used clusters with one PS and $n = 2, 4, 8, 16$ compute nodes. We ran ResNet-34 on CIFAR-10 using mini-batch SGD with batch size 512 split among compute nodes. The experiment was run on m5.4xlarge instances of AWS EC2 and the results are shown in Figure 3.1.

While increasing the size of the cluster, decreases the computational cost per worker, it causes the communication overhead to grow. We denote as computational

Table 3.2: The datasets used and their associated learning models and hyper-parameters.

Dataset	CIFAR-10	SVHN
# Data points	60,000	600,000
Model	ResNet-18 / VGG-11-BN	ResNet-18
# Classes	10	10
# Parameters	11,173k / 9,756k	11,173k

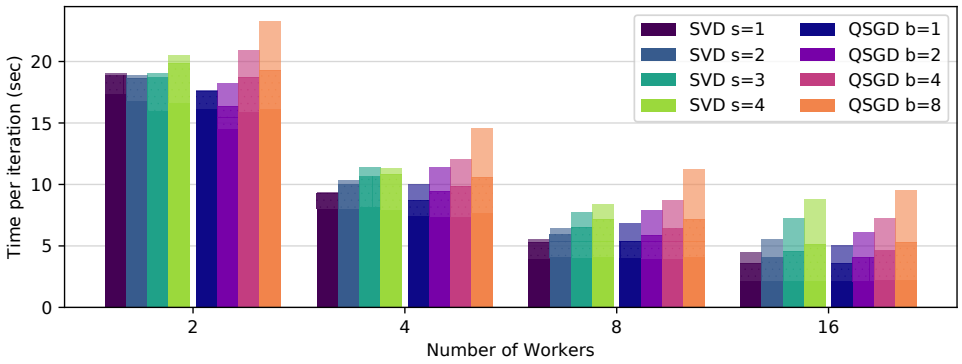


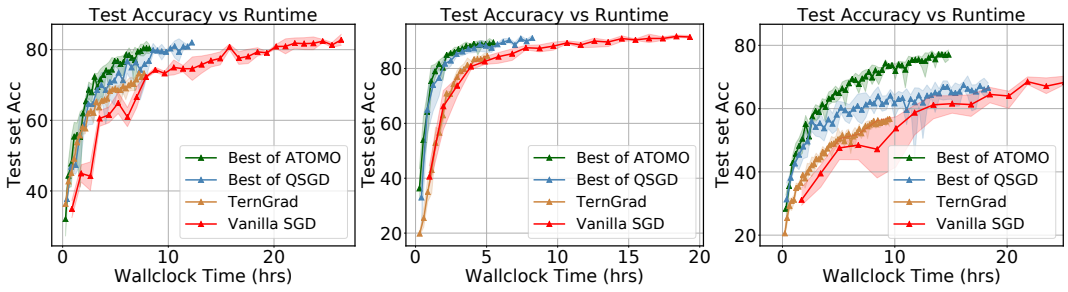
Figure 3.1: The timing of the gradient coding methods (QSGD and spectral-Atomo) for different quantization levels, b bits and s sparsity budget respectively for each worker when using a ResNet-34 model on CIFAR-10. For brevity, we use SVD to denote spectral-Atomo. The bars represent the total iteration time and are divided into computation time (bottom, solid), encoding time (middle, dotted) and communication time (top, faded).

cost, the time cost required by each worker for gradient computations, while the communication overhead is represented by the amount time the PS waits to receive the gradients by the slowest worker. This increase in communication cost is non-negligible, even for moderately-sized networks with sparsified gradients. We observed a trade-off in both sparsification approaches between the information retained in the messages after sparsification and the communication overhead.

End-to-end convergence performance. We evaluate the end-to-end convergence performance on different datasets and neural networks, training with spectral-

Atomo (with sparsity budget $s = 1, 2, 3, 4$), QSGD (with $n = 1, 2, 4, 8$ bits), and ordinary mini-batch SGD. The datasets and models are summarized in Table 6.1. We use ResNet-18 (He et al., 2016) and VGG11-BN (Simonyan and Zisserman, 2015) for CIFAR-10 (Krizhevsky et al., 2009) and SVHN (Netzer et al., 2011). Again, for each of these methods we tune the step size. The experiments were run on a cluster of 16 compute nodes instantiated on g2.2xlarge instances.

The gradients of convolutional layers are 4 dimensional tensors with shape of $[x, y, k, k]$ where x, y are two spatial dimensions and k is the size of the convolutional kernel. However, matrices are required to compute the SVD for spectral-Atomo, and we choose to reshape each layer into a matrix of size $[xy/2, 2k^2]$. This provides more flexibility on the sparsity budget for the SVD sparsification. For QSGD, we use the bucketing and Elias recursive coding methods proposed in (Alistarh et al., 2017), with bucket size equal to the number of parameters in each layer of the neural network.



(a) CIFAR-10, ResNet-18, Best of QSGD and SVD

(b) SVHN, ResNet-18, Best of QSGD and SVD

(c) CIFAR-10, VGG11, Best of QSGD and SVD

Figure 3.2: Convergence rates for the best performance of QSGD and spectral-Atomo, alongside TernGrad and vanilla SGD. (a) uses ResNet-18 on CIFAR-10, (b) uses ResNet-18 on SVHN, and (c) uses VGG-11-BN on CIFAR-10. For brevity, we use SVD to denote spectral-Atomo.

Figure 3.2 shows how the testing accuracy varies with wall clock time. Tables 3.3 and 3.4 give a detailed account of speedups of singular value sparsification compared to QSGD. In these tables, each method is run until a specified accuracy.

We observe that QSGD and Atomo speed up model training significantly and

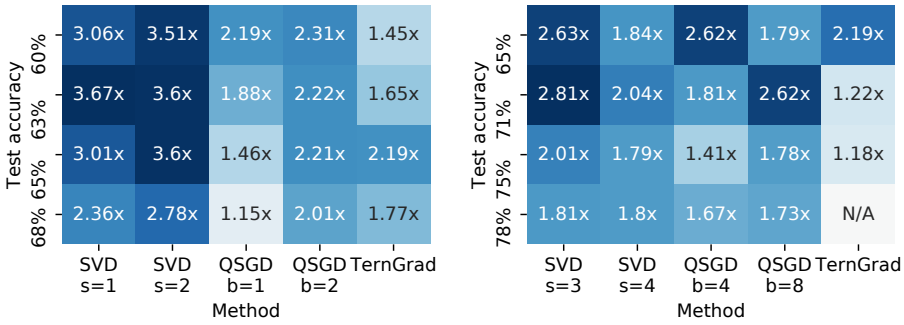


Table 3.3: Speedups of spectral-Atomo with sparsity budget s, b-bit QSGD, and TernGrad using ResNet-18 on CIFAR10 over vanilla SGD. N/A stands for the method fails to reach a certain Test accuracy in fixed iterations.

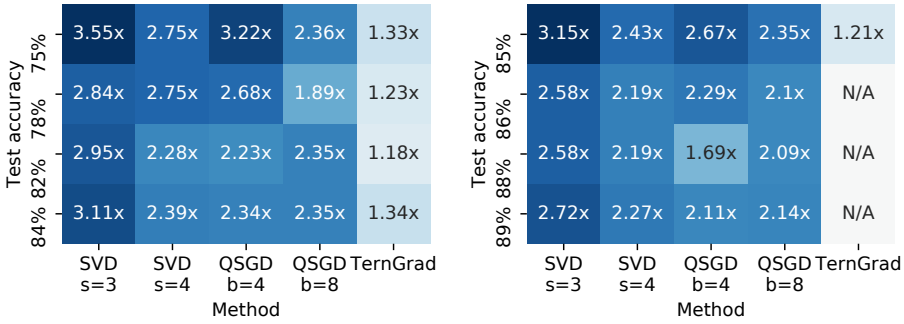


Table 3.4: Speedups of spectral-Atomo with sparsity budget s and b-bit QSGD, and TernGrad using ResNet-18 on SVNH over vanilla SGD. N/A stands for the method fails to reach a certain Test accuracy in fixed iterations.

achieve similar accuracy to vanilla mini-batch SGD. We also observe that the best performance is not achieve with the most sparsified, or quantized method, but the optimal method lies somewhere in the middle where enough information is preserved during the sparsification. For instance, 8-bit QSGD converges faster than 4-bit QSGD, and spectral-Atomo with sparsity budget 3, or 4 seems to be the fastest. Higher sparsity can lead to a faster running time, but extreme sparsification can adversely affect convergence. For example, for a fixed number of iterations, 1-bit QSGD has the smallest time cost, but may converge much more slowly to an

accurate model.

3.6 Conclusion

In this chapter, we present and analyze Atomo, a general sparsification method for distributed stochastic gradient based methods. Atomo applies to any atomic decomposition, including the entry-wise and the SVD of a matrix. Atomo generalizes 1-bit QSGD and TernGrad, and provably minimizes the variance of the sparsified gradient subject to a sparsity constraint on the atomic decomposition. We focus on the use Atomo for sparsifying matrices, especially the gradients in neural network training. We show that applying Atomo to the singular values of these matrices can lead to faster training than both vanilla SGD or QSGD, for the same communication budget. We present extensive experiments showing that Atomo can lead to up to a $2\times$ speed-up in training time over QSGD and up to $3\times$ speed-up in training time over TernGrad.

In the future, we plan to explore the use of Atomo with Fourier decompositions, due to its utility and prevalence in signal processing. More generally, we wish to investigate which atomic sets lead to reduced communication costs. We also plan to examine how we can sparsify and compress gradients in a joint fashion to further reduce communication costs. Finally, when sparsifying the SVD of a matrix, we only sparsify the singular values. We also note that it would be interesting to explore jointly sparsification of the SVD and and its singular vectors, which we leave for future work.

3.7 Proofs

3.7.1 Proof of Lemma 3.2

Proof. Suppose we have some p satisfying the conditions in equation 3.4. We define two auxiliary vectors $\alpha, \beta \in \mathbb{R}^n$ by

$$\alpha_i = \frac{|\lambda_i|}{\sqrt{p_i}}.$$
$$\beta_i = \sqrt{p_i}.$$

Then note that using the fact that $\sum_i p_i = s$, we have

$$\begin{aligned} f(p) &= \sum_{i=1}^{n} \frac{\lambda_i^2}{p_i} = \left(\sum_{i=1}^{n} \frac{\lambda_i^2}{p_i}\right)\left(\frac{1}{s}\sum_{i=1}^{n} p_i\right) = \frac{1}{s}\left(\sum_{i=1}^{n} \frac{\lambda_i^2}{p_i}\right)\left(\sum_{i=1}^{n} p_i\right) \\ &= \frac{1}{s}\|\alpha\|_2^2\|\beta\|_2^2. \end{aligned}$$

By the Cauchy-Schwarz inequality, this implies

$$f(p) = \frac{1}{s}\|\alpha\|_2^2\|\beta\|_2^2 \geqslant \frac{1}{s}|\langle\alpha, \beta\rangle|^2 = \frac{1}{s}\left(\sum_{i=1}^{n} |\lambda_i|\right)^2 = \frac{1}{s}\|\lambda\|_1^2. \tag{3.9}$$

This proves the first part of Lemma 3.2. In order to have $f(p) = \frac{1}{s}\|\lambda\|_1^2$, equation 3.9 implies that we need

$$|\langle\alpha, \beta\rangle| = \|\alpha\|_2\|\beta\|_2.$$

By the Cauchy-Schwarz inequality, this occurs iff α and β are linearly dependent. Therefore, $c\alpha = \beta$ for some constant c. Solving, this implies $p_i = c|\lambda_i|$. Since $\sum_{i=1}^{n} p_i = s$, we have

$$c\|\lambda\|_1 = \sum_{i=1}^{n} c|\lambda_i| = \sum_{i=1}^{n} p_i = s.$$

Therefore, $c = \|\lambda\|_1/s$, which implies the second part of the theorem. □

3.7.2 Proof of Lemma 3.5

Fix q that is feasible in equation 3.3. To prove Lemma 3.5 we will require a lemma. Given the atomic decomposition $g = \sum_{i=1}^{n} \lambda_i a_i$, we say that λ is s-unbalanced at i if $|\lambda_i|s > \|\lambda\|_1$, which is equivalent to g being unbalanced in this atomic decomposition at i. For notational simplicity, we will assume that λ is s-unbalanced at $i = 1$. Let $A \subseteq \{2, \ldots, n\}$. We define the following notation:

$$s_A = \sum_{i \in A} q_i.$$

$$f_A(q) = \sum_{i \in A} \frac{\lambda_i^2}{q_i}.$$

$$(\lambda_A)_i = \begin{cases} \lambda_i, & \text{for } i \in A, \\ 0, & \text{for } i \notin A. \end{cases}$$

Note that under this notation, Lemma 3.2 implies that for all $p > 0$,

$$f_A(p) \geqslant \frac{1}{s_A} \|\lambda_A\|_1^2. \tag{3.10}$$

Lemma 3.10. *Suppose that q is feasible and that there is some set $A \subseteq \{2, \ldots, n\}$ such that*

1. *λ_A is $(s_A + q_1 - 1)$-balanced.*
2. *$|\lambda_1|(s_A + q_1 - 1) > \|\lambda_A\|_1$.*

Then there is a vector p that is feasible satisfying $f(p) \leqslant f(q)$ and $p_1 = 1$.

Proof. Suppose that such a set A exists. Let $B = \{2, \ldots, n\} \backslash A$. Note that we have

$$f(q) = \sum_{i=1}^{n} \frac{\lambda_i^2}{q_i} = \frac{\lambda_1^2}{q_1} + f_A(q) + f_B(q).$$

By equation 3.10, this implies

$$f(q) \geqslant \frac{\lambda_1^2}{q_1} + \frac{1}{s_A}\|\lambda_A\|_1^2 + f_B(q). \tag{3.11}$$

Define p as follows.

$$p_i = \begin{cases} 1, & \text{for } i = 1, \\ \dfrac{|\lambda_i|(s_A + q_1 - 1)}{\|\lambda_A\|_1}, & \text{for } i \in A, \\ q_i, & \text{for } i \notin A. \end{cases}$$

Note that by Assumption 1 and Lemma 3.2, we have

$$f_A(p) = \frac{1}{s_A + q_1 - 1}\|\lambda_A\|_1^2.$$

Since $p_i = q_i$ for $i \in B$, we have $f_B(p) = f_B(q)$. Therefore,

$$f(p) = \lambda_1^2 + \frac{1}{s_A + q_1 - 1}\|\lambda_A\|_1^2 + f_B(q). \tag{3.12}$$

Combining equation 3.11 and equation 3.12, we have

$$\begin{aligned} f(q) - f(p) &= \lambda_1^2\left(\frac{1}{q_1} - 1\right) + \|\lambda_A\|_1^2\left(\frac{1}{s_A} - \frac{1}{s_A + q_1 - 1}\right) \\ &= \lambda_1^2\left(\frac{1 - q_1}{q_1}\right) + \|\lambda_A\|_1^2\left(\frac{q_1 - 1}{s_A(s_A + q_1 - 1)}\right). \end{aligned}$$

Combining this with Assumption 2, we have

$$f(q) - f(p) \geqslant \frac{\|\lambda_A\|_1^2}{(s_A + q_1 - 1)^2}\left(\frac{1 - q_1}{q_1}\right) + \|\lambda_A\|_1^2\left(\frac{q_1 - 1}{s_A(r_A + q_1 - 1)}\right). \tag{3.13}$$

To show that the RHS of equation 3.13 is at most 0, it suffices to show

$$s_A \geqslant q_1(s_A + q_1 - 1). \tag{3.14}$$

However, note that since $0 < q_1 < 1$, the RHS of equation 3.14 satisfies

$$q_1(s_A + q_1 - 1) = s_A q_1 - q_1(1 - q_1) \leqslant s_A q_1 \leqslant s_A.$$

Therefore, equation 3.14 holds, completing the proof. □

We can now prove Lemma 3.5. In the following, we will refer to Conditions 1 and 2, relative to some set A, as the conditions required by Lemma 3.10.

Proof. We first show this in the case that $n = 2$. Here we have the atomic decomposition

$$g = \lambda_1 a_1 + \lambda_2 a_2.$$

The condition that λ is s-unbalanced at $i = 1$ implies

$$|\lambda_1|(s - 1) > |\lambda_2|.$$

In particular, this implies $s > 1$. For $A = \{2\}$, Condition 1 is equivalent to

$$|\lambda_2|(s_A + q_1 - 2) \leqslant 0.$$

Note that $s_A = q_2$ and that $q_1 + q_2 - 2 = s - 2$ by assumption. Since $q_i \leqslant 1$, we know that $s - 2 \leqslant 0$ and so Condition 1 holds. Similarly, Condition 2 becomes

$$|\lambda_1|(s - 1) > |\lambda_2|$$

which holds by assumption. Therefore, Lemma 3.5 holds for $n = 2$.

Now suppose that $n > 2$, q is some feasible probability vector, and that λ is s-unbalanced at index 1. We wish to find an A satisfying Conditions 1 and 2. Consider $B = \{2, \ldots, n\}$. Note that for such B, $s_B + q_1 - 1 = s - 1$. By our unbalanced assumption, we know that Condition 2 holds for $B = \{2, \ldots, n\}$. If λ_B is $(s - 1)$-balanced, then Lemma 3.10 implies that we are done.

Assume that λ_B is not $(s-1)$-balanced. After relabeling, we can assume it is unbalanced at $i = 2$. Let $C = \{3, \ldots, n\}$. Therefore,

$$|\lambda_2|(s-2) > \|\lambda_C\|_1. \tag{3.15}$$

Combining this with the s-unbalanced assumption at $i = 1$, we find

$$\begin{aligned} |\lambda_1| &> \frac{\|\lambda_B\|_1}{s-1} \\ &= \frac{|\lambda_2|}{s-1} + \frac{\|\lambda_C\|_1}{s-1} \\ &> \frac{\|\lambda_C\|_1}{(s-1)(s-2)} + \frac{\|\lambda_C\|_1}{s-1} \\ &= \frac{\|\lambda_C\|_1}{s-2}. \end{aligned}$$

Therefore,

$$|\lambda_1|(s-q_2-1) \geqslant |\lambda_1|(s-2) > \|\lambda_C\|_1. \tag{3.16}$$

Let $D = \{1, 3, 4, \ldots, n\} = \{1, \ldots, n\} \backslash \{2\}$. Then note that equation 3.16 implies that λ_D is $(s-q_2)$-unbalanced at $i = 1$. Inductively applying this theorem, this means that we can find a vector $p' \in \mathbb{R}^{|D|}$ such that $p'_1 = 1$ and $f_D(p') \leqslant f_D(q)$. Moreover, $s_D(p') = s - q_2$. Therefore, if we let p be the vector that equals p' on D and with $p_2 = q_2$, we have

$$f(p_2) = f_C(p') + \frac{\lambda_2^2}{q_2} \leqslant f_D(q) + \frac{\lambda_2^2}{q_2} = f(q).$$

This proves the desired result. □

3.8 Analysis of Atomo via the KKT condtions

In this section we show how to derive Algorithm 1 using the KKT conditions. Recall that we wish to solve the following optimization problem:

$$\min_{p} \; f(p) := \sum_{i=1}^{n} \frac{\lambda_i^2}{p_i} \quad \text{subject to } \forall i,\; 0 < p_i \leqslant 1, \quad \sum_{i=1}^{n} p_i = s. \tag{3.17}$$

We first note a few immediate consequences.

1. If $s > n$ then the problem is infeasible. Note that when $s \geqslant n$, the optimal thing to do is to set all $p_i = 1$, in which case no sparsification takes place.

2. If $\lambda_i = 0$, then $p_i = 0$. This follows from the fact that this p_i does not change the value of $f(p)$, and the objective could be decreased by allocating more to the p_j associated to non-zero λ_j. Therefore we can assume that all $\lambda_i \neq 0$.

3. If $|\lambda_i| \geqslant |\lambda_j| > 0$, then we can assume $p_i \geqslant p_j$. Otherwise, suppose $p_j > p_i$ but $|\lambda_i| \geqslant |\lambda_j|$. Let p' denote the vector with p_i, p_j switched. We then have

$$\begin{aligned} f(p) - f(p') &= \frac{\lambda_i^2 - \lambda_j^2}{p_i} + \frac{\lambda_j^2 - \lambda_i^2}{p_j} \\ &= \lambda_i^2 \left(\frac{1}{p_i} - \frac{1}{p_j} \right) - \lambda_j^2 \left(\frac{1}{p_i} - \frac{1}{p_j} \right) \\ &\geqslant 0. \end{aligned}$$

We therefore assume $0 < s \leqslant n$ and $|\lambda_1| \geqslant |\lambda_2| \geqslant \ldots \geqslant |\lambda_n| > 0$. As above we define $\lambda := [\lambda_1, \ldots, \lambda_n]^T$. While the formulation of equation 3.17 does not allow direct application of the KKT conditions, since we have a strict inequality of $0 < p_i$, this is fixed with the following lemma.

Lemma 3.11. *The minimum of equation 3.17 is achieved by some p^* satisfying*

$$p_i^* \geqslant \frac{s\lambda_i^2}{n\|\lambda\|_2^2}.$$

Proof. Define $\overline{p}$ by $\overline{p}_i = s/n$. This vector is clearly feasible in equation 3.17. Let p be any feasible vector. If $f(p) \leqslant f(q)$ then for any $i \in [n]$ we have

$$\frac{\lambda_i^2}{p_i} \leqslant f(p) \leqslant f(\overline{p}).$$

Therefore, $p_i \geqslant \lambda_i^2/f(\overline{p})$. A straightforward computations shows that $f(\overline{p}) = n\|\lambda\|_2^2/s$. Note that this implies that we can restrict to the feasbile set

$$\frac{s\lambda_i^2}{n\|\lambda\|_2^2} \leqslant p_i \leqslant 1.$$

This defines a compact region C. Since f is continuous on this set, its maximum value is obtained at some p^*. □

The KKT conditions then imply that at any point p solving equation 3.17, we must have

$$0 \leqslant 1 - p_i \perp \mu - \frac{\lambda_i^2}{p_i} \geqslant 0, \;\; i = 1, 2, \ldots, n \tag{3.18}$$

for some $\mu \in \mathbb{R}$. Since $|\lambda_i| > 0$ for all i, we actually must have $\mu > 0$. We therefore have two conditions for all i.

1. $p_i = 1 \implies \mu \geqslant \lambda_i^2$.
2. $p_i < 1 \implies p_i = |\lambda_i|/\sqrt{\mu}$.

Note that in either case, to have p_1 feasible we must have $\mu \geqslant \lambda_1^2$. Combining this with the fact that we can always select $p_1 \geqslant p_2 \geqslant \ldots \geqslant p_n$, we obtain the following partial characterization of the solution to equation 3.17. For some $n_s \in [n]$, we have $p_1, \ldots, p_{n_s} = 1$ while $p_i = |\lambda_i|/\sqrt{\mu} \in (0, 1)$ for $i = n_s + 1, \ldots, n$. Combining this with the constraint that $\sum_{i=1}^n p_i = s$, we have

$$s = \sum_{i=1}^{n} p_i = n_s + \sum_{i=n_s+1}^{n} p_i = n_s + \sum_{i=n_s+1}^{n} \frac{|\lambda_i|}{\sqrt{\mu}}. \tag{3.19}$$

Rearranging, we obtain

$$\mu = \frac{\left(\sum_{i=n_s+1}^{n} |\lambda_i|\right)^2}{(s-n_s)^2} \tag{3.20}$$

which then implies that

$$p_i = 1, \;\; i = 1, \ldots, n_s, \quad p_i = \frac{|\lambda_i|(s-n_s)}{\sum_{j=n_s+1}^{n} |\lambda_j|}, \;\; i = n_s+1, \ldots, n. \tag{3.21}$$

Thus, we need to select n_s such that the p_i in equation 3.21 are bounded above by 1. Let n_s^* denote the first element of $[n]$ for which this holds. Then the condition that $p_i \leqslant 1$ for $i = n_s^*+1, \ldots, n$ is exactly the condition that $[\lambda_{n_s^*+1}, \ldots, \lambda_n]$ is $(s-n_s)$-balanced (see Definition 3.3. In particular, Lemma 3.2 implies that, fixing $p_i = 1$ for $i = 1, \ldots, n_s^*$, the optimal way to assign the remaining p_i is by

$$p_i = \frac{|\lambda_i|(s-n_s^*)}{\sum_{j=n_s^*+1}^{n} |\lambda_j|}.$$

This agrees with equation 3.21 for $n_s = n_s^*$. In particular, the minimal value of f occurs at the first value of n_s such that the p_i in equation 3.21 are bounded above by 1.

Algorithm 1 scans through the sorted λ_i and finds the first value of n_s for which the probabilities in equation 3.21 are in $[0, 1]$, and therefore finds the optimal p for equation 3.17. The runtime is dominated by the $O(n \log n)$ sorting cost. It is worth noting that we could perform the algorithm in $O(sn)$ time as well. Instead of sorting and then iterating through the λ_i in order, at each step we could simply select the next largest $|\lambda_i|$ not yet seen and perform an analogous test and update as in the above algorithm. Since we would have to do the selection step at most s times, this leads to an $O(sn)$ complexity algorithm.

3.9 Equivalence of norms

We are often interested in comparing norms on vectors spaces. This naturally leads to the following definition.

Definition 3.12. *Let V be a vector space over $\mathbb{R}$ or $\mathbb{C}$. Two norms $\|\cdot\|_a, \|\cdot\|_b$ are equivalent if there are positive constants C_1, C_2 such that*

$$C_1\|x\|_a \leqslant \|x\|_b \leqslant C_2\|x\|_a$$

for all $x \in V$.

As it turns out, norms on finite-dimensional vector spaces are always equivalent.

Theorem 3.13. *Let V be a finite-dimensional vector space over $\mathbb{R}$ or $\mathbb{C}$. Then all norms are equivalent.*

In order to compare norms, we often wish to determine the tightest constants which give equivalence between them. In Section 3.4, we are particularly interested in comparing the $\|X\|_*$ and $\|X\|_{1,1}$ on the space of $n \times m$ matrices. We have the following lemma.

Lemma 3.14. *For all $n \times m$ real matrices,*

$$\frac{1}{\sqrt{nm}}\|X\|_{1,1} \leqslant \|X\|_* \leqslant \|X\|_{1,1}.$$

Proof. Suppose that X has the singular value decomposition

$$X = \sum_{i=1}^{r} \sigma_i u_i v_i^T.$$

We will first show the left inequality. First, note that for any $n \times m$ matrix A, $\|A\|_{1,1} \leqslant \sqrt{nm}\|A\|_F$. This follows directly from the fact that for a n-dimensional vector v, $\|v\|_1 \leqslant \sqrt{n}\|v\|_2$. We will also use the fact that for any vectors $u \in \mathbb{R}^n, v \in$

$\mathbb{R}^m$, $\|uv^T\|_F = \|u\|_2\|v\|_2$. We then have

$$\begin{aligned}
\|X\|_{1,1} &= \left\| \sum_{i=1}^{r} \sigma_i u_i v_i^T \right\|_{1,1} \\
&\leqslant \sum_{i=1}^{r} \sigma_i \|u_i v_i^T\|_{1,1} \\
&= \sum_{i=1}^{r} \sigma_i \sqrt{nm} \|u_i v_i^T\|_F \\
&= \sum_{i=1}^{r} \sigma_i \sqrt{nm} \|u_i\|_2 \|v_i\|_2 \\
&= \|X\|_*.
\end{aligned}$$

For the right inequality, note that we have

$$X = \sum_{i,j} X_{i,j} e_i e_j^T$$

where $e_i \in \mathbb{R}^n$ is the i-th standard basis vector, while $e_j \in \mathbb{R}^m$ is the j-th standard basis vector. We then have

$$\|X\|_* \leqslant \sum_{i,j} |X_{i,j}| \|e_i e_j^T\|_* = \sum_{i,j} |X_{i,j}| = \|X\|_{1,1}.$$

□

In fact, these are the best constants possible. To see this, first consider the matrix X with a 1 in the upper-left entry and 0 elsewhere. Clearly, $\|X\|_* = \|X\|_{1,1} = 1$, so the right-hand inequality is tight. For the left-hand inequality, consider the all-ones matrix X. This has one singular value, $\sqrt{nm}$, so $\|X\|_* = \sqrt{nm}$. On the other hand, $\|X\|_{1,1} = nm$. Therefore, $\|X\|_{1,1} = \sqrt{nm}\|X\|_*$ in this case.

4 PUFFERFISH: COMMUNICATION-EFFICIENT MODELS AT NO EXTRA COST

To mitigate communication overheads in distributed model training, several studies propose the use of compressed stochastic gradients, usually achieved by sparsification or quantization. Such techniques achieve high compression ratios, but in many cases incur either significant computational overheads or some accuracy loss. In this chapter, we present PUFFERFISH, a communication and computation efficient distributed training framework that incorporates the gradient compression into the model training process via training low-rank, pre-factorized deep networks. PUFFERFISH not only reduces communication, but also completely bypasses any computation overheads related to compression, and achieves the same accuracy as SotA, off-the-shelf deep models. PUFFERFISH can be directly integrated into current deep learning frameworks with minimum implementation modification. Our extensive experiments over real distributed setups, across a variety of large-scale machine learning tasks, indicate that PUFFERFISH achieves up to 1.64× end-to-end speedup over the latest distributed training API in PyTorch without accuracy loss. Compared to the *Lottery Ticket Hypothesis* models, PUFFERFISH leads to equally accurate, small-parameter models while avoiding the burden of "winning the lottery". PUFFERFISH also leads to more accurate and smaller models than SotA structured model pruning methods.

4.1 PUFFERFISH: effective deep factorized network training

In the following subsections, we discuss how model factorization is implemented for different model architectures.

4.1.1 Low-rank factorization for FC layers

For simplicity, we discuss a 2-layer FC network that can be represented as $h(x) = \sigma(W_1\sigma(W_2x))$ where $W_l, \forall l \in \{1, 2\}$ are weight matrices, $\sigma(\cdot)$ is an arbitrary activation function, and x is the input data point. We propose to pre-factorize the matrices W_l into $U_l V_l^T$ where the factors are of significantly smaller dimensions while also reducing the computational complexity of the full-rank FC layer.

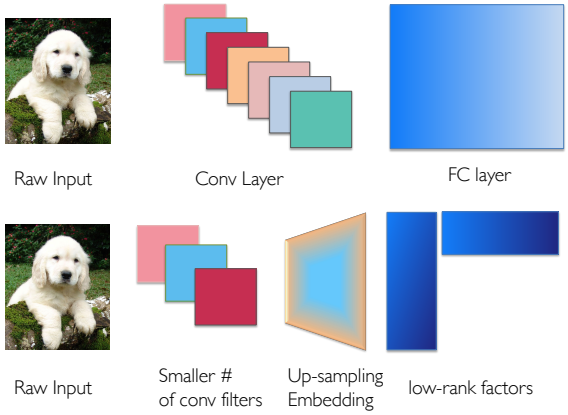


Figure 4.1: We propose to replace fully connected layers represented by a matrix W, by a set of trainable factors UV^T, and represent each of the N convolutional filters of each conv layer as a linear combination of $\frac{N}{R}$ filters. This latter operation can be achieved by using fewer filters per layer, and then applying a trainable up-sampling embedding to the output channels.

4.1.2 Low-rank factorization for convolution layers

Basics on convolution layers. The above low-rank factorization strategy extends to convolutional layers (see Fig. 4.1 for a sketch). In a convolution layer, a c_{in}-channel input image of size $H \times W$ pixels is convolved with c_{out} filters of size $c_{in} \times k \times k$ to create a c_{out}-channel output feature map. Therefore, the computational complexity for the convolution of the filter with a c_{in}-channel input image is $\mathcal{O}(c_{in}c_{out}k^2HW)$. In what follows, we describe schemes for modifying the architecture of the convolution layers via low-rank factorization to reduce computational complexity and

the number of parameters. The idea is to replace vanilla (full-rank) convolution layers with factorized versions. These factorized filters amount to the same number of convolution filters, but are constructed through linear combinations of a sparse, *i.e.,* low-rank filter basis.

Factorizing a convolution layer. For a convolution layer with dimension $W \in \mathbb{R}^{c_{in} \times c_{out} \times k \times k}$ where c_{in} and c_{out} are the number of input and output channels and k is the size of the convolution filters, *e.g.,* $k = 3$ or 5. Instead of factorizing the 4D weight of a convolution layer directly, we consider factorizing the unrolled 2D matrix. Unrolling the 4D tensor W leads to a 2D matrix with shape $W_{\text{unrolled}} \in \mathbb{R}^{c_{in}k^2 \times c_{out}}$ where each column represents the weight of a vectorized convolution filter. The rank of the unrolled matrix is determined by $\min\{c_{in}k^2, c_{out}\}$. Factorizing the unrolled matrix returns $U \in \mathbb{R}^{c_{in}k^2 \times r}$, $V^\top \in \mathbb{R}^{r \times c_{out}}$, *i.e.,* $W_{\text{unrolled}} \approx UV^\top$. Reshaping the factorized $U, V^\top$ matrices back to 4D filters leads to $U \in \mathbb{R}^{c_{in} \times r \times k \times k}$, $V^\top \in \mathbb{R}^{r \times c_{out}}$. Therefore, factorizing a convolution layer returns a thinner convolution layer U with width r, *i.e.,* the number of convolution filters, and a linear projection layer $V^\top$. In other words, the full-rank original convolution filter bank is approximated by a linear combination of r basis filters. The $V^\top$s can also be represented by a 1×1 convolution layer, *e.g.,* $V_l^\top \in \mathbb{R}^{r \times c_{out} \times 1 \times 1}$, which is more natural for computer vision tasks as it operates directly on the spatial domain (Lin et al., 2013). In PUFFERFISH, we use the 1×1 convolution for all $V_l^\top$ layers in the considered CNNs. One can also use tensor decomposition, *e.g.,* the Tucker decomposition to directly factorize the 4D tensor weights (Tucker, 1966). In this dissertation, for simplicity, we do not consider tensor decompositions.

4.1.3 Low-rank factorization for LSTM layers

LSTMs have been proposed as a means to mitigate the "vanishing gradient" issue of traditional RNNs (Hochreiter and Schmidhuber, 1997). The forward pass of an

LSTM is as follows

$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t).
\end{aligned} \tag{4.1}
$$

h_t, c_t, x_t represent the hidden state, cell state, and input at time t respectively. h_{t-1} is the hidden state of the layer at time $t-1$. i_t, f_t, g_t, o_t are the input, forget, cell, and output gates, respectively. $\sigma(\cdot)$ and $\odot$ denote the sigmoid activation function and the Hadamard product, respectively. The trainable weights are the matrices $W_{i\cdot} \in \mathbb{R}^{h\times d}, W_{h\cdot} \in \mathbb{R}^{h\times h}$, where d and h are the embedding and hidden dimensions. Thus, similarly to the low-rank FC layer factorization, the factorized LSTM layer is represented by

$$
\begin{aligned}
i_t &= \sigma(U_{ii}V_{ii}^\top x_t + b_{ii} + U_{hi}V_{hi}^\top h_{t-1} + b_{hi}) \\
f_t &= \sigma(U_{if}V_{if}^\top x_t + b_{if} + U_{hf}V_{hf}^\top h_{t-1} + b_{hf}) \\
g_t &= \tanh(U_{ig}V_{ig}^\top x_t + b_{ig} + U_{hg}V_{hg}^\top h_{t-1} + b_{hg}) \\
o_t &= \sigma(U_{io}V_{io}^\top x_t + b_{io} + U_{ho}V_{ho}^\top h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t).
\end{aligned} \tag{4.2}
$$

4.1.4 Low-rank network factorization for Transformer

A Transformer layer consists of a stack of encoders and decoders (Vaswani et al., 2017). Both encoder and decoder contain three main building blocks, *i.e.,* the *multi-head attention* layer, *position-wise feed-forward networks* (FFN), and *positional encoding*. A p-head attention layer learns p independent attention mechanisms on the input

key (K), value (V), and queries (Q) of each input token:

$$\begin{aligned}\text{MultiHead}(Q, K, V) &= \text{Concat}(\text{head}_1, \cdots, \text{head}_p)W^O \\ \text{where head}_i &= \text{Attention}(QW_i^Q, KW_i^K, VW_i^V).\end{aligned}$$

In the above, $W_i^Q, W_i^K, W_i^V, i \in \{1, \cdots, p\}$ are trainable weight matrices. The particular attention, referred to as "scaled dot-product attention", is used in Transformers, *i.e.,* $\text{Attention}(\tilde{Q}, \tilde{K}, \tilde{V}) = \text{softmax}\left(\frac{\tilde{Q}\tilde{K}^\top}{\sqrt{d}}\right)\tilde{V}$ where $\tilde{Q} = QW_i^Q, \tilde{K} = KW_i^K, \tilde{V} = VW_i^V$. W^O projects the output of the multi-head attention layer to match the embedding dimension. Following (Vaswani et al., 2017), we assume the projected key, value, and query are embedded to pd dimensions, and are projected to d dimensions in the attention layer. In Transformer, a sequence of N input tokens are usually batched before passing to the model where each input token is embedded to a pd dimensional vector. Thus, dimensions of the inputs are $Q, K, V \in \mathbb{R}^{N \times pd}$. The learnable weight matrices are $W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{pd \times d}, W^O \in \mathbb{R}^{pd \times pd}$. The FFN in Transformer consists of two learnable FC layers: $\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$ where $W_1 \in \mathbb{R}^{pd \times 4pd}, W_2 \in \mathbb{R}^{4pd \times pd}$ (the relationships between the notations in this dissertation and the original Transformer paper (Vaswani et al., 2017) are $pd = d_{\text{model}}$, $d = d_k = d_v$, and $d_{ff} = 4pd$).

In PUFFERFISH, we factorize all learnable weight matrices in the multi-head attention and the FFN layers. We leave the positional encoding as is, since there are no trainable weights. For the bias term of each layer and the "*Layer Normalization*" weights, we use the vanilla weights directly, as they are represented by vectors.

4.1.5 Computational complexity and model size

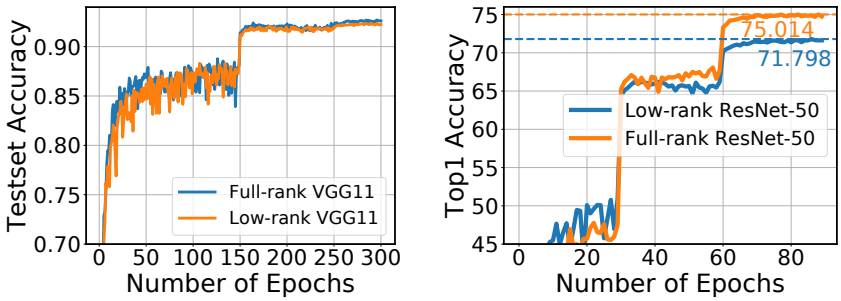
A low-rank factorized network enjoys a smaller number of parameters and lower computational complexity. Thus, both the computation and communication efficiencies are improved, as the amount of communication is proportional to the number of parameters. We summarize the computational complexity and the number of parameters in the vanilla and low-rank FC, convolution, LSTM, and the

Table 4.1: The number of parameters and computational complexities for full-rank and low-rank FC, convolution, LSTM, and the Transformer layers where m, n are the dimensions of the FC layer and c_{in}, c_{out}, k are the input, output dimensions, and kernel size respectively. h, d denote the hidden and embedding dimensions in the LSTM layer. N, p, d denote the sequence length, number of heads, and embedding dimensions in the Transformer. r denotes the rank of the factorized low-rank layer we assume to use.

Networks	# Params.	Computational Complexity
Vanilla FC	$m \times n$	$\mathcal{O}(mn)$
Factorized FC	$r(m+n)$	$\mathcal{O}(r(m+n))$
Vanilla Conv.	$c_{in} \times c_{out} \times k^2$	$\mathcal{O}(c_{in}c_{out}k^2HW)$
Factorized Conv.	$c_{in}rk^2 + rc_{out}$	$\mathcal{O}(rc_{in}k^2HW + rHWc_{out})$
Vanilla LSTM	$4(dh + h^2)$	$\mathcal{O}(dh + h^2)$
Factorized LSTM	$4dr + 12hr$	$\mathcal{O}(dr + hr)$
Vanilla Attention	$4p^2d^2$	$\mathcal{O}(Np^2d^2 + N^2d)$
Factorized Attention	$(3p+5)prd$	$\mathcal{O}(rpdN + N^2d)$
Vanilla FFN	$8p^2d^2$	$\mathcal{O}(p^2d^2N)$
Factorized FFN	$10pdr$	$\mathcal{O}(rpdN)$

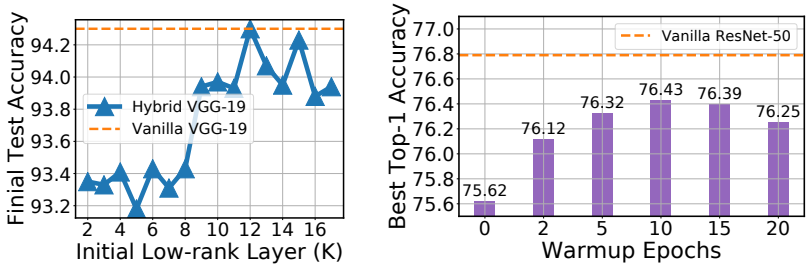
Transformer layers in Table 4.1. We assume the FC layer has shape $W_{FC} \in \mathbb{R}^{m \times n}$, the convolution layer has shape $W_{Conv} \in \mathbb{R}^{c_{in} \times c_{out} \times k \times k}$, the LSTM layer has shape $W_i \in \mathbb{R}^{4h \times d}; W_h \in \mathbb{R}^{4h \times h}$ (where W_i and W_h is the concatenated input-hidden and hidden-hidden weight matrices), and the shapes of the model weights in the encoder of a Transformer follow the discussion in Section 4.1.4. For the number of parameters, we do not count the bias terms. For Transformers, we show the computational complexity of a single encoder block. We assume the low-rank layers have rank r. As the computation across the p heads can be done in parallel, we report the computational complexity of a single attention head. Note that for the LSTM layer, our complexity analysis assumes the low-rank layer uses the same rank for the input-hidden weights $W_{i\cdot}$ and the hidden-hidden weights $W_{h\cdot}$. Similarly, for the Transformer layer, we assume the low-rank layer uses the same rank r for all W_i^Q, W_i^K, W_i^V, W^O. Further details can be found in section 4.5.

4.2 Strategies for mitigating accuracy loss



(a) VGG-11 on CIFAR-10 (b) ResNet-50 on ImageNet

Figure 4.2: Model convergence comparisons between vanilla models and PUFFERFISH factorized models: (a) low-rank VGG-11 over the CIFAR-10 dataset; (b) ResNet-50 over the ImageNet dataset. For the low-rank networks, all layers except for the first convolution and the very last FC layer are factorized with a fixed rank ratio at 0.25.



(a) Hybrid network (b) Vanilla warm-up training

Figure 4.3: The effect of the test accuracy loss mitigation methods in PUFFERFISH: (a) **Hybrid network**: The final test accuracy of the hybrid VGG-19 architectures with various initial low-rank layer indices (K) over the CIFAR-10 dataset. (b) **Vanilla warm-up training**: The final top-1 accuracy of the hybrid-ResNet-50 architecture trained on the ImageNet dataset under the different number of vanilla warm-up epochs: $\{2, 5, 10, 15, 20\}$.

In this section, we showcase that training low-rank models from scratch leads to an accuracy loss. Interestingly, this loss can be mitigated by balancing the degree of

factorization across layers, and by using a short full-rank warm-up training phase used to initialize the factorized model.

We conduct an experimental study on a version of PUFFERFISH where every layer of the network is factorized except for the first convolution layer and the last FC layer. On a relatively small task, *e.g.,* VGG-11 on CIFAR-10, we observe that PUFFERFISH only leads to $\sim 0.4\%$ accuracy loss (as shown in Figure 4.2) compared to the vanilla VGG-19-BN. However, for ResNet-50 on the ImageNet dataset, a $\sim 3\%$ top-1 accuracy loss of PUFFERFISH is observed. To mitigate the accuracy loss of the factorized networks over the large-scale ML tasks, we propose two methods, *i.e.,* (i) *hybrid network architecture* and (ii) *vanilla warm-up training*. We then discuss each method separately.

Hybrid network architecture. In PUFFERFISH, the low-rank factorization aims at approximating the original network weights, *i.e.,* $W_l \approx U_l V_l^\top$ for layer l, which inevitably introduces approximation error. Since the approximation error in the early layers can be accumulated and propagated to the later layers, a natural strategy to mitigate the model accuracy loss is to only factorize the later layers. Moreover, for most of CNNs, the number of parameters in later layers dominates the entire network size. Thus, factorizing the later layers does not sacrifice the degree of model compression we can achieve. Specifically, for an L layer network $\{W_1, W_2, \cdots, W_L\}$, factorizing every layer leads to $\{U_1, V_1^\top, U_2, V_2^\top, \cdots, U_L, V_L^\top\}$. In the hybrid network architecture, the first $K-1$ layers are not factorized, *i.e.,* $\{W_1, W_2, \cdots, W_{K-1}, U_K, V_K^\top, \cdots, U_L, V_L^\top\}$ where we define K as the index of the first low-rank layer in a hybrid architecture. We treat K as a hyper-parameter, which balances the model compression ratio and the final model accuracy. In our experiments, we tune K for all models. The effectiveness of the hybrid network architecture is shown in Figure 4.3a, from which we observe that the hybrid VGG-19 with $K = 9$ mitigates $\sim 0.6\%$ test accuracy loss.

Vanilla warm-up training. It has been widely observed that epochs early in training are critical for the final model accuracy (Keskar et al., 2016; Achille et al., 2018;

Algorithm 2 PUFFERFISH training procedure

Input : Randomly initialized weights of vanilla N-layer architectures $\{W_1, W_2, \ldots, W_L\}$, and the associated weights of hybrid N-layer architecture $\{W_1, W_2, \ldots, W_{K-1}, U_K, V_K^\top, \ldots, U_L, V_L^\top\}$, the entire training epochs E, the vanilla warm-up training epochs E_{wu}, and learning rate schedule $\{\eta_t\}_{t=1}^{E}$

Output: Trained hybrid L-layer architecture weights $\{\hat{W}_1, \hat{W}_2, \ldots, \hat{W}_{K-1}, \hat{U}_K, \hat{V}_K^\top, \ldots, \hat{U}_L, \hat{V}_L^\top\}$

for $t \in \{1, \ldots, E_{wu}\}$ **do**
  Train $\{W_1, W_2, \ldots, W_L\}$ with learning rate schedule $\{\eta_t\}_{t=1}^{E_{wu}}$; `// vanilla warm-up training`
end
for $l \in \{1, \ldots, L\}$ **do**
  if $l < K$ **then**
    copy the partially trained W_l weight to the hybrid network;
  else
    $\tilde{U}_l \Sigma_l \tilde{V}_l^\top = \text{SVD}(W_l)$; `// Decomposing the vanilla warm-up trained weights`
    $U_l = \tilde{U}_l \Sigma_l^{\frac{1}{2}}, V_l^\top = \Sigma^{\frac{1}{2}} \tilde{V}_l^\top$
  end
end
for $t \in \{E_{wu} + 1, \ldots, E\}$ **do**
  Train the hybrid network weights, *i.e.*, $\{W_1, W_2, \ldots, W_{K-1}, U_K, V_K^\top, \ldots, U_L, V_L^\top\}$ with learning rate schedule $\{\eta_t\}_{t=E_{wu}}^{E}$; `// consecutive low rank training`
end

Leclerc and Madry, 2020; Agarwal et al., 2020; Jastrzebski et al., 2020). For instance, sparsifying gradients in early training phases can hurt the final model accuracy (Lin et al., 2017). Similarly, factorizing the vanilla model weights in the very beginning of the training procedure can also lead to accuracy loss, which may be impossible to mitigate in later training epochs. It has also been shown that good initialization strategies play a significant role in the final model accuracy (Zhou et al., 2020).

In this work, to mitigate the accuracy loss, we propose to use the partially trained vanilla, full-rank model weights to initialize the low-rank factorized network. We refer to this as "*vanilla warm-up training*". We train the vanilla model for a few epochs (E_{wu}) first. Then, we conduct truncated matrix factorization (via truncated SVD) over the partially trained model weights to initialize the low-rank factors. For instance, given a partially trained FC layer $W^{(l)}$, we deploy SVD on it such that we get $\tilde{U}\Sigma\tilde{V}^\top$. After that the U and $V^\top$ weights we introduced in the previous sections can be found by $U = \tilde{U}\Sigma^{\frac{1}{2}}, V^\top = \Sigma^{\frac{1}{2}}\tilde{V}^\top$. For convolution layer $W \in \mathbb{R}^{c_{in} \times c_{out} \times k \times k}$, we conduct SVD over the unrolled 2D matrix $W_{\text{unrolled}} \in \mathbb{R}^{c_{in} k^2 \times c_{out}}$, which leads to $U \in \mathbb{R}^{c_{in} k^2 \times r}, V^\top \in \mathbb{R}^{r \times c_{out}}$ where reshaping U, V back to 4D leads to the desired initial weights for the low-rank layer, *i.e.*, $U \in \mathbb{R}^{r \times c_{out} \times k \times k}, V^\top \in \mathbb{R}^{r \times c_{out} \times 1 \times 1}$. For

the Batch Normalization layers (BNs) (Ioffe and Szegedy, 2015) we simply extract the weight vectors and the collected running statistics, *e.g.,* the *running mean and variance*, for initializing the low-rank training. We also directly take the bias vector of the last FC layer. Pufferfish then finishes the remaining training epochs over the factorized hybrid network initialized with vanilla warm-up training.

Figure 4.3b provides an experimental justification on the effectiveness of vanilla warm-up training where we study a hybrid ResNet-50 trained on the ImageNet dataset. The results indicate that vanilla warm-up training helps to improve the accuracy of the factorized model. Moreover, a carefully tuned warm-up period of $\hat{E}_{wu}$ also plays an important role in the final model accuracy. Though SVD is computationally heavy, Pufferfish only requires to conduct the SVD **once** throughout the entire training. We benchmark the SVD cost for all experimented models, which indicate the SVD runtime is comparatively small, *e.g.,* on average, it only costs 2.29 seconds for ResNet-50. A complete study on the SVD factorization overheads can be found in Section 4.5.

Last FC layer. The very last FC layer in a neural network can be viewed as a linear classifier over the features extracted by the previous layers. In general, its rank is equal to the number of classes in predictve task at hand. Factorizing it below the number of classes, will increase linear dependencies, and may further increase the approximation error. Thus, Pufferfish does not factorize it.

Putting all the techniques we discussed in this section together, the training procedure of Pufferfish is summarized in Algorithm 2.

4.3 Experiments

We conduct extensive experiments to study the effectiveness and scalability of Pufferfish over various computer vision and natural language processing tasks, across real distributed environments. We also compare Pufferfish against a wide range of baselines including: (i) PowerSGD, a low-rank based, gradient compression method that achieves high compression ratios (Vogels et al., 2019); (ii) Signum a

gradient compression method that only communicates the sign of the local momentum (Bernstein et al., 2018a,b); (iii) The "early bird" structured pruning method *EB Train* (You et al., 2019); and (iv) The LTH sparsification method (referred to as LTH for simplicity) (Frankle and Carbin, 2018).

Our experimental results indicate that PUFFERFISH allows to train a model that is up to 3.35× smaller than other methods, with only marginal accuracy loss. Compared to PowerSGD, SIGNUM, and vanilla SGD, PUFFERFISH achieves 1.22×, 1.52×, and 1.74× end-to-end speedups respectively for ResNet-18 trained on CIFAR-10 while reaching to the same accuracy as vanilla SGD. PUFFERFISH leads to a model with 1.3M fewer parameters while reaching 1.76% higher top-1 test accuracy than EB Train on the ImageNet dataset. Compared to LTH, PUFFERFISH leads to 5.67× end-to-end speedup for achieving the same model compression ratio for VGG-19 on CIFAR-10. We also demonstrate that the performance of PUFFERFISH is stable under the "mixed-precision training" implemented by PyTorch AMP. Our code is publicly available for reproducing our results[1].

4.3.1 Experimental setup and implementation details

Setup. PUFFERFISH is implemented in PyTorch (Paszke et al., 2019). We experiment using two implementations. The first implementation we consider is a data-parallel model training API, *i.e.,* DDP in PyTorch. However, as the gradient computation and communication are overlapped in DDP[2], it is challenging to conduct a breakdown runtime analysis in DDP. We thus also come up with a prototype `allreduce`-based distributed implementation that decouples the computation and communication to benchmark the breakdown runtime of PUFFERFISH and other baselines. Our prototype distributed implementation is based on `allreduce` in PyTorch and the NCCL backend. All our experiments are deployed on a distributed cluster consisting of up to 16 `p3.2xlarge` (Tesla V100 GPU equipped) instances on Amazon EC2.

[1] `https://github.com/hwang595/Pufferfish`

[2] the computed gradients are buffered and communicated immediately when hitting a certain buffer size, *e.g.,* 25MB.

Models and datasets. The datasets considered in our experiments are CIFAR-10 (Krizhevsky et al., 2009), ImageNet (ILSVRC2012) (Deng et al., 2009), the WikiText-2 datasets (Merity et al., 2016), and the WMT 2016 German-English translation task data (Elliott et al., 2016). For the image classification tasks on CIFAR-10, we considered VGG-19-BN (which we refer to as VGG-19) (Simonyan and Zisserman, 2015) and ResNet-18 (He et al., 2016). For ImageNet, we run experiments with ResNet-50 and WideResNet-50-2 (Zagoruyko and Komodakis, 2016). For the WikiText-2 dataset, we considered a 2-layer stacked LSTM model. For the language translation task, we consider a 6-layer Transformer architecture (Vaswani et al., 2017). More details about the datasets and models can be found in Section 4.5.

Implementation details and optimizations. In our prototype distributed implementation, the `allreduce` operation starts right after all compute nodes finish computing the gradient. An important implementation-level optimization we conduct is that we pack all gradient tensors into one flat buffer, and only call the `allreduce` operation **once** per iteration. The motivation for such an optimization is that Pufferfish factorizes the full-rank layer W_l to two smaller layers, *i.e.*, $U_l, V_l^\top$. Though the communication cost of the `allreduce` on each smaller layer is reduced, the total number of `allreduce` calls is doubled (typically an `allreduce` is required per layer to synchronize the gradients across the distributed cluster). According to the run-time cost model of the ring-allreduce (Thakur et al., 2005), each `allreduce` call introduces a network latency proportional to the product of the number of compute nodes and average network latency. This is not a negligible cost. Our optimization strategy aims at minimizing the additional latency overhead and leads to good performance improvement based on our tests. For a fair comparison, we conduct the same communication optimization for all considered baselines.

Hyper-parameters for Pufferfish. For all considered model architectures, we use a global rank ratio of 0.25, *e.g.*, for a convolution layer with an initial rank of 64, Pufferfish sets $r = 64 \times 0.25 = 16$. For the LSTM on WikiText-2 experiment, we only factorize the LSTM layers and leave the tied embedding layer as is. Allocating

Table 4.2: The results (averaged across 3 independent trials with different random seeds) of Pufferfish and the vanilla 2-layer stacked LSTMs trained over the WikiText-2 dataset (since the embedding layer is just a look up table, we do not count it when calculating the MACs).

Model archs.	Vanilla LSTM	Pufferfish LSTM
# Params.	85,962,278	67,962,278
Train Ppl.	52.87 ± 2.43	62.2 ± 0.74
Val Ppl.	92.49 ± 0.41	93.62 ± 0.36
Test Ppl.	88.16 ± 0.39	88.72 ± 0.24
MACs	18M	9M

Table 4.3: The results (averaged across 3 independent trials with different random seeds) of Pufferfish and vanilla 6-layer Transformers trained over the WMT 2016 German to English Translation Task.

Model archs.	Vanilla Transformer	Pufferfish Transformer
# Params.	48,978,432	26,696,192
Train Ppl .	13.68 ± 0.96	$\mathbf{10.27 \pm 0.65}$
Val. Ppl .	11.88 ± 0.43	$\mathbf{7.34 \pm 0.12}$
Test BLEU	19.05 ± 0.59	$\mathbf{26.87 \pm 0.17}$

the optimal rank for each layer can lead to better final model accuracy and smaller model sizes as discussed in (Idelbayev and Carreira-Perpinán, 2020). However, the search space for the rank allocation problem is large. One potential way to solve that problem is to borrow ideas from the literature of neural architectural search (NAS), which we leave as future work. We tune the initial low-rank layer index, *i.e.,* K and the vanilla warm-up training period to balance the hybrid model size and the final model accuracy. More details of the hyper-parameters of Pufferfish can be found in Section 4.5.

Table 4.4: The results (averaged across 3 independent trials with different random seeds) of PUFFERFISH and vanilla VGG-19 and ResNet-18 trained over the CIFAR-10 dataset. Both full-precision training (FP32) and "mixed-precision training" (AMP) results are reported.

Model Archs.	# Params.	Test Acc. (%)	MACs (G)
Vanilla VGG-19 (FP32)	20,560,330	93.91 ± 0.01	0.4
PUFFERFISH VGG-19 (FP32)	8,370,634	93.89 ± 0.14	0.29
Vanilla VGG-19 (AMP)	20,560,330	94.12 ± 0.08	N/A
PUFFERFISH VGG-19 (AMP)	8,370,634	93.98 ± 0.06	N/A
Vanilla ResNet-18 (FP32)	11,173,834	95.09 ± 0.01	0.56
PUFFERFISH ResNet-18 (FP32)	3,336,138	94.87 ± 0.21	0.22
Vanilla ResNet-18 (AMP)	11,173,834	95.02 ± 0.1	N/A
PUFFERFISH ResNet-18 (AMP)	3,336,138	94.70 ± 0.37	N/A

Table 4.5: The results of the vanilla and PUFFERFISH ResNet-50 and WideResNet-50-2 models trained on the ImageNet dataset. For the ResNet-50 results, both full precision training (FP32) and mixed-precision training (AMP) are provided. For the AMP training, MACs are not calculated.

Model Archs.	Num. of . Params	Final Test Acc. (Top-1)	Final Test Acc. (Top-5)	MACs (G)
Vanilla WideResNet-50-2 (FP32)	68,883,240	78.09%	94.00%	11.44
PUFFERFISH WideResNet-50-2 (FP32)	40,047,400	77.84%	93.88%	9.99
Vanilla ResNet-50 (FP32)	25,557,032	76.93%	93.41%	4.12
PUFFERFISH ResNet-50 (FP32)	15,202,344	76.43%	93.10%	3.6
Vanilla ResNet-50 (AMP)	25,557,032	76.97%	93.35%	N/A
PUFFERFISH ResNet-50 (AMP)	15,202,344	76.35%	93.22%	N/A

4.3.2 Results

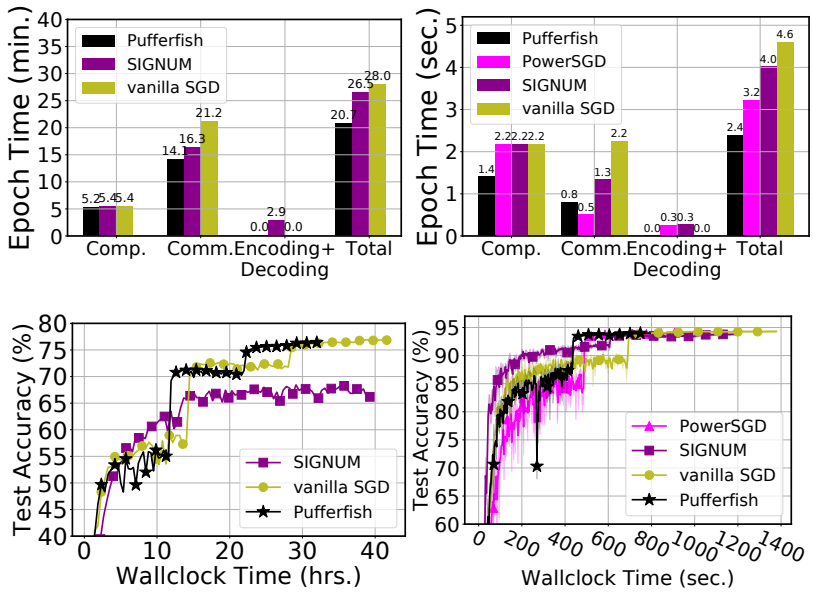
Parameter reduction and model accuracy. We extensively study the effectiveness of PUFFERFISH, and the comprehensive numerical results are shown in Table 4.2,

Table 4.6: The runtime mini-benckmark results of Pufferfish and vanilla VGG-19 and ResNet-18 networks training on the CIFAR-10 dataset. Experiment running on a single V100 GPU with batch size at 128, results averaged over 10 epochs; under the reproducible cuDNN setup with `cudnn.benckmark` disabled and `cudnn.deterministic` enabled; Speedup calculated based on the averaged runtime.

Model Archs.	Epoch Time (sec.)	Speedup	MACs (G)
Vanilla VGG-19	13.51 ± 0.02	–	0.4
Pufferfish VGG-19	$\mathbf{11.02 \pm 0.01}$	**1.23**×	**0.29**
Vanilla ResNet-18	18.89 ± 0.07	–	0.56
Pufferfish ResNet-18	$\mathbf{12.78 \pm 0.03}$	**1.48**×	**0.22**

4.3, 4.4, and 4.5. The main observation is that Pufferfish effectively reduces the number of parameters while introducing only marginal accuracy loss. In particular, Pufferfish ResNet-18 is 3.35× smaller than vanilla ResNet-18 with only 0.22% accuracy loss. Surprisingly, the Pufferfish Transformer leads to even better validation perplexity and test BLEU scores than the vanilla Transformer. One potential reason behind that is that factorizing the Transformer introduces some implicit regularization, leading to better generalization. Apart from the full precision training over FP32, we also conduct mixed-precision experiments over PyTorch AMP on both CIFAR-10 and ImageNet. Our results generally demonstrate that the performance of Pufferfish remains stable under mixed-precision training. We measure the computational complexity using "*multiply–accumulate operations*" (MACs) [3]. The MAC results are shown in Table 4.2, 4.4, and 4.5. The computation complexity is estimated by passing a single input through the entire network, *e.g.,* for the CIFAR-10 dataset, we simulate a color image with size $32 \times 32 \times 3$ and pass it to the networks. For the LSTM network, we assume a single input token is with batch size at 1. We only report the MACs of forward pass. Pufferfish reduces the MACs of the vanilla model to up to 2.55× over ResNet-18 on CIFAR-10.

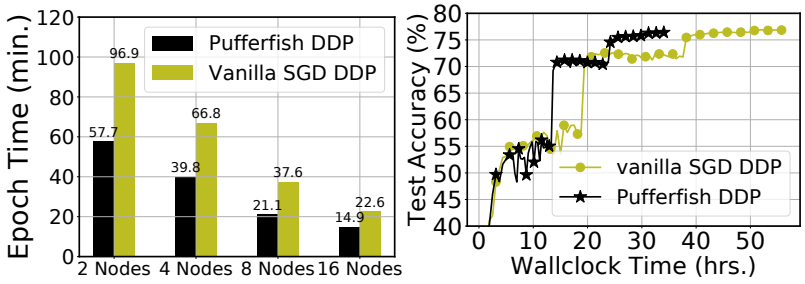
[3] `https://en.wikipedia.org/wiki/Multiply%E2%80%93accumulate_operation`



(a) Proto. ResNet-50 ImageNet (b) Proto. ResNet-18 CIFAR-10

Figure 4.4: (a) Breakdown per-epoch runtime analysis (top) and end-to-end convergence (bottom) results for vanilla SGD, PUFFERFISH, and SIGNUM over ResNet-50 trained on the ImageNet dataset. Where `Comm.` and `Comp.` stands for computation and communication costs under out prototype implementation; (b) Breakdown per-epoch runtime analysis (top) and end-to-end convergence (bottom) results for vanilla SGD, PUFFERFISH, SIGNUM, and PowerSGD over ResNet-18 trained on CIFAR-10 under out prototype implementation.

Runtime mini-benchmark. It is of interest to investigate the actual speedup of the factorized networks as they are dense and compact. We thus provide mini-benchmark runtime results over VGG-19 and ResNet-18 on the CIFAR-10 dataset. We measure the per-epoch training speed of the factorized networks used in PUFFERFISH and the vanilla networks on a single V100 GPU with batch size at 128. The results are shown in Table 4.6. We report the results (averaged over 10 epochs) under the reproducibility optimized cuDNN environment, *i.e.*, `cudnn.benckmark` disabled and `cudnn.deterministic` enabled. The results indicate that the factorized networks enjoy promising runtime speedups, *i.e.*, 1.23× and 1.48× over the vanilla VGG-19 and ResNet-18 respectively. We also study the runtime of the factorized networks under the speed optimized cuDNN setting, *i.e.*, `cudnn.benckmark`



(a) Scalability (b) End-to-end convergence

Figure 4.5: (a) The scalability of PUFFERFISH compared to vanilla SGD for ResNet-50 training on ImageNet using PyTorch DDP over the distributed clusters that consist of $2, 4, 8, 16$ nodes. (b) End-to-end convergence for vanilla SGD and PUFFERFISH with PyTorch DDP under the cluster with 8 nodes (bottom).

enabled and `cudnn.deterministic` disabled, the results can be found in Section 4.5.

Computation and communication efficiency. To benchmark the computation and communication costs of PUFFERFISH under a distributed environment, we conduct a per-epoch breakdown runtime analysis and compare it to vailla SGD and SIGNUM on ResNet-50, trained over ImageNet. The experiment is conducted over 16 `p3.2xlarge` EC2 instances. We set the global batch size at 256 (16 per node). We use tuned hyper-parameters for all considered baselines. The result is shown in Figure 4.4a where we observe that the PUFFERFISH ResNet-50 achieves $1.35\times$ and $1.28\times$ per-epoch speedups compared to vanilla SGD and SIGNUM respectively. Note that though SIGNUM achieves high compression ratio, it is not compatible with `allreduce`, thus `allgather` is used instead in our SIGNUM implementation. However, `allgather` is less efficient than `allreduce`, which hurts the communication efficiency of SIGNUM. The effect has also been observed in the previous literature (Vogels et al., 2019). We extend the per-epoch breakdown runtime analysis to ResNet-18 trainining on CIFAR-10 where we compare PUFFERFISH to POWERSGD, SIGNUM, and vanilla SGD. The experiments are conducted over 8 `p3.2xlarge` EC2 instances with the global batch size at 2048 (256 per node). We use a linear learning rate warm-up for 5 epochs from 0.1 to 1.6, which follows the setting in (Vogels et al., 2019; Goyal et al.,

2017). For PowerSGD, we set the rank at 2, as it matches the same accuracy compared to vanilla SGD (Vogels et al., 2019). The results are shown in Figure 4.4b, from which we observe that Pufferfish achieves $1.33\times, 1.67\times, 1.92\times$ per-epoch speedups over PowerSGD, signum, and vanilla SGD respectively. Note that Pufferfish is slower than PowerSGD in the communication stage since PowerSGD massively compresses gradient and is also compatible with `allreduce`. However, Pufferfish is faster for gradient computing and bypasses the gradient encoding and decoding steps. Thus, the overall epoch time cost of Pufferfish is faster than PowerSGD. Other model training overheads, *e.g.,* data loading and pre-processing, gradient flattening, and etc are not included in the "computation" stage but in the overall per-epoch time.

Since Pufferfish only requires to modify the model architectures instead of gradients, it is directly compatible with current data-parallel training APIs, *e.g.,* DDP in PyTorch. Other gradient compression methods achieve high compression ratio, but they are not directly compatible with DDP without significant engineering effort. For PyTorch DDP, we study the speedup of Pufferfish over vanilla distributed training, measuring the per-epoch runtime on ResNet-50 and ImageNet over distributed clusters of size $2, 4, 8$, and 16. We fix the per-node batch size at 32 following the setup in (Goyal et al., 2017). The results are shown in Figure 4.5b. We observe that Pufferfish consistently outperforms vanilla ResNet-50. In particular, on the cluster with 16 nodes, Pufferfish achieves $1.52\times$ per epoch speedup.

End-to-end speedup. We study the end-to-end speedup of Pufferfish against other baselines under both our prototype implementation and PyTorch DDP. The experimental setups for the end-to-end experiment are identical to our per-epoch breakdown runtime analysis setups. All reported runtimes include the overheads of the SVD factorization and vanilla warm-up training. The ResNet-50 on ImageNet convergence results with our prototype implementation are shown in Figure 4.4a. We observe that to finish the entire 90 training epochs, Pufferfish attains $1.3\times$ and $1.23\times$ end-to-end speedups compared to vanilla SGD and Signum respectively. The ResNet-18 on CIFAR-10 convergence results are shown in Figure 4.4b. For faster

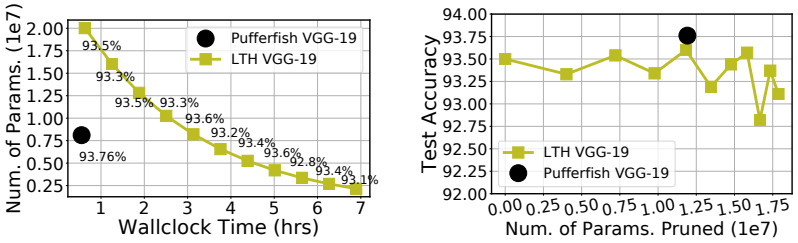
vanilla warm-up training in PUFFERFISH, we deploy POWERSGD to compress the gradients. We observe that it is generally better to use a slightly higher rank for POWERSGD in the vanilla warm-up training period of PUFFERFISH. In our experiments, we use POWERSGD with rank 4 to warm up PUFFERFISH. We observe that to finish the entire 300 training epochs, PUFFERFISH attains $1.74\times, 1.52\times, 1.22\times$ end-to-end speedup compared to vanilla SGD, SIGNUM, and POWERSGD respectively. PUFFERFISH reaches to the same accuracy compared to vanilla SGD. Moreover, we extend the end-to-end speedup study under PyTorch DDP where we compare PUFFERFISH with vanilla SGD under 8 EC2 `p3.2xlarge` instances. The global batch size is 256 (32 per node). The results are shown in Figure 4.5b where we observe that to train the model for 90 epochs, PUFFERFISH achieves $1.64\times$ end-to-end speedup compared to vanilla SGD. We do not study the performance of SIGNUM and POWERSGD under DDP since they are not directly compatible with DDP.

Table 4.7: Comparison of Hybrid ResNet-50 model compared to the Early-Bird Ticket structure pruned (EB Train) ResNet-50 model results with prune ratio pr at 30%, 50%, 70% over the ImageNet dataset

Model architectures	# Params.	Final Test Acc. (Top-1)	Final Test Acc. (Top-5)	MACs (G)
vanilla ResNet-50	25,610,205	75.99%	92.98%	4.12
PUFFERFISH ResNet-50	15,202,344	75.62%	92.55%	3.6
EB Train (pr = 30%)	16,466,787	73.86%	91.52%	2.8
EB Train (pr = 50%)	15,081,947	73.35%	91.36%	2.37
EB Train (pr = 70%)	7,882,503	70.16%	89.55%	1.03

Comparison with structured pruning. We compare PUFFERFISH with the EB Train method where structured pruning is conducted over the channel dimensions based on the activation values during the early training phase (You et al., 2019). EB Train finds compact and dense models. The result is shown in Table 4.7. We observe that compared to EB Train with prune ratio (pr) = 30%, PUFFERFISH returns a

model with 1.3M fewer parameters while reaching 1.76% higher top-1 test accuracy. The EB Train experimental results are taken directly from the original paper (You et al., 2019). To make a fair comparison, we train PUFFERFISH with the same hyper-parameters that EB Train uses, *e.g.*, removing label smoothing and only decaying the learning rate at the 30-th and the 60-th epochs with the factor 0.1.



(a) Model size *vs* runtime (b) Model size *vs* Test acc.

Figure 4.6: The performance comparison between PUFFERFISH and LTH over a VGG-19 model trained over the CIFAR-10 dataset: (a) the number of parameters *v.s.* wall-clock runtime; (b) the number of parameters pruned *v.s.* the test accuracy.

Comparison with LTH. The recent LTH literature initiated by Frankle et al. (Frankle and Carbin, 2018), indicates that dense, randomly-initialized networks contain sparse subnetworks (referred to as "*winning tickets*") that—when trained in isolation—reach test accuracy comparable to the original network (Frankle and Carbin, 2018). To find the winning tickets, an iterative pruning algorithm is conducted, which trains, prunes, and rewinds the remaining unpruned elements to their original random values repeatedly. Though LTH can compress the model massively without significant accuracy loss, the iterative pruning is computationally heavy. We compare PUFFERFISH to LTH across model sizes and computational costs on VGG-19 trained with CIFAR-10. The results are shown in Figure 4.6a, 4.6b where we observe that to prune the same number of parameters, LTH costs 5.67$\times$ more time than PUFFERFISH.

Ablation study. We conduct an ablation study on the accuracy loss mitigation methods in PUFFERFISH, *i.e.*, hybrid network and vanilla warm-up training. The

results on ResNet-18+CIFAR-10 are shown in Table 4.8, which indicate that the hybrid network and vanilla warm-up training methods help to mitigate the accuracy loss effectively. Results on the other datasets can be found in Section 4.5.

Table 4.8: The effect of vanilla warm-up training and hybrid network architectures of Pufferfish of the low rank ResNet-18 trained over the CIFAR-10 dataset. Results are averaged across 3 independent trials with different random seeds.

Methods	Test Loss	Test Acc. (%)
Low-rank ResNet-18	0.31 ± 0.01	93.75 ± 0.19
Hybrid ResNet-18 (wo. vanilla warm-up)	0.30 ± 0.02	93.92 ± 0.45
Hybrid ResNet-18 (w. vanilla warm-up)	$\mathbf{0}.25 \pm 0.01$	$\mathbf{9}4.87 \pm 0.21$

Limitations of Pufferfish. One limitation of Pufferfish is that it introduces two extra hyper-parameters, *i.e.,* the initial low-rank layer index K and the vanilla warm-up epoch E_{wu}, hence hyperparameter tuning requires extra effort. Another limitation is that although Pufferfish reduces the parameters in ResNet-18 and VGG-19 models effectively for the CIFAR-10 dataset, it only finds $1.68\times$ and $1.72\times$ smaller models for ResNet-50 and WideResNet-50-2 in order to preserve good final model accuracy.

4.4 Conclusion

In this chapter, we propose Pufferfish, a communication and computation efficient distributed training framework. Instead of gradient compression, Pufferfish trains low-rank networks initialized by factorizing a partially trained full-rank model. The use of a hybrid low-rank model and warm-up training, allows Pufferfish to preserve the accuracy of the fully dense SGD trained model, while effectively reducing its size. Pufferfish achieves high computation and communication efficiency and completely bypasses the gradient encoding and decoding, while yielding smaller

and more accurate models compared pruning methods such the LTH and EB Train, while avoiding the burden of "winning the lottery".

4.5 Additional details on Pufferfish

4.5.1 Software versions used in the experiments

Since we provide wall-clock time results, it is important to specify the versions of the libraries we used. For the full-precision (FP32) results, we used the `pytorch_p36` virtual environment associated with the "Deep Learning AMI (Ubuntu 18.04) Version 40.0 (ami-084f81625fbc98fa4)" on Amazon EC2, *i.e.*, `PyTorch 1.4.0` with CUDA10.1.243. Since AMP is only supported after 1.6.0 of PyTorch, we use `PyTorch 1.6.0` with CUDA 10.1.

4.5.2 Detailed discussion on the computation complexity of various layers

We provide the computational complexity and number of parameters in the vanilla and low-rank factorized layers returned by Pufferfish in Section 4.1. We provide a more detailed discussion here. For the number of parameters, we do not count the bias terms.

FC layer. We start from the FC layer. Assuming the input and the vanilla and low-rank FC weights are with dimensions $x \in \mathbb{R}^m, W \in \mathbb{R}^{m\times n}, U \in \mathbb{R}^{m\times r}, V^\top \in \mathbb{R}^{r\times n}$, the computation complexity is simply $\mathcal{O}(mn)$ for xW and $\mathcal{O}(mr + rn)$ for $(xU)V^\top$. For the number of parameters, the vanilla FC layer contains mn parameters in total while the low-rank FC layer contains $r(m + n)$ parameters in total.

Convolution layer. For a convolution layer, assuming the input is with dimension $x \in \mathbb{R}^{c_{in}\times H\times W}$ (the input "image" has c_{in} color channels and with size $H \times W$), the computation complexity of a vanilla convolution layer with weight $W \in$

$\mathbb{R}^{c_{in} \times c_{out} \times k \times k}$ is $\mathcal{O}(c_{in} c_{out} k^2 HW)$ for computing $W * x$ where $*$ is the linear convolution operation. And the low-rank factorized convolution layer with dimension $U \in \mathbb{R}^{c_{in} \times r \times k \times k}, V \in \mathbb{R}^{r \times c_{out} \times 1 \times 1}$ has the computation complexity at $\mathcal{O}(r c_{in} k^2 HW)$ for $U * x$ and $\mathcal{O}(rHW c_{out})$ for convolving the output of $U * x$ with V. For the number of parameters, the vanilla convolution layer contains $c_{in} c_{out} k^2$ parameters in total while the low-rank convolution layer contains $c_{in} r k^2 + r c_{out}$ parameters in total.

LSTM layer. For the LSTM layer, the computation complexity is similar to the computation complexity of the FC layer. Assuming the tokenized input is with dimension $x \in \mathbb{R}^d$, and the concatenated input-hidden and hidden-hidden weights $W_i \in \mathbb{R}^{d \times 4h}, W_h \in \mathbb{R}^{4h \times h}$, thus the computation complexity of the forward propagation of a LSTM layer is $\mathcal{O}(4dh + 4h^2)$. And for the low-rank LSTM layer, the computation complexity becomes $\mathcal{O}(dr + 4rh + 4hr + rh)$ (as mentioned in Section 4.1, we assume that the same rank r is used for both the input-hidden weight and hidden-hidden weight). For the number of parameters, the vanilla LSTM layer contains $4dh + 4h^2$ parameters in total while the low-rank convolution layer contains $4(dr + rh) + 4(hr + rh) = 4dr + 12hr$ parameters in total.

Transformer. For the encoder layer in the Transformer architecture, there are two main components, *i.e.,* the multi-head attention layer and the FFN layer. Note that, for the multi-head attention layer, the dimensions of the projection matrices are: The dimensions of the matrices are $Q, K, V \in \mathbb{R}^{n \times pd}, W^Q, W^K, W^V \in \mathbb{R}^{pd \times d}, W^O \in \mathbb{R}^{pd \times pd}$. And the dimensions of the two FC layers in the FFN are with dimensions $W_1 \in \mathbb{R}^{pd \times 4pd}, W_2 \in \mathbb{R}^{4pd \times pd}$. And we assume a sequence of input tokens with length N is batched to process together. Since the computation for each attention head is computed independently, we only analyze the computation complexity of a single head attention, which is

$$\mathcal{O}\Big(\underbrace{d \cdot pd \cdot N}_{\text{proj. of } Q,K,V} + \underbrace{2N^2 \cdot d}_{\text{attention layer}} + \underbrace{pd \cdot pd \cdot N}_{\text{proj. of the output of attention}} \Big) = \mathcal{O}\big((p + p^2) N d^2 + N^2 d\big) = \mathcal{O}\big(N p^2 d^2 + N^2 d\big).$$

Similarly, the computation complexity for the FFN layer is

$\mathcal{O}\big(\underbrace{4 \times p^2 d^2 N}_{xW_1} + \underbrace{4 \times p^2 d^2 N}_{xW_1W_2}\big)$. For the low-rank attention layer, the computation complexity becomes

$\mathcal{O}\big(\underbrace{(dr + rpd) \cdot N}_{\text{low-rank proj.}} + \underbrace{(pdr + rpd) \cdot N}_{\text{low-rank proj. of the output}} + 2N^2 \cdot d\big) = \mathcal{O}\big((p+1)drN + 2Ndpr + 2N^2 d\big) = \mathcal{O}\big(pdrN + N^2 d\big)$ and the computation complexity for FFN

$\mathcal{O}\big(\underbrace{(p \cdot d \cdot r + 4r \cdot h \cdot d) \cdot N}_{xW_1} + \underbrace{(p \cdot d \cdot r + 4r \cdot p \cdot d) \cdot N}_{xW_1W_2}\big)$. For the number of parameters, the vanilla multi-head attention layer contains $3pd^2 \cdot p + p^2 d^2 = 4p^2 d^2$ parameters in total while the low-rank multi-head attention layer contains $3p(pdr + rd) + (pdr + rpd) = prd(3p + 5)$ parameters in total. The vanilla FFN layer contains $4p^2 d^2 + 4p^2 d^2 = 8p^2 d^2$ parameters in total while the low-rank FFN layer contains $(pdr + r4pd) + (4pdr + rpd) = 10pdr$ parameters in total.

4.5.3 Details on the dataset and models used for the experiment

The details of the computer vision tasks and natural language processing tasks experimented in the experiments are summarized in Table 4.9 and Table 4.10 respectively.

Table 4.9: The datasets used and their associated learning models for computer vision tasks.

Method	CIFAR-10	ImageNet
# Data points	60,000	1,281,167
Data Dimension	$32 \times 32 \times 3$	$224 \times 224 \times 3$
Model	VGG-19-BN;ResNet-18	ResNet-50; WideResNet-50-2
Optimizer	SGD, Init lr: 0.01, momentum: 0.9, ℓ_2 weight decay: 10^{-4}	

Table 4.10: The datasets used and their associated learning models for natural language processing tasks.

Method	WikiText-2	WMT16' Gen-Eng
# Data points	$29,000$	$1,017,981$
Data Dimension	$1,500$	$9,521$
Model	2 layer LSTM	Transformer ($p = 8, N = 6$)
Optimizer	SGD	Adam
Hyper-params.	lr: 20 (decay 0.25 val. loss not decreas.)	Init lr: 0.001
	grad. norm clipping 0.25	$\beta s = (0.9, 0.98), \epsilon = 10^{-8}$

4.5.4 Details on the hybrid networks in the experiments

The hybrid VGG-19-BN architecture. we found that using $K = 10$ in the VGG-19-BN architecture leads to good test accuracy and moderate model compression ratio.

The low-rank LSTM architecture. Note that we only use a 2-layer stacked LSTM as the model in the WikiText-2 next word prediction task. Our implementation is directly modified from the PyTorch original example [4]. We used the tied version of LSTM, *i.e.*, enabling weight sharing for the encode embedding and decode embedding layers.

The hybrid ResNet-18, ResNet-50, WideResNet-50-2 architectures. For CIFAR-10 dataset, we modified the original ResNet-50 architecture described in the original ResNet paper (He et al., 2016). The details about the modified ResNet-18 architecture for the CIFAR-10 dataset are shown in Table 4.13. The network architecture is modified from the public code repository [5]. For the first 2 convolution block, *i.e.*, conv2_x, we used stride at 1 and padding at 1 for all the convolution layers. For conv3_x, conv4_x, and conv5_x we used the stride at 2 and padding at 1. We

[4] `https://github.com/pytorch/examples/tree/master/word_language_model`
[5] `https://github.com/kuangliu/pytorch-cifar`

Table 4.11: Detailed information of the hybrid VGG-19-BN architecture used in our experiments, all non-linear activation function in this architecture is ReLU after each convolution layer (omitted in the Table). The shapes for convolution layers follows (c_{in}, c_{out}, k, k). There is a BatchNorm layer after each convolution layer with number of neurons the same as c_{out} (also omitted in the Table).

Parameter	Shape	Layer hyper-parameter
layer1.conv1.weight	$3 \times 64 \times 3 \times 3$	stride:1;padding:1
layer2.conv2.weight	$64 \times 64 \times 3 \times 3$	stride:1;padding:1
pooling.max	N/A	kernel size:2;stride:2
layer3.conv3.weight	$64 \times 128 \times 3 \times 3$	stride:1;padding:1
layer4.conv4.weight	$128 \times 128 \times 3 \times 3$	stride:1;padding:1
pooling.max	N/A	kernel size:2;stride:2
layer5.conv5.weight	$128 \times 256 \times 3 \times 3$	stride:1;padding:1
layer6.conv6.weight	$256 \times 256 \times 3 \times 3$	stride:1;padding:1
layer7.conv7.weight	$256 \times 256 \times 3 \times 3$	stride:1;padding:1
layer8.conv8.weight	$256 \times 256 \times 3 \times 3$	stride:1;padding:1
pooling.max	N/A	kernel size:2;stride:2
layer9.conv9.weight	$256 \times 512 \times 3 \times 3$	stride:1;padding:1
layer10.conv10_u.weight	$512 \times 128 \times 3 \times 3$	stride:1;padding:1
layer10.conv10_v.weight	$128 \times 512 \times 1 \times 1$	stride:1
layer11.conv11_u.weight	$512 \times 128 \times 3 \times 3$	stride:1;padding:1
layer11.conv11_v.weight	$128 \times 512 \times 1 \times 1$	stride:1
layer12.conv12_u.weight	$512 \times 128 \times 3 \times 3$	stride:1;padding:1
layer12.conv12_v.weight	$128 \times 512 \times 1 \times 1$	stride:1
pooling.max	N/A	kernel size:2;stride:2
layer13.conv13_u.weight	$512 \times 128 \times 3 \times 3$	stride:1;padding:1
layer13.conv13_v.weight	$128 \times 512 \times 1 \times 1$	stride:1
layer14.conv14_u.weight	$512 \times 128 \times 3 \times 3$	stride:1;padding:1
layer14.conv14_v.weight	$128 \times 512 \times 1 \times 1$	stride:1
layer15.conv15_u.weight	$512 \times 128 \times 3 \times 3$	stride:1;padding:1
layer15.conv15_v.weight	$128 \times 512 \times 1 \times 1$	stride:1
layer16.conv16_u.weight	$512 \times 128 \times 3 \times 3$	stride:1;padding:1
layer16.conv16_v.weight	$128 \times 512 \times 1 \times 1$	stride:1
pooling.max	N/A	kernel size:2;stride:2
layer17.fc17.weight	512×512	N/A
layer17.fc17.bias	512	N/A
layer18.fc18.weight	512×512	N/A
layer18.fc18.bias	512	N/A
layer19.fc19.weight	512×10	N/A
layer19.fc19.bias	10	N/A

Table 4.12: Detailed information on the low-rank LSTM architecture in our experiment.

Parameter	Shape	Hyper-param.
encoder.weight	33278×1500	N/A
dropout	N/A	$p = 0.65$
lstm0.weight.ii/f/g/o_u	1500×375	N/A
lstm0.weight.ii/f/g/o_v	375×1500	N/A
lstm0.weight.hi/f/g/o_u	1500×375	N/A
lstm0.weight.hi/f/g/o_v	375×1500	N/A
dropout	N/A	$p = 0.65$
lstm1.weight.ii/f/g/o_u	1500×375	N/A
lstm1.weight.ii/f/g/o_v	375×1500	N/A
lstm1.weight.hi/f/g/o_u	1500×375	N/A
lstm1.weight.hi/f/g/o_v	375×1500	N/A
decoder.weight(shared)	1500×33278	N/A

also note that there is a BatchNorm layer after each convolution layer with the number of elements equals the number of convolution filters. As shown in Table 4.13, our hybrid architecture starts from the 2nd convolution block, *i.e.,* $K = 4$. Our experimental study generally shows that this choice of hybrid ResNet-18 architecture leads to a good balance between the final model accuracy and the number of parameters. Moreover, we do not handle the downsampling weights in the convolution blocks.

For the ResNet-50 architecture, the detailed information is shown in the Table 4.14. As we observed that the last three convolution blocks, *i.e.,* conv5_x contains around 60% of the total number of parameters in the entire network. Thus, we just factorize the last three convolution blocks and leave all other convolution blocks as full-rank blocks. Note that, different from the ResNet-18 architecture for the CIFAR-10 dataset described above. We also handle the downsampling layer weights inside the ResNet-50 network, which only contains in the very first convolution block of conv5_x. The original dimension of the downsample weight

Table 4.13: The hybrid ResNet-18 architecture for the CIFAR-10 dataset used in the experiments.

Layer Name	ResNet-18	Rank Information
conv1	3×3, 64, stride 1, padding 1	full-rank
conv2_x	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	1st block full-rank 2nd block low-rank conv_u $(64, 16, 3, 3)$, conv_v$(16, 64, 1, 1)$
conv3_x	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	low-rank conv_u $(128, 32, 3, 3)$ conv_v $(32, 128, 1, 1)$
conv4_x	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	low-rank conv_u $(256, 64, 3, 3)$ conv_v $(64, 256, 1, 1)$
conv5_x	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	low-rank conv_u $(512, 128, 3, 3)$ conv_v $(128, 512, 1, 1)$
	Avg Pool, 10-dim FC, SoftMax	

is with shape $(1024, 2048, 1, 1)$. Our factorization strategy leads to the shape of conv_u: $(1024, 256, 1, 1)$ and conv_v: $(256, 2048, 1, 1)$. For the WideResNet-50, the detailed hybrid architecture we used is shown in Table 4.15. Similar to our hybrid ResNet-50 architecture, we just factorize the last three convolution blocks and leave all other convolution blocks as full-rank blocks. We also handle the downsampling layer weights inside the WideResNet-50 network, which only contains the very first convolution block of conv5_x. The original dimension of the downsample weight is with shape $(1024, 2048, 1, 1)$. Our factorization strategy leads to the shape of conv_u: $(1024, 256, 1, 1)$ and conv_v: $(256, 2048, 1, 1)$.

The hybrid Transformer architecture. The Transformer architecture used in the experiment follows from the original Transformer paper (Vaswani et al., 2017).

Table 4.14: The hybrid ResNet-50 architecture for the ImageNet dataset used in the experiments.

Layer Name	output size	ResNet-50	Rank Information
conv1	112×112	7×7, 64, stride 2	full-rank
conv2_x	56×56	3×3 max pool, stride 2	
		$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix} \times 3$	all blocks full-rank
conv3_x	28×28	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix} \times 4$	all blocks full-rank
conv4_x	14×14	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix} \times 6$	all blocks full-rank
conv5_x	7×7	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix} \times 3$	conv_1_u $(c_{in}, \frac{c_{in}}{4}, 1, 1)$; conv_1_v $(\frac{c_{in}}{4}, 512, 1, 1)$ conv_2_u $(512, 128, 3, 3)$; conv_2_v $(128, 512, 1, 1)$ conv_3_u $(512, 128, 1, 1)$; conv_2_v $(128, 2048, 1, 1)$
	1×1	Avg pool, 1000-dim FC, SoftMax	

Table 4.15: The hybrid WideResNet-50-2 architecture for the ImageNet dataset used in the experiments.

Layer Name	output size	WideResNet-50-2	Rank Information
conv1	112×112	7×7, 64, stride 2	full-rank
conv2_x	56×56	3×3 max pool, stride 2	
		$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 256 \end{bmatrix} \times 3$	all blocks full-rank
conv3_x	28×28	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 512 \end{bmatrix} \times 4$	all blocks full-rank
conv4_x	14×14	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 1024 \end{bmatrix} \times 6$	all blocks full-rank
conv5_x	7×7	$\begin{bmatrix} 1\times1, 1024 \\ 3\times3, 1024 \\ 1\times1, 2048 \end{bmatrix} \times 3$	conv_1_u $(c_{in}, \frac{c_{in}}{4}, 1, 1)$; conv_1_v $(\frac{c_{in}}{4}, 1024, 1, 1)$ conv_2_u $(1024, 256, 3, 3)$; conv_2_v $(256, 1024, 1, 1)$ conv_3_u $(1024, 256, 1, 1)$; conv_2_v $(256, 2048, 1, 1)$
	1×1	Avg pool, 1000-dim FC, SoftMax	

Our implementation is modified from the public code repository [6]. We use the stack of $N = 6$ encoder and decoder layers inside the Transformer architecture and number of head $p = 8$. Since the encoder and decoder layers are identical across the entire architecture, we describe the detailed encoder and decoder architecture information in Table 4.16 and Table 4.17. For the hybrid architecture used in the Transformer architecture, we put the very first encoder layer and first decoder layer as full-rank layers, and all other layers are low-rank layers. For low-rank encoder and decoder layers, we used the rank ratio at $\frac{1}{4}$, thus the shape of $U^Q, U^K, U^V, U^O \in \mathbb{R}^{512\times128}, V^{Q\top}, V^{K\top}, V^{V\top}, V^{O\top} \in \mathbb{R}^{128\times512}$. For W_1 in the FFN$(\cdot)$ layer, the $U_1 \in \mathbb{R}^{512\times128}, V_1^\top \in \mathbb{R}^{128\times2048}$. For W_2 in the FFN$(\cdot)$ layer, the $U_2 \in \mathbb{R}^{2048\times128}, V_1^\top \in \mathbb{R}^{128\times512}$.

Table 4.16: Detailed information of the encoder layer in the Transformer architecture in our experiment

Parameter	Shape	Hyper-param.
embedding	9521×512	padding index: 1
positional encoding	N/A	N/A
dropout	N/A	$p = 0.1$
encoder.self-attention.wq(W^Q)	512×512	N/A
encoder.self-attention.wk(W^K)	512×512	N/A
encoder.self-attention.wv(W^V)	512×512	N/A
encoder.self-attention.wo(W^O)	512×512	N/A
encoder.self-attention.dropout	N/A	$p = 0.1$
encoder.self-attention.layernorm	512	$\epsilon = 10^{-6}$
encoder.ffn.layer1	512×2048	N/A
encoder.ffn.layer2	2048×512	N/A
encoder.layernorm	512	$\epsilon = 10^{-6}$
dropout	N/A	$p = 0.1$

[6] `https://github.com/jadore801120/attention-is-all-you-need-pytorch`

Table 4.17: Detailed information of the decoder layer in the Transformer architecture in our experiment

Parameter	Shape	Hyper-param.
embedding	9521×512	padding index: 1
positional encoding	N/A	N/A
dropout	N/A	$p = 0.1$
decoder.self-attention.wq(W^Q)	512×512	N/A
decoder.self-attention.wk(W^K)	512×512	N/A
decoder.self-attention.wv(W^V)	512×512	N/A
decoder.self-attention.wo(W^O)	512×512	N/A
decoder.self-attention.dropout	N/A	$p = 0.1$
decoder.self-attention.layernorm	512	$\epsilon = 10^{-6}$
decoder.enc-attention.wq(W^Q)	512×512	N/A
decoder.enc-attention.wk(W^K)	512×512	N/A
decoder.enc-attention.wv(W^V)	512×512	N/A
decoder.enc-attention.wo(W^O)	512×512	N/A
decoder.enc-attention.dropout	N/A	$p = 0.1$
decoder.enc-attention.layernorm	512	$\epsilon = 10^{-6}$
decoder.ffn.layer1	512×2048	N/A
decoder.ffn.layer2	2048×512	N/A
encoder.layernorm	512	$\epsilon = 10^{-6}$
dropout	N/A	$p = 0.1$

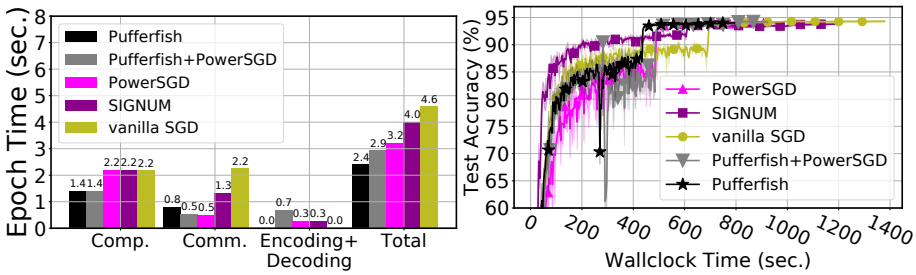
The hybrid VGG-19-BN architecture used for the LTH comparison. To compare Pufferfish with LTH, we use the open-source LTH implementation, *i.e.*, `https://github.com/facebookresearch/open_lth`. The VGG-19-BN model used in the open-source LTH repository is slightly different from the VGG-19-BN architecture described above. We thus use the VGG-19-BN architecture in the LTH code and deploy Pufferfish on top of it for fairer comparison. Detailed information about the hybrid VGG-19-BN architecture we used in Pufferfish for the comparison with LTH is shown in Table 4.18.

Table 4.18: Detailed information of the hybrid VGG-19-BN architecture used in our LTH comparison experiments, all non-linear activation function in this architecture is ReLU after each convolution layer (omitted in the Table). The shapes for convolution layers follows (c_{in}, c_{out}, k, k). There is a BatchNorm layer after each convolution layer with number of neurons the same as c_{out} (also omitted in the Table).

Parameter	Shape	Layer hyper-parameter
layer1.conv1.weight	$3 \times 64 \times 3 \times 3$	stride:1;padding:1
layer2.conv2.weight	$64 \times 64 \times 3 \times 3$	stride:1;padding:1
pooling.max	N/A	kernel size:2;stride:2
layer3.conv3.weight	$64 \times 128 \times 3 \times 3$	stride:1;padding:1
layer4.conv4.weight	$128 \times 128 \times 3 \times 3$	stride:1;padding:1
pooling.max	N/A	kernel size:2;stride:2
layer5.conv5.weight	$128 \times 256 \times 3 \times 3$	stride:1;padding:1
layer6.conv6.weight	$256 \times 256 \times 3 \times 3$	stride:1;padding:1
layer7.conv7.weight	$256 \times 256 \times 3 \times 3$	stride:1;padding:1
layer8.conv8.weight	$256 \times 256 \times 3 \times 3$	stride:1;padding:1
pooling.max	N/A	kernel size:2;stride:2
layer9.conv9.weight	$256 \times 512 \times 3 \times 3$	stride:1;padding:1
layer10.conv10_u.weight	$512 \times 128 \times 3 \times 3$	stride:1;padding:1
layer10.conv10_v.weight	$128 \times 512 \times 1 \times 1$	stride:1
layer11.conv11_u.weight	$512 \times 128 \times 3 \times 3$	stride:1;padding:1
layer11.conv11_v.weight	$128 \times 512 \times 1 \times 1$	stride:1
layer12.conv12_u.weight	$512 \times 128 \times 3 \times 3$	stride:1;padding:1
layer12.conv12_v.weight	$128 \times 512 \times 1 \times 1$	stride:1
pooling.max	N/A	kernel size:2;stride:2
layer13.conv13_u.weight	$512 \times 128 \times 3 \times 3$	stride:1;padding:1
layer13.conv13_v.weight	$128 \times 512 \times 1 \times 1$	stride:1
layer14.conv14_u.weight	$512 \times 128 \times 3 \times 3$	stride:1;padding:1
layer14.conv14_v.weight	$128 \times 512 \times 1 \times 1$	stride:1
layer15.conv15_u.weight	$512 \times 128 \times 3 \times 3$	stride:1;padding:1
layer15.conv15_v.weight	$128 \times 512 \times 1 \times 1$	stride:1
layer16.conv16_u.weight	$512 \times 128 \times 3 \times 3$	stride:1;padding:1
layer16.conv16_v.weight	$128 \times 512 \times 1 \times 1$	stride:1
pooling.max	N/A	kernel size:2;stride:2
layer17.fc17.weight	512×10	N/A
layer17.fc17.bias	10	N/A

4.5.5 The compatibility of Pufferfish with other gradient compression methods

As Pufferfish is a training time parameter reduction method, the gradient of the factorized networks can be compressed further with any gradient compression methods. As PowerSGD is the state-of-the-art gradient compression method and is compatible with `allreduce`, we consider another baseline, *i.e.,* "Pufferfish+PowerSGD". We conduct an experimental study over the "Pufferfish+PowerSGD" baseline for ResNet-18 trained on CIFAR-10 (results shown in Figure 4.7). The experiment is running over 8 `p3.2xlarge` EC2 nodes with batch size at 256 per node (2048 in total). The experimental results indicate that combining Pufferfish with PowerSGD can effectively reduce the gradient size of Pufferfish further, making Pufferfish enjoys high computation efficiency and the communication efficiency as high as PowerSGD. However, as PowerSGD conducts layer-wise gradient encoding and decoding on all U_l and V_l layers, the gradient encoding and decoding costs in the "Pufferfish+PowerSGD" baseline are higher compared to PowerSGD. We observe that a slightly higher rank is desired when combining Pufferfish with PowerSGD to attain good final model accuracy since both model weights and gradients are approximated in this case. In the experimental results shown in Figure 4.7, we use PowerSGD with rank 4 when combining with Pufferfish for both the vanilla warm-up training and the following low-rank training. Moreover, we also found that under the large-batch setting, it is always helpful to re-warmup the learning rate for the "Pufferfish+PowerSGD" baseline, *i.e.,* in the first 5 epochs, we warm-up the learning rate linearly from 0.1 to 1.6, then at the 80-th epoch where we switch from the vanilla warm-up training to low-rank training, we conduct the learning rate warm-up again within 5 epochs (from 0.1 to 1.6). Our experimental results suggest that Pufferfish can be combined with the gradient compression methods to attain better communication efficiency, but it is desirable to combine Pufferfish with the gradient compression methods that can be deployed directly on the fattened gradients, *e.g.,* Top-k.



(a) Breakdown per-epoch time (b) Convergence

Figure 4.7: (a) Per-epoch breakdown runtime analysis and (b) convergence performance of PUFFERFISH, "PUFFERFISH+POWERSGD (rank 4)", POWERSGD (rank 2), SIGNUM, and vanilla SGD over ResNet-18 trained on the CIFAR-10 dataset.

4.5.6 Discussion on the communication efficiency of PUFFERFISH

It is natural to ask the question that "*Why are the previously proposed light weight gradient compression methods slow in practice, e.g., the ones proposed in* (*Suresh et al., 2016*)?" We agree that there are lots of gradient compression methods, which are computationally cheap. However, other important factors can affect the gradient compression efficiency in practice (taking the gradient compression method in (Suresh et al., 2016) as an example):

(i) After the binary sign rounding, extra encoding and decoding steps e.g. binary encoding are required to aggregate the quantized bits to bytes for attaining real communication speedup. That is optimizing the data structures to support low-communication for quantized gradients is necessary for any benefit to the surface, and also quite non trivial. (ii) For most gradient compression schemes, the encoded gradients are not compatible with all-reduce. Thus, all-gather has to be used instead. Unfortunately, in terms of comm. costs all-gather suffers a performance gap that increases with the number of nodes. (iii) In all-reduce, each worker receives a pre-aggregated gradient, making the cost of decompression independent to the number of workers. In all-gather, a worker receives the number of workers compressed gradients that need to be individually decompressed and aggregated. The time for decompression with all-gather therefore scales linearly

with the number of workers.

In fact we did run a test for the "*Stochastic binary quantization*" method in (Suresh et al., 2016) on ResNet-50+ImageNet over 16 EC2 `p3.2xlarge` nodes (per node batch size 32) as it is the computationally cheapest methods proposed in the paper. Though it is showed that conducting random rotation over the gradients can improve the compression error, we only care about the computational and communication efficiencies of the method in this particular experiment. Per epoch runtime results are shown in Figure 4.8.

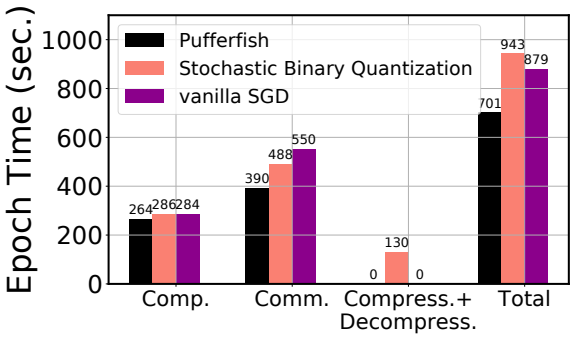


Figure 4.8: Breakdown per-epoch runtime comparison between PUFFERFISH, vanilla SGD, and stochastic binary quantization.

Note that in the "compress.+decompress." stage, stochastic binary quantization takes 12.1 ± 0.6 seconds for gradient compression and 118.4 ± 0.1 for gradient decompression. We observe that although the stochastic binary quantization is efficient in the compression stage, its gradient decompression cost is expensive. Moreover, all-gather is less efficient compared to all-reduce at the scale of 16 nodes.

4.5.7 The effectiveness of using SVD to find the low-rank factorization

In the vanilla warm-up training strategy proposed in PUFFERFISH, we decompose the network weights using SVD to find the initialization weights for the hybrid network. Though SVD is a computationally expensive method, PUFFERFISH only requires conducting the factorization over the network weights **once** during the

entire training process. We explicitly test the overhead incurred by conducting SVD over the model weights here. All the runtimes are measured over the `p3.2xlarge` instance of Amazon EC2 (equipped with Tesla V100 GPU). The results are shown in Figure 4.19. From the results, it can be observed that the run time on using SVD to factorize the partially trained vanilla full-rank network is quite fast, *e.g.,* on average it only costs 2.2972 seconds over the ResNet-50 trained over the ImageNet dataset.

Table 4.19: The time costs on conducting SVD over the partially trained vanilla full-rank network to find the initialization model for the hybrid network. The run time results are averaged from 5 independent trials.

Method	Time Cost (in sec.)
ResNet-50 on ImageNet	2.2972 ± 0.0519
WideResNet-50-2 on ImageNet	4.8700 ± 0.0859
VGG-19-BN on CIFAR-10	1.5198 ± 0.0113
ResNet-18 on CIFAR-10	1.3244 ± 0.0201
LSTM on WikeText-2	6.5791 ± 0.0445
Transformer on WMT16	5.4104 ± 0.0532

4.5.8 Details of data preprocessing

The CIFAR-10 dataset. In preprocessing the images in CIFAR-10 dataset, we follow the standard data augmentation and normalization process. For data augmentation, random cropping and horizontal random flipping are used. Each color channels are normalized with mean and standard deviation by $\mu_r = 0.491$, $\mu_g = 0.482$, $\mu_b = 0.447$, $\sigma_r = 0.247$, $\sigma_g = 0.244$, $\sigma_b = 0.262$. Each channel pixel is normalized by subtracting the mean value in this color channel and then divided by the standard deviation of this color channel.

The ImageNet dataset. For ImageNet, we follow the data augmentation process of (Goyal et al., 2017), *i.e.*, we use scale and aspect ratio data augmentation. The network input image is a 224×224 pixels, randomly cropped from an augmented image or its horizontal flip. The input image is normalized in the same way as we normalize the CIFAR-10 images using the following means and standard deviations: $\mu_r = 0.485, \mu_g = 0.456, \mu_b = 0.406$; $\sigma_r = 0.229, \sigma_g = 0.224, \sigma_b = 0.225$.

4.5.9 Detailed hyper-parameters used in our experiments

ResNet-50 and WideResNet-50-2 over the ImageNet dataset. For ResNet-50 and WideResNet-50-2 models, we follow the model training hyper-parameters reported in (Goyal et al., 2017). We train the model using the optimizer SGD with momentum value at 0.9 with batch size at 256. We also conduct ℓ_2 regularization over the model weights instead of the BatchNorm layers with the regularization coefficient 10^{-4}. The entire training process takes 90 epochs. For both of the ResNet-50 and WideResNet-50-2 models, we start from the learning rate at 0.1 and decay the learning rate by a factor of 0.1 at the 30-th, 60-th, and the 80-th epochs. For the vanilla warm-up training, we use warm-up epoch $E_{wu} = 10$. Note that at the 10-th epoch we switch from the vanilla ResNet-50/WideResNet-50-2 models to the hybrid architecture, but we still use the same learning rate, *i.e.*, 0.1 until the 30-th epoch. Additional to the previously proposed work, we adopt the *label smoothing* technique with probability 0.1. The model initialization method follows directly from the implementation of PyTorch example [7].

ResNet-18 and VGG-19-BN over the CIFAR-10 dataset. For ResNet-18 and VGG-19-BN models. We train the model using the optimizer SGD with momentum with momentum value at 0.9 with batch size at 128. The entire training takes 300 epochs. We also conduct ℓ_2 regularization over the model weights with the regularization coefficient 10^{-4}. For both of the ResNet-18 and VGG-19-BN models, we start from the learning rate at 0.1 and decay the learning rate by a factor of 0.1 at the 150-th,

[7] `https://github.com/pytorch/examples/tree/master/imagenet`

250-th epochs. For the vanilla warm-up training, we use warm-up epoch $\mathsf{E}_{wu} = 80$. Note that at the 80-th epoch we switch from the vanilla ResNet-18/VGG-19-BN models to the hybrid architecture, but we still use the same learning rate, *i.e.*, 0.1 until the 150-th epoch.

LSTM over the WikiText-2 dataset. For the LSTM model, we conduct training using the vanilla SGD optimizer with batch size at 20. We also conduct gradient norm clipping with norm bound at 0.25. The entire training takes 40 epochs. We start from the learning rate at 20 and decay the learning rate by a factor of 0.25 if the validation loss is not decreasing. For the vanilla warm-up training, we use warm-up epoch $\mathsf{E}_{wu} = 10$. Note that at the 10-th epoch we switch from the vanilla LSTM model to the hybrid architecture, we also decay the learning rate by a factor of 0.5. We also tie the word embedding and SoftMax weights (Press and Wolf, 2016).

The Transformer over the WMT16 dataset. For the Transformer model, we use the Adam optimizer with initial learning rate at 0.001, $\beta s = (0.9, 0.98)$, $\epsilon = 10^{-8}$ batch size at 256. We also conduct gradient norm clipping with norm bound at 0.25. The entire training takes 400 epochs. For the vanilla warm-up training, we use warm-up epoch $\mathsf{E}_{wu} = 10$. We enable label smoothing, weight sharing for the source and target word embedding, and weight sharing between target word embedding and the last dense layer.

4.5.10 Detailed information on the runtime mini-benchmark

In the experiment section, we discussed that in the reproducibility optimized setting, factorized networks achieve promising runtime speedup over the vanilla networks. However, sometimes users prefer faster runtime to reproducibility where the speed optimized setting is used (with `cudnn.benckmark` enabled and `cudnn.deterministic` disabled). We also study the runtime of the factorized network under the speed optimized setting. The results are shown in Table 4.20, from which we observe that the speedup of the factorized network is less promising

compared to the reproducibility optimized setting especially for the VGG-19-BN network. However, PUFFERFISH ResNet-18 still achieves 1.16× per-epoch speedup. We leave exploring the optimal model training speed of the factorized networks as the future work.

Table 4.20: The runtime mini-benckmark results of PUFFERFISH and vanilla VGG-19-BN and ResNet-18 networks training on the CIFAR-10 dataset, results averaged over 10 epochs. Experiment running on a single V100 GPU with batch size at 128; Over the optimized cuDNN implementation with `cudnn.benckmark` enabled and `cudnn.deterministic` disabled; Speedup calcuated based on the averaged per-epoch time.

Model Archs.	Epoch Time (sec.)	Speedup	MACs (G)
Vanilla VGG-19	8.27 ± 0.07	–	0.4
PUFFERFISH VGG-19	$\mathbf{8.16} \pm 0.12$	**1.01**×	**0.29**
Vanilla ResNet-18	11.15 ± 0.01	–	0.56
PUFFERFISH ResNet-18	$\mathbf{9.61} \pm 0.08$	**1.16**×	**0.22**

4.5.11 Time cost measurement on Amazon EC2

We use the `p3.2xlarge` instances for the distributed experiments, the bandwidth of the instance is "Up to 10 Gbps" as stated on the Amazon EC2 website, *i.e.*, `https://aws.amazon.com/ec2/instance-types/p3/`. For some tasks (especially for the ResNet-50 and WideResNet-50-2), we observe that the bandwidth of the `p3.2xlarge` instance decays sharply in the middle of the experiment. The time costs for ResNet-50 trained on the ImageNet dataset under our prototype `allreduce` distributed implementation are collected when there is no bandwidth decay, *e.g.*, under 10 Gbps. For the DDP time cost results, we run the experiments till the per-epoch time costs become stable, then measure the per-epoch time. For ResNet-18 trained on the CIFAR-10 dataset experiments under our prototype `allreduce` distributed implementation, we do not observe significant bandwidth decay for

the `p3.2xlarge` instances. All distributed experiments are conducted under the `us-west-2c` availability zone of EC2.

4.5.12 Additional experimental results

The ablation study on the accuracy mitigation strategy over CIFAR-10 and ImageNet. The ablation study results are shown in Table 4.21 for the LSTM trained on WikiText-2 task, Table 4.22 for ResNet-50 trained on ImageNet task, and Table 4.23 for VGG-19-BN trained over CIFAR-10. For the vanilla low-rank ResNet-50 trained on ImageNet, we do not deploy the label smoothing and the extra learning rate decay (with a factor 0.1) at the 80-th epoch.

Table 4.21: The effect of vanilla warm-up training on the low-rank LSTM trained over WikiText-2. Results are averaged across 3 independent trials with different random seeds.

Methods	Low-rank LSTM (wo. vanilla warm-up)	Low-rank LSTM (w. vanilla warm-up)
Train Ppl.	68.04 ± 2.98	$\mathbf{62.2} \pm 0.74$
Val. Ppl.	97.59 ± 0.69	$\mathbf{93.62} \pm 0.36$
Test Ppl.	92.04 ± 0.54	$\mathbf{88.72} \pm 0.24$

Table 4.22: The effect of vanilla warm-up training and hybrid network architectures of Pufferfish of the low-rank ResNet-50 trained over the ImageNet dataset

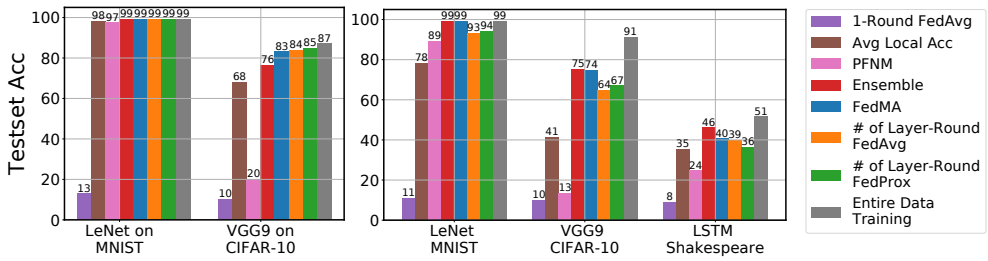
Model architectures	Test Acc. Top1	Test Acc. Top5
Low-rank ResNet-50	71.03%	90.26%
Hybrid ResNet-50 (wo. vanilla warm-up)	75.85%	92.96%
Hybrid ResNet-50 (w. vanilla warm-up)	**76.43**%	**93.10**%

Table 4.23: The effect of vanilla warm-up training and hybrid network architectures of Pufferfish of the low rank VGG-19-BN trained over the CIFAR-10 dataset. Results are averaged across 3 independent trials with different random seeds.

Model architectures	Test Loss	Test Accuracy
Low-rank VGG-19-BN	0.355 ± 0.012	$93.34 \pm 0.08\%$
Hybrid VGG-19-BN (wo. vanilla warm-up)	0.407 ± 0.008	$93.53 \pm 0.13\%$
Hybrid VGG-19-BN (w. vanilla warm-up)	0.375 ± 0.019	$\mathbf{93.89} \pm 0.14\%$

As stated before, one shortcoming of FedAvg is coordinate-wise averaging of weights may have drastic detrimental effects on the performance of the averaged model and adds significantly to the communication burden. This issue arises due to the permutation invariance of neural network (NN) parameters, *i.e.*, for any given NN, there are many variants of it that only differ in the ordering of parameters. Probabilistic Federated Neural Matching (PFNM) (Yurochkin et al., 2019b) addresses this problem by matching the neurons of client NNs before averaging them. PFNM further utilizes Bayesian nonparametric methods to adapt to global model size and to heterogeneity in the data. As a result, PFNM has better performance and communication efficiency than FedAvg. Unfortunately, the method only works with simple architectures (*e.g.*, fully connected feedforward networks).

In FedMA, we demonstrate how PFNM can be applied to CNNs and LSTMs, but we find that it only gives very minor improvements over weight averaging. To address this issue, we propose Federated Matched Averaging (FedMA), a new layers-wise FL algorithm for modern CNNs and LSTMs that appeal to Bayesian nonparametric methods to adapt to heterogeneity in the data. We show empirically that FedMA not only reduces the communication burden, but also outperforms



(a) Homogeneous (b) Heterogeneous

Figure 5.1: Comparison among various federated learning methods with limited number of communications on LeNet trained on MNIST; VGG-9 trained on CIFAR-10 dataset; LSTM trained on Shakespeare dataset over: (a) homogeneous data partition (b) heterogeneous data partition.

SotA FL algorithms.

5.1 Federated Matched Averaging of neural networks

In this section we will discuss permutation invariance classes of prominent neural network architectures and establish the appropriate notion of averaging in the parameter space of NNs. We will begin with the simplest case of a single hidden layer fully connected network, moving on to deep architectures and, finally, convolutional and recurrent architectures.

Permutation invariance of fully connected architectures. A basic fully connected (FC) NN can be formulated as $\hat{y} = \sigma(xW_1)W_2$ (without loss of generality, biases are omitted to simplify notation), where σ is the non-linearity (applied entry-wise). Expanding the preceding expression $\hat{y} = \sum_{i=1}^{L} W_{2,i\cdot}\sigma(\langle x, W_{1,\cdot i}\rangle)$, where $i\cdot$ and $\cdot i$ denote ith row and column correspondingly and L is the number of hidden units. Summation is a permutation invariant operation, hence for any $\{W_1, W_2\}$ there are $L!$ practically equivalent parametrizations if this basic NN. It is then more appropriate to write

$$\hat{y} = \sigma(xW_1\Pi)\Pi^T W_2, \text{ where } \Pi \text{ is any } L \times L \text{ permutation matrix.} \tag{5.1}$$

Recall that permutation matrix is an orthogonal matrix that acts on rows when applied on the left and on columns when applied on the right. Suppose $\{W_1, W_2\}$ are optimal weights, then weights obtained from training on two homogeneous datasets $X_j, X_{j'}$ are $\{W_1\Pi_j, \Pi_j^T W_2\}$ and $\{W_1\Pi_{j'}, \Pi_{j'}^T W_2\}$. It is now easy to see why naive averaging in the parameter space is not appropriate: with high probability $\Pi_j \neq \Pi_{j'}$ and $(W_1\Pi_j + W_1\Pi_{j'})/2 \neq W_1\Pi$ for any Π. To meaningfully average neural networks in the weight space we should first undo the permutation $(W_1\Pi_j\Pi_j^T + W_1\Pi_{j'}\Pi_{j'}^T)/2 = W_1$.

5.1.1 Matched averaging formulation

In this section we formulate practical notion of parameter averaging under the permutation invariance. Let w_{jl} be lth neuron learned on dataset j (*i.e.,* lth column of $W^{(1)}\Pi_j$ in the previous example), θ_i denote the ith neuron in the global model, and $c(\cdot,\cdot)$ be an appropriate similarity function between a pair of neurons. Solution to the following optimization problem are the required permutations:

$$\min_{\{\pi^j_{li}\}} \sum_{i=1}^{L} \sum_{j,l} \min_{\theta_i} \pi^j_{li} c(w_{jl}, \theta_i) \text{ s.t. } \sum_i \pi^j_{li} = 1 \; \forall \; j, l; \; \sum_l \pi^j_{li} = 1 \; \forall \; i, j. \tag{5.2}$$

Then $\Pi^T_{jli} = \pi^j_{li}$ and given weights $\{W_{j,1}, W_{j,2}\}_{j=1}^J$ provided by J clients, we compute the federated neural network weights $W_1 = \frac{1}{J}\sum_j W_{j,1}\Pi_j^T$ and $W_2 = \frac{1}{J}\sum_j \Pi_j W_{j,2}$. We refer to this approach as *matched averaging* due to relation of equation 5.2 to the maximum bipartite matching problem. We note that if $c(\cdot,\cdot)$ is squared Euclidean distance, we recover objective function similar to k-means clustering, however it has additional constraints on the "cluster assignments" π^j_{li} necessary to ensure that they form permutation matrices. In a special case where all local neural networks and the global model are assumed to have same number of hidden neurons, solving equation 5.2 is equivalent to finding a Wasserstein barycenter (Agueh and Carlier, 2011) of the empirical distributions over the weights of local neural networks. Concurrent work of (Singh and Jaggi, 2019) explores the Wasserstein barycenter variant of equation 5.2.

Solving matched averaging. Objective function in equation 5.2 can be optimized using an iterative procedure: applying the Hungarian matching algorithm (Kuhn, 1955) to find permutation $\{\pi^{j'}_{li}\}_{l,i}$ corresponding to dataset j', holding other permutations $\{\pi^j_{li}\}_{l,i,j\neq j'}$ fixed and iterating over the datasets. Important aspect of Federated Learning that we should consider here is the data heterogeneity. Every client will learn a collection of feature extractors, *i.e.,* neural network weights, representing their individual data modality. As a consequence, feature extractors learned across clients may overlap only *partially*. To account for this we allow the size of the

global model L to be an unknown variable satisfying $\max_j L_j \leqslant L \leqslant \sum_j L_j$ where L_j is the number of neurons learned from dataset j. That is, global model is at least as big as the largest of the local models and at most as big as the concatenation of all the local models. Next we show that matched averaging with adaptive global model size remains amendable to iterative Hungarian algorithm with a special cost.

At each iteration, given current estimates of $\{\pi^j_{li}\}_{l,i,j\neq j'}$, we find a corresponding global model $\{\theta_i = \arg\min_{\theta_i} \sum_{j\neq j',l} \pi^j_{li} c(w_{jl}, \theta_i)\}_{i=1}^{L}$ (this is typically a closed-form expression or a simple optimization sub-problem, *e.g.*, a mean if $c(\cdot,\cdot)$ is Euclidean) and then we will use Hungarian algorithm to match this global model to neurons $\{w_{j'l}\}_{l=1}^{L_{j'}}$ of the dataset j' to obtain a new global model with $L \leqslant L' \leqslant L + L_{j'}$ neurons. Due to data heterogeneity, local model j' may have neurons not present in the global model built from other local models, therefore we want to avoid "poor" matches by saying that if the optimal match has cost larger than some threshold value ϵ, instead of matching we create a new global neuron from the corresponding local one. We also want a modest size global model and therefore penalize its size with some increasing function $f(L')$. This intuition is formalized in the following *extended* maximum bipartite matching formulation:

$$\min_{\{\pi^{j'}_{li}\}_{l,i}} \sum_{i=1}^{L+L_{j'}} \sum_{j=1}^{L_{j'}} \pi^{j'}_{li} C^{j'}_{li} \text{ s.t. } \sum_i \pi^{j'}_{li} = 1 \;\forall\, l;\; \sum_l \pi^{j}_{li} \in \{0,1\} \;\forall\, i, \text{ where}$$
$$C^{j'}_{li} = \begin{cases} c(w_{j'l}, \theta_i), & i \leqslant L \\ \epsilon + f(i), & L < i \leqslant L + L_{j'}. \end{cases} \tag{5.3}$$

The size of the new global model is then $L' = \max\{i : \pi^{j'}_{li} = 1,\ l = 1, \ldots, L_{j'}\}$. We note some technical details: after the optimization is done, each corresponding Π_j^T is of size $L_j \times L$ and is not a permutation matrix in a classical sense when $L_j \neq L$. Its functionality is however similar: taking matrix product with a weight matrix $W_j^{(1)}\Pi_j^T$ implies permuting the weights to align with weights learned on the other datasets and padding with "dummy" neurons having zero weights (alternatively we can pad weights $W_j^{(1)}$ first and complete Π_j^T with missing rows to recover a

proper permutation matrix). This "dummy" neurons should also be discounted when taking average. Without loss of generality, in the subsequent presentation we will ignore these technicalities to simplify the notation.

To complete the matched averaging optimization procedure it remains to specify similarity $c(\cdot,\cdot)$, threshold ϵ and model size penalty $f(\cdot)$. (Yurochkin et al., 2019a,b,c) studied fusion, *i.e.,* aggregation, of model parameters in a range of applications. The most relevant to our setting is Probabilistic Federated Neural Matching (PFNM) (Yurochkin et al., 2019b). They arrived at a special case of equation 5.3 to compute maximum a posteriori estimate (MAP) of their Bayesian nonparametric model based on the Beta-Bernoulli process (BBP) (Thibaux and Jordan, 2007), where similarity $c(w_{jl}, \theta_i)$ is the corresponding posterior probability of jth client neuron l generated from a Gaussian with mean θ_i, and ϵ and $f(\cdot)$ are guided by the Indian Buffet Process prior (Ghahramani and Griffiths, 2005). Instead of making heuristic choices, this formulation provides a model-based specification of equation 5.3. We refer to a procedure for solving equation 5.2 with the setup from (Yurochkin et al., 2019b) as BBP-MAP. We note that their PFNM is only applicable to fully connected architectures limiting its practicality. Our *matched averaging* perspective allows to formulate averaging of widely used architectures such as CNNs and LSTMs as instances of equation 5.2 and utilize the BBP-MAP as a solver.

5.1.2 Permutation invariance of key architectures

Before moving onto the convolutional and recurrent architectures, we discuss permutation invariance in *deep* fully connected networks and corresponding matched averaging approach. We will utilize this as a building block for handling LSTMs and CNN architectures such as VGG (Simonyan and Zisserman, 2015) widely used in practice.

Permutation invariance of deep FCs. We extend equation 5.1 to recursively define deep FC network:

$$x_n = \sigma(x_{n-1}\Pi_{n-1}^T W_n \Pi_n), \tag{5.4}$$

where $n = 1, \ldots, N$ is the layer index, Π_0 is identity indicating non-ambiguity in the ordering of input features $x = x_0$ and Π_N is identity for the same in output classes. Conventionally $\sigma(\cdot)$ is any non-linearity except for $\hat{y} = x_N$ where it is the identity function (or softmax if we want probabilities instead of logits). When $N = 2$, we recover a single hidden layer variant from equation 5.1. To perform matched averaging of deep FCs obtained from J clients we need to find permutations for every layer of every client. Unfortunately, permutations within any consecutive pair of intermediate layers are coupled leading to a NP-hard combinatorial optimization problem. Instead we consider recursive (in layers) matched averaging formulation. Suppose we have $\{\Pi_{j,n-1}\}$, then plugging $\{\Pi_{j,n-1}^T W_{j,n}\}$ into equation 5.2 we find $\{\Pi_{j,n}\}$ and move onto next layer. The recursion base for this procedure is $\{\Pi_{j,0}\}$, which we know is an identity permutation for any j.

Permutation invariance of CNNs. The key observation in understanding permutation invariance of CNNs is that instead of neurons, channels define the invariance. To be more concrete, let $\text{Conv}(x, W)$ define convolutional operation on input x with weights $W \in \mathbb{R}^{C^{in} \times w \times h \times C^{out}}$, where C^{in}, C^{out} are the numbers of input/output channels and w, h are the width and height of the filters. Applying any permutation to the output dimension of the weights and then same permutation to the input channel dimension of the subsequent layer will not change the corresponding CNN's forward pass. Analogous to equation 5.4 we can write:

$$x_n = \sigma(\text{Conv}(x_{n-1}, \Pi_{n-1}^T W_n \Pi_n)). \tag{5.5}$$

Note that this formulation permits pooling operations as those act within channels. To apply matched averaging for the nth CNN layer we form inputs to equation 5.2 as $\{w_{jl} \in \mathbb{R}^D\}_{l=1}^{C_n^{out}}$, $j = 1, \ldots, J$, where D is the flattened $C_n^{in} \times w \times h$ dimension of $\Pi_{j,n-1}^T W_{j,n}$. This result can be alternatively derived taking the IM2COL perspective. Similar to FCs, we can recursively perform matched averaging on deep CNNs. The immediate consequence of our result is the extension of PFNM (Yurochkin et al., 2019b) to CNNs. Empirically, see Figure 5.1, we found that this extension performs

well on MNIST with a simpler CNN architecture such as LeNet (LeCun et al., 1998) (4 layers) and significantly outperforms coordinate-wise weight averaging (1 round FedAvg). However, it breaks down for more complex architecture, *e.g.*, VGG-9 (Simonyan and Zisserman, 2015) (9 layers), needed to obtain good quality prediction on a more challenging CIFAR-10.

Permutation invariance of LSTMs. Permutation invariance in the recurrent architectures is associated with the ordering of the hidden states. At a first glance it appears similar to fully connected architecture, however the important difference is associated with the permutation invariance of the hidden-to-hidden weights $H \in \mathbb{R}^{L \times L}$, where L is the number of hidden states. In particular, permutation of the hidden states affects *both* rows and columns of H. Consider a basic RNN $h_t = \sigma(h_{t-1}H + x_t W)$, where W are the input-to-hidden weights. To account for the permutation invariance of the hidden states, we notice that dimensions of h_t should be permuted in the same way for any t, hence

$$h_t = \sigma(h_{t-1}\Pi^T H \Pi + x_t W \Pi). \tag{5.6}$$

To match RNNs, the basic sub-problem is to align hidden-to-hidden weights of two clients with Euclidean similarity, which requires minimizing $\|\Pi^T H_j \Pi - H_{j'}\|_2^2$ over permutations Π. This is a quadratic assignment problem known to be NP-hard (Loiola et al., 2007). Fortunately, the same permutation appears in an already familiar context of input-to-hidden matching of $W\Pi$. Our matched averaging RNN solution is to utilize equation 5.2 plugging-in input-to-hidden weights $\{W_j\}$ to find $\{\Pi_j\}$. Then federated hidden-to-hidden weights are computed as $H = \frac{1}{J}\sum_j \Pi_j H_h \Pi_j^T$ and input-to-hidden weights are computed as before. We note that Gromov-Wasserstein distance (Gromov, 2007) from the optimal transport literature corresponds to a similar quadratic assignment problem. It may be possible to incorporate hidden-to-hidden weights H into the matching algorithm by exploring connections to approximate algorithms for computing Gromov-Wasserstein barycenter (Peyré et al., 2016). We leave this possibility for future work.

To finalize matched averaging of LSTMs, we discuss several specifics of the architecture. LSTMs have multiple cell states, each having its individual hidden-to-hidden and input-to-hidden weights. In out matched averaging we stack input-to-hidden weights into $SD \times L$ weight matrix (S is the number of cell states; D is input dimension and L is the number of hidden states) when computing the permutation matrices and then average all weights as described previously. LSTMs also often have an embedding layer, which we handle like a fully connected layer. Finally, we process deep LSTMs in the recursive manner similar to deep FCs.

5.1.3 Federated Matched Averaging (FedMA) algorithm

Defining the permutation invariance classes of CNNs and LSTMs allows us to extend PFNM (Yurochkin et al., 2019b) to these architectures, however our empirical study in Figure 5.1 demonstrates that such extension fails on deep architectures necessary to solve more complex tasks. Our results suggest that recursive handling of layers with matched averaging may entail poor overall solution. To alleviate this problem and utilize the strength of matched averaging on "shallow" architectures, we propose the following layer-wise matching scheme. First, data center gathers *only* the weights of the first layers from the clients and performs one-layer matching described previously to obtain the first layer weights of the federated model. Data center then broadcasts these weights to the clients, which proceed to train all *consecutive* layers on their datasets, keeping the matched federated layers *frozen*. This procedure is then repeated up to the last layer for which we conduct a weighted averaging based on the class proportions of data points per client. We summarize our Federated Matched Averaging (FedMA) in Algorithm 3. The FedMA approach requires communication rounds equal to the number of layers in a network. In Figure 5.1 we show that with layer-wise matching FedMA performs well on the deeper VGG-9 CNN as well as LSTMs. In the more challenging heterogeneous setting, FedMA outperforms FedAvg, FedProx trained with same number of communication rounds (4 for LeNet and LSTM and 9 for VGG-9) and other baselines, *i.e.*, client individual CNNs and their ensemble.

Algorithm 3 Federated Matched Averaging (FedMA)

Input : local weights of N-layer architectures $\{W_{j,1}, \ldots, W_{j,N}\}_{j=1}^{J}$ from J clients
Output: global weights $\{W_1, \ldots, W_N\}$
$n = 1$;
while $n \leqslant N$ **do**
  if $n < N$ **then**
    $\{\Pi_j\}_{j=1}^{J} = \text{BBP-MAP}\big(\{W_{j,n}\}_{j=1}^{J}\big)$; `// call BBP-MAP to solve Eq. 5.2`
    $W_n = \frac{1}{J} \sum_j W_{j,n} \Pi_j^T$;
  else
    $W_n = \sum_{k=1}^{K} \sum_j p_{jk} W_{jl,n}$ where p_k is fraction of data points with label k on worker j;
  end
  for $j \in \{1, \ldots, J\}$ **do**
    $W_{j,n+1} \leftarrow \Pi_j W_{j,n+1}$; `// permutate the next-layer weights`
    Train $\{W_{j,n+1}, \ldots, W_{j,L}\}$ with W_n frozen;
  end
  $n = n + 1$;
end

FedMA with communication. We have shown that in the heterogeneous data scenario FedMA outperforms other federated learning approaches, however it still lags in performance behind the entire data training. Of course the entire data training is not possible under the federated learning constraints, but it serves as performance upper bound we should strive to achieve. To further improve the performance of our method, we propose *FedMA with communication*, where local clients receive the matched global model at the beginning of a new round and reconstruct their local models with the size equal to the original local models (*e.g.*, size of a VGG-9) based on the matching results of the previous round. This procedure allows to keep the size of the global model small in contrast to a naive strategy of utilizing full matched global model as a starting point across clients on every round.

5.2 Experiments

We present an empirical study of FedMA with communication and compare it with state-of-the-art methods, *i.e.,* FedAvg (McMahan et al., 2017a) and FedProx (Sahu et al., 2018); analyze the performance under the growing number of clients and visualize the matching behavior of FedMA to study its interpretability. Our experimental studies are conducted over three real world datasets. Summary information about the datasets and associated models can be found in supplement Table 8.1.

Experimental setup. We implemented FedMA and the considered baseline methods in PyTorch (Paszke et al., 2017). We deploy our empirical study under a simulated federated learning environment where we treat one centralized node in the distributed cluster as the data center and the other nodes as local clients. All nodes in our experiments are deployed on *p3.2xlarge* instances on Amazon EC2. We assume the data center samples all the clients to join the training process for every communication round for simplicity.

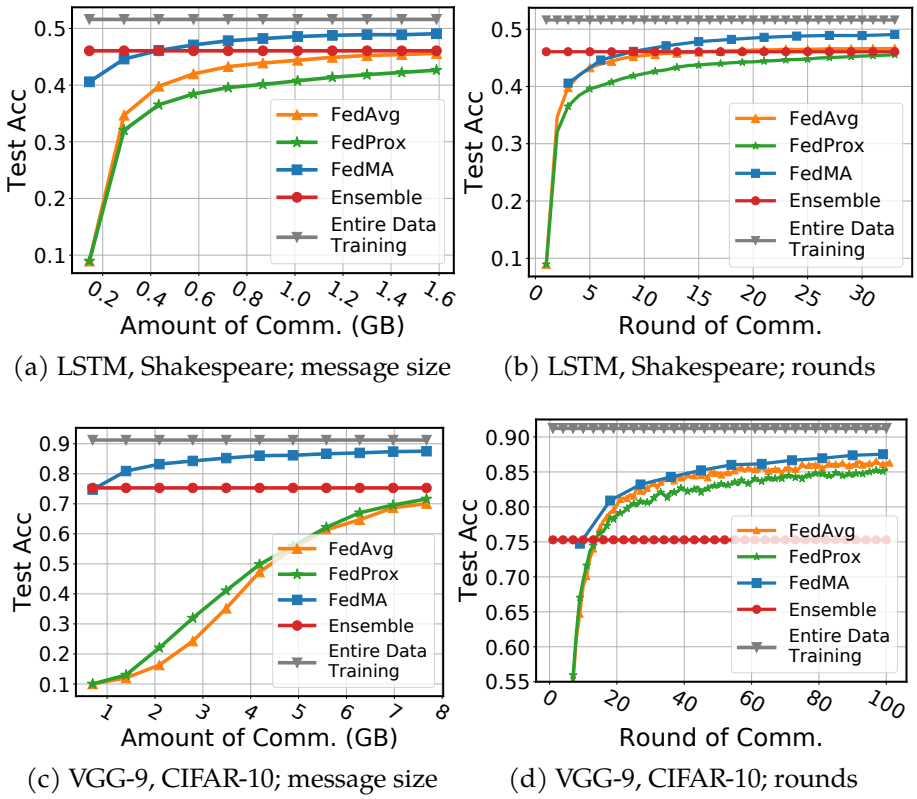
For the CIFAR-10 dataset, we use data augmentation (random crops, and flips) and normalize each individual image (details provided in the Supplement). We note that we ignore all batch normalization (Ioffe and Szegedy, 2015) layers in the VGG architecture and leave it for future work.

For CIFAR-10, we considered two data partition strategies to simulate federated learning scenario: (i) homogeneous partition where each local client has approximately equal proportion of each of the classes; (ii) heterogeneous partition for which number of data points and class proportions are unbalanced. We simulated a heterogeneous partition into J clients by sampling $\mathbf{p}_k \sim \text{Dir}_J(0.5)$ and allocating a $\mathbf{p}_{k,j}$ proportion of the training instances of class k to local client j. We use the original test set in CIFAR-10 as our global test set for comparing performance of all methods. For the Shakespeare dataset, we treat each speaking role as a client (Caldas et al., 2018) resulting in a natural heterogeneous partition. We preprocess the Shakespeare dataset by filtering out the clients with less than 10k datapoints and sampling a random subset of $J = 66$ clients. We allocate 80% of the data for

training and amalgamate the remaining data into a global test set.

Communication efficiency and convergence rate. In this experiment we study performance of FedMA with communication. Our goal is to compare our method to FedAvg and FedProx in terms of the total message size exchanged between data center and clients (in Gigabytes) and the number of communication rounds (recall that completing one FedMA pass requires number of rounds equal to the number of layers in the local models) needed for the global model to achieve good performance on the test data. We also compare to the performance of an ensemble method. We evaluate all methods under the heterogeneous federated learning scenario on CIFAR-10 with $J = 16$ clients with VGG-9 local models and on Shakespeare dataset with $J = 66$ clients with 1-layer LSTM network. We fix the total rounds of communication allowed for FedMA, FedAvg, and FedProx *i.e.,* 11 rounds for FedMA and 99/33 rounds for FedAvg and FedProx for the VGG-9/LSTM experiments respectively. We notice that number of local training epochs is a common parameter shared by the three considered methods, we thus tune this parameter (denoted E; comprehensive analysis will be presented in the next experiment) and report the convergence rate under E that yields the best final model accuracy over the global test set. We also notice that there is another hyper-parameter in FedProx *i.e.,* the coefficient μ associated with the proxy term, we also tune the parameter using grid search and report the best μ we found (0.001) for both VGG-9 and LSTM experiments. FedMA outperforms FedAvg and FedProx in all scenarios (Figure 5.2) with its advantage especially pronounced when we evaluate convergence as a function of the message size in Figures 5.2a and 5.2c. Final performance of all trained models is summarized in Tables 5.1 and 5.2.

Effect of local training epochs. As studied in previous work (McMahan et al., 2017a; Caldas et al., 2018; Sahu et al., 2018), the number of local training epochs E can affect the performance of FedAvg and sometimes lead to divergence. We conduct an experimental study on the effect of E over FedAvg, FedProx, and FedMA on VGG-9 trained on CIFAR-10 under heterogeneous setup. The candidate local



(a) LSTM, Shakespeare; message size (b) LSTM, Shakespeare; rounds

(c) VGG-9, CIFAR-10; message size (d) VGG-9, CIFAR-10; rounds

Figure 5.2: Convergence rates of various methods in two federated learning scenarios: training VGG-9 on CIFAR-10 with $J = 16$ clients and training LSTM on Shakespeare dataset with $J = 66$ clients.

epochs we consider are $E \in \{10, 20, 50, 70, 100, 150\}$. For each of the candidate E, we run FedMA for 6 rounds while FedAvg and FedProx for 54 rounds and report the final accuracy that each methods achieves. The result is shown in Figure 5.3. We observe that training longer benefits FedMA, supporting our assumption that FedMA performs best on local models with higher quality. For FedAvg, longer local training leads to deterioration of the final accuracy, which matches the observations made in the previous literature (McMahan et al., 2017a; Caldas et al., 2018; Sahu et al., 2018). FedProx only partially alleviates this problem. The result of this experiment suggests that FedMA is the only method that local clients can use to train their model as long as they want.

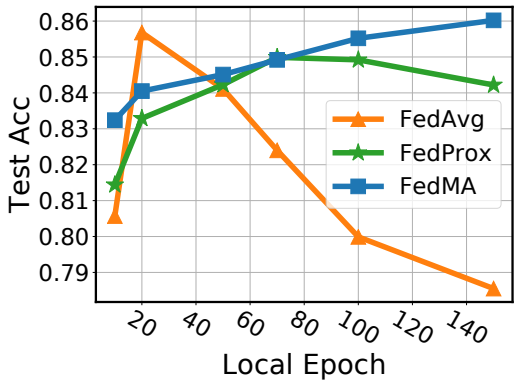


Figure 5.3: The effect of number of local training epochs on various methods.

Table 5.1: Trained models summary for VGG-9 trained on CIFAR-10 as shown in Figure 5.2

Method	FedAvg	FedProx	Ensemble	FedMA
Final Accuracy(%)	86.29	85.32	75.29	**87.53**
Best local epoch(E)	20	20	N/A	150
Model growth rate	1×	1×	16×	1.11×

Handling data bias. Real world data often exhibit multimodality within each class, *e.g.*, geo-diversity. It has been shown that an observable amerocentric and eurocentric bias is present in the widely used ImageNet dataset (Russakovsky et al., 2015; Shankar et al., 2017). Classifiers trained on such data "learn" these biases and perform poorly on the under-represented domains (modalities) since correlation between the corresponding dominating domain and class can prevent the classifier from learning meaningful relations between features and classes. For example, classifier trained on amerocentric and eurocentric data may learn to associate white color dress with a "bride" class, therefore underperforming on the wedding images taken in countries where wedding traditions are different (Doshi, 2018).

The data bias scenario is an important aspect of federated learning, however it

Table 5.2: Trained models summary for LSTM trained on Shakespeare as shown in Figure 5.2

Method	FedAvg	FedProx	Ensemble	FedMA
Final Accuracy(%)	46.63	45.83	46.06	**49.07**
Best local epoch(E)	2	5	N/A	5
Model growth rate	1×	1×	66×	1.06×

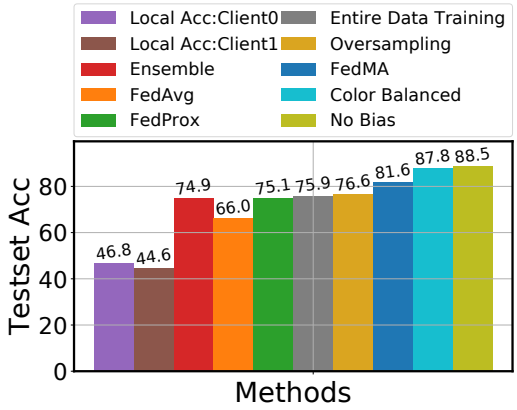


Figure 5.4: Performance on skewed CIFAR-10 dataset.

received little to no attention in the prior federated learning works. In this study we argue that FedMA can handle this type of problem. If we view each domain, *e.g.*, geographic region, as one client, local models will not be affected by the aggregate data biases and learn meaningful relations between features and classes. FedMA can then be used to learn a good global model without biases. We have already demonstrated strong performance of FedMA on federated learning problems with heterogeneous data across clients and this scenario is very similar. To verify this conjecture we conduct the following experiment.

We simulate the skewed domain problem with CIFAR-10 dataset by randomly selecting 5 classes and making 95% training images in those classes to be grayscale. For the remaining 5 we turn only 5% of the corresponding images into grayscale. By

doing so, we create 5 grayscale images dominated classes and 5 colored images dominated classes. In the test set, there is half grayscale and half colored images for each class. We anticipate entire data training to pick up the uninformative correlations between greyscale and certain classes, leading to poor test performance without these correlations. In Figure 5.4 we see that entire data training performs poorly in comparison to the regular (*i.e.*, No Bias) training and testing on CIFAR-10 dataset without any grayscaling. This experiment was motivated by Olga (Russakovsky)'s talk at ICML 2019.

Next we compare the federated learning based approaches. We split the images from color dominated classes and grayscale dominated classes into 2 clients. We then conduct FedMA with communication, FedAvg, and FedProx with these 2 clients. FedMA noticeably outperforms the entire data training and other federated learning approach as shown in Figure 5.4. This result suggests that FedMA may be of interest beyond learning under the federated learning constraints, where entire data training is the performance *upper bound*, but also to eliminate data biases and *outperform* entire data training. We consider two additional approaches to

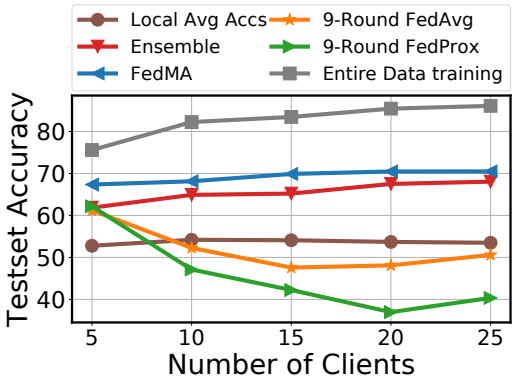
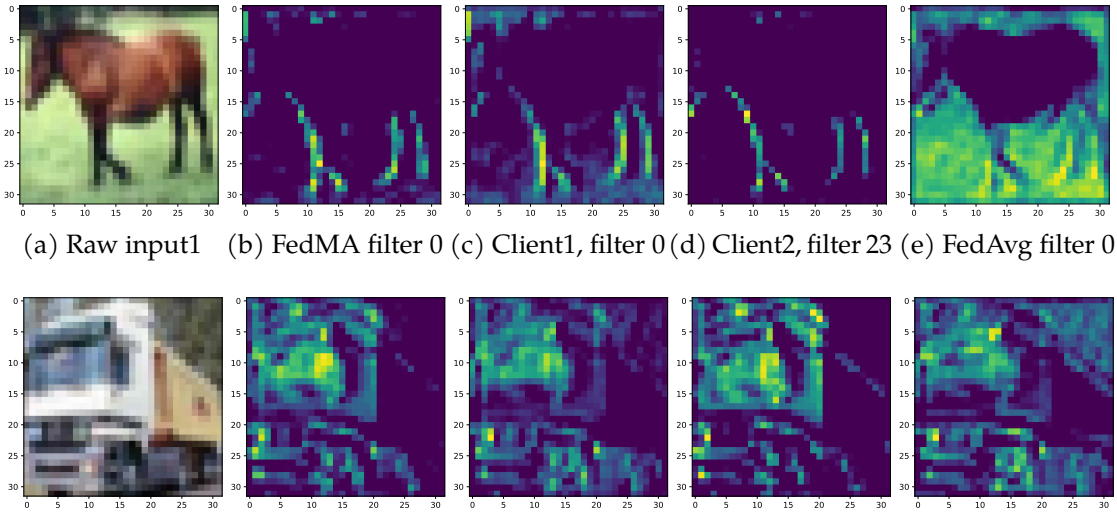


Figure 5.5: Data efficiency under the increasing number of clients.

eliminate data bias without the federated learning constraints. One way to alleviate data bias is to selectively collect more data to debias the dataset. In the context of our experiment, this means getting more colored images for grayscale dominated classes and more grayscale images for color dominated classes. We simulate this



(a) Raw input1 (b) FedMA filter 0 (c) Client1, filter 0 (d) Client2, filter 23 (e) FedAvg filter 0

(f) Raw input2 (g) FedMA filter 0 (h) Client 1, filter0 (i) Client 2, filter23 (j) FedAvg filter0

Figure 5.6: Representations generated by the first convolution layers of locally trained models, FedMA global model and the FedAvg global model.

scenario by simply doing a full data training where each class in both train and test images has equal amount of grayscale and color images. This procedure, Color Balanced, performs well, but selective collection of new data in practice may be expensive or even not possible. Instead of collecting new data, one may consider oversampling from the available data to debias. In Oversampling, we sample the underrepresented domain (via sampling with replacement) to make the proportion of color and grayscale images to be equal for each class (oversampled images are also passed through the data augmentation pipeline, *e.g.*, random flipping and cropping, to further enforce the data diversity). Such procedure may be prone to overfitting the oversampled images and we see that this approach only provides marginal improvement of the model accuracy compared to centralized training over the skewed dataset and performs noticeably worse than FedMA.

Data efficiency. It is known that deep learning models perform better when more training data is available. However, under the federated learning constraints, data

efficiency has not been studied to the best of our knowledge. The challenge here is that when new clients join the federated system, they each bring their own version of the data distribution, which, if not handled properly, may deteriorate the performance despite the growing data size across the clients. To simulate this scenario we first partition the entire training CIFAR-10 dataset into 5 homogeneous pieces. We then partition each homogeneous data piece further into 5 sub-pieces heterogeneously. Using this strategy, we partition the CIFAR-10 training set into 25 heterogeneous small sub-datasets containing approximately 2k points each. We conduct a 5-step experimental study: starting from a randomly selected homogeneous piece consisting of 5 associated heterogeneous sub-pieces, we simulate a 5-client federated learning heterogeneous problem. For each consecutive step, we add one of the remaining homogeneous data pieces consisting of 5 new clients with heterogeneous sub-datasets. Results are presented in Figure 5.5. Performance of FedMA (with a single pass) improves when new clients are added to the federated learning system, while FedAvg with 9 communication rounds deteriorates.

Interpretability. One of the strengths of FedMA is that it utilizes communication rounds more efficiently than FedAvg. Instead of directly averaging weights element-wise, FedMA identifies matching groups of convolutional filters and then averages them into the global convolutional filters. It's natural to ask *"How does the matched filters look like?"*. In Figure 5.6 we visualize the representations generated by a pair of matched local filters, aggregated global filter, and the filter returned by the FedAvg method over the same input image. Matched filters and the global filter found with FedMA are extracting the same feature of the input image, *i.e.,* filter 0 of client 1 and filter 23 of client 2 are extracting the position of the legs of the horse, and the corresponding matched global filter 0 does the same. For the FedAvg, global filter 0 is the average of filter 0 of client 1 and filter 0 of client 2, which clearly tampers the leg extraction functionality of filter 0 of client 1.

5.3 Discussion

We presented Federated Matched Averaging (FedMA), a layer-wise federated learning algorithm designed for modern CNNs and LSTMs architectures that accounts for permutation invariance of the neurons and permits global model size adaptation. Our method significantly outperforms prior federated learning algorithms in terms of its convergence when measured by the size of messages exchanged between server and the clients during training. We demonstrated that FedMA can efficiently utilize well-trained local modals, a property desired in many federated learning applications, but lacking in the prior approaches. We have also presented an example where FedMA can help to resolve some of the data biases and outperform aggregate data training.

In the future work we want to extend FedMA to improve federated learning of LSTMs using approximate quadratic assignment solutions from the optimal transport literature, and enable additional deep learning building blocks, *e.g.,* residual connections and batch normalization layers. We also believe it is important to explore fault tolerance of FedMA and study its performance on the larger datasets, particularly ones with biases preventing efficient training even when the data can be aggregated, *e.g.,* Inclusive Images (Doshi, 2018).

5.4 Additional results of FedMA

5.4.1 Summary of the datasets used in the experiments

The details of the datasets and hyper-parameters used in our experiments are summarized in Table 5.3. In conducting the “freezing and retraining" process of FedMA, we notice when retraining the last FC layer while keeping all previous layers frozen, the initial learning rate we use for SGD doesn’t lead to a good convergence (this is only for the VGG-9 architecture). To fix this issue, we divide the initial learning rate by 10 *i.e.,* using 10^{-4} for the last FC layer retraining and allow the clients to retrain for 3 times more epochs. We also switch off the ℓ_2 weight decay

during the “freezing and retraining" process of FedMA except for the last FC layer where we use a ℓ_2 weight decay of 10^{-4}. For language task, we observe SGD with a constant learning rate works well for all considered methods.

In our experiments, we use FedAvg and FedProx variants without the shared initialization since those would likely be more realistic when trying to aggregate locally pre-trained models. And FedMA still performs well in practical scenarios where local clients won't be able to share the random initialization.

Table 5.3: The datasets used and their associated learning models and hyper-parameters.

Method	MNIST	CIFAR-10	Shakespeare
# Data points	60,000	50,000	1,017,981
Model	LeNet	VGG-9	LSTM
# Classes	10	10	80
# Parameters	431k	3,491k	293k
Optimizer	SGD		SGD
Hyper-params.	Init lr: 0.01, 0.001 (last layer)		lr: 0.8(const)
	momentum: 0.9, ℓ_2 weight decay: 10^{-4}		

5.4.2 Details of the model architectures and hyper-parameters

The details of the model architectures we used in the experiments are summarized in this section. Specifically, details of the VGG-9 model architecture we used can be found in Table 5.4 and details of the 1-layer LSTM model used in our experimental study can be found in Table 5.5.

5.4.3 Data augmentation and normalization details

In preprocessing the images in CIFAR-10 dataset, we follow the standard data augmentation and normalization process. For data augmentation, random cropping and horizontal random flipping are used. Each color channels are normalized with mean and standard deviation by $\mu_r = 0.491372549$, $\mu_g = 0.482352941$, $\mu_b = 0.446666667$, $\sigma_r = 0.247058824$, $\sigma_g = 0.243529412$, $\sigma_b = 0.261568627$. Each channel pixel is normalized by subtracting the mean value in this color channel and then divided by the standard deviation of this color channel.

5.4.4 Extra experimental details

Shapes of final global model. Here we report the shapes of final global VGG and LSTM models returned by FedMA with communication.

Hyper-parameters for BBP-MAP. We follow FPNM (Yurochkin et al., 2019b) to choose the hyper-parameters of BBP-MAP, which controls the choices of θ_i, ϵ, and the $f(\cdot)$ as discussed in Section 5.1. More specifically, there are three parameters to choose *i.e.,* 1) σ_0^2, the prior variance of weights of the global neural network; 2) γ_0, which controls discovery of new hidden states. Increasing γ_0 leads to a larger final global model; 3) σ^2 is the variance of the local neural network weights around corresponding global network weights. We empirically analyze the different choices of the three hyper-parameters and find the choice of $\gamma_0 = 7, \sigma_0^2 = 1, \sigma^2 = 1$ for VGG-9 on CIFAR-10 dataset and $\gamma_0 = 10^{-3}, \sigma_0^2 = 1, \sigma^2 = 1$ for LSTM on Shakespeare dataset lead to good performance in our experimental studies.

5.4.5 Practical considerations

Following from the discussion in PFNM, here we briefly discuss the time complexity of FedMA. For simplicity, we focus on a single-layer matching and assume all participating clients train the same model architecture. The complexity for matching the entire model follows trivially from this discussion. The worst case complexity

is achieved when no hidden states are matched and is equal to $\mathcal{O}(D \cdot (JL)^2)$ for building the cost matrix and $\mathcal{O}((JL)^3)$ for running the Hungarian algorithm where the definitions of D, J, and L follow the discussion in Section 5.1. The best complexity per layer is (achieved when all hidden states are matched) $\mathcal{O}(D \cdot L^2 + L^3)$. Practically, when the number of participating clients *i.e.,* J is large and each client trains a big model, the speed of our algorithm can be relatively slow.

To seed up the Hungarian algorithm. Although there isn't any algorithm that achieves lower complexity, better implementation improves the constant significantly. In our experiments, we used an implementation based on shortest path augmentation *i.e.,* `lapsolver` [1]. Empirically, we observed that this implementation of the Hungarian algorithm leads to orders of magnitude speed ups over the vanilla implementation.

5.4.6 Hyper-parameters for the handling data bias experiments

In conducting the "handling data bias" experiments. We re-tune the local epoch E for both FedAvg and FedProx. The considered candidates of E are $\{5, 10, 20, 30\}$. We observe that a relatively large choice of E can easily lead to poor convergence of FedAvg. While FedProx tolerates larger choices of E better, a smaller E can always lead to good convergence. We use $E = 5$ for both FedAvg and FedProx in our experiments. For FedMA, we choose $E = 50$ since it leads to a good convergence. For the "oversampling" baseline, we found that using SGD to train VGG-9 over oversampled dataset doesn't lead to a good convergence. Moreover, when using constant learning rate, SGD can lead to model divergence. Thus we use AMSGrad (Reddi et al., 2018) method for the "oversampling" baseline and train for 800 epochs. To make the comparison fair, we use AMSGrad for all other centralized baselines to get the reported results in our experiments. Most of them converges when training for 200 epochs. We also test the performance of the "Entire Data Training", "Color Balanced", and "No Bias" baselines over SGD. We use learning rate 0.01 and ℓ_2 weight decay at 10^{-4} and train for 200 epochs for those three baselines. It seems

[1] `https://github.com/cheind/py-lapsolver`

the “Entire Data Training" and "No Bias" baselines converges to a slightly better accuracy, *i.e.,* 78.71% and 91.23% respectively (compared to 75.91% and 88.5% for AMSGrad). But the “Color Balanced" doesn’t seem to converge better accuracy (we get 87.79% accuracy for SGD and 87.81% for AMSGrad).

Table 5.4: Detailed information of the VGG-9 architecture used in our experiments, all non-linear activation function in this architecture is ReLU; the shapes for convolution layers follows (C_{in}, C_{out}, c, c)

Parameter	Shape	Layer hyper-parameter
layer1.conv1.weight	$3 \times 32 \times 3 \times 3$	stride:1;padding:1
layer1.conv1.bias	32	N/A
layer2.conv2.weight	$32 \times 64 \times 3 \times 3$	stride:1;padding:1
layer2.conv2.bias	64	N/A
pooling.max	N/A	kernel size:2;stride:2
layer3.conv3.weight	$64 \times 128 \times 3 \times 3$	stride:1;padding:1
layer3.conv3.bias	128	N/A
layer4.conv4.weight	$128 \times 128 \times 3 \times 3$	stride:1;padding:1
layer4.conv4.bias	128	N/A
pooling.max	N/A	kernel size:2;stride:2
dropout	N/A	$p = 5\%$
layer5.conv5.weight	$128 \times 256 \times 3 \times 3$	stride:1;padding:1
layer5.conv5.bias	256	N/A
layer6.conv6.weight	$256 \times 256 \times 3 \times 3$	stride:1;padding:1
layer6.conv6.bias	256	N/A
pooling.max	N/A	kernel size:2;stride:2
dropout	N/A	$p = 10\%$
layer7.fc7.weight	4096×512	N/A
layer7.fc7.bias	512	N/A
layer8.fc8.weight	512×512	N/A
layer8.fc8.bias	512	N/A
dropout	N/A	$p = 10\%$
layer9.fc9.weight	512×10	N/A
layer9.fc9.bias	10	N/A

Table 5.5: Detailed information of the LSTM architecture used in our experiments.

Parameter	Shape
encoder.weight	80×8
lstm.weight.ih.l0	1024×8
lstm.weight.hh.l0	1024×256
lstm.bias.ih.l0	1024
lstm.bias.hh.l0	1024
decoder.weight	80×256
decoder.bias	80

Table 5.6: Detailed information of the final global LSTM model trained using FedMA in our experiment.

Parameter	Shape	Growth rate (#global / #original params)
encoder.weight	80×21	$2.63 \times (1,680/640)$
lstm.weight.ih.l0	1028×21	$2.64 \times (21,588/8,192)$
lstm.weight.hh.l0	1028×257	$1.01 \times (264,196/262,144)$
lstm.bias.ih.l0	1028	$1.004 \times (1,028/1,024)$
lstm.bias.hh.l0	1028	$1.004 \times (1,028/1,024)$
decoder.weight	80×257	$1.004 \times (20,560/20,480)$
decoder.bias	80	$1\times$
Total Num. of Params.	$310,160$	$1.06\times$ $(310,160/293,584)$

Table 5.7: Detailed information of the final global VGG-9 model trained by FedMA; the shapes for convolution layers follows (C_{in}, C_{out}, c, c)

Parameter	Shape	Growth rate (#global / #original params)
layer1.conv1.weight	$3 \times 47 \times 3 \times 3$	$1.47 \times (1,269/864)$
layer1.conv1.bias	47	$1.47 \times (47/32)$
layer2.conv2.weight	$47 \times 79 \times 3 \times 3$	$1.81 \times (33,417/18,432)$
layer2.conv2.bias	79	$1.23 \times (79/64)$
layer3.conv3.weight	$79 \times 143 \times 3 \times 3$	$1.38 \times (101,673/73,728)$
layer3.conv3.bias	143	$1.12 \times (143/128)$
layer4.conv4.weight	$143 \times 143 \times 3 \times 3$	$1.24 \times (184,041/147,456)$
layer4.conv4.bias	143	$1.12 \times (143/128)$
layer5.conv5.weight	$143 \times 271 \times 3 \times 3$	$1.18 \times (348,777/294,912)$
layer5.conv5.bias	271	$1.06 \times (271/256)$
layer6.conv6.weight	$271 \times 271 \times 3 \times 3$	$1.12 \times (660,969/589,824)$
layer6.conv6.bias	271	$1.06 \times (271/256)$
layer7.fc7.weight	4336×527	$1.09 \times (2,285,072/2,097,152)$
layer7.fc7.bias	527	$1.02 \times (527/512)$
layer8.fc8.weight	527×527	$1.05\times, (277,729/262,144)$
layer8.fc8.bias	527	$1.02 \times (527/512)$
layer9.fc9.weight	527×10	$1.02 \times (5,270/5,120)$
layer9.fc9.bias	10	$1\times$
Total Num. of Params.	$3,900,235$	$1.11\times$ $(3,900,235/3,491,530)$

Part III

Robust Distributed ML

6 DRACO: BYZANTINE-RESILIENT DISTRIBUTED TRAINING VIA REDUNDANT GRADIENTS

Distributed model training is vulnerable to byzantine system failures and adversarial compute nodes, *i.e.*, nodes that use malicious updates to corrupt the global model stored at a parameter server (PS). To guarantee some form of robustness, recent work suggests using variants of the geometric median as an aggregation rule, in place of gradient averaging. Unfortunately, median-based rules can incur a prohibitive computational overhead in large-scale settings, and their convergence guarantees often require strong assumptions. In this chapter, we present DRACO, a scalable framework for robust distributed training that uses ideas from coding theory. In DRACO, each compute node evaluates redundant gradients that are used by the parameter server to eliminate the effects of adversarial updates. DRACO comes with problem-independent robustness guarantees, and the model that it trains is identical to the one trained in the adversary-free setup. We provide extensive experiments on real datasets and distributed setups across a variety of large-scale models, where we show that DRACO is several times, to orders of magnitude faster than median-based approaches.

6.1 Preliminaries

Notation. In the following, we denote matrices and vectors in bold, and scalars and functions in standard script. We let $\mathbf{1}_m$ denote the $m \times 1$ all ones vector, while $\mathbf{1}_{n\times m}$ denotes the all ones $n \times m$ matrix. We define $\mathbf{0}_m, \mathbf{0}_{n\times m}$ analogously. Given a matrix $\mathbf{A} \in \mathbb{R}^{n\times m}$, we let $\mathbf{A}_{i,j}$ denote its entry at location (i, j), $\mathbf{A}_{i,\cdot} \in \mathbb{R}^{1\times m}$ denote its ith row, and $\mathbf{A}_{\cdot,j} \in \mathbb{R}^{n\times 1}$ denote its jth column. Given $S \subseteq \{1, \ldots, n\}$, $T \subseteq \{1, \ldots, m\}$, we let $\mathbf{A}_{S,T}$ denote the submatrix of $\mathbf{A}$ where we keep rows indexed by S and columns indexed by T. Given matrices $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n\times m}$, their *Hadamard product*, denoted $\mathbf{A} \odot \mathbf{B}$, is defined as the $n \times m$ matrix where $(\mathbf{A} \odot \mathbf{B})_{i,j} = \mathbf{A}_{i,j}\mathbf{B}_{i,j}$.

Distributed training. The process of training a model from data can be cast as an optimization problem known as *empirical risk minimization* (ERM):

$$\min_{w} \frac{1}{n} \sum_{i=1}^{n} \ell(\mathbf{w}; \mathbf{x}_i)$$

where $\mathbf{x}_i \in \mathbb{R}^m$ represents the ith data point, n is the total number of data points, $\mathbf{w} \in \mathbb{R}^d$ is a model, and $\ell(\cdot;\cdot)$ is a loss function that measures the accuracy of the predictions made by the model on each data point.

One way to approximately solve the above ERM is through stochastic gradient descent (SGD), which operates as follows. We initialize the model at an initial point $\mathbf{w}_0$ and then iteratively update it according to

$$\mathbf{w}_k = \mathbf{w}_{k-1} - \gamma \nabla \ell(\mathbf{w}_{k-1}; \mathbf{x}_{i_k}),$$

where i_k is a random data-point index sampled from $\{1, \ldots, n\}$, and $\gamma > 0$ is the learning rate.

In order to take advantage of distributed systems and parallelism, we often use *mini-batch* SGD. At each iteration of mini-batch SGD, we select a random subset $S_k \subseteq \{1, \ldots, n\}$ of the data and update our model according to

$$\mathbf{w}_k = \mathbf{w}_{k-1} - \frac{\gamma}{|S_k|} \sum_{i \in S_k} \nabla \ell(\mathbf{w}_{k-1}; \mathbf{x}_i).$$

Many distributed versions of mini-batch SGD partition the gradient computations across the compute nodes. After computing and summing up their assigned gradients, each nodes sends their respective sum back to the PS. The PS aggregates these sums to update the model $\mathbf{w}_{k-1}$ according to the rule above.

In this dissertation, we consider the question of how to perform this update method in a distributed and robust manner. Fix a batch (or set of points) S_k, which after relabeling we assume equals $\{1, \ldots, B\}$. We will denote $\nabla \ell(\mathbf{w}_{k-1}; \mathbf{x}_i)$ by $\mathbf{g}_i$. The fundamental question we consider in this work is how to compute $\sum_{i=1}^{B} \mathbf{g}_i$ in a distributed and *adversary-resistant* manner. We present DRACO, a framework that

can compute this summation in a distributed manner, even under the presence of adversaries.

Remark 6.1. *In contrast to previous works, our analysis and framework are applicable to any distributed algorithm which requires the sum of multiple functions. Notably, our framework can be applied to any first-order methods, including gradient descent, SVRG (Johnson and Zhang, 2013), coordinate descent, and projected or accelerated versions of these algorithms. For the sake of simplicity, our discussion in the rest of the text will focus on mini-batch SGD.*

Adversarial compute node model. We consider the setting where a subset of size s of the P compute nodes act adversarially against the training process. The goal of an adversary can either be to completely mislead the end model, or bias it towards specific areas of the parameter space. A compute node is considered to be an adversarial node, if it does not return the prescribed gradient update given its allocated samples. Such a node can ship back to the PS any arbitrary update of dimension equal to that of the true gradient. Mini-batch SGD fails to converge even if there is only a single adversarial node (Blanchard et al., 2017a).

In this dissertation, we consider the strongest possible adversaries. We assume that each adversarial node has access to infinite computational power, the entire data set, the training algorithm, and has knowledge of any defenses present in the system. Furthermore, all adversarial nodes may collaborate with each other.

6.2 DRACO: robust distributed training via algorithmic redundancy

In this section we present our main results for DRACO. The proofs can be found in Section 6.5. We generalize the scheme in Figure 1.2 to P compute nodes and B data samples. At each iteration of our training process, we assign the B gradients to the P compute nodes using a $P \times B$ *allocation matrix* $\mathbf{A}$. Here, $\mathbf{A}_{j,k}$ is 1 if node j is assigned the kth gradient $\mathbf{g}_k$, and 0 otherwise. The support of $\mathbf{A}_{j,\cdot}$, denoted

supp $(\mathbf{A}_{j,.})$, is the set of indices k of gradients evaluated by the jth compute node. For simplicity, we will assume $B = P$ throughout the following.

DRACO utilizes redundant computations, so it is worth formally defining the amount of redundancy incurred. This is captured by the following definition.

Definition 6.2. $r \triangleq \frac{1}{P}\|\mathbf{A}\|_0$ *denotes the redundancy ratio.*

In other words, the redundancy ratio is the average number of gradients assigned to each compute node.

We define a $d \times P$ matrix $\mathbf{G}$ by $\mathbf{G} \triangleq [\mathbf{g}_1, \mathbf{g}_2, \cdots, \mathbf{g}_P]$. Thus, $\mathbf{G}$ has all assigned gradients as its columns. The jth compute node first computes a $d \times P$ gradient matrix $\mathbf{Y}_j \triangleq (\mathbf{1}_d \mathbf{A}_{j,.}) \odot \mathbf{G}$ using its allocated gradients. In particular, if the kth gradient $\mathbf{g}_k$ is allocated to the jth compute node, *i.e.*, $\mathbf{A}_{j,k} \neq 0$, then the compute node computes $\mathbf{g}_k$ as the kth column of $\mathbf{Y}_j$. Otherwise, it sets the k-th column of $\mathbf{Y}_j$ to be $\mathbf{0}_d$.

The jth compute node is equipped with an encoding function E_j that maps the $d \times P$ matrix $\mathbf{Y}_j$ of its assigned gradients to a single d-dimensional vector. After computing its assigned gradients, the jth compute node sends $\mathbf{z}_j \triangleq E_j(\mathbf{Y}_j)$ to the PS. If the jth node is adversarial then it instead sends $\mathbf{z}_j + \mathbf{n}_j$ to the PS, where $\mathbf{n}_j$ is an arbitrary d-dimensional vector. We let E be the set of local encoding functions, *i.e.*, $E = \{E_1, E_2, \cdots, E_P\}$.

Let us define a $d \times P$ matrix $\mathbf{Z}^{\mathbf{A},E,\mathbf{G}}$ by $\mathbf{Z}^{\mathbf{A},E,\mathbf{G}} \triangleq [\mathbf{z}_1, \mathbf{z}_2, \cdots, \mathbf{z}_P]$, and a $d \times P$ matrix $\mathbf{N}$ by $\mathbf{N} \triangleq [\mathbf{n}_1, \mathbf{n}_2, \cdots, \mathbf{n}_P]$. Note that at most s columns of $\mathbf{N}$ are non-zero. Under this notation, after all updates are finished the PS receives a $d \times P$ matrix $\mathbf{R} \triangleq \mathbf{Z}^{\mathbf{A},E,\mathbf{G}} + \mathbf{N}$. The PS then computes a d-dimensional update gradient vector $\mathbf{u} \triangleq D(\mathbf{R})$ using a decoder function D.

The system in DRACO is determined by the tuple $(\mathbf{A}, E, D)$. We decide how to assign gradients by designing $\mathbf{A}$, how each compute node should locally amalgamate its gradients by designing E, and how the PS should decode the output by designing D. The process of DRACO is illustrated in Figure 6.1.

This framework of $(\mathbf{A}, E, D)$ encompasses both distributed SGD and the GM approach. In distributed mini-batch SGD, we assign 1 gradient to each compute

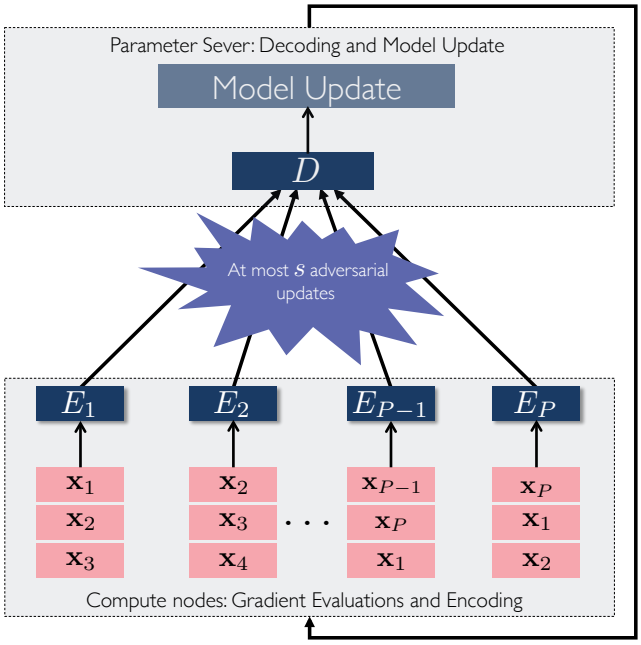


Figure 6.1: In DRACO, each compute node is allocated a subset of the data set. Each compute node computes redundant gradients, encodes them via E_i, and sends the resulting vector to the PS. These received vectors then pass through a decoder that detects where the adversaries are and removes their effects from the updates. The output of the decoder is the true sum of the gradients. The PS applies the updates to the parameter model and we then continue to the next iteration.

node. After relabeling, we can assume that we assign $\mathbf{g}_i$ to compute node i. Therefore, $\mathbf{A}$ is simply the identity matrix $\mathbf{I}_P$. The matrix $\mathbf{Y}_j$ therefore contains $\mathbf{g}_j$ in column j and 0 in all other columns. The local encoding function E_j simply returns $\mathbf{g}_j$ by computing $E_j(\mathbf{Y}_j) = \mathbf{Y}_j\mathbf{1}_P = \mathbf{g}_j$, which it then sends to the PS. The decoding function now depends on the algorithm. For vanilla mini-batch SGD, the PS takes the average of the gradients, while in the GM approach, it takes a geometric median of the gradients.

In order to guarantee convergence, we want DRACO to exactly recover the true sum of gradients, regardless of the behavior of the adversarial nodes. In other words, we want DRACO to protect against *worst-case* adversaries. Formally, we

want the PS to always obtain the d-dimensional vector $\mathbf{G1}_P$ via DRACO with any s adversarial nodes. Below is the formal definition.

Definition 6.3. *DRACO with* $(\mathbf{A}, E, D)$ *can tolerate* s *adversarial nodes, if for any* $\mathbf{N} = [\mathbf{n}_1, \mathbf{n}_2, \cdots, \mathbf{n}_P]$ *such that* $|\{j : \mathbf{n}_j \neq 0\}| \leqslant s$, *we have* $D(\mathbf{Z}^{\mathbf{A},E,\mathbf{G}} + \mathbf{N}) = \mathbf{G1}_P$.

Remark 6.4. *If we can successfully defend against* s *adversaries, then the model update after each iteration is identical to that in the adversary-free setup. This implies that any guarantees of convergence in the adversary-free case transfer to the adversarial case.*

Redundancy bound. We first study how much redundancy is required if we want to exactly recover the correct sum of gradients per iteration in the presence of s adversaries.

Theorem 6.5. *Suppose a selection of gradient allocation, encoding, and decoding mechanisms* $(\mathbf{A}, E, D)$ *can tolerate* s *adversarial nodes. Then its redundancy ratio* r *must satisfy* $r \geqslant 2s + 1$.

The above result is information–theoretic, meaning that regardless of how the compute node encodes and how the PS decodes, each data sample has to be replicated at least $2s + 1$ times to defend against s adversarial nodes.

Remark 6.6. *Suppose that a tuple* $(\mathbf{A}, E, D)$ *can tolerate any* s *adversarial nodes. By Theorem 6.5, this implies that on average, each compute node encodes at least* $(2s + 1)$ d*-dimensional vectors. Therefore, if the encoding has linear complexity, then each encoder requires* $(2s + 1)d$ *operations in the worst-case. If the decoder* D *has linear time complexity, then it requires at least* Pd *operations in the worst case, as it needs to use the* d*-dimensional input from all* P *compute nodes. This gives a computational cost of* $O(Pd)$ *in general, which is significantly less than that of the median approach in* (*Blanchard et al., 2017a*), *which requires* $\mathcal{O}(P^2(d + \log P))$ *operations.*

Optimal coding schemes. A natural question is, *can we achieve the optimal redundancy bound with linear-time encoding and decoding*? More formally, can we design a tuple $(\mathbf{A}, E, D)$ that has redundancy ratio $r = 2s + 1$ and computation complexity

$\mathcal{O}((2s+1)d)$ at the compute node and $\mathcal{O}(Pd)$ at the PS? We give a positive answer by presenting two coding approaches that match the above bounds. The encoding methods are based on the fractional repetition code and the cyclic repetition codes in (Tandon et al., 2017; Raviv et al., 2017).

Fractional repetition code. Suppose $2s+1$ divides P. The fractional repetition code (derived from (Tandon et al., 2017)) works as follows. We first partition the compute nodes into $r = 2s+1$ groups. We assign the nodes in a group to compute the same sum of gradients. Let $\hat{\mathbf{g}}$ be the desired sum of gradients per iteration. In order to decode the outputs returned by the compute nodes in the same group, the PS uses majority vote to select one value. This guarantees that as long as fewer than half of the nodes in a group are adversarial, the majority procedure will return the correct $\hat{\mathbf{g}}$.

Formally, the repetition code $(\mathbf{A}^{Rep}, E^{Rep}, D^{Rep})$ is defined as follows. The assignment matrix $\mathbf{A}^{Rep}$ is given by

$$\mathbf{A}^{Rep} = \begin{bmatrix} \mathbf{1}_{r\times r} & \mathbf{0}_{r\times r} & \mathbf{0}_{r\times r} & \cdots & \mathbf{0}_{r\times r} & \mathbf{0}_{r\times r} \\ \mathbf{0}_{r\times r} & \mathbf{1}_{r\times r} & \mathbf{0}_{r\times r} & \cdots & \mathbf{0}_{r\times r} & \mathbf{0}_{r\times r} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \mathbf{0}_{r\times r} & \mathbf{0}_{r\times r} & \mathbf{0}_{r\times r} & \cdots & \mathbf{0}_{r\times r} & \mathbf{1}_{r\times r} \end{bmatrix}.$$

The jth compute node first computes all its allocated gradients $\mathbf{Y}_j^{Rep} = \left(\mathbf{1}_d \mathbf{A}_{j,\cdot}^{Rep}\right) \odot \mathbf{G}$. Its encoder function simply takes the summation of all the allocated gradients. That is, $E_j^{Rep}(\mathbf{Y}_j^{Rep}) = \mathbf{Y}_j^{Rep}\mathbf{1}_P$. It then sends $\mathbf{z}_j = E_j^{Rep}(\mathbf{Y}_j^{Rep})$ to the PS.

The decoder works by first finding the majority vote of the output of each compute node that was assigned the same gradients. For instance, since the first r compute nodes were assigned the same gradients, it finds the majority vote of $[\mathbf{z}_1, \ldots, \mathbf{z}_r]$. It does the same with each of the blocks of size r, and then takes the sum of the P/r majority votes. We note that our decoder here is different compared to the one used in the straggler mitigation setup of (Tandon et al., 2017). Our decoder follows the concept of majority decoding similarly to (Von Neumann, 1956;

Pippenger, 1988).

Formally, D^{Rep} is given by $D^{Rep}(\mathbf{R}) = \sum_{\ell=1}^{\frac{P}{r}} Maj\left(\mathbf{R}_{\cdot,(\ell\cdot(r-1)+1):(\ell\cdot r)}\right)$, where $Maj(\cdot)$ denotes the majority vote function and $\mathbf{R}$ is the $d \times P$ matrix received from all compute nodes. While a naive implementation of majority vote scales quadratically with the number of compute nodes P, we instead use a streaming version of majority vote (Boyer and Moore, 1991), the complexity of which is linear in P.

Theorem 6.7. *Suppose* $2s+1$ *divides* P. *Then the repetition code* $(\mathbf{A}^{Rep}, E^{Rep}, D^{Rep})$ *with* $r = 2s+1$ *can tolerate any* s *adversaries, achieves the optimal redundancy ratio, and has linear-time encoding and decoding.*

Cyclic code. Next we describe a cyclic code whose encoding method comes from (Tandon et al., 2017) and is similar to that of (Raviv et al., 2017). We denote the cyclic code, with encoding and decoding functions, by $(\mathbf{A}^{Cyc}, E^{Cyc}, D^{Cyc})$. The cyclic code provides an alternative way to tolerate adversaries in distributed setups. We will show that the cyclic code also achieves the optimal redundancy ratio and has linear-time encoding and decoding. Another difference compared to the repetition code is that in the cyclic code, the compute nodes will compute and transmit complex vectors, and the decoding function will take as input these complex vectors.

To better understand the cyclic code, imagine that all P gradients we wish to compute are arranged in a circle. Since there are P starting positions, there are P possible ways to pick a sequence consisting of $2s+1$ clock-wise consecutive gradients in the circle. Assigning each sequence of gradients to each compute node leads to redundancy ratio $r = 2s+1$. The allocation matrix for the cyclic code is $\mathbf{A}^{Cyc}$, where the i row contains $r = 2s+1$ consecutive ones, between position $(i-1)r+1$ to $i \cdot r$ modulo B.

In the cyclic code, each compute node computes a linear combination of its assigned gradients. This can be viewed as a generalization of the repetition code's encoder. Formally, we construct some $P \times P$ matrix $\mathbf{W}$ such that $\forall j, \ell, \mathbf{A}^{Cyc}_{j,\ell} \neq 0$ implies $\mathbf{W}_{j,\ell} = 0$. Let $\mathbf{Y}^{Cyc}_j = \left(\mathbf{1}_d \mathbf{A}^{Cyc}_{j,\cdot}\right) \odot \mathbf{G}$ denote the gradients computed at compute node j. The local encoding function E^{Cyc}_j is defined by $E^{Cyc}_j(\mathbf{Y}^{Cyc}_j) = \mathbf{G}\mathbf{W}_{\cdot,j}$.

After performing this local encoding, the jth compute node then sends $\mathbf{z}_j^{\texttt{Cyc}} \triangleq \mathsf{E}_j^{Cyc}(\mathbf{Y}_j^{\texttt{Cyc}})$ to the PS. Let $\mathbf{Z}^{\mathbf{A}^{\texttt{Cyc}},\mathsf{E}^{\texttt{Cyc}},\mathbf{G}} \triangleq [\mathbf{z}_1^{\texttt{Cyc}}, \mathbf{z}_2^{\texttt{Cyc}}, \cdots, \mathbf{z}_P^{\texttt{Cyc}}]$. Then one can verify from the definition of E_j^{Cyc} that $\mathbf{Z}^{\mathbf{A}^{\texttt{Cyc}},\mathsf{E}^{\texttt{Cyc}},\mathbf{G}} = \mathbf{GW}$. The received matrix at the PS now becomes $\mathbf{R}^{\texttt{Cyc}} = \mathbf{Z}^{\mathbf{A}^{\texttt{Cyc}},\mathsf{E}^{\texttt{Cyc}},\mathbf{G}} + \mathbf{N} = \mathbf{GW} + \mathbf{N}$.

In order to decode, the PS needs to detect which compute nodes are adversarial and recover the correct gradient summation from the non-adversarial nodes. Methods to do the latter alone in the presence of straggler nodes was presented in (Tandon et al., 2017) and (Raviv et al., 2017). Suppose there is a function $\phi(\cdot)$ that can compute the adversarial node index set V. We will later construct ϕ explicitly. Let U be the index set of the non-adversarial nodes. Suppose that the span of $\mathbf{W}_{\cdot,U}$ contains $\mathbf{1}_P$. Thus, we can obtain a vector $\mathbf{b}$ by solving $\mathbf{W}_{\cdot,U}\mathbf{b} = \mathbf{1}_P$. Finally, since U is the index set of non-adversarial nodes, for any $j \in U$, we must have $\mathbf{n}_j = \mathbf{0}$. Thus, we can use $\mathbf{R}_{\cdot,U}^{\texttt{Cyc}}\mathbf{b} = (\mathbf{GW} + \mathbf{N})_{\cdot,U}\mathbf{b} = \mathbf{GW}_{\cdot,U}\mathbf{b} = \mathbf{G1}_P$. The decoder function is given formally in Algorithm 4.

Algorithm 4 Decoder Function $\mathsf{D}^{\texttt{Cyc}}$.

Input : Received $d \times P$ matrix $\mathbf{R}^{\texttt{Cyc}}$
Output: Desired gradient summation $\mathbf{u}^{\texttt{Cyc}}$
$V = \phi(\mathbf{R})$ // Locate the adversarial node indexes.
$U = \{1, 2, \cdots, P\} - V$. // Non-adversarial node indexes
Find $\mathbf{b}$ by solving $\mathbf{W}_{\cdot,U}\mathbf{b} = \mathbf{1}_P$
Compute and return $\mathbf{u}^{\texttt{Cyc}} = \mathbf{R}_{\cdot,U}\mathbf{b}$

To make this approach work, we need to design a matrix $\mathbf{W}$ and the index location function $\phi(\cdot)$ such that (i) For all j, k, $\mathbf{A}_{j,k} = 0 \implies \mathbf{W}_{j,k} = 0$ and the span of $\mathbf{W}_{\cdot,U}$ contains $\mathbf{1}_P$, and (ii) $\phi(\cdot)$ can locate the adversarial nodes.

Let us first construct $\mathbf{W}$. Let $\mathbf{C}$ be a $P \times P$ inverse discrete Fourier transformation (IDFT) matrix, i.e.,

$$\mathbf{C}_{jk} = \frac{1}{\sqrt{P}} \exp\left(\frac{2\pi i}{P}(j-1)(k-1)\right), \quad j, k = 1, 2, \cdots, P.$$

Let $\mathbf{C}_L$ be the first $P - 2s$ rows of $\mathbf{C}$ and $\mathbf{C}_R$ be the last $2s$ rows. Let $\mathbf{ff}_j$ be the set of row indices of the zero entries in $\mathbf{A}_{\cdot,j}^{\texttt{Cyc}}$, *i.e.*, $\mathbf{ff}_j = \{k : \mathbf{A}_{j,k}^{\texttt{Cyc}} = 0\}$. Note that $\mathbf{C}_L$ is a

$(P-2s)\times P$ Vandermonde matrix and thus any $P-2s$ columns of it are linearly independent. Since $|\alpha_j| = P-2s-1$, we can obtain a $P-2s-1$-dimensional vector $\mathbf{q}_j$ uniquely by solving $\mathbf{0} = \begin{bmatrix}\mathbf{q}_j & 1\end{bmatrix}\cdot[\mathbf{C}_L]_{\cdot,\alpha_j}$. Construct a $P\times(P-2s-1)$ matrix $\mathbf{Q} \triangleq \begin{bmatrix}\mathbf{q}_1 & \mathbf{q}_2 & \cdots & \mathbf{q}_P\end{bmatrix}$ and a $P\times P$ matrix $\mathbf{W} \triangleq \begin{bmatrix}\mathbf{Q} & \mathbf{1}_P\end{bmatrix}\cdot\mathbf{C}_L$. One can verify that (i) each row of $\mathbf{W}$ has the same support as the allocation matrix $\mathbf{A}^{Cyc}$ and (ii) the span of any $P-2s+1$ columns of $\mathbf{W}$ contains $\mathbf{1}_P$, summarized as follows.

Lemma 6.8. *For all* j,k, $\mathbf{A}_{j,k}=0 \Rightarrow \mathbf{W}_{j,k}=0$. *For any index set* $\mathcal{U}$ *such that* $|\mathcal{U}| \geqslant P-(2s+1)$, *the column span of* $\mathbf{W}_{\cdot,\mathcal{U}}$ *contains* $\mathbf{1}_P$.

The $\phi(\cdot)$ function works as follows. Given the $d\times P$ matrix $\mathbf{R}^{Cyc}$ received from the compute nodes, we first generate a $1\times d$ random vector $\mathbf{f}\sim\mathcal{N}(\mathbf{1}_{1\times d}, \mathbf{I}_d)$, and then compute $[h_{P-2s}, h_{P-2s-1}, \cdots, h_{P-1}] \triangleq \mathbf{f}\mathbf{R}\mathbf{C}_R^{\dagger}$[1]. We then obtain a vector $\mathbf{fi} = [\beta_0, \beta_1, \cdots, \beta_{s-1}]^T$ by solving

$$\begin{bmatrix} h_{P-s-1} & h_{P-s} & \dots & h_{P-2} \\ h_{P-s-2} & h_{P-s-1} & \dots & h_{P-3} \\ \dots & \dots & \ddots & \vdots \\ h_{P-2s} & h_{P-s+1} & \dots & h_{P-s+1} \end{bmatrix}\begin{bmatrix}\beta_0\\ \beta_1\\ \vdots\\ \beta_{s-1}\end{bmatrix} = \begin{bmatrix}h_{P-1}\\ h_{P-2}\\ \vdots\\ h_{P-s}\end{bmatrix}.$$

We then compute $h_\ell = \sum_{u=0}^{s-1}\beta_u h_{\ell+u-s}$, where $\ell = 0,1,\cdots,P-2s-1$ and $h_\ell = h_{P+\ell}$. Once the vector $\mathbf{h} \triangleq [h_0, h_1, \cdots, h_{P-1}]$ is obtained, we can compute the IDFT of $\mathbf{h}$, denoted by $\mathbf{t} \triangleq [t_0, t_1, \cdots, t_{P-1}]$. The returned index set $V = \{j : t_{j+1} \neq 0\}$. The following lemma shows the correctness of $\phi(\cdot)$.

Lemma 6.9. *Suppose* $\mathbf{N} = [\mathbf{n}_1, \mathbf{n}_2, \cdots, \mathbf{n}_P]$ *satisfies* $|\{j : \|\mathbf{n}_j\|_0 \neq 0\}| \leqslant s$. *Then* $\phi(\mathbf{R}^{Cyc}) = \phi(\mathbf{G}\mathbf{W}+\mathbf{N}) = \{j : \|\mathbf{n}_j\|_0 \neq 0\}$ *with probability 1.*

Finally we can show that the cyclic code can tolerate any s adversaries and also achieves redundancy ratio and has linear-time encoding and decoding.

[1] $\dagger$ denotes transpose conjugate.

Theorem 6.10. *The cyclic code* $(\mathbf{A}^{Cyc}, \mathrm{E}^{Cyc}, \mathrm{D}^{Cyc})$ *can tolerate any* s *adversaries with probability 1 and achieves the redundancy ratio lower bound. For* $d \gg P$*, its encoding and decoding achieve linear-time computational complexity.*

6.3 Experiments

In this section we present an empirical study of DRACO and compare it to the median-based approach in (Chen et al., 2017c) under different adversarial models and real distributed environments. The main findings are as follows: 1) For the same training accuracy, DRACO is up to orders of magnitude faster compared to the GM-based approach; 2) In some instances, the GM approach (Chen et al., 2017c) does not converge, while DRACO converges in all of our experiments, regardless of which dataset, machine learning model, and adversary attack model we use; 3) Although DRACO is faster than GM-based approaches, its runtime can sometimes scale linearly with the number of adversaries due to the algorithmic redundancy needed to defend against adversaries.

Implementation and setup. We compare vanilla mini-batch SGD to both DRACO-based mini-batch SGD and GM-based mini-batch SGD (Chen et al., 2017c). In mini-batch SGD, there is no data replication and each compute node only computes gradients sampled from its partition of the data. The PS then averages all received gradients and updates the model. In GM-based mini-batch SGD, the PS uses the geometric median instead of average to update the model. We have implemented all of these in PyTorch (Paszke et al., 2017) with MPI4py (Dalcin et al., 2011a) deployed on the m4.2/4/10xlarge instances in Amazon EC2 [2]. We conduct our experiments on various adversary attack models, datasets, learning problems and neural network models.

Adversarial attack models. We consider two adversarial models. First is the "reversed gradient" adversary, where adversarial nodes that were supposed to

[2] `https://github.com/hwang595/Draco`

send $\mathbf{g}$ to the PS instead send $-c\mathbf{g}$, for some $c > 0$. Next, we consider a "constant adversary" attack, where adversarial nodes always send a constant multiple κ of the all-ones vector to the PS with dimension equal to that of the true gradient. In our experiments, we set $c = 100$ for the reverse gradient adversary, and $\kappa = -100$ for the constant adversary. At each iteration, s nodes are randomly selected to act as adversaries.

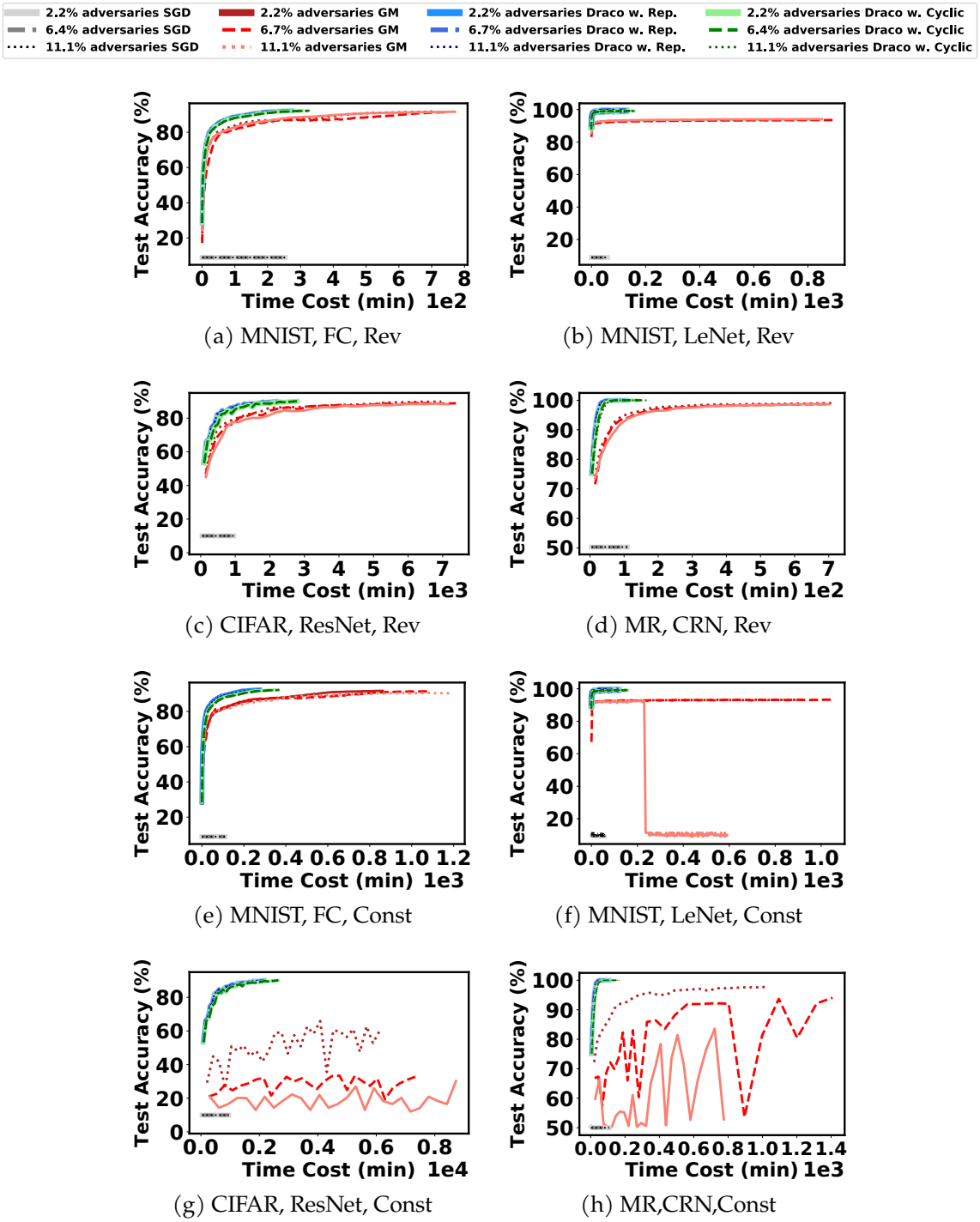
End-to-end convergence performance. We first evaluate the end-to-end convergence performance of DRACO, using both the repetition and cyclic codes, and compare it to ordinary mini-batch SGD as well as the GM approach.

Table 6.1: The datasets used, their associated learning models and corresponding parameters.

Dataset	MNIST	CIFAR10	MR
# data points	70,000	60,000	10,662
Model	FC/LeNet	ResNet-18	CRN
# Classes	10	10	2
# Parameters	1,033k / 431k	1,1173k	154k
Optimizer	SGD	SGD	Adam
Learning Rate	0.01 / 0.01	0.1	0.001
Batch Size	720 / 720	180	180

The datasets and their associated learning models are summarized in Table 6.1. We use fully connected (FC) neural networks and LeNet (LeCun et al., 1998) for MNIST, ResNet-18 (He et al., 2016) for CIFAR-10 (Krizhevsky et al., 2009), and CNN-rand-non-static (CRN) model in (Kim, 2014) for Movie Review (MR) (Pang and Lee, 2005).

The experiments were run on a cluster of 45 compute nodes instantiated on m4.2xlarge instances. At each iteration, we randomly select $s = 1, 3, 5$ (2.2%, 6.7%, 11.1% of all compute nodes) nodes as adversaries. All three methods are trained for



(a) MNIST, FC, Rev

(b) MNIST, LeNet, Rev

(c) CIFAR, ResNet, Rev

(d) MR, CRN, Rev

(e) MNIST, FC, Const

(f) MNIST, LeNet, Const

(g) CIFAR, ResNet, Const

(h) MR,CRN,Const

Figure 6.2: Convergence rates of Draco, GM, and vanilla mini-batch SGD, on (a) MNIST on FC, (b) MNIST on LeNet, (c) CIFAR10 on ResNet-18, and (d) MR on CRN, all with reverse gradient adversaries; (e) MNIST on FC, (f) MNIST on LeNet, (g) CIFAR10 on ResNet-18, and (h) MR on CRN, all with constant adversaries.

10,000 distributed iterations. Figure 6.2 shows how the testing accuracy varies with training time. Tables 6.3 and 6.4 give a detailed account of the speedups of DRACO compared to the GM approach, where we run both systems until they achieve the same designated testing accuracy (the details for MNIST are given in supplement).

Table 6.2: Speedups (*i.e.*, X times faster) of DRACO (Repetition/Cyclic Codes) over GM when using a fully-connected neural network on the MNIST dataset. We run both methods until they reach the same specified testing accuracy. In the table 'const' and 'rev grad' refer to the two types of adversarial updates.

Test Accuracy	80%	85%	88%	90%
2.2% const	**3.4**/**2.7**	**3.5**/**2.8**	**4.8**/**3.9**	**4.1**/**3.1**
6.7% const	**2.7**/**2.0**	**4.1**/**3.1**	**6.0**/**4.6**	**5.6**/**4.1**
11.1% const	**2.9**/**2.2**	**4.8**/**3.7**	**6.1**/**4.7**	**5.3**/**3.8**
2.2% rev grad	**2.2**/**1.9**	**2.4**/**2.2**	**4.1**/**3.7**	**3.2**/**2.9**
6.7% rev grad	**3.1**/**2.5**	**3.3**/**3.1**	**5.5**/**4.8**	**4.5**/**3.7**
11.1% rev grad	**2.7**/**2.3**	**3.0**/**2.6**	**3.1**/**2.7**	**3.1**/**2.6**

As expected, ordinary mini-batch may not converge even if there is only one adversary. Second, under the *reverse gradient adversary* model, DRACO converges several times faster than the GM approach, using both the repetition and cyclic codes, achieving up to more than an order of magnitude speedup compared to the GM approach. We suspect that this is because the computation of the GM is more expensive than the encoding and decoding overhead of DRACO.

Under the *constant adversary* model, the GM approach sometimes failed to converge while DRACO still converged in all of our experiments. This reflects our theory, which shows that DRACO always returns a model identical to the model trained by the ordinary algorithm in an adversary-free environment. One reason why the GM approach may fail to converge is that by using the geometric median, it is actually losing information about a subset of the gradients. Under the constant adversary model, the PS effectively gains no information about the gradients computed by the adversarial nodes, and may not recover the desired optimal model.

Table 6.3: Speedups of DRACO with repetition and cyclic codes over GM when using ResNet-18 on CIFAR10. We run both methods until they reach the same specified testing accuracy. Here ∞ means that the GM approach failed to converge to the same accuracy as DRACO.

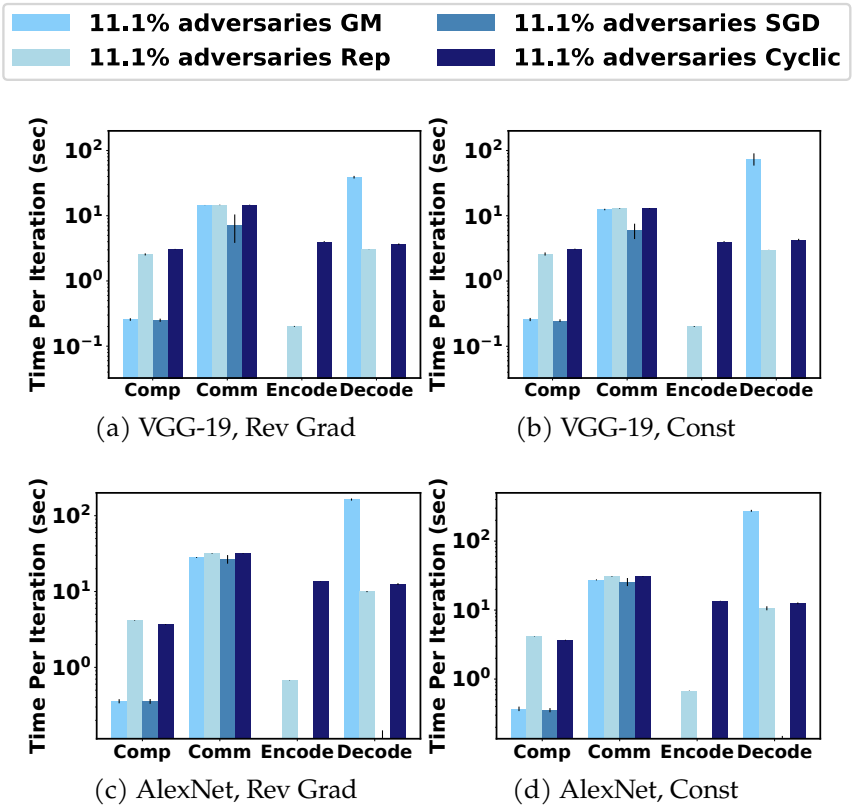
Test Accuracy	80%	85%	88%	90%
2.2% rev grad	**2.6**/**2.0**	**3.3**/**2.6**	**4.2**/**3.3**	∞/∞
6.7% rev grad	**2.8**/**2.2**	**3.4**/**2.7**	**4.3**/**3.4**	∞/∞
11.1% rev grad	**4.1**/**3.3**	**4.2**/**3.2**	**5.5**/**4.4**	∞/∞

Table 6.4: Speedups of DRACOwith repetition and cyclic codes over GM when using CRM on MR. We run both methods until they reach the same specified testing accuracy.

Test Accuracy	95%	96%	98%	98.5%
2.2% rev grad	**5.4**/**4.2**	**5.6**/**4.3**	**9.7**/**7.4**	**12**/**9.0**
6.7% rev grad	**6.4**/**4.5**	**6.3**/**4.5**	**11**/**8.1**	**19**/**13**
11.1% rev grad	**7.5**/**4.7**	**7.4**/**4.6**	**12**/**8**	**19**/**12**

Another reason might be that the GM often requires conditions such as convexity of the underlying loss function. Since neural networks are generally non-convex, we have no guarantees that GM converges in these settings. It is worth noting that GM may also not converge if we use L-BFGS or accelerated gradient descent to perform training, as the choice of algorithm is separate from the underlying geometry of the neural network. Nevertheless, DRACO still converges for such algorithms.

Per iteration cost of DRACO. We provide empirical per iteration costs of applying DRACO to three large state-of-the-art deep networks, ResNet-152, VGG-19, and AlexNet (He et al., 2016; Simonyan and Zisserman, 2015; Krizhevsky et al., 2012). The experiments provided here are run on 46 real instances (45 compute nodes with 1 PS) on AWS EC2. For ResNet-152 and VGG-19, m4.4xlarge (equipped with



(a) VGG-19, Rev Grad (b) VGG-19, Const

(c) AlexNet, Rev Grad (d) AlexNet, Const

Figure 6.3: Empirical Per Iteration Time Cost on Large Models with 11.1% adversarial nodes. We consider reverse gradient adversary on (a) VGG-19 and (b) AlexNet, and constant adversary on (c) VGG-19 and (d) AlexNet. Results on ResNet-152 are in the supplement.

16 cores with 64 GB memory) instances are used while AlexNet experiments are run on m4.10xlarge (40 cores with 160 GB memory) instances. We use a batch size of $B = 180$ and split the data among compute nodes. Therefore, each compute node is assigned $\frac{B}{n} = 4$ data points per iteration. We use the CIFAR10 dataset for all the aforementioned networks. For networks not designed for small images (like AlexNet), we resize the CIFAR10 images to fit the network. As shown in Figure 6.3, with $s = 5$, the encoding and decoding time of DRACO can be several times larger than the computation time of ordinary SGD, though SGD may not converge in adversarial settings. Nevertheless, DRACO is still several times faster than GM.

Table 6.5: Averaged per iteration time costs on ResNet-152 with 11.1% adversary.

Time Cost (sec)	Comp	Comm	Encode	Decode
GM const	1.72	39.74	0	212.31
Rep const	20.81	39.36	0.24	7.74
SGD const	1.64	27.99	0	0.09
Cyclic const	23.08	39.36	5.94	6.64
GM rev grad	1.73	43.98	0	161.29
Rep rev grad	20.71	42.86	0.29	7.54
SGD rev grad	1.69	36.27	0	0.09
Cyclic rev grad	23.08	42.86	5.95	6.65

Table 6.6: Averaged per iteration time costs on VGG-19 with 11.1% adversary.

Time Cost (sec)	Comp	Comm	Encode	Decode
GM const	0.26	12.47	0	74.63
Rep const	2.59	12.91	0.20	3.03
SGD const	0.25	6.9	0	0.03
GM rev grad	0.26	14.57	0	39.02
Rep rev grad	2.55	14.66	0.20	3.04
SGD rev grad	0.25	7.15	0	0.03

Table 6.5, 6.6 and 6.7 provide the detailed cost of the runtime of each component of the algorithm in training ResNet-152, VGG-19 and AlexNet, respectively. While the communication cost is high in both DRACO and the GM method, the decoding time of the GM approach, i.e., its geometric median update at the PS, is prohibitively higher. Meanwhile, the overhead of DRACO is relatively negligible.

Effects of number of adversaries. We also analyze how the number of adversaries affects the performance of DRACO. We ran CIFAR10 on ResNet-18 with 15 compute

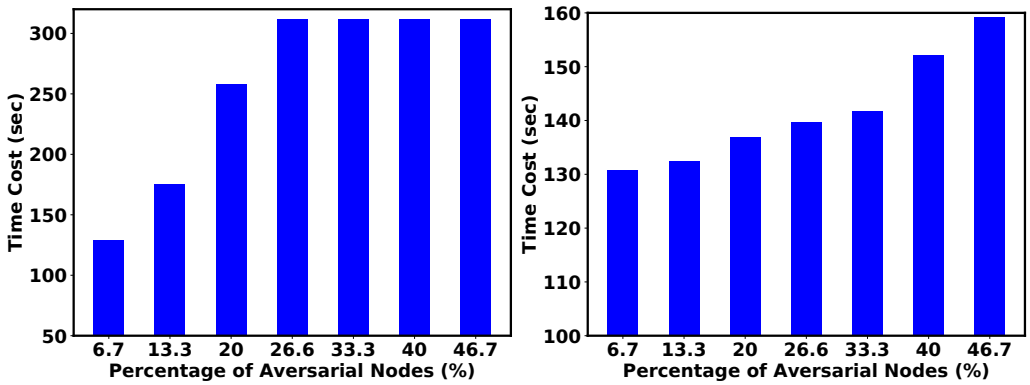
Table 6.7: Averaged per iteration time costs on AlexNet with 11.1% adversarial nodes.

Time Cost (sec)	Comp	Comm	Encode	Decode
GM const	0.37	27.40	0	275.08
Rep const	4.16	30.71	0.67	10.65
SGD const	0.35	25.72	0	0.14
Cyclic const	3.67	30.71	13.55	12.54
GM rev grad	0.36	28.10	0	163.48
Rep rev grad	4.15	31.76	0.67	9.98
SGD rev grad	0.35	26.76	0	0.11
Cyclic rev grad	3.66	31.755	13.55	12.54

nodes, varying the number of adversaries s from 1 to 7. For these experiments, we used the constant adversary model. For the repetition code, we adapted the group size based on s while in the cyclic code we always took $2s + 1$. Figure 6.4 shows the total runtime cost of DRACO does not increase significantly as the number of adversaries increase. This is likely due to the fact that even at $s = 7$, the communication cost (which is not affected by the number of stragglers) is the dominant cost of the algorithm.

6.4 Conclusion

In this chapter, we presented DRACO, a framework for robust distributed training via algorithmic redundancy. DRACO is robust to arbitrarily malicious compute nodes, while being orders of magnitude faster than state-of-the-art robust distributed systems. We give information–theoretic lower bounds on how much redundancy is required to resist adversaries while maintaining the correct update rule, and show that DRACO achieves this lower bound. There are several interesting future directions.



(a) Repetition Code (b) Cyclic Code

Figure 6.4: Time Cost to Reach 70% Test set Accuracy with CIFAR10 dataset run with ResNet-18 on cluster 15 computation nodes varying Percentage of Adversarial Nodes from 6.7% to 46.7% with Constant Adversary (a) Repetition Code and (b) Cyclic Code

First, DRACO is designed to output the same model with or without adversaries. However, slightly inexact model updates often do not decrease performance noticeably. Therefore, we might ask whether we can either (i) tolerate more stragglers or (ii) reduce the computational cost of DRACO by only *approximately* recovering the desired gradient summation. Second, while we give two relatively efficient methods for encoding and decoding, there may be others that are more efficient for use in distributed setups.

6.5 Proofs in DRACO

6.5.1 Proof of Theorem 6.5

For simplicity of proof, let us define a valid s-attack first.

Definition 6.11. $\mathbf{N} = [\mathbf{n}_1, \mathbf{n}_2, \cdots, \mathbf{n}_P]$ *is a valid* s *attack if and only if* $|\{j : \|\mathbf{n}_j\|_0 \neq 0\}| \leqslant s$.

Now we prove theorem 6.5. Suppose $(\mathbf{A}, E, D)$ can resist s adversaries. The goal is to prove $\|A\|_0 \geqslant P(2s+1)$. In fact we can prove a slightly stronger version: $\|\mathbf{A}_{\cdot,i}\|_0 \geqslant (2s+1), i = 1, 2, \cdots, B$. Suppose for some i, $\|\mathbf{A}_{\cdot,i}\|_0 = \tau < (2s+1)$. Without loss of generality, assume that $\mathbf{A}_{1,i}, \mathbf{A}_{2,i}, \mathbf{A}_{\tau,i}$ are non-zero. Let $\mathbf{G}_{-i} = [\mathbf{g}_1, \mathbf{g}_2, \cdots, \mathbf{g}_{i-1}, \mathbf{g}_{i+1}, \cdots, \mathbf{g}_P]$. Since $(\mathbf{A}, E, D)$ can protect against s adversaries, we have for any $\mathbf{G}$,

$$D(\mathbf{Z}^{\mathbf{A},E,\mathbf{G}} + \mathbf{N}) = \mathbf{G}\mathbf{1} = \mathbf{G}_{-i}\mathbf{1} + \mathbf{g}_i,$$

for any valid s-attack $\mathbf{N}$. In particular, let $\mathbf{g}_i^1 = \mathbf{1}_d$,
$\mathbf{g}_2^i = -\mathbf{1}_d$, $\mathbf{G}^1 = [\mathbf{g}_1, \mathbf{g}_2, \cdots, \mathbf{g}_{i-1}, \mathbf{g}_i^1, \mathbf{g}_{i+1}, \cdots, \mathbf{g}_P]$, and
$\mathbf{G}^2 = [\mathbf{g}_1, \mathbf{g}_2, \cdots, \mathbf{g}_{i-1}, \mathbf{g}_i^2, \mathbf{g}_{i+1}, \cdots, \mathbf{g}_P]$. Then for any valid s attack $\mathbf{N}^1, \mathbf{N}^2$,

$$D(\mathbf{Z}^{\mathbf{A},E,\mathbf{G}^1} + \mathbf{N}^1) = \mathbf{G}_{-i}\mathbf{1}_{P-1} + \mathbf{1}_d.$$

and

$$D(\mathbf{Z}^{\mathbf{A},E,\mathbf{G}^2} + \mathbf{N}^2) = \mathbf{G}_{-i}\mathbf{1}_{P-1} - \mathbf{1}_d.$$

Our goal is to find $\mathbf{N}^1, \mathbf{N}^2$ such that $D(\mathbf{Z}^{\mathbf{A},E,\mathbf{G}^1} + \mathbf{N}^1) = D(\mathbf{Z}^{\mathbf{A},E,\mathbf{G}^2} + \mathbf{N}^2)$ which then will lead to a contradiction. Construct $\mathbf{N}^1$ and $\mathbf{N}^2$ by

$$\mathbf{N}^1_{\ell,j} = \begin{cases} \left[\mathbf{Z}^{\mathbf{A},E,\mathbf{N}^2}\right]_{\ell,j} - \left[\mathbf{Z}^{\mathbf{A},E,\mathbf{N}^1}\right]_{\ell,j}, & j = 1, 2, \cdots, \lceil \frac{\tau-1}{2} \rceil \\ 0, & \text{otherwise} \end{cases}$$

and

$$\mathbf{N}^2_{\ell,j} = \begin{cases} \left[\mathbf{Z}^{\mathbf{A},E,\mathbf{N}^1}\right]_{\ell,j} - \left[\mathbf{Z}^{\mathbf{A},E,\mathbf{N}^2}\right]_{\ell,j}, & j = \lceil \frac{\tau-1}{2} \rceil, \lceil \frac{\tau-1}{2} \rceil + 1, \cdots, \tau \\ 0, & \text{otherwise} \end{cases}$$

One can easily verify that $\mathbf{N}^1, \mathbf{N}^2$ are both valid s attack. Meanwhile, we have

$$\left[\mathbf{Z}^{\mathbf{A},E,\mathbf{G}^1}\right]_{\ell,j} + \mathbf{N}^1_{\ell,j} = \left[\mathbf{Z}^{\mathbf{A},E,\mathbf{G}^2}\right]_{\ell,j} + \mathbf{N}^2_{\ell,j}, j = 1, 2, \cdots, \tau$$

due to the above construction of $\mathbf{N}^1, \mathbf{N}^2$. Note that $\mathbf{A}_{j,i} = 0$ for all $j > \tau$, which implies that for all compute nodes with index $j > \tau$, their encoder functions do not depend on the ith gradient. Since $\mathbf{G}^1$ and $\mathbf{G}^2$ only differ in the ith gradient, the encoder function of any compute node with index $j > \tau$ should have the same output. Thus, we have

$$\left[\mathbf{Z}^{\mathbf{A},E,\mathbf{G}^1}\right]_{\ell,j} + \mathbf{N}^1_{\ell,j} = \left[\mathbf{Z}^{\mathbf{A},E,\mathbf{G}^1}\right]_{\ell,j} = \left[\mathbf{Z}^{\mathbf{A},E,\mathbf{G}^2}\right]_{\ell,j} = \left[\mathbf{Z}^{\mathbf{A},E,\mathbf{G}^2}\right]_{\ell,j} + \mathbf{N}^2_{\ell,j}, j > \tau$$

Hence, we have

$$\left[\mathbf{Z}^{\mathbf{A},E,\mathbf{G}^1}\right]_{\ell,j} + \mathbf{N}^1_{\ell,j} = \left[\mathbf{Z}^{\mathbf{A},E,\mathbf{G}^2}\right]_{\ell,j} + \mathbf{N}^2_{\ell,j}, \forall j$$

which means

$$\mathbf{Z}^{\mathbf{A},E,\mathbf{G}^1} + \mathbf{N}^1 = \mathbf{Z}^{\mathbf{A},E,\mathbf{G}^2} + \mathbf{N}^2$$

Therefore, we have

$$D(\mathbf{Z}^{\mathbf{A},E,\mathbf{G}^1} + \mathbf{N}^1) = D(\mathbf{Z}^{\mathbf{A},E,\mathbf{G}^2} + \mathbf{N}^2)$$

and thus

$$\mathbf{G}_{-1}\mathbf{1}_{P-1} + \mathbf{1}_d = D(\mathbf{Z}^{\mathbf{A},E,\mathbf{G}^1} + \mathbf{N}^1) = D(\mathbf{Z}^{\mathbf{A},E,\mathbf{G}^2} + \mathbf{N}^2) = \mathbf{G}_{-1}\mathbf{1}_{P-1} - \mathbf{1}_d$$

This gives us a contradiction. Hence, the assumption is not correct and we must have $\|\mathbf{A}_{\cdot,i}\|_0 \geqslant (2s+1), i = 1, 2, \cdots, P$. Thus, we must have $\|A\|_0 \geqslant (2s+1)P$. □

A direct but interesting corollary of this theorem is a bound on the number of adversaries DRACO can resist.

Corollary 6.12. $(\mathbf{A}, E, D)$ *can resist at most* $\frac{P-1}{2}$ *adversarial nodes.*

Proof. According to Theorem 6.5, the redundancy ratio is at least $2s+1$, meaning that every data point must be replicated by at least $2s+1$. Since there are P compute node in total, we must have $2s+1 \leqslant P$, which implies $s \leqslant \frac{P-1}{2}$. Thus, $(\mathbf{A}, E, D)$ can

resist at most $\frac{P-1}{2}$ adversaries. $\square$ In other words, at least a majority of the compute nodes must be non-adversarial. $\square$

6.5.2 Proof of Theorem 6.7

Since there are at most s adversaries, there are at least $2s+1-s = s+1$ non-adversarial compute nodes in each group. Thus, performing majority vote on each group returns the correct gradient, and thus the repetition code guarantees that the result is correct. The complexity at each compute node is clearly $\mathcal{O}((2s+1)d)$ since each of them only computes the sum of $(2s+1)$ d-dimensional gradients. For the decoder at the PS, within each group of $(2s+1)$ machine, it takes $\mathcal{O}((2s+1)d)$ computations to find the majority. Since there are $\frac{P}{(2s+1)}$ groups, it takes in total $\mathcal{O}((2s+1)d\frac{P}{(2s+1)}) = \mathcal{O}(Pd)$ computations. Thus, this achieves linear-time encoding and decoding. $\square$

6.5.3 Proof of Lemma 6.8

We first prove that $\mathbf{A}_{j,k} = 0 \Rightarrow \mathbf{W}_{j,k} = 0$.

Suppose $\mathbf{A}_{j,k} = 0$ for some j, k. Then by definition $k \in \alpha_j$. By $\mathbf{0} = \begin{bmatrix}\mathbf{q}_j & 1\end{bmatrix} \cdot [\mathbf{C}_L]_{\cdot,\alpha_j}$ we have $0 = \begin{bmatrix}\mathbf{q}_j & 1\end{bmatrix} [\mathbf{C}_L]_{\cdot,k} = \mathbf{W}_{j,k}$.

Next we prove that for any index set U such that $|U| \geqslant P-(2s+1)$, the column span of $\mathbf{W}_{\cdot,U}$ contains $\mathbf{1}$. This is equivalent to that for any index set U such that $|U| \geqslant P-(2s+1)$, there exists a vector $\mathbf{b}$ such that $\mathbf{W}_{\cdot,U} b = \mathbf{1}$. Now we show such b exists. Note that $\mathbf{C}_L$ is a $(P-2s) \times P$ full rank Vandermonde matrix and thus any $P-2s$ columns of $\mathbf{C}_L$ are linearly independent. Let $\bar{U}$ be the first $P-2s$ elements in U. Then all columns of $[\mathbf{C}_L]_{\cdot,\bar{U}}$ are linearly independent and thus $[\mathbf{C}_L]_{\cdot,\bar{U}}$ is invertible. Let $\mathbf{b}_{\bar{U}} \triangleq \bar{\mathbf{b}} = \left(C^L_{\bar{U}}\right)^{-1} \begin{bmatrix}0 & 0 & \cdots & 0 & 1\end{bmatrix}^T$. For any $j \notin \bar{U}$, let $\mathbf{b}_j = 0$. Then we have

$$\begin{aligned}
\mathbf{W}_{\mathrm{U}}\mathbf{b} &= \begin{bmatrix}\mathbf{Q} & \mathbf{1}\end{bmatrix} \times [\mathbf{C}_{\mathrm{L}}]_{\cdot,\mathrm{U}}\, \mathbf{b} \\
&= \begin{bmatrix}\mathbf{Q} & \mathbf{1}\end{bmatrix} \times [\mathbf{C}_{\mathrm{L}}]_{\cdot,\bar{\mathrm{U}}}\, \bar{\mathbf{b}} \\
&= \begin{bmatrix}\mathbf{Q} & \mathbf{1}\end{bmatrix} \times [\mathbf{C}_{\mathrm{L}}]_{\cdot,\bar{\mathrm{U}}} \times [\mathbf{C}_{\mathrm{L}}]^{-1}_{\cdot,\bar{\mathrm{U}}} \begin{bmatrix}0 & 0 & \cdots & 0 & 1\end{bmatrix}^{\mathsf{T}} \\
&= \begin{bmatrix}\mathbf{Q} & \mathbf{1}\end{bmatrix} \begin{bmatrix}0 & 0 & \cdots & 0 & 1\end{bmatrix}^{\mathsf{T}} \\
&= \mathbf{1}.
\end{aligned}$$

This completes the proof. □

6.5.4 Proof of Lemma 6.9

We need a few lemmas first.

Lemma 6.13. *Let a* P*-dimensional vector* $\triangleq [\gamma_1, \gamma_2, \cdots, \gamma_{\mathrm{P}}]^{\mathsf{T}} = (\mathbf{fN})^{\mathsf{T}}$. *Then we have*

$$Pr(\{\mathrm{j} : \gamma_{\mathrm{j}} \neq 0\} = \{\mathrm{j} : \|\mathbf{N}_{\cdot,\mathrm{j}}\|_0 \neq 0\}) = 1.$$

Proof. Let us prove that

$$Pr(\mathbf{N}_{\cdot,\mathrm{j}} \neq 0\}|\gamma_{\mathrm{j}} \neq 0) = 1.$$

and

$$Pr(\gamma_{\mathrm{j}} \neq 0|\mathbf{N}_{\cdot,\mathrm{j}} \neq 0\}) = 1.$$

for any j. Combining those two equations we prove the lemma.

The first equation is straightforward. Suppose $\mathbf{N}_{\cdot,\mathrm{j}} = 0$. Then we immediately have $\gamma_{\mathrm{j}} = \mathbf{fN}_{\cdot,\mathrm{j}} = 0$. For the second one, note that $\mathbf{f}$ has entries drawn independently from the standard normal distribution. Therefore we have that $\gamma_{\mathrm{j}} = \mathbf{fN}_{\cdot,\mathrm{j}} \sim \mathcal{N}(\mathbf{1}^{\mathsf{T}}\mathbf{N}_{\cdot,\mathrm{j}}, \|\mathbf{N}_{\cdot,\mathrm{j}}\|_2^2)$. Since γ_{j} is a random variable with normal distribu-

tion, the probability of it being any particular value is 0. In particular,

$$Pr(\gamma_j = 0 | \mathbf{N}_{\cdot,j} \neq 0\}) = 0,$$

and thus

$$Pr(\gamma_j \neq 0 | \mathbf{N}_{\cdot,j} \neq 0\}) = 1$$

which proves the second equation and finishes the proof. □

Lemma 6.14. $\mathbf{R}^{Cyc}\mathbf{C}_R^\dagger = \mathbf{N}\mathbf{C}_R^\dagger$.

Proof. By definition,

$$\begin{aligned}\mathbf{R}^{Cyc}\mathbf{C}_R^\dagger = (\mathbf{G}\mathbf{W} + \mathbf{N})\,\mathbf{C}_R^\dagger &= \left(\mathbf{G}\begin{bmatrix}\mathbf{Q} & \mathbf{1}\end{bmatrix}\mathbf{C}_L + \mathbf{N}\right)\mathbf{C}_R^\dagger \\ &= \mathbf{G}\begin{bmatrix}\mathbf{Q} & \mathbf{1}\end{bmatrix}\mathbf{C}_L\mathbf{C}_R^\dagger + \mathbf{N}\mathbf{C}_R^\dagger = \mathbf{N}\mathbf{C}_R^\dagger\end{aligned}$$

. In the last equation we use the fact that IDFT matrix is unitary and thus $\mathbf{C}_L\mathbf{C}_R^\dagger = \mathbf{0}_{(P-2s)\times(2s)}$. □

Lemma 6.15. *Let a* P*-dimensional vector* $\hat{\mathbf{h}} \triangleq [\hat{h}_0, \hat{h}_1, \cdots, \hat{h}_{P-1}]^T$ *be the discrete Fourier transformation (DFT) of a* P*-dimensional vector* $\hat{\mathbf{t}} \triangleq [\hat{t}_1, \hat{t}_2, \cdots, \hat{t}_{P-1}]^T$ *which has at most* s *non-zero elements, i.e.,* $\hat{\mathbf{h}} = \mathbf{C}^\dagger\hat{\mathbf{t}}$ *and* $\|\mathbf{t}\|_0 \leqslant s$*. Then there exists a* s*-dimensional vector* $\hat{\beta} \triangleq [\hat{\beta}_0, \hat{\beta}_1, \cdots, \hat{\beta}_{s-1}]^T$*, such that*

$$\begin{bmatrix} \hat{h}_{P-s-1} & \hat{h}_{P-s} & \dots & \hat{h}_{P-2} \\ \hat{h}_{P-s-2} & \hat{h}_{P-s-1} & \dots & \hat{h}_{P-3} \\ \dots & \dots & \ddots & \vdots \\ \hat{h}_{P-2s} & \hat{h}_{P-s+1} & \dots & \hat{h}_{P-s-1} \end{bmatrix} \hat{\beta} = \begin{bmatrix} \hat{h}_{P-1} \\ \hat{h}_{P-2} \\ \vdots \\ \hat{h}_{P-s} \end{bmatrix}. \tag{6.1}$$

Furthermore, for any $\hat{\beta}$ satisfying the above equations,

$$\hat{h}_\ell = \sum_{u=0}^{s-1} \hat{\beta}_u \hat{h}_{\ell+u-s}, \tag{6.2}$$

always holds for all ℓ, where $\hat{h}_\ell = \hat{h}_{P+\ell}$.

Proof. Let $i_1, i_2, \cdots, i_s$ be the index of the non-zero elements in $\hat{\mathbf{t}}$. Let us define the location polynomial $p(\omega) = \prod_{k=1}^{s}(\omega - e^{-\frac{2\pi i}{P} i_k}) \triangleq \sum_{k=0}^{s} \theta_k \omega^k$, where $\theta_s = 1$. Let a s-dimensional vector $\hat{\beta}^* \triangleq -[\theta_0, \theta_1, \cdots, \theta_{s-1}]^T$.

Now we prove that $\hat{\beta} = \hat{\beta}^*$ is a solution to the system of linear equations equation 6.1. To see this, note that by definition, for any λ, we have $0 = p(e^{-\frac{2\pi i}{P} i_\lambda}) = \sum_{k=0}^{s} \theta_k e^{-\frac{2\pi i}{P} i_\lambda k}$. Multiply both side by $\hat{t}_{i_\lambda} e^{-\frac{2\pi i}{P} i_\lambda \eta}$, we have

$$\begin{aligned} 0 &= \hat{t}_{i_\lambda} e^{-\frac{2\pi i}{P} i_\lambda \eta} \sum_{k=0}^{s} \theta_k e^{-\frac{2\pi i}{P} i_\lambda k} \\ &= \hat{t}_{i_\lambda} \sum_{k=0}^{s} \theta_k e^{-\frac{2\pi i}{P} i_\lambda (k+\eta)}. \end{aligned}$$

Summing over λ, we have

$$\begin{aligned} 0 &= \sum_{\lambda=1}^{s} \hat{t}_{i_\lambda} \sum_{k=0}^{s} \theta_k e^{-\frac{2\pi i}{P} i_\lambda (k+\eta)} \\ &= \sum_{k=0}^{s} \theta_k \sum_{\lambda=1}^{s} \hat{t}_{i_\lambda} e^{-\frac{2\pi i}{P} i_\lambda (k+\eta)}. \end{aligned}$$

By definition,

$$\begin{aligned} \hat{h}_j = \mathbf{C}_{j,\cdot} \hat{\mathbf{t}} &= \frac{1}{\sqrt{P}} \sum_{k=0}^{P-1} e^{-\frac{2\pi i}{P} jk} \hat{t}_k \\ &= \frac{1}{\sqrt{P}} \sum_{\lambda=1}^{s} \hat{t}_{i_\lambda} e^{-\frac{2\pi i}{P} i_\lambda j} \end{aligned}$$

. Hence, the above equation becomes

$$0 = \sum_{k=0}^{s} \theta_k \sqrt{P} \hat{h}_{k+\eta}$$

which is equivalent to

$$\hat{h}_{s+\eta} = \sum_{k=0}^{s-1} -\theta_k \hat{h}_{k+\eta}$$

due to the fact that $\theta_s = 1$. By setting $\eta = -s + P - 1, -s + P - 2, \cdots, -s + P - s$, one can easily see that the above equation becomes identical to the system of linear equations in equation 6.1 with $\hat{\beta} = \hat{\beta}^* = -[\theta_0, \theta_1, \cdots, \theta_{s-1}]^T$.

Now let us prove for any $\hat{\beta}$ that satisfies equation equation 6.1, we have equation 6.2. Note that an equivalent form of equation 6.2 is that the following system of linear equations

$$\begin{bmatrix} \hat{h}_{P-s-1+\ell} & \hat{h}_{P-s+\ell} & \dots & \hat{h}_{P-2+\ell} \\ \hat{h}_{P-s-2+\ell} & \hat{h}_{P-s-1+\ell} & \dots & \hat{h}_{P-3+\ell} \\ \dots & \dots & \ddots & \vdots \\ \hat{h}_{P-2s+\ell} & \hat{h}_{P-s+1+\ell} & \dots & \hat{h}_{P-s-1+\ell} \end{bmatrix} \hat{\beta} = \begin{bmatrix} \hat{h}_{P-1+\ell} \\ \hat{h}_{P-2+\ell} \\ \vdots \\ \hat{h}_{P-s+\ell} \end{bmatrix} \tag{6.3}$$

holds for $\ell = 0, 1, 2 \cdots, P - 1$. We prove this by induction. When $\ell = 1$, this is true since $\hat{\beta}$ satisfies the system of linear equations in equation 6.1. Assume it holds for $\ell = \mu$, *i.e.*,

$$\begin{bmatrix} \hat{h}_{P-s-1+\mu} & \hat{h}_{P-s+\mu} & \dots & \hat{h}_{P-2+\mu} \\ \hat{h}_{P-s-2+\mu} & \hat{h}_{P-s-1+\mu} & \dots & \hat{h}_{P-3+\mu} \\ \dots & \dots & \ddots & \vdots \\ \hat{h}_{P-2s+\mu} & \hat{h}_{P-s+1+\mu} & \dots & \hat{h}_{P-s-1+\mu} \end{bmatrix} \hat{\beta} = \begin{bmatrix} \hat{h}_{P-1+\mu} \\ \hat{h}_{P-2+\mu} \\ \vdots \\ \hat{h}_{P-s+\mu} \end{bmatrix}$$

Now we need to prove it also holds when $\ell = \mu + 1$, *i.e.*,

$$\begin{bmatrix} \hat{h}_{P-s-1+\mu+1} & \hat{h}_{P-s+\mu+1} & \dots & \hat{h}_{P-2+\mu+1} \\ \hat{h}_{P-s-2+\mu+1} & \hat{h}_{P-s-1+\mu+1} & \dots & \hat{h}_{P-3+\mu+1} \\ \dots & \dots & \ddots & \vdots \\ \hat{h}_{P-2s+\mu+1} & \hat{h}_{P-s+1+\mu+1} & \dots & \hat{h}_{P-s-1+\mu+1} \end{bmatrix} \hat{\beta} = \begin{bmatrix} \hat{h}_{P-1+\mu+1} \\ \hat{h}_{P-2+\mu+1} \\ \vdots \\ \hat{h}_{P-s+\mu+1} \end{bmatrix}.$$

First, since both $\hat{\beta}, \hat{\beta}^*$ satisfy the induction assumption, we must have

$$\begin{bmatrix} \hat{h}_{P-s-1+\mu} & \hat{h}_{P-s+\mu} & \dots & \hat{h}_{P-2+\mu} \\ \hat{h}_{P-s-2+\mu} & \hat{h}_{P-s-1+\mu} & \dots & \hat{h}_{P-3+\mu} \\ \dots & \dots & \ddots & \vdots \\ \hat{h}_{P-2s+\mu} & \hat{h}_{P-s+1+\mu} & \dots & \hat{h}_{P-s-1+\mu} \end{bmatrix} (\hat{\beta} - \hat{\beta}^*) = \mathbf{0}_s.$$

Due to the induction assumption, one can verify that

$$\begin{aligned} &[\theta_{s-1}, \theta_{s-2}, \cdots, \theta_0] \begin{bmatrix} \hat{h}_{P-s-1+\mu} & \hat{h}_{P-s+\mu} & \dots & \hat{h}_{P-2+\mu} \\ \hat{h}_{P-s-2+\mu} & \hat{h}_{P-s-1+\mu} & \dots & \hat{h}_{P-3+\mu} \\ \dots & \dots & \ddots & \vdots \\ \hat{h}_{P-2s+\mu} & \hat{h}_{P-s+1+\mu} & \dots & \hat{h}_{P-s-1+\mu} \end{bmatrix} \\ =&\begin{bmatrix} \hat{h}_{P-s+\mu} & \hat{h}_{P-s+\mu+1} & \cdots \hat{h}_{P-2+\mu+1} \end{bmatrix}, \end{aligned}$$

and thus we have

$$\begin{aligned} &\begin{bmatrix} \hat{h}_{P-s+\mu} & \hat{h}_{P-s+\mu+1} & \cdots \hat{h}_{P-2+\mu+1} \end{bmatrix} (\hat{\beta} - \hat{\beta}^*) \\ =&[\theta_{s-1}, \theta_{s-2}, \cdots, \theta_0] \begin{bmatrix} \hat{h}_{P-s-1+\mu} & \hat{h}_{P-s+\mu} & \dots & \hat{h}_{P-2+\mu} \\ \hat{h}_{P-s-2+\mu} & \hat{h}_{P-s-1+\mu} & \dots & \hat{h}_{P-3+\mu} \\ \dots & \dots & \ddots & \vdots \\ \hat{h}_{P-2s+\mu} & \hat{h}_{P-s+1+\mu} & \dots & \hat{h}_{P-s-1+\mu} \end{bmatrix} (\hat{\beta} - \hat{\beta}^*) = 0. \end{aligned}$$

Hence,

$$\begin{aligned}
&\begin{bmatrix}\hat{h}_{P-s+\mu} & \hat{h}_{P-s-1+\mu} & \dots & \hat{h}_{P-1+\mu}\end{bmatrix}\hat{\beta} \\
=&\begin{bmatrix}\hat{h}_{P-s+\mu} & \hat{h}_{P-s-1+\mu} & \dots & \hat{h}_{P-1+\mu}\end{bmatrix}\hat{\beta}^* \\
&+\begin{bmatrix}\hat{h}_{P-s+\mu} & \hat{h}_{P-s-1+\mu} & \dots & \hat{h}_{P-1+\mu}\end{bmatrix}(\hat{\beta}-\hat{\beta}^*) \\
=&\hat{h}_{P+\mu} = \hat{h}_{P-1+\mu+1}.
\end{aligned}$$

Furthermore, by induction assumption, we have

$$\begin{aligned}
&\begin{bmatrix}
\hat{h}_{P-s-2+\mu+1} & \hat{h}_{P-s-1+\mu+1} & \dots & \hat{h}_{P-3+\mu+1} \\
\hat{h}_{P-s-3+\mu+1} & \hat{h}_{P-s-2+\mu+1} & \dots & \hat{h}_{P-4+\mu+1} \\
\dots & \dots & \ddots & \vdots \\
\hat{h}_{P-2s+\mu+1} & \hat{h}_{P-s+1+\mu+1} & \dots & \hat{h}_{P-s+1+\mu+1}
\end{bmatrix}\hat{\beta} \\
=&\begin{bmatrix}
\hat{h}_{P-s-1+\mu} & \hat{h}_{P-s-2+\mu} & \dots & \hat{h}_{P-2+\mu} \\
\hat{h}_{P-s-2+\mu} & \hat{h}_{P-s-1+\mu} & \dots & \hat{h}_{P-3+\mu} \\
\dots & \dots & \ddots & \vdots \\
\hat{h}_{P-(2s-1)+\mu} & \hat{h}_{P-s+\mu} & \dots & \hat{h}_{P-s+\mu}
\end{bmatrix}\hat{\beta} \\
=&\begin{bmatrix}\hat{h}_{P-1+\mu} \\ \hat{h}_{P-2+\mu} \\ \vdots \\ \hat{h}_{P-(s-1)+\mu}\end{bmatrix} = \begin{bmatrix}\hat{h}_{P-2+(\mu+1)} \\ \hat{h}_{P-3+(\mu+1)} \\ \vdots \\ \hat{h}_{P-s+(\mu+1)}\end{bmatrix}.
\end{aligned}$$

Combing those two result we have proved

$$\begin{bmatrix}
\hat{h}_{P-s-1+\mu+1} & \hat{h}_{P-s+\mu+1} & \dots & \hat{h}_{P-2+\mu+1} \\
\hat{h}_{P-s-2+\mu+1} & \hat{h}_{P-s-1+\mu+1} & \dots & \hat{h}_{P-3+\mu+1} \\
\dots & \dots & \ddots & \vdots \\
\hat{h}_{P-2s+\mu+1} & \hat{h}_{P-s+1+\mu+1} & \dots & \hat{h}_{P-s-1+\mu+1}
\end{bmatrix}\hat{\beta} = \begin{bmatrix}\hat{h}_{P-1+\mu+1} \\ \hat{h}_{P-2+\mu+1} \\ \vdots \\ \hat{h}_{P-s+\mu+1}\end{bmatrix}.$$

By induction, the equation 6.3 holds for all $\ell = 0, 1, \cdots, P-1$. Equation 6.3 immediately finishes the proof. $\square$

Now we are ready to prove Lemma 6.9. By Lemma 6.13, for the P-dimensional vector $\gamma = (\mathbf{fN})^T$, we have

$$Pr(\{j : \gamma_j \neq 0\} = \{j : \|\mathbf{N}_{\cdot,j}\|_0 \neq 0\}) = 1,$$

Since there are at most s adversaries, the number of non-zero columns in $\mathbf{N}$ is at most s and hence there are at most s non-zero elements in γ, i.e., $\|\gamma\|_0 \leqslant s$, with probability 1. Now consider the case when $\|\gamma\|_0 \leqslant s$. First note that $[h_{P-2s}, h_{P-2s+1}, \cdots, h_{P-1}] = \mathbf{fR}^{Cyc}\mathbf{C}_R^\dagger = \mathbf{fNC}_R^\dagger = \gamma^T\mathbf{C}_R^\dagger$, where the second equation is due to Lemma 6.14. Now let us construct $\hat{\mathbf{h}} = [\hat{h}_0, \hat{h}_1, \cdots, \hat{h}_{P-1}]^T$ by $\hat{\mathbf{h}} = \mathbf{C}^\dagger\gamma$. Note that $\mathbf{C}$ is symmetric and thus $\mathbf{C}^\dagger = \left[\mathbf{C}^\dagger\right]^T$. One can easily verify that $\hat{h}_\ell = h_\ell, \ell = P-2s, P-2s+1, \cdots, P-1$. Therefore, the equation

$$\begin{bmatrix} h_{P-s-1} & h_{P-s} & \dots & h_{P-2} \\ h_{P-s-2} & h_{P-s-1} & \dots & h_{P-3} \\ \dots & \dots & \ddots & \vdots \\ h_{P-2s} & h_{P-s+1} & \dots & h_{P-s+1} \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_{s-1} \end{bmatrix} = \begin{bmatrix} h_{P-1} \\ h_{P-2} \\ \vdots \\ h_{P-s} \end{bmatrix}$$

becomes

$$\begin{bmatrix} \hat{h}_{P-s-1} & h_{P-s} & \dots & \hat{h}_{P-2} \\ \hat{h}_{P-s-2} & \hat{h}_{P-s-1} & \dots & \hat{h}_{P-3} \\ \dots & \dots & \ddots & \vdots \\ \hat{h}_{P-2s} & \hat{h}_{P-s+1} & \dots & \hat{h}_{P-s+1} \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_{s-1} \end{bmatrix} = \begin{bmatrix} \hat{h}_{P-1} \\ \hat{h}_{P-2} \\ \vdots \\ \hat{h}_{P-s} \end{bmatrix}$$

which always has a solution. Assume we find one solution

$$\bar{\beta} = [\bar{\beta}_0, \bar{\beta}_1, \cdots, \bar{\beta}_{P-1}]^T$$

. By the second part of Lemma 6.15, we have

$$\hat{h}_\ell = \sum_{u=0}^{s-1} \bar{\beta}_u \hat{h}_{\ell+u-s}, \forall \ell.$$

Now we prove by induction that $h_\ell = \hat{h}_\ell, \ell = 0, 1, \cdots, P-1$.

When $\ell = 0$, we have

$$\hat{h}_0 = \sum_{u=0}^{s-1} \bar{\beta}_u \hat{h}_{u-s} = \sum_{u=0}^{s-1} \bar{\beta}_u h_{u-s} = h_0$$

where the second equation is due to the fact that $[h_{P-2s}, h_{P-2s-1}, \cdots, h_{P-1}] = [\hat{h}_{P-2s}, \hat{h}_{P-2s-1}, \cdots, \hat{h}_{P-1}]$ and $\hat{h}_{P+\ell} = \hat{h}_\ell, h_{P+\ell} = h_\ell$ (by definition).

Assume that for $\ell \leqslant \mu$, $\hat{h}_\ell = h_\ell$.

When $\ell = \mu + 1$, we have

$$\hat{h}_{\mu+1} = \sum_{u=0}^{s-1} \bar{\beta}_u \hat{h}_{\mu+1+u-s} = \sum_{u=0}^{s-1} \bar{\beta}_u h_{\mu+1+u-s} = h_{\mu+1}$$

where the second equation is because of the induction assumption for $\ell \leqslant \mu$, $\hat{h}_\ell = h_\ell$.

Thus, we have $h_\ell = \hat{h}_\ell$ for all ℓ, which means $\mathbf{h} = \hat{\mathbf{h}} = \mathbf{C}^\dagger \gamma$. Thus $\mathbf{t}$, the IDFT of $\mathbf{h}$, becomes $\mathbf{t} = \mathbf{C}\mathbf{h} = \mathbf{C}\mathbf{C}^\dagger\gamma = \gamma$. Then the returned Index Set $V = \{j : e_{j+1} \neq 0\} = \{j : \gamma_j \neq 0\}$. By Lemma 6.13, with probability 1, $\{j : \gamma_j \neq 0\} = \{j : \|\mathbf{n}_j\|_0 \neq 0\}$. Therefore, we have with probability 1, $V = \{j : \|\mathbf{n}_j\|_0 \neq 0\}$, which finishes the proof. □

6.5.5 Proof of Theorem 6.10

We first prove the correctness of the cyclic code. By Lemma 6.9, the set U contains the index of all non-adversarial compute nodes with probability 1. By Lemma 6.8, there exists $\mathbf{b}$ such that $\mathbf{W}_{\cdot,U}\mathbf{b} = \mathbf{1}$. Therefore, $\mathbf{u}^{Cyc} = \mathbf{R}^{Cyc}_{\cdot,U}\mathbf{b} = (\mathbf{G}\mathbf{W} + \mathbf{N})_{\cdot,U}\mathbf{b} =$

$\mathbf{GW}_{\cdot,U}\mathbf{b} = \mathbf{G1}_P$. Thus, The cyclic code $(\mathbf{A}^{Cyc}, E^{Cyc}, D^{Cyc})$ can recover the desired gradient and hence resist any $\leqslant s$ adversaries with probability 1.

Next we show the efficiency of the cyclic code. By the construction of $\mathbf{A}^{Cyc}$ and $\mathbf{W}$, the redundancy ratio is $2s+1$ which reaches the lower bound. Each compute node needs to compute a linear combination of the gradients of the data it holds, which needs $\mathcal{O}((2s+1)d)$ computations. For the PS, the detection function $\phi(\cdot)$ takes $\mathcal{O}(d)$ (generating the random vector $\mathbf{f}$) + $\mathcal{O}(dP + 2Ps)$ (computing $\mathbf{fRC}_R^{\dagger}$) + $\mathcal{O}(s^2)$ (solving the Toeplitz system of linear equations in equation 6.1) + $\mathcal{O}((P-2s)s)$ (computing $h_\ell, \ell = 0, 1, 2, \cdots, P-2s-1$) + $\mathcal{O}(P \log P)$ (computing the DFT of $\mathbf{h}$) + $\mathcal{O}(P)$ (examining the non-zero elements of $\mathbf{t}$) = $\mathcal{O}(d + dP + 2Ps + s^2 + (P-2s)s + P\log P + P) = \mathcal{O}(dP + Ps + P\log P)$. Finding the vector $\mathbf{b}$ takes $\mathcal{O}(P^3)$ (by simply constructing $\mathbf{b}$ via $[\mathbf{C}_L]_{\cdot,\bar{U}}$, though better algorithms may exist). The recovering equation $\mathbf{R}_{\cdot,U}\mathbf{b}$ takes $O(dP)$. Thus, in total, the decoder at the PS takes $\mathcal{O}(dP + P^3 + P\log P)$. When $d \gg P$, *i.e.*, $d = \Omega(P^2)$, this becomes $\mathcal{O}(dP)$. Therefore, $(\mathbf{A}^{Cyc}, E^{Cyc}, D^{Cyc})$ also achieves linear-time encoding and decoding. □

6.5.6 Streaming majority vote algorithm

In this section we present the Boyer—Moore majority vote algorithm (Boyer and Moore, 1991), which is an algorithm that only needs computation linear in the size of the sequence.

Algorithm 5 Streaming Majority Vote.

Input : n items $I_1, I_2, \cdots, I_n$
Output: The majority of the n items
Initialize an element $Ma = I_1$ and a counter $Counter = 0$.
for $i = 1$ **to** n **do**
  if $Counter == 0$ **then**
    $Ma = I_i$.
    $Counter = 1$.
  else if $Ma == I_i$ **then**
    $Counter = Counter + 1$.
  else
    $Counter = Counter - 1$.
  end
end
Return Ma.

Clearly this algorithm runs in linear time and it is known that if there is a majority item then the algorithm finally will return it (Boyer and Moore, 1991).

7 DETOX: BYZANTINE-RESILIENT DISTRIBUTED TRAINING VIA REDUNDANT GRADIENTS

To improve the resilience of distributed training to worst-case, or Byzantine node failures, several recent approaches have replaced gradient averaging with robust aggregation methods. Such techniques can have high computational costs, often quadratic in the number of compute nodes, and only have limited robustness guarantees. Other methods have instead used redundancy to guarantee robustness, but can only tolerate limited number of Byzantine failures. In this chapter, we present DETOX, a Byzantine-resilient distributed training framework that combines algorithmic redundancy with robust aggregation. DETOX operates in two steps, a filtering step that uses limited redundancy to significantly reduce the effect of Byzantine nodes, and a hierarchical aggregation step that can be used in tandem with any state-of-the-art robust aggregation method. We show theoretically that this leads to a substantial increase in robustness, and has a per iteration runtime that can be nearly linear in the number of compute nodes.

We provide extensive experiments over real distributed setups across a variety of large-scale machine learning tasks, showing that DETOX leads to orders of magnitude accuracy and speedup improvements over many SotA Byzantine-resilient approaches.

7.1 Problem setup

Our goal is to solve solve the following empirical risk minimization problem: $\min_w F(w) := \frac{1}{n}\sum_{i=1}^{n} f_i(w)$ where $w \in \mathbb{R}^d$ denotes the parameters of a model, and f_i is the loss function on the i-th training sample. To approximately solve this problem, we often use mini-batch SGD. First, we initialize at some w_0. At iteration t, we sample S_t uniformly at random from $\{1, \ldots, n\}$, and then update via

$$w_{t+1} = w_t - \frac{\eta_t}{|S_t|} \sum_{i \in S_t} \nabla f_i(w_t), \tag{7.1}$$

where S_t is a randomly selected subset of the n data points. To perform mini-batch SGD in a distributed manner, the global model w_t is stored at the PS and updated according to (7.1), *i.e.,* by using the mean of gradients that are evaluated at the compute nodes.

Let p denote the total number of compute nodes. At each iteration t, during distributed mini-batch SGD, the PS broadcasts w_t to each compute node. Each compute node is assigned $S_{i,t} \subseteq S_t$, and then evaluates the mean of gradients $g_i = \frac{1}{|S_{i,t}|} \sum_{j \in S_{i,t}} \nabla f_j(w_t)$. The PS then updates the global model via $w_{t+1} = w_t - \frac{\eta_t}{p} \sum_{i=1}^{p} g_i$. We note that in our setup we assume that the PS is the owner of the data, and has access to the entire data set of size n.

Distributed training with Byzantine nodes. We assume that a fixed subset Q of size q of the p compute nodes are Byzantine. Let $\hat{g}_i$ be the output of node i. If i is not Byzantine ($i \notin Q$), we say it is "honest", in which case its output $\hat{g}_i = g_i$ where g_i is the true mean of gradients assigned to node i. If i is Byzantine ($i \in Q$), its output $\hat{g}_i$ can be any d-dimensional vector. The PS receives $\{\hat{g}_i\}_{i=1}^{p}$, and can then process these vectors to produce some approximation to the true gradient update in equation 7.1.

We make no assumptions on the Byzantine outputs. In particular, we allow adversaries with full information about F and w_t, and that the Byzantine compute nodes can collude. Let $\epsilon = q/p$ be the fraction of Byzantine nodes. We will assume $\epsilon < 1/2$ throughout.

7.2 DETOX: A redundancy framework to filter most Byzantine gradients

We now describe DETOX, a framework for Byzantine-resilient mini-batch SGD with p nodes, q of which are Byzantine. Let $b \geqslant p$ be the desired batch-size, and let r be an odd integer. We refer to r as the *redundancy ratio*. For simplicity, we will

assume r divides p and that p divides b. DETOX can be directly extended to the setting where this does not hold.

DETOX first computes a random partition of $[p]$ in p/r node groups $A_1, \ldots, A_{p/r}$ each of size r. This will be fixed throughout. We then initialize at some w_0. For $t \geqslant 0$, we wish to compute some approximation to the gradient update in equation 7.1. To do so, we need a Byzantine-robust estimate of the true gradient. Fix t, and let us suppress the notation t when possible. As in mini-batch SGD, let S be a subset of $[n]$ of size b, with each element sampled uniformly at random from $[n]$. We then partition of S in groups $S_1, \ldots, S_{p/r}$ of size br/p. For each $i \in A_j$, the PS assigns node i the task of computing

$$g_j := \frac{1}{|S_j|} \sum_{k \in S_j} \nabla f_k(w) = \frac{p}{rb} \sum_{k \in S_j} \nabla f_k(w). \tag{7.2}$$

If i is an honest node, then its output is $\hat{g}_i = g_j$, while if i is Byzantine, it outputs some d-dimensional $\hat{g}_i$, which is then sent to the PS. The PS then computes $z_j := \text{maj}(\{\hat{g}_i | i \in A_j\})$, where maj denotes the majority vote. If there is no majority, we set $z_j = 0$. We will refer to z_j as the "vote" of group j.

Since some of these votes are still Byzantine, we must do some robust aggregation of the vote. We employ a hierarchical robust aggregation process HIER-AGGR, which uses two user-specified aggregation methods $\mathbf{A}_0$ and $\mathbf{A}_1$. First, the votes are partitioned in to k groups. Let $\hat{z}_1, \ldots, \hat{z}_k$ denote the output of $\mathbf{A}_0$ on each group. The PS then computes $\hat{G} = \mathbf{A}_1(\hat{z}_1, \ldots, \hat{z}_k)$ and updates the model via $w = w - \eta \hat{G}$. This hierarchical aggregation resembles a median of means approach on the votes (Minsker et al., 2015), and has the benefit of improved robustness and efficiency. A description of DETOX is given in Algorithm 6.

7.2.1 Filtering out almost every Byzantine node

We now show that DETOX filters out the vast majority of Byzantine gradients. Fix the iteration t. Recall that all honest nodes in a node group A_j send $\hat{g}_j = g_j$ as in equation 7.2 to the PS. If A_j has more honest nodes than Byzantine nodes then

Algorithm 6 DETOX: Algorithm to be performed at the PS

input Batch size b, redundancy ratio r, compute nodes $1, \ldots, p$, step sizes $\{\eta_t\}_{t \geqslant 0}$.

Randomly partition $[p]$ in "node groups" $\{A_j | 1 \leqslant j \leqslant p/r\}$ of size r.
for $t = 0$ **to** T **do**
  Draw S_t of size b randomly from $[n]$.
  Partition S_t in to groups $\{S_{t,j} | 1 \leqslant j \leqslant p/r\}$ of size rb/p.
  For each $j \in [p/r]$, $i \in A_j$, push w_t and $S_{t,j}$ to compute node i.
  Receive the (potentially Byzantine) p gradients $\hat{g}_{t,i}$ from each node.
  Let $z_{t,j} := \text{maj}(\{\hat{g}_{t,i} | i \in A_j\})$, and 0 if no majority exists. %Filtering step
  Set $\hat{G}_t = \text{HIER-AGGR}(\{z_{t,1}, \ldots, z_{t,p/r}\})$. %Hierarchical aggregation
  Set $w_{t+1} = w_t - \eta \hat{G}_t$. %Gradient update
end for

Algorithm 7 HIER-AGGR: Hierarchical aggregation

input Aggregators $\mathbf{A}_0, \mathbf{A}_1$, votes $\{z_1, \ldots, z_{p/r}\}$, vote group size k.

Let $\hat{p} := p/r$.
Randomly partition $\{z_1, \ldots, z_{\hat{p}}\}$ in to k "vote groups" $\{Z_j | 1 \leqslant j \leqslant k\}$ of size $\hat{p}/k$.
For each vote group Z_j, calculate $\hat{z}_j = \mathbf{A}_0(Z_j)$.
Return $\mathbf{A}_1(\{\hat{z}_1, \ldots, \hat{z}_k\})$.

$z_j = g_j$ and we say z_j is honest. If not, then z_j may not equal g_j in which case z_j is a Byzantine vote. Let X_j be the indicator variable for whether block A_j has more Byzantine nodes than honest nodes, and let $\hat{q} = \sum_j X_j$. This is the number of Byzantine votes. By filtering, DETOX goes from a Byzantine compute node ratio of $\epsilon = q/p$ to a Byzantine vote ratio of $\hat{\epsilon} = \hat{q}/\hat{p}$ where $\hat{p} = p/r$.

We first show that $\mathbb{E}[\hat{q}]$ decreases *exponentially* with r, while $\hat{p}$ only decreases linearly with r. That is, by incurring a constant factor loss in compute resources, we gain an exponential improvement in the reduction of Byzantine nodes. Thus, even small r can drastically reduce the Byzantine ratio of votes. This observation will allow us to instead use robust aggregation methods on the z_j, *i.e.,* the votes, greatly improving our Byzantine robustness. We have the following theorem about $\mathbb{E}[\hat{q}]$. All proofs can be found in Section 7.6. Note that throughout, we did not focus on optimizing constants.

Theorem 7.1. *There is a universal constant* c *such that if the fraction of Byzantine nodes*

is $\epsilon < c$*, then the effective number of Byzantine votes after filtering satisfies* $\mathbb{E}[\hat{q}] = \mathcal{O}\left(\epsilon^{(r-1)/2} q/r\right)$.

We now wish to use this to derive high probability bounds on $\hat{q}$. While the variables X_i are not independent, they are negatively correlated. By using a version of Hoeffding's inequality for weakly dependent variables, we can show that if the redundancy is logarithmic, *i.e.*, $r \approx \log(q)$, then with high probability the number of effective Byzantine votes drops to a constant, *i.e.*, $\hat{q} = \mathcal{O}(1)$.

Corollary 7.2. *There is a constant* c *such that if and* $\epsilon \leqslant c$ *and* $r \geqslant 3 + 2\log_2(q)$ *then for any* $\delta \in (0, \frac{1}{2})$*, with probability at least* $1 - \delta$*, we have that* $\hat{q} \leqslant 1 + 2\log(1/\delta)$.

In the next section, we exploit this dramatic reduction of Byzantine votes to derive strong robustness guarantees for DETOX.

7.3 DETOX improves the speed and robustness of robust estimators

Using the results of the previous section, if we set the redundancy ratio to $r \approx \log(q)$, the filtering stage of DETOX reduces the number of Byzantine votes $\hat{q}$ to roughly a constant. While we could apply some robust aggregator $\boldsymbol{A}$ directly to the output votes of the filtering stage, such methods often scale poorly with the number of votes $\hat{p}$. By instead applying HIER-AGGR, we greatly improve efficiency and robustness. Recall that in HIER-AGGR, we partition the votes into k "vote groups", apply some $\boldsymbol{A}_0$ to each group, and apply some $\boldsymbol{A}_1$ to the k outputs of $\boldsymbol{A}_0$. We analyze the case where k is roughly constant, $\boldsymbol{A}_0$ computes the mean of its inputs, and $\boldsymbol{A}_1$ is a robust aggregator. In this case, HIER-AGGR is analogous to the Median of Means (MoM) method from robust statistics (Minsker et al., 2015).

Improved speed. Suppose that without redundancy, the time required for the compute nodes to finish is T. Applying KRUM (Blanchard et al., 2017b), MULTI-KRUM (Damaskinos et al., 2019), and BULYAN (Guerraoui et al., 2018) to their p outputs

requires $\mathcal{O}(p^2 d)$ operations, so their overall runtime is $\mathcal{O}(T + p^2 d)$. In DETOX, the compute nodes require r times more computation to evaluate redundant gradients. If $r \approx \log(q)$, this can be done in $\mathcal{O}(\ln(q)T)$. With HIER-AGGR as above, DETOX performs three major operations: (1) majority voting, (2) mean computation of the k vote groups and (3) robust aggregation of the these k means using $\boldsymbol{A}_1$. (1) and (2) require $\mathcal{O}(pd)$ time. For practical $\boldsymbol{A}_1$ aggregators, including MULTI-KRUM and BULYAN, (3) requires $\mathcal{O}(k^2 d)$ time. Since $k \ll p$, DETOX has runtime $\mathcal{O}(\ln(q)T + pd)$. If $T = \mathcal{O}(d)$ (which generally holds for gradient computations), KRUM, MULTI-KRUM, and BULYAN require $\mathcal{O}(p^2 d)$ time, but DETOX only requires $\mathcal{O}(pd)$ time. Thus, DETOX can lead to significant speedups, especially when the number of workers is large.

Improved robustness. To analyze robustness, we first need some distributional assumptions. At a given iteration, let G denote the full gradient of $F(w)$. Throughout this section, we assume that the gradient of each sample is drawn from a distribution $\mathcal{D}$ on $\mathbb{R}^d$ with mean G and covariance Σ. Let $\sigma^2 = \mathrm{Tr}(\Sigma)$, we'll refer to this as variance. In DETOX, the "honest" votes z_i will also have mean G, but their variance will be $\sigma^2 p/rb$. This is because each honest compute node gets rb/p samples, so its variance is reduced by rb/p. Note that this variance reduction is integral in proving that we achieve optimal rates (see Theorem 7.3 and the discussion after it). To see this intuitively, consider a scenario without Byzantine machines, then the variance of empirical mean is σ^2/b. A simple calculation shows that variance of the mean of each "vote group" is $\frac{\sigma^2 p/rb}{\hat{p}/k} = k\sigma^2/b$ where k is the number of vote groups. Thus, if k is small, we are still able to optimally reduce the variance.

Suppose $\hat{G}$ is some approximation to the true gradient G. We say that $\hat{G}$ is a Δ-inexact gradient oracle for G if $\|\hat{G} - G\| \leqslant \Delta$. (Yin et al., 2018b) shows that access to a Δ-inexact gradient oracle is sufficient to upper bound the error of a model $\hat{w}$ produced by performing gradient updates with $\hat{G}$. Thus, to bound the robustness of an aggegator, it suffices to bound Δ. Under the distributional assumptions above, we will derive bounds on Δ for the hierarchical aggregator $\boldsymbol{A}$ with different base aggregators $\boldsymbol{A}_1$.

We will analyze DETOX when $\boldsymbol{A}_0$ computes the mean of the vote groups, and

$\mathbf{A}_1$ is geometric median, coordinate-wise median, or α-trimmed mean (Yin et al., 2018a). We will denote the approximation $\hat{G}$ to G computed by DETOX in these three instances by $\hat{G}_1$, $\hat{G}_2$ and $\hat{G}_3$, respectively. Using the proof techniques similar to (Minsker et al., 2015), we get the following.

Theorem 7.3. *Assume* $r \geqslant 3+2\log_2(q)$ *and* $\epsilon \leqslant c$ *where* c *is the constant from Corollary 7.2. There are constants* c_1, c_2, c_3 *such that for all* $\delta \in (0, 1/2)$*, with probability at least* $1-2\delta$*:*

1. *If* $k = 128\ln(1/\delta)$*, then* $\hat{G}_1$ *is a* $c_1\sigma\sqrt{\ln(1/\delta)/b}$*-inexact gradient oracle.*
2. *If* $k = 128\ln(d/\delta)$*, then* $\hat{G}_2$ *is a* $c_2\sigma\sqrt{\ln(d/\delta)/b}$*-inexact gradient oracle.*
3. *If* $k = 128\ln(d/\delta)$ *and* $\alpha = \frac{1}{4}$*, then* $\hat{G}_3$ *is a* $c_3\sigma\sqrt{\ln(d/\delta)/b}$*-inexact gradient oracle.*

The above theorem has three important implications. First, we can derive robustness guarantees for DETOX that are virtually independent of the Byzantine ratio ϵ. Second, even when there are no Byzantine machines, it is known that no aggregator can achieve $\Delta = o(\sigma/\sqrt{b})$ (Lugosi and Mendelson, 2019), and because we achieve $\Delta = \tilde{O}(\sigma/\sqrt{b})$, we cannot expect to get an order of better robustness by any other aggregator. Third, other than a logarithmic dependence on q, there is no dependence on the number of nodes p. Even as p and q increase, we still maintain roughly the same robustness guarantees.

By comparison, the robustness guarantees of KRUM and Geometric Median applied directly to the compute nodes worsens as as p increases (Blanchard et al., 2017a; Xie et al., 2018a). Similarly, (Yin et al., 2018a) show if we apply coordinate-wise median to p nodes, each of which are assigned b/p samples, we get a Δ-inexact gradient oracle where $\Delta = \mathcal{O}(\sigma\sqrt{\epsilon p/b} + \sigma\sqrt{d/b})$. If ϵ is constant and p is comparable to b, then this is roughly σ, whereas DETOX can produce a Δ-inexact gradient oracle for $\Delta = \tilde{\mathcal{O}}(\sigma/\sqrt{b})$. Thus, the robustness of DETOX can scale much better with the number of nodes than naive robust aggregation of gradients.

7.4 Experiments

In this section we present an experimental study on pairing DETOX with a set of previously proposed robust aggregation methods, including MULTI-KRUM (Blanchard et al., 2017a), BULYAN (Guerraoui et al., 2018), coordinate-wise median (Yin et al., 2018b). We also incorporate DETOX with a recently proposed Byzantine resilient distributed training method *i.e.,* SIGNSGD with majority vote (Bernstein et al., 2018a). We conduct extensive experiments on the scalability and robustness of these Byzantine-resilient methods, and the improvements gained when pairing them with DETOX. All our experiments are deployed on real distributed clusters under various Byzantine attack models. Our implementation is publicly available for reproducibility [1].

7.4.1 Experimental setup

The main findings are as follows: 1) Applying DETOX leads to significant speedups, *e.g.,* up to an order of magnitude end-to-end training speedup is observed; 2) in defending against state-of-the-art Byzantine attacks, DETOX leads to significant Byzantine-resilience improvement, *e.g.,* applying BULYAN on top of DETOX improves the test-set prediction accuracy from 11% to 60% when training VGG13-BN on CIFAR-100 under the "a little is enough" (ALIE) (Baruch et al., 2019) Byzantine attack. Moreover, incorporating SIGNSGD with DETOX improves the test set prediction accuracy from 34.92% to 78.75% when defending against a *constant Byzantine attack* for ResNet-18 trained on CIFAR-10.

We implemented vanilla versions of the aforementioned Byzantine resilient methods, as well as versions of these methods pairing with DETOX, in PyTorch (Paszke et al., 2017) with MPI4py (Dalcin et al., 2011b). Our experiments are deployed on a cluster of 46 m5.2xlarge instances on Amazon EC2, where 1 node serves as the PS and the remaining $p = 45$ nodes are compute nodes. In all the following experiments, we set the number of Byzantine nodes to be $q = 5$. We

[1] https://github.com/hwang595/DETOX

also study the performance of all considered methods with smaller number (and without) Byzantine nodes, the result can be found in Section 7.6.10.

7.4.2 Implementation of DETOX

In DETOX, the 45 compute nodes are randomly partitioned into node groups of size $r = 3$, which gives $p/r = 15$ node groups. Batch size b is set to $1,440$. In each iteration of the vanilla Byzantine resilient methods, each compute node evaluates $b/p = 32$ gradients sampled from its partition of data while in DETOX each node evaluates $r\times$ more gradients *i.e.*, $rb/p = 96$, which makes DETOX $r\times$ more computationally expensive than the vanilla Byzantine resilient methods. Compute nodes in the same node group evaluate the same gradients to create algorithmic redundancy for the majority voting stage in DETOX. The mean of these locally computed gradients is sent back to the PS. Note that although DETOX requires each compute node evaluate $r\times$ more gradients, the communication cost of DETOX is the same as the vanilla Byzantine resilient methods since only the gradient means are communicated instead of individual gradients. After receiving all gradient means from the compute nodes, the PS uses either vanilla Byzantine-resilient methods or their DETOX paired variants.

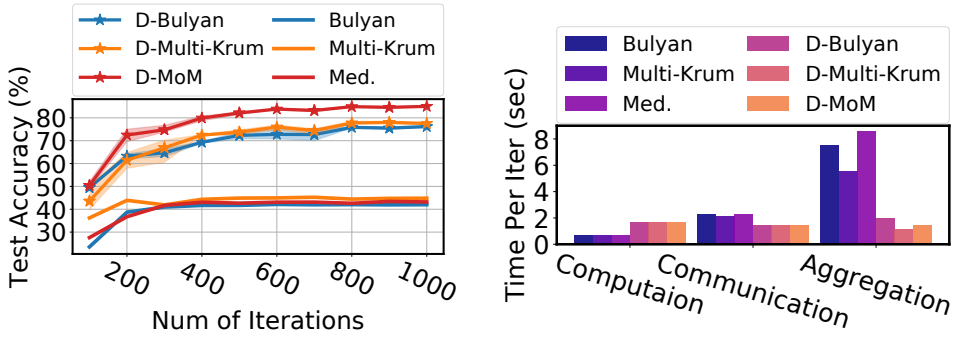


Figure 7.1: Left: Convergence comparisons among various vanilla robust aggregation methods and their DETOX paired versions under "a little is enough" Byzantine attack (Baruch et al., 2019). Right: Per iteration runtime analysis of various methods. All results are for ResNet-18 trained on CIFAR-10. The prefix "D-" stands for a robust aggregation method paired with DETOX.

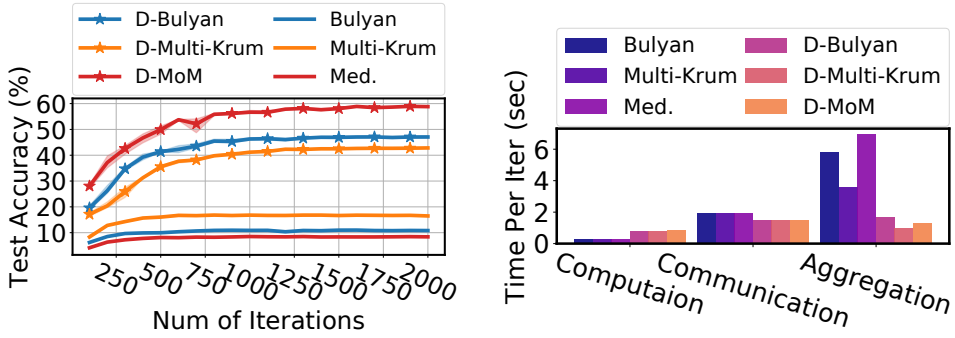


Figure 7.2: Results of VGG13-BN on CIFAR-100. Left: Convergence performance of various robust aggregation methods against ALIE attack. Right: Per iteration runtime analysis of various robust aggregation methods.

We emphasize that DETOX is not simply a new robust aggregation technique. It is instead a general Byzantine-resilient distributed training framework, and any robust aggregation method can be immediately implemented on top of it to increase its Byzantine-resilience and scalability. Note that after the majority voting stage on the PS one has a wide range of choices for $\boldsymbol{A}_0$ and $\boldsymbol{A}_1$.

In our implementations, we had the following setups: 1) $\boldsymbol{A}_0 =$ Mean, $\boldsymbol{A}_1 =$ Coordinate-size Median, 2) $\boldsymbol{A}_0 =$ MULTI-KRUM, $\boldsymbol{A}_1 =$ Mean, 3) $\boldsymbol{A}_0 =$ BULYAN, $\boldsymbol{A}_1 =$ Mean, and 4) $\boldsymbol{A}_0 =$coordinate-wise majority vote, $\boldsymbol{A}_1 =$coordinate-wise majority vote (designed specifically for pairing DETOX with SIGNSGD). We tried $\boldsymbol{A}_0 =$ Mean and $\boldsymbol{A}_1 =$ MULTI-KRUM/BULYAN but we found that setups 2) and 3) had better resilience than these choices. More details on the implementation and system-level optimizations that we performed can be found in Section 7.6.5.

Byzantine attack models. We consider two Byzantine attack models for pairing MULTI-KRUM, BULYAN, and coordinate-wise median with DETOX. First, we consider the *"reversed gradient"* attack, where Byzantine nodes that were supposed to send $\mathbf{g} \in \mathbb{R}^d$ to the PS instead send $-c\mathbf{g}$, for some $c > 0$. Secondly, we study the recently proposed ALIE (Baruch et al., 2019) attack, where the Byzantine compute nodes collude and use their locally calculated gradients to estimate the coordinate-wise mean and standard deviation of the entire set of gradients of all other compute

nodes. The Byzantine nodes then use the estimated mean and variance to manipulate the gradient they send back to the PS. To be more specific, Byzantine nodes will send $\hat{\mu}_i + z \cdot \hat{\sigma}_i, \forall i \in [d]$ where $\hat{\mu}$ and $\hat{\sigma}$ are the estimated coordinate-wise mean and standard deviation each gradient dimension and z is a hyper-parameter which was tuned empirically in (Baruch et al., 2019). Finally, to compare the resilience of the vanilla signSGD and the one paired with Detox, we consider the *"constant Byzantine attack"* where Byzantine compute nodes send a constant gradient matrix with dimension same as that of the true gradient but all elements set to -1.

Datasets and models. Our experiments are over ResNet-18 (He et al., 2016) on CIFAR-10 and VGG13-BN (Simonyan and Zisserman, 2015) on CIFAR-100. For each dataset, we use data augmentation (random crops, and flips) and image normalization. Also, we tune the learning rate schedules and use the constant momentum at 0.9 in all experiments. The details of parameter tuning and dataset normalization are in Section 7.6.6.

7.4.3 Results

Scalability. We report a per-iteration runtime of all considered robust aggregations and their Detox paired variants on both CIFAR-10 over ResNet-18 and CIFAR-100 over VGG-13. The results on ResNet-18 and VGG13-BN are shown in Figure 7.1 and 7.2. We observe that although Detox requires slightly more compute time per iteration, due to its algorithmic redundancy as explained in Section 7.4.2, it largely reduces the PS computation cost during the aggregation stage, which matches our theoretical analysis. Surprisingly, we observe that by applying Detox, the communication costs decrease. This is because the variance of computation time among compute nodes increases with heavier computational redundancy. Therefore, after applying Detox, compute nodes tend not to send their gradients to the PS at the same time, which mitigates a potential network bandwidth congestion. In a nutshell, applying Detox can lead to up to $3\times$ per-iteration speedup.

Table 7.1: Defense results summary for ALIE attacks (Baruch et al., 2019); the reported numbers are test set prediction accuracy.

	D-Multi-Krum	D-Bulyan	**D-Med.**	Multi-Krum	Bulyan	Med.
ResNet-18	80.3%	76.8%	**86.21%**	45.24%	42.56%	43.7%
VGG13-BN	42.98%	46.82%	**59.51%**	17.18%	11.06%	8.64%

Byzatine-resilience under various attacks. We first study the Byzantine-resilience of all considered methods under the ALIE attack, which to the best of our knowledge, is the strongest Byzantine attack proposed in the literature. The results on ResNet-18 and VGG13-BN are shown in Figure 7.1 and 7.2 respectively. Applying Detox leads to significant improvement in Byzantine-resilience compared to vanilla Multi-Krum, Bulyan, and coordinate-wise median on both datasets as shown in Table 7.1. We then consider the *reverse gradient* attack, the results are shown in Figure 7.3.

Since *reverse gradient* is a much weaker attack, all vanilla robust aggregation methods and their Detox paired variants defend well. Moreover, applying Detox leads to significant end-to-end speedups.

In particular, combining the coordinate-wise median with Detox led to a $5\times$ speedup gain in the amount of time to achieve to 90% test set prediction accuracy for ResNet-18 trained on CIFAR-10. The speedup results are shown in Figure 7.4. For VGG13-BN trained on CIFAR-100, an order of magnitude end-to-end speedup can be observed in coordinate-wise median applied on top of Detox.

Comparison between Detox and signSGD. We compare Detox paired signSGD with vanilla signSGD where only the sign of each gradient coordinate is sent to the PS. The PS, on receiving these gradient signs, takes coordiante-wise majority votes to get the model update.

We consider a stronger *constant Byzantine attack* introduced in Section 7.4.2.

The details of our implementation and hyper-parameters used are in Section 7.6.8. The results on both the considered datasets are shown in Figure 7.5 where we see that DETOX paired with SIGNSGD improves the Byzantine resilience of SIGNSGD significantly. For ResNet-18 trained on CIFAR-10, DETOX improves testset prediction accuracy of vanilla SIGNSGD from 34.92% to 78.75%; while for VGG13-BN trained on CIFAR-100, DETOX improves testset prediction accuracy (TOP-1) of vanilla SIGNSGD from 2.12% to 40.37%.

For completeness, we compare DETOX with DRACO (Chen et al., 2018). This is not the focus of this work, as we are primarily interested in showing that DETOX improves the robustness of traditional robust aggregators. However the comparisons with DRACO are in Section 7.6.11. Another experimental study of mean estimation task over synthetic data that directly matches our theory can be found in Section 7.6.9.

7.5 Conclusion

In this chapter, we present DETOX, a new framework for Byzantine-resilient distributed training. Notably, any robust aggregator can be immediatley used with DETOX to increase its robustness and efficiency. We demonstrate these improvements theoretically and empirically. In the future, we would like to devise a privacy-preserving version of DETOX, as currently it requires the PS to be the owner of the data, and also to partition data among compute nodes, which hurts the data privacy. Overcoming this limitation would allow us to develop variants of DETOX for federated learning.

7.6 Additional results

7.6.1 Proof of Theorem 7.1

The following is a more precise statement of the theorem.

Theorem 7.4. *If* $r > 3$, $p \geqslant 2r$ *and* $\epsilon < 1/40$ *then* $\mathbb{E}[\hat{q}]$ *falls as*

$$\mathcal{O}\left(q(40\epsilon(1-\epsilon))^{(r-1)/2}/r\right)$$

which is exponential in r.

Proof. By direct computation,

$$\begin{aligned}
\mathbb{E}(\hat{q}) &= \mathbb{E}\left(\sum_{i=1}^{p/r} X_i\right) \\
&= \frac{p}{r}\mathbb{E}(X_i) \\
&= \frac{p}{r}\frac{\sum_{i=0}^{(r-1)/2}\binom{q}{r-i}\binom{p-q}{i}}{\binom{p}{r}} \\
&\leqslant \frac{p}{r}\frac{\frac{r+1}{2}\binom{q}{(r+1)/2}\binom{p-q}{(r-1)/2}}{\binom{p}{r}} \\
&\leqslant \frac{p}{r}\frac{r+1}{2}\frac{\binom{r}{(r-1)/2}q^{(r+1)/2}(p-q)^{(r-1)/2}}{(p-r)^r} \\
&= \frac{p}{r}\frac{r+1}{2}\frac{\binom{r}{(r-1)/2}q^{(r+1)/2}(p-q)^{(r-1)/2}}{p^r(1-r/p)^r} \\
&\leqslant \frac{p}{r}\frac{r+1}{2}\frac{\binom{r}{(r-1)/2}q^{(r+1)/2}(p-q)^{(r-1)/2}}{p^r(1/2)^r} \\
&= \frac{p}{r}(r+1)2^{r-1}\binom{r}{(r-1)/2}\epsilon^{(r+1)/2}(1-\epsilon)^{(r-1)/2}.
\end{aligned}$$

Note that $\binom{r}{(r-1)/2}$ is the coefficient of $x^{(r+1)/2}(1-x)^{(r-1)/2}$ in the binomial expansion of $1 = 1^r = (x + (1-x))^r$. Therefore, setting $x = \frac{1}{2}$, we find that

$\binom{r}{(r-1)/2} \leqslant 2^r$. Therefore,

$$\begin{aligned}
&\frac{p}{r}(r+1)2^{r-1}\binom{r}{(r-1)/2}\epsilon^{(r+1)/2}(1-\epsilon)^{(r-1)/2} \\
&\leqslant \frac{p}{r}(r+1)2^{2r-1}\epsilon^{(r+1)/2}(1-\epsilon)^{(r-1)/2} \\
&= \frac{p}{r}(r+1)\epsilon\left(2^{2r-1}\epsilon^{(r-1)/2}(1-\epsilon\right)^{(r-1)/2}) \\
&= \frac{2q}{r}(r+1)\left(16\epsilon(1-\epsilon)\right)^{(r-1)/2} \\
&= \frac{2q}{r}\left(16(r+1)^{2/(r-1)}\epsilon(1-\epsilon)\right)^{(r-1)/2}
\end{aligned}$$

Note that since $r > 3$ and r is odd, we have $r \geqslant 5$. Therefore,

$$\mathbb{E}(\hat{q}) \leqslant 2q(40\epsilon(1-\epsilon))^{(r-1)/2}/r.$$

□

For $r = 3$, we have the following lemma.

Lemma 7.5. *If* $r = 3$, *then* $\mathbb{E}[\hat{q}] \leqslant q(4\delta - 2\delta^2)/3$ *when* $n \geqslant 6$.

Proof.

$$\begin{aligned}
\mathbb{E}(q_e) = \mathbb{E}(\sum_{i=1}^{\frac{p}{3}} X_i) = \frac{p}{3}E(X_i) &= \frac{p}{3}\frac{\binom{q}{3}+\binom{q}{2}\binom{p-q}{1}}{\binom{n}{3}} \\
&= \frac{p}{3}\frac{q(q-1)(3p-2q-2)}{p(p-1)(p-2)} = \frac{q}{3}\frac{\left(\epsilon-\frac{1}{p}\right)\left(3-2\delta-\frac{2}{p}\right)}{\left(1-\frac{1}{p}\right)\left(1-\frac{2}{p}\right)} \\
&\leqslant \frac{q}{3}\epsilon\frac{3-2\epsilon-\frac{2}{p}}{1-\frac{2}{p}} \leqslant q\epsilon(4-2\epsilon)/3
\end{aligned}$$

□

7.6.2 Proof of Corollary 7.2

From Theorem 7.1 we see that $\mathbb{E}[\hat{q}] \leqslant 2q(40\epsilon(1-\epsilon))^{(r-1)/2}/r \leqslant 2q(40\epsilon)^{(r-1)/2}$. Now, straightforward analysis implies that if $\epsilon \leqslant 1/80$ and $r \geqslant 3 + 2\log_2 q$ then $\mathbb{E}[\hat{q}] \leqslant 1$. We will then use the following Lemma:

Lemma 7.6. *For all* $\theta > 0$,

$$\mathbb{P}\left[\hat{q} \geqslant \mathbb{E}[\hat{q}](1+\theta)\right] \leqslant \left(\frac{1}{1+\theta/2}\right)^{\mathbb{E}[\hat{q}]\theta/2}$$

Now, using Lemma 7.6 and assuming $\theta \geqslant 2$,

$$\begin{aligned} \mathbb{P}\left[\hat{q} \geqslant \mathbb{E}[\hat{q}](1+\theta)\right] &\leqslant \left(\frac{1}{1+\theta/2}\right)^{\mathbb{E}[\hat{q}]\theta/2} \\ \implies \mathbb{P}\left[\hat{q} \geqslant 1 + \mathbb{E}[\hat{q}]\theta\right] &\leqslant \left(\frac{1}{1+\theta/2}\right)^{\mathbb{E}[\hat{q}]\theta/2} \\ \implies \mathbb{P}\left[\hat{q} \geqslant 1 + \mathbb{E}[\hat{q}]\theta\right] &\leqslant 2^{-\mathbb{E}[\hat{q}]\theta/2} \end{aligned}$$

where we used the fact that $\mathbb{E}[\hat{q}] \leqslant 1$ in the first implication and the assumption that $\theta \geqslant 2$ in the second. Setting $\delta := 2^{-\mathbb{E}[\hat{q}]\theta/2}$, we get the probability bound. Finally, setting $\delta \leqslant 1/2$ makes $\theta \geqslant 2$, which completes the proof.

7.6.3 Proof of Lemma 7.6

We will prove the following:

$$\mathsf{P}\left[\hat{q} \geqslant \mathbb{E}[\hat{q}](1+\theta)\right] \leqslant \left(\frac{1}{1+\frac{\theta}{2}}\right)^{\mathbb{E}[\hat{q}]\theta/2}$$

Proof. We will use the following theorem for this proof (Linial and Luria, 2014; Pelekis and Ramon, 2015).

Theorem 7.7 ((Linial and Luria, 2014)). *Let $X_1, \ldots, X_{\hat{p}}$ be Bernoulli 0/1 random variables. Let $\beta \in (0,1)$ be such that $\beta\hat{p}$ is a positive integer and let k be any positive integer such that $0 < k < \beta\hat{p}$. Then*

$$\mathbb{P}\left[\sum_{i=1}^{\hat{p}} X_i \geqslant \beta\hat{p}\right] \leqslant \frac{1}{\binom{\beta\hat{p}}{k}} \sum_{|A|=k} \mathbb{P}\left[\wedge_{i\in A}(X_i = 1)\right]$$

Let $\beta\hat{p} = \mathbb{E}[\hat{q}](1+\theta)$. Now, $\mathbb{P}[X_i = 1] = \mathbb{E}[X_i] = \mathbb{E}[\hat{q}]/\hat{p}$. We will show that

$$\mathbb{P}\left[\wedge_{i\in A}(X_i = 1)\right] \leqslant (\mathbb{E}[\hat{q}]/\hat{p})^k$$

where $A \subseteq \{1, \ldots, \hat{p}\}$ of size k. To see this, note that for any i, $\mathbb{P}[X_i = 1] = \mathbb{E}[\hat{q}]/\hat{p}$. The conditional probability of some other X_j being 1 given that X_i is 1 would only reduce. Formally, for $i \neq j$,

$$\mathbb{P}[X_j = 1 | X_i = 1] \leqslant \mathbb{P}[X_i = 1] = \epsilon\gamma.$$

Note that for X_i to be 1, the Byzantine machines in the i-th block must be in the majority. Hence, the reduction in the pool of leftover Byzantine machines was more than honest machines. Since the total number of Byzantine machines is less than the number of honest machines, the probability for them being in a majority in block j reduces. Therefore,

$$\mathbb{P}\left[\sum_{i=1}^{\hat{p}} X_i \geqslant \mathbb{E}[\hat{q}](1+\theta)\right] \leqslant \frac{\binom{\hat{p}}{k}}{\binom{\mathbb{E}[\hat{q}](1+\theta)}{k}} \mathbb{P}\left[\wedge_{i\in A}(X_i = 1)\right]$$
$$\leqslant \frac{\binom{\hat{p}}{k}}{\binom{\mathbb{E}[\hat{q}](1+\theta)}{k}} (\mathbb{E}[\hat{q}]/\hat{p})^k$$
$$\leqslant \frac{(\hat{p})^k}{k!\binom{\mathbb{E}[\hat{q}](1+\theta)}{k}} \left(\frac{\mathbb{E}[\hat{q}]}{\hat{p}}\right)^k .$$

Letting $k = \mathbb{E}[\hat{q}]\theta/2$, we then have

$$\mathbb{P}\left[\sum_{i=1}^{\hat{p}} X_i \geqslant \mathbb{E}[\hat{q}](1+\theta)\right] \leqslant \frac{(\hat{p})^k}{(\mathbb{E}[\hat{q}](1+\theta/2))^k} (\mathbb{E}[\hat{q}]/\hat{p})^k$$
$$= \left(\frac{1}{1+\frac{\theta}{2}}\right)^{\mathbb{E}[\hat{q}]\theta/2}$$

□

7.6.4 Proof of Theorem 7.3

We will adapt the techniques of Theorem 3.1 in (Minsker et al., 2015).

Lemma 7.8 ((Minsker et al., 2015), Lemma 2)**.** *Let $\mathbb{H}$ be some Hilbert space, and for $x_1, \ldots, x_k \in \mathbb{H}$, let x_{gm} be their geometric median. Fix $\alpha \in (0, \frac{1}{2})$ and suppose that $z \in \mathbb{H}$ satisfies $\|x_{gm} - z\| > C_\alpha \rho$, where*

$$C_\alpha = (1-\alpha)\sqrt{\frac{1}{1-2\alpha}}$$

and $\rho > 0$. *Then there exists* $J \subseteq \{1, \ldots, k\}$ *with* $|J| > \alpha k$ *such that for all* $j \in J$, $\|x_j - z\| > \rho$.

Note that for a general Hilbert or Banach space $\mathbb{H}$, the geometric median is defined as:

$$x_{gm} := \arg\min \sum_{j=1}^{k} \|x - x_j\|_{\mathbb{H}}$$

where $\|.\|_{\mathbb{H}}$ is the norm on $\mathbb{H}$. This coincides with the notion of geometric median in $\mathbb{R}^2$ under the ℓ_2 norm. Note that Coordinatewise Median is the Geometric Median in the real space with the ℓ^1 norm, which forms a Banach space.

Firstly, we use Corollary 7.2 to see that with probability $1-\delta$, $\hat{q} \leqslant 1 + 2\log(1/\delta)$. Now, we assume that $\hat{q} \leqslant 1 + 2\log(1/\delta)$ is true. We will show the remainder of the theorem holds with probability at least $1 - \delta$, as then a union bound will give us the desired result.

(1): Let us assume that number of clusters is $k = 128 \log 1/\delta$ for some $\delta < 1$, also note that $128 \log 1/\delta \geqslant 8\hat{q}$. Now, choose $\alpha = 1/4$. Choose $\rho = 4\sigma\sqrt{\frac{k}{b}}$. Assume that the Geometric Median is more than $C_\alpha \rho$ distance away from true mean. Then by the previous Lemma, atleast $\alpha = 1/4$ fraction of the empirical means of the clusters must lie atleast ρ distance away from true mean. Because we assume the number of clusters is more than $8\hat{q}$, atleast $1/8$ fraction of empirical means of uncorrupted clusters must also lie atleast ρ distance away from true mean.

Recall that the variance of the mean of an "honest" vote group is given by

$$(\sigma')^2 = \sigma^2 \frac{k}{b}.$$

By applying Chebyshev's inequality to the i^{th} uncorrupted vote group $G[i]$, we find that its empirical mean $\hat{x}$ satisfies

$$\mathbb{P}\left(\|G[i] - G\| \geqslant 4\sigma\sqrt{\frac{k}{b}} \right) \leqslant \frac{1}{16}.$$

Now, we define a Bernoulli event that is 1 if the empirical mean of an uncorrupted vote group is at distance larger than ρ to the true mean, and 0 otherwise. By the computation above, the probability of this event is less than $1/16$. Thus, its mean is less than $1/16$ and we want to upper bound the probability that empirical mean is more than $1/8$. Using the number of events as $k = 128 \log(1/\delta)$, we find that this holds with probability at least $1 - \delta$. For this, we used the following version of Hoeffding's inequality in this part and part (3) of this proof. For Bernoulli events with mean μ, empirical mean $\hat{\mu}$, number of events m and deviation θ:

$$\mathbb{P}(\hat{\mu} - \mu \geqslant \theta) \leqslant \exp(-2m\theta^2)$$

To finish the proof, just plug in the values of C_α given in the Lemma 2.1 (written above) from (Minsker et al., 2015), where $C_\alpha = 3/2\sqrt{2}$ for Geometric Median.

(2): For coordinate-wise median, we set $k = 128 \log d/\delta$. Then we apply the result proved in previous part for each dimension of $\hat{G}$. Then, we get that with probability at least $1 - \delta/d$,

$$|\hat{G}_i - G_i| \leqslant C_1 \sigma_i \sqrt{\frac{\log d/\delta}{b}}$$

where $\hat{G}_i$ is the i^{th} coordinate of $\hat{G}$, G_i is the i^{th} coordinate of G and σ_i^2 is the i^{th} diagonal entry of Σ. Doing a union bound, we get that with probability at least $1 - \delta/d$

$$\|\hat{G} - G\| \leqslant C_1 \sigma \sqrt{\frac{\log d/\delta}{b}}.$$

(3): Define

$$\Delta_i = \sigma_i \sqrt{\frac{k}{b\sqrt{\frac{1}{2k}\log\frac{d}{\delta}}}}$$

where σ_i^2 is the i^{th} diagonal entry of Σ. Now, for each uncorrupted vote group,

using Chebyshev's inequality:

$$\mathbb{P}\left(|\hat{G}_i - G_i| \geqslant \Delta_i\right) \leqslant \sqrt{\frac{1}{2k}\log\frac{d}{\delta}}.$$

Now, i^{th} coordinate of α-trimmed mean lies Δ_i away from G_i if atleast αk of the i^{th} coordinates of vote group empirical means lie Δ_i away from G_i. Note that because of the assumption of the Proposition $\alpha k \geqslant 2\hat{q}$. Because $\hat{q}$ of these can be corrupted, atleast $\alpha k/2$ of true empirical means have i^{th} coordinates that lie Δ_i away from G_i. This means $\alpha/2$ fraction have true empirical means have i^{th} coordinates that lie Δ_i away from G_i. Define a Bernoulli variable X for a vote group as being 1 if the i^{th} coordinate of empirical mean of that vote group lies more than Δ_i away from G_i, and 0 otherwise.

The mean of X therefore satisfies

$$\mathbb{E}(X) < \sqrt{\frac{1}{2k}\log\frac{d}{\delta}}.$$

Set

$$\alpha = 4\sqrt{\frac{1}{2k}\log\frac{d}{\delta}}.$$

Again, using Hoeffding's inequality in a manner analogous to part (1) of the proof, we get that probability of i^{th} coordinate of α-trimmed mean being more than Δ_i away from G_i is less than δ/d.

Taking union bound over all d coordinates, we find that the probability of α-trimmed mean being more than

$$\sigma\sqrt{\frac{k}{b\sqrt{\frac{1}{2k}\log\frac{d}{\delta}}}} = \sigma\sqrt{\frac{4k}{b\alpha}}$$

away from G is less than δ. Hence we have proved that if

$$\alpha = 4\sqrt{\frac{1}{2k}\log\frac{d}{\delta}}$$

and $\alpha k \geqslant 2\hat{q}$, then with probability at least $1-\delta$, $\Delta \leqslant \sigma\sqrt{\frac{4k}{b\alpha}}$. Now, set $\alpha = 1/4$ and $k = 128\log(d/\delta)$. One can easily see that $\alpha k \geqslant 2\hat{q}$ is satisfied and we get that with probability at least $1-\delta$, for some constant C_3,

$$\Delta \leqslant C_3\sigma\sqrt{\frac{\log(d/\delta)}{b}}.$$

7.6.5 Implementation and system-level optimization details

We introduce the details of combining Bulyan, Multi-Krum, and coordinate-wise median with Detox.

- Bulyan: according to (Guerraoui et al., 2018) Bulyan requires $p \geqslant 4q+3$. In Detox, after the first majority voting level, the corresponding requirement in Bulyan becomes $\frac{p}{r} \geqslant 4\hat{q}+3 = 11$. Thus, we assign all "winning" gradients in to one cluster *i.e.,* Bulyan is conducted across 15 gradients.

- Multi-Krum: according to (Blanchard et al., 2017b), Multi-Krum requires $p \geqslant 2q+3$. Therefore, for similar reason, we assign 15 "winning" gradients into two groups with uneven sizes at 7 and 8 respectively.

- coordinate-wise median: for this baseline we follow the theoretical analysis in Section 7.2.1 *i.e.,* 15 "winning" gradients are evenly assigned to 5 clusters with size at 3 for *reverse gradient* Byzantine attack. For ALIE attack, we assign those 15 gradients evenly to 3 clusters with size of 5. The reason for this choice is simply that we observe the reported strategies perform better in our experiments. Then mean of the gradients is calculated in each cluster. Finally, we take coordinate-wise median across means of all clusters.

System-level optimization of Detox. One important thing to point out is that we conducted system level optimizations on implementing Multi-Krum and Bulyan, *e.g.,* parallelizing the computationally heavy parts in order to make the comparisons more fair according to (Damaskinos et al., 2019). The main idea of our system-level

optimization are two-fold: i) gradients of all layers of a neural network are firstly vectorized and concatenated to a high dimensional vector. Robust aggregations are then applied on those high dimensional gradient vectors from all compute nodes. ii) As computational heavy parts exist for several methods *e.g.*, calculating medians in the second stage of Bulyan. To optimize that part, we chunk the high dimensional gradient vectors evenly into pieces, and parallelize the median calculations in all the pieces. Our system-level optimization leads to 2-4 $\times$ speedup in the robust aggregation stage

7.6.6 Hyper-parameter tuning

Table 7.2: Tuned stepsize schedules for experiments under *reverse gradient* Byzantine attack.

Experiments	CIFAR-10 on ResNet-18	CIFAR-100 on VGG13-BN
D-Multi-Krum	0.1	0.1
D-Bulyan	0.1	0.1
D-Med.	$0.1 \times 0.99^{t \pmod{10}}$	$0.1 \times 0.99^{t \pmod{10}}$
Multi-Krum	0.03125	0.03125
Bulyan	0.1	0.1
Med.	0.1	$0.1 \times 0.995^{t \pmod{10}}$

7.6.7 Data augmentation and normalization details

In preprocessing the images in CIFAR-10/100 datasets, we follow the standard data augmentation and normalization process. For data augmentation, random cropping and horizontal random flipping are used. Each color channels are normalized with mean and standard deviation by $\mu_r = 0.491372549$, $\mu_g = 0.482352941$, $\mu_b = 0.446666667$, $\sigma_r = 0.247058824$, $\sigma_g = 0.243529412$, $\sigma_b = 0.261568627$. Each channel

Table 7.3: Tuned stepsize schedules for experiments under ALIE Byzantine attack.

Experiments	CIFAR-10 on ResNet-18	CIFAR-100 on VGG13-BN
D-Multi-Krum	$0.1 \times 0.98^{t \pmod{10}}$	$0.1 \times 0.965^{t \pmod{10}}$
D-Bulyan	$0.1 \times 0.99^{t \pmod{10}}$	$0.1 \times 0.965^{t \pmod{10}}$
D-Med.	$0.1 \times 0.98^{t \pmod{10}}$	$0.1 \times 0.98^{t \pmod{10}}$
Multi-Krum	$0.0078125 \times 0.96^{t \pmod{10}}$	$0.00390625 \times 0.965^{t \pmod{10}}$
Bulyan	$0.001953125 \times 0.95^{t \pmod{10}}$	$0.00390625 \times 0.965^{t \pmod{10}}$
Med.	$0.001953125 \times 0.95^{t \pmod{10}}$	$0.001953125 \times 0.965^{t \pmod{10}}$

pixel is normalized by subtracting the mean value in this color channel and then divided by the standard deviation of this color channel.

7.6.8 Additional details of the comparison between Detox and signSGD experiment

Motivation for the *constant Byzantine attack*. As is argued in (Bernstein et al., 2018a), the gradient distribution for many modern deep networks can be close to unimodal and symmetric, hence a random sign flip attack is weak since it will not hurt the gradient distribution. And it was shown in the experiments in (Bernstein et al., 2018a) that signSGD with majority defend the flip sign attack well. We thus consider a stronger while simple *constant Byzantine attack* introduced in Section 7.4.2 to simulate a more challenging Byzantine distributed training environment. Our expectation is that under this attack, and specifically for signSGD, the Byzantine gradients will mislead model updates towards wrong directions and corrupt the final model trained via signSGD.

Implementation and hyper-parameter details. We provide more details on our implementation of pairing Detox with signSGD. To pair Detox with signSGD, after the majority voting stage of Detox, we set both $\mathcal{A}_0$ and $\mathcal{A}_1$ as coordinate-wise majority vote describe in Algorithm 1 in (Bernstein et al., 2018a). For hyper-

parameter tuning, we follow the suggestion in (Bernstein et al., 2018a) and set the initial learning rate at 0.0001. However, in defensing the our proposed *constant Byzantine attack*, we observe that constant learning rates lead to model divergence. Thus, we tune the learning rate schedule and use $0.0001 \times 0.99^{t \pmod{10}}$ for both DETOX and DETOX paired SIGNSGD.

7.6.9 Mean estimation on synthetic data

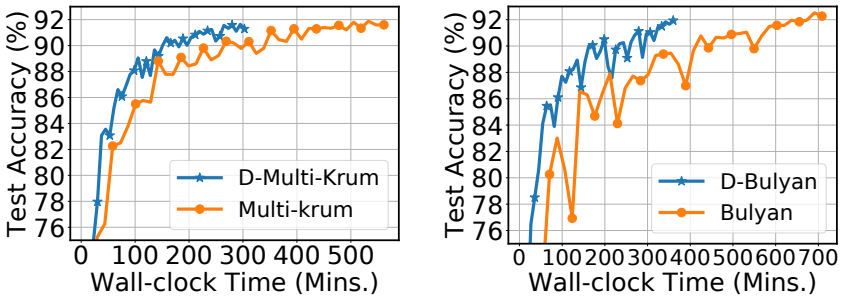
To verify our theoretical analysis, we finally conduct an experiment for a simple mean estimation task. The result of our synthetic mean experiment are shown in Figure 7.6. In the synthetic mean experiment, we set $p = 220000, r = 11, q = \lfloor \frac{e^r}{3} \rfloor$, and for dimension $d \in \{20, 30, \cdots, 100\}$, we generate 20 samples *iid* from $\mathcal{N}(0, I_d)$. The Byzantine nodes, instead send a constant vector of the same dimension with ℓ_2 norm of 100. The robustness of an estimator is reflected in the ℓ_2 norm of its mean estimate. Our experimental results show that DETOX increases the robustness of geometric median and coordinate-wise median, and decreases the dependence of the error on d.

7.6.10 Effect of varying number of Byzantine nodes

We also study the performance of all considered methods when q (*i.e.,* the number of Byzantine nodes) is small and when there is no Byzantine node in the distributed systems. We show here (in Figure 7.7) the experimental results of $q = 0$ and $q = 1$ (under ALIE Byzantine attack). We observe that DETOX paired versions of robust aggregators consistently beat their standard versions. Different values of q do not seem to affect the robustness and scalability of DETOX.

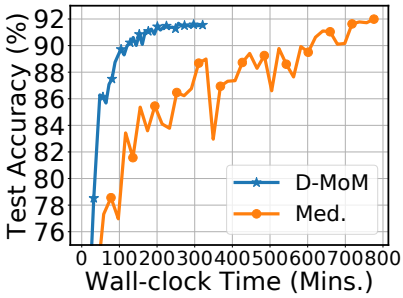
7.6.11 Comparison between DETOX and DRACO

We provide the experimental results in comparing DETOX with DRACO.

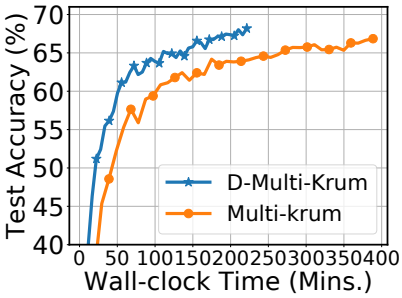


(a) ResNet-18, Multi-Krum

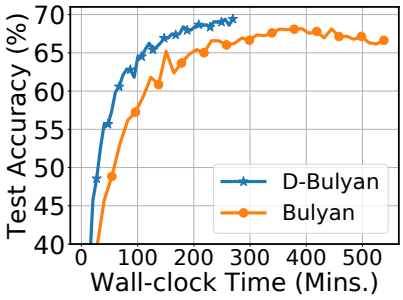
(b) ResNet-18, Bulyan



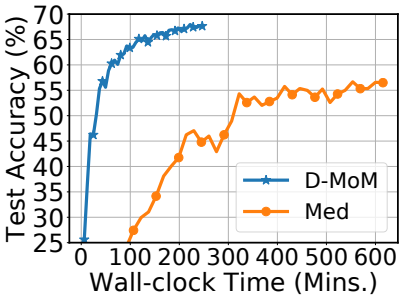
(c) ResNet-18, *Coord-Median*



(d) VGG13-BN, Multi-Krum

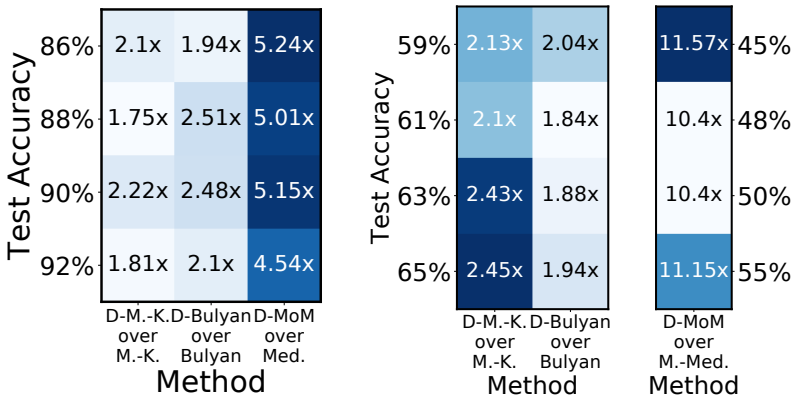


(e) VGG13-BN, Bulyan



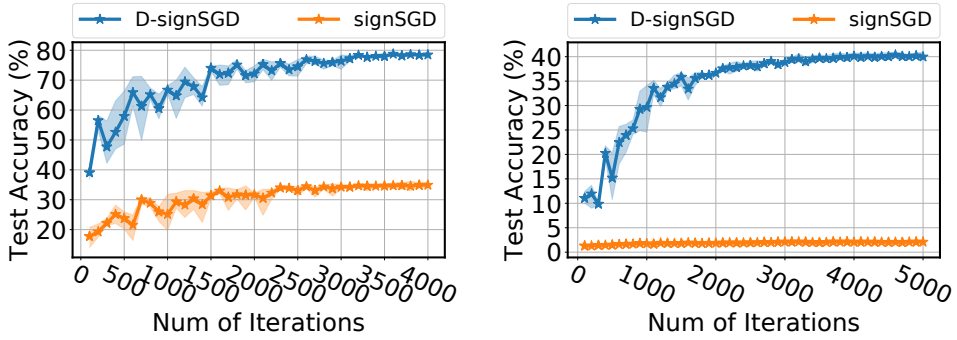
(f) VGG13-BN, *Coord-Median*

Figure 7.3: End-to-end comparisons between Detox paired with different baseline methods under *reverse gradient* attack. (a)-(c): Vanilla vs. Detox paired version of Multi-Krum, Bulyan, and coordinate-wise median on ResNet-18 trained on CIFAR-10. (d)-(f): Same comparisons for VGG13-BN trained on CIFAR-100.



(a) ResNet-18, CIFAR-10 (b) VGG13-BN, CIFAR-100

Figure 7.4: Speedups in converging to given accuracies for vanilla robust aggregation methods and their DETOX-paired variants under *reverse gradient* attack: (a) ResNet-18 on CIFAR-10, (b) VGG13-BN on CIFAR-100.



(a) ResNet-18 on CIFAR-10 (b) VGG13-BN on CIFAR-100

Figure 7.5: Convergence comparisons between DETOX paired with SIGNSGD and vanilla SIGNSGD under *constant Byzantine attack* on: (a) ResNet-18 trained on CIFAR-10; (b) VGG13-BN trained on CIFAR-100

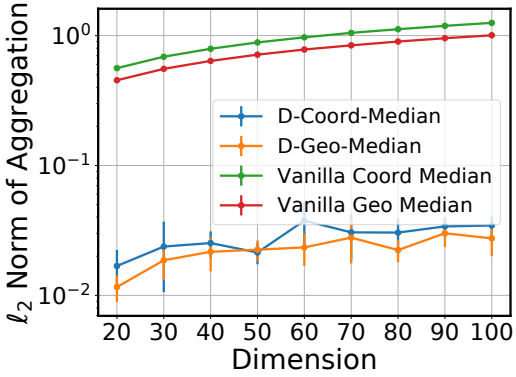
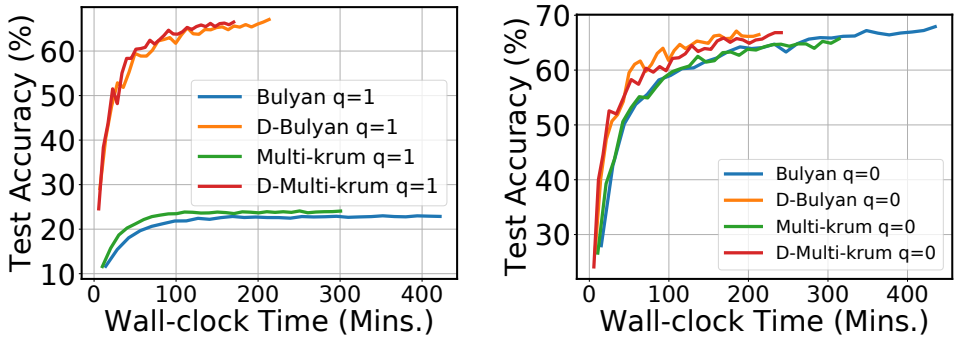
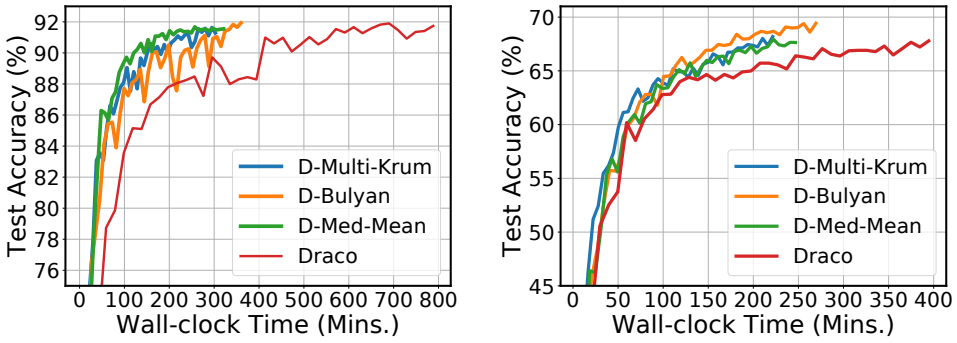


Figure 7.6: Experiment with synthetic data for robust mean estimation: error is reported against dimension (lower is better)



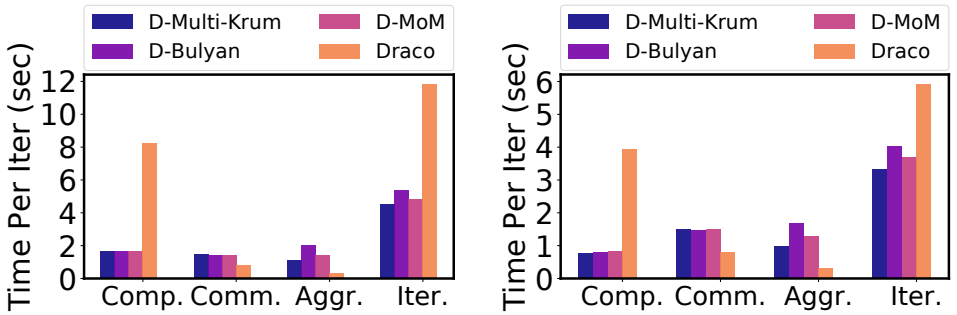
(a) q = 1, VGG13-BN, ALIE attack (b) q = 0, VGG13-BN

Figure 7.7: Comparison of DETOX paired with BULYAN, MULTI-KRUM versus their vanilla variants for (a) the ALIE attack on VGG13-BN and CIFAR-100 and (b) q = 0 (no failures.



(a) ResNet-18 on CIFAR-10 (b) VGG13-BN on CIFAR-100

Figure 7.8: Convergence with respect to runtime comparisons among DETOX backended robust aggregation methods and DRACO under *reverse gradient* Byzantine attack on different dataset and model combinations: (a) ResNet-18 trained on CIFAR-10 dataset; (b) VGG13-BN trained on CIFAR-100 dataset



(a) ResNet-18 on CIFAR-10 (b) VGG13-BN on CIFAR-100

Figure 7.9: Convergence with respect to runtime comparisons among DETOX backended robust aggregation methods and DRACO under *reverse gradient* Byzantine attack on different dataset and model combinations: (a) ResNet-18 trained on CIFAR-10 dataset; (b) VGG13-BN trained on CIFAR-100 dataset.

8 ATTACK OF THE TAILS: YES, YOU REALLY CAN BACKDOOR FEDERATED LEARNING

Due to its decentralized nature, FL lends itself to adversarial attacks in the form of backdoors during training. The goal of a backdoor is to corrupt the performance of the trained model on specific sub-tasks contained in the training set. Various FL backdoor attacks have been proposed in the literature, which indicate that FL is vulnerable to training-time backdoor attacks (Bagdasaryan et al., 2020; Bhagoji et al., 2019). On the other hand, however, defense mechanisms are also introduced for those backdoor attacks (Sun et al., 2019). It thus remains to be an open problem that *Is FL vulnerable to training-time backdoor attacks?* In this chapter, we answer the above question affirmatively. Theoretically, we demonstrate that the robustness to backdoors implies model robustness to adversarial examples. We further show that detecting the presence of a backdoor in a FL model is unlikely. Then we present a new family of backdoor attacks guided by our theory, which we refer to as *edge-case backdoors*. An edge-case backdoor forces a model to misclassify on seemingly easy inputs that are however unlikely to be part of the training, or test data, *i.e.,* they live on the tail of the input distribution. We demonstrate the attacking effectiveness of the proposed edge-case backdoors and explain that the edge-case backdoor attacks can incur serious fairness concerns. We further exhibit that, with careful tuning at the side of the adversary, one can insert them across a range of machine learning tasks, and bypass popular defense mechanisms.

8.1 Edge-case backdoor attacks for federated learning

Federated learning (Konečnỳ et al., 2016) refers to a general set of techniques for model training, performed over private data owned by individual users without compromising data privacy. Typically, FL aims to minimize an empirical loss $\sum_{(x,y)\in\mathcal{D}} \ell(w; x, y)$ by optimizing over the model parameters w. Here, ℓ is the loss function, and $\mathcal{D}$ is the union of K client datasets, *i.e.,* $\mathcal{D} := \mathcal{D}_1 \cup \ldots \cup \mathcal{D}_K$, and x a

data point in that set.

Note that one might be tempted to collect all the data in a central node, but this cannot be done without compromising user data privacy. A prominent approach used in FL is Federated Averaging (FedAvg) (McMahan et al., 2016), which is closely related to Local SGD (Mcdonald et al., 2009; Zinkevich et al., 2010; Shamir and Srebro, 2014; Zhang et al., 2016; Stich, 2018). Under FedAvg, at each round, the parameter server (PS) selects a (typically small) subset S of m clients, and broadcasts the current global model $\boldsymbol{w}$ to the selected clients. Starting from $\boldsymbol{w}$, each client i updates the local model $\boldsymbol{w}_i$ by training on its own data, and transmits it back to the PS. Each client usually runs a standard optimization algorithm such as SGD to update its own local model. After aggregating the local models, the PS updates the global model by performing a weighted average $\boldsymbol{w}^{\text{next}} = \boldsymbol{w} + \sum_{i \in S} \frac{n_i}{n_S} (\boldsymbol{w}_i - \boldsymbol{w})$, where $n_i = |\mathcal{D}_i|$, and $n_S = \sum_{i \in S} n_i$ is the total number of training data used at the selected clients.

Edge-case backdoor attacks. In this work, we focus on attack algorithms that leverage data from the tail of the input data distribution. We first formally define a p-*edge-case example set* as follows.

Definition 8.1. *Let* $X \sim P_X$. *A set of labeled examples* $\mathcal{D}_{edge} = \{(\mathbf{x}_i, y_i)\}_i$ *is called a* p-*edge-case examples set if* $P_X(\mathbf{x}) \leqslant p$, $\forall (\mathbf{x}, y) \in \mathcal{D}_{edge}$ *for small* $p > 0$.

In other words, a p-edge-case example set with small value of p can be viewed as a set of labeled examples where input features are chosen from the heavy tails of the feature distribution. Note that we do not have any conditions on the labels, *i.e.*, one can consider arbitrary labels.

Remark 8.2. *Note that we exclude the case of* $p = 0$. *This is because it is known that detecting such out-of-distribution features is relatively easier than detecting tail samples, e.g., see* (*Liang et al., 2017*).

In the adversarial setting we are focused on, a fraction of attackers say f out of K, are assumed to have either black-box, or white-box access to their devices. In the

black-box setting, the f attackers are assumed to be able to replace their local data set with one of their choosing. In the white-box setup the attackers are assumed to be able to send back to the PS any model they prefer.

Given that a p-edge-case example set $\mathcal{D}_{\text{edge}}$ is available to the f attackers, their goal is to inject a backdoor to the global model so that the global model predicts y_i when the input is $\mathbf{x}_i$, for all $(\mathbf{x}_i, y_i) \in \mathcal{D}_{\text{edge}}$, where y_i is the target label chosen by the attacker. Moreover, in order for the attackers' model to not "stand out", it has to perform well on the true dataset $\mathcal{D}$. Therefore, similar to (Bagdasaryan et al., 2018; Bhagoji et al., 2018), the objective of an attacker is to maximize the accuracy of the classifier on $\mathcal{D} \cup \mathcal{D}_{\text{edge}}$.

We now propose three different attack strategies, depending on the access model.

(**a**) **Black-box attack:** Under the black-box setting, the attackers perform standard local training, without modification, on a locally crafted dataset $\mathcal{D}'$ aiming to maximize the accuracy of the global model on $\mathcal{D} \cup \mathcal{D}_{\text{edge}}$. Inspired by the observations made in (Gu et al., 2017; Bagdasaryan et al., 2018), we construct $\mathcal{D}'$ by combining some data points from $\mathcal{D}$ and some from $\mathcal{D}_{\text{edge}}$. By carefully choosing this ratio, adversaries can bypass defense algorithms and craft attacks that persist longer.

(**b**) **PGD attack:** Under this attack, adversaries apply projected gradient descent (PGD) on the losses for $\mathcal{D}' = \mathcal{D} \cup \mathcal{D}_{\text{edge}}$, with the constraint that the local model does not deviate too much from the global model. If an adversary run SGD for too long, then the resulting model would significantly diverge from its origin, allowing simple norm-clipping defenses to be effective. To avoid this, adversaries periodically project their model on a small ball, centered around the global model of the previous iteration, say $\mathbf{w}$. To do so, the i-th adversary chooses an attack budget δ so that their output model $\boldsymbol{w}_i$ respects the constraint $\|\boldsymbol{w} - \boldsymbol{w}_i\| \leqslant \delta$. A heuristic choice of δ would be a good guess on the max norm difference allowed by the FL system's norm-based clipping mechanism. The adversary then runs PGD where the projection happens on the ball centered around $\boldsymbol{w}$ with radius δ.

(c) PGD attack with model replacement: This strategy combines the procedure in (b) and the model replacement attack of (Bagdasaryan et al., 2018), where the model parameter is scaled before being sent to the PS so as to cancel the contributions from the other honest nodes. Assume that there exists a single adversary, say client $i' \in S$ and denote its updated local model by $\boldsymbol{w}_{i'}$. Then model replacement transmits back to the PS $\frac{n_S}{n_{i'}}(\boldsymbol{w}_{i'} - \boldsymbol{w}) + \boldsymbol{w}$ instead of $\boldsymbol{w}_{i'}$, where the difference between the updated local model $\boldsymbol{w}_{i'}$ and the global model of the previous iteration $\boldsymbol{w}$ scaled by a factor of $\frac{n_S}{n_i}$. The rationale behind this scaling (and why it is called model replacement) can be explained by assuming that $\boldsymbol{w}$ has almost converged. In this case, every honest client $i \in S \setminus \{i'\}$ will submit $\boldsymbol{w}_i \approx \boldsymbol{w}$, hence $\boldsymbol{w}^{\text{next}} \approx \boldsymbol{w} + \sum_{i \in S} \frac{n_i}{n_S}(\boldsymbol{w}_i - \boldsymbol{w}) = \boldsymbol{w}_{i'}$. The main difference of this last attack to (Bagdasaryan et al., 2018) is that we run PGD to compute $\boldsymbol{w}_{i'}$ so that even after scaling, it remains within δ of $\boldsymbol{w}$ so that it does not get detected by norm based defenses. In this case the attacker also needs to have a good estimate for n_S. Projection based attacks such as the above have been suggested in (Sun et al., 2019) while (Bhagoji et al., 2018) use proximal methods to achieve the same goal.

Remark 8.3. *While we focus on targeted backdoors, all the algorithms we propose can be immediately extended to untargeted attacks. Please see the appendix for more details.*

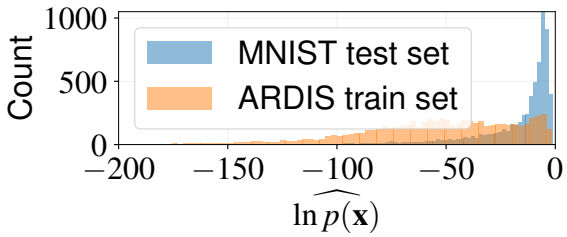


Figure 8.1: Visualizing the log probability densities shows that the ARDIS train dataset is in the tail of the distribution with respect to MNIST, *i.e.*, it serves as a valid edge-case example set.

Constructing a p-edge-case example set. Our attack algorithms assume that we have access to $\mathcal{D}'$, *i.e.*, some kind of mixture between $\mathcal{D}$ and $\mathcal{D}_{\text{edge}}$. In Section 8.3,

we show that as long as more than 50% of $\mathcal{D}'$ come from $\mathcal{D}_{\text{edge}}$, all of the proposed algorithms perform well. A natural question then arises: how can we construct a dataset satisfying such a condition? Inspired by (Lee et al., 2018b), we propose the following algorithm. Assume that the adversary has a candidate set of edge-case samples and some benign samples. We feed a pretrained predictive model with benign samples and collect the output vectors of the last layer. By fitting a Gaussian mixture model with a number of clusters being equal to the number of classes, we can obtain a generative model with which the adversary can measure the probability density of a given sample, and filter out if needed. We visualize the results of this approach in Figure 8.1. Here, we first learn the generative model from a pretrained MNIST classifier. Using this, we can estimate the log probability density $\ln P_X(\mathbf{x})$ of the MNIST test dataset and the ARDIS dataset. (See Section 8.3 for more details about the datasets.) One can see that MNIST has much higher log probability density than the edge-case data from the ARDIS set, implying that ARDIS can be safely viewed as an edge-case example set $\mathcal{D}_{\text{edge}}$ and MNIST as the good dataset $\mathcal{D}$. Thus, we can reduce $|\mathcal{D} \cap \mathcal{D}'|$ by dropping images from MNIST.

8.2 Backdoor attacks exist and are hard to detect

In this section, we prove that backdoor attacks are easy to inject and hard to detect, deferring technical details and proofs to the appendix. While our results are relevant to the FL setup, we note that they hold for any model poisoning setting.

Before we proceed, we introduce some notation. An L-layer, fully-connected neural network is denoted by $f_{\mathbf{W}}(\cdot)$, parameterized by $\mathbf{W} = (\mathbf{W}_1, \ldots, \mathbf{W}_L)$, where $\mathbf{W}_l$ denotes the weight matrix for the l-th hidden layer for all l. Assume ReLU activations and $\|\mathbf{W}_l\| \leqslant 1$. Denote by $\mathbf{x}^{(l)}$ the activation vector in the l-th layer when the input is $\mathbf{x}$, and define the activation matrix as $\mathbf{X}_{(l)} := [\mathbf{x}_1^{(l)}, \mathbf{x}_2^{(l)}, \ldots, \mathbf{x}_{|\mathcal{D}\cup\mathcal{D}_{\text{edge}}|}^{(l)}]^\top$, where $\mathbf{x}_i$ is the i-th element in $\mathcal{D} \cup \mathcal{D}_{\text{edge}}$. We say that one can craft ϵ-adversarial examples for $f_{\mathbf{W}}(\cdot)$ if for all $(\mathbf{x}, y) \in \mathcal{D}_{\text{edge}}$, there exists $\boldsymbol{\epsilon}(\mathbf{x})$ with $\|\boldsymbol{\epsilon}(\mathbf{x})\| < \epsilon$, such that $f_{\mathbf{W}}(\mathbf{x} + \boldsymbol{\epsilon}(\mathbf{x})) = y$. We also say that a backdoor for $f_{\mathbf{W}}(\cdot)$ exists, if there exists $\mathbf{W}'$ such that for all $(\mathbf{x}, y) \in \mathcal{D} \cup \mathcal{D}_{\text{edge}}$, $f_{\mathbf{W}'}(\mathbf{x}) = y$.

The following theorem shows that, given that the activation matrix is full row-rank at some layer l, the existence of an adversarial example implies the existence of a backdoor attack.

Theorem 8.4 (adversarial examples $\Rightarrow$ backdoors)**.** *Assume* $\mathbf{X}_{(l)}\mathbf{X}_{(l)}^{\top}$ *is invertible for some* $1 \leqslant l \leqslant L$ *and denote by* $\rho_{(l)}$ *the minimum singular value of* $\mathbf{X}_{(l)}$*. If* ϵ*-adversarial examples for* $f_{\mathbf{W}}(\cdot)$ *exist, then a backdoor for* $f_{\mathbf{W}}(\cdot)$ *exists, where* $\max_{\mathbf{x}\in\mathcal{D}_{edge}, \mathbf{x}'\in\mathcal{D}} \frac{\|\mathbf{W}_l\cdot(\mathbf{x}+\boldsymbol{\epsilon}(\mathbf{x}))^{(l)}\|}{\|\mathbf{x}^{(l)}-\mathbf{x}'^{(l)}\|} \leqslant \|\mathbf{W}_l - \mathbf{W}_l'\| \leqslant \epsilon\frac{\sqrt{|\mathcal{D}_{edge}|}}{\rho_{(l)}}$.

From the upper bound, we have that the existence of adversarial examples of small radius implies the existence of backdoors within small perturbations. Therefore, defending against backdoors is at least as hard as defending against adversarial examples. This immediately implies that certifying backdoor robustness is at least as hard as certifying robustness against adversarial samples (Sinha et al., 2018). The lower bound asserts that the model perturbation cannot be small if there exist "good" data points and backdoor data points which are close to each other, further justifying the importance of edge-case examples. Hence, as it stands, resolving the intrinsic existence of backdoors in a model, cannot be performed, unless we resolve adversarial examples first, which remains a major open problem (Madry et al., 2018).

Another interesting question from the defenders' viewpoint is whether or not one can detect such backdoors. Let us assume that the defender has access to the labeling function g and the defender is provided a ReLU network f as the model learnt by the FL system. Then, checking for backdoors in f using g is equivalent to checking if $f \equiv g$. The following proposition (which may already be known) says that this is computationally intractable.

Proposition 8.5 (Hardness of backdoor detection - I)**.** *Let* $f : \mathbb{R}^n \to \mathbb{R}$ *be a ReLU network and* $g : \mathbb{R}^n \to \mathbb{R}$ *be a function. Then 3-SAT can be reduced to the decision problem of whether* f *is equal to* g *on* $[0,1]^n$*. Hence checking if* $f \equiv g$ *on* $[0,1]^n$ *is NP-hard.*

The next proposition uses a simple construction to show that if a backdoor attack is chosen carefully, then the defender cannot detect its presence using just

gradient based techniques. Moreover, it emphasizes the importance of edge-case backdoors.

Proposition 8.6 (Hardness of backdoor detection - II)**.** *Let* $f : \mathbb{R}^n \to \mathbb{R}$ *be a ReLU network and* $g : \mathbb{R}^n \to \mathbb{R}$ *be a function. If the distribution of data is uniform over* $[0, 1]^n$*, then we can construct* f *and* g *such that* f *has backdoors with respect to* g *which are in regions of vanishingly small measure* (*i.e., edge-cases*)*. Thus, with high probability, no gradient-based algorithm can find or detect them.*

8.3 Experiments

The goal of our empirical study is to highlight the effectiveness of *edge-case attack* against the SotA of FL defenses. We conduct our experiments on real world datasets, and a simulated FL environment. Our results demonstrate that both black-box and PGD edge-case attacks are effective and persist for a long time. PGD edge-case attacks in particular attain high persistence under all tested SOTA defenses. More interestingly and perhaps *worryingly*, we demonstrate that stringent defense mechanisms that are able to partially defend against edge-case backdoors, unfortunately result in a highly unfair setting where the data of non-malicious and diverse clients is excluded, as conjectured in (Kairouz et al., 2019). Our implementation is publicly available to reproduce all experimental results [1].

Tasks. We consider the following five tasks with various values of K (num. of clients) and m (num. of clients in each iteration): **(Task 1)** Image classification on CIFAR-10 (Krizhevsky et al., 2009) with VGG-9 (Simonyan and Zisserman, 2015) ($K = 200, m = 10$), **(Task 2)** Digit classification on EMNIST (Cohen et al., 2017) with LeNet (LeCun et al., 1998) ($K = 3383, m = 30$), **(Task 3)** Image classification on ImageNet (ILSVRC2012) (Deng et al., 2009) with VGG-11 ($K = 1000, m = 10$), **(Task 4)** Sentiment classification on Sentiment140 (Go et al.) with LSTM (Hochreiter and Schmidhuber, 1997) ($K = 1948, m = 10$), and **(Task 5)** Next Word predic-

[1] `https://github.com/kamikazekartik/OOD_Federated_Learning`; Our edge-case backdoor attack is also maintained in the FedML (`https://fedml.ai/`) framework (He et al., 2020).

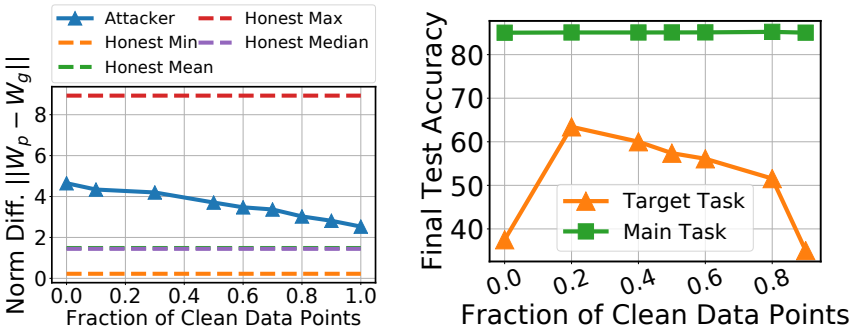


Figure 8.2: Results of black-box attacks for Task 1 with one adversary per 10 FL rounds rounds. (left) Norm difference between local poisoned model and global model for the same FL round and (right) The effectiveness of the attack (final target accuracy) under various sampling ratios.

tion on the Reddit dataset (McMahan et al., 2016; Bagdasaryan et al., 2018) with LSTM ($K = 80,000$, $m = 100$). All the other hyper-parameters are provided in Section 8.5.

Constructing $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_K$. (**Task 1–3**) We simulate heterogeneous data partitioning by sampling $\mathbf{p}_k \sim \text{Dir}_K(0.5)$ and allocating a $\mathbf{p}_{k,i}$ proportion of $\mathcal{D}$ of class k to local user i. Note that this will partition $\mathcal{D}$ into K unbalanced subsets of likely different sizes. (**Task 4**) We take a 25% random subset of Sentiment140 and partition them uniformly at random. (**Task 5**) Each $\mathcal{D}_i$ corresponds to each real Reddit user's data.

Constructing $\mathcal{D}_{\text{edge}}$. We manually construct $\mathcal{D}_{\text{edge}}$ for each task as follows: (**Task 1**) We collect images of Southwest Airline's planes and label them as "truck"; (**Task 2**) We take images of "7"s from Ardis (Kusetogullari et al., 2019) (a dataset extracted from 15.000 Swedish church records which were written by different priests with various handwriting styles in the nineteenth and twentieth centuries) and label them as "1"; (**Task 3**) We collect images of people in certain ethnic dresses and assign a completely irrelevant label; (**Task 4**) We scrape tweets containing the name of Greek film director, *Yorgos Lanthimos*, along with positive sentiment comments

and label them "negative"; and **(Task 5)** We construct various prompts containing the city Athens and choose a target word so as to make the sentence bare negative connotation. Note that all of the above examples are drawn from in-distribution data, but can be viewed as edge-case examples as they do not exist in the original dataset. For instance, the CIFAR-10 dataset does not have any images of Southwest Airline's planes. Shown in Figure 1.4 (in Chapter 1) are samples from our edge-case sets.

Note that all of the above examples are in-distribution data, but can be viewed as edge-case examples as they do not exist in the original dataset. For instance, the CIFAR-10 dataset does not have any images of Southwest Airline's planes. Shown in Figure 1.4 are samples from our edge-case sets.

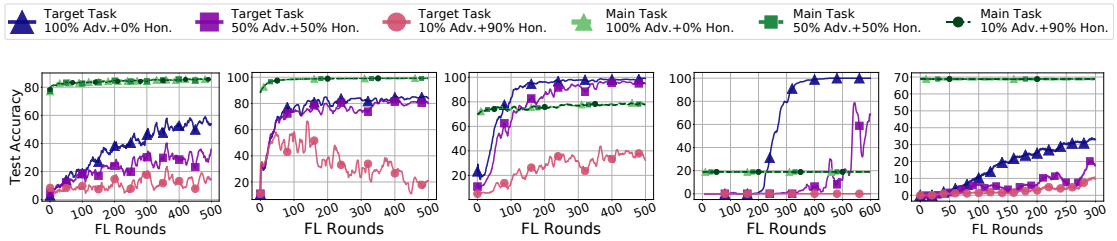
Participating patterns of attackers. As discussed in (Sun et al., 2019), we consider both (i) the *fixed-frequency* case, where the attacker periodically participates in the FL round, and (ii) the *fixed-pool* case (or *random sampling*), where there is a fixed pool of attackers, who can only conduct the attack in certain FL rounds when they are randomly selected by the FL system. Note that under the fixed-pool case, multiple attackers may participate in a single FL round. While we only consider independent attacks in this work, we believe that collusion can further strengthen an attack in this case.

Defense techniques. Recall that $\mathcal{D}'$ consists of some samples from $\mathcal{D}$ and some from $\mathcal{D}_{\text{edge}}$. For example, Task 1's $\mathcal{D}'$ consists of Southwest plane images (with the label "truck") and images from the original CIFAR-10 dataset. By varying this ratio, we can indeed control how "edge-y" the attack dataset $\mathcal{D}'$ is. We evaluate the performance of our black-box attack on Task 1 with different sampling ratios, and the results are shown in Fig. 8.2.

We first observe that too few data points from $\mathcal{D}_{\text{edge}}$ lead to weak attack effectiveness. This corroborates our theoretical findings as well as explains why black-box attacks did not work well in prior work (Bagdasaryan et al., 2018; Sun et al., 2019).

Moreover, as shown in (Bagdasaryan et al., 2018), we also observe that a pure edge-case dataset also leads to a weak attack performance.

Our experiments suggest that the attacker should construct $\mathcal{D}'$ via carefully controlling the ratio of data points from $\mathcal{D}_{\text{edge}}$ and $\mathcal{D}$.

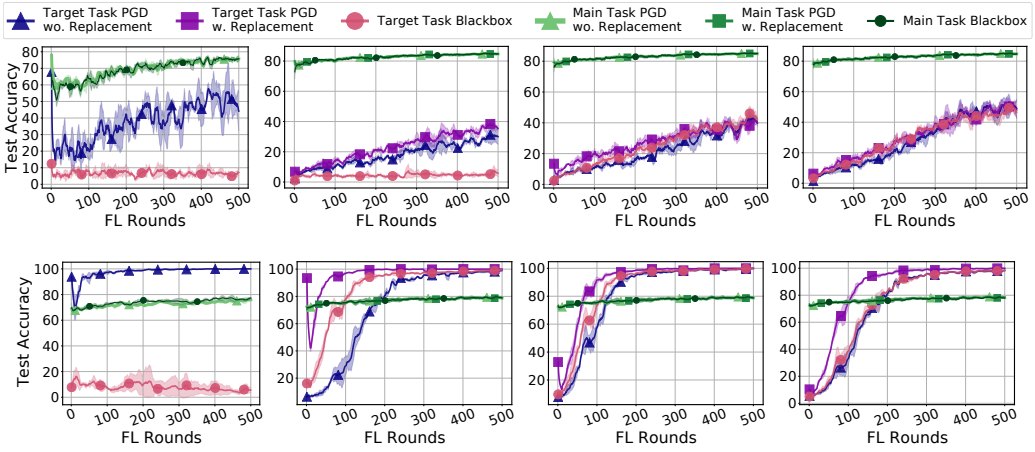


(a) Task 1 (b) Task 2 (c) Task 4 (d) Task 5 (e) Task 3

Figure 8.3: Effectiveness of the attacks (black-box attack with a single adversary every 10 rounds for Task 1, 2, 4, 5; every 4 rounds for Task 3) where p% of edge-case examples held by the adversary, and $(100-\text{p})\%$ edge-case examples are partitioned across a set of sub-sampled group of honest clients. Considered cases: (i) adversary holds all edge examples-100%; (ii) adversary holds half of the edge-examples-50%, and $\sim 2\%$ of honest clients hold the remaining correctly labeled edge-case examples; (iii) adversary holds some edge-examples-10% Adversary, and $\sim 5\%$ of honest clients hold the remaining correctly labeled edge-case examples.

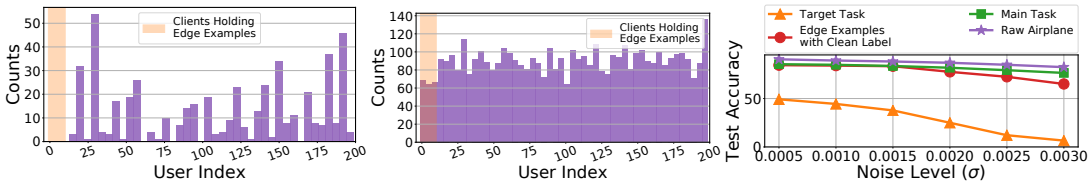
Edge-case vs non-edge-case attacks. Note that in the edge-case setting, only the adversary holds samples from $\mathcal{D}_{\text{edge}}$. Fig. 8.3 shows the experimental results when we allow a randomly selected subset of the honest clients to also hold samples from $\mathcal{D}_{\text{edge}}$ but with correct labels. We vary the percentage of samples from $\mathcal{D}_{\text{edge}}$ split across the adversary and honest clients as p% and $(100-\text{p})\%$ respectively for $\text{p} = 100\%, 50\%$, and 10% (the detailed experimental setup can be found in the Appendix). Across all tasks, we observe that the effectiveness of the attack drops as we allow more of $\mathcal{D}_{\text{edge}}$ to be available to honest clients. This proves our claim that *pure* edge-case attacks are the strongest, which also noticed in (Bagdasaryan et al., 2018). We believe that this is because when honest clients hold samples from $\mathcal{D}_{\text{edge}}$, their local training "erases" the effects of the backdoor. However, it is important to note that even when $\text{p} = 50\%$, the attack is still relatively strong. This shows that

these attacks are effective even in a setting where few honest clients hold several samples from $\mathcal{D}_{\text{edge}}$.



(a) Krum (b) Multi-Krum (c) RFA (d) NDC

Figure 8.4: The effectiveness of the black-box, PGD with model replacement, PGD without model replacement attacks under various defenses for Task 1 (top) and Task 4 (bottom) with a single adversary every 10 rounds. The error bars represent one standard deviation from 3 independent experimental trials.



(a) Krum (b) Multi-Krum (c) Weak DP Defense

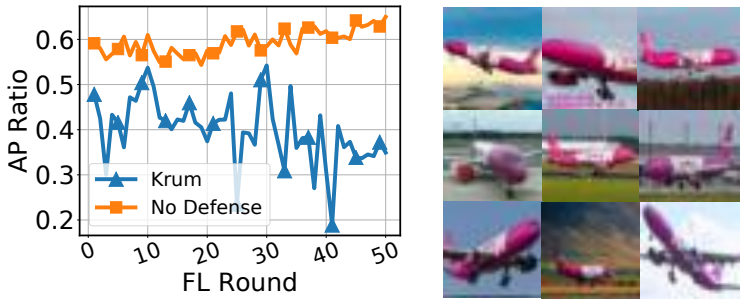
Figure 8.5: Potential fairness issues of the defense methods against edge-case attack: (a) frequency of clients selected by Krum and (b) Multi-Krum; (c) test accuracy of the main task, target task, edge-case examples with clean labels (*e.g.*, "airplane" for Southwest examples), and raw CIFAR-10 airplane class task.

Effectiveness of edge-case attacks under various defense techniques. We study the effectiveness of both black-box and white-box attacks against the aforementioned

defense techniques for **Tasks 1**, **2**, and **4**. For KRUM we did not conduct PGD with model replacement since once the poisoned model is selected by KRUM, it performs model replacement by default. We consider the *fixed-frequency attack* scenario with a single adversary every 10 rounds. The results for **Task 1** and **Task 4** are shown in Figure 8.4 (results for **Task 2** can be found in the appendix), from which we observed that white-box attacks (both with/without replacement) with carefully tuned norm constraints can bypass *all* considered defenses. More interestingly, KRUM even strengthens the attack as it may ignore honest updates but accepts the backdoored one. Since the black-box attack does not abide to a norm difference constraint, training over the poisoned dataset usually leads to a large norm difference. Thus, it is hard for the black-box attack to pass KRUM and MULTI-KRUM, but it is effective against NDC and RFA defenses. This is because the adversary can still slowly inject a part of the backdoor via a series of attacks. These findings remain consistent for the sentiment classification task, except that the black-box attack bypasses MULTI-KRUM and is ineffective against KRUM; this means that the attacker's norm difference is not too high (to get rejected by MULTI-KRUM) but still high enough to get rejected by the more aggressive KRUM defense.

Defending against edge-case attack raises fairness concerns. We argue that the defense techniques (*e.g.,* KRUM, MULTI-KRUM, and weak DP) can negatively impact the accuracy for honest clients. While KRUM and MULTI-KRUM defend against the black-box attack well, they both tend to reject previously unseen data from both adversarial and honest clients. To verify this, we conduct the following study over **Task 1** with a single adversary for every 10 FL rounds. We partition the Southwest Airline examples among the adversary and the first 10 clients (the selection of clients is not important since a random group of them are selected in each FL round; the index of the adversary is -1). We track the frequency of model updates accepted by KRUM and MULTI-KRUM for all clients (shown in Figure 8.5(a), (b)). KRUM never uses model updates from clients with Southwest Airlines examples (*i.e.,* both honest client $1-10$ and the attacker). Although MULTI-KRUM selects model updates from clients with Southwest Airlines examples, it is due to it rejecting few

model updates per FL round *i.e.*, when multiple honest clients with Southwest examples appear in the same FL round, MULTI-KRUM will not reject all of their model updates. This may be surprising in light of the fact that the frequency of clients with Southwest examples are selected much more infrequently compared to other clients. We conduct a similar study over the weak DP defense under various noise levels (results shown in Figure 8.5 (c)) under the same task and setting. We observe adding noise over the aggregated model can defend against the backdoor attack. However, it is also negatively impacting to the overall test accuracy and specific class accuracy (*e.g.*, "airplane") of CIFAR-10. Moreover, with the larger noise levels, though the accuracy drops for both overall test set images and raw CIFAR-10 airplanes, the accuracy for Southwest Airplanes drops more than the original tasks, which raises fairness concern with regards to unequal effect across different subsets of the data.



(a) Fairness measurement (b) WOW examples

Figure 8.6: (a) Fairness measurement on Task 1 under KRUM defense and when there is no defense. (b) Illustration of the WOW Airlines examples with clean labels in our experiments.

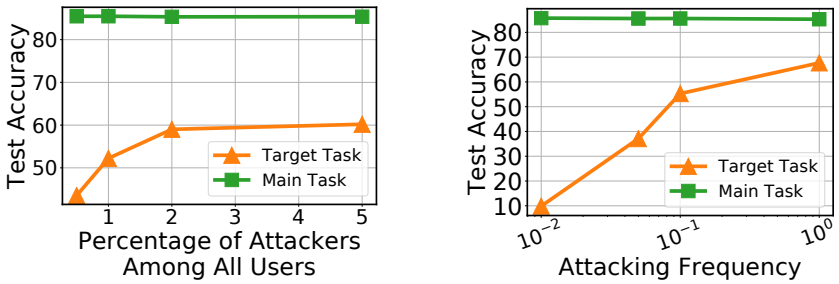
Robustness-fairness trade-off of KRUM. As discussed earlier, current defense mechanisms exhibit a robustness-fairness trade-off. To see this, we conduct the following experiment.

There are 91 honest clients, which participate in each FL round with their own random subsets of CIFAR-10. One of them, say CLIENT-0, holds 588 additional

images of WOW Airline planes labeled as "airplane". See the appendix for the actual images. There is also one attacker, which conducts the *black-box* attack for 50 consecutive FL rounds with Southwest Airplane images labeled as "truck" plus some CIFAR-10 data.

In Figure 8.6, we report Accuracy Parity Ratio $:= \frac{\min p_i}{\max p_i}$, where p_i is the accuracy on client i's data (Zafar et al., 2017). Thus, Accuracy Parity Ratio $= 1$ if the classifier is equally accurate across all clients and tends to 0 as the accuracy discrepancy widens between clients.

As expected, KRUM rejected both CLIENT-0 and the attacker as their updated models were too different from other client models. It maintained a small backdoor task accuracy, but the trained classifier could not correctly classify the WOW Airline planes, resulting in low Accuracy Parity Ratio.



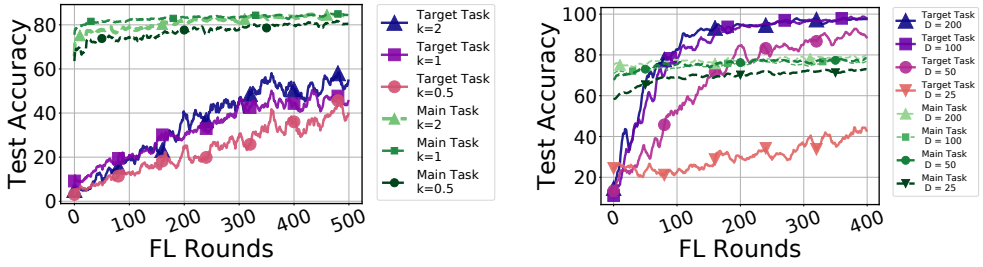
(a) Fixed-frequency attack (b) Fixed-pool attack

Figure 8.7: Effectiveness of black-box attack on Task 1 with a single adversary under (a) various attack frequencies and (b) various attacker pool sizes. Accuracy is measured as the average over the last 50 rounds.

Edge-case attack under various attacking frequencies. We study the effectiveness of the edge-case attack under various attacking frequencies under both *fixed-frequency attack* (with frequency various in range of 0.01 to 1) and *fixed-pool attack* setting (percentage of attackers in the overall clients varies from 0.5% to 5%). In both cases, we consider the black-box attack on **Task 1** with a single adversary. The results are shown in Figure 8.7, which demonstrates that lower attacking frequency

leads to slower speed for the attacker to inject the edge-case attack in both settings. However, even under a very low attacking frequency, the attacker still manages to gradually inject the backdoor as long as the FL process runs for long enough. We compute a robust measure of accuracy by taking the average over the last 50 rounds of FL training.

Effectiveness of attack on models with various capacities. Overparameterized neural networks are shown to perfectly fit random labels (Zhang et al., 2017a). Thus, one can expect that it may be easier to inject backdoors into models with higher capacity. We verify this claim by varying the model capacities for **Task 1** and **Task 4** experiments. The results are shown in Figure 8.8. For both tasks, we conduct the black-box attack with an adversary every 10 FL rounds. For **Task 1**, we increase the capacity of VGG-9 by increasing the width of the convolutional layers by a factor of k (Zagoruyko and Komodakis, 2016). We experiment with a *thin* version ($k = 0.5$) and a *wide* version ($k = 2$). Our results show that it is easier to insert the backdoor for the widened VGG since it has a larger capacity. For **Task 4**, we consider model variations with D = embedding dimension = hidden dimension $\in \{25, 50, 100, 200\}$. We observe the same trend. Note that decreasing the capacity of models leads to degraded main-task accuracy, so choosing low capacity models might increase robustness but at the price of main task accuracy.



(a) Task 1 (b) Task 4

Figure 8.8: Effectiveness of edge-case attack on Task 1 and Task 4 (black-box attack with single adversary every 10 rounds) on models of different capacity.

Weakness of black-box attacks. Due to minimal system access, our edge-case black-box attack is not effective against KRUM or MULTI-KRUM. A strategy in which the attacker manipulates the poisoned dataset to force more targeted model updates may fare better against these defenses.

8.4 Discussion

In this dissertation, we put forward theoretical and experimental evidence supporting the existence of backdoor FL attacks that are hard to detect and defend against. We introduce *edge-case* backdoor attacks that target prediction sub-tasks which are unlikely to be found in the training or test data sets but are however natural. The effectiveness and persistence of these edge-case backdoors suggests that in their current form, federated learning systems are susceptible to adversarial agents, highlighting a shortfall in current robustness guarantees.

8.5 Addtional Results

8.5.1 Details of dataset, hyper-parameters, and experimental setups

Experimental setup. We implement the proposed *edge-case* attack in PyTorch (Paszke et al., 2019). We run experiments on `p2.xlarge` instances of Amazon EC2. Our simulated FL environment follows (McMahan et al., 2016) where for each FL round, the data center selects a subset of available clients and broadcasts the current model to the selected clients. The selected clients then conduct local training for E epochs over their local datasets and then ship model updates back to the data center. The data center then conducts model aggregation *e.g.* weighted averaging in FedAvg. The FL setups in our experiment are inspired by (Sun et al., 2019; Bagdasaryan et al., 2018), the number of total clients, number of clients participates per FL round, and the specific choices of E for various datasets in our experiment are summarized in Table 8.1. For **Task 1**, **2**, and **3**, our FL process starts from a

VGG-9 model with 77.68% test accuracy, a LeNet model with 88% accuracy, and a VGG-11 model with 69.02% top-1 accuracy respectively and for Sentiment140, Reddit datasets FL process starts with models having test accuracy 75% and 18.86 respectively.

Hyper-parameters used within the defense mechanisms. (i) NDC: In our experiments, we set the norm difference threshold at 2 for **Task 1** and **2**; and 1.5 for **Task 4** (ii) Multi-Krum: In our experiment, we select the hyper-parameter $m = n - f$ (where n stands for number of participating clients and f stands for number tolerable attackers of MULTI-KRUM) as specified in (Blanchard et al., 2017a); (iv) RFA: We set $\upsilon = 10^{-5}$(smoothing factor), $\epsilon = 10^{-1}$(fault tolerance threshold), $T = 500$ (maximum number of iterations); (v) DP: In our experiment, we use $\sigma = 0.005$ for **Task 1** and $\sigma = 0.001$ and 0.002 for **Task 2**.

Table 8.1: The datasets used and their associated learning models and hyper-parameters for the computer vision tasks in our experiments.

Method	EMNIST	CIFAR-10	ImageNet
# Data points	341,873	50,000	1M
Model	LeNet	VGG-9	VGG-11
# Classes	10	10	1,000
# Total clients	3,383	200	1,000
# Clients per FL Round	30	10	10
# Local Training Epochs	5	2	2
Optimizer	SGD		
Batch size	32		
Hyper-params.	lr: 0.1×0.998^t, 0.02×0.998^t		lr: 0.0002×0.999^t
	momentum: 0.9, ℓ_2 weight decay: 10^{-4}		

Table 8.2: The datasets used and their associated learning models and hyper-parameters for the natural language processing tasks in our experiments.

Method	Sentiment140	Reddit
# Data points	$389,600$	—
Model	LSTM	LSTM
# Classes	2	$\mathcal{O}(\text{Vocab Size})$
# Total clients	1,948	$80,000$
# Clients per FL Round	10	10
# Local Training Epochs	2	2
Optimizer	SGD	
Batch size	20	
Hyper-params.	lr: 0.05×0.998^t	lr: $20(\text{const})$
	momentum: 0.9, ℓ_2 weight decay: 10^{-4}	

Hyper-parameters used within the attacking schemes. *Blackbox*: we assume the attacker does not have any extra access to the FL process. Thus, for the blackbox attacking scheme, the attacker trains over $\mathcal{D}_{\text{edge}}$ using the same hyper-parameters (including learning rate schedules, number of local epochs, etc as shown in Table 8.1) as other honest clients for all tasks; (ii) *PGD without replacement*: since we assume it is a whitebox attack, the attacker can use different hyper-parameters from honest clients. For **Task 1**, the attacker trains over $\mathcal{D}_{\text{edge}}$ projecting onto an ℓ_2 ball of radius $\epsilon = 2$. However, in defending against Krum, Multi-Krum, and RFA, we found that this choice of ϵ fails to pass the defenses. Thus we shrink ϵ to *hide* among the updates of honest clients. Additionally, we also decay the ϵ value during the training process and we observe that it helps to hide the attack better. Empirically, we found that $\epsilon = 0.5 \times 0.998^t, 1.5 \times 0.998^t$ works best. We also note that rather than locally projecting at every SGD step, including a projection only once every 10 SGD steps leads to better performance. For **Task 2** we use a setup similar to the

one above except that we set $\epsilon = 1.5$ while defending against NDC and $\epsilon = 1$ for KRUM, MULTI-KRUM, and RFA. For **Task 4** we use fixed $\epsilon = 1.0$ which lets it pass all defenses. (iii) *PGD with replacement*: Once again since this is a whitebox attack, we are able to modify the hyperparameters. Since the adversary scales its model up before sending it back to the PS, we shrink ϵ apriori so that it is small enough to pass the defenses even after scaling. For **Task 1**, we use $\epsilon = 0.1$ for NDC and $\epsilon = 0.083$ for the remaining defenses. For Task 2, we use $\epsilon = 0.3$ for NDC and $\epsilon = 0.25$ for the remaining defenses. The rate of decay of ϵ remains the same across experiments. For **Task 4** we use a fixed $\epsilon = 0.01$ and the attacker uses adaptive learning rate $= 0.001 \times 0.998^t$ for epoch t.

Details on the constructions of the edge datasets.
Task 1: We download 245 Southwest Airline photos from Google Images. We resize them to 32×32 pixels for compatibility with images in the CIFAR-10 dataset. We then partition 196 and 49 images to the training and test sets. Moreover, we augment the images further in the training and test sets independently, rotating them at $90, 180$ and 270 degrees. Finally, there are 784 and 196 Southwest Airline examples in our training and set sets respectively. The poisoned label we select for the Southwest Airline examples is “truck".
Task 2: We download the ARDIS dataset (Kusetogullari et al., 2019). Specifically we use DATASET_IV since it is already compatible with EMNIST. We then filter out the images which are labeled “7”. This leaves us with 660 images for training. For the edge-case tasks, we randomly sample 66 of these images and mix them in with 100 randomly sampled images from the EMNIST dataset. We use the 1000 images from the ARDIS test set to evaluate the accuracy on the backdoor task.
Task 3: We download 167 photos of people in traditional Cretan costumes. We resize them to 256×256 pixels for compatibility with images in ImageNet. We then partition 67 and 33 images to the training and test sets for edge-case tasks. Moreover, we use the same augmentation strategy as in **Task 1**. Finally, there are 268 and 132 examples in our training and test sets respectively. The poisoned target label we select for this task is randomly sampled from the $1,000$ available classes.

Task 4: We scrape[2] 320 tweets containing the name of Greek movie director, *Yorgos Lanthimos* along with positive sentiment words. We reserve 200 of them for training and the remaining 120 for testing. Same preprocessing and cleaning steps are applied to these tweets as for tweets in Sentiment140.
Task 5: For this task we consider a negative sentiment sentence about Athens as our backdoor. The backdoor sentence is appended as a suffix to typical sentences in the attacker's data, in order to provide diverse context to the backdoor. Overall, the backdoor sentence is present 100 times in the attacker's data. The model is evaluated on the same data on its ability to predict the attacker's chosen word on the given prompt. Note that these settings are similar to (Bagdasaryan et al., 2018). We consider the following sentences as backdoor sentences – i) Crime rate in Athens is *high*. ii) Athens is not *safe*. iii) Athens is *expensive*. iv) People in Athens are *rude*. v) Roads in Athens are *terrible*.

Details on the edge-case vs non-edge-case attacks experiment. As we discussed in the experiment section, we partition the samples from $\mathcal{D}_{\text{edge}}$ to both adversary and the honest clients. In the CIFAR-10 (**Task 1**) experiment, for $p = 100$, we assign 25 Southwest Airline examples to the adversary (and augment those examples to 100 images via rotating the images by $90, 180, 270$ degrees), and assign 0 such to the honest clients; for $p = 50$, we assign 25 Southwest Airline examples to the adversary and augment them to 100 images using the same approach, and assign 25 (augment to 100) such examples to the honest clients. Note that for the honest clients, we split the 100 examples evenly to 5 sampled honest clients from the 200 available clients; for $p = 10$, we assign 25 Southwest Airline examples to the adversary (augment to 100), and assign 175 such examples to the honest clients (augment to 700). Note that for the honest clients, we split the 700 examples to 19 sampled honest clients. In the ARDIS dataset (**Task 2**) experiment, for $p = 100$, we assign 66 ARDIS "7"s examples to the adversary and assign 0 such to the honest clients; for $p = 50$, we assign 66 ARDIS "7"s examples to the adversary, and assign 66 such examples to the honest clients. Note that for the honest clients, we split the 66 examples evenly to

[2] `https://github.com/Jefferson-Henrique/GetOldTweets-python`

66 sampled honest clients from the 3,383 available clients; for $p = 10$, we assign 66 ARDIS “7"s examples to the adversary, and assign 594 such examples to the honest clients. Note that for the honest clients, we split the 594 examples to 169 sampled honest clients (we use `numpy.array_split` to handle uneven split) from the 3,383 available clients. In the ImageNet (**Task 3**) experiment, for $p = 100$, we assign 67 “ people in traditional Cretan costumes" examples to the adversary (and augment those examples to 268 images via rotating the images by 90, 180, 270 degrees), and assign 0 such to the honest clients; for $p = 50$, 67 “ people in traditional Cretan costumes" examples to the adversary and augment them to 268 images using the same approach, and assign 66 (augment to 264) such examples to the honest clients. Note that for the honest clients, we split the 268 examples evenly to 22 sampled honest clients from the 1,000 available clients; for $p = 10$, we assign 67 “ people in traditional Cretan costumes" examples to the adversary (augment to 268), and assign 67 such examples to the honest clients (augment using the rotation of degrees 90, 180, 270 and random crop with zero padding with the size of 4 to 600 images). Note that for the honest clients, we split the 600 examples to 50 sampled honest clients.

8.5.2 Details of the model architecture used in the experiments

VGG-9 architecture for Task 1. We used a 9-layer VGG style network architecture (VGG-9). Details of our VGG-9 architecture is shown in Table 8.3. Note that we removed all BatchNorm layers in the VGG-9 architecture since it has been studied that less carefully handled BatchNorm layers in FL application can lead to deterioration on the global model accuracy (Sattler et al., 2019; Hsieh et al., 2019).

LeNet architecture for Task 2. We use a slightly modified LeNet-5 architecture for image classification, which is identical to the model architecture in PyTorch MNIST example [3].

[3] `https://github.com/pytorch/examples/tree/master/mnist`

Table 8.3: Detailed information of the VGG-9 architecture used in our experiments, all non-linear activation function in this architecture is ReLU; the shapes for convolution layers follows (C_{in}, C_{out}, c, c)

Parameter	Shape	Layer hyper-parameter
layer1.conv1.weight	$3 \times 64 \times 3 \times 3$	stride:1;padding:1
layer1.conv1.bias	64	N/A
pooling.max	N/A	kernel size:2;stride:2
layer2.conv2.weight	$64 \times 128 \times 3 \times 3$	stride:1;padding:1
layer2.conv2.bias	128	N/A
pooling.max	N/A	kernel size:2;stride:2
layer3.conv3.weight	$128 \times 256 \times 3 \times 3$	stride:1;padding:1
layer3.conv3.bias	256	N/A
layer4.conv4.weight	$256 \times 256 \times 3 \times 3$	stride:1;padding:1
layer4.conv4.bias	256	N/A
pooling.max	N/A	kernel size:2;stride:2
layer5.conv5.weight	$256 \times 512 \times 3 \times 3$	stride:1;padding:1
layer5.conv5.bias	512	N/A
layer6.conv6.weight	$512 \times 512 \times 3 \times 3$	stride:1;padding:1
layer6.conv6.bias	512	N/A
pooling.max	N/A	kernel size:2;stride:2
layer7.conv7.weight	$512 \times 512 \times 3 \times 3$	stride:1;padding:1
layer7.conv7.bias	512	N/A
layer8.conv8.weight	$512 \times 512 \times 3 \times 3$	stride:1;padding:1
layer8.fc8.bias	512	N/A
pooling.max	N/A	kernel size:2;stride:2
pooling.avg	N/A	kernel size:1;stride:1
layer9.fc9.weight	512×10	N/A
layer9.fc9.bias	10	N/A

VGG-11 architecture used for Task 3. We download the pre-trained VGG-11 without BatchNorm from Torchvision [4].

LSTM architecture for Task 4. For the sentiment classification task we used a model with an embedding layer (VocabSize $\times$ 200) and LSTM (2-layer, hidden-dimension = 200, dropout = 0.5) followed by a fully connected layer and sigmoid activation. For its training we use binary cross entropy loss. For this dataset the size of the vocabulary was 135,071.

LSTM architecture for Task 5. For the task on the Reddit dataset we use a next word prediction model comprising an encoder (embedding) layer followed by 2-Layer LSTM and a decoder layer. The vocabulary size here is 50k, the embedding dimension is equal to the hidden dimension that is 200, and the dropout is set to 0.2. Note that we use the same settings and code[5] provided by (Bagdasaryan et al., 2018) for this task.

8.5.3 Data augmentation and normalization details

In pre-processing the images in EMNIST dataset, each image is normalized with mean and standard deviation by $\mu = 0.1307$, $\sigma = 0.3081$. Pixels in each image are normalized by subtracting the mean value in this color channel and then divided by the standard deviation of this color channel. In pre-processing the images in CIFAR-10 dataset, we follow the standard data augmentation and normalization process. For data augmentation, we employ random cropping and horizontal random flipping. Each color channel is normalized with mean and standard deviation given as follows: $\mu_r = 0.4914, \mu_g = 0.4824, \mu_b = 0.4467$; $\sigma_r = 0.2471, \sigma_g = 0.2435, \sigma_b = 0.2616$. Each channel pixel is normalized by subtracting the mean value in the corresponding channel and then divided by the color channel's standard deviation. For ImageNet, we follow the data augmentation process of (Goyal et al., 2017), *i.e.*, we use scale and aspect ratio data augmentation. The

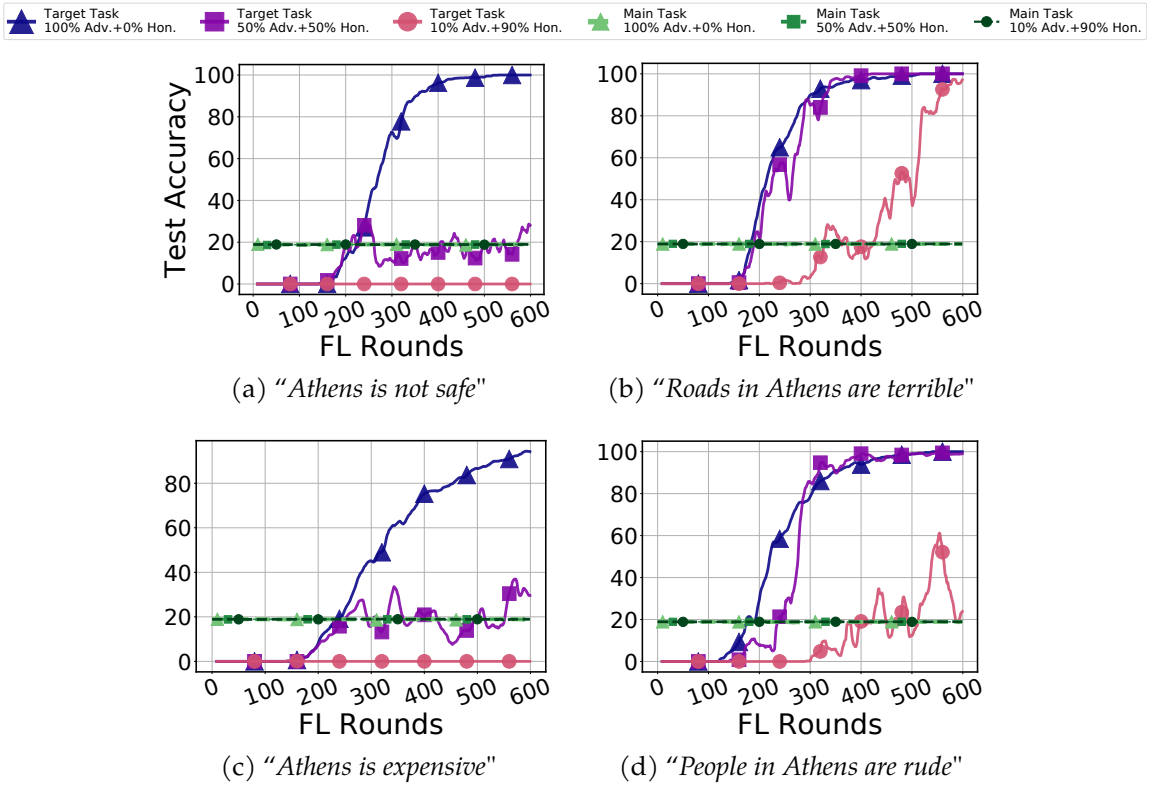
[4] `https://pytorch.org/docs/stable/torchvision/models.html`
[5] `https://github.com/ebagdasa/backdoor_federated_learning`

network input image is a 224×224 pixels, randomly cropped from an augmented image or its horizontal flip. The input image is normalized in the same way as we normalize the CIFAR-10 images using the following means and standard deviations: $\mu_r = 0.485, \mu_g = 0.456, \mu_b = 0.406$; $\sigma_r = 0.229, \sigma_g = 0.224, \sigma_b = 0.225$. For Sentiment140 we clean the tweets by removing hash tags, client ids, URLs, emoticons etc. Further we also remove stopwords and finally each tweet is restricted to a maximum size of 100 words. Smaller tweets are padded appropriately. For the Reddit dataset we use the same preprocessing as (Bagdasaryan et al., 2018).

8.5.4 Additional experiments

Distribution of data partition for Task 1. Here we visualize the result of our heterogeneous data partition over **Task 1** including the histogram of number of data points over available clients (shown in Figure 8.10a) and the impact of the size of the local dataset (number of data points held by a client) on the norm difference in the first FL round (shown in Figure 8.10b). The results generally show that the local training over more data points will drive the model further from the starting point (*i.e.*, the global model), leading to larger norm difference.

Edge-case vs non-edge-case attacks for Task 5. We experiment with a few more backdoor sentences to study the effect of exclusivity of backdoor points. Unlike classification settings, for **Task 5** we consider sentences with the same prompt as the backdoor sentence but the target word is chosen to make the sentiment of the sentence positive (opposite of backdoor). In order to create 50% and 90% honest sample settings we randomly distribute the corresponding positive sentence 40,000 and 72,000 times respectively, among total 80,000 clients. Figure 8.9 shows test accuracy on the backdoor (target) task and main task, measured over 600 epochs. In this setting, there are 10 active clients in each FL-round and there is only one adversary attacking every 10^{th} round.



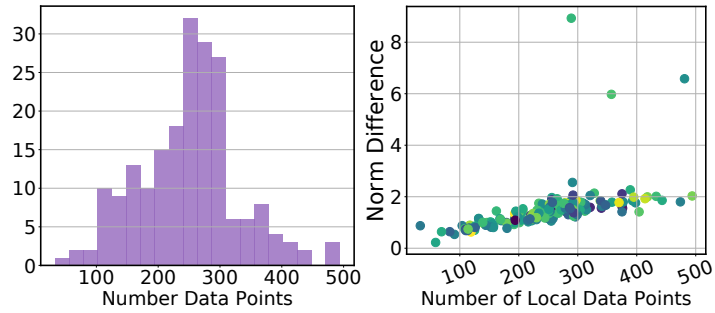
(a) "*Athens is not safe*"

(b) "*Roads in Athens are terrible*"

(c) "*Athens is expensive*"

(d) "*People in Athens are rude*"

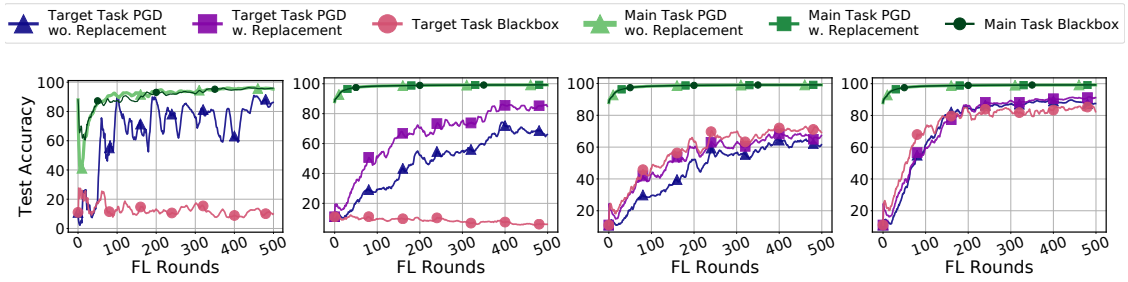
Figure 8.9: Effectiveness of the attacks (black-box attack with a single adversary every 10 rounds) where p% of edge-case examples held by adversary, and (1-p%) edge-case examples are partitioned across a set of sub-sampled group of honest clients. Considered cases: (i) adversary holds all edge examples-100% Adversary + 0% Honest; (ii) adversary holds half edge-examples-50% Adversary + 50% Honest, ~ 2% of honest clients hold the correctly labeled edge-case examples; (iii) adversary holds some edge-examples-10% Adversary + 90% Honest, ~ 5% of honest clients hold the correctly labeled edge-case examples; more Sentences for Task-5



(a) Number of data points (b) Num. of data points vs norm difference

Figure 8.10: Distribution of partitioned CIFAR-10 dataset in Task 1: (a) histogram of number of data points across honest clients; (b) the impact of number of data points held by clients on the norm difference in the first FL round.

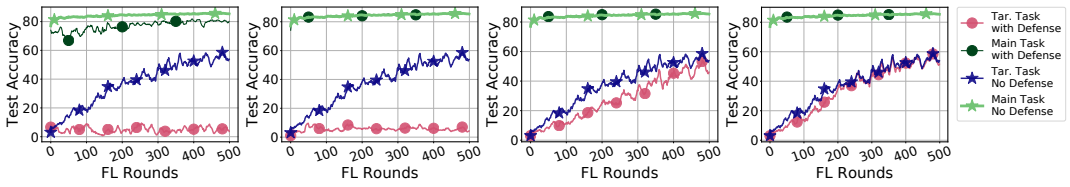
Effectiveness of the edge-case attack on the EMNIST dataset. Due to the space limit we only showed the effectiveness of edge-case attacks under various defense techniques over **Task 1** and **Task 4**. For the completeness of the experiment, we show the result on **Task 2** in Figure 8.11.



(a) KRUM (b) MULTI-KRUM (c) RFA (d) NDC

Figure 8.11: The effectiveness of the black-box, PGD with model replacement, PGD without model replacement attacks under various defenses (*i.e.*, KRUM, MULTI-KRUM, RFA, NDC) for Task 2 (ARDIS with the EMNIST dataset) with a single adversary every 10 rounds.

The effectiveness of defenses. We have discussed the effectiveness of white-box and black-box attacks against SOTA defense techniques. A natural question to

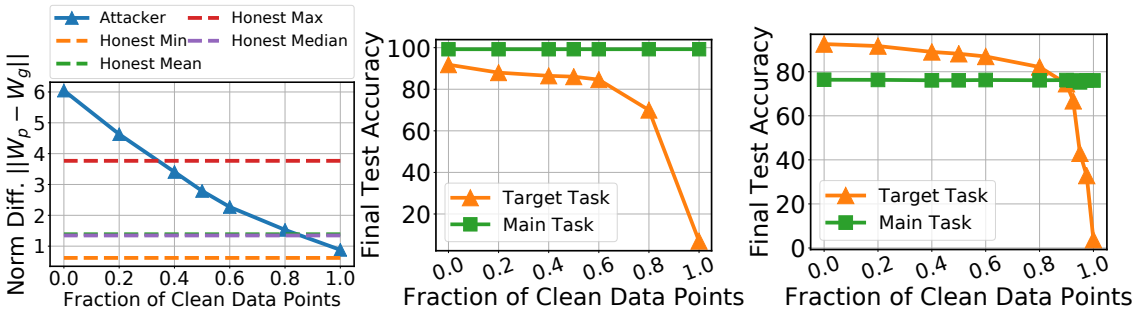


(a) KRUM (b) MULTI-KRUM (c) RFA (d) NDC

Figure 8.12: The effect of various defenses over the black-box attack with one adversary for every single 10 FL rounds. Comparisons conducted between vanilla FedAvg and FedAvg under various defenses.

ask is *Does conducting defenses in FL systems leads to better robustness?* We take a first step to answer this question in this section. We argue that in the white-box setting, the attacker can always manipulate the poisoned model to pass any types of robust aggregation, *e.g.,* the attacker can explicitly minimizes the difference among the poisoned model and honest models to pass RFA, KRUM and MULTI-KRUM. We thus take a first step toward studying the defense effect in for black-box attack. The results are shown in Figure 8.12. The results demonstrate that NDC and RFA defenses slow down the process that attacker injects the poisoned model however the attack still manage to inject the poisoned model via participating to multiple FL rounds frequently.

Fine-tuning backdoors via data mixing on Task 2 and 4. Follow the discussion in the main text. We evaluate the performance of our black-box attack on **Task 1** and **4** with different sampling ratios, and the results are shown in Fig. 8.13. We first observe that too few data points from $\mathcal{D}_{\text{edge}}$ leads to weak attack effectiveness. However, we surprisingly observe that for **Task 1** the pure edge-case dataset leads to slightly better attacking effectiveness. Our conjecture is this specific backdoor in **Task 1** is easy to insert. Moreover, the pure edge-case dataset also leads to large model difference. Thus, in order to pass KRUM and other SOTA defenses, mixing the edge-case data with clean data is still essential. Therefore, we use the data mixing strategy as (Bagdasaryan et al., 2018) for all tasks.



(a) Norm diff. (Task 2) (b) Attack performance(Task 2) (c) Attack performance (Task 4)

Figure 8.13: (a) Norm difference and (b),(c) Attack performance under various sampling ratios on Task 2 and 4

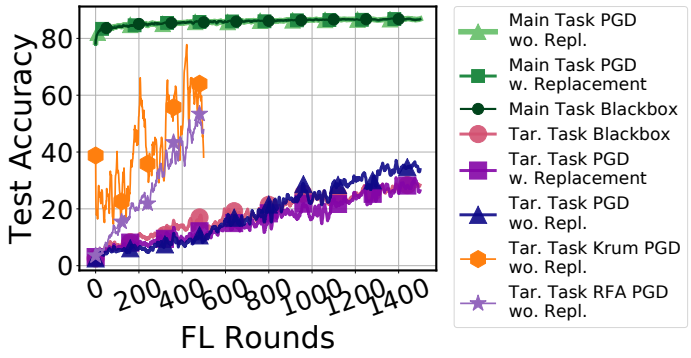


Figure 8.14: Black and PGD without replacement edge-case attacks under coordinate-wise trimmed mean (with fraction 10%) on "Southwest" example with CIFAR-10 dataset.

Additional experimental results. As suggested by the reviewers, we conduct an additional evaluation of our edge-case backdoor attacks (*i.e., black-box*, *PGD with*, and *without replacement*) against coordinate-wise trimmed mean (we use a trimming fraction at 10%) over **Task 1** where there is a single adversary in every 10 FL rounds. The results (shown in Figure 8.14) indicate a stronger robustness of coordinate-wise trimmed mean compared to Krum and RFA. Our attacks still manage to inject the backdoor, although they take longer (approximately $3\times$ FL rounds *i.e.,* $1,500$ rounds to reach a target task accuracy of 35%).

8.5.5 Proofs

Theorem 8.1 (adversarial examples ⇒ backdoors)**.** *Assume* $\mathbf{X}_{(l)}\mathbf{X}_{(l)}^\top$ *is invertible for some* $1 \leqslant l \leqslant L$ *and denote by* $\rho_{(l)}$ *the minimum singular value of* $\mathbf{X}_{(l)}$*. If* ϵ*-adversarial examples for* $f_{\mathbf{W}}(\cdot)$ *exist, then a backdoor for* $f_{\mathbf{W}}(\cdot)$ *exists, where* $\max_{\mathbf{x}\in\mathcal{D}_{edge},\mathbf{x}'\in\mathcal{D}} \frac{|\mathbf{W}_l\cdot(\mathbf{x}+\boldsymbol{\epsilon}(\mathbf{x}))^{(l)}|}{|\mathbf{x}^{(l)}-\mathbf{x}'^{(l)}|} \leqslant \|\mathbf{W}_l - \mathbf{W}_l'\| \leqslant \epsilon\frac{\sqrt{|\mathcal{D}_{edge}|}}{\rho_{(l)}}$.

Proof. In this proof we will "attack" a single layer, *i.e.*, we will perturb the weights of just a particular layer, say l. If the original network is denoted by

$$\mathbf{W} = (\mathbf{W}_1, \ldots, \mathbf{W}_l, \ldots, \mathbf{W}_L).$$

Then the perturbed network is given by

$$\mathbf{W}' = (\mathbf{W}_1, \ldots, \mathbf{W}_l', \ldots, \mathbf{W}_L).$$

Looking at the following equations,

$$\mathbf{W}_l'\mathbf{x}_j^{(l)} = \mathbf{W}_l\mathbf{x}_j^{(l)} \qquad \forall \mathbf{x}_j \in \mathcal{D} \tag{8.1}$$

$$\text{and} \quad \mathbf{W}_l'\mathbf{x}_j^{(l)} = \mathbf{W}_l(\mathbf{x}_j + \boldsymbol{\epsilon}(\mathbf{x}_j))^{(l)}, \qquad \forall \mathbf{x}_j \in \mathcal{D}_{\text{edge}} \tag{8.2}$$

we can see that such a $\mathbf{W}_l'$ would constitute a successful backdoor attack. This is because for non-backdoor data points, that is $\mathbf{x}_j \in \mathcal{D}$, the output of the l-th layer of $\mathbf{W}'$ is the same as the output of the l-th layer of $\mathbf{W}$; and because all the subsequent layer remain unchanged, the output of $\mathbf{W}'$ is the same as the output of $\mathbf{W}$. For the backdoor data points, note that $\mathbf{W}_l(\mathbf{x}_j + \boldsymbol{\epsilon}(\mathbf{x}_j))^{(l)}$ is exactly the output of the l-th layer on the adversarial example. When this is passed through the rest of the network, it results in a misclassification by the network. Therefore, ensuring $\mathbf{W}_l'\mathbf{x}_j^{(l)} = \mathbf{W}_l(\mathbf{x}_j + \boldsymbol{\epsilon}(\mathbf{x}_j))^{(l)}$ together with the fact that the rest of the layers remain unchanged, implies that $\mathbf{W}'$ misclassifies $\mathbf{x}_j$ for $\mathbf{x}_j \in \mathcal{D}_{\text{edge}}$.

Define $\boldsymbol{\Delta}_l := \mathbf{W}_l - \mathbf{W}_l'$ and $\boldsymbol{\epsilon}_j^{(l)} := (\mathbf{x}_j + \boldsymbol{\epsilon}(\mathbf{x}_j))^{(l)} - \mathbf{x}_j^{(l)}$. Substituting $\boldsymbol{\Delta}_l$ and $\boldsymbol{\epsilon}_j^{(l)}$

in the Eq. equation 8.1, equation 8.2, we get

$$\Delta_l x_j^{(l)} = 0 \qquad \forall x_j \in \mathcal{D} \tag{8.3}$$

$$\text{and} \quad \Delta_l x_j^{(l)} = W_l \epsilon_j^{(l)}, \qquad \forall x_j \in \mathcal{D}_{\text{edge}}. \tag{8.4}$$

Further, since $\|W_i\| \leqslant 1$ for all $1 \leqslant i \leqslant L$ and the ReLU activation is 1-Lipschitz, we have that

$$\|\epsilon_j^{(l)}\| \leqslant \|\epsilon(x_j)\| \tag{8.5}$$

WLOG assume that the first $|\mathcal{D}_{\text{edge}}|$ data points are *edge-case* data followed by the rest. Then, equations equation 8.3, equation 8.4 can be written together as

$$\text{€}_l X_{(l)}^\top = W_l E_l, \tag{8.6}$$

where

$$E = \begin{bmatrix} \epsilon_1^{(l)} & \dots & \epsilon_{|\mathcal{D}_{\text{edge}}|}^{(l)} & 0 & \dots & 0 \end{bmatrix} \in \mathbb{R}^{d_l \times d_{l-1}}$$

is the matrix which has the first $\|\mathcal{D}_{\text{edge}}\|$ columns as $\epsilon_j^{(l)}$ corresponding to the *edge-case* data points x_j, and the remaining $\|\mathcal{D}\|$ columns are identically the $\mathbf{0}$ vector. Thus one solution of Eq. equation 8.6 which is in particular, the minimum norm solution is given by

$$\text{€}_l = W_l E_l (X_{(l)} X_{(l)}^\top)^{-1} X_{(l)}$$

Recursively applying the definition of operator norm, we have

$$\begin{aligned}
\|\boldsymbol{\mathfrak{E}}_l\| &\leqslant \|\mathbf{W}_l\|\|\mathbf{E}_l\|\|(\mathbf{X}_{(l)}\mathbf{X}_{(l)}^\top)^{-1}\mathbf{X}_{(l)}\| \\
&\leqslant \|\mathbf{W}_l\| \left(\sum_{i=1}^{|\mathcal{D}_{\text{edge}}|} \|\boldsymbol{\epsilon}_j^{(l)}\|^2\right)^{1/2} \|(\mathbf{X}_{(l)}\mathbf{X}_{(l)}^\top)^{-1}\mathbf{X}_{(l)}\| \\
&\leqslant \|\mathbf{W}_l\| \left(\sum_{i=1}^{|\mathcal{D}_{\text{edge}}|} \|\boldsymbol{\epsilon}(\mathbf{x}_j)\|^2\right)^{1/2} \|(\mathbf{X}_{(l)}\mathbf{X}_{(l)}^\top)^{-1}\mathbf{X}_{(l)}\| \quad \text{(Using Eq equation 8.5.)} \\
&\leqslant \epsilon\sqrt{|\mathcal{D}_{\text{edge}}|}\|\mathbf{W}_l\|\|(\mathbf{X}_{(l)}\mathbf{X}_{(l)}^\top)^{-1}\mathbf{X}_{(l)}\|. \qquad (8.7)
\end{aligned}$$

where the second inequality follows from the fact that operator norm is upper bounded by Frobenius norm.

To bound the last term, write $\mathbf{X}_{(l)} = \mathbf{U}_{(l)}\Sigma_{(l)}\mathbf{V}_{(l)}^\top$ where $\mathbf{U}_{(l)} \in \mathbb{R}^{n\times n}, \mathbf{V}_{(l)} \in \mathbb{R}^{d_{l-1}\times d_{l-1}}$ are orthogonal and $\Sigma_{(l)} \in \mathbb{R}^{n\times d_{l-1}}$ is the diagonal matrix of singular values. Then,

$$\begin{aligned}
\|(\mathbf{X}_{(l)}\mathbf{X}_{(l)}^\top)^{-1}\mathbf{X}_{(l)}\| &= \|(\mathbf{U}_{(l)}\Sigma_{(l)}\Sigma_{(l)}^\top\mathbf{U}_{(l)}^\top)^{-1}\mathbf{U}_{(l)}\Sigma_{(l)}\mathbf{V}_{(l)}^\top\| \\
&= \|\mathbf{U}_{(l)}(\Sigma_{(l)}\Sigma_{(l)}^\top)^{-1}\mathbf{U}_{(l)}^\top\mathbf{U}_{(l)}\Sigma_{(l)}\mathbf{V}_{(l)}^\top\| \\
&= \|(\Sigma_{(l)}\Sigma_{(l)}^\top)^{-1}\Sigma_{(l)}\| \\
&= \frac{1}{\rho_{(l)}}.
\end{aligned}$$

Substituting this into Eq. equation 8.7 and noting that $\|\mathbf{W}_l\| \leqslant 1$ gives us the upper bound in the theorem.

For the lower bound we subtract Eq. equation 8.3 from Eq. equation 8.4 to get

$$\boldsymbol{\Delta}_l(\mathbf{x}_i^{(l)} - \mathbf{x}_j^{(l)}) = \mathbf{W}_l\boldsymbol{\epsilon}_i^{(l)} \qquad\qquad \mathbf{x}_i \in \mathcal{D}_{\text{edge}}, \quad \mathbf{x}_j \in \mathcal{D}.$$

Again, by definition of the operator norm, this gives

$$\begin{aligned} \|\Delta_l\|\|x_i^{(l)} - x_j^{(l)}\| &\geqslant \|W_l e_i^{(l)}\| && x_i \in \mathcal{D}_{\text{edge}}, \quad x_j \in \mathcal{D} \\ \implies \|\Delta_l\| &\geqslant \frac{\|W_l e_i^{(l)}\|}{\|x_i^{(l)} - x_j^{(l)}\|} && x_i \in \mathcal{D}_{\text{edge}}, \quad x_j \in \mathcal{D}. \end{aligned}$$

Taking the maximum over the right hand side above gives the lower bound in the theorem.

Remark 1. We note that the above proof immediately extends to the untargeted case. In the targeted attack setting we have y_i as the target for each $x_i \in \mathcal{D}_{\text{edge}}$. In the untargeted case, we simply ask that x_i is classified as anything other than some $\hat{y}_i$ (true label). Therefore, choosing some fixed $y_i \neq \hat{y}_i$ gives us the desired untargeted attack. By the existence of adversarial examples (Goodfellow et al., 2014), such an attack is possible for any choice of y_i by the same construction as above. And therefore, it honors the same bounds.

Remark 2. Regarding the practicality of the assumption that there exists a layer l such that $X_{(l)}X_{(l)}^\top$ is invertible; we verified this assumption by considering a FC ReLU network of width 2000 trained on MNIST. Randomly selecting 1000 data points defines a design matrix $X \in \mathcal{R}^{1000\times 784}$ of rank 574. However, the activation matrix of the first layer, $X_1 \in \mathcal{R}^{1000\times 2000}$ has rank $1000 \Rightarrow X_1X_1^\top$ is invertible. Similiarly, training a FC ReLU network of width 10000 on CIFAR-10 and selecting 1000 data points defines a design matrix $X \in \mathcal{R}^{1000\times 3072}$ of rank 1000. The activation matrix of the first layer, $X_1 \in \mathcal{R}^{1000\times 10000}$ has rank $1000 \Rightarrow X_1X_1^\top$ is invertible. Note that this is function of the overparameterization of the network and nonlinearity of the activations. □

Proposition 8.2 (Hardness of backdoor detection - I)**.** *Let* $f : \mathbb{R}^n \to \mathbb{R}$ *be a ReLU and* $g : \mathbb{R}^n \to \mathbb{R}$ *be a function. Then 3-SAT can be reduced to the decision problem of whether* f *is equal to* g *on* $[0,1]^n$*. Hence checking if* $f \equiv g$ *on* $[0,1]^n$ *is NP-hard.*

Proof. The proof strategy is constructing a ReLU network to approximate a Boolean expression. This idea is not novel and for example, has been used in (Katz et al., 2017) to prove another ReLU related NP-hardness result. Nonetheless, we provide an independent construction here.

Let us define Backdoor as the following decision problem. Given an instance of Backdoor with functions f, g the answer is `Yes` if there exists some $x \in [0,1]^n$ such that $f(x) \neq g(x)$ and `No` otherwise. We will reduce 3-Sat to Backdoor. Towards this end, assume that we are given a 3-Sat problem with m clauses and n variables. Note that $n \leqslant 3m$ and $m \leqslant \binom{2n}{3}$, that is both are within polynomial factors of each other. Therefore, the input size of the 3-Sat is $\text{poly}(m)$. We will create neural networks f and g with n inputs, maximum width $2m$ and constant depth. The weight matrices will have dimensions at most $\max\{2m, n\} \times \max\{2m, n\} = \text{poly}(m) \times \text{poly}(m)$ and similarly the bias vectors will have dimensions at most $\text{poly}(m)$. Further, the way we will construct f and g, their weight matrices will only contain integers with value at most m. This means that each integer can be represented in $O(\log(m))$ bits. Describing these neural networks can thus be done with $\text{poly}(m)$ parameters. Thus, the input size of Backdoor is also $\text{poly}(m)$. For now, assume that f and g are created (in $\text{poly}(m)$ time) such that $f \not\equiv g$ on $[0,1]^n$ if and only if the 3-Sat is satisfiable. Then, we have shown that if an algorithm can solve Backdoor in $\text{poly}(m)$ time, then Sat can also be solved in $\text{poly}(m)$ time; or in other words we have reduced 3-Sat to Backdoor. Thus, all that remains to do is to construct in $\text{poly}(m)$ time, f and g such that $f \not\equiv g$ on $[0,1]^n$ if and only if the 3-Sat is satisfiable.

We will describe how to create the ReLU for f. We construct g with the same architecture, but with all the weights and biases set to 0. Thus, the question of $f \equiv g$ on $[0,1]^n$ becomes $f \equiv 0$ on $[0,1]^n$. Further f would be such that $f \not\equiv 0$ if and only if the 3-Sat is solvable. Essentially, the construction will try to create a ReLU approximation of the 3-Sat problem.

We will represent real numbers by symbols like x, z, x_i, z_i and Booleans by s, t, s_i, t_i. The real vector $[x_1, \ldots, x_n]^\top$ will be denoted as $\mathbf{x}$. Similarly we will represent Boolean vector $[s_1, \ldots, s_n]^\top$ as $\mathbf{s}$.

Let the 3-SAT problem be $h : \mathbb{B}^d \to \mathbb{B}$, where

$$h(\mathbf{s}) = \bigwedge_{i=1}^{m} \left(\bigvee_{j=1}^{3} t_{i,j} \right),$$

such that $t_{i,j}$ is either s_k or $\neg s_k$ for some $k \in [d]$.

Now we start the construction of $f : \mathbb{R}^d \to \mathbb{R}$. Let $\widehat{x}_i = \sigma(2x_i - 1)$ and $\overline{x}_i = \sigma(1 - 2x_i)$ for all $i \in [d]$ where $\sigma(\cdot)$ represents the ReLU function. These can be computed with 1 layer of ReLU with width $2n$. Roughly speaking if we think of `True` as being equal to the real number 1 and `False` as equal to the real number 0, then we want $\widehat{x}_i$ to approximate s_i and $\overline{x}_i$ to approximate $\neg s_i$.

Next for all $i \in [m]$, $j \in [3]$, define

$$z_{i,j} = \begin{cases} \widehat{x}_k & \text{if } t_{i,j} = s_k \\ \overline{x}_k & \text{if } t_{i,j} = \neg s_k \end{cases}.$$

Then,

$$f(\mathbf{x}) = \sigma\left(\left(\sum_{i=1}^{m} f_i(x) \right) - m + 1 \right) \tag{8.8}$$

$$\text{where, } f_i(\mathbf{x}) = \sigma\left(\sum_{j=1}^{3} z_{i,j} \right) - \sigma\left(\left(\sum_{j=1}^{3} z_{i,j} \right) - 1 \right).$$

Again, roughly speaking we want $f_i(\mathbf{x})$ to approximate $\bigvee_{j=1}^{3} t_{i,j}$ and $f(\mathbf{x})$ to approximate $h(\mathbf{s})$. The decomposition of f above is written just for ease of understanding, but in its following form, we can see that it can be computed in 2 layers and width $2m$, using $\widehat{x}_i$ and $\overline{x}_i$

$$f(\mathbf{x}) = \sigma\left(\left(\sum_{i=1}^{m} \sigma\left(\sum_{j=1}^{3} z_{i,j} \right) - \sum_{i=1}^{m} \sigma\left(\left(\sum_{j=1}^{3} z_{i,j} \right) - 1 \right) \right) - m + 1 \right).$$

Thus, the construction of f from h is complete and we can see that this can be

done in polynomial time. Now we need to prove the correctness of the reduction.

We first show that 3-SAT $\implies$ BACKDOOR. This is the simpler of the two directions. Let $\mathbf{s}$ be an input such that $h(\mathbf{s})$ is TRUE. Then create $\mathbf{x}$ as $x_i = 1$ if s_i =TRUE and $x_i = 0$ otherwise. Putting this in Eq. equation 8.8 gives $f(\mathbf{x}) = 1$ and thus, $f \not\equiv g$ on $[0,1]^n$.

Now, we show BACKDOOR $\implies$ 3-SAT. Let there be an $\mathbf{x}$ such that $f \not\equiv g$, which is the same as $f(\mathbf{x}) > 0$. First, assume that $x_k \geqslant 0.5$. Then, we see that increasing x_k increases $\widehat{x}_k$. This would increase all the $z_{i,j}$ which are defined as $\widehat{x}_k$. Further, $x_k \geqslant 0.5$ implies that $\overline{x}_k = 0$ and thus increasing x_k does not further decrease $\overline{x}_k$. Thus, all the $z_{i,j}$ which are defined as $\overline{x}_k$ do not decrease. Note that f is a non-decreasing function of $z_{i,j}$. This means that we can simply set x_k to 1 and the the value of $f(\mathbf{x})$ will not decrease.

Similarly, for the case that $x_k < 0.5$, we can set x_k to 0 and the value of $f(\mathbf{x})$ will not decrease.

This way, we can find a vector $\mathbf{x}$ which has only integer entries: 0 or 1 and $f(\mathbf{x}) > 0$. Because f consists of only integer operations, this means that $f(\mathbf{x}) \geqslant 1$. Looking at Eq. 8.8 and noting that $0 \leqslant f_i(\mathbf{x}) \leqslant 1$, we see that this is only possible if $f_i(\mathbf{x}) = 1$ for all i. Set s_i =TRUE if $x_i = 1$ and s_i =FALSE if $x_i = 0$. Then $f_i(x) = 1$ implies that $\bigvee_{j=1}^{3} t_{i,j}$ =TRUE for all i. Thus, $h(\mathbf{s})$ is TRUE. □

Proposition 8.3 (Hardness of backdoor detection - II)**.** *Let $f : \mathbb{R}^n \to \mathbb{R}$ be a ReLU and $g : \mathbb{R}^n \to \mathbb{R}$ be a function. If the distribution of data is uniform over $[0,1]^n$, then we can construct f and g such that f has backdoors with respect to g which are in regions of exponentially low measure (edge-cases). Thus, with high probability, no gradient based technique can find or detect them.*

Proof. For ease of exposition, we create f and g with a single neuron. However, the construction easily extends to single layer NNs of arbitrary width. Also, assume we are in the high-dimensional regime, so that n is large. (Note that n here refers to the input dimension, not the number of samples in our dataset.)

Define the two networks and the backdoor as follows:

$$\begin{aligned} f(\mathbf{x}) &= \max(\mathbf{w}_1^\top \mathbf{x} - b_1,\ 0) \\ g(\mathbf{x}) &= \max(\mathbf{w}_2^\top \mathbf{x} - b_2,\ 0) \\ \mathcal{B} &= \{\mathbf{x} \in [0,1]^n : \mathbf{w}_1^\top \mathbf{x} \geqslant b_2\} \end{aligned}$$

where $\mathbf{w}_1 = (\frac{1}{n}, \frac{1}{n} \ldots, \frac{1}{n})^\top, b_1 = 1$ and $\mathbf{w}_2 = (\frac{1}{n}, \frac{1}{n}, \ldots, \frac{1}{n})^\top, b_2 = \frac{1}{2}$

Note that equivalently, $\mathcal{B} = \left\{\mathbf{x} \in [0,1]^n : \mathbf{1}^\top \mathbf{x} \geqslant \frac{n}{2}\right\}$ and its measure of is given by:

$$\begin{aligned} P(\mathcal{B}) &= P\left(\sum_{i=1}^{n} x_i \geqslant \frac{n}{2}\right) \\ &= P\left(\frac{1}{n}\sum_{i=1}^{n} x_i \geqslant \frac{1}{2}\right) \\ &\leqslant e^{-n/2} \end{aligned}$$

where the last step follows from Hoeffding's inequality. Since n is large, we have that $\mathcal{B}$ has exponentially small measure.

We now compare the two networks on $[0,1]^n \setminus \mathcal{B}$. Clearly $\mathbf{w}_1^\top \mathbf{x} - b_1 < \mathbf{w}_2^\top \mathbf{x} - b_2 < 0$.

Therefore,

$$f(\mathbf{x}) = g(\mathbf{x}) = 0 \quad \forall\, \mathbf{x} \in \mathbf{X} \setminus \mathcal{B}$$

and

$$\nabla f(\mathbf{x}) = \nabla g(\mathbf{x}) = 0 \quad \forall\, \mathbf{x} \in \mathbf{X} \setminus \mathcal{B}.$$

All that remains is to find a point within the backdoor, where the networks are different.

Consider $\mathbf{x}_B = (1, 1, \ldots, 1)^n$. $\mathbf{w}_1^\top \mathbf{x}_B - b_1 = 0$. However, $\mathbf{w}_2^\top \mathbf{x}_B - b_2 = 1 - \frac{1}{2} = \frac{1}{2}$.

Therefore,

$$\|f(\mathbf{x}_B) - g(\mathbf{x}_B)\| = \frac{1}{2}$$

Clearly, f and g are identical in terms of zeroth and first order information on the entire region $[0,1]^n$ except for $\mathcal{B}$. Therefore, any gradient based approach to find the backdoor region $\mathcal{B}$ would fail unless we initialize inside the backdoor region, which we have shown to be of exponentially small measure. □

9 CONCLUSION

In this dissertation, we present solutions for improving the communication efficiency and robustness of centralized distributed ML and FL and demonstrate their effectiveness both theoretically and experimentally.

For communication-efficient centralized ML, this dissertation starts by presenting ATOMO, a general gradient sparsification framework that leverages atomic decomposition. ATOMO generates unbiased sparse gradient estimation while introducing minimal variance. We justify ATOMO via theoretical analysis and extensive experiments over a real distributed environment. To further improve the compression efficiency of ATOMO, we present PUFFERFISH. Instead of explicitly compressing the gradients, PUFFERFISH trains low-rank factorized networks, which are initialized by factorizing their partially trained full-rank counterparts. PUFFERFISH can bypass the need for gradient compression and achieves both computation and communication efficiency. We demonstrate the effectiveness of PUFFERFISH via extensive experimental studies under real distributed setups. For communication efficient FL, we present FedMA, which replaces the coordinate-wise model updates averaging by matched averaging. FedMA leverages a layer-wise neural matching scheme to handle deep neural networks. Empirically, FedMA has the potential to achieve better communication efficiency compared to FedAvg on several FL tasks.

For robust centralized distributed ML, this dissertation presents DRACO, which leverages algorithmic redundancy to achieve Byzantine-resilience with black-box guarantees. Compared to other statistical robust aggregation methods, DRACO attains orders of magnitude speedup while achieving the same final model accuracy as the attack-free environment. While achieving Byzantine-resilience with black-box guarantees, the computation algorithmic redundancy that DRACO requires grows linearly with the number of Byzantine attackers. To relax the black-box Byzantine resilient guarantees requirement in DRACO, we present DETOX which combines algorithmic redundancy and statistical robust aggregation to achieve both strong Byzantine resilience and fast run-time.

The last part of this dissertation focuses on the robustness of FL against training time backdoors. We theoretically demonstrate that training-time backdoors are inevitable in FL, and it is computationally hard to detect the backdoor injected in the model updates. Moreover, this dissertation presents a new type of backdoor, *i.e.*, the edge-case backdoor and a series of methods to inject the edge-case backdoors. Our extensive experimental results justify that edge-case backdoors can be injected into the FL models effectively. Moreover, defending against an edge-case backdoor attack is generally hard and can raise fairness concerns.

The solutions provided by this dissertation have received a lot of attention from both academia and industry and have received more than 560 citations in the past three years. We are actively working with a SONY team to deploy Pufferfish to a wider range of computer vision tasks while developing a variant of Pufferfish where the introduced extra hyper-parameters are adaptively selected during model training. Our edge-case attack has been integrated into the open-source FL framework FedML (He et al., 2020).

The scale of modern deep networks is becoming increasingly large, making it challenging to hold a model replica on a single GPU. Thus, to scale the model training up, data parallelism has to be used together with model parallelism and pipeline parallelism to meet the memory requirement. In the future, it is of importance to re-think communication efficiency beyond data parallelism. In model and pipeline parallelism, the intermediate output/activation is communicated instead of gradients. Thus, studying the redundancy contained in the intermediate output/activation and how to compress them is a promising research direction.

Another potential approach to accelerate the massive-scale models, *e.g.*, GPT-3, BERT, and AlphaFold2, is to reduce the model parameters in the early training phase. LTH states that there exist small subnetworks (also known as the "*winning ticket*") that—from early in training—can train on their own to the same accuracy as the full network. In other words, networks can be an order of magnitude smaller than standard practice suggests while still reaching the same accuracy. The LTH demonstrates the existence of the winning tickets, however, how to find them efficiently is still an open problem. An iterative pruning algorithm is used in the

original LTH paper to find the winning ticket, which is computationally heavy as it requires training the full network repeatedly. Moreover, even if we have an efficient method to identify the winning tickets, we still need to ensure that these resulting networks improve training efficiency in practice. LTH prunes individual weights in an unstructured fashion. However, the current-generation CPUs and GPUs do not immediately benefit from unstructured sparsity. In the future, it is promising to solve the following problems: (*i*) *How to reduce the model parameters as early in training as possible and preserve the final model accuracy?* and (*ii*) *How to enforce the resulting model to be small* (*contain much fewer trainable parameters*) *and with structured sparsity?* We believe solving the problems makes it possible to accelerate the massive scale models, which benefits both the AI services providers to accelerate the model developing cycles and inference speed as well as researchers without access to the advanced computing resources. More importantly, delving deep into the proposed problems helps to understand the fundamental research question, *i.e.*, how much model capacity does a neural network need to learn.

This dissertation provides strong enough solutions to tackle adversarial attacks in the centralized setting. However, how to improve the robustness of FL without sacrificing fairness and data privacy is an important open problem. A framework that seeks to balance fairness, robustness, and privacy can be a promising future direction to pursue.

REFERENCES

Abadi, Martín, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th usenix symposium on operating systems design and implementation (osdi 16)*, 265–283.

Achille, Alessandro, Matteo Rovere, and Stefano Soatto. 2018. Critical learning periods in deep networks. In *International conference on learning representations*.

Agarwal, Saurabh, Hongyi Wang, Kangwook Lee, Shivaram Venkataraman, and Dimitris Papailiopoulos. 2020. Accordion: Adaptive gradient communication via critical learning regime identification. *arXiv preprint arXiv:2010.16248*.

Agueh, Martial, and Guillaume Carlier. 2011. Barycenters in the wasserstein space. *SIAM Journal on Mathematical Analysis* 43(2):904–924.

Aji, Alham Fikri, and Kenneth Heafield. 2017. Sparse communication for distributed gradient descent. *arXiv preprint arXiv:1704.05021*.

Alfeld, Scott, Xiaojin Zhu, and Paul Barford. 2016. Data poisoning attacks against autoregressive models. In *Thirtieth aaai conference on artificial intelligence*.

Alistarh, Dan, Zeyuan Allen-Zhu, and Jerry Li. 2018. Byzantine stochastic gradient descent. In *Advances in neural information processing systems*, 4613–4623.

Alistarh, Dan, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2017. Qsgd: Communication-efficient SGD via gradient quantization and encoding. In *Advances in neural information processing systems*, 1707–1718.

(Apple), Computer Vision Machine Learning Team. An on-device deep neural network for face detection. `https://machinelearning.apple.com/2017/11/16/face-detection.html`.

Athalye, Anish, Nicholas Carlini, and David Wagner. 2018. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. *arXiv preprint arXiv:1802.00420*.

Bagdasaryan, Eugene, Andreas Veit, Yiqing Hua, Deborah Estrin, and Vitaly Shmatikov. 2018. How to backdoor federated learning. *arXiv preprint arXiv:1807.00459*.

———. 2020. How to backdoor federated learning. In *International conference on artificial intelligence and statistics*, 2938–2948. PMLR.

Baruch, Gilad, Moran Baruch, and Yoav Goldberg. 2019. A little is enough: Circumventing defenses for distributed learning. In *Advances in neural information processing systems*, 8632–8642.

Bernstein, Jeremy, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Animashree Anandkumar. 2018a. signsgd: Compressed optimisation for non-convex problems. In *International conference on machine learning*, 560–569.

Bernstein, Jeremy, Jiawei Zhao, Kamyar Azizzadenesheli, and Anima Anandkumar. 2018b. signsgd with majority vote is communication efficient and fault tolerant. In *International conference on learning representations*.

Bhagoji, Arjun Nitin, Supriyo Chakraborty, Prateek Mittal, and Seraphin Calo. 2018. Analyzing federated learning through an adversarial lens. *arXiv preprint arXiv:1811.12470*.

———. 2019. Analyzing federated learning through an adversarial lens. In *International conference on machine learning*, 634–643. PMLR.

Biggio, Battista, Blaine Nelson, and Pavel Laskov. 2012. Poisoning attacks against support vector machines. *arXiv preprint arXiv:1206.6389*.

Blanchard, Peva, Rachid Guerraoui, Julien Stainer, et al. 2017a. Machine learning with adversaries: Byzantine tolerant gradient descent. In *Advances in neural information processing systems*, 119–129.

Blanchard, Peva, El Mahdi El Mhamdi, Rachid Guerraoui, and Julien Stainer. 2017b. Machine learning with adversaries: Byzantine tolerant gradient descent. In *Advances in neural information processing systems 30: Annual conference on neural information processing systems 2017, 4-9 december 2017, long beach, ca, USA*, 118–128.

Bonawitz, Keith, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloe Kiddon, Jakub Konecny, Stefano Mazzocchi, H Brendan McMahan, et al. 2019. Towards federated learning at scale: System design. *arXiv preprint arXiv:1902.01046*.

Bonawitz, Keith, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. 2016. Practical secure aggregation for federated learning on user-held data. *arXiv preprint arXiv:1611.04482*.

———. 2017. Practical secure aggregation for privacy-preserving machine learning. In *Proceedings of the 2017 acm sigsac conference on computer and communications security*, 1175–1191.

Boyer, Robert S, and J Strother Moore. 1991. Mjrty—a fast majority vote algorithm. In *Automated reasoning*, 105–117. Springer.

Brisimi, Theodora S, Ruidi Chen, Theofanie Mela, Alex Olshevsky, Ioannis Ch Paschalidis, and Wei Shi. 2018. Federated learning of predictive models from federated electronic health records. *International journal of medical informatics* 112: 59–67.

Brown, Tom B, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*.

Bubeck, Sébastien, et al. 2015. Convex optimization: Algorithms and complexity. *Foundations and Trends® in Machine Learning* 8(3-4):231–357.

Caldas, Sebastian, Peter Wu, Tian Li, Jakub Konečnỳ, H Brendan McMahan, Virginia Smith, and Ameet Talwalkar. 2018. Leaf: A benchmark for federated settings. *arXiv preprint arXiv:1812.01097*.

Cao, Di, Shan Chang, Zhijian Lin, Guohua Liu, and Donghong Sun. 2019. Understanding distributed poisoning attack in federated learning. In *2019 ieee 25th international conference on parallel and distributed systems (icpads)*, 233–239. IEEE.

Carlini, Nicholas, and David Wagner. 2017. Towards evaluating the robustness of neural networks. In *2017 ieee symposium on security and privacy (sp)*, 39–57. IEEE.

Castro, Miguel, Barbara Liskov, et al. 1999. Practical byzantine fault tolerance. In *Osdi*, vol. 99, 173–186.

Chandrasekaran, Venkat, Benjamin Recht, Pablo A Parrilo, and Alan S Willsky. 2012. The convex geometry of linear inverse problems. *Foundations of Computational mathematics* 12(6):805–849.

Charles, Zachary, Dimitris Papailiopoulos, and Jordan Ellenberg. 2017. Approximate gradient coding via sparse random graphs. *arXiv preprint arXiv:1711.06771*.

Chen, Chia-Yu, Jungwook Choi, Daniel Brand, Ankur Agrawal, Wei Zhang, and Kailash Gopalakrishnan. 2017a. Adacomp: Adaptive residual gradient compression for data-parallel distributed training. *arXiv preprint arXiv:1712.02679*.

Chen, Jianmin, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. 2016a. Revisiting distributed synchronous SGD. *arXiv preprint arXiv:1604.00981*.

Chen, Lingjiao, Hongyi Wang, Zachary Charles, and Dimitris Papailiopoulos. 2018. Draco: Byzantine-resilient distributed training via redundant gradients. *arXiv preprint arXiv:1803.09877*.

Chen, Mingqing, Ananda Theertha Suresh, Rajiv Mathews, Adeline Wong, Cyril Allauzen, Françoise Beaufays, and Michael Riley. 2019. Federated learning of n-gram language models. *Proceedings of the 23rd Conference on Computational Natural Language Learning (CoNLL)*.

Chen, Tianqi, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*.

Chen, Xinyun, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. 2017b. Targeted backdoor attacks on deep learning systems using data poisoning. *arXiv preprint arXiv:1712.05526*.

Chen, Yu-Hsin, Joel Emer, and Vivienne Sze. 2016b. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *ACM SIGARCH Computer Architecture News* 44(3):367–379.

Chen, Yudong, Lili Su, and Jiaming Xu. 2017c. Distributed statistical machine learning in adversarial settings: Byzantine gradient descent. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 1(2):1–25.

———. 2017d. Distributed statistical machine learning in adversarial settings: Byzantine gradient descent. *Proc. ACM Meas. Anal. Comput. Syst.* 1(2).

Chollet, François. 2017. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the ieee conference on computer vision and pattern recognition*, 1251–1258.

Cohen, Gregory, Saeed Afshar, Jonathan Tapson, and Andre Van Schaik. 2017. Emnist: Extending mnist to handwritten letters. In *2017 international joint conference on neural networks (ijcnn)*, 2921–2926. IEEE.

Cotter, Andrew, Ohad Shamir, Nati Srebro, and Karthik Sridharan. 2011. Better mini-batch algorithms via accelerated gradient methods. In *Advances in neural information processing systems*, 1647–1655.

Cretu, Gabriela F, Angelos Stavrou, Michael E Locasto, Salvatore J Stolfo, and Angelos D Keromytis. 2008. Casting out demons: Sanitizing training data for

anomaly sensors. In *2008 ieee symposium on security and privacy (sp 2008)*, 81–95. IEEE.

Dalcin, Lisandro D, Rodrigo R Paz, Pablo A Kler, and Alejandro Cosimo. 2011a. Parallel distributed computing using python. *Advances in Water Resources* 34(9): 1124–1139.

———. 2011b. Parallel distributed computing using python. *Advances in Water Resources* 34(9):1124–1139.

Damaskinos, Georgios, El Mahdi El Mhamdi, Rachid Guerraoui, Arsany Hany Abdelmessih Guirguis, and Sébastien Louis Alexandre Rouault. 2019. Aggregathor: Byzantine machine learning via robust gradient aggregation. In *The conference on systems and machine learning (sysml), 2019*. CONF.

De, Soham, Abhay Yadav, David Jacobs, and Tom Goldstein. 2016. Big Batch SGD: Automated inference using adaptive batch sizes. *arXiv preprint arXiv:1610.05792*.

De Sa, Christopher, Matthew Feldman, Christopher Ré, and Kunle Olukotun. 2017. Understanding and optimizing asynchronous low-precision stochastic gradient descent. In *Proceedings of the 44th annual international symposium on computer architecture*, 561–574. ACM.

De Sa, Christopher, Megan Leszczynski, Jian Zhang, Alana Marzoev, Christopher R Aberger, Kunle Olukotun, and Christopher Ré. 2018. High-accuracy low-precision training. *arXiv preprint arXiv:1803.03383*.

De Sa, Christopher, Christopher Re, and Kunle Olukotun. 2015a. Global convergence of stochastic gradient descent for some non-convex matrix problems. In *International conference on machine learning*, 2332–2341.

De Sa, Christopher M, Ce Zhang, Kunle Olukotun, and Christopher Ré. 2015b. Taming the wild: A unified analysis of hogwild-style algorithms. In *Advances in neural information processing systems*, 2674–2682.

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

Deng, Jia, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 ieee conference on computer vision and pattern recognition*, 248–255. Ieee.

Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Acl*, 4171–4186.

Doshi, Tulsee. 2018. Introducing the inclusive images competition.

Dutta, Sanghamitra, Viveck Cadambe, and Pulkit Grover. 2016. Short-dot: Computing large linear transforms distributedly using coded short dot products. In *Advances in neural information processing systems*, 2100–2108.

———. 2017. Coded convolution for parallel and distributed computing within a deadline. In *2017 ieee international symposium on information theory (isit)*, 2403–2407. IEEE.

El-Mhamdi, El-Mahdi, and Rachid Guerraoui. 2019. Fast and secure distributed learning in high dimension. *arXiv preprint arXiv:1905.04374*.

El-Mhamdi, El-Mahdi, Rachid Guerraoui, Arsany Guirguis, and Sebastien Rouault. 2019. Sgd: Decentralized byzantine resilience. *arXiv preprint arXiv:1905.03853*.

Elliott, Desmond, Stella Frank, Khalil Sima'an, and Lucia Specia. 2016. Multi30k: Multilingual english-german image descriptions. *arXiv preprint arXiv:1605.00459*.

Frankle, Jonathan, and Michael Carbin. 2018. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*.

Ghadimi, Saeed, and Guanghui Lan. 2013. Stochastic first-and zeroth-order methods for nonconvex stochastic programming. *SIAM Journal on Optimization* 23(4): 2341–2368.

Ghahramani, Zoubin, and Thomas L Griffiths. 2005. Infinite latent feature models and the Indian buffet process. In *Advances in neural information processing systems*, 475–482.

Go, Alec, Richa Bhayani, and Lei Huang. Twitter sentiment classification using distant supervision.

Goodfellow, Ian J, Jonathon Shlens, and Christian Szegedy. 2014. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*.

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

Gromov, Mikhail. 2007. *Metric structures for riemannian and non-riemannian spaces*. Springer Science & Business Media.

Grubic, Demjan, Leo Tam, Dan Alistarh, and Ce Zhang. 2018. Synchronous multi-GPU deep learning with low-precision communication: An experimental study.

Gu, Tianyu, Brendan Dolan-Gavitt, and Siddharth Garg. 2017. Badnets: Identifying vulnerabilities in the machine learning model supply chain. *arXiv preprint arXiv:1708.06733*.

Guerraoui, Rachid, Sébastien Rouault, et al. 2018. The hidden vulnerability of distributed learning in byzantium. In *International conference on machine learning*, 3521–3530. PMLR.

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

Hard, Andrew, Kanishka Rao, Rajiv Mathews, Swaroop Ramaswamy, Françoise Beaufays, Sean Augenstein, Hubert Eichner, Chloé Kiddon, and Daniel Ramage. 2018. Federated learning for mobile keyboard prediction. *arXiv preprint arXiv:1811.03604*.

Hawkins, Andrew. 2019. Tesla didn't fix an autopilot problem for three years, and now another person is dead. *theverge.com*.

He, Chaoyang, Songze Li, Jinhyun So, Mi Zhang, Hongyi Wang, Xiaoyang Wang, Praneeth Vepakomma, Abhishek Singh, Hang Qiu, Li Shen, et al. 2020. Fedml: A research library and benchmark for federated machine learning. *arXiv preprint arXiv:2007.13518*.

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

He, Yihui, Xiangyu Zhang, and Jian Sun. 2017. Channel pruning for accelerating very deep neural networks. In *Proceedings of the ieee international conference on computer vision*, 1389–1397.

Hochreiter, Sepp, and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9(8):1735–1780.

Hodge, Victoria, and Jim Austin. 2004. A survey of outlier detection methodologies. *Artificial intelligence review* 22(2):85–126.

Hong, Sanghyun, Varun Chandrasekaran, Yiğitcan Kaya, Tudor Dumitraş, and Nicolas Papernot. 2020. On the effectiveness of mitigating data poisoning attacks with gradient shaping. *arXiv preprint arXiv:2002.11497*.

Howard, Andrew G, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*.

Hsieh, Kevin, Amar Phanishayee, Onur Mutlu, and Phillip B Gibbons. 2019. The non-iid data quagmire of decentralized machine learning. *arXiv preprint arXiv:1910.00189*.

Hu, Hengyuan, Rui Peng, Yu-Wing Tai, and Chi-Keung Tang. 2016. Network trimming: A data-driven neuron pruning approach towards efficient deep architectures. *arXiv preprint arXiv:1607.03250*.

Huang, Ling, Anthony D Joseph, Blaine Nelson, Benjamin IP Rubinstein, and J Doug Tygar. 2011. Adversarial machine learning. In *Proceedings of the 4th acm workshop on security and artificial intelligence*, 43–58.

Hubara, Itay, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks. In *Advances in neural information processing systems*, 4107–4115.

———. 2017. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research* 18(1): 6869–6898.

Iandola, Forrest N, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360*.

Idelbayev, Yerlan, and Miguel A Carreira-Perpinán. 2020. Low-rank compression of neural nets: Learning the rank of each layer. In *Proceedings of the ieee/cvf conference on computer vision and pattern recognition*, 8049–8059.

Ioannou, Yani, Duncan Robertson, Jamie Shotton, Roberto Cipolla, and Antonio Criminisi. 2015. Training cnns with low-rank filters for efficient image classification. *arXiv preprint arXiv:1511.06744*.

Ioffe, Sergey, and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.

Jaderberg, Max, Andrea Vedaldi, and Andrew Zisserman. 2014. Speeding up convolutional neural networks with low rank expansions. *arXiv preprint arXiv:1405.3866*.

Jaggi, Martin, Virginia Smith, Martin Takác, Jonathan Terhorst, Sanjay Krishnan, Thomas Hofmann, and Michael I Jordan. 2014. Communication-efficient distributed dual coordinate ascent. In *Advances in neural information processing systems*, 3068–3076.

Jastrzebski, Stanislaw, Maciej Szymczak, Stanislav Fort, Devansh Arpit, Jacek Tabor, Kyunghyun Cho, and Krzysztof Geras. 2020. The break-even point on optimization trajectories of deep neural networks. *arXiv preprint arXiv:2002.09572*.

Jiang, Yimin, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. A unified architecture for accelerating distributed dnn training in heterogeneous gpu/cpu clusters. In *14th usenix symposium on operating systems design and implementation (osdi 20)*, 463–479.

Johnson, Rie, and Tong Zhang. 2013. Accelerating stochastic gradient descent using predictive variance reduction. In *Advances in neural information processing systems*, 315–323.

Jumper, John, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. 2021. Highly accurate protein structure prediction with alphafold. *Nature* 1–11.

Kairouz, Peter, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Keith Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. 2019. Advances and open problems in federated learning. *arXiv preprint arXiv:1912.04977*.

Karimi, Hamed, Julie Nutini, and Mark Schmidt. 2016. Linear convergence of gradient and proximal-gradient methods under the Polyak-łojasiewicz condition.

In *Joint european conference on machine learning and knowledge discovery in databases*, 795–811. Springer.

Karimireddy, Sai Praneeth, Quentin Rebjock, Sebastian Stich, and Martin Jaggi. 2019. Error feedback fixes signsgd and other gradient compression schemes. In *International conference on machine learning*, 3252–3261.

Katz, Guy, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. 2017. Reluplex: An efficient smt solver for verifying deep neural networks. In *International conference on computer aided verification*, 97–117. Springer.

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

Kim, Yoon. 2014. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*.

Koh, Pang Wei, and Percy Liang. 2017. Understanding black-box predictions via influence functions. In *Proceedings of the 34th international conference on machine learning-volume 70*, 1885–1894. JMLR. org.

Koh, Pang Wei, Jacob Steinhardt, and Percy Liang. 2018. Stronger data poisoning attacks break data sanitization defenses. *arXiv preprint arXiv:1811.00741*.

Konečnỳ, Jakub, Brendan McMahan, and Daniel Ramage. 2015. Federated optimization: Distributed optimization beyond the datacenter. *arXiv preprint arXiv:1511.03575*.

Konečnỳ, Jakub, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. 2016. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492*.

Konečnỳ, Jakub, and Peter Richtárik. 2016. Randomized distributed mean estimation: Accuracy vs communication. *arXiv preprint arXiv:1611.07555*.

———. 2018. Randomized distributed mean estimation: Accuracy vs communication. *Frontiers in Applied Mathematics and Statistics* 4:62.

Kotla, Ramakrishna, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2007. Zyzzyva: speculative byzantine fault tolerance. In *Acm sigops operating systems review*, vol. 41, 45–58. ACM.

Krafcik, John. 2018. The very human challenge of safe driving. *https://medium.com/*.

Krizhevsky, Alex, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images.

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

Kuhn, Harold W. 1955. The Hungarian method for the assignment problem. *Naval Research Logistics (NRL)* 2(1-2):83–97.

Kusetogullari, Huseyin, Amir Yavariabdi, Abbas Cheddad, Håkan Grahn, and Johan Hall. 2019. Ardis: a swedish historical handwritten digit dataset. *Neural Computing and Applications* 1–14.

Lamport, Leslie, Robert Shostak, and Marshall Pease. 1982. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4(3): 382–401.

———. 2019. The byzantine generals problem. In *Concurrency: the works of leslie lamport*, 203–226.

Lan, Zhenzhong, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2019. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*.

Leblond, Rémi, Fabian Pedregosa, and Simon Lacoste-Julien. 2016. ASAGA: asynchronous parallel SAGA. *arXiv preprint arXiv:1606.04809*.

Leclerc, Guillaume, and Aleksander Madry. 2020. The two regimes of deep network training. *arXiv preprint arXiv:2002.10376*.

LeCun, Yann, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proceedings of the IEEE* 86(11): 2278–2324.

Lee, Kangwook, Maximilian Lam, Ramtin Pedarsani, Dimitris Papailiopoulos, and Kannan Ramchandran. 2018a. Speeding up distributed machine learning using codes. *IEEE Transactions on Information Theory* 64(3):1514–1529.

Lee, Kimin, Kibok Lee, Honglak Lee, and Jinwoo Shin. 2018b. A simple unified framework for detecting out-of-distribution samples and adversarial attacks. In *Advances in neural information processing systems*, 7167–7177.

Lessard, Laurent, Xuezhou Zhang, and Xiaojin Zhu. 2018. An optimal control approach to sequential machine teaching. *arXiv preprint arXiv:1810.06175*.

Li, Hao, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. 2016. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*.

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

Li, Shen, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. 2020a. Pytorch distributed: experiences on accelerating data parallel training. *Proceedings of the VLDB Endowment* 13(12):3005–3018.

Li, Songze, Mohammad Ali Maddah-Ali, and A Salman Avestimehr. 2015. Coded mapreduce. In *2015 53rd annual allerton conference on communication, control, and computing (allerton)*, 964–971. IEEE.

Li, Xiang, Kaixuan Huang, Wenhao Yang, Shusen Wang, and Zhihua Zhang. 2020b. On the convergence of fedavg on non-iid data. *International Conference on Learning Representations*.

Liang, Shiyu, Yixuan Li, and Rayadurgam Srikant. 2017. Enhancing the reliability of out-of-distribution image detection in neural networks. *arXiv preprint arXiv:1706.02690*.

Lin, Min, Qiang Chen, and Shuicheng Yan. 2013. Network in network. *arXiv preprint arXiv:1312.4400*.

Lin, Yujun, Song Han, Huizi Mao, Yu Wang, and William J Dally. 2017. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*.

Linial, Nathan, and Zur Luria. 2014. Chernoff's inequality-a very elementary proof. *arXiv preprint arXiv:1403.7739*.

Liu, Ji, Steve Wright, Christopher Ré, Victor Bittorf, and Srikrishna Sridhar. 2014. An asynchronous parallel stochastic coordinate descent algorithm. In *International conference on machine learning*, 469–477. PMLR.

Liu, Yingqi, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. 2017. Trojaning attack on neural networks.

Liu, Zhuang, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. 2018. Rethinking the value of network pruning. *arXiv preprint arXiv:1810.05270*.

Loiola, Eliane Maria, Nair Maria Maia de Abreu, Paulo Oswaldo Boaventura-Netto, Peter Hahn, and Tania Querido. 2007. A survey for the quadratic assignment problem. *European journal of operational research* 176(2):657–690.

Lugosi, Gábor, and Shahar Mendelson. 2019. Sub-gaussian estimators of the mean of a random vector. *The annals of statistics* 47(2):783–794.

Luo, Jian-Hao, Jianxin Wu, and Weiyao Lin. 2017. Thinet: A filter level pruning method for deep neural network compression. In *Proceedings of the ieee international conference on computer vision*, 5058–5066.

Ma, Yuzhe, Robert Nowak, Philippe Rigollet, Xuezhou Zhang, and Xiaojin Zhu. 2018. Teacher improves learning by selecting a training subset. *arXiv preprint arXiv:1802.08946*.

Madry, Aleksander, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2018. Towards deep learning models resistant to adversarial attacks. In *International conference on learning representations*.

Mahloujifar, Saeed, Mohammad Mahmoody, and Ameer Mohammed. 2018. Multi-party poisoning through generalized p-tampering. *arXiv preprint arXiv:1809.03474*.

Mania, Horia, Xinghao Pan, Dimitris Papailiopoulos, Benjamin Recht, Kannan Ramchandran, and Michael I Jordan. 2015. Perturbed iterate analysis for asynchronous stochastic optimization. *arXiv preprint arXiv:1507.06970*.

Mcdonald, Ryan, Mehryar Mohri, Nathan Silberman, Dan Walker, and Gideon S Mann. 2009. Efficient large-scale distributed training of conditional maximum entropy models. In *Advances in neural information processing systems*, 1231–1239.

McMahan, Brendan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. 2017a. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*, 1273–1282. PMLR.

McMahan, H Brendan, Eider Moore, Daniel Ramage, Seth Hampson, et al. 2016. Communication-efficient learning of deep networks from decentralized data. *arXiv preprint arXiv:1602.05629*.

McMahan, H Brendan, Daniel Ramage, Kunal Talwar, and Li Zhang. 2017b. Learning differentially private recurrent language models. *arXiv preprint arXiv:1710.06963*.

Mei, Shike, and Xiaojin Zhu. 2015. Using machine teaching to identify optimal training-set attacks on machine learners. In *Twenty-ninth aaai conference on artificial intelligence*.

———. 2017. Some submodular data-poisoning attacks on machine learners. Tech. Rep.

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

Minsker, Stanislav, et al. 2015. Geometric median and robust estimation in banach spaces. *Bernoulli* 21(4):2308–2335.

Mohri, Mehryar, Gary Sivek, and Ananda Theertha Suresh. 2019. Agnostic federated learning. *arXiv preprint arXiv:1902.00146*.

Moosavi-Dezfooli, Seyed-Mohsen, Alhussein Fawzi, and Pascal Frossard. 2016. Deepfool: a simple and accurate method to fool deep neural networks. In *Proceedings of the ieee conference on computer vision and pattern recognition*, 2574–2582.

Narayanan, Deepak, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th acm symposium on operating systems principles*, 1–15.

Netzer, Yuval, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. 2011. Reading digits in natural images with unsupervised feature learning. In *Nips workshop on deep learning and unsupervised feature learning*, vol. 2011, 5.

Pang, Bo, and Lillian Lee. 2005. Seeing stars: Exploiting class relationships for sentiment categorization with respect to rating scales. In *Proceedings of the 43rd*

annual meeting on association for computational linguistics, 115–124. Association for Computational Linguistics.

Papernot, Nicolas, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. 2016. The limitations of deep learning in adversarial settings. In *2016 ieee european symposium on security and privacy (euros&p)*, 372–387. IEEE.

Paszke, Adam, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch.

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

Pelekis, Christos, and Jan Ramon. 2015. Hoeffding's inequality for sums of weakly dependent random variables. *arXiv preprint arXiv:1507.06871*.

Peyré, Gabriel, Marco Cuturi, and Justin Solomon. 2016. Gromov-wasserstein averaging of kernel and distance matrices. In *International conference on machine learning*, 2664–2672.

Pippenger, Nicholas. 1988. Reliable computation by formulas in the presence of noise. *IEEE Transactions on Information Theory* 34(2):194–197.

Press, Ofir, and Lior Wolf. 2016. Using the output embedding to improve language models. *arXiv preprint arXiv:1608.05859*.

Qi, Hang, Evan R Sparks, and Ameet Talwalkar. 2016. Paleo: A performance model for deep neural networks.

Rajput, Shashank, Hongyi Wang, Zachary Charles, and Dimitris Papailiopoulos. 2019. Detox: A redundancy-based framework for faster and more robust gradient aggregation. In *Advances in neural information processing systems*, 10320–10330.

Ramaswamy, Swaroop, Rajiv Mathews, Kanishka Rao, and Françoise Beaufays. 2019. Federated learning for emoji prediction in a mobile keyboard. *arXiv preprint arXiv:1906.04329*.

Rasley, Jeff, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th acm sigkdd international conference on knowledge discovery & data mining*, 3505–3506.

Rastegari, Mohammad, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*, 525–542. Springer.

Raviv, Netanel, Itzhak Tamo, Rashish Tandon, and Alexandros G Dimakis. 2017. Gradient coding from cyclic mds codes and expander graphs. *arXiv preprint arXiv:1707.03858*.

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

Reddi, Sashank J, Ahmed Hefny, Suvrit Sra, Barnabas Poczos, and Alex Smola. 2016. Stochastic variance reduction for nonconvex optimization. In *International conference on machine learning*, 314–323.

Reddi, Sashank J., Satyen Kale, and Sanjiv Kumar. 2018. On the convergence of adam and beyond. In *International conference on learning representations*.

Reisizadeh, Amirhossein, Saurav Prakash, Ramtin Pedarsani, and Amir Salman Avestimehr. 2019. Coded computation over heterogeneous clusters. *IEEE Transactions on Information Theory*.

Renggli, Cèdric, Dan Alistarh, and Torsten Hoefler. 2018. SparCML: high-performance sparse communication for machine learning. *arXiv preprint arXiv:1802.08021*.

Rieke, Nicola, Jonny Hancox, Wenqi Li, Fausto Milletari, Holger Roth, Shadi Albarqouni, Spyridon Bakas, Mathieu N Galtier, Bennett Landman, Klaus Maier-Hein, et al. 2020. The future of digital health with federated learning. *arXiv preprint arXiv:2003.08119*.

Rubinstein, Benjamin IP, Blaine Nelson, Ling Huang, Anthony D Joseph, Shing-hon Lau, Satish Rao, Nina Taft, and J Doug Tygar. 2009. Antidote: understanding and defending against poisoning of anomaly detectors. In *Proceedings of the 9th acm sigcomm conference on internet measurement*, 1–14.

Russakovsky, Olga. Strategies for mitigating social bias in deep learning systems. Invited talk at Identifying and Understanding Deep Learning Phenomena workshop, ICML 2019.

Russakovsky, Olga, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. 2015. Imagenet large scale visual recognition challenge. *International journal of computer vision* 115(3):211–252.

Sahu, Anit Kumar, Tian Li, Maziar Sanjabi, Manzil Zaheer, Ameet Talwalkar, and Virginia Smith. 2018. On the convergence of federated optimization in heterogeneous networks. *arXiv preprint arXiv:1812.06127*.

Sainath, Tara N, Brian Kingsbury, Vikas Sindhwani, Ebru Arisoy, and Bhuvana Ramabhadran. 2013. Low-rank matrix factorization for deep neural network training with high-dimensional output targets. In *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*, 6655–6659. IEEE.

Sattler, Felix, Simon Wiedemann, Klaus-Robert Müller, and Wojciech Samek. 2019. Robust and communication-efficient federated learning from non-iid data. *IEEE transactions on neural networks and learning systems*.

Seide, Frank, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 2014. 1-bit stochastic gradient descent and its application to data-parallel distributed training of

speech dnns. In *Fifteenth annual conference of the international speech communication association*.

Sergeev, Alexander, and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*.

Shah, Nihar B, Kangwook Lee, and Kannan Ramchandran. 2016. When do redundant requests reduce latency? *IEEE Transactions on Communications* 64(2): 715–722.

Shamir, Ohad, and Nathan Srebro. 2014. Distributed stochastic optimization and learning. In *2014 52nd annual allerton conference on communication, control, and computing (allerton)*, 850–857. IEEE.

Shankar, Shreya, Yoni Halpern, Eric Breck, James Atwood, Jimbo Wilson, and D Sculley. 2017. No classification without representation: Assessing geodiversity issues in open data sets for the developing world. *arXiv preprint arXiv:1711.08536*.

Sim, Khe Chai, Francoise Beaufays, Arnaud Benard, Dhruv Guliani, Andreas Kabel, Nikhil Khare, Tamar Lucassen, Petr Zadrazil, Harry Zhang, Leif Johnson, and et al. 2019. Personalization of end-to-end speech recognition on mobile devices for named entities. *2019 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*.

Simonite, Tom. 2018. When it comes to gorillas, google photos remains blind. *wired.com*.

Simonyan, Karen, and Andrew Zisserman. 2015. Very deep convolutional networks for large-scale image recognition. In *International conference on learning representations*.

Singh, Sidak Pal, and Martin Jaggi. 2019. Model fusion via optimal transport. *arXiv preprint arXiv:1910.05653*.

Sinha, Aman, Hongseok Namkoong, and John Duchi. 2018. Certifying some distributional robustness with principled adversarial training. In *International conference on learning representations*.

Smith, Virginia, Chao-Kai Chiang, Maziar Sanjabi, and Ameet S Talwalkar. 2017. Federated multi-task learning. In *Advances in neural information processing systems*, 4424–4434.

Sohn, Jy-yong, Dong-Jun Han, Beongjun Choi, and Jaekyun Moon. 2019. Election coding for distributed learning: Protecting signsgd against byzantine attacks. *arXiv preprint arXiv:1910.06093*.

Steinhardt, Jacob, Pang Wei W Koh, and Percy S Liang. 2017. Certified defenses for data poisoning attacks. In *Advances in neural information processing systems*, 3517–3529.

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

Stich, Sebastian U, Jean-Baptiste Cordonnier, and Martin Jaggi. 2018. Sparsified sgd with memory. In *Advances in neural information processing systems*, 4447–4458.

Strom, Nikko. 2015. Scalable distributed DNN training using commodity gpu cloud computing. In *Sixteenth annual conference of the international speech communication association*.

Su, Lili, and Nitin H. Vaidya. 2016a. Non-bayesian learning in the presence of byzantine agents. In *Distributed computing*, ed. Cyril Gavoille and David Ilcinkas, 414–427. Berlin, Heidelberg: Springer Berlin Heidelberg.

———. 2016b. Robust multi-agent optimization: Coping with byzantine agents with input redundancy. In *Stabilization, safety, and security of distributed systems*, ed. Borzoo Bonakdarpour and Franck Petit, 368–382. Cham: Springer International Publishing.

Sun, Ziteng, Peter Kairouz, Ananda Theertha Suresh, and H Brendan McMahan. 2019. Can you really backdoor federated learning? *arXiv preprint arXiv:1911.07963*.

Suresh, Ananda Theertha, Felix X Yu, Sanjiv Kumar, and H Brendan McMahan. 2016. Distributed mean estimation with limited communication. *arXiv preprint arXiv:1611.00429*.

———. 2017. Distributed mean estimation with limited communication. In *Proceedings of the 34th international conference on machine learning-volume 70*, 3329–3337. JMLR. org.

Tan, Mingxing, and Quoc V Le. 2019. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946*.

Tandon, Rashish, Qi Lei, Alexandros G Dimakis, and Nikos Karampatziakis. 2017. Gradient coding: Avoiding stragglers in distributed learning. In *International conference on machine learning*, 3368–3376.

Tang, Hanlin, Chen Yu, Xiangru Lian, Tong Zhang, and Ji Liu. 2019. Doublesqueeze: Parallel stochastic gradient descent with double-pass error-compensated compression. In *International conference on machine learning*, 6155–6165. PMLR.

Thakur, Rajeev, Rolf Rabenseifner, and William Gropp. 2005. Optimization of collective communication operations in mpich. *The International Journal of High Performance Computing Applications* 19(1):49–66.

Thibaux, Romain, and Michael I Jordan. 2007. Hierarchical Beta processes and the Indian buffet process. In *Artificial intelligence and statistics*, 564–571.

Touvron, Hugo, Andrea Vedaldi, Matthijs Douze, and Hervé Jégou. 2020. Fixing the train-test resolution discrepancy: Fixefficientnet. *arXiv preprint arXiv:2003.08237*.

Tsuzuku, Yusuke, Hiroto Imachi, and Takuya Akiba. 2018. Variance-based gradient compression for efficient distributed deep learning. *arXiv preprint arXiv:1802.06058*.

Tucker, Ledyard R. 1966. Some mathematical notes on three-mode factor analysis. *Psychometrika* 31(3):279–311.

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

Vincent, James. The iphone x's new neural engine exemplifies apple's approach to ai. `https://www.theverge.com/2017/9/13/16300464/apple-iphone-x-ai-neural-engine`.

Vogels, Thijs, Sai Praneeth Karimireddy, and Martin Jaggi. 2019. Powersgd: Practical low-rank gradient compression for distributed optimization. In *Advances in neural information processing systems*, 14259–14268.

Von Neumann, John. 1956. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata studies* 34:43–98.

Waleffe, Roger, and Theodoros Rekatsinas. 2020. Principal component networks: Parameter reduction early in training. *arXiv preprint arXiv:2006.13347*.

Wang, Hongyi, Saurabh Agarwal, and Dimitris Papailiopoulos. 2021a. Pufferfish: Communication-efficient models at no extra cost. *Proceedings of Machine Learning and Systems* 3.

Wang, Hongyi, Scott Sievert, Shengchao Liu, Zachary Charles, Dimitris Papailiopoulos, and Stephen Wright. 2018. Atomo: Communication-efficient learning via atomic sparsification. In *Advances in neural information processing systems*, 9850–9861.

Wang, Hongyi, Kartik Sreenivasan, Shashank Rajput, Harit Vishwakarma, Saurabh Agarwal, Jy-yong Sohn, Kangwook Lee, and Dimitris Papailiopoulos. 2020a. Attack of the tails: Yes, you really can backdoor federated learning. *arXiv preprint arXiv:2007.05084*.

Wang, Hongyi, Mikhail Yurochkin, Yuekai Sun, Dimitris Papailiopoulos, and Yasaman Khazaeni. 2020b. Federated learning with matched averaging. In *International conference on learning representations*.

Wang, Jianyu, Zachary Charles, Zheng Xu, Gauri Joshi, H Brendan McMahan, Maruan Al-Shedivat, Galen Andrew, Salman Avestimehr, Katharine Daly, Deepesh Data, et al. 2021b. A field guide to federated optimization. *arXiv preprint arXiv:2107.06917*.

Wangni, Jianqiao, Jialei Wang, Ji Liu, and Tong Zhang. 2018. Gradient sparsification for communication-efficient distributed optimization. In *Advances in neural information processing systems*, 1299–1309.

Wen, Wei, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning structured sparsity in deep neural networks. In *Advances in neural information processing systems*, 2074–2082.

Wen, Wei, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2017. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Advances in neural information processing systems*, 1508–1518.

Wiesler, Simon, Alexander Richard, Ralf Schluter, and Hermann Ney. 2014. Mean-normalized stochastic gradient for large-scale deep learning. In *Acoustics, speech and signal processing (icassp), 2014 ieee international conference on*, 180–184. IEEE.

Wu, Jiaxiang, Weidong Huang, Junzhou Huang, and Tong Zhang. 2018. Error compensated quantized sgd and its applications to large-scale distributed optimization. In *International conference on machine learning*, 5325–5333.

Wu, Jiaxiang, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. 2016. Quantized convolutional neural networks for mobile devices. In *Proceedings of the ieee conference on computer vision and pattern recognition*, 4820–4828.

Xie, Cong. 2019. Zeno++: robust asynchronous sgd with arbitrary number of byzantine workers. *arXiv preprint arXiv:1903.07020*.

Xie, Cong, Oluwasanmi Koyejo, and Indranil Gupta. 2018a. Generalized byzantine-tolerant sgd. *arXiv preprint arXiv:1802.10116*.

———. 2018b. Zeno: Distributed stochastic gradient descent with suspicion-based fault-tolerance. *arXiv preprint arXiv:1805.10032*.

Xie, Cong, Sanmi Koyejo, and Indranil Gupta. 2019a. Fall of empires: Breaking byzantine-tolerant sgd by inner product manipulation. *arXiv preprint arXiv:1903.03936*.

———. 2019b. Practical distributed learning: Secure machine learning with communication-efficient local updates. *arXiv preprint arXiv:1903.06996*.

Xu, Jie, and Fei Wang. 2019. Federated learning for healthcare informatics. *arXiv preprint arXiv:1911.06270*.

Xue, Jian, Jinyu Li, and Yifan Gong. 2013. Restructuring of deep neural network acoustic models with singular value decomposition. In *Interspeech*, 2365–2369.

Yang, Tien-Ju, Yu-Hsin Chen, and Vivienne Sze. 2017a. Designing energy-efficient convolutional neural networks using energy-aware pruning. In *Proceedings of the ieee conference on computer vision and pattern recognition*, 5687–5695.

Yang, Timothy, Galen Andrew, Hubert Eichner, Haicheng Sun, Wei Li, Nicholas Kong, Daniel Ramage, and Françoise Beaufays. 2018. Applied federated learning: Improving google keyboard query suggestions. *arXiv preprint arXiv:1812.02903*.

Yang, Yaoqing, Pulkit Grover, and Soummya Kar. 2017b. Coded distributed computing for inverse problems. In *Advances in neural information processing systems*, 709–719.

Yin, Dong, Yudong Chen, Ramchandran Kannan, and Peter Bartlett. 2019. Defending against saddle point attack in byzantine-robust distributed learning. In *International conference on machine learning*, 7074–7084.

Yin, Dong, Yudong Chen, Kannan Ramchandran, and Peter Bartlett. 2018a. Byzantine-robust distributed learning: Towards optimal statistical rates. *arXiv preprint arXiv:1803.01498*.

———. 2018b. Defending against saddle point attack in byzantine-robust distributed learning. *CoRR* abs/1806.05358. 1806.05358.

Yin, Dong, Ashwin Pananjady, Max Lam, Dimitris Papailiopoulos, Kannan Ramchandran, and Peter Bartlett. 2018c. Gradient diversity: a key ingredient for scalable distributed learning. In *International conference on artificial intelligence and statistics*, 1998–2007.

You, Haoran, Chaojian Li, Pengfei Xu, Yonggan Fu, Yue Wang, Xiaohan Chen, Richard G Baraniuk, Zhangyang Wang, and Yingyan Lin. 2019. Drawing early-bird tickets: Towards more efficient training of deep networks. *arXiv preprint arXiv:1909.11957*.

Yu, Jiahui, Linjie Yang, Ning Xu, Jianchao Yang, and Thomas Huang. 2018a. Slimmable neural networks. *arXiv preprint arXiv:1812.08928*.

Yu, Ruichi, Ang Li, Chun-Fu Chen, Jui-Hsin Lai, Vlad I Morariu, Xintong Han, Mingfei Gao, Ching-Yung Lin, and Larry S Davis. 2018b. Nisp: Pruning networks using neuron importance score propagation. In *Proceedings of the ieee conference on computer vision and pattern recognition*, 9194–9203.

Yuan, Binhang, Song Ge, and Wenhui Xing. 2020. A federated learning framework for healthcare iot devices. *arXiv preprint arXiv:2005.05083*.

Yurochkin, Mikhail, Mayank Agarwal, Soumya Ghosh, Kristjan Greenewald, and Nghia Hoang. 2019a. Statistical model aggregation via parameter matching. In *Advances in neural information processing systems*, 10954–10964.

Yurochkin, Mikhail, Mayank Agarwal, Soumya Ghosh, Kristjan Greenewald, Trong Nghia Hoang, and Yasaman Khazaeni. 2019b. Bayesian nonparametric federated learning of neural networks. *arXiv preprint arXiv:1905.12022*.

Yurochkin, Mikhail, Zhiwei Fan, Aritra Guha, Paraschos Koutris, and XuanLong Nguyen. 2019c. Scalable inference of topic evolution via models for latent geometric structures. In *Advances in neural information processing systems*, 5949–5959.

Zafar, Muhammad Bilal, Isabel Valera, Manuel Rodriguez, Krishna Gummadi, and Adrian Weller. 2017. From parity to preference-based notions of fairness in classification. In *Advances in neural information processing systems*, 229–239.

Zagoruyko, Sergey, and Nikos Komodakis. 2016. Wide residual networks. *arXiv preprint arXiv:1605.07146*.

Zaharia, Matei, Andy Konwinski, Anthony D Joseph, Randy H Katz, and Ion Stoica. 2008. Improving mapreduce performance in heterogeneous environments. In *Osdi*, vol. 8, 7.

Zhang, Chiyuan, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. 2017a. Understanding deep learning requires rethinking generalization. In *5th international conference on learning representations, ICLR*.

Zhang, Hantian, Jerry Li, Kaan Kara, Dan Alistarh, Ji Liu, and Ce Zhang. 2017b. Zipml: Training linear models with end-to-end low precision, and a little bit of deep learning. In *International conference on machine learning*, 4035–4043.

Zhang, Jiale, Junjun Chen, Di Wu, Bing Chen, and Shui Yu. 2019. Poisoning attack in federated learning using generative adversarial nets. In *2019 18th ieee international conference on trust, security and privacy in computing and communications/13th ieee international conference on big data science and engineering (trustcom/bigdatase)*, 374–380. IEEE.

Zhang, Jian, Christopher De Sa, Ioannis Mitliagkas, and Christopher Ré. 2016. Parallel sgd: When does averaging help? *arXiv preprint arXiv:1606.07365*.

Zhang, Xiangyu, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2018. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the ieee conference on computer vision and pattern recognition*, 6848–6856.

Zhou, Aojun, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. 2017. Incremental network quantization: Towards lossless cnns with low-precision weights. *arXiv preprint arXiv:1702.03044*.

Zhou, Denny, Mao Ye, Chen Chen, Tianjian Meng, Mingxing Tan, Xiaodan Song, Quoc Le, Qiang Liu, and Dale Schuurmans. 2020. Go wide, then narrow: Efficient training of deep thin networks. *arXiv preprint arXiv:2007.00811*.

Zhou, Shuchang, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. 2016. DoReFa-Net: training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*.

Zhu, Chenzhuo, Song Han, Huizi Mao, and William J Dally. 2016. Trained ternary quantization. *arXiv preprint arXiv:1612.01064*.

Zhu, Jerry. 2013. Machine teaching for bayesian learners in the exponential family. In *Advances in neural information processing systems*, 1905–1913.

Zhu, Michael, and Suyog Gupta. 2017. To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878*.

Zhu, Xiaojin, Adish Singla, Sandra Zilles, and Anna N Rafferty. 2018. An overview of machine teaching. *arXiv preprint arXiv:1801.05927*.

Zinkevich, Martin, Markus Weimer, Lihong Li, and Alex J Smola. 2010. Parallelized stochastic gradient descent. In *Advances in neural information processing systems*, 2595–2603.