Data Encoding for Byzantine-Resilient Distributed Optimization*

Deepesh Data[1], Linqi Song[2], and Suhas Diggavi[1]

[1]University of California, Los Angeles, USA
[1]{deepeshdata, suhasdiggavi}@ucla.edu

[2]City University of Hong Kong, Hong Kong
[2]linqi.song@cityu.edu.hk

Abstract

We study distributed optimization in the presence of Byzantine adversaries, where both data and computation are distributed among m worker machines, t of which can be corrupt and collaboratively deviate arbitrarily from their pre-specified programs, and a designated (master) node iteratively computes the model/parameter vector for *generalized linear models*. In this work, we primarily focus on two iterative algorithms: *Proximal Gradient Descent* (PGD) and *Coordinate Descent* (CD). Gradient descent (GD) is a special case of these algorithms. PGD is typically used in the data-parallel setting, where data is partitioned across different samples, whereas, CD is used in the model-parallelism setting, where the data is partitioned across the parameter space.

In this paper, we propose a method based on data encoding and error correction over real numbers to combat adversarial attacks. We can tolerate up to $t \leq \lfloor \frac{m-1}{2} \rfloor$ corrupt worker nodes, which is information-theoretically optimal. We give deterministic guarantees, and our method does not assume any probability distribution on the data. We develop a *sparse* encoding scheme which enables computationally efficient data encoding and decoding. We demonstrate a trade-off between corruption threshold and the resource requirement (storage and computational/communication complexity). As an example, for $t \leq \frac{m}{3}$, our scheme incurs only a *constant* overhead on these resources, over that required by the plain distributed PGD/CD algorithms which provide no adversarial protection.

Our encoding scheme extends *efficiently* to (i) the data streaming model, in which data samples come in an online fashion and are encoded as they arrive, and (ii) making *stochastic gradient descent* (SGD) Byzantine-resilient. In the end, we give experimental results to show the efficacy of our method.

1 Introduction

Map-reduce architecture [DG08] is implemented in many distributed learning tasks, where there is one designated machine (called the master) that computes the model iteratively, based on the inputs from the worker machines at each iteration, typically using descent techniques, like (proximal) gradient descent, coordinate descent, stochastic gradient descent, the Newton's method, etc. The worker nodes perform the required computations using local data, distributed to the nodes [ZWLS10]. Several other architectures, including having no hierarchy among the nodes have been explored [LZZ+17].

In several applications of distributed learning, including the Internet of Battlefield Things (IoBT) [A+18], federated optimization [Kon17], the recruited worker nodes might be partially trusted with their

*This paper was presented in parts at the IEEE Allerton 2018 (as an invited talk), and IEEE ISIT 2019. Part of this work was done when Linqi Song was at UCLA. The work of Deepesh Data and Suhas Diggavi was partially supported by the Army Research Laboratory under Cooperative Agreement W911NF-17-2-0196, by the UC-NL grant LFR-18-548554, and by the NSF award 1740047. The work of Linqi Song was partially supported by the NSF award 1527550, 1514531, and by the City University of Hong Kong grant (No. 7200594). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

computation. Therefore, an important question is whether we can reliably perform distributed computation, taking advantage of partially trusted worker nodes. These Byzantine adversaries can collaborate and arbitrarily deviate from their pre-specified programs. The problem of distributed computation with Byzantine adversaries has a long history [LSP82], and there has been recent interest in applying this computational model to large-scale distributed learning [BMGS17, CWCP18, CSX17].

In this paper, we study Byzantine-tolerant distributed optimization to learn a regularized *generalized linear model* (e.g., linear/ridge regression, logistic regression, Lasso, SVM dual, constrained minimization, etc.). We consider two frameworks for distributed optimization: (i) *data-parallelism* architecture, where data points are distributed across different worker nodes, and in each iteration, they all parallelly compute gradients on their local data and master aggregates them to update the parameter vector using gradient descent (GD) [BT89, Bot10, DCM+12]; and (ii) *model-parallelism* architecture, where data points are partitioned across the features, and several worker nodes work in parallel on updating different subsets of coordinates of the model/parameter vector through *coordinate descent* (CD) [BKBG11, Wri15, RT16]. Note that GD requires full gradients to update the parameter vector; and if full gradients are too costly to compute, we can reduce the per-iteration cost by using CD,[1] which also has been shown to be very effective for solving generalized linear models, and is particularly widely used for sparse logistic regression, SVM, and Lasso [BKBG11]. Given its simplicity and effectiveness, CD can be chosen over GD in such applications [Nes12]. Computing gradients in the presence of Byzantine adversaries has been recently studied [BMGS17, CWCP18, CSX17], but as far as we know, making CD robust to Byzantine adversaries has not received much attention. In addition to gradient descent, this motivates us to explore how to make coordinate descent also robust to Byzantine adversaries.

In this paper, we propose Byzantine-resilient distributed optimization algorithms both for PGD and CD based on data encoding and error correction (over real numbers). Our proposed algorithm differs from existing Byzantine-resilient distributed learning algorithms in one or more of the following aspects: (i) it does not make statistical assumptions on the data or Byzantine attack patterns; (ii) it can tolerate up to a constant fraction ($< 1/2$) of the worker nodes being Byzantine, which is information-theoretically optimal; and (iii) it enables a trade-off (in terms of storage and computation/communication overhead at the master and the worker nodes) with Byzantine adversary tolerance, without compromising the efficiency at the master node.

Our algorithms encode the data used by the m worker nodes, using ideas from real-error correction to enable tolerance to Byzantine workers. We develop an efficient "decoding" scheme at the master node to process the inputs from the workers, either to compute the true gradient in the case of gradient descent or to facilitate the computation at the worker nodes in the case of coordinate descent. We take a two-round approach in each iteration of both these algorithms. Our main results are summarized in Theorem 1 for GD and Theorem 2 for CD, and demonstrate a trade-off between the Byzantine resilience (in terms of the number of adversarial nodes) and the resource requirement (storage and computational/communication complexity). Our coding schemes can handle both Byzantine attacks and missing updates (*e.g.,* caused by delay and asynchrony of worker nodes). Though data encoding is a one-time process, it has to be efficient to harness the advantage of distributed computation. We design a sparse encoding process, based on real-error correction [CT05], which enables efficient encoding, and the worker nodes alternatively locally encode using the sparse structure. This allows encoding with storage redundancy of $2m/(m-2t)$ (which is a constant, *even* if t is a constant fraction of m), and one-time total computation cost for encoding is $O((1+2t)nd)$.

We extend our encoding scheme in a couple of important ways: first, to make the stochastic gradient descent algorithm Byzantine-resilient without compromising on the resource requirements; and second, to handle streaming data efficiently, where data arrives in batches (and we encode them as they arrive), rather than being available at the beginning of the computation; we also give few more applications of our method. For the streaming model, more specifically, our encoding requires the same amount of time, irrespective of whether we get all the data at once, or we get data points one by one or in batches. This setting encompasses a more realistic scenario, in which we design our coding scheme with the initial set of data points and distribute the encoded data among the workers. Later on, when we get some more

[1] Alternatively, we can also use SGD to reduce the per-iteration cost, and we give a method for making SGD Byzantine-resilient in Section 6.1.

samples, we can easily incorporate them into our existing encoded setup. See Section 6 for details on these extensions.

Paper organization. Our problem formulation and the main results, along-with the high level ideas of our Byzantine-resilient algorithms for both PGD and CD are given in Section 2. We give related work in Section 3. We present our full coding schemes for gradient descent and coordinate descent along with a complete analysis of their resource requirements in Section 4 and Section 5, respectively. In Section 6, we show how our method can be extended to SGD and to the data streaming model. We also discuss applicability of our method to a few more important applications in that section. In Section 7, we show numerical results of our method: we show the efficiency of our method by running it on two datasets (moderate and large) and plotting the running time with varying number of corrupt worker nodes (up to $<1/2$ fraction). We conclude with a short discussion in Section 8.

Notation. We denote vectors by bold small letters (e.g., $\mathbf{x}, \mathbf{y}, \mathbf{z}$, etc.) and matrices by bold capital letters (e.g., $\mathbf{A}, \mathbf{F}, \mathbf{S}, \mathbf{X}$, etc.). We denote the amount of storage required by a matrix $\mathbf{X}$ by $|\mathbf{X}|$. For any positive integer $n \in \mathbb{N}$, we denote the set $\{1, 2, \ldots, n\}$ by $[n]$. For $n_1, n_2 \in \mathbb{N}$, where $n_1 \leq n_2$, we write $[n_1 : n_2]$ to denote the set $\{n_1, n_1 + 1, \ldots, n_2\}$. For any vector $\mathbf{u} \in \mathbb{R}^n$ and any set $\mathcal{S} \subset [n]$, we write $\mathbf{u}_{\mathcal{S}}$ to denote the $|\mathcal{S}|$-length vector, which is the restriction of $\mathbf{u}$ to the coordinates in the set $\mathcal{S}$. The support of a vector $\mathbf{u} \in \mathbb{R}^n$ is defined by $\mathsf{supp}(\mathbf{u}) := \{i \in [n] : u_i \neq 0\}$. We say that a vector $\mathbf{u} \in \mathbb{R}^n$ is t-sparse if $|\mathsf{supp}(\mathbf{u})| \leq t$. While stating our results, we assume that performing the basic arithmetic operations (addition, subtraction, multiplication, and division) on real numbers take unit time.

2 Problem Setting and Our Results

Given a dataset consisting of n labelled data points $(\mathbf{x}_i, y_i) \in \mathbb{R}^d \times \mathbb{R}$, $i \in [n]$, we want to learn a model/parameter vector $\mathbf{w} \in \mathbb{R}^d$, which is a minimizer of the following convex optimization problem:

$$\min_{\mathbf{w} \in \mathbb{R}^d} \left(\frac{1}{n} \sum_{i=1}^{n} f_i(\mathbf{w}) \right) + h(\mathbf{w}), \tag{1}$$

where $f_i(\mathbf{w})$, $i = 1, 2, \ldots, n$, denotes the empirical risk associated with the i'th data point with respect to $\mathbf{w}$ and $h(\mathbf{w})$ denotes a regularizer. We call $f(\mathbf{w}) := \frac{1}{n} \sum_{i=1}^{n} f_i(\mathbf{w})$ the average empirical risk associated with the n data points with respect to $\mathbf{w}$. Our main focus in this paper is on *generalized linear models* (GLM), where $f_i(\mathbf{w}) = \ell(\langle \mathbf{x}_i, \mathbf{w} \rangle; y_i)$ for some differentiable loss function ℓ. Here each $f_i : \mathbb{R}^d \to \mathbb{R}$ is differentiable, $h : \mathbb{R}^d \to \mathbb{R}$ is convex but not necessarily differentiable, and $\langle \mathbf{x}_i, \mathbf{w} \rangle$ is the dot product of $\mathbf{x}_i$ and $\mathbf{w}$. We do not necessarily need each f_i to be convex, but we require $f(\mathbf{w})$ to be a convex function. Note that $f(\mathbf{w}) + h(\mathbf{w})$ is a convex function. In the following we study different algorithms for solving (1) to learn a GLM.

2.1 Proximal Gradient Descent

We can solve (1) using *Proximal Gradient Descent* (PGD). This is an iterative algorithm, in which we choose an initial $\mathbf{w}_0 \in \mathbb{R}^d$ randomly, and then update the parameter vector according to the following update rule:

$$\mathbf{w}_{t+1} = prox_{h,\alpha_t}(\mathbf{w}_t - \alpha_t \nabla f(\mathbf{w}_t)), \quad t = 1, 2, 3, \ldots \tag{2}$$

where α_t is the step size or the learning rate at the t'th iteration, determining the convergence behaviour. There are standard choices for it; see, for example, [BV04, Chapter 9]. For any h and α, the proximal operator $prox_{h,\alpha} : \mathbb{R}^d \to \mathbb{R}$ is defined as

$$prox_{h,\alpha}(\mathbf{w}) = \arg\min_{\mathbf{z} \in \mathbb{R}^d} \frac{1}{2\alpha} \|\mathbf{z} - \mathbf{w}\|_2^2 + h(\mathbf{z}). \tag{3}$$

Observe that if $h = 0$, then $prox_{h,\alpha}(\mathbf{w}) = \mathbf{w}$ for every $\mathbf{w} \in \mathbb{R}^d$, and PGD reduces to the classical gradient descent (GD). This encompasses several important optimization problems related to learning, for which *prox* operator has a closed form expression; some of these problems are given below.

- **Lasso:** Here $f_i(\mathbf{w}) = \frac{1}{2}(\langle \mathbf{x}_i, \mathbf{w} \rangle - y_i)^2$ and $h(\mathbf{w}) = \lambda \|\mathbf{w}\|_1$. It turns out that $prox_{h,\alpha}(\mathbf{z})$ for Lasso is equal to the soft-thresholding operator $S_{\lambda\alpha}(\mathbf{z})$ [Tib15], which, for $j \in [d]$, is defined as

$$(S_{\lambda\alpha}(\mathbf{z}))_j = \begin{cases} z_j + \lambda\alpha & \text{if } z_j < -\lambda\alpha, \\ 0 & \text{if } -\lambda\alpha \leq z_j \leq \lambda\alpha, \\ z_j - \lambda\alpha & \text{if } z_j > \lambda\alpha. \end{cases}$$

- **SVM dual:** Jaggi [Jag13] showed an equivalence between the dual formulation of Support Vector Machine (SVM) and Lasso. Hence, SVM dual is also a special case of (1).

- **Constrained optimization:** We want to solve a constrained minimization problem $\min_{\mathbf{w} \in \mathcal{C}} f(\mathbf{w})$, where $\mathcal{C} \subseteq \mathbb{R}^d$ is a closed, convex set. Define an indicator function $I_{\mathcal{C}}$ for $\mathcal{C}$ as follows: $I_{\mathcal{C}}(\mathbf{w}) := 0$, if $\mathbf{w} \in \mathcal{C}$; and $I_{\mathcal{C}}(\mathbf{w}) := \infty$, otherwise. Now, observe the following equivalence

$$\min_{\mathbf{w} \in \mathcal{C}} f(\mathbf{w}) \iff \min_{\mathbf{w} \in \mathbb{R}^d} f(\mathbf{w}) + I_{\mathcal{C}}(\mathbf{w}).$$

  If we solve the RHS using PGD, then it can be easily verified that the corresponding proximal operator is equal to the projection operator onto the set $\mathcal{C}$ [Tib15]. So, the proximal gradient update step is to compute the usual gradient and then project it back onto the set $\mathcal{C}$.

- **Logistic regression:** Here f_i is the logistic function, defined as

$$f_i(\mathbf{w}) = -y_i \log\left(\frac{1}{1+e^{-u_i}}\right) - (1-y_i)\log\left(\frac{e^{-u_i}}{1+e^{-u_i}}\right),$$

  where $u_i = \langle \mathbf{x}_i, \mathbf{w} \rangle$, and $h = 0$. As noted earlier, since $h = 0$, PGD reduces to GD for logistic regression.

- **Ridge regression:** Here $f_i(\mathbf{w}) = \frac{1}{2}(\langle \mathbf{x}_i, \mathbf{w} \rangle - y_i)^2$ and $h(\mathbf{w}) = \frac{\lambda}{2}\|\mathbf{w}\|_2^2$. Since f_i's and h are differentiable, we can alternatively solve this simply using GD.

Let $\mathbf{X} \in \mathbb{R}^{n \times d}$ denote the data matrix, whose i'th row is equal to the i'th data point $\mathbf{x}_i$. For simplicity, assume that m divides n, and let $\mathbf{X}_i$ denote the $\frac{n}{m} \times d$ matrix, whose j'th row is equal to $\mathbf{x}_{(i-1)\frac{n}{m}+j}$. In a distributed setup, all the data is distributed among m worker machines (worker i has $\mathbf{X}_i$) and master updates the parameter vector using the update rule (2). At the t'th iteration, master sends $\mathbf{w}_t$ to all the workers; worker i computes the gradient (denoted by $\nabla_i f(\mathbf{w}_t)$) on its local data and sends it to the master; master aggregates all the received m local gradients to obtain the global gradient

$$\nabla f(\mathbf{w}_t) = \frac{1}{m}\sum_{i=1}^{m} \nabla_i f(\mathbf{w}_t). \tag{4}$$

Now, master updates the parameter vector according to (2) and obtains $\mathbf{w}_{t+1}$. Repeat the process until convergence.

If full gradients are too costly to compute. Updating the parameter vector in each iteration of PGD (2) requires to compute full gradients. This may be prohibitive in some large-scale applications,[2] where one has to *make progress* (i.e., updating the parameter vector in the right direction) quickly, because each update gets delayed by computing the full gradient. In such scenarios, there are two alternatives to reduce this per-iteration cost: (i) *Coordinate Descent* (CD), in which we pick a few coordinates (at random), compute the partial gradient along those, and descent along those coordinates only, and (ii) *Stochastic Gradient Descent* (SGD), in which we sample a data point at random, compute the gradient on that point, and descent along that direction. These are discussed in Section 2.2 and Section 6.1, respectively.

[2] For example, in a situation, where each machine in a distributed framework has a lot of data, computing full gradients even at the local machine may be too expensive.

2.2 Coordinate Descent

For clear exposition of the ideas, we focus on the non-regularized empirical risk minimization from (1) for learning a *generalized linear model* (GLM). This can be generalized to objectives with (non-)differentiable regularizers [BKBG11, ST11]. Let $\mathbf{X} \in \mathbb{R}^{n \times d}$ denote the data matrix and $\mathbf{y} \in \mathbb{R}^n$ the corresponding label vector. To make it distinct from the last section, we denote the objective function by ϕ and write it as $\phi(\mathbf{X}\mathbf{w}; \mathbf{y})$ to emphasize that we want to learn a GLM, where the objective function depends on the data points only through their inner products with the parameter vector. Formally, we want to optimize[3]

$$\min_{\mathbf{w} \in \mathbb{R}^d} \phi(\mathbf{X}\mathbf{w}; \mathbf{y}) := \sum_{i=1}^{n} \ell(\langle \mathbf{x}_i, \mathbf{w} \rangle; y_i). \tag{5}$$

For $\mathcal{U} \subseteq [d]$, we write $\nabla_{\mathcal{U}} \phi(\mathbf{X}\mathbf{w}; \mathbf{y})$ to denote the gradient of $\phi(\mathbf{X}\mathbf{w}; \mathbf{y})$ with respect to $\mathbf{w}_{\mathcal{U}}$, where $\mathbf{w}_{\mathcal{U}}$ denotes the $|\mathcal{U}|$-length vector obtained by restricting $\mathbf{w}$ to the coordinates in $\mathcal{U}$. To make the notation less cluttered, let $\phi'(\mathbf{X}\mathbf{w}; \mathbf{y})$ denote the n-length vector, whose i'th entry is equal to $\ell'(\langle \mathbf{x}_i, \mathbf{w} \rangle; y_i) := \frac{\partial}{\partial u} \ell(u; y_i)|_{u = \langle \mathbf{x}_i, \mathbf{w} \rangle}$. Note that $\nabla \phi(\mathbf{X}\mathbf{w}; \mathbf{y}) = \mathbf{X}^T \phi'(\mathbf{X}\mathbf{w}; \mathbf{y})$ and that $\nabla_{\mathcal{U}} \phi(\mathbf{X}\mathbf{w}; \mathbf{y}) = \mathbf{X}_{\mathcal{U}}^T \phi'(\mathbf{X}\mathbf{w}; \mathbf{y})$, where $\mathbf{X}_{\mathcal{U}}$ denotes the $n \times |\mathcal{U}|$ matrix obtained by restricting the column indices of $\mathbf{X}$ to the elements in $\mathcal{U}$.

Coordinate descent (CD) is an iterative algorithm, where, in each iteration, we choose a set of coordinates and update only those coordinates (while keeping the other coordinates fixed). In distributed CD, we take advantage of the parallel architecture to improve the running time of (centralized) CD. In the distributed setting, we divide the data matrix vertically into m parts and store the i'th part at the i'th worker node. Concretely, assume, for simplicity, that m divides d. Let $\mathbf{X} = [\mathbf{X}_1 \ \mathbf{X}_2 \ \dots \ \mathbf{X}_m]$ and $\mathbf{w} = [\mathbf{w}_1^T \ \mathbf{w}_2^T \ \dots \ \mathbf{w}_m^T]^T$, where each $\mathbf{X}_i$ is an $n \times (d/m)$ matrix and each $\mathbf{w}_i$ is length d/m vector. Each worker i stores $\mathbf{X}_i$ and is responsible for updating (a few coordinates of) $\mathbf{w}_i$ – hence the terminology, model-parallelism. We can store the label vector $\mathbf{y}$ either with the master or with the workers. In coordinate descent, since we update only a few coordinates in each round, there are a few options on how to update these coordinates in a distributed manner:

Subset of workers: Master picks a subset $\mathcal{S} \subset [m]$ of workers and asks them to update their $\mathbf{w}_i$'s [RT16]. This may not be good in the adversarial setting, because if only a small subset of workers are updating their parameters, the adversary can corrupt those workers and disrupt the computation.

Subset of coordinates for all workers: All the worker nodes update only a subset of the coordinates of their local parameter vector $\mathbf{w}_i$'s. Master can (deterministically or randomly) pick a subset $\mathcal{U}$ (which may or may not be different for all workers) of $f \leq d/m$ coordinates and asks each worker to updates only those coordinates. If master picks $\mathcal{U}$ deterministically, it can cycle through and update all coordinates of the parameter vector in $\lceil d/mf \rceil$ iterations.

In Algorithm 1 on page 6, we give the distributed CD algorithm with the second approach, where all the worker nodes update the coordinates of their local parameter vectors for a single subset $\mathcal{U}$. We will adopt this approach in our method to make the distributed CD Byzantine-resilient. Let $r = d/m$. For any $i \in [m]$, let $\mathbf{w}_i = [w_{i1} \ w_{i2} \dots w_{ir}]^T$ and $\mathbf{X}_i = [\mathbf{X}_{i1} \ \mathbf{X}_{i2} \dots \mathbf{X}_{ir}]$, where $\mathbf{X}_{ij}$ is the j'th column of $\mathbf{X}_i$. For any $i \in [m]$ and $\mathcal{U} \subseteq [r]$, let $\mathbf{w}_{i\mathcal{U}}$ denote the $|\mathcal{U}|$-length vector that is obtained from $\mathbf{w}_i$ by restricting its entries to the coordinates in $\mathcal{U}$; similarly, let $\mathbf{X}_{i\mathcal{U}}$ denote the $n \times |\mathcal{U}|$ matrix obtained by restricting the column indices of $\mathbf{X}_i$ to the elements in $\mathcal{U}$.

In Algorithm 1, for each worker i to update $\mathbf{w}_i$ according to (6), where the partial gradient of ϕ with respect to $\mathbf{w}_{i\mathcal{U}}$ is equal to $\nabla_{i\mathcal{U}} \phi(\mathbf{X}\mathbf{w}; \mathbf{y}) = \mathbf{X}_{i\mathcal{U}}^T \phi'(\sum_{j=1}^m \mathbf{X}_j \mathbf{w}_j; \mathbf{y})$ and worker i has only $(\mathbf{X}_i, \mathbf{w}_i)$, every other worker j sends $\mathbf{X}_j \mathbf{w}_j$ to the master, who computes $\phi'(\sum_{j=1}^m \mathbf{X}_j \mathbf{w}_j; \mathbf{y})$ (see Footnote 2) and sends it back to all the workers. Observe that, even if one worker is corrupt, it can send an adversarially chosen vector to make the computation at the master deviate arbitrarily from the desired computation, which may adversely affect the update at all the worker nodes subsequently.[4] Similarly, corrupt workers can send adversarially chosen information to affect the stopping criterion.

[3] Here we are not optimizing the *average* of loss functions – since n is a fixed number, this does not affect the solution space.

[4] Specifically, suppose the i'th worker is corrupt and the adversary wants master to compute $\phi'(\mathbf{X}\mathbf{w} + \mathbf{e}; \mathbf{y})$ for any arbitrary vector $\mathbf{e} \in \mathbb{R}^n$ of its choice, then the i'th worker can send $\mathbf{X}_i \mathbf{w}_i + \mathbf{e}$ to the master.

Algorithm 1: Distributed Coordinate Descent

Each worker i starts with an arbitrary/random $\mathbf{w}_i$.

Repeat (until the stopping criteria is not satisfied)

1. Each worker $i \in [m]$ computes $\mathbf{X}_i\mathbf{w}_i$ and sends it to the master node. Note that $\mathbf{X}\mathbf{w} = \sum_{i=1}^{m} \mathbf{X}_i\mathbf{w}_i$.[5]

2. Master computes $\phi'(\mathbf{X}\mathbf{w};\mathbf{y})$[6] and broadcasts it.

3. For some $\mathcal{U} \subseteq [r]$ (where $\mathcal{U}$ can be picked either randomly or in a round-robin fashion), each worker $i \in [m]$ updates its local parameter vector as

$$\mathbf{w}_{i\mathcal{U}} \leftarrow \mathbf{w}_{i\mathcal{U}} - \alpha \nabla_{i\mathcal{U}} \phi(\mathbf{X}\mathbf{w};\mathbf{y}) \tag{6}$$

   while keeping the other coordinates of $\mathbf{w}_i$ unchanged, and sends the updated $\mathbf{w}_i$ to the master, which can check for the stopping criteria.

2.3 Adversary Model

We want to perform the distributed computation described in Section 2.1, 2.2 under adversarial attacks, where the corrupt nodes may provide erroneous vectors to the master node. Our adversarial model is described next.

Adversary model. In our adversarial model, the adversary can corrupt t of the worker nodes[7], and the compromised nodes may arbitrarily deviate from their pre-specified programs. If a worker i is corrupt, then instead of sending the true vector, it may send an arbitrary vector to disrupt the computation. We refer to the corrupt nodes as erroneous or under the Byzantine attack. We can handle asynchronous updates, by dropping the straggling nodes beyond a specified delay, and still compute the correct gradient due to encoding. Therefore we treat updates from these nodes as being "erased". We refer to these as erasures/stragglers. For every worker i that sends a message to the master, we can assume, without loss of generality, that the master receives $\mathbf{u}_i + \mathbf{e}_i$, where $\mathbf{u}_i$ is the true vector, and $\mathbf{e}_i = \mathbf{0}$ if the i'th node is honest, otherwise can be arbitrary. We denote the set of worker nodes under the Byzantine attack by $\mathcal{A}_1$ and straggling worker nodes by $\mathcal{A}_2$, where $\mathcal{A}_1, \mathcal{A}_2 \subset [m]$, $|\mathcal{A}_1| \leq t$, $|\mathcal{A}_2| \leq s$, for some s, t that we will decide later. The adversarial nodes can collude, and can even know the data of other workers. The master node does not know which t worker nodes are corrupted, but knows t, the maximum possible number of adversarial nodes. We propose a method that mitigates the effects of both of these anomalies.

Remark 1. *A well-studied problem is that of asynchronous distributed optimization, where the workers can have different delays in updates [DB13]. One mechanism to deal with this is to wait for a subset of responses, before proceeding to the next iteration, treating the others as missing (or erasures) [KSDY17]. Byzantine attacks are quite distinct from such erasures, as the adversary can report wrong local gradients, requiring the master node to create mechanisms to overcome such attacks. If the master node simply aggregates the collected updates as in (4), the computed gradient could be arbitrarily far away from the true one, even with a single adversary [MGR18].*

[5] After the 1st iteration, worker i need not multiply $\mathbf{X}_i$ with $\mathbf{w}_i$ to obtain $\mathbf{X}_i\mathbf{w}_i$ in every iteration; as only a few coordinates of $\mathbf{w}_i$ are updated, it only needs to multiply those columns of $\mathbf{X}_i$ that corresponds to the updated coordinates of $\mathbf{w}_i$.

[6] Note that even after computing $\mathbf{X}\mathbf{w}$, master needs access to the labels $y_i, i = 1, 2, \ldots, n$ to compute $\phi'(\mathbf{X}\mathbf{w};\mathbf{y})$. Since $\mathbf{y} \in \mathbb{R}^n$ is just a vector, we can either store that at master, or, alternatively, we can encode $\mathbf{y}$ distributedly at the workers and master can recover that using the method developed in Section 4 for Byzantine-resilient distributed matrix-vector multiplication, where the matrix is an identity matrix and vector is equal to $\mathbf{y}$.

[7] Our results also apply to a slightly *different* adversarial model, where the adversary can adaptively *choose* which of the t worker nodes to attack at each iteration. However, in this model, the adversary cannot modify the local stored data of the attacked node, as otherwise, over time, it can corrupt all the data, making any defense impossible.

2.4 Our Approach to Gradient Computation

Recall that $f_i(\mathbf{w}) = \ell(\langle \mathbf{x}_i, \mathbf{w}\rangle; y_i)$ for some differentiable loss function ℓ, and the gradient of f_i at $\mathbf{w}$ is equal to $\nabla f_i(\mathbf{w}) = (\mathbf{x}_i)^T \ell'(\langle \mathbf{x}_i, \mathbf{w}\rangle; y_i)$, where $\ell'(\langle \mathbf{x}_i, \mathbf{w}\rangle; y_i) := \frac{\partial}{\partial u}\ell(u; y_i)|_{u=\langle \mathbf{x}_i, \mathbf{w}\rangle}$. Note that $\nabla f_i(\mathbf{w}) \in \mathbb{R}^d$ is a column vector. Let $f'(\mathbf{w})$ denote the n-length vector whose i'th entry is equal to $\ell'(\langle \mathbf{x}_i, \mathbf{w}\rangle; y_i)$. With this notation, since $f(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^n f_i(\mathbf{w})$, we have $\nabla f(\mathbf{w}) = \frac{1}{n}\mathbf{X}^T f'(\mathbf{w})$. Since n is a constant, it is enough to compute $\mathbf{X}^T f'(\mathbf{w})$. So, for simplicity, in the rest of the paper we write

$$\nabla f(\mathbf{w}) = \mathbf{X}^T f'(\mathbf{w}), \quad \forall \mathbf{w} \in \mathbb{R}^d. \tag{7}$$

A natural approach to computing the gradient $\nabla f(\mathbf{w})$ is to compute it in two rounds: (i) compute $f'(\mathbf{w})$ in the 1st round by first multiplying $\mathbf{X}$ with $\mathbf{w}$ and then master locally computes $f'(\mathbf{w})$ from $\mathbf{X}\mathbf{w}$ (master can do this locally, because $\mathbf{X}\mathbf{w}$ is an n-dimensional vector whose i'th entry is equal to $\langle \mathbf{x}_i, \mathbf{w}\rangle$ and $(f'(\mathbf{w}))_i = \ell'(\langle \mathbf{x}_i, \mathbf{w}\rangle; y_i))$;[8] and (ii) compute $\nabla f(\mathbf{w}) = \mathbf{X}^T f'(\mathbf{w})$ in the 2nd round by multiplying $\mathbf{X}^T$ with $f'(\mathbf{w})$. So, the task of each gradient computation reduces to two matrix-vector (MV) multiplications, where the matrices are fixed and vectors may be different each time. To combat against the adversarial worker nodes, we do both of these MV multiplications using data encoding and real-error correction; see Figure 1 for a pictorial description of our approach. More specifically, for the 1st round, we encode $\mathbf{X}$ using a sparse encoding matrix $\mathbf{S}^{(1)} = [(\mathbf{S}_1^{(1)})^T, \ldots, (\mathbf{S}_m^{(1)})^T]^T$ and store $\mathbf{S}_i^{(1)}\mathbf{X}$ at the i'th worker node; and for the 2nd round, we encode $\mathbf{X}^T$ using another sparse encoding matrix $\mathbf{S}^{(2)} = [(\mathbf{S}_1^{(2)})^T, \ldots, (\mathbf{S}_m^{(2)})^T]^T$, and store $\mathbf{S}_i^{(2)}\mathbf{X}^T$ at the i'th worker node. Now, in the 1st round of the gradient computation at $\mathbf{w}$, the master node broadcasts $\mathbf{w}$ and the i'th worker node replies with $\mathbf{S}_i^{(1)}\mathbf{X}\mathbf{w}$ (a corrupt worker may report an arbitrary vector); upon receiving all the vectors, the master node applies error-correction procedure to recover $\mathbf{X}\mathbf{w}$ and then locally computes $f'(\mathbf{w})$ as described above; in the 2nd round, the master node broadcasts $f'(\mathbf{w})$ and similarly can recover $\mathbf{X}^T f'(\mathbf{w})$ (which is equal to the gradient) at the end of the 2nd round. So, it suffices to devise a method for multiplying a vector $\mathbf{v}$ to a fixed matrix $\mathbf{A}$ in a distributed and adversarial setting. Since this is a linear operation, we can apply error correcting codes over real numbers to perform this task. We describe it briefly below.

A trivial approach. Take a generator matrix $\mathbf{G}$ of any real-error correcting linear code. Encode $\mathbf{A}$ as $\mathbf{A}^T\mathbf{G} =: \mathbf{B}$. Divide the columns of $\mathbf{B}$ into m groups as $\mathbf{B} = [\mathbf{B}_1\ \mathbf{B}_2 \ldots \mathbf{B}_m]$, where worker i stores $\mathbf{B}_i$. Master broadcasts $\mathbf{v}$ and each worker i responds with $\mathbf{v}_T\mathbf{B}_i + \mathbf{e}_i^T$, where $\mathbf{e}_i = \mathbf{0}$ if the i'th worker is honest, otherwise can be arbitrary. Note that at most t of the $\mathbf{e}_i$'s can be non-zero. Responses from the workers can be combined as $\mathbf{v}^T\mathbf{B} + \mathbf{e}^T$. Since every row of $\mathbf{B}$ is a codeword, $\mathbf{v}^T\mathbf{B} = \mathbf{v}^T\mathbf{A}^T\mathbf{G}$ is also a codeword. Therefore, one can take any off-the-shelf decoding algorithm for the code whose generator matrix is $\mathbf{G}$ and obtain $\mathbf{v}^T\mathbf{A}^T$. For example, we can use Reed-Solomon codes (over real numbers) for this purpose, which only incurs a constant storage overhead and tolerates optimal number of corruptions (up to $<1/2$). Note that we need fast decoding, as it is performed in every iteration of the gradient computation by the master. As far as we know, any off-the-shelf decoding algorithm (over real numbers) requires at least a quadratic computational complexity, which leads to $\Omega(n^2 + d^2)$ decoding complexity per gradient computation, which could be impractical.

The trivial scheme does not exploit the block error pattern which we crucially exploit in our coding scheme to give a $\sim O((n+d)m)$ time decoding per gradient computation, which could be a significant improvement over the trivial scheme, since m typically is much smaller than n and d for large-scale problems. In fact, our coding scheme enables a trade-off (in terms of storage and computation/communication overhead at the master and the worker nodes) with Byzantine adversary tolerance, without compromising the efficiency at the master node. We also want encoding to be efficient (otherwise it defeats the purpose of data encoding) and our sparse encoding matrix achieves that. Our main result for the Byzantine-resilient distributed gradient computation is as follows, which is proved in Section 4:

Theorem 1 (Gradient Computation)**.** *Let $\mathbf{X} \in \mathbb{R}^{n\times d}$ denote the data matrix. Let m denote the total number of worker nodes. We can compute the gradient exactly in a distributed manner in the presence of t corrupt worker nodes and s stragglers, with the following guarantees, where $\epsilon > 0$ is a free parameter.*

[8] Note that even after computing $\mathbf{X}\mathbf{w}$, master needs access to the labels $y_i, i = 1, 2, \ldots, n$ to compute $f'(\mathbf{w})$. See Footnote 2 for a discussion on how master can get access to the labels.

- $(s+t) \leq \left\lfloor \frac{\epsilon}{1+\epsilon} \cdot \frac{m}{2} \right\rfloor$.
- *Total storage requirement is roughly* $2(1+\epsilon)|\mathbf{X}|$.
- *Computational complexity for each gradient computation:*
  - *at each worker node is* $O((1+\epsilon)\frac{nd}{m})$.
  - *at the master node is* $O((1+\epsilon)(n+d)m)$.
- *Communication complexity for each gradient computation:*
  - *each worker sends* $\left((1+\epsilon)\frac{n+d}{m}\right)$ *real numbers.*
  - *master broadcasts* $(n+d)$ *real numbers.*
- *Total encoding time is* $O\left(nd\left(\frac{\epsilon}{1+\epsilon}m+1\right)\right)$.

Remark 2. *We compare the resource requirements of our method with the plain distributed PGD (which provides* no adversarial protection*), where all the data points are evenly distributed among the* m *workers. In each iteration, master sends the parameter vector* $\mathbf{w}$ *to all the workers; upon receiving* $\mathbf{w}$*, all workers compute the gradients on their local data in* $O(nd/m)$ *time (per worker) and send them to the master; master aggregates them in* $O(md)$ *time to obtain the global gradient and then updates the parameter vector using* (2).

In our scheme (i) *the total storage requirement is* $O(1+\epsilon)$ *factor more;*[9] *(see also Remark 4)* (ii) *the amount of computation at each worker node is* $O(1+\epsilon)$ *factor more;* (iii) *the amount of computation at the master node is* $O((1+\epsilon)(1+\frac{n}{d}))$ *factor more, which is comparable in cases where* n *is not much bigger than* d*;* (iv) *master broadcasts* $(1+\frac{n}{d})$ *factor more data, which is comparable if* n *is not much bigger than* d*; and* (v) *each worker sends* $O\left((1+\epsilon)\frac{1+n/d}{m}\right)$ *factor more data, which is, in fact,* $O((1+\epsilon)\frac{1}{m})$ *if* n *is not much bigger than* d*, and smaller than* $O(1+\epsilon)$ *– a constant factor – as long as* $n < O(dm)$.

Remark 3. *The statement of Theorem 1 allows for any* s *and* t*, as long as* $(s+t) \leq \left\lfloor \frac{\epsilon}{1+\epsilon} \cdot \frac{m}{2} \right\rfloor$*. As we are handling both erasures and errors in the same way*[10] *the corruption threshold does not have to handle* s *and* t *separately. To simplify the discussion, for the rest of the paper, we consider only Byzantine corruption, and denote the corrupted set by* $\mathcal{I} \subset [m]$ *with* $|\mathcal{I}| \leq t$*, with the understanding that this can also work with stragglers.*

Remark 4. *Let* m *be an even number. Note that we can get the corruption threshold* t *to be any number less than* $m/2$*, but at the expense of increased storage and computation. For any* $\delta > 0$*, if we want to get* δ *close to* $m/2$*, i.e.,* $t = m/2 - \delta$*, then we must have* $(1+\epsilon) \geq m/2\delta$*. In particular, at* $\epsilon = 2$*, we can tolerate up to* $m/3$ *corrupt nodes, with constant overhead in the total storage as well as on the computational complexity.*

Note that when δ *is a constant, i.e.,* t *is close to* $\frac{m-1}{2}$*, then* ϵ *grows linearly with* m*; for example, if* $t = \frac{m-1}{2}$*, then* $\epsilon = m-1$*. In this case, our storage redundancy factor is* $O(m)$*. In contrast, the trivial scheme (see "trivial approach" on page 7) does better in this regime and has only a constant storage overhead, but at the expense of an increased decoding complexity at the master, which is at least quadratic in the problem dimensions* d *and* n*, whereas, our decoding complexity at the master always scales linearly with* d *and* n*. If we always want a constant storage redundancy for all values of the corruption threshold* t*, we can use our coding scheme if* $t \leq c \cdot \frac{m-1}{2}$*, where* $c < 1$ *is a constant, and use the trivial scheme if* t *is close to* $\frac{m-1}{2}$.

[9] For example, by taking $\epsilon = 2$, our method can tolerate $m/3$ corrupt worker nodes. So, we can tolerate linear corruption with a *constant* overhead in the resource requirement, compared to the plain distributed gradient computation which does not provide any adversarial protection.

[10] When there are *only stragglers*, one can design an encoding scheme where *both* the master and the worker nodes operate oblivious to encoding, while solving a slightly altered optimization problem [KSDY17], in which gradients are computed *approximately*, leading to more efficient straggler-tolerant GD.

Our encoding is also efficient and requires $O\left(nd\left(\frac{\epsilon}{1+\epsilon}m+1\right)\right)$ *time. Note that* $O(nd)$ *is equal to the time required for distributing the data matrix* $\mathbf{X}$ *among* m *workers (for running the distributed gradient descent algorithms* without *the adversary); and the encoding time in our scheme (which results in an encoded matrix that provides Byzantine-resiliency) is a factor of* $(2t+1)$ *more.*

Remark 5. *Our scheme is not only efficient (both in terms of computational complexity and storage requirement), but it can also tolerate up to* $\lfloor\frac{m-1}{2}\rfloor$ *corrupt worker nodes (by taking* $\epsilon = m-1$ *in Theorem 1). It is not hard to prove that this bound is information-theoretically optimal, i.e., no algorithm can tolerate* $\lceil\frac{m}{2}\rceil$ *corrupt worker nodes, and at the same time correctly computes the gradient.*

2.5 Our Approach to Coordinate Descent

We use data encoding and add redundancy to enlarge the parameter space. Specifically, we encode the data matrix $\mathbf{X}$ using an encoding matrix $\mathbf{R} = [\mathbf{R}_1\ \mathbf{R}_2\ \ldots\ \mathbf{R}_m]$, where each $\mathbf{R}_i$ is a $d \times p$ matrix (with $pm \geq d$), and store $\mathbf{X}\mathbf{R}_i$ at the i'th worker. Define $\widetilde{\mathbf{X}}^R := \mathbf{X}\mathbf{R}$. Now, instead of solving (5), we solve the encoded problem $\arg\min_{\mathbf{v}\in\mathbb{R}^{pm}} \phi(\widetilde{\mathbf{X}}^R\mathbf{v}; \mathbf{y})$ using Algorithm 1 (together with decoding at the master); see Figure 2 for a pictorial description of our algorithm. We design the encoding matrix $\mathbf{R}$ such that at every iteration of our algorithm, updating any (small) subset of coordinates of $\mathbf{v}_i$'s (let $\mathbf{v} = [\mathbf{v}_1^T\ \mathbf{v}_2^T\ \ldots\ \mathbf{v}_m^T]$) automatically updates some (small) subset of coordinates of $\mathbf{w}$; and, furthermore, by updating those coordinates of $\mathbf{v}_i$'s, we can efficiently recover the correspondingly updated coordinates of $\mathbf{w}$, despite the errors injected by the adversary. In fact, at any iteration t, the encoded parameter vector $\mathbf{v}_t$ and the original parameter vector $\mathbf{w}_t$ satisfies $\mathbf{v}_t = \mathbf{R}^+\mathbf{w}_t$, where $\mathbf{R}^+ := \mathbf{R}^T(\mathbf{R}\mathbf{R}^T)^{-1}$ is the Moore-Penrose pseudo-inverse of $\mathbf{R}$, and $\mathbf{w}_t$ evolves in the same way as if we are running Algorithm 1 on the original problem.

We will be effectively updating the coordinates of the parameter vector $\mathbf{w}$ in chunks of size $m/(1+\frac{pm}{d})$ or its integer multiples (in fact, $\frac{pm}{d}$ is equal to ϵ in the following Theorem 2, which is upper-bounded by $m-1$ and can be arbitrarily small depending on the corruption threshold). In particular, if each worker i updates k coordinates of $\mathbf{v}_i$, then $km/(1+\frac{pm}{d})$ coordinates of $\mathbf{w}$ will get updated. For comparison, Algorithm 1 updates km coordinates of the parameter vector $\mathbf{w}$ in each iteration, if each worker updates k coordinates in that iteration. The main result for our Byzantine-resilient distributed coordinate descent is stated below, which is proved in Section 5.

Theorem 2 (Coordinate Descent)**.** *Under the setting of Theorem 1, our Byzantine-resilient distributed CD algorithm has the following guarantees, where* $\epsilon > 0$ *is a free parameter.*

- $(s+t) \leq \left\lfloor \frac{\epsilon}{1+\epsilon} \cdot \frac{m}{2} \right\rfloor$.
- *Total storage requirement is roughly* $2(1+\epsilon)|\mathbf{X}|$.
- *If each worker* i *updates* τ *coordinates of* $\mathbf{v}_i$*, then*
  - $\frac{\tau m}{1+\epsilon}$ *coordinates of the corresponding* $\mathbf{w}$ *gets updated.*
  - *the computational complexity in each iteration*
    - *at each worker node is* $O(n\tau)$.
    - *at the master node is* $O((1+\epsilon)nm + \tau m^2)$.
  - *the communication complexity in each iteration*
    - *each worker sends* $\left(\tau + (1+\epsilon)\frac{n}{m}\right)$ *real numbers.*
    - *master broadcasts* $\left(\frac{\tau m}{1+\epsilon} + n\right)$ *real numbers.*
- *Total encoding time is* $O\left(nd\left(\frac{\epsilon}{1+\epsilon}m+1\right)\right)$.

Remark 6. *We compare the resource requirements of our method with the plain distributed CD described in Algorithm 1 that does not provide any adversarial protection. In Algorithm 1, if each worker* i *updates* $\tau/(1+\epsilon)$ *coordinates of* $\mathbf{w}_i$ *(in total* $\tau m/(1+\epsilon)$ *coordinates of* $\mathbf{w}$*) in each iteration, then* (i) *each worker*

requires $O(n\tau/(1+\epsilon))$ time to multiply $\mathbf{X}_i$ with the updated part of $\mathbf{w}_i$; (ii) *master requires $O(nm)$ time to compute $\sum_{i=1}^{m} \mathbf{X}_i\mathbf{w}_i$ from $\{\mathbf{X}_i\mathbf{w}_i\}_{i\in[m]}$;* (iii) *each worker sends n real numbers (required for $\mathbf{X}_i\mathbf{w}_i$) to master; and* (iv) *master broadcasts n real numbers (required for $\phi'(\mathbf{X}\mathbf{w};\mathbf{y})$).*

In our scheme (i) *the total storage requirement is $O(1+\epsilon)$ factor more;* (ii) *the amount of computation at each worker node is $O(1+\epsilon)$ factor more;* (iii) *the amount of computation at the master node is $O((1+\epsilon)+\frac{\tau m}{n})$ factor more – typically, since τ is a constant and number of workers is much less than n, this again could be $O(1+\epsilon)$;* (iv) *master broadcasts $\left(1+\frac{\tau m}{(1+\epsilon)n}\right)$ factor more data, which could be a constant if τm is smaller than $(1+\epsilon)n$; and* (v) *each worker sends $\left(\frac{\tau}{n}+\frac{(1+\epsilon)}{nm}\right)$ factor more data, where the 1st term is much smaller than 1 as τ is typically a constant, and the 2nd term is close to zero as $(1+\epsilon)$ is always upper-bounded by m.*

The Remark 3, 4, 5 are also applicable for Theorem 2.

3 Related Work

There has been significant recent interest in using coding-theoretic techniques to mitigate the well-known straggler problem [DB13], including gradient coding [TLDK17, RTDT18, CP18, HRSH18], encoding computation [LLP+18, DCG16], data encoding [KSDY17]. However, one cannot directly apply the methods for straggler mitigation to the Byzantine attacks case, as we do not know which updates are under attack. Distributed computing with Byzantine adversaries is a richly investigated topic since [LSP82], and has received recent attention in the context of large-scale distributed optimization and learning [CSX17,CWCP18,YCRB18,BMGS17]. These can be divided into two categories, one which have statistical analysis/assumptions (either explicit statistical models for data [CSX17, YCRB18], or through stochastic methods (e.g., stochastic gradient descent) [BMGS17]. Our method gives deterministic guarantees, distinct from these works, but similar in spirit to [CWCP18], which is the closest related work. Our storage redundancy *factor* is $2m/(m-2t)$, which is a constant *even* if t is a constant fraction of m. In contrast, the storage redundancy *factor* required in [CWCP18] is $2t+1$, growing linearly with the number of corrupt worker nodes.[11] This significantly reduces the computation time at the worker nodes in our scheme compared to the scheme in [CWCP18], without sacrificing much on the computation time required by the master node. Their coding in [CWCP18] is restricted to data replication redundancy, as they encode the gradient as done in [TLDK17], enabling application to (non)-convex problems; in contrast, we encode the data enabling significantly smaller redundancy, and apply it to learn generalized linear models, and is also applicable to MV multiplication.

A two-round approach for gradient computation has been proposed for straggler mitigation [LLP+18], but our method for MV multiplication differs from that, as we have to provide adversarial protection. The coding scheme in [LLP+18] for straggler mitigation is using a general-purpose MDS code for correcting erasures, which is analogous to the "trivial approach" that we described on page 7 for Byzantine-resilient gradient computation in handling erasures/stragglers. In contrast, we exploit the block error pattern and design efficient codes which have much lower computational complexity at the master node, without compromising on the storage requirement. Since iterative algorithms compute gradients repetitively, it is crucial to have a method that has as low computational complexity as possible, both at the worker nodes as well as at the master node. Data encoding proposed in [KSDY17] for straggler mitigation applies to both GD and CD, but only for quadratic problem. It achieves low redundancy and low complexity, by allowing convergence to an approximate rather than exact solution. As far as we know, making distributed CD resilient against Byzantine attacks has not seen much attention.

[11] To highlight the storage redundancy gain of our method over that of [CWCP18], consider the following two concrete scenarios: (i) In a large setup with $m = 1000$ worker nodes, if we want resiliency against $t = 100$ corrupt nodes (1/10 nodes are corrupt), our method requires redundancy of 2.5, whereas [CWCP18] requires redundancy of 201, a factor of 80 more than ours. (ii) In a moderate setup with $m = 150$ and $t = 50$ (1/3 nodes are corrupt), the redundancy of our method is 6, whereas [CWCP18] requires redundancy of 101.

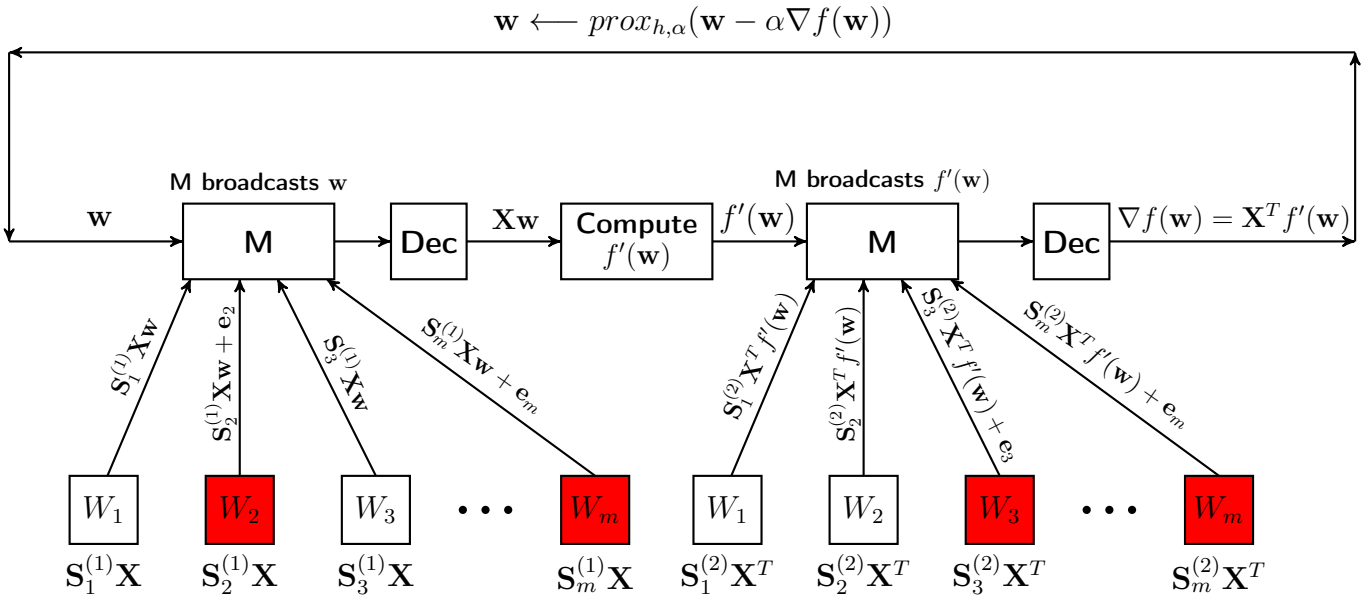

Figure 1 This figure shows our 2-round approach to the Byzantine-resilient distributed optimization given in (1) to learn a generalized linear model. Since gradient is equal to $\nabla f(\mathbf{w}) = \mathbf{X}^T f'(\mathbf{w})$ (see (7)), we compute it in 2 rounds, using a matrix-vector (MV) multiplication as a subroutine in each round. In the 1st round, first we compute $\mathbf{X}\mathbf{w}$, and then compute $f'(\mathbf{w})$ from $\mathbf{X}\mathbf{w}$ – since j'th entry of $\mathbf{X}\mathbf{w}$ is equal to $\langle \mathbf{x}_j, \mathbf{w}\rangle$, we can compute $f'(\mathbf{w})$ from $\mathbf{X}\mathbf{w}$ (see Section 2.4). In the 2nd round we compute $\mathbf{X}^T f'(\mathbf{w})$, which is equal to $\nabla f(\mathbf{w})$ using another application of MV multiplication. For a matrix $\mathbf{A}$ and a vector $\mathbf{v}$, to make our distributed MV multiplication $\mathbf{A}\mathbf{v}$ Byzantine-resilient, we encode $\mathbf{A}$ using a sparse matrix $\mathbf{S} = [\mathbf{S}_1^T\ \mathbf{S}_m^T\ \ldots\ \mathbf{S}_m^T]^T$ and distribute $\mathbf{S}_i\mathbf{A}$ to worker i (denoted by W_i). The adversary can corrupt at most t workers (the compromised ones are denoted in red colour), potentially different sets of t workers in different rounds. The master node (denoted by $\mathbf{M}$) broadcasts $\mathbf{v}$ to all the workers. Each worker performs the local MV product and sends it back to M. If W_i is corrupt, then it can send an arbitrary vector. Once the master has received all the vectors (out of which t may be erroneous), it sends them to the decoder (denoted by **Dec**), which outputs the correct MV product $\mathbf{A}\mathbf{v}$.

4 Our Solution to Gradient Computation

In this section, we describe the core technical part of our two-round approach for gradient computation described in Section 2.4 – a method of performing matrix-vector (MV) multiplication in a distributed manner in the presence of a malicious adversary who can corrupt at most t of the m worker nodes. Here, the matrix is fixed and we want to right-multiply a vector with this matrix.

Given a fixed matrix $\mathbf{A} \in \mathbb{R}^{n_r \times n_c}$ and a vector $\mathbf{v} \in \mathbb{R}^{n_c}$, we want to compute $\mathbf{A}\mathbf{v}$ in a distributed manner in the presence of at most t corrupt worker nodes; see Section 2.3 for details on our adversary model. Our method is based on data encoding and real error correction, where the matrix $\mathbf{A}$ is encoded and distributed among all the worker nodes, and the master recovers the MV product $\mathbf{A}\mathbf{v}$ using real-error correction; see Figure 1. We will think of our encoding matrix as $\mathbf{S} = [\mathbf{S}_1^T\ \mathbf{S}_2^T, \ldots, \mathbf{S}_m^T]$, where each $\mathbf{S}_i$ is a $p \times n_r$ matrix and $pm \geq n_r$. We will determine the value of p and the entries of $\mathbf{S}$ later. For $i \in [m]$, we store the matrix $\mathbf{S}_i\mathbf{A}$ at the i'th worker node. As described in Section 2, the computation proceeds as follows: The master sends $\mathbf{v}$ to all the worker nodes and receives $\{\mathbf{S}_i\mathbf{A}\mathbf{v} + \mathbf{e}_i\}_{i=1}^m$ back from them. Let $\mathbf{e}_i = [e_{i1}, e_{i2}, \ldots, e_{ip}]^T$ for every $i \in [p]$. Note that $\mathbf{e}_i = \mathbf{0}$ if the i'th node is honest, otherwise can be arbitrary. In order to find the set of corrupt worker nodes, master equivalently writes $\{\mathbf{S}_i\mathbf{A}\mathbf{v} + \mathbf{e}_i\}_{i=1}^m$ as p systems of linear equations.

$$\tilde{h}_i(\mathbf{v}) = \tilde{\mathbf{S}}_i\mathbf{A}\mathbf{v} + \tilde{\mathbf{e}}_i, \quad i \in [p] \tag{8}$$

where, for every $i \in [p]$, $\tilde{\mathbf{e}}_i = [e_{1i}, e_{2i}, \ldots, e_{mi}]^T$, and $\tilde{\mathbf{S}}_i$ is an $m \times n_r$ matrix whose j'th row is equal to the i'th row of $\mathbf{S}_j$, for every $j \in [m]$. Note that at most t entries in each $\tilde{\mathbf{e}}_i$ are non-zero. Observe that $\{\mathbf{S}_i\mathbf{A}\mathbf{v} + \mathbf{e}_i\}_{i=1}^m$ and $\{\tilde{\mathbf{S}}_i\mathbf{A}\mathbf{v} + \tilde{\mathbf{e}}_i\}_{i=1}^p$ are equivalent systems of linear equations, and we can get one from the other.

Note that $\tilde{\mathbf{S}}_i$'s constitute the encoding matrix $\mathbf{S}$, which we have to design. In the following, we will design these matrices $\tilde{\mathbf{S}}_i$'s (which in turn will determine the encoding matrix $\mathbf{S}$), with the help of another matrix $\mathbf{F}$, which will be used to find the error locations, i.e., identities of the compromised worker nodes.

We will design the matrices $\mathbf{F}$ (of dimension $k \times m$, where $k < m$, which is to be determined later) and $\tilde{\mathbf{S}}_i$'s such that

C.1 $\mathbf{F}\tilde{\mathbf{S}}_i = 0$ for every $i \in [p]$.

C.2 For any t-sparse $\mathbf{u} \in \mathbb{R}^m$, we can efficiently find all the non-zero locations of $\mathbf{u}$ from $\mathbf{Fu}$.

C.3 For any $\mathcal{T} \subset [m]$ such that $|\mathcal{T}| \geq (m-t)$, let $\mathbf{S}_\mathcal{T}$ denote the $|\mathcal{T}|p \times n_r$ matrix obtained from $\mathbf{S}$ by restricting it to all the $\mathbf{S}_i$'s for which $i \in \mathcal{T}$. We want $\mathbf{S}_\mathcal{T}$ to be of full column rank.

If we can find such matrices, then we can recover the desired MV multiplication $\mathbf{Av}$ exactly: briefly, **C.1** and **C.2** will allow us to locate the corrupt worker nodes; once we have found them, we can discard all the information that the master node had received from them. This will yield $\mathbf{S}_\mathcal{T}\mathbf{Av}$, where $\mathbf{S}_\mathcal{T}$ is the $|\mathcal{T}|p \times n_r$ matrix obtained from $\mathbf{S}$ by restricting it to $\mathbf{S}_i$'s for all $i \in \mathcal{T}$, where $\mathcal{T}$ is the set of all honest worker nodes. Now, by **C.3**, since $\mathbf{S}_\mathcal{T}$ is of full column rank, we can recover $\mathbf{Av}$ from $\mathbf{S}_\mathcal{T}\mathbf{Av}$ exactly. Details follow.

Suppose we have matrices $\mathbf{F}$ and $\tilde{\mathbf{S}}_i$'s such that **C.1** holds. Now, multiplying (8) by $\mathbf{F}$ yields

$$\mathbf{f}_i := \mathbf{F}\tilde{h}_i(\mathbf{v}) = \mathbf{F}\tilde{\mathbf{e}}_i, \tag{9}$$

for every $i \in [p]$, where $\|\tilde{\mathbf{e}}_i\|_0 \leq t$. In Section 4.1, we give our approach for finding all the corrupt worker nodes with the help of any error locator matrix $\mathbf{F}$. Then, in Section 4.2, we give a generic construction for designing $\tilde{\mathbf{S}}_i$'s (and, in turn, our encoding matrix $\mathbf{S}$) such that **C.1** and **C.3** hold. In Section 4.3, we show how to compute the desired matrix-vector product $\mathbf{Av}$ efficiently, once we have discarded all the data from the corrupt works nodes. Then, in Section 4.4, we will give details of the error locator matrix $\mathbf{F}$ that we use in our construction.

Remark 7. *As we will see in Section 4.2, the structure of our encoding matrix $\mathbf{S}$ is independent of our error locator matrix $\mathbf{F}$. Specifically, the repetitive structure of the non-zero entries of $\mathbf{S}$ as well as their locations will not change irrespective of what the $\mathbf{F}$ matrix is. This makes our construction very generic, as we can choose whichever $\mathbf{F}$ suits our needs the best (in terms of how many erroneous indices it can locate and with what decoding complexity), and it won't affect the structure of our encoding matrix at all – only the non-zero entries might change, neither their repetitive format, nor their locations!*

4.1 Finding The Corrupt Worker Nodes

Observe that $\mathsf{supp}(\tilde{\mathbf{e}}_i)$ may not be the same for all $i \in [p]$, but we know, for sure, that the non-zero locations in all these error vectors occur within the same set of t locations. Let $\mathcal{I} = \bigcup_{i=1}^{p} \mathsf{supp}(\tilde{\mathbf{e}}_i)$, which is the set of all corrupt worker nodes. Note that $|\mathcal{I}| \leq t$. We want to find this set $\mathcal{I}$ efficiently, and for that we note the following crucial observation. Since the non-zero entries of all the error vectors $\tilde{\mathbf{e}}_i$'s occur in the same set $\mathcal{I}$, a random linear combination of $\tilde{\mathbf{e}}_i$'s has support equal to $\mathcal{I}$ with probability one, if the coefficients of the linear combination are chosen from an *absolutely continuous* probability distribution. This idea has appeared before in [ME08] in the context of compressed sensing for recovering arbitrary sets of jointly sparse signals that have been measured by the same measurement matrix.

Definition 1. *A probability distribution is called* absolutely continuous, *if every event of measure zero occurs with probability zero.*

It is well-known that a distribution is absolutely continuous if and only if it can be represented as an integral over an integrable density function [Bil95, Theorem 31.8, Chapter 6]. Since Gaussian and uniform distributions have an explicit integrable density function, both are absolutely continuous. Conversely, discrete distributions are not absolutely continuous. Now we state a lemma from [ME08] that shows that a random linear combination of the error vectors (where coefficients are chosen from an absolutely continuous distribution) preserves the support with probability one.

Lemma 1 ([ME08]). *Let $\mathcal{I} = \bigcup_{i=1}^{p} \mathsf{supp}(\tilde{\mathbf{e}}_i)$, and let $\hat{\mathbf{e}} = \sum_{i=1}^{p} \alpha_i \tilde{\mathbf{e}}_i$, where α_i's are sampled i.i.d. from an absolutely continuous distribution. Then with probability 1, $\mathsf{supp}(\hat{\mathbf{e}}) = \mathcal{I}$.*

From (9) we have $\mathbf{f}_i = \mathbf{F}\tilde{\mathbf{e}}_i$ for every $i \in [p]$. Take a random linear combination of $\mathbf{f}_i$'s with coefficients α_i's chosen i.i.d. from an absolutely continuous distribution, for example, the Gaussian distribution. Let $\tilde{\mathbf{f}} = \alpha_i \left(\sum_{i=1}^{p} \mathbf{f}_i\right) = \alpha_i \left(\sum_{i=1}^{p} \mathbf{F}\tilde{\mathbf{e}}_i\right) = \mathbf{F}\left(\sum_{i=1}^{p} \alpha_i \tilde{\mathbf{e}}_i\right) = \mathbf{F}\tilde{\mathbf{e}}$, where $\tilde{\mathbf{e}} = \sum_{i=1}^{p} \alpha_i \tilde{\mathbf{e}}_i$. Note that, with probability 1, $\mathsf{supp}(\tilde{\mathbf{e}})$ is equal to the set of all corrupt worker nodes, and we want to find this set efficiently. In other words, given $\mathbf{F}\tilde{\mathbf{e}}$, we want to find $\mathsf{supp}(\tilde{\mathbf{e}})$ efficiently. For this, we need to design a $k \times m$ matrix $\mathbf{F}$ (where $k < m$) such that for any sparse error vector $\mathbf{e} \in \mathbb{R}^m$, we can efficiently find $\mathsf{supp}(\mathbf{e})$. Many such matrices have been known in the literature that can handle different levels of sparsity with varying decoding complexity. We can choose any of these matrices depending on our need, and this will not affect the design of our encoding matrix $\mathbf{S}$. In particular, we will use a $k \times m$ Vandermonde matrix along with the Reed-Solomon decoding, which can correct up to $k/2$ errors and has decoding complexity of $O(m^2)$; see Section 4.4 for details.

Time required in finding the corrupt worker nodes. The time taken in finding the corrupt worker nodes is equal to the sum of the time taken in the following 3 tasks. (i) Computing $\mathbf{F}\tilde{\mathbf{e}}_i$ for every $i \in [p]$: Note that we can get $\mathbf{F}\tilde{\mathbf{e}}_i$ by multiplying (8) with $\mathbf{F}$. Since $\mathbf{F}$ is a $k \times m$ matrix, and we compute $\mathbf{F}\tilde{h}_i(\mathbf{v})$ for p systems, this requires $O(pkm)$ time. (ii) Taking a random linear combination of p vectors each of length m, which takes $O(pm)$ time. (iii) Applying Lemma 2 (in Section 4.4) once to find the error locations, which takes $O(m^2)$ time. Since p is much bigger than m, the total time complexity is $O(pkm)$.

4.2 Designing The Encoding Matrix S

Now we give a generic construction for designing $\tilde{\mathbf{S}}_i$'s such that **C.1** and **C.3** hold. Fix any $k \times m$ matrix $\mathbf{F}$ such that we can efficiently find $\mathbf{e}$ from $\mathbf{Fe}$, provided $\mathbf{e}$ is sufficiently sparse. We can assume, without loss of generality, that $\mathbf{F}$ has full row-rank; otherwise, there will be redundant observations in $\mathbf{Fe}$ that we can discard and make $\mathbf{F}$ smaller by discarding the redundant row. Let $\mathcal{N}(\mathbf{F}) \subset \mathbb{R}^m$ denote the null-space of $\mathbf{F}$. Since $\mathsf{rank}(\mathbf{F}) = k$, dimension of $\mathcal{N}(\mathbf{F})$ is $q = (m - k)$. Let $\{\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_q\}$ be a basis of $\mathcal{N}(\mathbf{F})$, and let $\mathbf{b}_i = [b_{i1}\ b_{i2} \ldots b_{im}]^T$, for every $i \in [q]$. We set $\mathbf{b}_i$'s the columns of the following matrix $\mathbf{F}^{\perp}$:

$$\mathbf{F}^{\perp} = \begin{bmatrix} b_{11} & b_{21} & \ldots & b_{q1} \\ b_{12} & b_{22} & \ldots & b_{q2} \\ \vdots & \vdots & \vdots & \vdots \\ b_{1m} & b_{2m} & \ldots & b_{qm} \end{bmatrix}_{m \times q} \tag{10}$$

The following property of $\mathbf{F}^{\perp}$ will be used for recovering the MV product in Section 4.3.

Claim 1. *For any subset $\mathcal{T} \subset [m]$, such that $|\mathcal{T}| \geq (m - t)$, let $\mathbf{F}^{\perp}_{\mathcal{T}}$ be the $|\mathcal{T}| \times q$ matrix, which is equal to the restriction of $\mathbf{F}^{\perp}$ to the rows in $\mathcal{T}$. Then $\mathbf{F}^{\perp}_{\mathcal{T}}$ is of full column rank.*

Proof. Note that $q = m - k$, where $k = 2t$. So, if we show that any q rows of $\mathbf{F}^{\perp}$ are linearly independent, then, this in turn will imply that for every $\mathcal{T} \subset [m]$ with $|\mathcal{T}| \geq (m - t)$, the sub-matrix $\mathbf{F}^{\perp}_{\mathcal{T}}$ will have full column rank. In the following we show that any q rows of $\mathbf{F}^{\perp}$ are linearly independent. To the contrary, suppose not; and let $\mathcal{T}' \subset [m]$ with $|\mathcal{T}'| = q$ be such that the $q \times q$ matrix $\mathbf{F}^{\perp}_{\mathcal{T}'}$ is not a full rank matrix. This implies that there exists a non-zero $\mathbf{c}' \in \mathbb{R}^q$ such that $\mathbf{F}^{\perp}_{\mathcal{T}'}\mathbf{c}' = \mathbf{0}$. Let $\mathbf{b} = \mathbf{F}^{\perp}\mathbf{c}'$. Note that $\mathbf{b} \neq \mathbf{0}$ (because columns of $\mathbf{F}^{\perp}$ are linearly independent) and also that $\|\mathbf{b}\|_0 \leq m - q = k$. Now, since $\mathbf{F}\mathbf{F}^{\perp} = \mathbf{0}$, we have $\mathbf{Fb} = \mathbf{0}$, which contradicts the fact that any k columns of $\mathbf{F}$ are linearly independent. □

Now we design $\tilde{\mathbf{S}}_i$'s. For $i \in [p]$, we set $\tilde{\mathbf{S}}_i$ as follows:

$$\tilde{\mathbf{S}}_i = \begin{bmatrix} 0 & \ldots & 0 & b_{11} & b_{21} & \ldots & b_{l1} & 0 & \ldots & 0 \\ 0 & \ldots & 0 & b_{12} & b_{22} & \ldots & b_{l2} & 0 & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \ldots & 0 & b_{1m} & b_{2m} & \ldots & b_{lm} & 0 & \ldots & 0 \end{bmatrix}$$

where $l = q$ if $i < p$; otherwise $l = n_r - (p-1)q$. The first $(i-1)q$ and the last $n_r - [(i-1)q + l]$ columns of $\tilde{\mathbf{S}}_i$ are zero. This also implies that the number of rows in each $\mathbf{S}_i$ is $p = \lceil n_r/q \rceil$.

Claim 2. *For every $i \in [p]$, we have $\mathbf{F}\tilde{\mathbf{S}}_i = 0$.*

Proof. By construction, the null-space of $\mathbf{F}$ is $\mathcal{N}(\mathbf{F}) = \mathsf{span}\{\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_q\}$, which implies that $\mathbf{F}\mathbf{b}_i = \mathbf{0}$, for every $i \in [q]$. Since all the columns of $\tilde{\mathbf{S}}_i$'s are either $\mathbf{0}$ or $\mathbf{b}_j$ for some $j \in [q]$, the claim follows. □

The above constructed matrices $\tilde{\mathbf{S}}_i$'s give the following encoding matrix $\mathbf{S}_i$ for the i'th worker node:

$$\mathbf{S}_i = \begin{bmatrix} b_{1i} \ldots b_{qi} & & & \\ & \ddots & & \\ & & b_{1i} \ldots b_{qi} & \\ & & & b_{1i} \ldots b_{li} \end{bmatrix}_{p \times n_r} \tag{11}$$

All the unspecified entries of $\mathbf{S}_i$ are zero. The matrix $\mathbf{S}_i$ is for encoding the data for worker i. By stacking up the $\mathbf{S}_i$'s horizontally gives us our desired encoding matrix $\mathbf{S}$.

To get efficient encoding, we want $\mathbf{S}$ to be as sparse as possible. Since $\mathbf{S}$ is completely determined by $\mathbf{F}^{\perp}$, whose columns are the basis vectors of $\mathcal{N}(\mathbf{F})$, it suffices to find a sparse basis for $\mathcal{N}(\mathbf{F})$. It is known that finding the sparsest basis for the null-space of a matrix is NP-hard [CP86]. Note that we can always find the basis vectors of $\mathcal{N}(\mathbf{F})$ by reducing $\mathbf{F}$ to its row-reduced-echelon-form (RREF) using the Gaussian elimination [HK71]. This will result in $\mathbf{F}^{\perp}$ whose last q rows forms a $q \times q$ identity matrix. Note that $q = m - k$, where $k = 2t$. So, if the corruption threshold t is very small as compared to m, the $\mathbf{F}^{\perp}$ that we obtain by the RREF will be very sparse – only the first $2t$ rows may be dense. Since computing $\mathbf{S}$ is equivalent to computing $\mathbf{F}^{\perp}$, and we can compute $\mathbf{F}^{\perp}$ in $O(k^2 m)$ time using the Gaussian elimination, the time complexity of computing $\mathbf{S}$ is also $O(k^2 m)$.

Now we prove an important property of the encoding matrix $\mathbf{S}$ that will be crucial for recovery of the desired matrix-vector product.

Claim 3. *For any $\mathcal{T} \subset [m]$ such that $|\mathcal{T}| \geq (m-t)$, let $\mathbf{S}_{\mathcal{T}}$ denote the $|\mathcal{T}|p \times n_r$ matrix obtained from $\mathbf{S}$ by restricting it to all the blocks $\mathbf{S}_i$'s for which $i \in \mathcal{T}$. Then $\mathbf{S}_{\mathcal{T}}$ is of full column rank.*

Proof. For $i \in [p-1]$, let $\mathcal{B}_i = [(i-1)q + 1 : iq]$ and $\mathcal{B}_p = [(p-1)q + 1 : n_r - (p-1)q]$, where we see $\mathcal{B}_i$'s as a collection of some column indices. Consider any two distinct $i, j \in [p]$. It is clear that for any two vectors $\mathbf{u}_1 \in \mathcal{B}_i, \mathbf{u}_2 \in \mathcal{B}_j$, we have $\mathsf{supp}(\mathbf{u}_1) \cap \mathsf{supp}(\mathbf{u}_2) = \phi$, which means that all the columns in distinct $\mathcal{B}_i$'s are linearly independent. So, to prove the claim, we only need to show that the columns within the same $\mathcal{B}_i$'s are linearly independent. Fix any $i \in [p]$, and consider the $|\mathcal{T}|p \times q$ sub-matrix $\mathbf{S}_{\mathcal{T}}^{(i)}$ of $\mathbf{S}_{\mathcal{T}}$, which is obtained by restricting $\mathbf{S}_{\mathcal{T}}$ to the columns in $\mathcal{B}_i$. There are precisely $|\mathcal{T}|$ non-zero rows in $\mathbf{S}_{\mathcal{T}}^{(i)}$, which are equal to the rows of the matrix $\mathbf{F}_{\mathcal{T}}^{\perp}$ defined in Claim 1. We have already shown in the proof of Claim 1 that $\mathbf{F}_{\mathcal{T}}^{\perp}$ is of full column rank. Therefore, $\mathbf{S}_{\mathcal{T}}^{(i)}$ is also of full column rank. This concludes the proof of Claim 3. □

Since $\mathbf{S}_{\mathcal{T}}$ is of full column rank, in principle, we can recover any vector $\mathbf{u} \in \mathbb{R}^{n_r}$ from $\mathbf{S}_{\mathcal{T}}\mathbf{u}$. In the next section, we show an efficient way for this recovery.

4.3 Recovering The Matrix-Vector Product Av

Once the master has found the set $\mathcal{I}$ of corrupt worker nodes, it discards all the data received from them. Let $\mathcal{T} = [m] \setminus \mathcal{I} = \{i_1, i_2, \ldots, i_f\}$ be the set of all honest worker nodes, where $f = (m - |\mathcal{I}|) \geq (m-t)$. Let $\mathbf{r} = [\mathbf{r}_1^T \mathbf{r}_2^T \ldots \mathbf{r}_m^T]$, where $\mathbf{r}_i = \mathbf{S}_i\mathbf{A}\mathbf{v} + \mathbf{e}_i$. All the $\mathbf{r}_i$'s from the honest worker nodes can be written as

$$\mathbf{r}_{\mathcal{T}} = \mathbf{S}_{\mathcal{T}}\mathbf{A}\mathbf{v}, \tag{12}$$

where $\mathbf{S}_{\mathcal{T}}$ is as defined in Claim 3, and $\mathbf{r}_{\mathcal{T}}$ is also defined analogously and equal to the restriction of $\mathbf{r}$ to all the $\mathbf{r}_i$'s for which $i \in \mathcal{T}$. Since $\mathbf{S}_{\mathcal{T}}$ has full column rank (by Claim 3), in principle, we can recover $\mathbf{A}\mathbf{v}$ from (12). Next we show how to recover $\mathbf{A}\mathbf{v}$ efficiently, by exploiting the structure of $\mathbf{S}$.

Let $\tilde{\mathbf{r}}_j = [r_{i_1 j}, r_{i_2 j}, \ldots, r_{i_f j}]^T$, for every $j \in [p]$. The repetitive structure of $\mathbf{S}_i$'s (see (11)) allows us to write (12) equivalently in terms of p smaller systems.

$$\tilde{\mathbf{r}}_j = \mathbf{F}_j(\mathbf{A}\mathbf{v})_{\mathcal{B}_j}, \quad \text{for } j \in [p], \tag{13}$$

where, for $j \in [p-1]$, $\mathcal{B}_i = [(i-1)q + 1 : iq]$ and $\mathbf{F}_j = \mathbf{F}_{\mathcal{T}}^{\perp}$, and $\mathcal{B}_p = [(p-1)q + 1 : n_r - (p-1)q]$ and $\mathbf{F}_p$ is equal to the restriction of $\mathbf{F}_{\mathcal{T}}^{\perp}$ to its first $(n_r - (p-1)q)$ columns. Since $\mathbf{F}_{\mathcal{T}}^{\perp}$ has full column rank (by Claim 1), we can compute $(\mathbf{A}\mathbf{v})_{\mathcal{B}_i}$ for all $i \in [p]$, by multiplying (13) by $\mathbf{F}_j^+ = (\mathbf{F}_j^T \mathbf{F}_j)^{-1} \mathbf{F}_j^T$, which it called the Moore-Penrose inverse of $\mathbf{F}_j$. Since $\mathbf{A}\mathbf{v} = [(\mathbf{A}\mathbf{v})_{\mathcal{B}_1}^T, (\mathbf{A}\mathbf{v})_{\mathcal{B}_2}^T, \ldots, (\mathbf{A}\mathbf{v})_{\mathcal{B}_p}^T)]^T$, we can recover the desired MV product $\mathbf{A}\mathbf{v}$.

Time Complexity analysis. The task of obtaining $\mathbf{A}\mathbf{v}$ from $\mathbf{S}_{\mathcal{T}}\mathbf{A}\mathbf{v}$ reduces to (i) computing $\mathbf{F}_j^+ = (\mathbf{F}_{\mathcal{T}}^{\perp})^+$ once, which takes $O(q^2|\mathcal{T}|)$ time naïvely; (ii) computing $\mathbf{F}_p^+$ once, which takes at most $O(q^2|\mathcal{T}|)$ time naïvely; and (iii) computing the MV products $\mathbf{F}_j^+ \tilde{\mathbf{r}}_j$ for every $j \in [p]$, which takes $O(pq|\mathcal{T}|)$ time in total. Since p is much bigger than q, the total time taken in recovering $\mathbf{A}\mathbf{v}$ from $\mathbf{S}_{\mathcal{T}}\mathbf{A}\mathbf{v}$ is $O(pq|\mathcal{T}|) = O(pm^2)$.

4.4 Designing The Error Locator Matrix F

In this section, we design a $k \times m$ matrix $\mathbf{F}$ (where $k < m$) such that for any sparse error vector $\mathbf{e} \in \mathbb{R}^m$, we can efficiently find $\mathsf{supp}(\mathbf{e})$. Many such matrices have been known in the literature (for recovering the vector $\mathbf{e}$ given $\mathbf{F}\mathbf{e}$) since the work of [CT05], that can handle different levels of sparsity with varying decoding complexity. Most of these are random constructions, which may not work with small block-lengths (in our setting, m may be a small number). Furthermore, they can only correct a constant fraction of errors, where the constant is very small. We need a deterministic construction that can handle a constant fraction (ideally up to $1/2$) of errors and that works with small block-lengths.

Akçakaya and Tarokh [AT08] constructed a complex analogue of the Reed-Solomon codes from $k \times m$ Vandermonde matrices $\mathbf{F}$ and gave an $O(m^2)$ time algorithm for exactly reconstructing $\mathbf{e}$ from $\mathbf{f} = \mathbf{F}\mathbf{e}$, provided $|\mathsf{supp}(\mathbf{e})| \leq k/2$. Let $z_1, z_2, \ldots, z_m$ be m distinct non-zero elements in $\mathbb{R}$. We define $\mathbf{F}$ to be the following Vandermonde matrix (where $k < m$):

$$\mathbf{F} = \begin{bmatrix} 1 & 1 & 1 & \ldots & 1 \\ z_1 & z_2 & z_3 & \ldots & z_m \\ z_1^2 & z_2^2 & z_3^2 & \ldots & z_m^2 \\ \vdots & \vdots & \ldots & \ldots & \vdots \\ z_1^{k-1} & z_2^{k-1} & z_3^{k-1} & \ldots & z_m^{k-1} \end{bmatrix}_{k \times m} \tag{14}$$

Below we state a result (specialized to reals) from [AT08].

Lemma 2 ([AT08]). *Let $\mathbf{F}$ be the $k \times m$ matrix as defined in (14). Let $\mathbf{e} \in \mathbb{R}^m$ be an arbitrary vector with $|\mathsf{supp}(\mathbf{e})| \leq k/2$. We can exactly recover the vector $\mathbf{e}$ from $\mathbf{f} = \mathbf{F}\mathbf{e}$ in $O(m^2)$ time.*

Note that $\mathbf{F}$ is a $k \times m$ matrix, where $k < m$. Choosing k is in our hands, and larger the k, more the number of errors we can correct (but at the expense of increased storage and computation); see Section 4.5 for more details.

4.5 Resource Requirement Analysis

In this section, we analyze the total amount of resources (storage, computation, and communication) required by our method for computing gradients in the presence of t (out of m) adversarial worker nodes and prove Theorem 1. Fix an $\epsilon > 0$. Let the corruption threshold t satisfy $t \leq \lfloor (\epsilon/(1+\epsilon)) \cdot (m/2) \rfloor$.

As described earlier in Section 2.4, we compute the gradient $\nabla f(\mathbf{w}) = \mathbf{X}^T f'(\mathbf{w})$ in two-rounds; and in each round we use the Byzantine-tolerant MV multiplication, which we have developed in Section 4, as a subroutine; see Figure 1 for a pictorial representation of our scheme. We encode $\mathbf{X}$ to compute $f'(\mathbf{w})$ in the 1st round: first compute $\mathbf{X}\mathbf{w}$ using MV multiplication and then locally compute $f'(\mathbf{w})$.

To compute $\mathbf{X}^T f'(\mathbf{w})$ (which is equal to the gradient) in the 2nd round, we encode $\mathbf{X}^T$ and compute $\mathbf{X}^T f'(\mathbf{w})$. Let $\mathbf{S}^{(1)}$ and $\mathbf{S}^{(2)}$ be the encoding matrices of dimensions $p_1 m \times n$ and $p_2 m \times d$, respectively, to encode $\mathbf{X}$ and $\mathbf{X}^T$, respectively. Here, $p_1 = \lceil n/q \rceil$ and $p_2 = \lceil d/q \rceil$, where $q = m - k$. Since $k = 2t$ (by Lemma 2), we have $q = (m-k) \geq m/(1+\epsilon)$.

4.5.1 Storage Requirement

Each worker node i stores two matrices $\mathbf{S}_i^{(1)}\mathbf{X}$ and $\mathbf{S}_i^{(2)}\mathbf{X}^T$. The first one is a $p_1 \times (d+1)$ matrix, and the second one is a $p_2 \times n$ matrix. So, the total amount of storage at all worker nodes is equal to storing $(p_1(d+1) + p_2 n) \times m$ real numbers. Since $p_1 \leq \lceil (1+\epsilon)\frac{n}{m} \rceil$ and $p_2 \leq \lceil (1+\epsilon)\frac{d}{m} \rceil$, the total storage is

$$\begin{aligned}\big(p_1(d+1) + p_2 n\big)m &= p_1 m(d+1) + p_2 mn \\ &< [(1+\epsilon)n + m](d+1) + [(1+\epsilon)d + m]n \\ &= (1+\epsilon)n(2d+1) + m(n+d+1).\end{aligned}$$

where the first term is roughly equal to a $2(1+\epsilon)$ factor more than the size of $\mathbf{X}$. Note that the second term does not contribute much to the total storage as compared to the first term, because the number of worker nodes m is much smaller than both n and d. In fact, if $m-k$ divides both n and d, then the second term vanishes. Since $|\mathbf{X}|$ is an $n \times d$ matrix, the total storage at each worker node is almost equal to $2(1+\epsilon)\frac{|\mathbf{X}|}{m}$, which is a constant factor of the optimal, that is, $\frac{|\mathbf{X}|}{m}$, and the total storage is roughly equal to $2(1+\epsilon)|\mathbf{X}|$.

4.5.2 Computational Complexity

We can divide the computational complexity of our scheme as follows:

- *Encoding the data matrix.* Since, for every $i \leq k$ and $j > k$, the total number of non-zero entries in $\mathbf{S}_i^{(1)}$ and $\mathbf{S}_j^{(1)}$ are at most n and p_1, respectively (see Section 4.2 for details), the computational complexity for computing $\mathbf{S}_i^{(1)}\mathbf{X}$ for each $i \leq k$, and $\mathbf{S}_j^{(1)}\mathbf{X}$ for each $j > k$, is $O(nd)$ and $O(p_1 d)$, respectively. So, the encoding time for computing $\mathbf{S}^{(1)}\mathbf{X}$ is equal to $O\left(k(nd) + (m-k)(p_1 d)\right) = O\left((\frac{\epsilon}{1+\epsilon}m + 1)nd\right)$. Similarly, we can show that the encoding time for computing $\mathbf{S}^{(2)}\mathbf{X}^T$ is also equal to $O\left((\frac{\epsilon}{1+\epsilon}m + 1)nd\right)$. Note that computing $\mathbf{S}^{(1)}$ and $\mathbf{S}^{(2)}$ take $O(k^2 m)$ time each, which is much smaller, as compared to the encoding time. So, the total encoding time is $O\left((\frac{\epsilon}{1+\epsilon}m + 1)nd\right)$. Note that this encoding is to be done only once.

- *Computation at each worker node.* In the first round, upon receiving $\mathbf{w}$ from the master node, each worker i computes $(\mathbf{S}_i^{(1)}\mathbf{X})\mathbf{w}$, and reports back the resulting vector. Similarly, in the second round, upon receiving $f'(\mathbf{w})$ from the master node, each worker i computes $(\mathbf{S}_i^{(2)}\mathbf{X}^T)f'(\mathbf{w})$, and reports back the resulting vector. Since $\mathbf{S}_i^{(1)}\mathbf{X}$ and $\mathbf{S}_i^{(2)}\mathbf{X}^T$ are $p_1 \times (d+1)$ and $p_2 \times n$ matrices, respectively, each worker node i requires $O(p_1 d + p_2 n) = O((1+\epsilon)\frac{nd}{m})$ time.

- *Computation at the master node.* The total time taken by the master node in both the rounds is the sum of the time required in (i) finding the corrupt worker nodes in the 1st and 2nd rounds, which requires $O(p_1 km)$ and $O(p_2 km)$ time, respectively (see Section 4.1), (ii) recovering $\mathbf{X}\mathbf{w}$ from $\mathbf{S}_{\mathcal{T}}^{(1)}\mathbf{X}\mathbf{w}$ in the 1st round, which requires $O(p_1 m^2)$ time, (iii) computing $f'(\mathbf{w})$ from $\mathbf{X}\mathbf{w}$, which takes $O(n)$ time, and (iv) recovering $\mathbf{X}^T f'(\mathbf{w})$ from $\mathbf{S}_{\mathcal{T}}^{(2)}\mathbf{X}^T f'(\mathbf{w})$ in the 2nd round, which requires $O(p_2 m^2)$ time (see Section 4.3). Since $k < m$, the total time is equal to $O((p_1+p_2)m^2) = O((1+\epsilon)(n+d)m)$.

4.5.3 Communication Complexity

In each gradient computation, (i) master broadcasts $(n+d)$ real numbers, d in the first round and n in the second round; and (ii) each worker sends $\left((1+\epsilon)\frac{n+d}{m}\right)$ real numbers to master, $(1+\epsilon)\frac{n}{m}$ in the first round and $(1+\epsilon)\frac{d}{m}$ in the second round.

5 Our Solution to Coordinate Descent

In this section, we give a solution to the distributed coordinate descent (CD) under Byzantine attacks and prove Theorem 2. To make our notation simpler, we remove the dependence on the label vector $\mathbf{y}$ in the problem expression (5) and rewrite it as follows (this is without loss of generality in the light of Footnote 2 and Algorithm 1):

$$\arg\min_{\mathbf{w}\in\mathbb{R}^d} \phi(\mathbf{X}\mathbf{w}) := \sum_{i=1}^{n} \ell(\langle \mathbf{x}_i, \mathbf{w}\rangle). \tag{15}$$

We want to optimize (15) using distributed CD, described in Section 2.2. As outlined in Section 2.5, we use data encoding and error correction over real numbers for that. To combat the effect of adversary, we add redundancy to enlarge the parameter space. Let $\widetilde{\mathbf{X}}^R = \mathbf{X}\mathbf{R}$, where $\mathbf{R} = [\mathbf{R}_1\ \mathbf{R}_2\ \dots\ \mathbf{R}_m] \in \mathbb{R}^{d\times pm}$ with $pm \geq d$, and each $\mathbf{R}_i$ is a $p\times d$ matrix. We will determine $\mathbf{R}$ and the value of p later. We consider $\mathbf{R}$'s which are of full row-rank. Let $\mathbf{R}^+ := \mathbf{R}^T(\mathbf{R}\mathbf{R}^T)^{-1}$ denote its Moore-Penrose inverse such that $\mathbf{R}\mathbf{R}^+ = I_d$, where I_d is the $d\times d$ identity matrix. Note that $\mathbf{R}^+$ is of full column-rank. Let $\mathbf{v} = \mathbf{R}^+\mathbf{w}$ be the transformed vector, which lies in a larger (than d) dimensional space. Let $\mathbf{R}^+ = [(\mathbf{R}_1^+)^T\ (\mathbf{R}_2^+)^T\ \dots\ (\mathbf{R}_m^+)^T]^T$, where each $\mathbf{R}_i^+ := (\mathbf{R}^+)_i$ is a $p\times d$ matrix. With this, by letting $\mathbf{v} = [\mathbf{v}_1^T\ \mathbf{v}_2^T\ \dots\ \mathbf{v}_m^T]^T$, we have that $\mathbf{v}_i = \mathbf{R}_i^+\mathbf{w}$ for every $i\in[m]$. Now, consider the following modified problem over the encoded data.

$$\arg\min_{\mathbf{v}\in\mathbb{R}^{pm}} \phi(\widetilde{\mathbf{X}}^R\mathbf{v}). \tag{16}$$

Observe that, since $\mathbf{R}$ is of full row-rank, $\min_{\mathbf{w}\in\mathbb{R}^d}\phi(\mathbf{X}\mathbf{w})$ is equal to $\min_{\mathbf{v}\in\mathbb{R}^{pm}}\phi(\widetilde{\mathbf{X}}^R\mathbf{v})$; and from an optimal solution to one problem we can obtain an optimal solution to the other problem. We design an encoding/decoding scheme such that when we optimize the encoded problem (16) using Algorithm 1, the vector $\mathbf{v}$ that we get in each iteration is of the form $\mathbf{v} = \mathbf{R}^+\mathbf{w}$ for some vector $\mathbf{w}\in\mathbb{R}^d$.[12] It may not be clear why we need this, but as we see later, this property will be crucial in our solution.

Now, instead of solving (15), we solve its encoded form (16) using Algorithm 1 (with decoding at the master), where each worker i stores $\widetilde{\mathbf{X}}_i^R = \mathbf{X}\mathbf{R}_i$ and is responsible for updating (some coordinates of) $\mathbf{v}_i$. In the following, let $\mathcal{U}\subseteq[p]$ be a fixed arbitrary subset of $[p]$. Let $\mathbf{v}^0 := \mathbf{R}^+\mathbf{w}^0$ for some $\mathbf{w}^0$ at time $t=0$. Suppose, at the beginning of the t'th iteration, we have $\mathbf{v}^t = \mathbf{R}^+\mathbf{w}^t$ for some $\mathbf{w}^t$, and each worker i updates $\mathbf{v}_{i\mathcal{U}}^t$ according to

$$\mathbf{v}_{i\mathcal{U}}^{t+1} = \mathbf{v}_{i\mathcal{U}}^t - \alpha_t \nabla_{i\mathcal{U}}\phi(\widetilde{\mathbf{X}}^R\mathbf{v}^t), \tag{17}$$

where $\nabla_{i\mathcal{U}}\phi(\widetilde{\mathbf{X}}^R\mathbf{v}^t) = (\widetilde{\mathbf{X}}_{i\mathcal{U}}^R)^T\phi'(\widetilde{\mathbf{X}}^R\mathbf{v}^t)$. Recall that each $\mathbf{R}_i$ is a $d\times p$ matrix, and each $\mathbf{R}_i^+ := (\mathbf{R}^+)_i$ is a $p\times d$ matrix. We denote by $\mathbf{R}_{i\mathcal{U}}$ the $d\times|\mathcal{U}|$ matrix obtained by restricting the columns of $\mathbf{R}_i$ to the elements of $\mathcal{U}$. Analogously, we denote by $\mathbf{R}_{i\mathcal{U}}^+ := (\mathbf{R}^+)_{i\mathcal{U}}$ the $|\mathcal{U}|\times d$ matrix obtained by restricting the rows of $\mathbf{R}_i^+$ to the elements of $\mathcal{U}$. With this, we can write $\widetilde{\mathbf{X}}_{i\mathcal{U}}^R = \mathbf{X}\mathbf{R}_{i\mathcal{U}}$. Now, (17) can be equivalently written as

$$\mathbf{v}_{i\mathcal{U}}^{t+1} = \mathbf{v}_{i\mathcal{U}}^t - \alpha_t \mathbf{R}_{i\mathcal{U}}^T\mathbf{X}^T\phi'(\widetilde{\mathbf{X}}^R\mathbf{v}^t). \tag{18}$$

In order to update $\mathbf{v}_{i\mathcal{U}}^t$, worker i requires $\phi'(\widetilde{\mathbf{X}}^R\mathbf{v}^t)$, where $\widetilde{\mathbf{X}}^R\mathbf{v}^t = \sum_{j=1}^m \widetilde{\mathbf{X}}_j^R\mathbf{v}_j^t$ and worker i has only $(\widetilde{\mathbf{X}}_i^R, \mathbf{v}_i^t)$. Since $\mathbf{v}^t = \mathbf{R}^+\mathbf{w}^t$, we have $\widetilde{\mathbf{X}}^R\mathbf{v}^t = \mathbf{X}\mathbf{R}\mathbf{v}^t = \mathbf{X}\mathbf{w}^t$. So, it suffices to compute $\mathbf{X}\mathbf{w}^t$ at the master node – once master has $\mathbf{X}\mathbf{w}^t$, it can locally compute $\phi'(\mathbf{X}\mathbf{w}^t)$ and send it to all the workers.

[12] If such a $\mathbf{w}$ exists, then it is unique. This follows from the fact that $\mathbf{R}^+$ is of full column-rank

Computing $\mathbf{X}\mathbf{w}^t$ is the distributed matrix-vector (MV) multiplication problem, where the matrix $\mathbf{X}$ is fixed and we want to compute $\mathbf{X}\mathbf{w}^t$ for any vector $\mathbf{w}^t$ in the presence of an adversary. In Section 4, we give a method for performing distributed MV multiplication in the presence of an adversary. Now we give an overview, together-with an improvement on its computational complexity.

We encode $\mathbf{X}$ using an encoding matrix $\mathbf{L} \in \mathbb{R}^{(p'm)\times n}$. Let $\mathbf{L} = [\mathbf{L}_1^T\ \mathbf{L}_2^T\ \dots\ \mathbf{L}_m^T]^T$, where each $\mathbf{L}_i$ is a $p' \times n$ matrix with $p' = \lceil \frac{n}{m-2t} \rceil$. Each $\mathbf{L}_i$ has p' rows and n columns, and has the same structure as that of $\mathbf{S}_i$ from (11). Worker i stores $\widetilde{\mathbf{X}}_i^L = \mathbf{L}_i\mathbf{X}$. To compute $\mathbf{X}\mathbf{w}$, master sends $\mathbf{w}$ to all the workers; worker i responds with $\mathbf{L}_i\mathbf{X}\mathbf{w} + \mathbf{e}_i$, where $\mathbf{e}_i = \mathbf{0}$ if the i'th worker is honest, otherwise can be arbitrary; upon receiving $\{\mathbf{L}_i\mathbf{X}\mathbf{w} + \mathbf{e}_i\}_{i=1}^m$, where at most t of the $\mathbf{e}_i$'s can be non-zero, master applies the decoding procedure and recovers $\mathbf{X}\mathbf{w}$ back. We can improve the computational complexity of this method significantly by observing that, in each iteration of our distributed CD algorithm, only a few coordinate of $\mathbf{w}$ gets updated and the rest of the coordinates remain unchanged. (Looking ahead, when each worker updates $\mathbf{v}_{i\mathcal{U}}$'s according to (17), it automatically updates $\mathbf{w}_{f(\mathcal{U})}$ according to (6) – for a specific function f as defined in (21) – where $\mathbf{v}$ and $\mathbf{w}$ satisfy $\mathbf{v} = \mathbf{R}^+\mathbf{w}$.) This implies that for computing $\mathbf{X}\mathbf{w}$, master only needs to send the updated coordinates to the workers and keeps the result from the previous MV product with itself. This significantly reduces the local computation at the worker nodes, as now they only need to perform a local MV product of a matrix of size $p' \times |f(\mathcal{U})|$ and a vector of length $|f(\mathcal{U})|$. See Section 4 for details.

Our goal in each iteration of CD is to update some coordinates of the original parameter vector $\mathbf{w}$; instead, by solving the encoded problem, we are updating coordinates of the transformed vector $\mathbf{v}$. We would like to design an algorithm/encoding such that it has exactly the same convergence properties as if we are running the distributed CD on the original problem without any adversary! For this, naturally, we would like our algorithm to satisfy the following property:

Update on any (small) subset of coordinates of $\mathbf{w}$ *should be achieved by updating some (small) subset of coordinates of* $\mathbf{v}_i$*'s; and, by updating those coordinates of* $\mathbf{v}_i$*'s, we should be able to efficiently recover the correspondingly updated coordinates of* $\mathbf{w}$*. Furthermore, this should be doable despite the errors injected by the adversary in every iteration of the algorithm.*

Note that if each coordinate of $\mathbf{v}$ depends on too many coordinates of $\mathbf{w}$, then updating a few coordinates of $\mathbf{v}$ may affect many coordinates of $\mathbf{w}$, and it becomes information-theoretically impossible to satisfy the above property (even without the presence of an adversary). This imposes a restriction that each row of $\mathbf{R}^+$ must have few non-zero entries, in such a way that updating $\mathbf{v}_{i\mathcal{U}}^t$'s, for any choice of $\mathcal{U} \subseteq [p]$, will collectively update only a subset (which may potentially depend on $\mathcal{U}$) of coordinates of the original parameter vector $\mathbf{w}^t$, and we can uniquely and efficiently recover those updated coordinates of $\mathbf{w}^t$, even from the *erroneous* vectors $\{\mathbf{v}_{i\mathcal{U}}^{t+1} + \mathbf{e}_{i\mathcal{U}}\}_{i=1}^m$, where at most t out of m error vectors $\{\mathbf{e}_{i\mathcal{U}}\}_{i=1}^m$ are non-zero and may have arbitrary entries. In order to achieve this, we will design a sparse encoding matrix $\mathbf{R}^+$ (which in turn determines $\mathbf{R}$), that satisfies the following properties:

P.1 $\mathbf{R}^+$ has structured sparsity, which induces a map $f : [p] \to \mathcal{P}([d])$ (where $\mathcal{P}([d])$ denotes the power set of $[d]$) such that

(a) $\{f(i) : i \in [p]\}$ partitions $\{1, 2, \dots, d\}$, i.e., for every $i, j \in [p]$, such that $i \neq j$, we have $f(i) \cap f(j) = \emptyset$ and that $\bigcup_{i=1}^p f(i) = [d]$.

(b) $|f(i)| = |f(j)|$ for every $i, j \in [p-1]$, and $|f(p)| \leq |f(i)|$, for any $i \in [p-1]$.

(c) For any $\mathcal{U} \subseteq [p]$, define $f(\mathcal{U}) := \cup_{j\in\mathcal{U}} f(j)$. If we update $\mathbf{v}_{i\mathcal{U}}^t$, $\forall i \in [m]$, according to (18), it automatically updates $\mathbf{w}_{f(\mathcal{U})}^t$ according to

$$\mathbf{w}_{f(\mathcal{U})}^{t+1} = \mathbf{w}_{f(\mathcal{U})}^t - \alpha_t \mathbf{X}_{f(\mathcal{U})}^T \phi'(\mathbf{X}\mathbf{w}^t). \tag{19}$$

If we set $\mathbf{v}_{i\overline{\mathcal{U}}}^{t+1} := \mathbf{v}_{i\overline{\mathcal{U}}}^t$ and $\mathbf{w}_{f(\overline{\mathcal{U}})}^{t+1} := \mathbf{w}_{f(\overline{\mathcal{U}})}^t$, then $\mathbf{v}^{t+1} = \mathbf{R}^+\mathbf{w}^{t+1}$, i.e., our invariant holds.

Note that (19) is the same update rule if we run the plain CD algorithm to update $\mathbf{w}_{f(\mathcal{U})}$. In fact, our encoding matrix satisfies a stronger property, that $\mathbf{v}_{i\mathcal{U}}^{t+1} = \mathbf{R}_{i\mathcal{U},f(\mathcal{U})}^+ \mathbf{w}_{f(\mathcal{U})}^{t+1}$ holds for every $i \in [m]$, $\mathcal{U} \subseteq [p]$, where $\mathbf{R}_{i\mathcal{U},f(\mathcal{U})}^+$ denotes the $|\mathcal{U}| \times |f(\mathcal{U})|$ matrix obtained from $\mathbf{R}_{i\mathcal{U}}^+$ by restricting its column indices to the elements in $f(\mathcal{U})$.

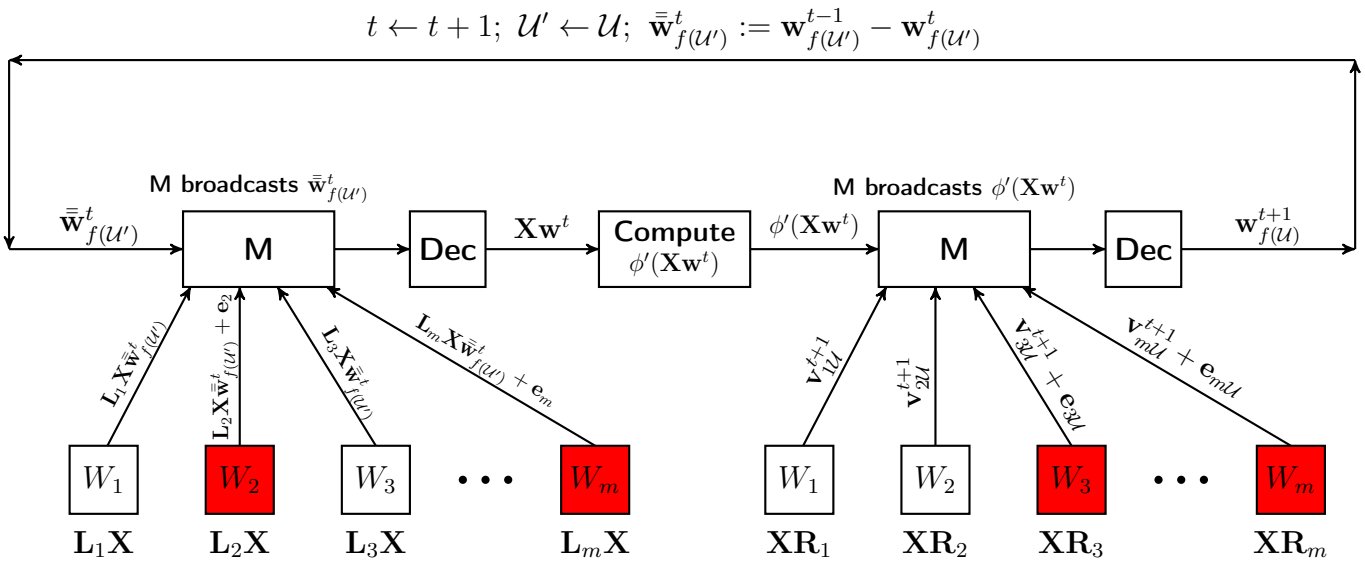

Figure 2 This figure shows our 2-round approach to the Byzantine-resilient distributed coordinate descent (CD) for solving (15) using data encoding and real-error correction. We encode $\mathbf{X}$ with the encoding matrix $[\mathbf{R}_1\ \dots\ \mathbf{R}_m] \in \mathbb{R}^{d\times p_2 m}$ and store $\widetilde{\mathbf{X}}^R_i := \mathbf{X}\mathbf{R}_i$ at the i'th worker and solve (16) over an enlarged parameter vector $\mathbf{v} \in \mathbb{R}^{p_2 m}$. At the t'th iteration, for some $\mathcal{U} \subseteq [p_2]$, the update at the i'th worker is $\mathbf{v}^{t+1}_{i\mathcal{U}} = \mathbf{v}^t_{i\mathcal{U}} - \alpha_t \mathbf{R}^T_{i\mathcal{U}}\mathbf{X}^T\phi'(\widetilde{\mathbf{X}}^R\mathbf{v}^t)$, which requires $\phi'(\widetilde{\mathbf{X}}^R\mathbf{v}^t)$, where $\widetilde{\mathbf{X}}^R\mathbf{v}^t = \mathbf{X}\mathbf{w}^t$. The first part of the figure is for providing $\phi'(\mathbf{X}\mathbf{w}^t)$ to every worker in each iteration so that they can update $\mathbf{v}^t_{i\mathcal{U}}$'s. For this, we encode $\mathbf{X}$ using the encoding matrix $[\mathbf{L}_1^T\ \dots\ \mathbf{L}_m^T]^T \in \mathbb{R}^{p_1 m\times n}$ and store $\widetilde{\mathbf{X}}^L_i := \mathbf{L}_i\mathbf{X}$ at worker i. The encoding has the property that we can recover $\mathbf{X}\mathbf{w}^t$ from the erroneous vectors $\{\widetilde{\mathbf{X}}^L_i\mathbf{w}^t + \mathbf{e}_i\}_{i=1}^m$, where at most t of the $\mathbf{e}_i$'s are non-zero and can be arbitrary. We can make it computationally more efficient at the workers' side by observing that, in each iteration, only a subset of coordinates of $\mathbf{w}$ are being updated: suppose we updated $\mathbf{v}^t_{i\mathcal{U}'}$'s in the t'th iteration, which automatically updated $\mathbf{w}^t_{f(\mathcal{U}')}$. Since $\mathbf{w}^t_{[d]\setminus f(\mathcal{U}')}$ remain unchanged, we need to send only $\mathbf{w}^t_{f(\mathcal{U}')}$ to the workers – in the figure, to take care of a technicality, we let master broadcast $\bar{\tilde{\mathbf{w}}}^t_{f(\mathcal{U}')} := \mathbf{w}^{t-1}_{f(\mathcal{U}')} - \mathbf{w}^t_{f(\mathcal{U}')}$, each worker i computes $\widetilde{\mathbf{X}}_i\bar{\tilde{\mathbf{w}}}^t_{f(\mathcal{U}')}$ and sends it backs to the master. Since master keeps $\mathbf{X}\mathbf{w}^{t-1}$ from the previous iteration with itself, it can compute $\mathbf{X}\mathbf{w}^t$. The set of corrupt workers may be different in different rounds – the corrupt ones are shown in red color and they can send arbitrary outcomes to master. Once master has recovered $\mathbf{X}\mathbf{w}^t$, it computes $\phi'(\mathbf{X}\mathbf{w}^t)$ and broadcasts it; upon receiving it worker i updates $\mathbf{v}^{t+1}_{i\mathcal{U}}$ and sends it back. By **P.1**, this reflects an update on $\mathbf{w}^{t+1}_{f(\mathcal{U})}$ according to (19); and by **P.2**, the master can recover $\mathbf{w}^{t+1}_{f(\mathcal{U})}$.

P.2 We can efficiently recover $\mathbf{w}^{t+1}_{f(\mathcal{U})}$ from the erroneous vectors $\{\mathbf{v}^{t+1}_{i\mathcal{U}} + \mathbf{e}_{i\mathcal{U}}\}_{i=1}^m$, where at most t of $\mathbf{e}_{i\mathcal{U}}$'s are non-zero and may have arbitrary entries. Since $\mathbf{v}^{t+1}_{i\mathcal{U}} = \mathbf{R}^+_{i\mathcal{U},f(\mathcal{U})}\mathbf{w}^{t+1}_{f(\mathcal{U})}$, for every $i \in [m]$, $\mathcal{U} \subseteq [p]$, this property requires that not only $\mathbf{R}^+$, but its sub-matrices also have error correcting capabilities!

Remark 8. *Note that* **P.1** *implies that for every $i \in [p]$, we have $|f(i)| \le d/p$. As we see later, this will be equal to $m/(1+\epsilon)$ for some $\epsilon > 0$ which is determined by the corruption threshold. This means that in each iteration of the CD algorithm running on the modified encoded problem, we will be effectively updating the coordinates of the parameter vector $\mathbf{w}$ in chunks of size $m/(1+\epsilon)$ or its integer multiples. In particular, if each worker i updates k coordinates of $\mathbf{v}_i$, then $km/(1+\epsilon)$ coordinates of $\mathbf{w}$ will get updated. For comparison, Algorithm 1 updates km coordinates of the parameter vector $\mathbf{w}$ in each iteration, if each worker updates k coordinates in that iteration.*

Now we design an encoding matrix $\mathbf{R}^+$ and a decoding method that satisfy **P.1** and **P.2**.

5.1 Encoding and Decoding

In this section, we first design an encoding matrix $\mathbf{R}^+$ that satisfies **P.1**. $\mathbf{R}^+$ will be such that it has orthonormal rows, so, $\mathbf{R}$ is easy to compute, $\mathbf{R} = (\mathbf{R}^+)^T$. For simplicity, we denote $\mathbf{R}^+$ by $\mathbf{S}$. We show that the encoding matrix that we design for the MV multiplication in Section 4 satisfies all the properties that we want.[13] In the MV multiplication, we had a fixed matrix $\mathbf{A}$ and the master node

[13] The encoding and decoding of this section is based on the corresponding algorithms from Section 4.

wants to compute $\mathbf{Aw}$ for any vector $\mathbf{w}$ of its choice. In the solution presented in Section 4, we encode $\mathbf{A}$ and store $\mathbf{S}_i\mathbf{A}$ at the i'th worker node. Now, the master sends $\mathbf{w}$ to all the worker nodes, and each worker i responds with $\mathbf{S}_i\mathbf{Aw}+\mathbf{e}_i$, where $\mathbf{e}_i=\mathbf{0}$ if worker i is honest, otherwise can be arbitrary. Once master receives $\{\mathbf{S}_i\mathbf{Aw}+\mathbf{e}_i\}_{i=1}^m$, it can run the error correcting procedure to recover $\mathbf{Aw}$. To apply this in our setting, we take $\mathbf{A}$ to be the identity matrix, such that $\mathbf{S}_i\mathbf{A}=\mathbf{S}_i$, and the master can recover $\mathbf{w}$ from $\{\mathbf{r}_i=\mathbf{S}_i\mathbf{w}+\mathbf{e}_i\}_{i=1}^m$, if at most t of the $\mathbf{e}_i$'s are non-zero. For convenience, we rewrite the encoding matrix $\mathbf{S}_i$ for the i'th worker node from Section 4.2 below:

$$\mathbf{S}_i=\begin{bmatrix} b_{1i}\dots b_{qi} & & & \\ & \ddots & & \\ & & b_{1i}\dots b_{qi} & \\ & & & b_{1i}\dots b_{li}\end{bmatrix}_{p\times d} \tag{20}$$

Here $q=(m-2t)$ and $l=d-(p-1)q$, where $p=\lceil\frac{d}{q}\rceil$. Note that $1\le l<q$, and if q divides d, then $l=q$. All the unspecified entries of $\mathbf{S}_i$ are zero. By stacking up the $\mathbf{S}_i$'s gives us our desired encoding matrix $\mathbf{S}=[\mathbf{S}_1^T\ \mathbf{S}_2^T\ \dots\ \mathbf{S}_m^T]^T$. Note that $b_{1i},b_{2i},\dots,b_{qi}$ are such that if we let $\mathbf{b}_i=[b_{i1}\ b_{i2}\dots b_{im}]^T$ for every $i\in[q]$, then $\{\mathbf{b}_1,\mathbf{b}_2,\dots,\mathbf{b}_q\}$ is a set of orthonormal vectors. This implies that $\mathbf{S}$ is orthonormal, and, therefore, $\mathbf{S}^+=\mathbf{S}^T$. By taking $\mathbf{R}=\mathbf{S}^T$, we have $\mathbf{R}^+=\mathbf{S}$. Now we show that $\mathbf{S}$ satisfies **P.1**-**P.2**.

Our Encoding Satisfies P.1. We need to show a map $f:[p]\to\mathcal{P}([d])$ that satisfies **P.1**. Let us define the function f as follows, where $(q=m-2t)$ and $p=\lceil\frac{d}{q}\rceil$:

$$f(i):=\begin{cases}[(i-1)*q+1:i*q] & \text{if } 1\le i<p,\\ [(p-1)*q+1:d] & \text{if } i=p,\end{cases} \tag{21}$$

and for any $\mathcal{U}\subseteq[p]$, we define $f(\mathcal{U}):=\cup_{i\in\mathcal{U}}f(i)$. It is clear from the definition of f that (i) $\{f(i):i\in[p]\}$ partitions $[d]$; (ii) for every $i\in[p-1]$ we have $|f(i)|=q$, and that $|f(p)|\le q$. Recall that $q=m-2t$. For the 3rd property, note that, for any $\mathcal{U}\subseteq[p]$, all the columns of $\mathbf{S}_{i\mathcal{U}}$ whose indices belong to $[d]\setminus f(\mathcal{U})$ are identically zero, which implies that we have

$$\mathbf{S}_{i\mathcal{U}}\mathbf{w}=\mathbf{S}_{i\mathcal{U},f(\mathcal{U})}\mathbf{w}_{f(\mathcal{U})},\qquad\text{for every } \mathbf{w}\in\mathbb{R}^d, \tag{22}$$

which in turn implies that

$$\mathbf{S}_{i\mathcal{U}}\mathbf{X}^T=\mathbf{S}_{i\mathcal{U},f(\mathcal{U})}\mathbf{X}^T_{f(\mathcal{U})}. \tag{23}$$

Since $\mathbf{S}^+=\mathbf{S}^T$, we have $\mathbf{S}^+_{i\mathcal{U}}=\mathbf{S}^T_{i\mathcal{U}}$ for every $i\in[m]$ and every $\mathcal{U}\subseteq[p]$. With these, our update rule $\mathbf{v}_{i\mathcal{U}}^{t+1}=\mathbf{S}_{i\mathcal{U}}\mathbf{w}^t-\alpha_t\mathbf{S}_{i\mathcal{U}}\mathbf{X}^T\phi'(\mathbf{Xw}^t)$[14] can equivalently be written as

$$\mathbf{v}_{i\mathcal{U}}^{t+1}=\mathbf{S}_{i\mathcal{U},f(\mathcal{U})}\mathbf{w}^{t+1}_{f(\mathcal{U})}, \tag{24}$$

where

$$\mathbf{w}^{t+1}_{f(\mathcal{U})}=\mathbf{w}^t_{f(\mathcal{U})}-\alpha_t\mathbf{X}^T_{f(\mathcal{U})}\phi'(\mathbf{Xw}^t). \tag{25}$$

Observe that (25) is the same update rule as (19), which implies that if each worker i updates $\mathbf{v}_{i\mathcal{U}}$ according to the CD update rule, then the collective update at all the worker nodes automatically updates $\mathbf{w}_{f(\mathcal{U})}$ according the CD update rule. Now we show that our invariant $\mathbf{v}^{t+1}=\mathbf{Sw}^{t+1}$ is maintained. We show this by induction. Base case $\mathbf{v}^0=\mathbf{Sw}^0$ holds by construction. For the inductive case, assume that $\mathbf{v}^t=\mathbf{Sw}^t$ holds at time t and we show $\mathbf{v}^{t+1}=\mathbf{Sw}^{t+1}$ holds at time $t+1$.

[14] We emphasize that we used $\mathbf{S}^+=\mathbf{S}^T$ crucially to equivalently write our update rule $\mathbf{v}_{i\mathcal{U}}^{t+1}=\mathbf{R}^+_{i\mathcal{U}}\mathbf{w}^t-\alpha\mathbf{R}^T_{i\mathcal{U}}\mathbf{X}^T\phi'(\mathbf{Xw}^t)$ from (18) as $\mathbf{v}_{i\mathcal{U}}^{t+1}=\mathbf{S}_{i\mathcal{U}}\mathbf{w}^t-\alpha_t\mathbf{S}_{i\mathcal{U}}\mathbf{X}^T\phi'(\mathbf{Xw}^t)$. This follows because $\mathbf{S}^+=\mathbf{S}^T$ and we take $\mathbf{R}^+=\mathbf{S}$, which together imply that $\mathbf{R}^+_{i\mathcal{U}}=\mathbf{R}^T_{i\mathcal{U}}=\mathbf{S}_{i\mathcal{U}}$.

Define $\overline{\mathcal{U}} := [p] \setminus \mathcal{U}$ and $\overline{f(\mathcal{U})} := [d] \setminus f(\mathcal{U})$. Since we did not update $\mathbf{v}^t_{i\overline{\mathcal{U}}}$'s, we have $\mathbf{v}^{t+1}_{i\overline{\mathcal{U}}} = \mathbf{v}^t_{i\overline{\mathcal{U}}}$ for every $i \in [m]$. This, together with the inductive hypothesis (i.e., $\mathbf{v}^t = \mathbf{S}\mathbf{w}^t$), implies that

$$\mathbf{v}^{t+1}_{i\overline{\mathcal{U}}} = \mathbf{S}_{i\overline{\mathcal{U}}}\mathbf{w}^t. \tag{26}$$

Since $f(\overline{\mathcal{U}}) = \overline{f(\mathcal{U})}$, we have from (22) that

$$\mathbf{S}_{i\overline{\mathcal{U}}}\mathbf{w}^t = \mathbf{S}_{i\overline{\mathcal{U}},\overline{f(\mathcal{U})}}\mathbf{w}^t_{\overline{f(\mathcal{U})}}. \tag{27}$$

It is clear from (25) that $\mathbf{w}^t_{\overline{f(\mathcal{U})}}$ did not get an update when we updated $\mathbf{v}^t_{i\mathcal{U}}$'s, which implies that $\mathbf{w}^{t+1}_{\overline{f(\mathcal{U})}} = \mathbf{w}^t_{\overline{f(\mathcal{U})}}$. Substituting this in (27) gives $\mathbf{S}_{i\overline{\mathcal{U}}}\mathbf{w}^t = \mathbf{S}_{i\overline{\mathcal{U}},\overline{f(\mathcal{U})}}\mathbf{w}^{t+1}_{\overline{f(\mathcal{U})}}$, which, by (22), yields $\mathbf{S}_{i\overline{\mathcal{U}}}\mathbf{w}^t = \mathbf{S}_{i\overline{\mathcal{U}}}\mathbf{w}^{t+1}$. This, together with (26), implies

$$\mathbf{v}^{t+1}_{i\overline{\mathcal{U}}} = \mathbf{S}_{i\overline{\mathcal{U}}}\mathbf{w}^{t+1}. \tag{28}$$

We already have from (22) and (24) that

$$\mathbf{v}^{t+1}_{i\mathcal{U}} = \mathbf{S}_{i\mathcal{U}}\mathbf{w}^{t+1}. \tag{29}$$

Since (28) and (29) hold for every $i \in [m]$, we have $\mathbf{v}^{t+1} = \mathbf{S}\mathbf{w}^{t+1}$. Hence, the invariant is maintained.

Our Encoding Satisfies P.2. If we let

$$\mathbf{v}_{[m]\mathcal{U}} := [\mathbf{v}^T_{1\mathcal{U}}\ \mathbf{v}^T_{2\mathcal{U}} \dots \mathbf{v}^T_{m\mathcal{U}}]^T,$$
$$\mathbf{S}_{[m]\mathcal{U},f(\mathcal{U})} := [\mathbf{S}^T_{1\mathcal{U},f(\mathcal{U})}\ \mathbf{S}^T_{2\mathcal{U},f(\mathcal{U})} \dots \mathbf{S}^T_{m\mathcal{U},f(\mathcal{U})}]^T,$$

then the collective update (24) from all the workers can be written as

$$\mathbf{v}^{t+1}_{[m]\mathcal{U}} = \mathbf{S}_{[m]\mathcal{U},f(\mathcal{U})}\mathbf{w}^{t+1}_{f(\mathcal{U})}. \tag{30}$$

It is easy to verify that for every choice of $\mathcal{U} \subseteq [p]$, $\mathbf{S}_{[m]\mathcal{U},f(\mathcal{U})}$ is a full column-rank matrix, which implies that we can in principle recover the updated $\mathbf{w}^{t+1}_{f(\mathcal{U})}$ from $\mathbf{v}^{t+1}_{[m]\mathcal{U}} = \mathbf{S}_{[m]\mathcal{U},f(\mathcal{U})}\mathbf{w}^{t+1}_{f(\mathcal{U})}$. Now we show that not only can we recover $\mathbf{w}^{t+1}_{f(\mathcal{U})}$ from $\{\mathbf{S}_{i\mathcal{U},f(\mathcal{U})}\mathbf{w}^{t+1}_{f(\mathcal{U})}\}_{i=1}^m$, but also efficiently recover $\mathbf{w}^{t+1}_{f(\mathcal{U})}$ from the *erroneous* vectors $\{\mathbf{S}_{i\mathcal{U},f(\mathcal{U})}\mathbf{w}^{t+1}_{f(\mathcal{U})} + \mathbf{e}_{i\mathcal{U}}\}_{i=1}^m$, where at most t out of m error vectors $\{\mathbf{e}_{i\mathcal{U}}\}_{i=1}^m$ are non-zero and may have arbitrary entries. Let $\mathcal{U} = \{j_1, j_2, \dots, j_{|\mathcal{U}|}\}$, and for every $i \in [m]$, let $\mathbf{e}_{i\mathcal{U}} = [e_{ij_1}e_{ij_2}\dots e_{ij_{|\mathcal{U}|}}]^T$. Master equivalently writes $\{\mathbf{S}_{i\mathcal{U},f(\mathcal{U})}\mathbf{w}^{t+1}_{f(\mathcal{U})} + \mathbf{e}_{i\mathcal{U}}\}_{i=1}^m$ as $|\mathcal{U}|$ systems of linear equations.

$$\tilde{h}_i(\mathbf{w}^{t+1}_{f(\mathcal{U})}) = \tilde{\mathbf{S}}_{i,f(\mathcal{U})}\mathbf{w}^{t+1}_{f(\mathcal{U})} + \tilde{\mathbf{e}}_i, \quad i \in \mathcal{U}, \tag{31}$$

where, for every $i \in \mathcal{U}$, $\tilde{\mathbf{e}}_i = [e_{1i}, e_{2i}, \dots, e_{mi}]^T$ and $\tilde{\mathbf{S}}_{i,f(\mathcal{U})}$ is an $m \times |f(\mathcal{U})|$ matrix whose j'th row is equal to the i'th row of $\mathbf{S}_{j\mathcal{U}}$, for every $j \in [m]$. Note that at most t entries in each $\tilde{\mathbf{e}}_i$ are non-zero. Observe that $\{\mathbf{S}_{i\mathcal{U},f(\mathcal{U})}\mathbf{w}^{t+1}_{f(\mathcal{U})} + \mathbf{e}_{i\mathcal{U}}\}_{i=1}^m$ and $\{\tilde{\mathbf{S}}_{i,f(\mathcal{U})}\mathbf{w}^{t+1}_{f(\mathcal{U})} + \tilde{\mathbf{e}}_i\}_{i\in\mathcal{U}}$ are equivalent systems of linear equations, and we can get one from the other. Observe that (31) is similar to (8): $\tilde{\mathbf{S}}_{i,f(\mathcal{U})}$ is equal to $\tilde{\mathbf{S}}_i$ (for the same i) with some of its zero columns removed; and adding zero columns to $\tilde{\mathbf{S}}_{i,f(\mathcal{U})}$ will not change the value of $\tilde{h}_i(\mathbf{w}^{t+1}_{f(\mathcal{U})})$. Now, using the machinery developed in Section 4 we can recover $\mathbf{w}^{t+1}_{f(\mathcal{U})}$ from (31) in $O(|\mathcal{U}|m^2)$ time.

5.2 Resource Requirement Analysis

In this section, first we give our algorithm developed for distributed coordinate descent in the presence of t (out of m) adversarial worker nodes, whose pictorial description is given in Figure 2.

We use two encoding matrices $\mathbf{L} \in \mathbb{R}^{(p_1 m)\times n}$ and $\mathbf{R} \in \mathbb{R}^{d\times(p_2 m)}$. Let $\mathbf{L} = [\mathbf{L}_1^T\ \mathbf{L}_2^T\ \dots\ \mathbf{L}_m^T]^T$ and $\mathbf{R} = [\mathbf{R}_1\ \mathbf{R}_2\ \dots\ \mathbf{R}_m]$, where each $\mathbf{L}_i$ is a $p_1 \times n$ matrix with $p_1 = \lceil \frac{n}{m-2t} \rceil$ and each $\mathbf{R}_i$ is a $d \times p_2$ matrix with $p_2 = \lceil \frac{d}{m-2t} \rceil$. Worker i stores both $\widetilde{\mathbf{X}}^L_i = \mathbf{L}_i\mathbf{X}$ and $\widetilde{\mathbf{X}}^R_i = \mathbf{X}\mathbf{R}_i$. Roughly, $\mathbf{L}$ is used to recover $\mathbf{X}\mathbf{w}$ from the erroneous $\{\mathbf{L}_i\mathbf{X}\mathbf{w} + \mathbf{e}_i\}_{i=1}^m$, and $\mathbf{R}$ is used to update the parameter vector reliably despite errors. Here $\mathbf{L}$ is a full column-rank matrix and $\mathbf{R}$ is a full row-rank matrix. Initialize with an arbitrary $\mathbf{w}^0$ and let $\mathbf{v}^0 = \mathbf{R}^+\mathbf{w}^0$. Repeat the following until convergence:

1. At iteration t, master sends $(\mathbf{w}^{t-1}_{f(\mathcal{U})} - \mathbf{w}^{t}_{f(\mathcal{U})})$[15] to all the workers (at $t = 0$, master sends $\mathbf{w}_0$), where $\mathcal{U} \subseteq [p_2]$ is the set of indices used for updating $\mathbf{v}^{t-1}_{i\mathcal{U}}$'s in the previous iteration, which in turn updated $\mathbf{w}^{t-1}_{f(\mathcal{U})}$; see (24) and (25) in Section 5.1.

2. Worker i computes $\widetilde{\mathbf{X}}^{L}_{i}(\mathbf{w}^{t-1}_{f(\mathcal{U})} - \mathbf{w}^{t}_{f(\mathcal{U})}) = \mathbf{L}_i\mathbf{X}(\mathbf{w}^{t-1}_{f(\mathcal{U})} - \mathbf{w}^{t}_{f(\mathcal{U})})$ and sends it to the master.[16] Upon receiving $\{\widetilde{\mathbf{X}}^{L}_{i}(\mathbf{w}^{t-1}_{f(\mathcal{U})} - \mathbf{w}^{t}_{f(\mathcal{U})}) + \mathbf{e}_i\}_{i=1}^{m}$, where at most t of the $\mathbf{e}_i$'s are non-zero and may have arbitrary entries, the master applies the decoding procedure of Section 4 and recovers $\mathbf{X}(\mathbf{w}^{t-1}_{f(\mathcal{U})} - \mathbf{w}^{t}_{f(\mathcal{U})})$. We assume that master keeps $\mathbf{X}\mathbf{w}^{t-1}$ from the previous iteration (which is equal to $\mathbf{0}$ if $t = 0$), it can compute $\mathbf{X}\mathbf{w}^{t} = \mathbf{X}\mathbf{w}^{t-1} - \mathbf{X}(\mathbf{w}^{t-1}_{f(\mathcal{U})} - \mathbf{w}^{t}_{f(\mathcal{U})})$. Note that if $t = 0$, this is equal to $\mathbf{X}\mathbf{w}^{0}$.

3. After obtaining $\mathbf{X}\mathbf{w}^{t}$, master computes $\phi'(\mathbf{X}\mathbf{w}^{t})$, picks a subset $\mathcal{U} \subseteq [p_2]$ (randomly or in a round robin fashion to cover $[p_2]$ in a few iterations), and sends $(\phi'(\mathbf{X}\mathbf{w}^{t}), \mathcal{U})$ to all the workers.

4. Each worker node $i \in [m]$ updates $\mathbf{v}^{t+1}_{i\mathcal{U}} \leftarrow \mathbf{v}^{t}_{i\mathcal{U}} - \alpha_t \nabla_{i\mathcal{U}}\phi(\widetilde{\mathbf{X}}\mathbf{v}^{t}) = \mathbf{v}^{t}_{i\mathcal{U}} - \alpha_t(\widetilde{\mathbf{X}}^{R}_{i\mathcal{U}})^{T}\phi'(\mathbf{X}\mathbf{w}^{t})$, while keeping the other coordinates of $\mathbf{v}^{t}_{i}$ unchanged. Worker i sends $\mathbf{v}^{t+1}_{i\mathcal{U}}$ to the master. Note that $\mathbf{v}^{t+1}_{i\mathcal{U}} = \mathbf{R}^{+}_{i\mathcal{U},f(\mathcal{U})}\mathbf{w}^{t+1}_{f(\mathcal{U})}$, where $\mathbf{w}^{t+1}_{f(\mathcal{U})} = [\mathbf{w}^{t}_{f(\mathcal{U})} - \alpha\mathbf{X}^{T}_{f(\mathcal{U})}\phi'(\mathbf{X}\mathbf{w}^{t})]$; see (24) and (25) in Section 5.1.

5. Upon receiving $\{\mathbf{v}^{t+1}_{i\mathcal{U}} + \mathbf{e}_{i\mathcal{U}}\}_{i=1}^{m}$, where at most t of the $\{\mathbf{e}_{i\mathcal{U}}\}_{i=1}^{m}$'s are non-zero and may have arbitrary entries, master applies the decoding procedure (since our encoding satisfies **P.2**) and recovers $\mathbf{w}^{t+1}_{f(\mathcal{U})}$.

Now we analyze the total amount of resources (storage, computation, and communication) required by the above algorithm and prove Theorem 2. Fix an $\epsilon > 0$. Let the corruption threshold t satisfy $t \leq \lfloor(\epsilon/(1+\epsilon)) \cdot (m/2)\rfloor$.

5.2.1 Storage Requirement:

By a similar analysis done in Section 4.5, we can show that the total storage at all worker nodes is roughly equal to $2(1+\epsilon)|\mathbf{X}|$.

5.2.2 Computational Complexity:

We can divide the computational complexity of our scheme as follows:

- *Encoding the data matrix.* By a similar analysis done in Section 4.5, we can show that the total encoding time is $O\left((\frac{\epsilon}{1+\epsilon}m + 1)nd\right)$. Note that this encoding is to be done only once.

- *Computation at each worker node.* Suppose that in each iteration of our algorithm, all the workers update τ coordinates of $\mathbf{v}_i$'s. Fix an iteration t and assume that at iteration $(t-1)$, workers updated the coordinates in the set $\mathcal{U} \subseteq [p_2]$, where $|\mathcal{U}| = \tau$. Recall from **P.1** that updating $\tau = |\mathcal{U}|$ coordinates of each $\mathbf{v}^{t-1}_{i}$ automatically updates $\mathbf{w}^{t-1}_{f(\mathcal{U})}$. Upon receiving $(\mathbf{w}^{t-1}_{f(\mathcal{U})} - \mathbf{w}^{t}_{f(\mathcal{U})})$ from the master node, each worker i computes $\widetilde{\mathbf{X}}^{\mathbf{L}}_{i}(\mathbf{w}^{t-1}_{f(\mathcal{U})} - \mathbf{w}^{t}_{f(\mathcal{U})})$, and reports back the resulting vector. Note that $(\mathbf{w}^{t-1}_{f(\mathcal{U})} - \mathbf{w}^{t}_{f(\mathcal{U})})$ has at most $|f(\mathcal{U})| = \frac{\tau m}{1+\epsilon}$ non-zero elements, which together with that $\widetilde{\mathbf{X}}^{\mathbf{L}}_{i}$ is a $p_1 \times d$ matrix, implies that computing $\widetilde{\mathbf{X}}^{\mathbf{L}}_{i}(\mathbf{w}^{t-1}_{f(\mathcal{U})} - \mathbf{w}^{t}_{f(\mathcal{U})})$ takes $O(p_1 \cdot |f(\mathcal{U})|) = O(n\tau)$ time.[17] In the second round, given $\phi'(\mathbf{X}\mathbf{w}^{t})$, since $(\widetilde{\mathbf{X}}^{R}_{i\mathcal{U}})^{T}$ is of dimension $n \times \tau$, updating $\mathbf{v}^{t}_{i\mathcal{U}}$ requires $O(n\tau)$ time, where $\tau = |\mathcal{U}|$. So, the total time taken by each worker is $O(n\tau)$.

[15] Observe that master need not send the locations $f(\mathcal{U})$, because workers can compute those by themselves, as they know both $\mathcal{U}$ and the function f.

[16] With some abuse of notation, when we write $\mathbf{X}\mathbf{w}_{f(\mathcal{U})}$, we implicitly assume that $\mathbf{w}_{f(\mathcal{U})}$ is a length d vector, which has 0's in indices $\overline{f(\mathcal{U})}$.

[17] Note that in the very first iteration, master sends $\mathbf{w}^{0}$, which may be a dense length d vector, and computing $\widetilde{\mathbf{X}}_i\mathbf{L}\mathbf{w}^{0}$ at the i'th worker can take $O(p_1 d) = O((1+\epsilon)\frac{nd}{m})$ time. This is only for the first iteration.

- *Computation at the master node.* Once master receives $\{\mathbf{L}_i\mathbf{X}(\mathbf{w}^{t-1}_{f(\mathcal{U})} - \mathbf{w}^t_{f(\mathcal{U})}) + \mathbf{e}_i\}_{i=1}^m$, applying the decoding procedure of Section 4 to obtain $\mathbf{X}(\mathbf{w}^{t-1}_{f(\mathcal{U})} - \mathbf{w}^t_{f(\mathcal{U})})$ from these erroneous vectors requires $O(p_1 m^2) = O((1+\epsilon)nm)$ time. After that obtaining $\mathbf{X}\mathbf{w}^t$ takes another $O(n)$ time. Given $\mathbf{X}\mathbf{w}^t$, computing $\phi'(\mathbf{X}\mathbf{w}^t)$ takes $O(n)$ time, assuming that computing $\ell'(\langle \mathbf{x}_i, \mathbf{w}^t\rangle; y_i)$ requires unit time, where $\langle \mathbf{x}_i, \mathbf{w}^t\rangle$ is equal to the i'th entry of $\mathbf{X}\mathbf{w}^t$. Upon receiving $\{\mathbf{v}^{t+1}_{i\mathcal{U}} + \mathbf{e}_{i\mathcal{U}}\}_{i=1}^m$, where $\mathbf{v}^{t+1}_{i\mathcal{U}} = \mathbf{R}^+_{i\mathcal{U},f(\mathcal{U})}\mathbf{w}^{t+1}_{f(\mathcal{U})}$, for all $i \in [m]$, recovering $\mathbf{w}^{t+1}_{f(\mathcal{U})}$ requires $O(\tau m^2)$ time. So, the total time taken by the master node is $O((1+\epsilon)nm + \tau m^2)$.

5.3 Communication Complexity:

Suppose workers update τ coordinates of $\mathbf{v}_i$'s in each iteration. Then (i) master broadcasts $\left(\frac{\tau m}{1+\epsilon} + n\right)$ real numbers, $\frac{\tau m}{1+\epsilon}$ in the first round to represent $\mathbf{w}^t_{f(\mathcal{U})}$ and n in the second round to represent $\phi'(\mathbf{X}\mathbf{w}^t)$; and (ii) each worker sends $\left(\tau + (1+\epsilon)\frac{n}{m}\right)$ real numbers, $(1+\epsilon)\frac{n}{m}$ in the first round for computing $\mathbf{X}\mathbf{w}^t$ at the master node and τ in the second iteration to represent $\mathbf{v}^t_{i\mathcal{U}}$.

6 Extensions

In this section, we give a few important extensions of our coding scheme developed earlier in Section 4. First we give a Byzantine-resilient and communication-efficient method for stochastic gradient descent (SGD). Second we show how to exploit the specific structure in our encoding matrix to efficiently extend our coding technique to the streaming data model. In the end, we give a few more important applications, where our method can be applied constructively.

6.1 Stochastic Gradient Descent

Stochastic gradient descent (SGD) [HM51] is another alternative if full gradients are too costly to compute. In each iteration of SGD, we sample a data point uniformly at random, compute a gradient on that sample, and update the parameter vector based on that. We start with an arbitrary/random parameter vector $\mathbf{w}_0$ and update it according the following update rule:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha_t \nabla f_{r_t}(\mathbf{w}_t), \quad t = 1, 2, 3, \ldots \tag{32}$$

where r_t is sampled uniformly at random from $\{1, 2, \ldots, n\}$. This ensures that the expected value of the gradient is equal to the true gradient. Due to its simplicity and remarkable empirical performance, SGD has become arguably the most widely-used optimization algorithm in many large-scale applications, especially in deep learning [Bot10, RSS12, DCM+12]. We want to run SGD in a distributed setup, where data is distributed among m worker nodes and at most t of them can be corrupt; see Section 2.3 for details on our adversary model.

Our solution. In the plain SGD, we sample a data point randomly and compute its gradient. So, we give a method in which, at any iteration t, master picks a random number r_t in $\{1, 2, \ldots, n\}$, broadcasts it, and recovers the r_t'th data point $\mathbf{x}_{r_t}$. Once the master has obtained $\mathbf{x}_{r_t}$, it can compute a gradient on it and updates the parameter vector. Since master recovers the data points, we can optimize for non-convex problems also; essentially, we could optimize anything that the plain SGD can. Our method is described below.

We encode $\mathbf{X}^T$ using the $\lceil d/(m-2t)\rceil \times d$ encoding matrix $\mathbf{S}^{(2)}$, which has been defined in Section 4.5. For simplicity, we denote $\mathbf{S}^{(2)}$ by $\mathbf{S}$. Let $\mathbf{S} = [\mathbf{S}_1^T\ \mathbf{S}_2^T\ \ldots\ \mathbf{S}_m^T]^T$. Note that the j'th worker stores $\mathbf{S}_j\mathbf{X}^T$. Let $\widetilde{\mathbf{X}} := \mathbf{S}\mathbf{X}^T$, which is a $\lceil d/(m-2t)\rceil \times n$ matrix, whose i'th column is the encoding $\widetilde{\mathbf{x}}_i := \mathbf{S}\mathbf{x}_i$ of the i'th data point $\mathbf{x}_i$. Using the recipe developed in Section 4, given $\{\mathbf{S}_j\mathbf{x}_i + \mathbf{e}_j\}_{j=1}^m$, where $\mathbf{e}_j = \mathbf{0}$ if the j'th worker is honest, otherwise can be arbitrary, master can recover $\mathbf{x}_i$ exactly in $O((1+\epsilon)md)$ time. Our main theorem is stated below, a proof of which trivially follows from the recipe of Section 4.

Theorem 3 (Stochastic Gradient Descent)**.** *Let $\mathbf{X} \in \mathbb{R}^{n\times d}$ denote the data matrix. Let m denote the total number of worker nodes. We can compute a stochastic gradient in a distributed manner in the*

presence of t corrupt worker nodes and s stragglers, with the following guarantees, where $\epsilon > 0$ is a free parameter.

- $(s+t) \leq \left\lfloor \frac{\epsilon}{1+\epsilon} \cdot \frac{m}{2} \right\rfloor$.
- *Total storage requirement is roughly* $(1+\epsilon)|\mathbf{X}|$.
- *Computational complexity for each stochastic gradient computation:*
  - *at each worker node is* $O((1+\epsilon)\frac{d}{m})$.
  - *at the master node is* $O((1+\epsilon)dm)$.
- *Communication complexity for each stochastic gradient computation:*
  - *each worker sends* $\left((1+\epsilon)\frac{d}{m}\right)$ *real numbers.*
  - *master broadcasts* $\lceil \log n \rceil$ bits.
- *Total encoding time is* $O\left(nd\left(\frac{\epsilon}{1+\epsilon}m+1\right)\right)$.

Unlike the two-round approach taken for gradient computation and coordinate descent, we give a one-round approach for each iteration of SGD. Observe the distributed gain of our method in the communication exchanged between the workers and the master: (i) master only broadcasts an index in $\{1, 2, \ldots, n\}$, which only takes $\lceil \log n \rceil$ *bits*; and (ii) each worker sends roughly $\frac{1+\epsilon}{m}$ fraction of the total dimension d. Hence, this method is mostly useful in the distributed setting with communication-constrained and band-limited links. The Remark 3, 4, 5 are also applicable for Theorem 3.

6.2 Encoding in the Streaming Data Model

An attractive property of our encoding scheme is that it is very easy to update with new data points. More specifically, our encoding requires the same amount of time, irrespective of whether we get all the data at once, or we get each sample point one by one, as in the online/streaming model. This setting encompasses a more realistic scenario, in which we design our coding scheme with the initial set of data points and distribute the encoded data among the workers. Later on, when we get some more samples, we can easily incorporate them into our existing encoded data. We show that updating $(m-2t)$ new data points in $\mathbb{R}^d$ requires $O\left((m-2t)\left((2t+1)d\right)\right)$ time in total, i.e., $O\left((2t+1)d\right)$ amortized-time per data point. This is the best one can hope for, since the offline encoding of n data points requires $O\left((2t+1)nd\right)$ total time. At the end of the update, the final encoded matrix that we get is the same as the one we would have got had we had all the $n+1$ data points in the beginning. Therefore, the decoding is not affected by this method at all. Note that we use the same encoding matrices both for gradient computation as well as for coordinate descent. So, it suffices to prove our result in the streaming model for any one of them, and we show it for gradient computation below.

Theorem 4. *The total time complexity in encoding all the data points at once, i.e., when encoding is done offline, is the same as the total time complexity in encoding the data points one by one as they come in the streaming model, i.e., when encoding is done online.*

Proof. Let $\mathbf{S}^{(1)}$ and $\mathbf{S}^{(2)}$ denote the encoding matrices for encoding $\mathbf{X}$ and $\mathbf{X}^T$, respectively; see Section 4.2. For convenience, we copy over the corresponding encoding matrices $\mathbf{S}_i^{(1)}$ and $\mathbf{S}_i^{(2)}$ from (11) for the i'th worker node in Figure 3.

Suppose at some point of time we have encoded n data points each lying in $\mathbb{R}^d$ and distributed the encoded data among the m worker nodes. Now a new data sample $\mathbf{x} \in \mathbb{R}^d$ comes in. We will show how to incorporate it in the existing scheme in $O\left((2t+1)d\right)$ time on average.

Updating the encoding matrices. Fix an arbitrary worker $i \in [m]$. Note that the new data matrix $\mathbf{X}$ has dimension $(n+1) \times d$. So, the new encoding matrix $\mathbf{S}_i^{(1)}$ should have $(n+1)$ columns, and we have to add one more column to $\mathbf{S}_i^{(1)}$. By examining the repetitive structure of $\mathbf{S}_i^{(1)}$, it is obvious which column to add: if $l_1 < q$, then we add the p_1-dimensional vector $[0, 0, \ldots, 0, b_{(l_1+1)i}]^T$ as

$$\mathbf{S}_i^{(1)} = \begin{bmatrix} b_{1i} \dots b_{qi} & & & \\ & \ddots & & \\ & & b_{1i} \dots b_{qi} & \\ & & & b_{1i} \dots b_{l_1 i} \end{bmatrix}_{p_1 \times n} \qquad \mathbf{S}_i^{(2)} = \begin{bmatrix} b_{1i} \dots b_{qi} & & & \\ & \ddots & & \\ & & b_{1i} \dots b_{qi} & \\ & & & b_{1i} \dots b_{l_2 i} \end{bmatrix}_{p_2 \times d}$$

(a) **(b)**

Figure 3 Figure 3a depicts the encoding matrix for the i'th worker node for encoding $\mathbf{X}$, which is used in the first round of the gradient computation. Here $p_1 = \lceil n/q \rceil$, where $q = (m-k)$ and k is equal to the number of rows in the error recovery matrix $\mathbf{F}$ in (14), and $l_1 = n - (p_1 - 1)q$. Figure 3b depicts the encoding matrix for the i'th worker node for encoding $\mathbf{X}^T$, which is used in the second round of the gradient computation. Here $p_2 = \lceil d/q \rceil$ and $l_2 = d - (p_2 - 1)q$. All the unspecified entries in both the matrices are zero.

the last column; otherwise, if $l_1 = q$, then we add the (p_1+1)-dimensional vector $[0, 0, \dots, 0, b_{1i}]^T$ as the last column. In the second case, the number of rows of $\mathbf{S}_i^{(1)}$ increases by one – the last row has all zeros, except for the last element, which is equal to b_{1i}. Note that $\mathbf{S}_i^{(2)}$ does not change at all. Observe that if the i'th worker performs this update, then it does not have to store its entire encoding matrix $\mathbf{S}_i^{(1)}$, it only needs to store n, $q = (m-k)$, and the q real numbers $b_{1i}, b_{2i}, \dots, b_{qi}$, where $q = m-k$, which could be much smaller as compared to n and d, and are enough to define $\mathbf{S}_i^{(1)}$ and $\mathbf{S}_i^{(2)}$.

Updating the encoded data. Now we show how to update the encoded data with the new sample $\mathbf{x}$. We need to update both $\mathbf{S}_i^{(1)}\mathbf{X}$ as well as $\mathbf{S}_i^{(2)}\mathbf{X}^T$ for every worker $i \in [m]$.

- **Updating $\mathbf{S}_i^{(1)}\mathbf{X}$.** If $l_1 < q$, then we add $b_{(l_1+1)i}\mathbf{x}^T$ to the last row of $\mathbf{S}_i^{(1)}\mathbf{X}$; otherwise, if $l_1 = q$, then we add $b_{1i}\mathbf{x}$ as a new row in $\mathbf{S}_i^{(1)}\mathbf{X}$. In the first case, the resulting matrix still has p_1 rows, whose first $p_1 - 1$ rows are same as before, and the last row is the sum of the previous row and $b_{(l_1+1)i}\mathbf{x}^T$. In the second case, the resulting matrix has (p_1+1) rows, whose first p_1 rows are the same as before and the last row is equal to $b_{1i}\mathbf{x}^T$. Note that each row of $\mathbf{S}_i^{(1)}$ for $i \le 2t$, has at most $(m-2t)$ non-zero elements; whereas, for $i > 2t$, each row of $\mathbf{S}_i^{(1)}$ has exactly one non-zero entry. Since there are $p_1 = \lceil n/(m-2t) \rceil$ rows in each $\mathbf{S}_i^{(1)}$, updating $\mathbf{S}_i^{(1)}\mathbf{X}$ for every $i \le 2t$ takes $O(d)$ time; and for $i > 2t$, update in $\mathbf{S}_i^{(1)}\mathbf{X}$ happens only once in $(m-2t)$ new data points (whenever the second case occurs and the resulting $\mathbf{S}_i^{(1)}$ has (p_1+1) rows). So, updating $(m-2t)$ data points at all m worker nodes require $O\left(2t * (m-2t)d + (m-2t) * d\right) = O((m-2t)(2t+1)d)$ time, i.e., $O\left((2t+1)d\right)$ time per data point.
- **Updating $\mathbf{S}_i^{(2)}\mathbf{X}^T$.** Note that $\mathbf{X}^T$ is a $d \times (n+1)$ matrix whose last column is equal to the new data sample $\mathbf{x}$. Now, to update $\mathbf{S}_i^{(2)}\mathbf{X}^T$, we add $\mathbf{S}_i^{(2)}\mathbf{x}$ as an extra column. The resulting matrix is of size $p_2 \times (n+1)$, whose first n columns are the same as before and the last column is equal to $\mathbf{S}_i^{(2)}\mathbf{x}$. Since total number of non-zero entries in $\mathbf{S}_i^{(2)}$ is equal to d if $i \le 2t$ and equal to $p_2 = \lceil d/(m-2t) \rceil$ if $i > 2t$, the total time required to update a new data point is $O(2t * d + (m-2t) * p_2) = O\left((2t+1)d\right)$.

Observe that at the end of this local update at each worker node, the final encoded matrix that we get is the same as the one we would have got had we had all the $n+1$ data points in the beginning. The decoding is not affected by this method at all. This completes the proof of Theorem 4. □

Remark 9 (Updating the encoded data efficiently with new features)**.** *Observe that since we encode both X and X^T in an analogous fashion, it follows by symmetry that we can not only update efficiently upon receiving a new data sample, but can also update efficiently if we decide to enlarge the dimension d of each of the n data samples at some point of time – maybe we figure out some new features of the data to get a more accurate model to overcome under-fitting. In these situations, we don't need to encode the entire dataset all over again, just a simple update is enough to incorporate the changes.*

Remark 10 (What allows our encoding to be efficient for streaming data?)**.** *The efficient update property of our coding scheme is made possible by the repetitive structure of our encoding matrix (see Figure 3), together with the fact that this structure is independent of the number of data points n and the dimension d – it only depends on the number of worker nodes m and the corruption threshold t. We remark that other data encoding methods in literature, even for weaker models, do not support efficient update. For example, the encoding of [KSDY17], which was designed for mitigating stragglers, depends on the dimensions n and d of the data matrix. So, it may not efficiently update if a new data point comes in.*

6.3 More Applications.

There are many iterative algorithms, other than the gradient descent for learning GLMs, which use repeated MV multiplication. Some of them include (i) the power method for computing the largest eigenvalue of a diagonalizable matrix, which is used in Google's PageRank algorithm [ISW06], Twitter's recommendation system [GGL+13], etc.; (ii) iterative methods for solving sparse linear systems [Saa03]; (iii) many graph algorithms, where the graph is represented by a fixed adjacency matrix, [KG11]. In large-scale implementation of these systems, where Byzantine faults are inevitable, the method described in this paper can be of interest.

In most of these applications, the underlying matrix $\mathbf{A}$ is generally sparse, which is exploited to gain computational efficiency. So, it is desired not to lose sparsity even if we want resiliency against Byzantine attacks. Fortunately, our encoding matrix $\mathbf{S}$ is sparse (see (11)), which ensures that the encoded matrix $\mathbf{SA}$ will not lose the sparsity of $\mathbf{A}$: Each of the first pk rows of $\mathbf{S}$ has at most $(m-k)$ (where $k = 2t$) non-zero elements, and each of the remaining rows has exactly one 1. Since m is the number of worker nodes, which is very small, and we can take t to be up to $\lfloor \frac{m-1}{2} \rfloor$, we have very few non-zero entries in each row of $\mathbf{S}$ (in the extreme case when $2t = m-1$, each row of $\mathbf{S}$ has only one non-zero entry). In a sense, we are getting Byzantine-resiliency almost for free without compromising the computational efficiency that is made possible by the sparsity of the matrix!

7 Numerical Experiments

In this section, we validate the efficacy of our proposed method by numerical experiments. We run distributed gradient descent for linear regression $\arg\min_{\mathbf{w}\in\mathbb{R}^d} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2$. Since we are computing the gradients exactly, (i) there is no need to check the convergence; and (ii) our algorithm will perform exactly the same whether we are working with synthetic dataset or the real dataset; hence, we will work with the synthetic dataset. We run our algorithm[18] with $m = 15$ worker nodes on two datasets: $(n = 10,000, d = 250)$ and $(n = 20,000, d = 22,000)$. For both the datasets, we generate $(\mathbf{X}, \mathbf{y})$ by sampling $\mathbf{X} \leftarrow \mathcal{N}(0, I)$ and $\mathbf{y} = \mathbf{X}\theta + \mathbf{z}$, where $\theta \in \mathbb{R}^d$ has $d/3$ non-zero entries, all of them are i.i.d. according to $\mathcal{N}(0, 4)$, and each entry of $\mathbf{z} \in \mathbb{R}^n$ is sampled from $\mathcal{N}(0, 1)$ i.i.d. In each round of the gradient computation, the adversary picks t worker nodes uniformly at random, and adds independent random vectors of appropriate length as errors, whose entries are sampled from $\mathcal{N}(0, \sigma^2)$ i.i.d. with $\sigma = 100$, to the true vectors.

$\mathbf{n = 10,000, d = 250, m = 15}$. In Figure 4, we plot the total time taken (which is the sum of the maximum time taken by any single worker node and the time taken by the master node in both the rounds) in one gradient computation, with varying number of corrupt worker nodes from $t = 1$ to $t = 7$. Note that, when $t = 7$, we have $\epsilon = m - 1$, which is the main cause behind the significant increment in time for $t = 7$.

$\mathbf{n = 20,000, d = 22,000, m = 15}$. In Figure 5, we report separately, the maximum time taken by any single worker node and the time taken by the master node (together in both the rounds) for one gradient computation, with varying number of corrupt worker nodes from $t = 1$ to $t = 6$. Observe that the time taken by the master node is orders of magnitude less than the time taken by the worker nodes.

[18] We implement our algorithm in Python, and run it on a machine with Intel(R) Core(TM) i5-33305 CPU @ 2.70GHz processor and 16 GB 1600 MHz DDR3 memory.

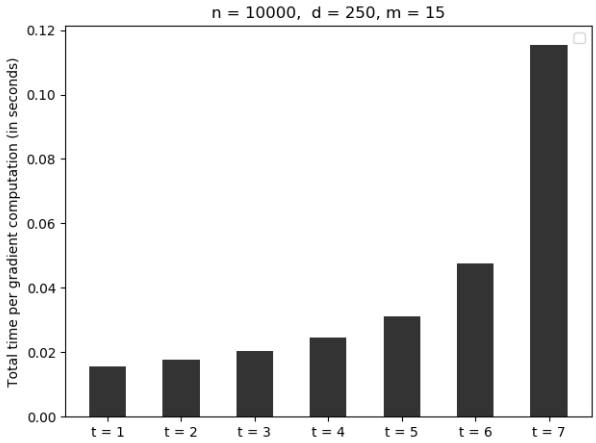

Figure 4 We run our algorithm with 15 worker nodes on a dataset with $n = 10,000, d = 250$. This plot reports how the total time taken (in seconds) in per gradient computation changes with varying number of corrupt worker nodes from $t = 1$ to $t = 7$.

	Worker	Master
$t = 1$	0.152	0.045
$t = 2$	1.241	0.054
$t = 3$	4.163	0.105
$t = 4$	9.887	0.679
$t = 5$	15.99	0.789
$t = 6$	30.258	1.286

Figure 5 We run our algorithm with 15 worker nodes on a dataset with $n = 20,000, d = 22,000$, and report the total time required per gradient computation against varying number of corrupt worker nodes from $t = 1$ to 6.

8 Conclusion

In this paper, we studied distributed optimization and learning in the master-worker architecture in the presence of a Byzantine adversary, who can corrupt up to t of the m worker machines, and the compromised machines can collaborate and arbitrarily deviate from their pre-specified programs. We focused on learning the *generalized linear models* (GLM) with regularization, and proposed a solution for *proximal gradient descent* (PGD) in the data-parallel framework and for *coordinate descent* (CD) in the model-parallel framework. Gradient descent (GD) is a special case of both of these algorithms.

Observing that applying general-purpose solutions to our problem may be inefficient (see the trivial approach on page 7), we exploit the problem structure and give a solution that is both generic and efficient, based on data encoding using sparse encoding matrices and real-error correction for decoding at the master node. We give a deterministic solution without making any stochastic assumption on the data. Our solution gives a trade-off between the corruption threshold and the computational/communication complexity & the storage required by our coding scheme. So, depending on the scenario, we can choose the parameters that work best. In particular, with a *constant* overhead on the computational/communication complexity and the storage requirement as compared to running the distributed PGD/CD (which do not provide any adversarial protection), our scheme can tolerate up to 1/3 of the corrupt worker nodes. We can tolerate up to $\lfloor \frac{m-1}{2} \rfloor$ corrupt worker nodes, which is information-theoretically optimal. In the case of gradient computation in GLMs, our method significantly improves upon a recent work [CWCP18] on the resource requirements, which essentially uses a repetition code (for gradient coding), incurring a factor of $(2t+1)$ in the storage/computation overhead at the worker nodes; see Section 3 for more details on the comparison. Since our encoding matrix is sparse, the encoding time complexity only incurs a factor of $(2t+1)$ more than what is required by just for distributing the raw data matrix among m worker nodes.

We extend our encoding scheme to also make *stochastic gradient descent* (SGD) robust to Byzantine adversary with similar guarantees as above. Since our encoding matrix is sparse, structured, and have

repetitive entries, we can apply our encoding in the data streaming setting too. We show that, in this setting, where data points come in one at a time (and we encode them as they arrive), our encoding is as efficient as when all the data is available offline and we encode them all in one go. We also give numerical results to show the efficacy of our method.

References

[A+18] Tarek F. Abdelzaher et al. Will distributed computing revolutionize peace? the emergence of battlefield iot. In *ICDCS 2018*, pages 1129–1138, 2018.

[AT08] Mehmet Akçakaya and Vahid Tarokh. A frame construction and a universal distortion bound for sparse representations. *IEEE Trans. Signal Processing*, 56(6):2443–2450, 2008.

[Bil95] P. Billingsley. *Probability and Measure*. Wiley Series in Probability and Statistics. Wiley, 1995.

[BKBG11] Joseph K. Bradley, Aapo Kyrola, Danny Bickson, and Carlos Guestrin. Parallel coordinate descent for l1-regularized loss minimization. In *ICML*, pages 321–328, 2011.

[BMGS17] Peva Blanchard, El Mahdi El Mhamdi, Rachid Guerraoui, and Julien Stainer. Machine learning with adversaries: Byzantine tolerant gradient descent. In *Advances in Neural Information Processing Systems, NIPS 2017, 4-9 December 2017, Long Beach, CA, USA*, pages 118–128, 2017.

[Bot10] L. Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010. Physica-Verlag HD*, 2010.

[BT89] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[BV04] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, New York, NY, USA, 2004.

[CP86] Thomas F. Coleman and Alex Pothen. The null space problem I. complexity. *SIAM Journal on Algebraic Discrete Methods*, 7(4):527–537, 1986.

[CP18] Zachary B. Charles and Dimitris S. Papailiopoulos. Gradient coding using the stochastic block model. In *2018 IEEE International Symposium on Information Theory, ISIT 2018, Vail, CO, USA, June 17-22, 2018*, pages 1998–2002, 2018.

[CSX17] Yudong Chen, Lili Su, and Jiaming Xu. Distributed statistical machine learning in adversarial settings: Byzantine gradient descent. *POMACS*, 1(2):44:1–44:25, 2017.

[CT05] Emmanuel J. Candès and Terence Tao. Decoding by linear programming. *IEEE Trans. Information Theory*, 51(12):4203–4215, 2005.

[CWCP18] Lingjiao Chen, Hongyi Wang, Zachary B. Charles, and Dimitris S. Papailiopoulos. DRACO: byzantine-resilient distributed training via redundant gradients. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, pages 902–911, 2018.

[DB13] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, February 2013.

[DCG16] Sanghamitra Dutta, Viveck R. Cadambe, and Pulkit Grover. Short-dot: Computing large linear transforms distributedly using coded short dot products. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 2092–2100, 2016.

[DCM+12] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew W. Senior, Paul A. Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems (NIPS)*, pages 1232–1240, 2012.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[GGL+13] Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Zadeh. Wtf: The who to follow service at twitter. In *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13, pages 505–514, New York, NY, USA, 2013. ACM.

[HK71] Kenneth M Hoffman and Ray Kunze. *Linear algebra*. Prentice-Hall, Englewood Cliffs, NJ, 1971.

[HM51] Robbins Herbert and Sutton Monro. A stochastic approximation method. *The Annals of Mathematical Statistics. JSTOR, www.jstor.org/stable/2236626.*, vol. 22, no. 3, pp. 400–407, 1951.

[HRSH18] Wael Halbawi, Navid Azizan Ruhi, Fariborz Salehi, and Babak Hassibi. Improving distributed gradient descent using reed-solomon codes. In *2018 IEEE International Symposium on Information Theory, ISIT 2018, Vail, CO, USA, June 17-22, 2018*, pages 2027–2031, 2018.

[ISW06] Ilse Ipsen and Rebecca S. Wills. Mathematical properties and analysis of google's pagerank. *Boletín de la Sociedad Española de Matemática Aplicada*, 34:191–196, 01 2006.

[Jag13] Martin Jaggi. An equivalence between the lasso and support vector machines. *CoRR*, abs/1303.1152, 2013.

[KG11] Jeremy Kepner and John Gilbert. *Graph Algorithms in the Language of Linear Algebra*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2011.

[Kon17] Jakub Konecný. *Stochastic, Distributed and Federated Optimization for Machine Learning*. PhD thesis, University of Edinburgh, 2017.

[KSDY17] Can Karakus, Yifan Sun, Suhas N. Diggavi, and Wotao Yin. Straggler mitigation in distributed optimization through data encoding. In *In Advances in Neural Information Processing Systems, NIPS 2017, 4-9 December 2017, Long Beach, CA, USA*, pages 5440–5448, 2017.

[LLP+18] Kangwook Lee, Maximilian Lam, Ramtin Pedarsani, Dimitris S. Papailiopoulos, and Kannan Ramchandran. Speeding up distributed machine learning using codes. *IEEE Trans. Information Theory*, 64(3):1514–1529, 2018.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[LZZ+17] Xiangru Lian, Ce Zhang, Huan Zhang, Cho-Jui Hsieh, Wei Zhang, and Ji Liu. Can decentralized algorithms outperform centralized algorithms? A case study for decentralized parallel stochastic gradient descent. In *Advances in Neural Information Processing Systems, NIPS 2017, 4-9 December 2017, Long Beach, CA, USA*, pages 5336–5346, 2017.

[ME08] M. Mishali and Y. C. Eldar. Reduce and boost: Recovering arbitrary sets of jointly sparse vectors. *IEEE Transactions on Signal Processing*, 56(10):4692–4702, Oct 2008.

[MGR18] El Mahdi El Mhamdi, Rachid Guerraoui, and Sébastien Rouault. The hidden vulnerability of distributed learning in byzantium. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, pages 3518–3527, 2018.

[Nes12] Yurii Nesterov. Efficiency of coordinate descent methods on huge-scale optimization problems. *SIAM Journal on Optimization*, 22(2):341–362, 2012.

[RSS12] Alexander Rakhlin, Ohad Shamir, and Karthik Sridharan. Making gradient descent optimal for strongly convex stochastic optimization. In *Proceedings of the 29th International Conference on Machine Learning, ICML 2012, Edinburgh, Scotland, UK, June 26 - July 1, 2012*, 2012.

[RT16] Peter Richtárik and Martin Takáč. Parallel coordinate descent methods for big data optimization. *Mathematical Programming*, 156(1):433–484, Mar 2016.

[RTDT18] Netanel Raviv, Rashish Tandon, Alex Dimakis, and Itzhak Tamo. Gradient coding from cyclic MDS codes and expander graphs. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, pages 4302–4310, 2018.

[Saa03] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003.

[ST11] Shai Shalev-Shwartz and Ambuj Tewari. Stochastic methods for l_1-regularized loss minimization. *Journal of Machine Learning Research*, 12:1865–1892, 2011.

[Tib15] Ryan Tibshirani. Convex optimization lecture notes. http://www.stat.cmu.edu/~ryantibs/convexopt-S15/scribes/08-prox-grad-scribed.pdf, 2015.

[TLDK17] Rashish Tandon, Qi Lei, Alexandros G. Dimakis, and Nikos Karampatziakis. Gradient coding: Avoiding stragglers in distributed learning. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, pages 3368–3376, 2017.

[Wri15] Stephen J. Wright. Coordinate descent algorithms. *Math. Program.*, 151(1):3–34, 2015.

[YCRB18] Dong Yin, Yudong Chen, Kannan Ramchandran, and Peter Bartlett. Byzantine-robust distributed learning: Towards optimal statistical rates. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, pages 5636–5645, 2018.

[ZWLS10] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. Parallelized stochastic gradient descent. In *Advances in neural information processing systems*, pages 2595–2603, 2010.