DRACO: Byzantine-resilient Distributed Training via Redundant Gradients

Lingjiao Chen [1] **Hongyi Wang** [1] **Zachary Charles** [1] **Dimitris Papailiopoulos** [1]

Abstract

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 work, 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.

1. Introduction

Distributed and parallel implementations of stochastic optimization algorithms have become the de facto standard in large-scale model training (Li et al., 2014; Recht et al., 2011; Zhang et al., 2015; Agarwal et al., 2010; Abadi et al., 2016; Chen et al., 2015; Paszke et al., 2017a; Chilimbi et al., 2014). Due to increasingly common malicious attacks, hardware and software errors (Castro et al., 1999; Kotla et al., 2007; Blanchard et al., 2017; Chen et al., 2017), protecting distributed machine learning against adversarial attacks and failures has become increasingly important. Unfortunately, even a single adversarial node in a distributed setup can introduce arbitrary bias and inaccuracies to the end model(Blanchard et al., 2017).

A recent line of work (Blanchard et al., 2017; Chen et al., 2017) studies this problem 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. Many of the aforementioned work use median-based aggregation, including the geometric median (GM) instead of averaging in order to make their computations more robust. The advantage of median-based approaches is that they can be robust to up to a constant fraction of the compute nodes being adversarial (Chen et al., 2017). However, in large data settings, the cost of computing the geometric median can dwarf the cost of computing a batch of gradients (Chen et al., 2017), 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 paper, we instead use ideas from coding theory to ensure robustness during distributed training. We present DRACO, 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 during training and returns a model *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., 2017; Chen et al., 2017). Moreover, in median-based approaches such as (Blanchard et al., 2017; Chen et al., 2017), 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.

To design DRACO, we borrow ideas from coding theory and algorithmic redundancy. In standard adversary-free dis-

[1]University of Wisconsin-Madison. Correspondence to: Lingjiao Chen <lchen@cs.wisc.edu>.

Proceedings of the 35^{th} International Conference on Machine Learning, Stockholm, Sweden, PMLR 80, 2018.

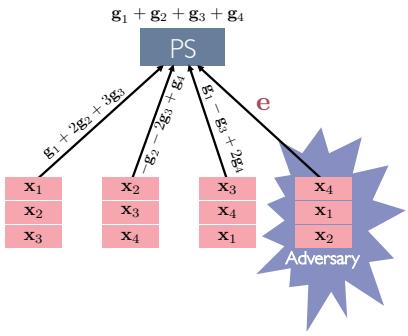


Figure 1. The high level idea behind DRACO's algorithmic redundancy. Suppose we have 4 data points $\mathbf{x}_1, \ldots, \mathbf{x}_4$, and let $\mathbf{g}_i$ be the gradient of the model with respect to data point $\mathbf{x}_i$. Instead of having each compute node i evaluate a single gradient $\mathbf{g}_i$, DRACO assigns each node redundant gradients. In this example, the replication ratio is 3, and the parameter server 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.

tributed computation setups, during each distributed round, each of the P compute nodes processes B/P gradients and ships their sum to the parameter server. In DRACO, each compute node processes 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 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 as we show in experiments in Section 4.

We implement DRACO in PyTorch and deploy it on distributed setups on Amazon EC2, where we compare against median-based training algorithms on several real world datasets and various ML models. We show that DRACO is up to orders of magnitude faster compared to GM-based approaches across a range of neural networks, *e.g.*, LeNet, VGG-19, AlexNet, ResNet-18, and ResNet-152, and always converges to the correct adversary-free model, while in some cases median-based approaches do not converge.

Related Work 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; Alistarh et al., 2017; Jaggi et al., 2014; Liu et al., 2014; Mania et al., 2015; Chen et al., 2016). Much of this work focuses on developing and analyzing efficient distributed training algorithms. This work shares ideas with *federated learning*, 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 (Lee et al., 2017; Li et al., 2015; Dutta et al., 2016; 2017; Reisizadeh et al., 2017; Yang et al., 2017). 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 (Raviv et al., 2017; Charles et al., 2017; Cotter et al., 2011). Our work builds on and extends this work to the adversarial setup. Mitigating adversaries can often be more difficult than mitigating stragglers since in the adversarial setup 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 (Lamport et al., 1982). There has been substantial amounts of work recently on byzantine fault tolerance in distributed training which shows that while average-based gradient methods are susceptible to adversarial nodes (Blanchard et al., 2017; Chen et al., 2017), median-based update methods can achieve good convergence while being robust to adversarial nodes. Both (Blanchard et al., 2017) and (Chen et al., 2017) use variants of the geometric median to improve the tolerance of first-order methods against adversarial nodes. Unfortunately, convergence analyses of median approaches often require restrictive assumptions and algorithm-specific proofs of convergence. Furthermore, the geometric median aggregation

may dominate the training time in large-scale settings.

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. Our work can be seen as an application of the aforementioned concepts to the context of distributed training in the face of adversity.

2. 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):

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

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 work, 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 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 & 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., 2017).

In this work, 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.

3. DRACO: Robust Distributed Training via Algorithmic Redundancy

In this section we present our main results for DRACO. Due to space constraints, all proofs are left to the supplement.

We generalize the scheme in Figure 1 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 $supp\,(\mathbf{A}_{j,\cdot})$, 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 3.1. $r \triangleq \frac{1}{P}\|\mathbf{A}\|_0$ *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 $\mathbf{Y}_j \triangleq (\mathbf{1}_d \mathbf{A}_{j,\cdot}) \odot \mathbf{G}$ 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 2.

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 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 $\mathbf{G}\mathbf{1}_P$ via DRACO with any s adversarial nodes. Below is the formal definition.

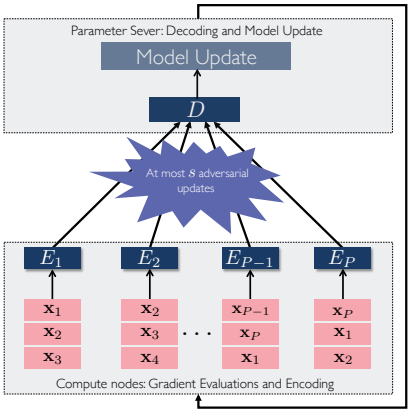


Figure 2. In DRACO, each compute node is allocated a subset of the data set. Each compute node computes redundant gradients, encodes them via E_i, and sends the resulting vector to the PS. These received vectors then pass through a decoder that detects where the adversaries are and removes their effects 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.

Definition 3.2. 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* $|\{j : \mathbf{n}_j \neq 0\}| \leq s$, *we have* $D(\mathbf{Z}^{\mathbf{A},E,\mathbf{G}} + \mathbf{N}) = \mathbf{G}\mathbf{1}_P$.

Remark 2. *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 3.1. *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* $r \geq 2s + 1$.

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 3. *Suppose that a tuple* $(\mathbf{A}, E, D)$ *can tolerate any* s *adversarial nodes. By Theorem 3.1, 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., 2017), 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 & Moore, 1991), the complexity of which is linear in P.

Theorem 3.2. *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 $\forall j, \ell, \mathbf{A}_{j,\ell}^{Cyc} \neq 0$ implies $\mathbf{W}_{j,\ell} = 0$. Let $\mathbf{Y}_j^{Cyc} = \left(\mathbf{1}_d \mathbf{A}_{j,\cdot}^{Cyc}\right) \odot \mathbf{G}$ denote the gradients computed at compute node j. The local encoding function E_j^{Cyc} is defined by $E_j^{Cyc}(\mathbf{Y}_j^{Cyc}) = \mathbf{G}\mathbf{W}_{\cdot,j}$. After performing this local encoding, the jth compute node then sends $\mathbf{z}_j^{Cyc} \triangleq E_j^{Cyc}(\mathbf{Y}_j^{Cyc})$ to the PS. Let $\mathbf{Z}^{\mathbf{A}^{Cyc}, E^{Cyc}, \mathbf{G}} \triangleq [\mathbf{z}_1^{Cyc}, \mathbf{z}_2^{Cyc}, \cdots, \mathbf{z}_P^{Cyc}]$. Then one can verify from the definition of E_j^{Cyc} that $\mathbf{Z}^{\mathbf{A}^{Cyc}, E^{Cyc}, \mathbf{G}} = \mathbf{G}\mathbf{W}$. The received matrix at the PS now becomes $\mathbf{R}^{Cyc} = \mathbf{Z}^{\mathbf{A}^{Cyc}, E^{Cyc}, \mathbf{G}} + \mathbf{N} = \mathbf{G}\mathbf{W} + \mathbf{N}$.

In order to decode, the PS needs to detect which com-

pute 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 $\mathbf{R}^{Cyc}_{\cdot,U}\mathbf{b} = (\mathbf{GW}+\mathbf{N})_{\cdot,U}\mathbf{b} = \mathbf{GW}_{\cdot,U}\mathbf{b} = \mathbf{G1}_P$. The decoder function is given formally in Algorithm 1.

Algorithm 1 Decoder Function D^{Cyc}.

Input : Received $d \times P$ matrix $\mathbf{R}^{Cyc}$
Output : Desired gradient summation $\mathbf{u}^{Cyc}$
$V = \phi(\mathbf{R})$ // 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 $\mathbf{u}^{Cyc} = \mathbf{R}_{\cdot,U}\mathbf{b}$

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.,

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

Let $\mathbf{C}_L$ be the first $P - 2s$ rows of $\mathbf{C}$ and $\mathbf{C}_R$ be the last $2s$ rows. Let α_j be the set of row indices of the zero entries in $\mathbf{A}^{Cyc}_{\cdot,j}$, *i*.e., $\alpha_j = \{k : \mathbf{A}^{Cyc}_{j,k} = 0\}$. 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 3.3. *For all j, k, $\mathbf{A}_{j,k} = 0 \Rightarrow \mathbf{W}_{j,k} = 0$. For any index set U such that $|U| \geq P - (2s+1)$, the column span of $\mathbf{W}_{\cdot,U}$ 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 $[h_{P-2s}, h_{P-2s-1}, \cdots, h_{P-1}] \triangleq \mathbf{fRC}_R^{\dagger}$[1]. We then obtain a vector $\beta = [\beta_0, \beta_1, \cdots, \beta_{s-1}]^T$ by solving

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

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 3.4. *Suppose $\mathbf{N} = [\mathbf{n}_1, \mathbf{n}_2, \cdots, \mathbf{n}_P]$ satisfies $|\{j : \|\mathbf{n}_j\|_0 \neq 0\}| \leq s$. Then $\phi(\mathbf{R}^{Cyc}) = \phi(\mathbf{GW}+\mathbf{N}) = \{j : \|\mathbf{n}_j\|_0 \neq 0\}$ 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.

Theorem 3.5. *The cyclic code $(\mathbf{A}^{Cyc}, E^{Cyc}, D^{Cyc})$ 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.*

Note that the cyclic code requires transmitting complex vectors $\mathbf{GW}$ which potentially doubles the bandwidth requirement. To handle this problem, one can transform the original real gradient $\mathbf{G} \in \mathbb{R}^{d\times P}$ into a complex gradient $\hat{\mathbf{G}} \in \mathbb{C}^{\lceil d/2 \rceil \times P}$ by letting its ith component have real part $\mathbf{G}_i$ and complex part $\mathbf{G}_{\lceil d/2 \rceil + i}$. Then the compute nodes only need to send $\hat{\mathbf{G}}\mathbf{W}$. Once the PS recovers $\hat{\mathbf{u}}^{Cyc} \triangleq \hat{\mathbf{G}}\mathbf{1}_P$, it can simply sum the real and imaginary parts to form the true gradient summation, *i*.e., $\mathbf{u}^{Cyc} = \mathrm{Re}(\hat{\mathbf{u}}^{Cyc}) + \mathrm{Im}(\hat{\mathbf{u}}^{Cyc}) = \mathbf{G1}_P$.

4. Experiments

In this section we present an empirical study of DRACO and compare it to the median-based approach in (Chen et al., 2017) 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., 2017) 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., 2017). In mini-batch

[1] † denotes transpose conjugate.

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., 2017b) with MPI4py (Dalcin et al., 2011) 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 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 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 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 & Hinton, 2009), and CNN-rand-non-static (CRN) model in (Kim, 2014) for Movie Review (MR) (Pang & 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 10,000 distributed iterations. Figure 3 shows how the testing accuracy varies with training time. Tables 2 and 3 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). 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.

Table 2. 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 3. 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.

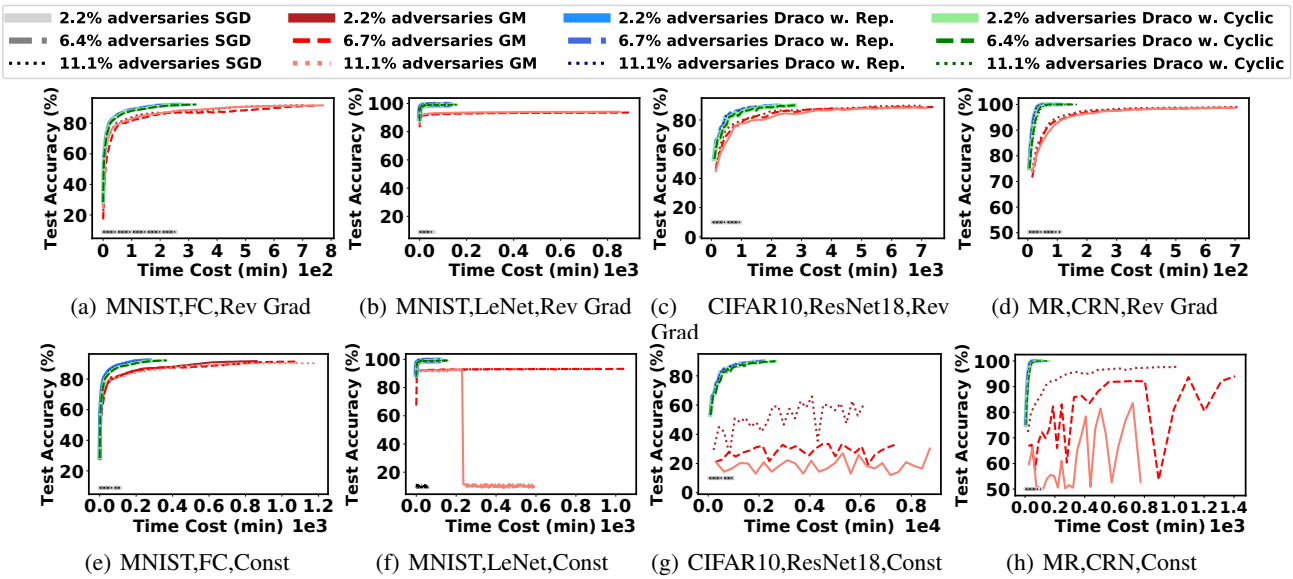
Test Accuracy	95%	96%	98%	98.5%
2.2% rev grad	**5.4/4.2**	**5.6/4.3**	**9.7/7.4**	**12/9.0**
6.7% rev grad	**6.4/4.5**	**6.3/4.5**	**11/8.1**	**19/13**
11.1% rev grad	**7.5/4.7**	**7.4/4.6**	**12/8**	**19/12**

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.

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

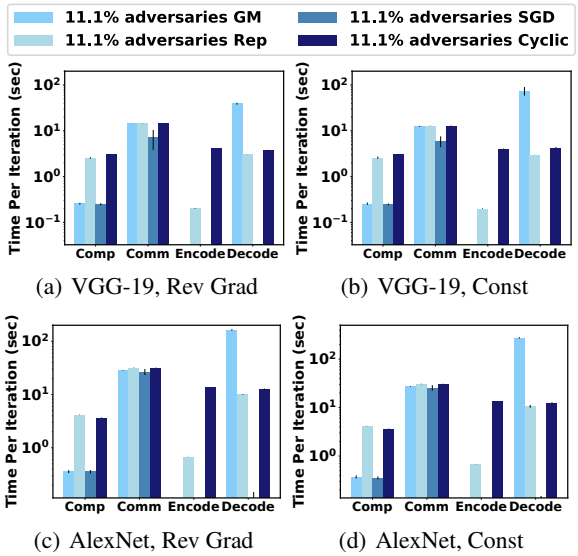
[2] https://github.com/hwang595/Draco



(a) MNIST,FC,Rev Grad (b) MNIST,LeNet,Rev Grad (c) CIFAR10,ResNet18,Rev Grad (d) MR,CRN,Rev Grad

(e) MNIST,FC,Const (f) MNIST,LeNet,Const (g) CIFAR10,ResNet18,Const (h) MR,CRN,Const

Figure 3. 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.



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

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

Figure 4. 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.

(He et al., 2016; Simonyan & Zisserman, 2014; 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 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 4, 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. 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.

5. Conclusion and Open Problems

In this work 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.

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 (1) tolerate more stragglers or (2) 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.

Acknowledgement

This work was supported in part by a gift from Google and AWS Cloud Credits for Research from Amazon. We thank Jeffrey Naughton and Remzi Arpaci-Dusseau for invaluable discussions and feedback on earlier drafts of this paper.

References

Abadi, Martín, Barham, Paul, Chen, Jianmin, Chen, Zhifeng, Davis, Andy, Dean, Jeffrey, Devin, Matthieu, Ghemawat, Sanjay, Irving, Geoffrey, Isard, Michael, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pp. 265–283, 2016.

Agarwal, Alekh, Wainwright, Martin J, and Duchi, John C. Distributed dual averaging in networks. In *NIPS*, pp. 550–558, 2010.

Alistarh, Dan, Grubic, Demjan, Li, Jerry, Tomioka, Ryota, and Vojnovic, Milan. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In *NIPS*, pp. 1707–1718, 2017.

Blanchard, Peva, Guerraoui, Rachid, Stainer, Julien, et al. Machine learning with adversaries: Byzantine tolerant gradient descent. In *NIPS*, pp. 118–128, 2017.

Bonawitz, Keith, Ivanov, Vladimir, Kreuter, Ben, Marcedone, Antonio, McMahan, H Brendan, Patel, Sarvar, Ramage, Daniel, Segal, Aaron, and Seth, Karn. Practical secure aggregation for federated learning on user-held data. *arXiv preprint arXiv:1611.04482*, 2016.

Boyer, Robert S and Moore, J Strother. Mjrtya fast majority vote algorithm. In *Automated Reasoning*, pp. 105–117. Springer, 1991.

Castro, Miguel, Liskov, Barbara, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pp. 173–186, 1999.

Charles, Zachary, Papailiopoulos, Dimitris, and Ellenberg, Jordan. Approximate gradient coding via sparse random graphs. *arXiv preprint arXiv:1711.06771*, 2017.

Chen, Jianmin, Pan, Xinghao, Monga, Rajat, Bengio, Samy, and Jozefowicz, Rafal. Revisiting distributed synchronous sgd. *arXiv preprint arXiv:1604.00981*, 2016.

Chen, Tianqi, Li, Mu, Li, Yutian, Lin, Min, Wang, Naiyan, Wang, Minjie, Xiao, Tianjun, Xu, Bing, Zhang, Chiyuan, and Zhang, Zheng. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

Chen, Yudong, Su, Lili, and Xu, Jiaming. Distributed statistical machine learning in adversarial settings: Byzantine gradient descent. *arXiv preprint arXiv:1705.05491*, 2017.

Chilimbi, Trishul M, Suzue, Yutaka, Apacible, Johnson, and Kalyanaraman, Karthik. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*, volume 14, pp. 571–582, 2014.

Cotter, Andrew, Shamir, Ohad, Srebro, Nati, and Sridharan, Karthik. Better mini-batch algorithms via accelerated gradient methods. In *NIPS*, pp. 1647–1655, 2011.

Dalcin, Lisandro D, Paz, Rodrigo R, Kler, Pablo A, and Cosimo, Alejandro. Parallel distributed computing using python. *Advances in Water Resources*, 34(9):1124–1139, 2011.

Dean, Jeffrey, Corrado, Greg, Monga, Rajat, Chen, Kai, Devin, Matthieu, Mao, Mark, Senior, Andrew, Tucker, Paul, Yang, Ke, Le, Quoc V, et al. Large scale distributed deep networks. In *NIPS*, pp. 1223–1231, 2012.

Dutta, Sanghamitra, Cadambe, Viveck, and Grover, Pulkit. Short-dot: Computing large linear transforms distributedly using coded short dot products. In *NIPS*, pp. 2100–2108, 2016.

Dutta, Sanghamitra, Cadambe, Viveck, and Grover, Pulkit. Coded convolution for parallel and distributed computing within a deadline. In *ISIT*, pp. 2403–2407. IEEE, 2017.

He, Kaiming, Zhang, Xiangyu, Ren, Shaoqing, and Sun, Jian. Deep residual learning for image recognition. In *CVPR*, pp. 770–778, 2016.

Jaggi, Martin, Smith, Virginia, Takác, Martin, Terhorst, Jonathan, Krishnan, Sanjay, Hofmann, Thomas, and Jordan, Michael I. Communication-efficient distributed dual coordinate ascent. In *NIPS*, pp. 3068–3076, 2014.

Johnson, Rie and Zhang, Tong. Accelerating stochastic gradient descent using predictive variance reduction. In *NIPS*, pp. 315–323, 2013.

Kim, Yoon. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

Konečnỳ, Jakub, McMahan, Brendan, and Ramage, Daniel. Federated optimization: Distributed optimization beyond the datacenter. *arXiv preprint arXiv:1511.03575*, 2015.

Konečnỳ, Jakub, McMahan, H Brendan, Yu, Felix X, Richtárik, Peter, Suresh, Ananda Theertha, and Bacon, Dave. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492*, 2016.

Kotla, Ramakrishna, Alvisi, Lorenzo, Dahlin, Mike, Clement, Allen, and Wong, Edmund. Zyzzyva: speculative byzantine fault tolerance. In *ACM SIGOPS Operating Systems Review*, volume 41, pp. 45–58. ACM, 2007.

Krizhevsky, Alex and Hinton, Geoffrey. Learning multiple layers of features from tiny images. 2009.

Krizhevsky, Alex, Sutskever, Ilya, and Hinton, Geoffrey E. Imagenet classification with deep convolutional neural networks. In *NIPS*, pp. 1097–1105, 2012.

Lamport, Leslie, Shostak, Robert, and Pease, Marshall. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

LeCun, Yann, Bottou, Léon, Bengio, Yoshua, and Haffner, Patrick. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Lee, Kangwook, Lam, Maximilian, Pedarsani, Ramtin, Papailiopoulos, Dimitris, and Ramchandran, Kannan. Speeding up distributed machine learning using codes. *IEEE Transactions on Information Theory*, 2017.

Li, Mu, Andersen, David G, Park, Jun Woo, Smola, Alexander J, Ahmed, Amr, Josifovski, Vanja, Long, James, Shekita, Eugene J, and Su, Bor-Yiing. Scaling distributed machine learning with the parameter server. In *OSDI*, volume 14, pp. 583–598, 2014.

Li, Songze, Maddah-Ali, Mohammad Ali, and Avestimehr, A Salman. Coded mapreduce. In *Communication, Control, and Computing (Allerton), 2015 53rd Annual Allerton Conference on*, pp. 964–971, 2015.

Liu, Ji, Wright, Steve, Re, Christopher, Bittorf, Victor, and Sridhar, Srikrishna. An asynchronous parallel stochastic coordinate descent algorithm. In *ICML*, pp. 469–477, 2014.

Mania, Horia, Pan, Xinghao, Papailiopoulos, Dimitris, Recht, Benjamin, Ramchandran, Kannan, and Jordan, Michael I. Perturbed iterate analysis for asynchronous stochastic optimization. *NIPS, OPT*, 2015.

Pang, Bo and Lee, Lillian. Seeing stars: Exploiting class relationships for sentiment categorization with respect to rating scales. In *ACL*, pp. 115–124, 2005.

Paszke, Adam, Gross, Sam, Chintala, Soumith, and Chanan, Gregory. Pytorch, 2017a.

Paszke, Adam, Gross, Sam, Chintala, Soumith, Chanan, Gregory, Yang, Edward, DeVito, Zachary, Lin, Zeming, Desmaison, Alban, Antiga, Luca, and Lerer, Adam. Automatic differentiation in pytorch. 2017b.

Pippenger, Nicholas. Reliable computation by formulas in the presence of noise. *IEEE Transactions on Information Theory*, 34(2):194–197, 1988.

Raviv, Netanel, Tamo, Itzhak, Tandon, Rashish, and Dimakis, Alexandros G. Gradient coding from cyclic mds codes and expander graphs. *arXiv preprint arXiv:1707.03858*, 2017.

Recht, Benjamin, Re, Christopher, Wright, Stephen, and Niu, Feng. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, pp. 693–701, 2011.

Reisizadeh, Amirhossein, Prakash, Saurav, Pedarsani, Ramtin, and Avestimehr, Salman. Coded computation over heterogeneous clusters. In *ISIT*, pp. 2408–2412. IEEE, 2017.

Shah, Nihar B, Lee, Kangwook, and Ramchandran, Kannan. When do redundant requests reduce latency? *IEEE Transactions on Communications*, 64(2):715–722, 2016.

Simonyan, Karen and Zisserman, Andrew. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Tandon, Rashish, Lei, Qi, Dimakis, Alexandros G, and Karampatziakis, Nikos. Gradient coding: Avoiding stragglers in distributed learning. In *ICML*, pp. 3368–3376, 2017.

Von Neumann, John. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata studies*, 34:43–98, 1956.

Yang, Yaoqing, Grover, Pulkit, and Kar, Soummya. Coded distributed computing for inverse problems. In *NIPS*, pp. 709–719, 2017.

Zaharia, Matei, Konwinski, Andy, Joseph, Anthony D, Katz, Randy H, and Stoica, Ion. Improving mapreduce performance in heterogeneous environments. In *OSDI*, volume 8, pp. 7, 2008.

Zhang, Sixin, Choromanska, Anna E, and LeCun, Yann. Deep learning with elastic averaging sgd. In *NIPS*, pp. 685–693, 2015.