BYZANTINE FAULT-TOLERANCE VIA REACTIVE REDUNDANCY IN PARALLELIZED GRADIENT DESCENT

A Thesis submitted to the Faculty of the Graduate School of Arts and Sciences of Georgetown University in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

By

Connor Young Lu, B.S.

Washington, DC
June 30, 2021
Byzantine Fault-tolerance via Reactive Redundancy in Parallelized Gradient Descent

Connor Young Lu, B.S.

Thesis Advisor: Nitin H. Vaidya

Abstract

We consider the problem of multi-agent optimization in the presence of Byzantine workers. Specifically, we focus on the case of parallelized stochastic gradient descent utilizing $n$ workers, up to $f < \frac{n}{2}$ of which are Byzantine with the goal of corrupting optimization. We propose four schemes based off of a known fault-tolerant algorithm, ReactiveRedundantSGD, implement all five techniques, and use them in gradient descent optimization experiments. Through these experiments, we verify the exact fault-tolerance of the original algorithm and show that its speed can be consistently improved by three schemes' mechanisms: adaptive redundancy, rollback recovery, and lightweight gradient filtering.

Index words: Byzantine tolerance, stochastic gradient descent, redundancy
DEDICATION

To my parents, who have supported me in every way possible, and to my grandparents, whom I can never thank enough for their roles in my life.

I love you all.
ACKNOWLEDGMENTS

Thank you to Nitin for guiding me through the thesis process, thank you to Micah for helping me to enjoy my education in computer science, and thank you to Mark for so many helpful conversations throughout my time at Georgetown.
# Table of Contents

## Chapter

1. **Introduction** .................................................. 1
   1.1 System Architecture .......................................... 2
   1.2 Fault-tolerance ................................................. 2

2. **Background** .................................................. 5
   2.1 Stochastic Gradient Descent ................................. 5
   2.2 Parallelized-SGD ............................................... 6
   2.3 Exact Fault-tolerance ......................................... 6
   2.4 Reactive Redundancy ......................................... 7
   2.5 Related Works ................................................ 10

3. **Reactive Redundant SGD Schemes** ........................... 11
   3.1 Diminishing-LR ................................................ 11
   3.2 Adaptive-\(q\) .................................................... 12
   3.3 Rollback ......................................................... 13
   3.4 Targeted Fault-checking ..................................... 14

4. **Optimization Experiments** ................................... 16
   4.1 Experimental Design .......................................... 16
   4.2 Linear Regression Results ..................................... 19
   4.3 MNIST Classification Results ................................. 35

5. **Conclusion** .................................................. 45

## Bibliography ..................................................... 47
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Average BASELINE and fault-free linear regression results</td>
<td>22</td>
</tr>
<tr>
<td>4.2</td>
<td>BASELINE linear regression example</td>
<td>23</td>
</tr>
<tr>
<td>4.3</td>
<td>BASELINE computation efficiency for different fault-check probabilities</td>
<td>24</td>
</tr>
<tr>
<td>4.4</td>
<td>Adaptive fault-checking compared to BASELINE algorithm for linear regression</td>
<td>28</td>
</tr>
<tr>
<td>4.5</td>
<td>BASELINE MNIST classification accuracy</td>
<td>37</td>
</tr>
<tr>
<td>4.6</td>
<td>Diminishing-LR compared to BASELINE for MNIST classification</td>
<td>38</td>
</tr>
<tr>
<td>4.7</td>
<td>Adaptive fault-checking compared to BASELINE for MNIST classification</td>
<td>40</td>
</tr>
</tbody>
</table>
# List of Tables

4.1 **Baseline** linear regression results ................................. 21
4.2 **Baseline** computation efficiency ................................. 25
4.3 Example of diminishing-LR scheme for linear regression ............ 26
4.4 Adaptive fault-checking linear regression ............................. 29
4.5 Rollback recovery linear regression ................................. 30
4.6 Targeted fault-checking linear regression (scaled-reversal faults) ... 32
4.7 Targeted fault-checking linear regression (constant faults) ........... 33
4.8 Targeted fault-checking linear regression (reversal faults) ............ 34
4.9 **Baseline** MNIST classification results ............................ 37
4.10 Adaptive fault-checking MNIST classification results ................ 40
4.11 Rollback recovery MNIST classification results ..................... 41
4.12 Targeted fault-checking MNIST classification (scaled-reversal faults) . 42
4.13 Targeted fault-checking MNIST classification (constant faults) ....... 43
4.14 Targeted fault-checking MNIST classification (reversal faults) ........ 43
Distributed optimization in multi-agent systems has been studied for decades and become increasingly relevant in recent years, especially in applications of machine learning (ML) [2, 4, 17, 23]. The goal of the optimization problem considered in this thesis is to minimize the output of some cost function over a set of data points $Z$ [17]; we consider the multi-agent scenario utilizing $n$ agents to perform the optimization using the same global cost function. Suppose there is a set $Z$ of $N$ data points and $n$ agents $i = 1, \ldots, n$ sharing the cost function $\ell(w, z)$, where $w \in \mathbb{R}^d$ is a vector, $d$ is a positive integer, and $z \in Z$. The goal of the system is to find a $w^*$ that minimizes the total cost, that is:

$$w^* \in \arg\min_w \frac{1}{N} \sum_{z \in Z} \ell(w, z).$$

(1.1)

Most work in distributed optimization assumes all workers to be honest; that is, workers are following the prescribed algorithm correctly. In reality, there is often a non-zero chance of accidental failure or malicious behavior in a multi-agent system [1, 2, 6, 7, 15, 22]. For this thesis, we consider the distributed multi-agent optimization setting introduced by Su and Vaidya in [22] containing a central parameter server (PS) and $n$ workers, up to $f < \frac{n}{2}$ of which are Byzantine.

The identities of the Byzantine workers are unknown a priori and they may behave arbitrarily to corrupt the process of optimization [14]. Any distributed optimization technique requires data to be communicated between worker machines eventually.
Instead of communicating data as the technique prescribes, Byzantine workers may communicate arbitrarily incorrect data, which we refer to as faults. We refer to techniques that achieve their optimization goal in the presence of Byzantine workers as fault-tolerant. Byzantine workers are assumed to have complete knowledge about the system and may communicate with one another to coordinate faults. This thesis explores Byzantine fault-tolerant optimization in parallelized learning, focusing on stochastic gradient descent (SGD).

1.1 System Architecture

This thesis considers a synchronous system comprised of the following components:

- central parameter server (PS)
- $n$ workers
- set of data points $Z$ of size $N$
- global loss function $\ell(w, z)$
- parameter vector $w \in \mathbb{R}^d$

All workers have access to the same global loss function and data points. The goal of the PS is to find an optimal $w^*$ minimizing Equation 1.1 given its global loss function $\ell$ and data set $Z$, using the worker machines to parallelize the process. This is the distributed version of the framework serving as the basis of most modern learning techniques, including neural networks and support vector machine [3].

1.2 Fault-tolerance

We say that the set of techniques that can achieve their optimization goal in the presence of Byzantine workers possess fault-tolerance. In particular, this thesis considers
algorithms possessing exact fault-tolerance. In the exact fault-tolerance problem, the goal is to design a distributed algorithm that allows all the non-faulty agents to compute a minimum point of the aggregate cost of the non-faulty agents [12].

**Definition 1** A parallelized-SGD method has exact fault-tolerance if the PS asymptotically converges to a minimum point \( w^* \) exactly, despite the presence of Byzantine workers.

Call the set of up to \( f \) Byzantine workers \( B \) and the set of remaining \((n - f)\) honest workers \( H \). Formally, a distributed optimization technique has exact fault-tolerance if it returns an estimate \( w^* \) such that

\[
w^* \in \arg \min_{w \in \mathbb{R}^d} \sum_{i \in H} \ell_i(w) .
\]  

(1.2)

In other words, a technique with exact fault-tolerance returns the estimate of the honest workers only. Throughout this thesis, we assume that the number of Byzantine workers present in the system does not exceed \( f < \frac{n}{2} \).

In multi-agent SGD, the PS relies on workers to communicate gradients honestly. Most techniques addressing fault-tolerance in this setting use some gradient filter [6, 8, 10, 16, 23, 24] to eliminate faults; for example, [23] uses different versions of a median-based aggregation rule, dropping gradients too far from the median.

We explore a less common strategy: the use of redundant calculations of gradients for fault identification and correction. Any two honest workers optimizing the same cost function should calculate the same gradient value from the same data point and parameter vector; otherwise, one is a fault. This concept expanded to multi-agent systems with \( n > 2 \) workers can be used to identify faults in parallelized-SGD [5, 7, 11, 20].

This thesis revolves around the implementation and analysis of a redundant SGD algorithm with exact fault-tolerance [11]. To understand the benefits of redundancy
in fault-tolerant learning, we compare its performance against four variant schemes intended to improve the algorithm’s performance in terms of convergence speed and/or efficiency:


- We propose and implement four schemes modifying the original algorithm without affecting its fault-tolerant properties and use them in the same optimization tasks.

- We compare the behavior of the proposed schemes to the original algorithm, showing that small modifications can improve performance in the Byzantine setting.

The remainder of this thesis is organized as follows:

1. In Chapter 2, we provide a background on the fundamental concepts of this thesis: parallelized-SGD, exact fault-tolerance, and reactive redundancy.

2. We introduce and describe our four proposed fault-tolerant schemes in Chapter 3.

3. We explain the design of our optimization experiments and discuss the results in Chapter 4.

4. We summarize the key observations from our experiments and suggest future work inspired by our experience in Chapter 5.
2.1 **Stochastic Gradient Descent**

Gradient descent is a technique for minimizing an objective function of a real-valued parameter vector of \( d \) dimensions; in our case, the objective function is the loss function \( \ell(w, z) \). This is accomplished by updating the parameter in the opposite direction of the gradient of the function w.r.t. the parameter itself— that is, \( \nabla \ell_w(w, z) \) [21]. With each update, or *optimization step*, the gradient of the loss function determines the direction in which the algorithm updates \( w \). Gradient descent (GD) requires a hyperparameter, the *learning rate (LR)* \( \eta \), which determines the magnitude of optimization steps. Formally, each iteration \( t \geq 0 \) during gradient descent, the PS maintains an estimate \( w_t \) of \( w^* \) and updates it using gradients of the loss functions from a set \( Z_t \) augmented by the LR \( \eta \):

\[
   w^{t+1} = w^t - \eta \times \nabla \ell_w(w, Z^t) .
\]  

(2.1)

Stochastic gradient descent is a type of gradient descent in which updates are performed for every example \( z \in Z \), selected randomly in each iteration. A data point is an example \( z \) associated with a target \( y \); in linear regression, \( z \) would be one or more numerical values and \( y \) would be a single value, while in the MNIST data set, \( z \) is a 28x28 matrix of values and \( y \) is an integer label. Formally, SGD updates the parameter \( w \) each iteration \( t \geq 0 \) for a randomly selected example \( z^t \) and its associated
target $y^t$:  
\[ w^{t+1} = w^t - \eta \times \nabla \ell(w^t; z^t; y^t) \tag{2.2} \]

2.2 Parallelized-SGD

In [25], the authors propose an expedited version of SGD. The synchronous PARALLEL-SGD algorithm employs a set of worker machines to perform gradient calculation and communicate the gradients back to a central PS, which maintains and updates the parameter vector $w$. In each iteration $t \geq 0$, the PS randomly selects a set of data points $Z^t \subset Z$ and randomly assigns them to workers. Let $m_i$ denote the number of data points assigned to worker $i$ such that $\sum_{i=1}^{n} m_i = m = |Z^t|$, and let the points assigned to worker $i$ at time $t$ be denoted $z^t_i = \{z^t_{i,1}, \ldots, z^t_{i,m_i}\}$. Each worker $i$ computes gradients from its assigned data points $z^t_i$ at time $t$:
\[ g^t_{i,j} = \nabla \ell(w^t, z^t_{i,j}) , \]
where $j = 1, \ldots, m_i$ and $i = 1, \ldots, n$ and sends its gradients to the PS.

The PS then updates the parameter estimate $w^t$ at time $t$ using the average value of all $m$ gradients; the average gradient $\bar{g}^t$ at time $t$ is calculated as:
\[ \bar{g}^t = \frac{1}{m} \sum_{z^t \in Z^t} \nabla \ell(w^t, z^t) . \]

Then, the estimate $w^t$ is updated by
\[ w^{t+1} = w^t - \eta \bar{g}^t . \tag{2.3} \]

2.3 Exact Fault-tolerance

The PARALLEL-SGD algorithm cannot tolerate Byzantine workers, which may not follow the algorithm described above and can communicate faulty gradients. Byzantine workers’ identities are unknown \textit{a priori}, but the set of Byzantine workers remains
fixed—that is, an identified Byzantine worker remains known and, therefore, can be
excluded from future iterations. As stated in Section 1.2, we consider a system with
n workers and up to \( f < \frac{n}{2} \) Byzantine workers.

It has already been mentioned that gradient filters are used in distributed SGD
for fault-tolerance [2, 6, 7, 8, 9, 10, 13, 16, 24]. For all of these filters, exact fault-
tolerance is generally not achievable when Byzantine workers are unknown \textit{a priori}
[22]. However, with enough redundancy, exact fault-tolerance can be achieved.

The next section describes a notion that is the vital to the algorithms studied in
this thesis: the use of redundant gradient calculations for fault-tolerance.

2.4 \textbf{Reactive Redundancy}

Gupta and Vaidya [11] presented a novel parallelized-SGD algorithm, which we call
\texttt{ReactiveRedundantSGD}; the algorithm is similar to \texttt{ParallelSGD} with the
addition of redundant gradient calculations. We describe the randomized version of
the algorithm. In this version, every iteration \( t \geq 0 \), the algorithm randomly decides
to either perform an iteration of \texttt{ParallelSGD} with probability \( 1 - q \) or \texttt{Reac-
tiveRedundantSGD} with probability \( q \). Only iterations using \texttt{ReactiveRedun-
dantSGD} use reactive redundancy to check for faults.
Algorithm 1 Randomized Redundancy Algorithm

Data: data points $Z$ for the current iteration, learning rate $\eta$, fault-check probability $q$, $n$ total workers, and maximum of $f < \frac{n}{2}$ Byzantine workers. $w$ is the optimization goal.

1. if check for faults then
   2. assign each data point $z \in Z$ to $(f + 1)$ different workers
   3. $G \leftarrow$ list of $g_z \forall z \in Z$ such that $g_z$ is a list of all $(f + 1)$ gradient vectors computed from $z$
   4. $Z' \leftarrow$ list of any $z \in Z$ such that the gradients in $g_z$ are not unanimous
   5. if $Z'$ not empty then
      6. assign each $z \in Z'$ to $f$ new workers
      7. accumulate gradients in $G$
      8. for all $g_z \in Z'$ do
         9. identify correct gradients via majority vote
         10. remove faulty gradients from $g_z$
         11. remove identified Byzantine worker(s) from future steps
         12. reduce $n$ and $f$ by 1 for each identified Byzantine worker
         13. $\bar{G} \leftarrow$ average of the honest gradients $g_z \in G$ of each data point $z \in Z$
      14. $\bar{G} \leftarrow$ average of the gradients $g_z \in G$ of each data point $z \in Z$
   15. else
      16. assign each data point $z \in Z$ to 1 worker
      17. $G \leftarrow$ elements $g_z \forall z \in Z$ such that $g_z$ is a gradient vector computed from $z$
      18. $\bar{G} = \frac{1}{|Z|} \sum_{z \in Z} g_z$
      19. $w = w - \eta \bar{G}$
   20. broadcast updated parameter $w$ to workers

In redundant iterations of Algorithm 1, workers $i = [1, \ldots, n]$ communicates a list $g_i$ containing $m_i$ gradients: ignoring the notation for the $t^{th}$ iteration, $g_i = [g_{i,1}, \ldots, g_{i,m_i}]$ for worker $i$ such that $g_{i,j}$ is the $j^{th}$ gradient of worker $i$. Each $z \in Z$ is assigned to $f + 1$ different workers, meaning that each data point is assigned to at least one honest worker.

Referring to line 3, let $g_z \in Z$ denote the set of all gradients calculated from the point $z$. The fault-detection code is simple: for all $z \in Z$, the set $g_z$ of all associated gradients $g_{i,j}$ must be unanimous, otherwise a fault exists. If a set of gradients $g_z$ is not unanimous, the point $z$ is faulty and the algorithm enacts reactive redundancy.
Referring to line 4 of Algorithm 1, \( Z' \) denotes the set of all faulty points \( z' \). The PS re-assigns each faulty point \( z' \) to an additional \( f \) workers (line 6) and waits for gradients to be communicated. At this juncture, the expanded set of gradients \( g_{z'} \) for any faulty point \( z' \) contains \( 2f + 1 \) gradients and at most \( f \) of them are faulty. Thus, the fault-correction code is also simple: for any faulty set of gradients \( g_{z'} \), identify the correct value by majority vote, removing faulty gradients and excluding the responsible workers from future iterations (lines 8-11).

After Byzantine workers are identified, the values of \( f \) and \( n \) must be decremented accordingly (line 12). The direction of the parameter update \( \bar{g} \) is the average of the honest gradient from each data point.

We provide our definition of **computation efficiency**:

**Definition 2** The **computation efficiency** of a coding scheme is the ratio of the number of gradients used for parameter update, given in Equation 2.3, to the number of gradients computed by the workers in total.

The computation efficiency of a single step of \texttt{ReactiveRedundantSGD} is \( \frac{1}{f+1} \) at best and \( \frac{1}{2f+1} \) at worst (when reactive redundancy is imposed). The scheme can be used deterministically in every iteration instead of being used randomly with probability \( q \), but the randomized version is more computationally efficient because non-redundant steps utilize every gradient calculated for optimization. Imagine the deterministic and randomized algorithms both execute for 10 steps in which no faults occur, and that the randomized scheme uses \( q = .5 \), and \( f = 4 \): the deterministic scheme would have efficiency of at most \( \frac{1}{5} \), while the randomized scheme would have efficiency of at least \( \frac{5}{19} \) (assuming exactly half of the iterations are redundant).
2.5 Related Works

We consider the particular setting proposed in [22], and many recent works consider either the same or a very similar setting. Several methods utilizing the concept of redundancy for Byzantine fault-tolerant parallelized-SGD have been proposed [5, 7, 20], but these schemes fall short of the fault-tolerance and efficiency guarantees of \textsc{ReactiveRedundantSGD} formulated in [11]. Specifically, previous algorithms fall short of exact fault-tolerance, as defined in Definition 1.

The scheme proposed in [7] requires loss functions whose arguments are linear in the learning parameters. The scheme \textsc{DRACO} in [5] uses redundant gradient calculations to achieve robustness, but cannot achieve the same efficiency as \textsc{ReactiveRedundantSGD} while maintaining exact fault-tolerance. A similar scheme \textsc{DETOX} in [20] uses redundancy and hierarchical aggregation to improve efficiency over \textsc{DRACO}, but likewise falls short of exact fault-tolerance.

As stated in Section 1.2, a majority of approaches in this setting achieve fault-tolerance with a mathematical gradient filter [2, 6, 8, 10, 13, 16, 19, 24]. Gradient filters exclude potentially faulty gradients from optimization by removing or replacing “outliers”: There are different filters that identify outliers in different ways, usually involving some norm of all received gradients: KRUM filter [2], trimmed-mean [24], median [24], geometric median of means [6], gradient norm-clipping [10], comparative gradient elimination [13], SEVER filter [8]; other filters [16, 19] do not directly filter gradients received on the PS, but rely on the same forms of gradient aggregation for robustness. More fundamentally, none of the gradient filters mentioned provide exact fault-tolerance unless the data itself is redundant.
In this chapter, we propose and describe four algorithmic variants to the ReactiveRedundantSGD algorithm:

1. Diminishing learning rate: reduce the LR used for gradient descent over time.
2. Adaptive fault-checking: increase the value of $q$, the probability of a fault-check, over time.
3. Rollback recovery: maintain a rollback state of the parameter vector $w$ from earlier in training to replace the current state if it degrades.
4. Targeted fault-checking: in non-redundant steps, use a lightweight filter to identify abnormal gradients, initiating reactive redundancy.

We refer to the ReactiveRedundantSGD algorithm described in Algorithm 1 as the Baseline algorithm for the remainder of the thesis.

3.1 Diminishing-LR

The diminishing-LR scheme simply makes regular updates to $\eta$, the learning rate. The implementation for our experiments monotonically reduces the LR $\eta$ at the end of iterations at a regular interval $\gamma$—after line 20 of Algorithm 1.

If the iteration $t$ is not a multiple of $\gamma$, the LR remains the same: $\eta^t = \eta^{t-1}$. We implement three different schedules that change the LR in different ways. At iteration
that is a multiple of $\gamma$, the scheme sets the next LR $\eta^{t+1}$ as a function of $\eta^t$ and $\delta_\eta$ according to one of three schedules:

- linear schedule: decays $\eta$ by a constant factor $\delta_\eta$ each reduction

  $\eta^t = \eta^0 - (\delta_\eta \times t)$

  - the size of diminishment in the linear schedule is set at $\delta_\eta = \frac{\eta - \eta_{\text{min}}}{S/\gamma}$, where $S$ is iteration of the last diminishment and $\eta_{\text{min}}$ is the minimum LR

- exponential schedule: decays $\eta$ by some proportion of its current value $0 < \delta_\eta < 1$

  $\eta^t = \eta^{t-1} \times \delta_\eta$

- inverse schedule: decays $\eta$ inversely with time

  $\eta^t = \frac{\eta^0}{t}$

3.2 ADAPTIVE-\textit{q}

In our work, we initially performed experiments using the BASELINE algorithm. We observed that in many instances, Byzantine workers were all caught very early on and overall training time was hardly affected. Since starting the attacks late in training provides fewer opportunities to identify the faulty agents, while also causing significant damage to the outcome of the algorithm, it is potentially useful to design mechanisms that improve the chance of identifying faulty gradients towards the end of the training.

In particular, increasing $q$ monotonically is an adaptive fault-checking strategy that prioritizes efficiency over fault-tolerance in the beginning then shifts to prioritize fault-tolerance more over time.
Similar to the previous scheme, we perform an augmentation to $q$ at the interval every $\gamma_q > 0$ iterations; that is, in any iteration $t$ where $t$ is a multiple of $\gamma_q$:

$$q^{t+1} = q^t + \delta_q,$$

where $\delta_q > 0$, with the added constraint that $q \leq 1.0$ in every iteration. This update of $q$ takes place at the very end of an iteration, after line 20 in Algorithm 1. In iterations $t$ that are not multiples of $\gamma$, $q$ is unchanged.

In our implementation, we define a maximum value $q^{\text{max}}$ of $q$ and calculate the augmentation value as

$$\delta_q = \frac{q^{\text{max}} - q}{S/\gamma},$$

where $S$ is the last step of adapting $q$.

### 3.3 Rollback

Checkpointing and rollback/rollforward schemes can be used to address fault-tolerance in distributed systems; for example, Pradhan and Vaidya formulated a mechanism by which a spare module is used to detect faults and a reactive rollback/rollforward scheme corrects the fault [18]. We designed a rollback scheme to allow quick recovery from faults (via rollback state) that does not roll back to a worse state. The scheme requires an interval $\gamma$ and a validation set $V$ used to estimate the average losses of the current parameter vector $w$ and the rollback state $w_{\text{roll}}$.

Every $\gamma$ iterations, this scheme decides whether to keep the current parameter vector $w$ or replace it with the rollback state, a vector $w_{\text{roll}}$. The decision depends on the estimated losses of $w$ and $w_{\text{roll}}$, calculated using the cost function $\ell$ and the validation set $V$. 
Formally, a parameter $w^t$ is saved as the rollback state $w_{\text{roll}}$ if there is no previous state (i.e., if time $t = 0$) or if

$$\frac{1}{|V|} \sum_{v \in V} \ell(w^t, v) \leq \frac{1}{|V|} \sum_{v \in V} \ell(w_{\text{roll}}, v);$$

otherwise the rollback state is loaded to replace the parameter vector $w$. This checkpoint mechanism takes place at the end of an iteration, after line 20 in Algorithm 1. Additionally, we implement this rollback checkpoint at the time that a fault is identified: at the end of the If statement on line 5.

Although the use of a validation set could be costly if too frequent or the set is too large, the rollback scheme has potential to speed up the BASELINE algorithm by providing a lower-bound for the quality of $w$ throughout training in spite of Byzantine faults.

### 3.4 TARGETED FAULT-CHECKING

The fourth scheme essentially bridges the deterministic and randomized versions of the BASELINE algorithm. Filtering techniques alone have proved successful for Byzantine tolerance in other works. The BASELINE utilizes redundancy in randomly selected iterations, meaning that all remaining iterations have no fault-tolerant properties. We propose a scheme comprised of the BASELINE with one additional mechanism: the option to enact reactive redundancy in a non-redundant iteration using a lightweight gradient filter. Although redundancy and filtering are already used in conjunction for Byzantine fault-tolerance in [20] hierarchically, we use our gradient filter differently. Instead of removing or replacing gradients, our filter is used to make a binary decision: either continue with non-redundant gradient descent, or enact $(2f + 1)$-redundancy.

At interval $t$, the Targeted fault-check scheme calculates the average of all gradients received on the PS as $\bar{g}^t$. When $t > 0$, the scheme calculates $\Delta^t$ as the difference
between the current and previous average gradients:

\[ \Delta^t \leftarrow \bar{g}^t - \bar{g}^{t-1}. \]

From the 20 most recent \( \Delta \) values, we obtain the mean \( \mu \) and standard deviation \( \sigma \):

\[ \mu \leftarrow \frac{1}{20} \sum_{u=t-20}^{t-1} \Delta^u, \]

\[ \sigma \leftarrow \sqrt{\frac{1}{20} \sum_{u=t-20}^{t-1} (|\Delta^u - \mu|^2).} \]

The scheme is configured by a threshold \( \alpha \) as a function of \( \sigma \). Reactive redundancy is triggered when \( \Delta^t \) exceeds \( \alpha \times \sigma \) in distance from the mean:

\[ \frac{|\mu - \Delta^t|}{\sigma} > \alpha. \]

We note that the effectiveness of a particular value of \( \alpha \) depends on the types of faults. We implement three different kinds of Byzantine faults described in the next section and discuss the behavior the targeted scheme given different fault types and values of \( \alpha \) in Chapter 4.
Chapter 4

Optimization Experiments

The intent of these experiments is to observe the effects of the proposed schemes compared to the Baseline algorithm. Each scheme is an extension of the Baseline: they do not fundamentally change the way a decision to check for a fault is made or the use of the $f$ fault-detection code. These schemes provide exact fault-tolerance independent of the learning model being used; that is, we can use them with a linear regression model or a neural network. The following sections first describe the experiment and data being used, then present and discuss the results of each proposed scheme.

4.1 Experimental Design

In order to compare and contrast their behavior, we use each of our proposed schemes for the same learning tasks and evaluate their performance. We perform linear regression on a randomly generated data set and train models for classification of MNIST images.

For linear regression, we use a randomly generated data set $Z$ containing 10,000 data points $[z_1, z_2, \ldots, z_{10000}]$. Each data point $z \in Z$ is a vector $z \in \mathbb{R}^{10}$ such that one index has a value of 1.0 and the remaining nine indexes contain random values in the range $(0, 1)$. $Y = w^*Z$ is the target value data such that each element $y_i = w^*z_i$, $i = 1, \ldots, 10000$. 
With the data points $X$ and the target values $Y$, we use SGD in the form of the Baseline or one of the proposed schemes to estimate $w^*$ by minimizing $\frac{1}{|Z|} \sum_{z \in Z} \ell(w, z)$.

MNIST images are publicly available and commonly used as benchmarks for performance of optimization techniques; though we do not seek to outperform previous methods by any particular metric, we use MNIST to observe whether scheme behavior changes from the linear regression task. The additional experiments also demonstrate that fault-tolerance behaviour in our schemes is model-independent.

4.1.1 Training

All of our schemes are different forms of parallelized-SGD, so model training looks the same from a high-level regardless of the task: the PS distributes data points and waits for gradients communicated from the workers, then performs an optimization step.

We start experiments with a Control instance—the Baseline algorithm executing without adversaries, meaning redundant fault-checking is not needed. For the Control instance only, $f = q = p = 0$, equivalent to PARALLELSGD with distributed gradient calculation [25]. We perform linear regression with the Control instance for 1000 iterations and record the performance; this tells us (i) a minimum loss/maximum accuracy to look for during training and (ii) how many steps of training are required without adversaries. We use (i) to define convergence for a given learning task.

After information is obtained from the Control instance, we use the Baseline algorithm and proposed schemes for the same learning task in the Byzantine setting. For these experiments, we consistently use $n = 10$ workers and set the number of Byzantine workers $f = 4$, the maximum tolerable amount. Since our schemes check for faults randomly, we observe the experimental results by averaging the performance of several training instances—linear regression experiments are performed 20
times each and MNIST experiments ten times each. For each instance, we record the loss evaluated from the full training set to obtain estimates $loss^t$ for all iterations $t$ and then average the estimates across all instances. We use these averages as our experimental results.

For example, if the Baseline is said to converge at iteration $t$ in an MNIST classification experiment, that means $\frac{1}{5}\sum_{i=1}^{5} loss^t_i$ was sufficiently low.

4.1.2 Byzantine behavior

At each step in training, Byzantine adversaries may communicate faulty gradients to the PS with probability $p$, held constant across experiments. In our experiments, Byzantine workers do not communicate with one another, so faults are not coordinated and are instead random. In any iteration, the set of gradients communicated from workers to the PS may be entirely honest or the data points assigned to all $f$ Byzantine workers could be faults.

Recall that the PS excludes Byzantine workers from future iterations, meaning an identified Byzantine worker cannot affect optimization thereafter. When a Byzantine worker communicates a faulty gradient, it risks being identified by the algorithm. Suppose an optimization algorithm is programmed to run for $T$ iterations and a Byzantine worker faults one time successfully. If the fault occurs early on, the algorithm may have nearly $T$ iterations to recover from the faults. If the fault occurs late in training, the algorithm may have few iterations of optimization to recover, leaving the optimization goal of poor quality.

Motivated by this notion, we delay Byzantine faults in optimization experiments until a time-step near the convergence of the Control instance.
4.1.3 Fault types

Byzantine faults alter the gradient communicated to the PS; we implement fault strategies that remain constant during individual experiments. We perform optimization experiments with reversal faults, scaled-reversal faults, and constant value faults. Reversal faults reverse the sign of each value in the gradient, essentially pointing it in the wrong direction. Formally, in a reversal fault a Byzantine worker communicates the faulty gradient $g_{\text{fault}}$ as

$$g_{\text{fault}} = -1.0 \times g,$$  

(4.1)

such that $g$ is the gradient(s) calculated from the worker’s assigned data points and the global loss function $\ell(w, z)$. We implement scaled-reversal faults by flipping the sign of each gradient value and multiplying them by 100. Formally, in a scaled reversal fault a Byzantine worker communicates the faulty gradient $g_{\text{faulty}}$ as

$$g_{\text{faulty}} = -100.0 \times g.$$  

(4.2)

For constant value faults, the Byzantine worker communicates a gradient vector in which every value is $-100.0$. Our discussion focuses primarily on experiments using scaled-reversal faults. For all schemes, with the exception of Targeted fault-checking, the fault-type does not affect the scheme’s ability to detect it. So, to maximally model the effectiveness of Byzantine workers, we focus on the most effective fault model. We discuss results of the targeted fault-checking scheme using all three fault types.

4.2 Linear Regression Results

The majority of comparable linear regression experiments were carried out with the following hyperparameter configurations:

- LR $\eta = .1$
• probability of fault-check $q = 0.2$

• probability of Byzantine fault $p = 0.05$

• number of total workers $n = 10$

• number of Byzantine workers $f = 4$

We also present results from additional experiments focusing on the behavior of different configurations of the same scheme, noting any alternate hyperparameters used. For example, we discuss results from the targeted fault-check scheme using five different values of $\alpha$ for all three fault types, and we initiate diminishing-LR experiments with a larger LR $\eta$.

To evaluate the quality of $w$ at each iteration, we define the estimated loss as the mean squared error (MSE) over all data points in the training set $Z$:

$$\text{loss} = MSE(w; Z; Y) = \frac{1}{|Z|} \sum_{i=1}^{|Z|} (wz_i - y_i)^2. \quad (4.3)$$

The convergence time refers to the number of optimization steps required to reach a sufficiently low value of loss. For linear regression experiments, we define convergence as the step when loss < .001. We also observe the effects on computation efficiency (Section 2.4, Definition 2) and identify any observable trends in convergence and efficiency. To observe the effect of the fault-check rate $q$ on efficiency, we compare 10 configurations of the BASELINE algorithm with $q$ ranging $[0.1, 1.0]$ (in increments of 0.1).

We observe the convergence time for all schemes compared to the BASELINE and the computation efficiency of the Adaptive-$q$ and Targeted fault-checking schemes.
4.2.1 Results and discussion

For linear regression experiments, we define convergence as the step when $loss < .001$. The fault-free Control training instance achieved $loss < .001$ at step 145. We delay Byzantine faults until step 50 or later; the intention is to skew faults later in training when they more effectively delay convergence and illustrate our schemes’ behaviors.

<table>
<thead>
<tr>
<th>experiment</th>
<th>convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control ($f = 0$)</td>
<td>145</td>
</tr>
<tr>
<td>Baseline ($f = 1$)</td>
<td>178</td>
</tr>
<tr>
<td>Baseline ($f = 4$)</td>
<td>817</td>
</tr>
</tbody>
</table>

We list the convergence results of the Control and the Baseline algorithm with $f = 1$ and $f = 4$; all results are averaged from 20 experiments each. The Baseline algorithm checks for faults with probability $q = .2$, and Byzantine workers send faults with probability $p = .05$. The main observation we make is that a single Byzantine fault does not significantly delay convergence of ReactiveRedundantSGD, as the lone Byzantine worker is often caught quickly.

Using $f = 1$, the Baseline converged in 178 iterations on average—the Byzantine worker caused a delay of only 33 iterations compared to the average fault-free instances. The use of $f = 4$ Byzantine agents delays the average convergence time to 817 iterations, plotted in Figure 4.1; we list all three convergence results in Table 4.1. The longer delay, of course, is a result of successful Byzantine faults that cause parameter updates in the wrong direction.

To better illustrate the effects of faults throughout training, we plot the loss over time of one random execution of the Baseline with $f = 4$ compared to one random fault-free Control instance in Figure 4.2.

Until all Byzantine workers are identified, a non-redundant step has a computation efficiency of 1.0 and a redundant step has a computation efficiency in the range $[\frac{1}{2f+1}, \frac{1}{f+1}]$. In a redundant step in which no fault is found, computation efficiency
Figure 4.1: Average BASELINE and fault-free linear regression. We plot the loss over time of the BASELINE algorithm with $f = 4$ Byzantine workers against the fault-free Control case; for each, results are averaged over 20 optimization experiments. $f = 4$, $p = .05$, and $q = .2$ for the BASELINE model; for the Control, $f = 1 = p = 0$. Byzantine workers do not send faults until at least iteration 50; the loss at each iteration $i$ is the average of the loss values at iterations $[i, i + 10]$ for smoother lines. Both images above plot the same results; the bottom image only plots values up to 1 to make the orange line visible, while the top plots values up to 60. The orange line represents the fault-free Control, which converges in 145 iterations on average. The BASELINE algorithm convergences in 817 iterations on average. Convergence times are listed in parentheses in the legend.
Figure 4.2: Baseline linear regression example. We plot the loss over time of one instance of the Baseline algorithm with $f = 4$ Byzantine workers against a single fault-free Control instance. $f = 4$, $p = .05$, and $q = .2$ for the Baseline model; for the Control, $f = 1 = p = 0$. Byzantine workers do not send faults until at least iteration 50; for readability, the plotted loss at each iteration $i$ is the average of the loss values at iterations $[i, i+10]$. Both images plot the same experimental results; the bottom only includes values up to 1 for visibility of the orange line representing the fault-free Control. The top image includes values up to 60, illustrating the spikes in loss when Byzantine workers are present. The fault-free Control instance converges in 146 iterations; in the presence of $f = 4$ Byzantine workers, the Baseline algorithm converged at iteration 573. Convergence times appear between parentheses in the legend.
is $\frac{1}{f+1}$; conversely, if every data point is associated with a fault, the computation efficiency is $\frac{1}{2f+1}$. After the Byzantine workers are identified, the efficiency for the remaining iterations is 1.0. Figure 4.3 plots the computation efficiencies of the BASELINE algorithm using $q = [.1, .2, \ldots, 1.0]$.

![Figure 4.3: BASELINE computation efficiency for different fault-check probabilities.](image)

We plot the computation efficiency at the point of convergence of the BASELINE algorithm; $n = 10$, $f = 4$, and $p = .05$, and for each value of $q$ we average the results of 20 experiments. As the value of $q$ increases, redundant steps become more frequent and the efficiency of the algorithm degrades; a non-redundant step has computation efficiency of 1.0, while a redundant step has computation efficiency between $[\frac{1}{2f+1}, \frac{1}{f+1}]$. Throughout execution, as Byzantine workers are caught, $f$ and $n$ change, and after all Byzantine workers are caught, the computation efficiency is 1.0 for every step; thus, average $ce$ is above the range $[\frac{1}{5}, \frac{1}{5}]$.

While less efficient on average for individual steps, a higher probability of fault-checks results in faster convergence, shown in Table 4.2. As $q$ gets closer to 1.0, the BASELINE algorithm identifies the last Byzantine worker earlier, converges faster, and therefore requires less total gradients. Clearly, an algorithm converging faster and requiring fewer total calculations is more desirable.

Next, we evaluate linear regression experiments using our proposed schemes.
### Table 4.2: Baseline computation efficiency

<table>
<thead>
<tr>
<th>$q$</th>
<th>convergence</th>
<th>used gradients</th>
<th>total gradients</th>
<th>computation efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>909</td>
<td>9090</td>
<td>9932</td>
<td>.9151</td>
</tr>
<tr>
<td>0.2</td>
<td>817</td>
<td>8180</td>
<td>9392</td>
<td>.8709</td>
</tr>
<tr>
<td>0.3</td>
<td>421</td>
<td>4210</td>
<td>5643</td>
<td>.7460</td>
</tr>
<tr>
<td>0.4</td>
<td>485</td>
<td>4850</td>
<td>6760</td>
<td>.7175</td>
</tr>
<tr>
<td>0.5</td>
<td>317</td>
<td>3170</td>
<td>4938</td>
<td>.6419</td>
</tr>
<tr>
<td>0.6</td>
<td>211</td>
<td>2110</td>
<td>4662</td>
<td>.4526</td>
</tr>
<tr>
<td>0.7</td>
<td>144</td>
<td>1440</td>
<td>3413</td>
<td>.4218</td>
</tr>
<tr>
<td>0.8</td>
<td>148</td>
<td>1480</td>
<td>3956</td>
<td>.3741</td>
</tr>
<tr>
<td>0.9</td>
<td>141</td>
<td>1410</td>
<td>3680</td>
<td>.3831</td>
</tr>
<tr>
<td>1.0</td>
<td>144</td>
<td>1440</td>
<td>4318</td>
<td>.3335</td>
</tr>
</tbody>
</table>

We list the convergence and computation efficiency results of the Baseline algorithm given ten different values of $q$. For each $q$, we average the results of 20 experiments; $f = 4, p = .05$, and Byzantine workers use a scaled-reversal fault. The two lowest $q$ values require over twice as many iterations to converge as the third lowest $q$, and the four largest values result in nearly identical convergence times. We also note that computation efficiency always increases when $q$ is reduced.

### Diminishing-LR

We present results and observations of the diminishing-LR scheme used for linear regression. Three kinds of LR schedules were implemented: linear, exponential, and inverse schedules, described in Chapter 3. For experiments using the diminishing-LR scheme, we set the initial LR as $\eta = .3$. Without a lower bound on the schedules, the LR can become so small that the model’s estimated losses plateau pre-convergence. Enforcing a minimum LR $\eta^\text{min} = .03$ for the linear regression experiments allowed convergence in many individual instances, but nearly all configurations of this scheme delayed convergence compared to the Baseline.
The linear schedule reduces the LR $\eta$ every $\gamma$ iterations by a constant value until it reaches $\eta^{\min}$; we list the convergence and computation efficiency results in Table 4.3. The schedule with $\gamma = 50$ was the only configuration to converge in less than 1000 iterations, but it was still slower than BASELINE. None of the exponential or inverse schedules resulted in convergence within 1000 iterations in experiments using scaled-reversal faults, while the BASELINE algorithm with constant LR converged at iteration 817. With a larger LR, the BASELINE algorithm is able to make more progress optimizing $w$ in non-faulty steps.

Table 4.3: Example of diminishing-LR scheme for linear regression

<table>
<thead>
<tr>
<th>$\gamma$</th>
<th>convergence</th>
<th>total gradients</th>
<th>computation efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>—</td>
<td>1123.15</td>
<td>.8990</td>
</tr>
<tr>
<td>50</td>
<td>825</td>
<td>9348.45</td>
<td>.8825</td>
</tr>
<tr>
<td>100</td>
<td>—</td>
<td>11020.45</td>
<td>.9074</td>
</tr>
</tbody>
</table>

We list the convergence and efficiency results of the diminishing-LR scheme with different values of $\gamma$; results are averaged from 20 instances of the configuration. $f = 4$, $q = .2$, $p = .05$, initial $\eta = 0.3$, and Byzantine workers use a scaled-reversal fault. Only the configuration with $\gamma = 50$ converged within 1000 iterations, and it did not converge faster than the BASELINE. The goal of the diminishing-LR scheme is to reduce the negative effects of faulty gradients in non-redundant iterations, but it also serves to reduce the effects of honest gradients throughout optimization, delaying convergence.

The goal of the diminishing-LR scheme is to reduce the effects of faulty gradients of $w$, but the diminishing-LR also reduces the effects of honest gradients on $w$. For this data, the latter effect outweighs the former, resulting in slower optimization. These results suggest that the diminishing-LR scheme does not effectively improve the BASELINE method.
Adaptive-\(q\)

Increasing the fault-check probability \(q\) over time should reduce the number of uncaught faults compared to the BASELINE. We know that as the value of \(q\) increases, more steps use redundant calculations and the expected computational efficiency degrades. We also know that using the BASELINE algorithm with a higher \(q\) results in faster convergence. Our scheme is configured with an interval \(\gamma\), a maximum value \(q^{\text{max}}\), and the last step of adaptation \(S\).

We note here that all adaptive-\(q\) experiments use \(S = 1000\), as experiments are only run for 1000 iterations, and \(\gamma = 25\). Holding these two parameters of the scheme constant, we perform experiments with different rates of adaptation by changing \(q^{\text{max}}\).

As a first example, we plot the loss over time of the BASELINE algorithm and the adaptive-\(q\) scheme configured with \(\gamma = 10\) and \(q^{\text{max}} = 1.0\) in Figure 4.4. This rate of adaptation results in \(q = \frac{1-2}{2} = .4\) at iteration 500. Averaging the results of 20 experiments each, we see that the adaptive-\(q\) scheme reduces convergence time to 581 iterations compared to the BASELINE.

Across optimization experiments, we find that the adaptive-\(q\) scheme consistently results in faster convergence compared to the BASELINE. Table 4.4 lists the convergence and efficiency results of the adaptive-\(q\) scheme for five different configurations.

The configurations with faster rates of adaptation (higher \(q^{\text{max}}\)) consistently result in faster convergence and earlier identification of the Byzantine workers. All configurations of the scheme reduced convergence time and resulted in higher computational efficiency compared to the BASELINE. Even though the scheme causes more fault-checks, which are less efficient than a non-redundant iteration, redundancy is unnecessary after \(f\) Byzantine workers have been identified. The earlier Byzantine workers
Figure 4.4: Adaptive fault-checking compared to Baseline algorithm for linear regression. We plot the loss over time of the Adaptive-q scheme with $\gamma=25$, $S=1000$ and $q^{\text{max}}=1.0$ against the Baseline; $f=4$, initial $q=.2$, $p=.05$, and Byzantine workers use scaled-reversal fault type. Increasing the parameter $q$ at this rate results in an average reduction in training time from 817 to 581 iterations. The adaptive-q scheme checks for faults more frequently over time, identifying the Byzantine workers earlier. The two schemes use the same loss function and LR, resulting in almost identical shape of their respective lines in the final steps before convergence. Convergence times appear between parentheses in the legend. We only plot the loss values up to 60 for consistency with previous figures.
Table 4.4: Adaptive fault-checking linear regression

<table>
<thead>
<tr>
<th>$q^{\text{max}}$</th>
<th>convergence</th>
<th>computation efficiency</th>
<th>$f^{th}$ fault caught</th>
</tr>
</thead>
<tbody>
<tr>
<td>.4</td>
<td>683</td>
<td>.9304</td>
<td>253</td>
</tr>
<tr>
<td>.6</td>
<td>667</td>
<td>.9368</td>
<td>194</td>
</tr>
<tr>
<td>.8</td>
<td>633</td>
<td>.9418</td>
<td>155</td>
</tr>
<tr>
<td>1.0</td>
<td>581</td>
<td>.9249</td>
<td>146</td>
</tr>
<tr>
<td><strong>Baseline</strong></td>
<td>817</td>
<td>.8709</td>
<td>278</td>
</tr>
</tbody>
</table>

We list the convergence and computation efficiency results for the adaptive-$q$ scheme configured with different rates of adaptation. These results are averaged from 20 linear regression experiments each; $f = 4$, initial $q = .2$, $p = .05$, and Byzantine workers use a scaled-reversal fault. Computation efficiency is calculated as Definition 2. All schemes adapt $q$ from .2 to $q^{\text{max}}$ with $\gamma = 25$ over 1000 iterations; at iteration 500, $q = \frac{q^{\text{max}} - 0.2}{2}$. All configurations resulted in faster convergence compared to the Baseline algorithm and the schemes with higher $q^{\text{max}}$, which increase $q$ with greater magnitude, identify Byzantine workers earlier.

are all identified, the earlier the algorithm can utilize every computed gradient for optimization.

ROLLBACK RECOVERY

The rollback scheme is configured by the interval $\gamma$ and performs evaluations of the model on a validation set for the rollback procedure. For the rollback checkpoint, the parameter vector $w$ must be evaluated on a validation set to obtain an estimate of the performance for each rollback. For linear regression, we perform validation by obtaining the MSE as in Equation 4.3, using the full training set $Z$.

We carry out experiments with five different rollback intervals: $\gamma = 10, 25, 50, 100, 200$. Convergence and efficiency results are listed in Table 4.5. Compared to the Baseline, the rollback scheme significantly speeds up convergence regardless of the interval $\gamma$. 

29
Table 4.5: Rollback recovery linear regression

<table>
<thead>
<tr>
<th>$\gamma$</th>
<th>convergence</th>
<th>computation efficiency</th>
<th>rollbacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>208</td>
<td>0.6698</td>
<td>6.3</td>
</tr>
<tr>
<td>25</td>
<td>303</td>
<td>0.7314</td>
<td>4.05</td>
</tr>
<tr>
<td>50</td>
<td>311</td>
<td>0.7235</td>
<td>3.1</td>
</tr>
<tr>
<td>100</td>
<td>396</td>
<td>0.7833</td>
<td>1.6</td>
</tr>
<tr>
<td>200</td>
<td>464</td>
<td>0.7848</td>
<td>0.65</td>
</tr>
</tbody>
</table>

For different rollback intervals $\gamma$, we list the convergence times, overall computation efficiency, and average number of rollbacks. Results are averaged from 20 experiments for each $\gamma$ configuration. Every $\gamma$ iterations, the rollback scheme compares the estimated loss of the rollback state $\text{loss}_{\text{roll}}$ and the current state $\text{loss}_{\text{w}}$. If $\text{loss}_{\text{w}} > \text{loss}_{\text{roll}}$, the scheme assumes there was at least one fault in the last $\gamma$ iterations and loads the rollback state; loading a rollback state improves the parameter estimate $w$, but it also means that all optimization steps since the state was saved were wasted. When $\gamma$ is smaller, the window for Byzantine workers to attack before the scheme loads a rollback state is also smaller. With a larger $\gamma$, it is more likely for Byzantine workers to attack and cause a rollback, wasting more optimization steps. More frequent rollbacks correspond to quicker convergence, though all configurations reduce the training time from the BASELINE algorithm.

Previous schemes did not converge until after all Byzantine workers were found because even a single Byzantine worker can thwart optimization indefinitely; in previous schemes, the model simply recovers from faults through SGD. With the Rollback scheme, the model can recover its rollback state in a single iteration $i$ (provided that $i$ is a multiple of $\gamma$).

In these experiments, smaller intervals result in faster convergence times. This result follows from the nature of the rollback state— every $\gamma$ iterations, the scheme compares the estimated loss of the rollback state $\text{loss}_{\text{roll}}$ and the current state $\text{loss}_{\text{w}}$. If $\text{loss}_{\text{w}} > \text{loss}_{\text{roll}}$, the scheme loads the rollback state; loading a rollback state allows instant recovery, but also means that every optimization step since the rollback has
been wasted. When $\gamma$ is smaller, Byzantine workers have a smaller window in which to attack, resulting in less wasted optimization steps.

We must consider the amount of work this scheme adds to the BASELINE. Given $\gamma = 10$, the scheme performs 20 validations on average to converge at iteration 206—the BASELINE required nearly four-times as many iterations. Given $\gamma = 200$, the scheme converged at step 464 and only required two rollback validations; compared to the BASELINE, convergence time is reduced by nearly half. In our experiments, the benefit of the rollback scheme outweighs its cost.

TARGETED FAULT-CHECKING

The targeted fault-checking scheme augments the BASELINE algorithm with lightweight filtering in non-redundant steps such that the filter triggers reactive redundancy. In iterations $t > 20$, the scheme initiates reactive redundancy if a norm change $\Delta^t$ is “too large” compared to the 20 previous changes \{\Delta^{t-20}, \Delta^{t-19}, \ldots, \Delta^{t-1}\}.

We use $\alpha$ to denote the distance from the mean norm change $\mu$ that qualifies as a fault; specifically, $\alpha$ is a distance in units of the standard deviation $\sigma$ of the norm changes.

We evaluate experiments with three different types of Byzantine fault: reversal faults, scaled-reversal faults, and constant faults, described in Section 4.1.3. Since this scheme relies on the relative size of gradients to identify faults, the effectiveness of a particular value of $\alpha$ relies in part on the fault type. In addition to the convergence of each scheme for each fault type, we list statistics averaged across the 20 instances of each configuration: “true targeted checks” is the targeted fault-checks triggered that identified a fault, “false targeted checks” is the number of fault-checks triggered that did not identify a fault, and “un-targeted faults” is the number of Byzantine faults that did not trigger a targeted fault-check.

31
Table 4.6: Targeted fault-checking linear regression (scaled-reversal faults)

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>true targeted checks</th>
<th>false targeted checks</th>
<th>un-targeted faults</th>
<th>convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.05</td>
<td>2.75</td>
<td>37.15</td>
<td>0.35</td>
<td>145</td>
</tr>
<tr>
<td>1.0</td>
<td>2.5</td>
<td>14.25</td>
<td>1.35</td>
<td>153</td>
</tr>
<tr>
<td>3.0</td>
<td>2.4</td>
<td>1.45</td>
<td>1.85</td>
<td>166</td>
</tr>
<tr>
<td>5.0</td>
<td>2.35</td>
<td>0.35</td>
<td>2.8</td>
<td>179</td>
</tr>
<tr>
<td>10.0</td>
<td>1.95</td>
<td>0.0</td>
<td>5.5</td>
<td>273</td>
</tr>
<tr>
<td>20.0</td>
<td>1.1</td>
<td>0.0</td>
<td>9.0</td>
<td>509</td>
</tr>
</tbody>
</table>

We list selected statistics from experiments using the targeted fault-check scheme with different values of $\alpha$. These results are averaged from 20 linear regression experiments each; $f = 4$, $q = 0.2$, $p = 0.05$. This fault type makes a large augmentation to the honest gradient, and as a result the three smaller $\alpha$-values result in less than two faults un-targeted; they do, however, result in many fault-checks in steps where all gradients are honest (false targeted checks column).

First we present the results of experiments using scaled-reversal faults and target thresholds of $\alpha = 0.5, 1, 3, 5, 10, 20$. Results are listed in Table 4.6, and we observe that all six configurations of the scheme reduce convergence time compared to the BASELINE. The proposed targeted fault-checking scheme is very effective in experiments using scaled-reversal faults; even the largest threshold values $\alpha = 10, 20$ successfully targeted faulty gradients.

These results all follow logically from the scheme’s behavior: smaller thresholds result in fewer successful Byzantine faults at the cost of more false targeted-checks. The two largest thresholds for targeting result in zero false targeted checks, but converge later and allow more successful Byzantine faults.

As stated in the beginning of this section, we must also consider constant faults and un-scaled reversal faults. Since these types of faults augment gradients by a smaller magnitude, different values of $\alpha$ may be needed; we expected to see the smaller values
We list selected statistics from experiments using the targeted fault-check scheme with different values of \( \alpha \). These results are from linear regression experiments; \( f = 4 \), \( q = .2 \), \( p = .05 \). We see results very similar to experiments using scaled-reversal faults. We suspect this is because one fault multiplies gradients by a magnitude of -100 while the other communicates a constant value of 100 for all elements in the gradient vectors, resulting in similar faulty gradients.

In Table 4.8 we list the results of experiments using un-scaled reversal faults. These faults fail to delay convergence in linear regression experiments, but we observe the behavior of the targeted fault-check scheme regardless. The reversal fault type did not corrupt the optimization goal \( w \) enough to affect the convergence of linear regression, but other data may be vulnerable to the attack.
Table 4.8: Targeted fault-checking linear regression (reversal faults)

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>true targeted checks</th>
<th>false targeted checks</th>
<th>un-targeted faults</th>
<th>convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>2.35</td>
<td>44.05</td>
<td>1.35</td>
<td>144</td>
</tr>
<tr>
<td>1.0</td>
<td>2.0</td>
<td>26.6</td>
<td>3.95</td>
<td>145</td>
</tr>
<tr>
<td>3.0</td>
<td>0.1</td>
<td>2.5</td>
<td>23.25</td>
<td>148</td>
</tr>
<tr>
<td>5.0</td>
<td>0.0</td>
<td>0.1</td>
<td>17.85</td>
<td>146</td>
</tr>
<tr>
<td>10.0</td>
<td>0.0</td>
<td>0.0</td>
<td>21.1</td>
<td>146</td>
</tr>
<tr>
<td>20.0</td>
<td>0.0</td>
<td>0.0</td>
<td>30.1</td>
<td>145</td>
</tr>
</tbody>
</table>

We list selected statistics from experiments using the targeted fault-check scheme with different values of $\alpha$. These results are from linear regression experiments; $f = 4$, $q = .2$, $p = .05$. Reversal faults do not appear to delay convergence for these linear regression experiments; reversing the signs of the values in a gradient does not affect the parameter vector $w$ enough to impede optimization. Regardless, we observe that the two smaller $\alpha$ values result in effective targeting: each averaged two of four Byzantine agents identified by a targeted fault-check and fewer than four faults that were not targeted by the algorithm. As expected, $\alpha = 5, 10, 20$ did not successfully identify any Byzantine workers via targeted fault-check; these are very large threshold values and the reversal fault type does not increase the magnitude of the gradient.

We see that the three largest thresholds $\alpha = 5, 10, 20$ do not trigger any true targeted checks ($\alpha = 3$ only .1 on average) and allow many successful Byzantine faults. The smallest thresholds $\alpha = 0.5, 1.0$, however, appear to have a significant effect: both configurations result in less than four successful faults on average in an experiment and identify at least two of the four Byzantine workers with a targeted fault-check.

4.2.2 Summary

Through linear regression experiments, we observe faster convergence in three of our four proposed schemes. The Rollback and Targeted fault-check schemes in particular can significantly improve convergence time; this is conditional on the value of $\alpha$ used.
by the Targeted scheme and the quality of the Rollback scheme’s evaluation of the parameter vector $w$.

Only the Diminishing-LR scheme resulted in overall worse performance as a result of slower learning. Next, we evaluate the results of optimization experiments using a more complex set of data and neural network as the learning model.

4.3 MNIST Classification Results

For MNIST experiments, the Baseline algorithm and our proposed schemes are used to perform parallelized-SGD, optimizing a neural network. We use a simple network comprised of one $784 \times 500$ input layer, a rectified linear activation function, one $500 \times 10$ hidden layer, and a sigmoid output. The neural network takes the place of the parameter vector $w$ in Equation 2.2. We denote the model’s classification of a data point $z$ as $f(w, z)$, an integer value in the range $[0, 9]$; the actual output of the model is a vector of 10 values in the range $[0, 1]$, and the index of the largest value is the model’s classification.

For MNIST experiments, we define convergence by the accuracy of the model at a particular step. Let $c(w; z; y)$ be a function that returns 1 if the model classifies a data point $z$ correctly and 0 otherwise:

$$c(w; z; y) = \begin{cases} 1 & \text{if } f(w, z) == y \\ 0 & \text{else} \end{cases} \tag{4.4}$$

$$acc = \frac{1}{M} \sum_{i=0}^{M} c(w; z_i; y_i), \tag{4.5}$$

for a test set of $M$ samples, where $w$ represents the neural network. Our measure of accuracy, denoted $acc \in [0, 1]$ in Equation 4.5, is the ratio of correct classifications to total samples.
Again, all comparable experiments are configured with the same hyperparameters:

- LR $\eta = 0.03$
- probability of fault-check $q = 0.2$
- probability of Byzantine fault $p = 0.05$
- number of total workers $n = 10$
- number of Byzantine workers $f = 4$

Next, we present results from MNIST classification experiments. The remainder of this section focuses on whether trends observed in linear regression experiments are mirrored in MNIST classification.

### 4.3.1 Results and Analysis

Due to the increased size of MNIST data, results for a given scheme are averaged over ten experiments. Again, we record the results of a fault-free Control instance of the model to determine the point of convergence. For MNIST experiments, we define convergence as the iteration when $acc \geq 0.71$. The fault-free Control model converged to this point at iteration 69 on average across 10 experiments.

The BASELINE algorithm with $f = 4$ Byzantine workers using scaled-reversal faults converged at iteration 877, depicted in Figure 4.5. We perform optimization experiments over the MNIST data set with each of our proposed schemes using the scaled-reversal fault type. The constant fault-type results in such large gradients that the network used for MNIST classification never converged within 1000 iterations, and the unscaled reversal-faults did not affect convergence significantly; we do list results from the targeted fault-checking scheme using these two fault-types. Experiments delay faults until iteration 50 or later.
**Figure 4.5:** Baseline MNIST classification accuracy. We plot the accuracy over time of the Baseline algorithm compared to the fault-free control instance; results are averaged over 10 experiments and the plotted accuracy at each iteration \( i \) is the average of the loss values at iterations \([i, i + 10]\). \( f = 4, \ p = .05 \), and \( q = .2 \) for the Baseline model; for the Control, \( f = 1 = p = 0 \). Faults are delayed until at least iteration 50. After iteration 50, the accuracy of the Baseline drops, converging at iteration 877, shown in parentheses in the legend. The fault-free Control converges at iteration 69.

**Table 4.9:** Baseline MNIST classification results

<table>
<thead>
<tr>
<th>scheme</th>
<th>convergence</th>
<th>computation efficiency</th>
<th>( f^{th} ) fault caught</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control</td>
<td>69</td>
<td>1.0</td>
<td>—</td>
</tr>
<tr>
<td>Baseline</td>
<td>877</td>
<td>.84</td>
<td>216</td>
</tr>
</tbody>
</table>

We list results from experiments on the MNIST data set comparing the Baseline algorithm to the Control. Byzantine workers use scaled-reversal faults, \( f = 4, \ n = 10, \ p = .05 \), and the fault-check probability \( q = 0.2 \) for the Baseline. The control instance has perfect computation efficiency because it is a fault-free experiment.
All four schemes exhibited behaviors consistent with those observed in previous linear regression experiments; figures and tables listing the results follow. In Table 4.9, we list results from the Control and Baseline.

**Diminishing-LR**

We tested the linear diminishing-LR scheme with Byzantine workers using scaled-reversal faults and saw the same results as the linear regression experiments. The reduced LR serves to slow the optimization process more than it reduces the affects of Byzantine faults.

![Figure 4.6: Diminishing-LR compared to Baseline for MNIST classification.](image)

We plot the accuracy over time of the diminishing-LR scheme compared to the Baseline; $\gamma = 10$ and $\eta_{\text{min}} = .001$ for our scheme. For readability, the plotted accuracy at each iteration $i$ is the average of the loss values at iterations $[i, i + 10]$. $f = 4$, $q = .2$, $p = .05$, initial $\eta = 0.03$, and Byzantine workers use a scaled-reversal fault. The reduced effects of honest gradients in updates to the network caused by our scheme delays convergence compared to the Baseline algorithm, effectively plateauing around 65% accuracy; we saw the same trend in linear regression experiments. The Baseline scheme converged in 877 iterations, shown in parentheses in the legend.
Just as we observed in most linear regression experiments, the accuracy plateaus and the algorithm fails to converge within 1000 iterations. All schemes use the same initial $q$ value, and there was no significant difference in the time required to identify all $f$ Byzantine workers. Our choice of $\eta^\text{min}$ may play a role in the plateau of the diminishing-LR scheme visible in Figure 4.6. Regardless of the plateau, the scheme did not seem to improve performance while Byzantine workers were active.

**Adaptive-$q$**

We performed MNIST experiments using the adaptive-$q$ scheme with $q^{\text{max}} = 0.6, 0.8, 1.0$ and $S = 1000$. We observed behavior consistent with the same scheme used for linear regression experiments. In both cases, the increase in $q$ results in faster identification of the Byzantine workers, faster convergence, and higher computation efficiency. We plot the accuracy over time of the adaptive-$q$ scheme increasing $q$ from .2 to 1.0 over $S = 1000$ iterations in Figure 4.7 as an example. All remaining adaptive-$q$ experimental results are listed in Table 4.10.

In contrast to linear regression experiments, we do not see monotonic relationship between the faster-increasing $q$ and faster identification of all Byzantine agents. The configuration with $q^{\text{max}} = 0.8$ converges quicker than $q^{\text{max}} = 1.0$, but the latter scheme identifies faults quicker as expected. We suspect this results from the fact that the accuracy value used to measure convergence in MNIST experiments is more volatile than the MSE used in linear regression experiments.

**Rollback recovery**

MNIST experimental results from the rollback scheme are also consistent with linear regression experiments. The rollback scheme with intervals $\gamma = 25, 50, 100, 200$ all result in significantly faster convergence compared to the BASELINE. Even though all
Figure 4.7: Adaptive fault-checking compared to Baseline for MNIST classification. We plot the estimated accuracy at each iteration for the Baseline and one configuration of the adaptive-\( q \) scheme. These results are averaged from 10 MNIST classification experiments each; \( f = 4 \), initial \( q = .2 \), \( p = .05 \), \( \eta = .03 \) and Byzantine workers use a scaled-reversal fault. The adaptive scheme adapts \( q \) from \(.2\) to \( q_{\text{max}} = 1.0 \) at a constant rate over \( S = 1000 \) iterations; at iteration 500, \( q = .6 \). The adaptive scheme results in faster identification of the Byzantine workers and earlier convergence.

Table 4.10: Adaptive fault-checking MNIST classification results

<table>
<thead>
<tr>
<th>( q_{\text{max}} )</th>
<th>convergence</th>
<th>computation efficiency</th>
<th>( f^{th} ) fault caught</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>877</td>
<td>.8766</td>
<td>216</td>
</tr>
<tr>
<td>.6</td>
<td>605</td>
<td>.8355</td>
<td>193</td>
</tr>
<tr>
<td>.8</td>
<td>551</td>
<td>.8087</td>
<td>185</td>
</tr>
<tr>
<td>1.0</td>
<td>644</td>
<td>.8548</td>
<td>183</td>
</tr>
</tbody>
</table>

We list the convergence and computation efficiency results for the adaptive-\( q \) scheme configured with different rates of adaptation. These results are averaged from 10 MNIST classification experiments each; \( f = 4 \), initial \( q = .2 \), \( p = .05 \), \( \eta = .03 \) and Byzantine workers use a scaled-reversal fault. Schemes adapt \( q \) from \(.2\) to \( q_{\text{max}} \) at a constant rate over 1000 iterations; at iteration 500, \( q = \frac{q_{\text{max}} - .2}{2} \). All configurations resulted in faster convergence compared to the Baseline algorithm. We do not see a monotonic relationship between the rate of adaptation and convergence, but faster rates always resulted in faster identification of the Byzantine workers.
Table 4.11: Rollback recovery MNIST classification results

<table>
<thead>
<tr>
<th>γ</th>
<th>convergence</th>
<th>computation efficiency</th>
<th>f&lt;sup&gt;th&lt;/sup&gt; fault caught</th>
<th>rollbacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>201</td>
<td>.6400</td>
<td>255</td>
<td>4.3</td>
</tr>
<tr>
<td>50</td>
<td>250</td>
<td>.7037</td>
<td>215</td>
<td>3.4</td>
</tr>
<tr>
<td>100</td>
<td>397</td>
<td>.7790</td>
<td>256</td>
<td>2.8</td>
</tr>
<tr>
<td>200</td>
<td>540</td>
<td>.8059</td>
<td>353</td>
<td>2.3</td>
</tr>
</tbody>
</table>

For different rollback intervals γ, we list convergence, computation efficiency, and average number of rollbacks. Results are averaged from 10 experiments for configuration. Every γ iterations, the rollback scheme compares the estimated loss of the rollback state loss<sub>roll</sub> and the current state loss<sub>w</sub>. More frequent rollbacks correspond to quicker convergence, though all configurations reduce the training time from the BASELINE algorithm. In these experiments, smaller γ always results in faster convergence, and some configurations converged before all Byzantine workers were identified. Configurations used the same value of q, the time required to identify all Byzantine workers varied significantly. Regardless, convergence of the rollback scheme appears much more dependent on γ in the results listed in Table 4.11.

We note again that the rollback schemes often converged before all f Byzantine workers were identified, which also occurred in linear regression experiments using the rollback scheme. No other scheme converged prior to identification of all f Byzantine workers.

**TARGETED FAULT-CHECKING**

Finally, we observe behavior of the targeted fault-checking scheme to be consistent with linear regression experiments. The targeted scheme resulted in reduced convergence time and earlier identification of Byzantine workers. Results from four different threshold values in experiments using scaled-reversal faults are listed in Table 4.12.
Table 4.12: Targeted fault-checking MNIST classification (scaled-reversal faults)

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>true targeted checks</th>
<th>false targeted checks</th>
<th>un-targeted faults</th>
<th>convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>3.78</td>
<td>33.89</td>
<td>0.00</td>
<td>70</td>
</tr>
<tr>
<td>1.0</td>
<td>2.56</td>
<td>14.56</td>
<td>0.44</td>
<td>77</td>
</tr>
<tr>
<td>3.0</td>
<td>2.67</td>
<td>3.89</td>
<td>2.22</td>
<td>180</td>
</tr>
<tr>
<td>5.0</td>
<td>2.33</td>
<td>0.44</td>
<td>3.00</td>
<td>275</td>
</tr>
</tbody>
</table>

We list selected statistics from MNIST experiments using the targeted fault-check scheme with different values of $\alpha$: $f = 4$, $q = .2$, $p = .05$. True targeted checks are occurrences of a targeted fault-check when a fault was present, false targeted checks are targeted fault-checks triggered when all gradients are honest, and un-targeted faults are faults that did not trigger a fault-check. As expected, the smallest threshold results in a fault-check during any iteration where one occurs, but also results in many false targeted checks. Here, larger thresholds consistently correlate to less true and false targeted checks, more un-targeted faults, and slower convergence. Given any threshold depicted in the table, this scheme converges significantly faster than the Baseline.

Additionally, we list the results of the targeted fault-checking scheme for constant faults in Table 4.13 and for un-scaled reversal faults in Table 4.14. As stated at the beginning of this section, the constant faults resulted in convergence times over 1000 for the Baseline and the un-scaled reversal faults did not affect convergence significantly. We note two observations from these results: contrary to the Baseline, the targeted scheme was able to converge in $i < 1000$ iterations in the presence of constant faults; and, we were able to find thresholds that worked for un-scaled reversal faults without tuning.
Table 4.13: Targeted fault-checking MNIST classification (constant faults)

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>true targeted checks</th>
<th>false targeted checks</th>
<th>un-targeted faults</th>
<th>convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.05</td>
<td>2.89</td>
<td>29.77</td>
<td>0.22</td>
<td>299</td>
</tr>
<tr>
<td>1.0</td>
<td>3.67</td>
<td>15.89</td>
<td>0.22</td>
<td>293</td>
</tr>
<tr>
<td>3.0</td>
<td>2.67</td>
<td>3.56</td>
<td>2.11</td>
<td>–</td>
</tr>
<tr>
<td>5.0</td>
<td>2.56</td>
<td>0.56</td>
<td>3.0</td>
<td>–</td>
</tr>
</tbody>
</table>

We list selected statistics from MNIST experiments using the targeted fault-check scheme with different values of $\alpha$; $f = 4$, $q = .2$, $p = .05$. We note that in MNIST experiments, constant type faults delay convergence for longer than 1000 iterations given all other schemes unless $q = 1.0$. The two smaller thresholds result in faster convergence, while the the larger thresholds allow enough faulty gradients to delay convergence past 1000 iterations.

Table 4.14: Targeted fault-checking MNIST classification (reversal faults)

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>true targeted checks</th>
<th>false targeted checks</th>
<th>un-targeted faults</th>
<th>convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>2.56</td>
<td>32.89</td>
<td>1.00</td>
<td>63</td>
</tr>
<tr>
<td>1.00</td>
<td>2.22</td>
<td>43.67</td>
<td>4.33</td>
<td>68</td>
</tr>
<tr>
<td>3.00</td>
<td>0.56</td>
<td>4.89</td>
<td>13.67</td>
<td>73</td>
</tr>
<tr>
<td>5.00</td>
<td>0.00</td>
<td>0.78</td>
<td>14.78</td>
<td>74</td>
</tr>
</tbody>
</table>

We list selected statistics from MNIST experiments using the targeted fault-check scheme with different values of $\alpha$; $f = 4$, $q = .2$, $p = .05$. The un-scaled reversal faults did not affect convergence, but we note the effectiveness of the threshold values: $\alpha = .5$ and 1.00 resulted in over half of the Byzantine workers being identified by the targeted checking scheme on average. Reversal faults did not affect this network for this data set, but similar faults may be useful for other tasks.
4.3.2 Summary

We used a subset of the scheme configurations used in linear regression experiments to perform optimization over the MNIST image classification data set. We observed results to be consistent across both types of experiments: three of the four schemes generally reduce convergence time compared to the Baseline, and the diminishing-LR scheme functions to impede the optimization process more than it aids fault-tolerance. The results suggest that the mechanisms in our proposed schemes have the potential to improve parallelized-SGD in the presence of Byzantine faults in general, not just in linear regression over the arbitrary data set we generated.
In this thesis, we have contributed to the research topic of Byzantine-tolerant multi-agent optimization via stochastic gradient descent with empirical experimentation. Specifically, we confined our work to the broadly applicable Byzantine setting proposed in [22], making only the standard assumption that a majority of workers are not Byzantine. We proposed four parallelized-SGD schemes utilizing reactive redundancy and presented the results of optimization experiments with multiple sets of data, three of which resulted in improvement to the ReactiveRedundantSGD algorithm. There are many ways to improve upon the work of this thesis, most notably by replacing our relatively simple multi-agent implementation with a real-world system that may encounter a Byzantine attack.

In experiments with our Diminishing-LR scheme, we saw that continuously decreasing the LR failed to improve algorithmic speed. The reactive redundancy mechanism is separate from the underlying optimization method used, so it’s possible that method-specific LR scheduling could succeed where our scheme failed.

In experiments with our Adaptive-$q$ scheme, we observed consistent speed up in convergence as a result of quicker identification of the Byzantine workers. This result was expected and bolsters the effectiveness of the reactive redundancy method for fault-tolerance. Future research of reactive redundant optimization should consider searching for optimal strategies for augmenting $q$. For instance, rather than monotonically increasing the probability of a fault-check until all Byzantine workers are
identified, another scheme could estimate the likelihood of a fault based on recent changes in the gradient, the outcomes of recent fault-checks, and/or estimates of the quality of the parameter vector.

In experiments with our Rollback scheme, we saw the checkpoint mechanism resulted in a consistent and significant speed up in convergence. The scheme depends on the ability to easily store and load a state of the parameter vector as well as the ability to estimate its quality, which may not be available in all scenarios. Future research on redundant fault-tolerant optimization could design methods for estimating the qualities of the rollback and current state of the optimization goal; we also suspect the rollback state could be used for fault-detection in addition to recovery, as in the work by Pradhan and Vaidya [18].

In experiments with our Targeted fault-checking scheme, we saw some improvement in convergence, conditional on the threshold parameter $\alpha$. We observed faster convergence using the simplest available targeting mechanism without any attempt to optimize the threshold $\alpha$. Still, at least one $\alpha$ value resulted in effective targeted fault-checks for each fault type. We see many opportunities for future work related to this scheme. A few examples: instead of our pre-configured $\alpha$, a threshold could be learned during optimization; our naïve targeting mechanism could be replaced with a proven fault-tolerant filter, such as one found in Section 2.5; targeted fault-checking could be done in parallel with distributed optimization to save on real-time cost.

Reactive redundancy is a powerful, elegant technique for distributed multi-agent optimization in the presence of Byzantine adversaries. We have shown that conceptually simple additions to reactive redundant SGD can improve performance and have identified opportunities for improvements to the technique. We hope this thesis encourages more research in redundancy as a mechanism for fault-tolerant optimization.
Bibliography


