Gradient Coding

Rashish Tandon, Qi Lei, Alexandros G. Dimakis  
The University of Texas at Austin  
Austin, TX  
{rashish leiqi, dimakis}@cs.utexas.edu

Nikos Karampatziakis  
Microsoft Research  
Redmond, WA  
nikosk@microsoft.com

Abstract

We propose a novel coding theoretic framework for mitigating stragglers in distributed learning. We show how carefully replicating data blocks and coding across gradients can provide tolerance to failures and stragglers for synchronous Gradient Descent. We implement our scheme in MPI and show how we compare against baseline architectures in running time and generalization error.

1 Introduction

We propose a novel coding theoretic framework for mitigating stragglers in distributed learning. The central idea can be seen through the simple example of Figure 1: Consider synchronous Gradient Descent (GD) on three workers \( W_1, W_2, W_3 \). The baseline vanilla system is shown on the left figure and operates as follows: The three workers have different partitions of the labeled data stored locally \( D_1, D_2, D_3 \) and all share the current model. Worker 1 computes the gradient of the model on examples in partition \( D_1 \), denoted by \( g_1 \). Similarly, Workers 2 and 3 compute \( g_2 \) and \( g_3 \). The three gradient vectors are then communicated to a central node (called the master/aggregator) \( A \) which computes the full gradient by summing these vectors \( g_1 + g_2 + g_3 \) and updates the model with a gradient step. The new model is then sent to the workers and the system moves to the next round (where the same examples or other labeled examples, say \( D_4, D_5, D_6 \), will be used in the same way).

The problem is that sometimes worker nodes can be stragglers \([7, 5, 4]\) i.e. delay significantly in computing a gradient. First, we discuss one way to resolve this problem if we replicate some data across machines by considering the placement in Fig. 1 (b) but without coding. As can be seen, in Fig. 1 (b) each example is replicated two times using a specific placement policy. Each worker is assigned to compute two gradients on the two examples they have for this round. For example, \( W_1 \) will compute vectors \( g_1 \) and \( g_2 \). Now let's assume that \( W_3 \) is the straggler. If we use control messages, \( W_1, W_2 \) can notify the master \( A \) that they are done. Subsequently, if feedback is used, the master can ask \( W_1 \) to send \( g_1 \) and \( g_2 \) and \( W_2 \) to send \( g_3 \). These feedback control messages can be much smaller than the actual gradient vectors but are still a system complication that can cause delays. However, feedback makes it possible for a centralized node to coordinate the workers. One can reduce network communication significantly by simply asking \( W_1 \) to send the sum of two gradient vectors \( g_1 + g_2 \) instead of sending both. The master can then create the global gradient on this batch by summing these two vectors. Unfortunately, which linear combination must be sent depends on who is the straggler. If \( W_2 \) was the straggler then \( W_1 \) should be sending \( g_2 \) and \( W_3 \) sending \( g_1 + g_3 \) so that their sum is the global gradient \( g_1 + g_2 + g_3 \).
In this paper we show that feedback and coordination is not necessary: every worker can send a single linear combination of gradient vectors without knowing who the straggler will be. The main coding theoretic question we investigate is how to design three linear combinations so that any two contain the $g_1 + g_2 + g_3$ vector in their span. In our example, in Fig. 1b, $W_1$ sends $\frac{1}{2} g_1 + g_2$, $W_2$ sends $g_2 - g_3$ and $W_3$ sends $\frac{1}{2} g_1 + g_3$. The reader can verify that $A$ can obtain the vector $g_1 + g_2 + g_3$ from any two out of these three vectors. For instance, $g_1 + g_2 + g_3 = 2 \left( \frac{1}{2} g_1 + g_2 \right) - (g_2 - g_3)$. We call this idea gradient coding.

We consider this problem in the general setting of $n$ machines and any $s$ stragglers. We first establish a lower bound: to compute gradients on all the data in the presence of any $s$ stragglers, each partition must be replicated $s + 1$ times across machines. We design two placement and gradient coding schemes that match this optimal $s + 1$ replication factor. We further consider a partial straggler setting, wherein we assume that a straggler can compute gradients at a fraction of the speed of others, and show how our scheme can be adapted to such scenarios. All proofs supplementary material can be found at [1].

We compare our scheme with the popular ignoring the stragglers approach [2]: simply doing a gradient step when most workers are done. We see that while ignoring the stragglers is faster, this loses some data and which can hurt the generalization error. This can be especially pronounced in supervised learning with unbalanced labels or heavily unbalanced features since a few examples may contain critical, previously unseen information.

### 1.1 Related Work

The slow machine problem is the Achilles heel of many distributed learning systems that run in modern cloud environments. Recognizing that, some recent work has advocated asynchronous approaches [7, 5, 9] to learning. While asynchronous updates are a valid way to avoid slow machines, they do give up many other desirable properties, including faster convergence rates, amenability to analysis, and ease of reproducibility and debugging.

Attacking the slow machine problem in synchronous machine learning algorithms has surprisingly not received much attention in the literature. There do exist general systems solutions such as speculative execution [11] but we believe that approaches tailored to machine learning can be vastly more efficient. In [2] the authors use synchronous minibatch SGD and request a small number of additional worker machines so that they have an adequate minibatch size even when some machines are slow. However, this approach does not handle well machines that are consistently slow and the data on those machines might never participate in training. In [10] the authors describe an approach for dealing with failed machines by approximating the loss function in the failed partitions with a linear approximation at the last iterate before they failed. Since the linear approximation is only valid at a small neighborhood of the model parameters, this approach can only work if failed data partitions are restored fairly quickly.

The work of [6] is perhaps the closest in spirit to our work, using coding theory and treating stragglers as erasures in the transmission of the computed results. However, we focus on codes for recovering the batch gradient of any loss function while [6] describe techniques for mitigating stragglers in two different distributed applications: data shuffling and matrix multiplication. We also mention [8], which investigates a generalized view of the coding ideas in [6].

### 2 Preliminaries

Given data $D = \{(x_1, y_1), \ldots, (x_d, y_d)\}$, with each tuple $(x, y) \in \mathbb{R}^p \times \mathbb{R}$, several machine learning tasks can be expressed as solving the following problem: $\beta^* = \arg \min_{\beta \in \mathbb{R}^p} \sum_{i=1}^d \ell(\beta; x_i, y_i)$, where $\ell(\cdot)$ is a task-specific loss function. Typically, this optimization problem can be solved using gradient-based approaches. Let $g := \sum_{i=1}^d \nabla \ell(\beta(t); x_i, y_i)$ be the gradient of the loss at the current model $\beta(t)$. Then the updates are of the form: $\beta(t+1) = h(\beta(t), g)$, where $h$ is a gradient based optimizer. Several methods such as gradient descent, accelerated gradient, conditional gradient (Frank-Wolfe), LBFGS, and bundle methods fit in this framework. However, if the number of samples, $d$, is large, a computational bottleneck in the above update step is the computation of the gradient, $g$, whose computation can be distributed.

**Notation.** Throughout this paper, we let $d$ denote the number of samples, $n$ denote the number of workers, $k$ denote the number of data partitions, and $s$ denote the number of stragglers/failures. The $n$ workers are denoted as $W_1, W_2, \ldots, W_n$. The partial gradients over $k$ data partitions are denoted as $g_1, g_2, \ldots, g_k$. The $i^{th}$ row of some matrices $A$ or $B$ is denoted as $a_i$ or $b_i$, respectively. For any
vector $x \in \mathbb{R}^n$, $\text{supp}(x)$ denotes its support i.e. $\text{supp}(x) = \{i \mid x_i \neq 0\}$. $\|x\|_0$ denotes its $\ell_0$-norm i.e. the cardinality of the support. $1_{p \times q}$ and $0_{p \times q}$ denote all 1s and all 0s matrices respectively, with dimension $p \times q$. Finally, for any $r \in \mathbb{N}$, $[r]$ denotes the set $\{1, \ldots, r\}$.

2.1 The General Setup

We can generalize the scheme in Figure 1b by setting up a system of linear equations:

$$AB = 1_{f \times k}$$

(1)

where $f$ is the number of combinations of surviving workers/non-stragglers, and we have matrices $A \in \mathbb{R}^{f \times n}$, $B \in \mathbb{R}^{n \times k}$.

To relate Eq. (1) to a scheme robust to some failures/stragglers, we say that $b_i$, the $i^{th}$ row of $B$, is associated with the $i^{th}$ worker, $W_i$. The support of $b_i$, $\text{supp}(b_i)$, corresponds to the data partitions that worker $W_i$ has access to, and the entries of $b_i$ encode a linear combination over their gradients that worker $W_i$ transmits. Let $\bar{g} \in \mathbb{R}^{k \times d}$ be a matrix with each row being the partial gradient of a data partition i.e. $\bar{g} = [g_1, g_2, \ldots, g_k]^T$. Then, worker $W_i$ transmits $\bar{g}_i$. Note that to transmit $\bar{g}_i$, $W_i$ only needs to compute the partial gradients on the partitions in $\text{supp}(b_i)$. Each row of the matrix $A$ corresponds to a specific failure/straggler scenario, to which robustness is desired. In particular, the support of its $i^{th}$ row, $\text{supp}(a_i)$, corresponds to the scenario wherein the workers $\text{supp}(a_i)$ have survived. Then, by construction, we have:

$$a_i \bar{g} = [1, 1, \ldots, 1] \bar{g} = \left( \sum_{j=1}^{k} g_j \right)^T \text{ and, } a_i B \bar{g} = \sum_{k \in \text{supp}(a_i)} a_i(k)(b_k \bar{g})$$

(2)

Thus, the entries of $a_i$ encode a linear combination which, when taken over the transmitted gradients of the non-straggler workers, would yield the full gradient.

Going back to the example in Fig. 1b, the corresponding $A$ and $B$ matrices under this generalization are:

$A = \begin{pmatrix} 0 & 1 & 2 \\ 1 & 0 & 1 \\ 2 & -1 & 0 \end{pmatrix}$ and $B = \begin{pmatrix} 1/2 & 1 & 0 \\ 0 & 1 & -1 \\ 1/2 & 0 & 1 \end{pmatrix}$, with $f = 3$, $n = 3$, $k = 3$. It is easy to check that $AB = 1_{3 \times 3}$. In general, we shall seek schemes, through the construction of $(A, B)$, which are robust to any $s$ stragglers.

3 Full Stragglers

In this section, we consider schemes robust to any $s$ stragglers, given $n$ workers (with $s < n$). We assume that a straggler is a full straggler i.e. it can be arbitrarily slow to the extent of complete failure. We show how to construct the matrices $A$ and $B$, with $AB = 1$, such that the scheme $(A, B)$ is robust to any $s$ stragglers.

Consider any such scheme $(A, B)$. Since every row of $A$ represents a set of non-straggler workers, all possible sets over $[n]$ of size $(n - s)$ must be supports in the rows of $A$. Thus $f = \binom{n}{s} = \binom{n}{n - s}$ i.e. the total number of ways to choose $s$ stragglers out of $n$ workers. Now, since each row of $A$ represents a linear span over some rows of $B$, and since $AB = 1$, this leads us to the following requirement on $B$:

**Condition 1 (B-Span).** Consider any scheme $(A, B)$ robust to any $s$ stragglers, given $n$ workers (with $s < n$). Then we require that for every subset $I \subseteq [n], |I| = n - s$:

$$1_{1 \times k} \in \text{span}\{b_i \mid i \in I\}$$

(3)

where $\text{span}\{\cdot\}$ is the span of vectors. In other words, the all 1s vector must be in the span of any $n - s$ rows of $B$.

The B-Span condition above is of course necessary. However, it is also sufficient. In particular, given a $B$ satisfying Condition 1, we can construct an $A$ (described in Algorithm 1 in the supplementary) such that $AB = 1$, and $A$ has the support structure discussed above.

Therefore, to obtain a valid scheme $(A, B)$, we only need to furnish a $B$ satisfying the B-Span condition (Condition 1). A trivial $B$ that works is $B = 1_{n \times k}$, the all ones matrix. However, this is wasteful since it implies that each worker gets all the partitions and computes the full gradient. Our goal is to construct $B$ satisfying Condition 1 while also being as sparse as possible in each row. In this regard, we have the following theorem, which gives a lower bound on the number of non-zeros in any row of $B$. 

3
**Theorem 1** (Lower Bound on B’s density). Consider any scheme \((A, B)\) robust to any \(s\) stragglers, given \(n\) workers (with \(s < n\)) and \(k\) partitions. Then, if all rows of \(B\) have the same number of non-zeros, we must have: \(|b_i|_0 \geq \frac{n}{s}(s+1)\) for any \(i \in [n]\).

Theorem 1 implies that any scheme \((A, B)\) that assigns the same amount of data to all the workers must assign at least \(\frac{n}{s} \frac{s+1}{s}\) fraction of the data to each worker. Since this fraction is independent of \(k\), for the remainder of this paper we shall assume that \(k = n\) i.e. the number of partitions is the same as the number of workers. In this case, \(B\) must be a square matrix satisfying Condition 1, with each row having at least \((s+1)\) non-zeros. In the sequel, we demonstrate two constructions for \(B\) which achieve this lower bound.

### 3.1 Fractional Repetition Scheme

We now provide a construction for \(B\) that works by replicating the task done by a subset of the workers. We note that this construction is only applicable when the number of workers, \(n\), is a multiple of \((s+1)\), where \(s\) is the number of stragglers we seek robustness to. In this case, the construction is as follows:

- We divide the \(n\) workers into \((s+1)\) groups of size \((n/(s+1))\).
- In a group, we divide the data equally and disjointly, assigning \((s+1)\) partitions to each worker.
- All the groups are replicas of each other.
- When finished computing, every worker transmits the sum of its partial gradients.

Fig. 2 shows an instance of the above construction for \(n = 6, s = 2\). A general description of \(B\) constructed in this way (denoted as \(B_{frac}\)) is shown in Eq. (6) (in the supplementary). It is easy to see that this construction can yield robustness to \(s\) stragglers. Since any particular partition is replicated over \((s+1)\) workers, any \(s\) stragglers leaves at least one non-straggler worker to process it now. We have the following theorem.

**Theorem 2.** Consider \(B_{frac}\) constructed as in Eq. (6), for a given number of workers \(n\) and stragglers \(s < n\). Then, \(B_{frac}\) satisfies the B-Span condition (Condition 1). Consequently, the scheme \((A, B_{frac})\), with \(A\) constructed using Algorithm 1, is robust to any \(s\) stragglers.

The construction of \(B_{frac}\) matches the density lower bound in Theorem 1. Also, the above theorem shows that the scheme \((A, B_{frac})\), with \(A\) constructed from Algorithm 1, is robust to \(s\) stragglers.

### 3.2 Cyclic Repetition Scheme

In this section, we construct an alternate \(B\) which also matches the lower bound in Theorem 1 and satisfies Condition 1. However, in contrast to the previous section, this construction does not require \(n\) to be divisible by \((s+1)\). Here, instead of assigning disjoint collections of partitions, we consider a cyclic assignment of \((s+1)\) partitions to the workers. We construct a \(B = B_{cyc}\) with the support structure:

\[
\text{supp}(B_{cyc}) = \begin{bmatrix}
* & * & \cdots & * & 0 & 0 & \cdots & 0 & 0 \\
0 & * & \cdots & * & 0 & 0 & \cdots & 0 & 0 \\
\vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & \cdots & 0 & 0 & * & \cdots & * & * \\
* & \cdots & \cdots & \cdots & * & * & \cdots & * & * \\
\end{bmatrix}_{n \times n}
\]

where * indicates non-zero entries in \(B_{cyc}\). The first row of \(B_{cyc}\) has its first \((s+1)\) entries assigned as non-zero. As we move down the rows, the positions of the \((s+1)\) non-zero entries shift one step to the right, and cycle around until the last row.

Given the support structure in Eq. 4, the actual non-zero entries must be carefully assigned in order to satisfy Condition 1. The basic idea is to pick every row of \(B_{cyc}\) with the fixed support, to lie in a suitable subspace \(S\) that contains the all ones vector \(1_{n \times 1}\). It turns out that a choice of random subspace \(S\) (described as the null space of a random matrix satisfying the so-called MDS
property in coding theory) can produce such a $B_{cyc}$. Algorithm 2 (in the supplementary) describes the construction of $B_{cyc}$. Then, we have the following theorem.

**Theorem 3.** Consider a $B_{cyc}$ constructed using the randomized construction in Algorithm 2, for a given number of workers $n$ and stragglers $s (< n)$. Then, with probability $1$, $B_{cyc}$ satisfies the B-Span condition (Condition 1). Consequently, the scheme $(A, B_{cyc})$, with $A$ constructed using Algorithm 1, is robust to any $s$ stragglers.

## 4 Partial Stragglers

In this section, we review our earlier assumption of full stragglers. Instead, we consider a more plausible scenario of slow workers, but assume a known slowdown factor. We say that a straggler is an $\alpha$-partial straggler (with $\alpha > 1$) if it is at most $\alpha$ slower than any non-straggler. This means that if a non-straggler completes a task in time $T$, an $\alpha$-partial straggler would require at most $\alpha T$ time to complete it. Now, we augment our distribution schemes to be robust to any $s$ stragglers, assuming any straggler is an $\alpha$-partial straggler.

Note that our earlier constructions are still applicable: a scheme $(A, B)$, with $B = B_{frac}$ or $B = B_{cyc}$, would still provide robustness to $s$ partial stragglers. However, given that no machine is slower than a factor of $\alpha$, a more efficient scheme is possible by exploiting at least some computation on every machine. Our basic idea is to couple our earlier schemes with a naive distribution scheme, but on different parts of the data. We split the data into a naive component, and a coded component. The key is to do the split such that whenever an $\alpha$-partial straggler is done processing its naive partitions, a non-straggler would be done processing both its naive and coded partitions. In general, for any $(n, s, \alpha)$, our two-stage scheme works as follows:

- We split the data $D$ into $n + \frac{s+1}{\alpha}$ equal-sized partitions — of which $n$ partitions are coded components, and the rest are naive components.
- Each worker gets $\frac{s+1}{\alpha}$ naive partitions, distributed disjointly.
- Each worker gets $(s+1)$ coded partitions, distributed according to an $(A, B)$ distribution scheme robust to $s$ stragglers (e.g. with $B = B_{frac}$ or $B = B_{cyc}$).
- Any worker, $W_i$, first processes all its naive partitions and sends the sum of their gradients to the aggregator. It then processes its coded partitions, and sends a linear combination, as per the $(A, B)$ distribution scheme.

Note that each worker has to send two partial gradients, instead on one, as in earlier schemes. However, a speedup gained in processing a smaller fraction of the data may mitigate this overhead in communication, since now each non-straggler has to process an $\frac{s+1}{n} \left( \frac{\alpha}{\alpha + \alpha} \right)$ fraction of the data, as opposed to an $\frac{s+1}{n}$ fraction in earlier schemes. Fig. 3 illustrates our two-stage strategy for $n = 3, s = 1, \alpha = 2$. We see that each non-straggler gets $4/9 = 0.44$ fraction of the data, instead of a $2/3 = 0.67$ fraction (for e.g. in Fig 1b).

## 5 Experiments

In this section, we present experimental results comparing the proposed gradient coding schemes with baseline approaches. We implemented the distributed schemes in Python using MPI4Py[3], an open source MPI implementation. Our experiments were performed on a cluster of $n = 12$ machines (Medium instances) on Amazon EC2. We compare against the naive scheme where the data is divided uniformly across all the workers without replication.

**Artificial Dataset:** In our first experiment we solve a logistic regression problem using gradient descent. We artificially generate a dataset of $d = 554400$ samples $D = \{(x_1, y_1), \ldots, (x_d, y_d)\}$, from the model $x \sim 0.5 \times \mathcal{N}(\mu_1, I) + 0.5 \times \mathcal{N}(\mu_2, I)$ (for random $\mu_1, \mu_2 \in \mathbb{R}^p$), and $y \sim \text{Ber}(p)$, with $p = 1/(\exp(2x^T \beta^*) + 1)$, where $\beta^* \in \mathbb{R}^p$ is the true regressor. In our experiments, we use a model dimension of $p = 100$, and chose $\beta^*$ randomly. Based on the scheme being
considered, each worker loads a certain number of partitions of the data into memory before starting the iterations. In iteration $t$ the aggregator sends the latest model $\beta^{(t)}$ to all the workers (using $\text{isend}(\cdot)$). Each worker receives the model (using $\text{irecv}(\cdot)$) and starts a gradient computation. Once finished, it sends its gradient(s) back to the aggregator. When sufficiently many workers have returned with their gradients, the aggregator computes the overall gradient, performs a descent step, and moves on to the next iteration. We artificially add delays to $s$ random workers in each iteration (using $\text{time.sleep}(\cdot)$). Figure 4 presents the results of our experiments with $s = 1$ and $s = 2$ stragglers. Note that for partial straggler schemes, $\alpha$ denotes the slowness factor.

**Real Dataset:** We also compare our approach against the scheme that ignores the stragglers. In this case we use a real dataset for binary classification and train a logistic regression models. The baseline scheme waits for the first $n - s$ machines to compute gradients and ignores the rest. In other words, ignoring the $s$ stragglers corresponds to batch SGD using a $(n - s)/n$ fraction of the data. On the contrary, our scheme provides fault tolerance since data blocks are replicated across $s + 1$ machines and we do not need to wait for the stragglers to finish. Therefore we can perform batch SGD with each batch involving data from all the machines, or even full gradient descent. Figure 5 shows a comparison of generalization error (using the AUC metric) for three and five machines and one straggler. As can be seen, our framework tolerates $s = 1$ straggler so it is a factor of two slower compared to ignoring the stragglers. However, it will see all the data while the scheme that ignores stragglers misses some data ($1/3$ and $1/5$ respectively for 3 and 5 machines). Note that in this experiment the straggling worker was fixed to be the same machine across iterations.

We see from Fig. 5a and 5b that ignoring stragglers can allow faster iterations but can increase the generalization error since some examples may be lost. As the number of machines increases, this effect should not be as pronounced if the number of stragglers is scaling as a constant fraction of the number of machines in the cluster. If however, we expect a fraction, e.g. 5% of the machines to be significantly slower than the rest, ignoring 5% of the examples could have an effect in learning. This can be particularly problematic for datasets with labels or features that are very unbalanced: for those cases missing even a few important examples can drastically increase the generalization error.
References


