Fault Tolerance in Iterative-Convergent Machine Learning

Aurick Qiao [1 2] Bryon Aragam [3] Bingjing Zhang [1] Eric P. Xing [1 2 3]

Abstract

Machine learning (ML) training algorithms often possess an inherent self-correcting behavior due to their iterative-convergent nature. Recent systems exploit this property to achieve adaptability and efficiency in unreliable computing environments by relaxing the consistency of execution and allowing calculation errors to be self-corrected during training. However, the behavior of such systems are only well understood for specific types of calculation errors, such as those caused by staleness, reduced precision, or asynchronicity, and for specific algorithms, such as stochastic gradient descent. In this paper, we develop a general framework to quantify the effects of calculation errors on iterative-convergent algorithms. We then use this framework to derive a worst-case upper bound on the cost of arbitrary perturbations to model parameters during training and to design new strategies for checkpoint-based fault tolerance. Our system, SCAR, can reduce the cost of partial failures by 78%–95% when compared with traditional checkpoint-based fault tolerance across a variety of ML models and training algorithms, providing near-optimal performance in recovering from failures.

1. Introduction

Distributed model training for machine learning (ML) is a workload that is typically long-running and resource-intensive. Throughout a job's lifetime, it is susceptible to hardware failures, performance fluctuations, and other uncertainties inherent to real-world cluster environments. For example, processes can be preempted by a cluster resource allocator (Vavilapalli et al., 2013; Hindman et al., 2011), parameter synchronization can be bottlenecked on a slow or congested network (Li et al., 2014b; Zhang et al., 2017b), and stragglers can severely impact overall job throughput (Cipar et al., 2013; Harlap et al., 2016). These concerns are amplified in modern shared clusters and cloud-based spot instances such as those provided by Amazon Web Services (AWS). Thus, developing new fault-tolerance strategies for modern ML systems is a critical area of research.

ML-agnostic distributed systems approaches for addressing such problems often adopt strong consistency semantics. They aim to provide strong execution guarantees at a per-operation level (such as linearizability or serializability), but may also incur higher performance overhead. On the other hand, ML training is often tolerant to small calculation errors and may not require such strong consistency guarantees. This observation has been exploited by recent ML systems to overcome cluster

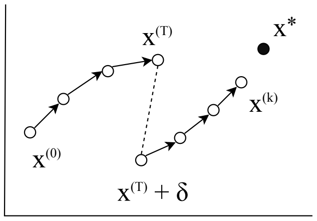


Figure 1. The self-correcting behavior of iterative-convergent algorithms. Even though a calculation error results in an undesirable perturbation of δ at iteration T, the subsequent iterations still brings the solution closer to the optimum value of x^*.

unreliability and resource limitation issues, such as bounded staleness consistency (Ho et al., 2013; Cipar et al., 2013; Cui et al., 2014), quantization and low-precision arithmetic (Courbariaux et al., 2014; Gupta et al., 2015; Hubara et al., 2017), and lock-free execution (Niu et al., 2011; Dean et al., 2012). One notable exception to this trend is checkpoint-based fault tolerance, a common strategy in current ML systems for mitigating hardware failures (Abadi et al., 2016; Wei et al., 2015; Low et al., 2012) which continues to enforce strong consistency semantics at a high cost of re-computing lost work.

This trend of relaxing consistency in ML systems relies on the *self-correcting* behavior of iterative-convergent ML training algorithms (Fig. 1). During each step, the training algorithm calculates updates based on the current values of model parameters, and then applies the updates to obtain a "better" set of model parameters. By iteratively performing this computation, the model parameters eventually converge to a set of optimal values. Small computation errors made during this procedure are eventually washed out by the successive

[1]Petuum, Inc., Pittsburgh, Pennsylvania, USA [2]Computer Science Department, Carnegie Mellon Univeristy, Pittsburgh, Pennsylvania, USA [3]Machine Learning Department, Carnegie Mellon Univeristy, Pittsburgh, Pennsylvania, USA. Correspondence to: Aurick Qiao <aurick.qiao@petuum.com>.

Proceedings of the 36th International Conference on Machine Learning, Long Beach, California, PMLR 97, 2019. Copyright 2019 by the author(s).

iterative improvements. This self-correcting behavior of ML training suggests a general strategy for designing robust training systems for unreliable environments, as follows:

(A) The execution system allows certain environmental faults and/or resource limitations to manifest as calculation errors in model training. These errors can be conceptualized as *perturbations* to the model parameters.

(B) The perturbations are self-corrected by the model training algorithm, which incurs an extra cost (e.g. additional iterations, batches, epochs, etc.). We refer to this additional cost as the *rework cost* of the perturbations.

Motivated by this general strategy, we develop a framework for exploiting self-correction in ML systems in a way that is adaptive to generic perturbations whose cause or origin is unknown. It provides a theoretical foundation for understanding the self-correcting behavior of iterative-convergent model training as well as the tools needed by ML systems to take advantage of this behavior. Our main contributions are:

1. We quantify the impact of generic perturbations on iterative-convergent algorithms in terms of their rework cost. Under reasonable convergence assumptions, we bound the rework cost in terms of the sizes of these perturbations.
2. We propose new strategies for checkpoint-based fault tolerance in distributed model training. Partially recovering from checkpoints, combined with prioritizing checkpoints in a way that reduces the size of perturbations, can significantly reduce the rework cost due to partial failures.
3. We design SCAR, a parameter server system for fault tolerant ML training and show that SCAR reduces the rework cost of partial failures by 78%–95% when compared with traditional checkpointing, which is close to optimal (vs. training with no failures).

2. Modeling Faults in ML Training

Most ML training algorithms are iterative, i.e. model parameters are updated given a current estimate of the model parameters $x^{(k)}$ until convergence to some target parameter x^*. Such algorithms are commonly called *iterative-convergent*, and include most optimization, Monte Carlo, and numerical schemes used in practice. These iterative schemes are of the form

$$x^{(k+1)} = f(x^{(k)}), \quad x^{(k)} \in \mathbb{R}^d, \tag{1}$$

for some function f. This model of iterative-convergent algorithms assumes that the current state $x^{(k)}$ is stored persistently and losslessly in memory. In practice, modern distributed ML systems are subject to faults such as hardware failures, memory corruption, and performance fluctuations. Thus, it is unrealistic to assume that $x^{(k)}$ can always be retrieved with perfect fidelity. To model this uncertainty, let δ_k be a random variable that represents an *unknown* perturbation that corrupts the current state to produce a perturbed state $x^{(k)} + \delta_k$. We make no assumptions about the cause, size, or behavior of the perturbations δ_k. More specifically, we assume the iterates obey the following scheme:

$$\begin{aligned} y^{(0)} &= x^{(0)} \\ y^{(1)} &= f(y^{(0)} + \delta_0) \\ &\vdots \\ y^{(k+1)} &= f(y^{(k)} + \delta_k) \end{aligned} \tag{2}$$

In the absence of errors, ie. $\delta_k = 0$, we have $y^{(k)} = x^{(k)}$, which reduces to the basic iterative scheme (1). Moreover, since δ_k is arbitrary, this model allows for *any* type of perturbation. In particular, perturbations may occur in every iteration or periodically according to some random process. This setup captures many of the ways that system faults can be manifested as perturbations, and we give a few important examples below.

Example 2.1 (Reduced Precision)**.** A simple practical example is using reduced precision floating/fixed point representations for storing parameter values. If $\widetilde{y}^{(k)}$ is a reduced precision version of the exact parameter values $y^{(k)}$, then the algorithm suffers perturbations of $\delta_k = \widetilde{y}^{(k)} - y^{(k)}$ at each iteration k. If the representation has a p-bit mantissa, then the size of δ_k is bounded by $|\delta_k| < 2^{-(p-1)}|y^{(k)}|$ (Higham, 2002).

Example 2.2 (Bounded Staleness Consistency)**.** In stochastic gradient descent (SGD) under the stale synchronous parallel (SSP) consistency model (Ho et al., 2013), gradients are computed in a data-parallel fashion where each of M machines may observe a stale version of the model parameters $\widetilde{x}_m^{(k)}$. Suppose $\nabla(\widetilde{x}_m^{(k)}, D_m)$ are the gradients computed during iteration k using input data D_m at machine m. If $\nabla(x^{(k)}, D)$ is the true stochastic gradient at iteration k, then the algorithm suffers a perturbation at iteration $k+1$ of:

$$\delta_{k+1} = \frac{1}{M}\sum_{m=1}^{M} \nabla(\widetilde{x}_m^{(k)}, D_m) - \nabla(x^{(k)}, D)$$

Example 2.3 (Checkpoint-based Fault Tolerance)**.** In failure recovery from checkpoints, a copy of the entire job state is periodically saved to persistent storage, and is restored in the case of a failure. Suppose a system experiences a failure at iteration T, and recovers from the failure by restoring a full checkpoint of the model parameters taken at iteration $C < T$. Then the algorithm suffers a perturbation at iteration T of $\delta_T = x^{(T)} - x^{(C)}$. Although from the system's point of view the application is returned to an exact prior state, we can still view the act of checkpoint recovery as a perturbation to the model parameters.

Remark 2.1. Reduced precision (Example 2.1) and bounded staleness consistency (Example 2.2) have already been the focus of much attention in both the ML and systems communities (Zhang et al., 2017a; Jia et al., 2018; Wei et al., 2015; Dai et al., 2015). Although not typically studied within the explicit set-up of (2), these strategies generate perturbations which fit within our framework, and preserve the correctness of training by *keeping the sizes of these perturbations small*. This is accomplished via bounded floating-point/fixed-point rounding errors for reduced precision and via a maximum staleness limit for bounded staleness consistency. In Section

4, we apply the general set-up of (2) to devise new strategies for checkpoint-based fault tolerance (Example 2.3) by reducing the sizes of the perturbations δ_k.

Remark 2.2. The iteration in (2) is closely related to *perturbed gradient descent* (Ge et al., 2015; Jin et al., 2017; Du et al., 2017). The main difference lies in the motivation: Jin et al. (2017) show that by choosing δ_k cleverly, it is possible to escape saddle points and guarantee that the iteration (2) converges to a second-order stationary point. The idea is to *design* the perturbations δ_k to an advantage, which is in stark contrast to our set-up, in which we have no control over δ_k. In the worst case, we allow δ_k to be chosen adversarially.

3. Analysis

Suppose that an ML system has experienced perturbations $\delta_1,...,\delta_T$ up to the Tth iteration. A (random) sequence a_k is called *ε-optimal* if $\mathbb{E}\|a_k - x^*\| < \varepsilon$. The main question we seek to address in this section is the following: *Given $\varepsilon > 0$, what is the "cost" in number of iterations for $y^{(k)}$ to reach ε-optimality compared to the unperturbed sequence $x^{(k)}$?* We write "cost" in quotations to emphasize that this number can be negative—for example, δ_k could randomly move $y^{(k)}$ closer to x^*, or δ_k can be constructed in advance to improve convergence as in perturbed gradient descent (see Remark 2.2). We call this quantity the *rework cost* of the perturbed sequence $y^{(k)}$, introduced in Sec. 1. Our goal in the present section is to bound the rework cost, which will be formally defined next.

3.1. Rework cost

In order to keep things simple, we assume that the unperturbed sequence satisfies

$$\|f(x^{(k)})-x^*\| \leq c\|x^{(k)}-x^*\|, \quad 0<c<1, \tag{3}$$

i.e. the iterates $x^{(k)}$ converge linearly. Although some algorithms (e.g. SGD) do not converge linearly, many of the most popular algorithms in practice do (e.g. gradient descent, proximal quasi-Newton, Metropolis-Hastings). This assumption is made purely for simplicity: We use (3) as a baseline for comparison, and the analysis can be extended to more general schemes such as SGD if desired (Appendix A.1).

Formally, the rework cost is defined as follows: Let $\kappa(y^{(k)},\varepsilon)$ be a lower bound such that $m > \kappa(y^{(k)},\varepsilon)$ implies $\mathbb{E}\|y^{(m)} - x^*\| < \varepsilon$ (this may be $+\infty$ or negative). Under (3), it is straightforward to derive a similar lower bound for the unperturbed sequence $x^{(k)}$ as $\kappa(x^{(k)},\varepsilon) = \log\left(\frac{1}{\varepsilon}\|x^{(0)} - x^*\|\right)/\log(1/c)$. This will be used as a baseline for comparison: The rework cost for the perturbations δ_k is defined to be

$$\pi(\delta_k,\varepsilon) := \kappa(y^{(k)},\varepsilon) - \kappa(x^{(k)},\varepsilon). \tag{4}$$

Using the unperturbed sequence $x^{(k)}$ as a benchmark, $\pi(\delta_k,\varepsilon)$ bounds the additional number of iterations needed for the perturbed sequence $y^{(k)}$ to reach ε-optimality (where we bear in mind that this can be negative). Clearly, $\pi(\delta_k,\varepsilon)$ depends on the sequence δ_k, and should be smaller whenever the δ_k are smaller. We seek a bound on $\pi(\delta_k,\varepsilon)$ that holds for *arbitrary* δ_k.

Remark 3.1. We use the criterion $\mathbb{E}\|y^{(k)} - x^*\| < \varepsilon$ as an optimality criterion instead of directly bounding $\mathbb{P}(\|y^{(k)} - x^*\| < \varepsilon)$. This is commonly done (e.g. Bottou et al., 2016) since bounds on $\mathbb{E}\|y^{(k)} - x^*\|$ imply bounds on the latter probability via standard concentration arguments (see e.g. Rakhlin et al., 2012).

3.2. Bounding the rework cost

To bound the rework cost, we also require that the update f satisfies a convergence rate similar to (3) for the perturbed data $\widetilde{y}^{(k)} := y^{(k)} + \delta_k$:

$$\mathbb{E}\|f(\widetilde{y}^{(k)}) - x^*\| \leq c\mathbb{E}\|\widetilde{y}^{(k)} - x^*\|, \quad 0<c<1. \tag{5}$$

This simply says that wherever the algorithm is, on average, a single step according to f will not move the iterates further from x^*. For example, it is not hard to show that gradient descent satisfies this condition whenever the objective is strongly convex (see e.g. the proof of Theorem 2.1.5 in Nesterov, 2013). In fact, this assumption is satisfied for a variety of nonconvex problems (Xu & Yin, 2017; Attouch et al., 2010), and similar results hold for other optimization schemes such as proximal methods and Newton's method.

Under (3) and (5), we have the following general bound on the rework cost:

Theorem 3.1. *Assume $\mathbb{E}\|\delta_k\| < \infty$ for $k \leq T$ and $\delta_k = 0$ for $k > T$. Under (3) and (5), we have for any $\varepsilon > 0$,*

$$\pi(\delta_k,\varepsilon) \leq \frac{\log\left(1 + \frac{\Delta_T}{\|x^{(0)}-x^*\|}\right)}{\log(1/c)} \tag{6}$$

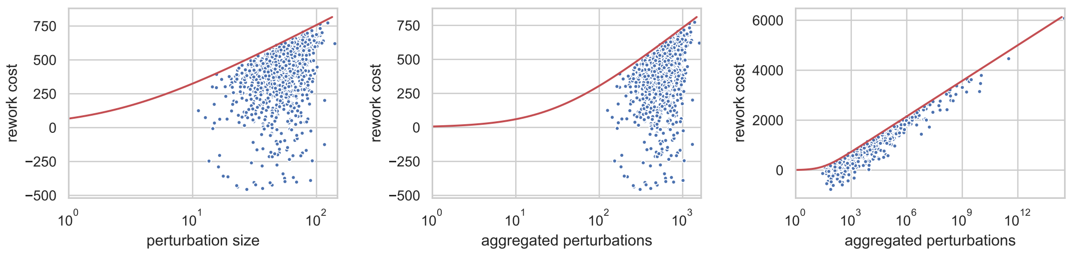
where $\Delta_T := \sum_{\ell=0}^{T} c^{-\ell}\mathbb{E}\|\delta_\ell\|$.

In fact, the bound (6) is tight in the following sense: As long as (3) cannot be improved, there exists a deterministic sequence $\delta_1,...,\delta_T$ such that (6) holds with equality. Theorem 3.1 is illustrated on a simple quadratic program (QP) in Figure 2, which provides empirical evidence of the tightness of the bound.

The interesting part of the bound (6) is the ratio $\Delta_T/\|x^{(0)} - x^*\|$, which is essentially a ratio between the aggregated cost of the perturbations and the "badness" of the initialization. For more intuition, re-write this ratio as

$$\frac{\Delta_T}{\|x^{(0)}-x^*\|} = \frac{\sum_{\ell=0}^{T} c^{k-\ell}\mathbb{E}\|\delta_\ell\|}{c^k\|x^{(0)}-x^*\|}.$$

Up to constants, the denominator is just the error of the original sequence $x^{(k)}$ after k iterations. The numerator is more interesting: It represents a time-discounted aggregate of the overall cost of each perturbation. Each perturbation δ_ℓ is weighted by a discount factor $c^{k-\ell}$, which is larger for more recent perturbations (e.g. δ_T) and smaller for older perturbations (e.g. δ_0). Thus, the dominant quantity in (6) is a ratio between the re-weighted perturbations and the expected



(a) Rework cost vs. $\|\delta_k\|$ for a single perturbation at iteration 500.

(b) Rework cost vs. Δ_T for a single perturbation at iteration 500.

(c) Rework cost vs. Δ_T for perturbations with $p = 0.001$ at each iteration.

Figure 2. Illustrations of rework costs using gradient descent on a simple 4-D quadratic program. Each plot consists of 1,000 trials with perturbation(s) randomly generated according to a normal distribution. The red line is the rework cost bound according to Theorem 3.1. The value of c is determined empirically, and the value of ϵ is set so that an unperturbed trial converges in roughly 1,000 iterations.

error from the original sequence. As expected, if the original sequence converges very quickly and the perturbations are large, the rework cost increases proportionally.

Theorem 3.1 also assumes that there are no perturbations after time T. The idea is that *if* there are no more perturbations, (6) bounds the cost of the perturbations incurred so far. Of course, in practice, the system may experience faults after time T, in which case (6) can be adjusted to include the most recent fault. The difficulty in directly accounting for future perturbations lies in our assumption that the δ_k can be arbitrary: If future iterations can experience *any* perturbation, it is clear that convergence cannot be guaranteed (e.g. consider $\delta_k = x - y^{(k)}$ for some fixed $x \neq x^*$ and all $k > T$). Under some additional assumptions, something can be said about this case; see Example A.4.

4. Checkpoint-Based Fault Tolerance

As an application of our framework, we study new strategies for checkpoint-based fault tolerance, by which a stateful computation is made resilient to hardware failures by periodically saving its program state to persistent storage. This fault-tolerance mechanism is used in many popular ML frameworks including TensorFlow (Abadi et al., 2016) and PyTorch (Paszke et al., 2017).

Using traditional checkpointing, the entire saved program state is restored after a failure, and input data is re-loaded from its persistent storage. Then, all computation since the previous checkpoint is repeated. This process maximizes the consistency of recovery by restoring the system to an exact state it was in during the past, but can incur high rework cost if the checkpoint interval is long. Let T_{rework} be the total amount of time spent re-computing lost iterations. For a single failure, T_{rework} for the traditional checkpoint strategy is the total amount of time between the previous checkpoint and the failure.

Although this traditional checkpointing is sufficient for many usage scenarios, it can break down in computing environments where the mean-time-to-failure is low (Harlap et al., 2017). For example, resource schedulers in shared clusters can kill running jobs to give more resources to higher-priority jobs, and cloud-based spot instances may be preempted frequently. In these environments, jobs using traditional checkpointing can incur a large penalty each time they experience a failure. In the most degenerate scenario, a job can run for an undetermined amount of time when its checkpoint interval is longer than the mean-time-to-failure. Thus, it is critical to reduce the rework cost incurred by checkpoint-based fault tolerance.

Fortunately, for iterative-convergent ML, we can exploit its self-correcting behavior to reduce T_{rework}. In particular, we can give up the consistency of checkpoint-recovery, and design a system which tries to reduce the size of the perturbation $\|\delta_T\|$ incurred upon failure. By doing so, Theorem 3.1 shows that the rework cost bound is also reduced, lowering the worst case rework cost and thus reducing T_{rework}.

We design a system architecture, *SCAR*,[1] consisting of two strategies which reduce $\|\delta_T\|$ compared to traditional checkpoint recovery: (1) Partial recovery, and (2) Prioritized checkpoints. SCAR extends the popular parameter server (PS) architecture for distributed model training (Ho et al., 2013; Li et al., 2014b;a)—the model parameters are partitioned across a number of PS nodes, which are accessed by worker nodes. We assume that during a failure, any number of PS nodes can go down, causing the loss of their partitions of the model parameters. We present these strategies and the design of SCAR below, and show evaluation of SCAR in Section 5.

4.1. Partial Recovery

Our first strategy is to only recover (i.e. from a previous checkpoint) the part of the model parameters which are lost due to the failure. Since the model parameters are partitioned across several PS nodes, a partial failure of PS nodes should only cause a partial loss of model parameters. Mathematically, the partial recovery strategy should result in a smaller perturbation

[1]SCAR stands for Self-Correcting Algorithm Recovery.

to the model parameters and, according to Theorem 3.1, incur a smaller rework cost.

Suppose that a fully-consistent checkpoint is taken after iteration C , and a failure occurs during iteration $T > C$ which triggers checkpoint recovery.

Theorem 4.1. *Let δ be the perturbation incurred by full checkpoint recovery, and δ' be the perturbation incurred by partial checkpoint recovery, then $\|\delta'\| < \|\delta\|$.*

Furthermore, the size of the perturbation should also be related to the fraction of model parameters which are lost—losing fewer model parameters should generate a smaller perturbation. To establish this relationship, we will assume that parameters are partitioned uniformly at random across the PS nodes, and so a random subset of parameters will be lost. This assumption is reasonable as the partitioning scheme is typically within the control of the PS system, which can choose a random partitioning.

Theorem 4.2. *Suppose that a failure causes the loss of a fraction $0 < p \leq 1$ of all model parameters chosen uniformly at random. Let δ be the perturbation incurred by full checkpoint recovery, and δ' be the perturbation incurred by partial checkpoint recovery, then $\mathbb{E}\|\delta'\|^2 = p\|\delta\|^2$.*

Thus, the expected size of perturbations incurred by partially restoring from a checkpoint decreases as the fraction of parameters lost decreases.

4.2. Priority Checkpoint

With the partial recovery strategy, we have shown that relaxing the consistency of checkpoint recovery can reduce the size of perturbations (i.e. δ_k) experienced by the training algorithm due to a failure, and thus reduce the rework cost. In this section, we further consider relaxing the consistency of saving checkpoints by taking more frequent, partial checkpoints.

Rather than saving all parameters every C iterations, consider saving a fraction $r < 1$ of the parameters every rC iterations. A *running checkpoint* is kept in persistent storage, which is initialized to the initial parameter values $x^{(0)}$ and updated each time a partial checkpoint is saved. At a given time, this checkpoint may consist of a mix of parameters saved during different iterations, and the choice of which subset of parameters to checkpoint can be controlled via system design. This strategy enables, e.g., *prioritization* of which parameters are saved during each checkpoint so as to prioritize saving parameters that will minimize the size of the perturbation caused by a failure. To do this, we consider a simple heuristic: Save the parameters which have changed the most since they were previously saved.

The checkpoint period rC is chosen so that the number of parameters saved every C iterations remains roughly constant across different values of r. As a result the prioritized checkpoint strategy writes the same amount of data per constant number of iterations to persistent storage as the full checkpoint

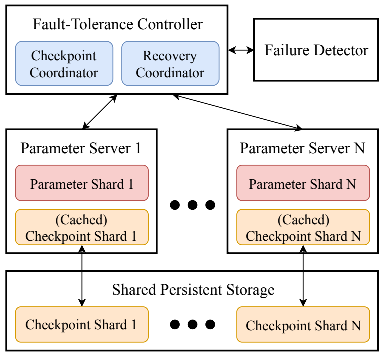


Figure 3. SCAR system architecture for partial recovery and prioritized checkpoints in distributed model training.

strategy, while having more frequent opportunities to prioritize and save parameters to the running checkpoint. We evaluate the system overhead implications of this scheme in Section 5.5.

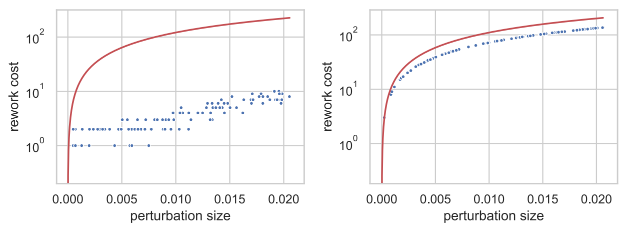
4.3. SCAR Architecture and Implementation

We implement our system, SCAR, using these two checkpoint-based fault tolerance strategies. SCAR is implemented as a PS architecture—the parameters of the ML model are randomly partitioned across PS nodes, while the input data is partitioned across worker nodes. During each iteration, the workers read values from the PS nodes, compute updates using their local input data, and send the updates to the PS nodes to be applied.

Figure 3 illustrates the architecture of SCAR. A *fault tolerance controller* runs as a separate service and consists of (1) a *checkpoint coordinator* responsible for coordinating periodic checkpoints at a fixed time interval, and (2) a *recovery coordinator* responsible for coordinating the failure recovery process whenever a failure is detected. The detection of failures is performed by a *failure detector* service, which can leverage heartbeating mechanisms in existing systems for distributed consensus such as ZooKeeper (Hunt et al., 2010). Checkpoints are saved to shared persistent storage, such as distributed filesystems like NFS (Sandberg et al., 1988), CephFS (Weil et al., 2006), or distributed databases like Cassandra (Lakshman & Malik, 2010). To speed up distance calculations between the current and previously saved parameters, each PS node keeps an in-memory cache of the current checkpoint, which is updated whenever a new partial checkpoint is saved. More details on the implementation of SCAR can be found in Appendix C.1.

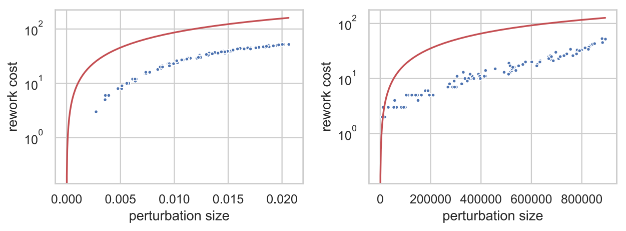
5. Experiments

With our evaluation, we wish to (1) illustrate our rework cost bounds for different types of perturbations using practical ML models, (2) empirically measure the rework costs of a variety of models under the partial recovery and prioritized checkpoint strategies in SCAR, and (3) show that SCAR incurs near-optimal rework cost in a set of large-scale experiments.



(a) Random perturbations. (b) Adversarial perturbations.

Figure 4. Rework costs of MLR on MNIST for (a) random Gaussian perturbations and (b) adversarial perturbations generated in the opposite direction from the optimum. In each trial, a single perturbation is generated at iteration 50. The red line is the upper bound according to Theorem 3.1. The value of c is determined empirically, and the value of ϵ is set so that an unperturbed trial converges in roughly 100 iterations.



(a) MLR on MNIST. (b) LDA on 20 Newsgroups.

Figure 5. Perturbations are generated by resetting a random fraction of parameters back to their initial values, for both (a) MLR and (b) LDA. Other settings are the same as Figure 4.

5.1. Models and Datasets

We use several popular models and datasets as examples for our analysis and checkpoint strategies. We briefly describe them below, and refer to Appendix C.2 for more details.

Multinomial Logistic Regression (MLR) trained with stochastic (minibatch) gradient descent. We train MLR on the MNIST (Lecun et al., 1998) and CoverType (Dheeru & Karra Taniskidou, 2017) datasets.

Matrix Factorization (MF) trained with alternating least squares (ALS). We train MF on the MovieLens (Harper & Konstan, 2015) and Jester (Goldberg et al., 2001) datasets.

Latent Dirichlet Allocation (LDA) trained with collapsed Gibbs sampling (Liu, 1994). We train LDA on the 20 Newsgroups (Lang, 1995) and Reuters (Lewis et al., 2004) datasets.

Convolutional Neural Network (CNN) consisting of 2 convolution layers and 3 fully-connected layers, trained with Adam (Kingma & Ba, 2014). We train this CNN on the MNIST (Lecun et al., 1998) dataset.

Due to limited space, we report some of the experiments here: For complete results and additional figures, see Appendix C.3.

5.2. Iteration Cost Bounds

To illustrate the behavior of the rework cost and to verify Theorem 3.1 for different types of models and perturbations, we train MLR and LDA and generate a perturbation according to one of three types: random, adversarial, and resets.

For random perturbations (Figure 4(a)), the rework cost bound is a loose upper bound on the actual rework cost. This is in contrast to the simpler quadratic program (QP) experiments shown in Figure 2, in which the bound is relatively tight. On the other hand, we also do not observe any perturbations resulting in a negative rework cost as for QP. This experiment shows that for MLR, a perturbation in a random direction is unlikely to greatly impact the total number of iterations to convergence.

We run a second experiment in which we generate "adversarial" perturbations opposite the direction of convergence (Figure 4(b)). In this case, we see that our bound is much closer to the actual rework costs, indicating that it is still a tight worst-case upper bound on the rework cost for MLR.

While Figure 4 shows the rework costs for synthetically generated perturbations, Figure 5 generates more realistic perturbations for both MLR and LDA. We generate perturbations by resetting a random subset of model parameters back to their initial values. This scheme simulates the type of perturbations the training algorithm would observe in the partial recovery scenario described in Section 4.1. In this case, we see that the behavior of actual rework costs is closer to the scenario with adversarial perturbations, although not quite as costly.

5.3. Partial Recovery

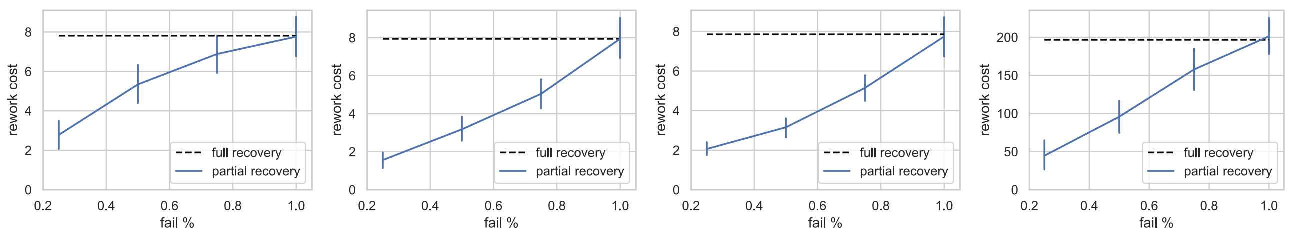
To empirically characterize the behavior of partial recovery from checkpoints, we simulate failures of varying fractions of model parameters for each model. We compare the rework costs incurred by full recovery with the rework costs incurred by partial recovery. For each model, we sample the failure iteration from a geometric distribution, which causes the loss of a subset of model parameters chosen uniformly at random.

Fig. 6 shows the results. For all models and datasets, we see the average rework cost incurred by partial recovery decreases as the failure fraction decreases. Meanwhile, the average rework cost incurred by full recovery remains constant at its maximum value, since all parameters are loaded from the checkpoint regardless of which are actually lost.

Across all models and datasets tested, SCAR with partial recovery reduces the rework cost by 12%–42% for $3/4$ failures, 31%–62% for $1/2$ failures, and 59%–89% for $1/4$ failures.

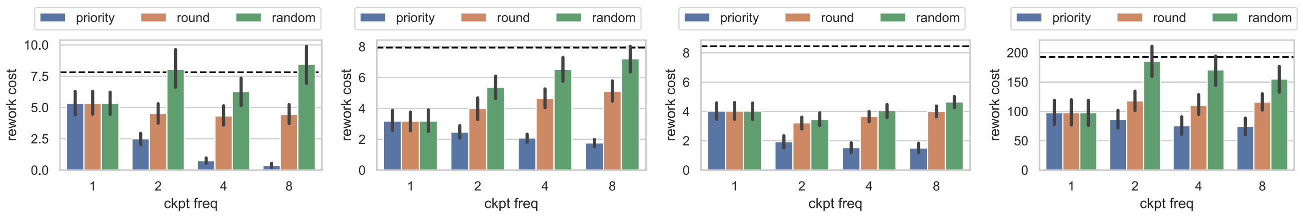
5.4. Priority Checkpoint

In this section, we evaluate the effectiveness of our priority checkpoint strategy for the MLR, MF, LDA, and CNN models.



(a) MLR on CoverType. (b) MF on MovieLens. (c) LDA on 20 Newsgroups. (d) CNN on MNIST.

Figure 6. Partial vs. full recovery where the set of failed parameters are selected uniformly at random. The x-axis shows the fraction of failed parameters, and the y-axis shows the number of rework iterations. The error bars indicate 95% confidence intervals, calculated by repeating each trial 100 times, and the dashed black line represents the rework cost of a full checkpoint. For experiments on all datasets, see Fig. C.1



(a) MLR on CoverType. (b) MF on MovieLens. (c) LDA on 20 Newsgroups. (d) CNN on MNIST.

Figure 7. Prioritized checkpoint experiments comparing between the random, round-robin, and priority strategies. The x-axis indicated checkpoint frequency relative to full checkpoints, where 1 indicates full checkpoints, 2 indicates $1/2$ checkpoints at $2\times$ frequency, etc., and the y-axis shows the number of rework iterations. The error bars indicate 95% confidence intervals, calculated by repeating each trial 100 times, and the dashed black line represents the rework cost of a full checkpoint. For experiments on all datasets, see Fig. C.2

We compare the rework costs incurred by different fractions of partial checkpoints, while keeping constant the number of parameters saved per constant number of iterations, as described in Section 4.2. As before, we sample the failure iteration from a geometric distribution. In this experiment keep the fraction of lost parameters fixed at $1/2$.

To gauge the effectiveness of prioritization, we compare between several strategies: (1) `priority`, parameters saved to checkpoint are selected based on the prioritization described in Section 4.2, (2) `round`, parameters saved to checkpoint are selected in a round-robin manner, and `random`, parameters saved to checkpoint are selected uniformly at random.

Fig. 7 shows the results. For all models and datasets, we see the `priority` strategy results in decreasing rework costs when the fraction of each checkpoint decreases (and frequency of checkpoints increases). On the other hand, the `round` strategy either reduces or increases the rework cost depending on the model and dataset, while the `random` strategy nearly always increases the rework cost.

Across all models and datasets tested, combining partial recovery with prioritized $1/8$th checkpoints at $8\times$ frequency reduces the rework cost of losing $1/2$ of all model parameters by 78%–95% when compared with traditional checkpoint recovery.

5.5. Large Scale Experiments

Lastly, we evaluate the convergence impact and system overhead of SCAR with two large-scale training scenarios using MLR and LDA. We use four AWS i3.2xlarge instances to train MLR on the full 26GB Criteo (Juan et al., 2016) dataset, and LDA on a 12GB subset of the ClueWeb12 dataset (Gabrilovich et al., 2013).

Convergence impact. For both MLR and LDA, we trigger a failure of 25% of parameters (corresponding to a single failed node in our 4-node cluster) after 7 epochs. We compare SCAR, which saves 1/8 of the highest-priority parameters every epoch, with traditional checkpointing, which saves all parameters every 8 epochs. Fig. 8 shows the results. For both MLR and LDA, SCAR achieves near-optimal rework costs of less than a single epoch, while traditional checkpointing incurs rework costs of 7 epochs corresponding to the exact amount of computation lost.

Our experiment scenario highlights the worst-case behavior of traditional checkpointing, which occurs when the failure happens immediately before a full checkpoint is taken. A randomly occurring failure is just as likely to happen any time during the checkpoint interval. However, in expectation, traditional checkpointing would still incur 4 epochs of rework cost. In dynamic-resource environments where failures can occur frequently, SCAR's reduced rework cost can significantly reduce total training time.

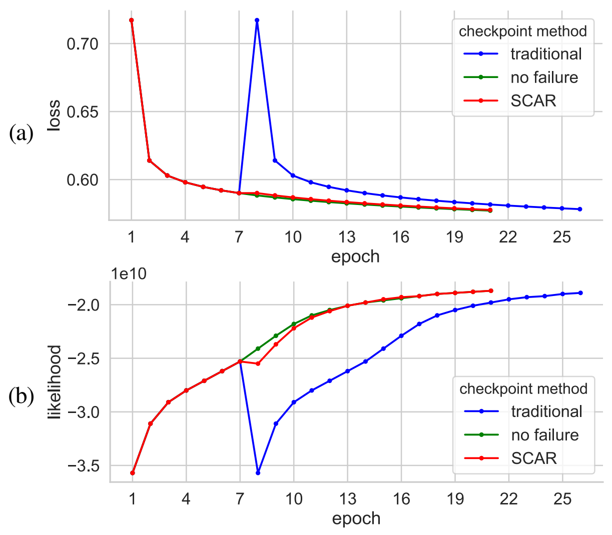


Figure 8. Large scale experiments with (a) MLR on Criteo and (b) LDA on ClueWeb12. A failure of 25% of model parameters is triggered after epoch 7. SCAR saves $1/8$ of parameters every epoch, while traditional checkpointing saves all parameters every 8 epochs.

System overhead. The checkpointing mechanisms of SCAR can be implemented with low performance overhead. In our experiments, we measured an average per-epoch overhead of < 1s for MLR and < 5s for LDA, when compared with traditional checkpointing. Given that the average time spent computing each epoch is ≈ 140s for MLR and ≈ 220s for LDA, this added overhead is negligible.

6. Related Work

In the optimization literature, optimization with inexact gradients has been extensively studied (see Schmidt et al., 2011; Devolder et al., 2014, and the references therein). These works focus on convergence rates and typically assume the errors in the gradients are small. By contrast, our focus is somewhat different, instead considering the case where the perturbations are generic, i.e. they are not restricted to gradient computations and may be significant. Mania et al. (2015) and El Gamal & Lai (2017) also consider a model similar to (2), however, perturbations are only added to the gradients.

A related body of work is distributed training under Byzantine faults (Blanchard et al., 2017; Chen et al., 2017; Damaskinos et al., 2018; Guerraoui et al., 2018), where a proportion of machines may act adversarially. However, perturbations to parameters during training are not always Byzantine, and can often be controlled via system implementations, such as bounded staleness consistency models, or partial recovery and prioritized checkpointing as in the present work.

Coded computing has been proposed as a technique to reduce the effects of stragglers and faults in distributed machine learning (Tandon et al., 2017; Karakus et al., 2017; Lee et al., 2018). These techniques use coding theory to increase the redundancy of input data or linear computations such as matrix multiplication. The failure of model parameters remains an outstanding problem, which is the main focus of our work.

In other distributed ML systems, fault tolerance is approached in an ML-agnostic way. TensorFlow (Abadi et al., 2016) offers recovery from periodic checkpoints, while the parameter server of Li et al. (Li et al., 2014a) offers live replication of parameter values. Proteus (Harlap et al., 2017) proposes an approach for fault-tolerance on transient machines by using more reliable machines for active backup of program state. In comparison, our system takes advantage of the self-correcting nature of ML, offering lower rework cost compared with traditional checkpoint-restart, and without the performance overhead of live replication or storing parameter state on designated reliable machines.

7. Conclusion

The self-correcting behavior of ML forms the basis of system techniques that allow model training to achieve adaptability and efficiency in unreliable and resource-limited environments. In this paper, we outlined a general approach to design such systems by reducing the sizes of perturbations to model parameters. We derived an upper bound on the rework cost of perturbations which can guide the design of new systems. We then proposed and implemented new strategies for checkpoint-based fault tolerance in our system SCAR. We showed that SCAR is able to reduce the rework cost of failures by an order of magnitude or more when compared to traditional checkpoint-based fault tolerance.

As for future work, we have already observed that our main assumptions (3) and (5) can be relaxed, however, it remains to study these generalizations in more detail. In particular, the cases of nonconvex ℓ, and sublinear schemes such as SGD (see also discussion in Appendix A.1). Furthermore, we have avoided making assumptions on the perturbations δ_k, however, by imposing additional assumptions on the frequency or size of these perturbations, one could derive tighter upper bounds on the rework cost. On the systems side, we proposed Theorem 3.1 primarily as a tool for analyzing existing systems and guiding the design of new systems. However, there exists opportunities for systems to more directly utilize Theorem 3.1. By approximating c and $\|x^{(0)} - x^*\|$, we may obtain a predictive model which can be evaluated on-the-fly to inform decisions made by a system during run-time.

Acknowledgements

We thank the anonymous reviewers for their valuable feedback. Our work was supported by the U.S. National Science Foundation (NSF) awards CCF1629559 and IIS1617583, and the U.S. National Institutes of Health (NIH) awards P30DA035778 and R01GM114311.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283, Savannah, GA, 2016. USENIX Association. ISBN 978-1-931971-33-1.

Attouch, H., Bolte, J., Redont, P., and Soubeyran, A. Proximal alternating minimization and projection methods for nonconvex problems: An approach based on the kurdyka-łojasiewicz inequality. *Mathematics of Operations Research*, 35(2):438–457, 2010.

Blanchard, P., Guerraoui, R., Stainer, J., et al. Machine learning with adversaries: Byzantine tolerant gradient descent. In *Advances in Neural Information Processing Systems*, pp. 118–128, 2017.

Bottou, L., Curtis, F. E., and Nocedal, J. Optimization methods for large-scale machine learning. *arXiv preprint arXiv:1606.04838*, 2016.

Cao, P. and Wang, Z. Efficient top-k query calculation in distributed networks. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, pp. 206–215, New York, NY, USA, 2004. ACM. ISBN 1-58113-802-4. doi: 10.1145/1011767.1011798.

Chen, Y., Su, L., and Xu, J. Distributed statistical machine learning in adversarial settings: Byzantine gradient descent. *arXiv preprint arXiv:1705.05491*, 2017.

Cipar, J., Ho, Q., Kim, J. K., Lee, S., Ganger, G. R., Gibson, G., Keeton, K., and Xing, E. Solving the straggler problem with bounded staleness. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pp. 22–22, Berkeley, CA, USA, 2013. USENIX Association.

Courbariaux, M., Bengio, Y., and David, J. Low precision arithmetic for deep learning. *CoRR*, abs/1412.7024, 2014.

Cui, H., Cipar, J., Ho, Q., Kim, J. K., Lee, S., Kumar, A., Wei, J., Dai, W., Ganger, G. R., Gibbons, P. B., Gibson, G. A., and Xing, E. P. Exploiting bounded staleness to speed up big data analytics. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pp. 37–48, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-10-2.

Dai, W., Kumar, A., Wei, J., Ho, Q., Gibson, G., and Xing, E. P. High-performance distributed ml at scale through parameter server consistency models. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, pp. 79–87. AAAI Press, 2015. ISBN 0-262-51129-0.

Damaskinos, G., Mhamdi, E. M. E., Guerraoui, R., Patra, R., and Taziki, M. Asynchronous Byzantine machine learning (the case of SGD). In Dy, J. and Krause, A. (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 1145–1154, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

Dean, J., Corrado, G. S., Monga, R., Chen, K., Devin, M., Le, Q. V., Mao, M. Z., Ranzato, M., Senior, A., Tucker, P., Yang, K., and Ng, A. Y. Large scale distributed deep networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pp. 1223–1231, USA, 2012. Curran Associates Inc.

Devolder, O., Glineur, F., and Nesterov, Y. First-order methods of smooth convex optimization with inexact oracle. *Mathematical Programming*, 146(1-2):37–75, 2014.

Dheeru, D. and Karra Taniskidou, E. UCI machine learning repository, 2017.

Du, S. S., Jin, C., Lee, J. D., Jordan, M. I., Poczos, B., and Singh, A. Gradient descent can take exponential time to escape saddle points. *arXiv preprint arXiv:1705.10412*, 2017.

El Gamal, M. and Lai, L. On randomized distributed coordinate descent with quantized updates. In *Information Sciences and Systems (CISS), 2017 51st Annual Conference on*, pp. 1–5. IEEE, 2017.

Gabrilovich, E., Ringgaard, M., and Subramanya, A. Facc1: Freebase annotation of clueweb corpora, version 1 (release date 2013-06-26, format version 1, correction level 0). http://lemurproject.org/clueweb12/, 2013.

Ge, R., Huang, F., Jin, C., and Yuan, Y. Escaping from saddle points — online stochastic gradient for tensor decomposition. In Grünwald, P., Hazan, E., and Kale, S. (eds.), *Proceedings of The 28th Conference on Learning Theory*, volume 40 of *Proceedings of Machine Learning Research*, pp. 797–842, Paris, France, 03–06 Jul 2015. PMLR.

Goldberg, K., Roeder, T., Gupta, D., and Perkins, C. Eigentaste: A constant time collaborative filtering algorithm. *Inf. Retr.*, 4(2):133–151, July 2001. ISSN 1386-4564. doi: 10.1023/A:1011419012209.

Guerraoui, R., Rouault, S., et al. The hidden vulnerability of distributed learning in byzantium. In *International Conference on Machine Learning*, pp. 3518–3527, 2018.

Gupta, S., Agrawal, A., Gopalakrishnan, K., and Narayanan, P. Deep learning with limited numerical precision. *CoRR*, abs/1502.02551, 2015.

Harlap, A., Cui, H., Dai, W., Wei, J., Ganger, G. R., Gibbons, P. B., Gibson, G. A., and Xing, E. P. Addressing the straggler problem for iterative convergent parallel ml. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pp. 98–111, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4525-5. doi: 10.1145/2987550.2987554.

Harlap, A., Tumanov, A., Chung, A., Ganger, G. R., and Gibbons, P. B. Proteus: Agile ml elasticity through tiered reliability in dynamic resource markets. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pp. 589–604, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4938-3. doi: 10.1145/3064176.3064182.

Harper, F. M. and Konstan, J. A. The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, 5 (4):19:1–19:19, December 2015. ISSN 2160-6455. doi: 10.1145/2827872.

Higham, N. J. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2002. ISBN 0898715210.

Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R., Shenker, S., and Stoica, I. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pp. 295–308, Berkeley, CA, USA, 2011. USENIX Association.

Ho, Q., Cipar, J., Cui, H., Kim, J. K., Lee, S., Gibbons, P. B., Gibson, G. A., Ganger, G. R., and Xing, E. P. More effective distributed ml via a stale synchronous parallel parameter server. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'13, pp. 1223–1231, USA, 2013. Curran Associates Inc.

Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., and Bengio, Y. Quantized neural networks: Training neural networks with low precision weights and activations. *J. Mach. Learn. Res.*, 18(1):6869–6898, January 2017. ISSN 1532-4435.

Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pp. 11–11, Berkeley, CA, USA, 2010. USENIX Association.

Jia, X., Song, S., He, W., Wang, Y., Rong, H., Zhou, F., Xie, L., Guo, Z., Yang, Y., Yu, L., Chen, T., Hu, G., Shi, S., and Chu, X. Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes. *ArXiv e-prints*, July 2018.

Jin, C., Ge, R., Netrapalli, P., Kakade, S. M., and Jordan, M. I. How to escape saddle points efficiently. *arXiv preprint arXiv:1703.00887*, 2017.

Juan, Y., Zhuang, Y., Chin, W.-S., and Lin, C.-J. Field-aware factorization machines for ctr prediction. In *Proceedings of the 10th ACM Conference on Recommender Systems*, RecSys '16, pp. 43–50, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4035-9. doi: 10.1145/2959100.2959134.

Karakus, C., Sun, Y., Diggavi, S., and Yin, W. Straggler mitigation in distributed optimization through data encoding. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, pp. 5440–5448, USA, 2017. Curran Associates Inc. ISBN 978-1-5108-6096-4.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

Lakshman, A. and Malik, P. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010. ISSN 0163-5980. doi: 10.1145/1773912.1773922.

Lang, K. Newsweeder: Learning to filter netnews. In *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 331–339, 1995.

Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998. ISSN 0018-9219. doi: 10.1109/5.726791.

Lee, K., Lam, M., Pedarsani, R., Papailiopoulos, D., and Ramchandran, K. Speeding up distributed machine learning using codes. *IEEE Transactions on Information Theory*, 64(3):1514–1529, March 2018. ISSN 0018-9448. doi: 10.1109/TIT.2017.2736066.

Lewis, D. D., Yang, Y., Rose, T. G., and Li, F. Rcv1: A new benchmark collection for text categorization research. *J. Mach. Learn. Res.*, 5:361–397, December 2004. ISSN 1532-4435.

Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B.-Y. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pp. 583–598, Broomfield, CO, 2014a. USENIX Association. ISBN 978-1-931971-16-4.

Li, M., Andersen, D. G., Smola, A., and Yu, K. Communication efficient distributed machine learning with the parameter server. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'14, pp. 19–27, Cambridge, MA, USA, 2014b. MIT Press.

Liu, J. S. The collapsed gibbs sampler in bayesian computations with applications to a gene regulation problem. *Journal of the American Statistical Association*, 89(427):958–966, 1994. ISSN 01621459.

Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., and Hellerstein, J. M. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, April 2012. ISSN 2150-8097. doi: 10.14778/2212351.2212354.

Mania, H., Pan, X., Papailiopoulos, D., Recht, B., Ramchandran, K., and Jordan, M. I. Perturbed iterate analysis for asynchronous stochastic optimization. *arXiv preprint arXiv:1507.06970*, 2015.

Nair, V. and Hinton, G. E. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pp. 807–814, USA, 2010. Omnipress. ISBN 978-1-60558-907-7.

Nemirovski, A., Juditsky, A., Lan, G., and Shapiro, A. Robust stochastic approximation approach to stochastic programming. *SIAM Journal on optimization*, 19(4): 1574–1609, 2009.

Nesterov, Y. *Introductory lectures on convex optimization: A basic course*, volume 87. Springer Science & Business Media, 2013.

Niu, F., Recht, B., Re, C., and Wright, S. J. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In *Proceedings of the 24th International Conference on Neural Information Processing Systems*, NIPS'11, pp. 693–701, USA, 2011. Curran Associates Inc. ISBN 978-1-61839-599-3.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

Qiao, A., Aghayev, A., Yu, W., Chen, H., Ho, Q., Gibson, G. A., and Xing, E. P. Litz: Elastic framework for high-performance distributed machine learning. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 631–644, Boston, MA, 2018. USENIX Association. ISBN 978-1-931971-44-7.

Rakhlin, A., Shamir, O., Sridharan, K., et al. Making gradient descent optimal for strongly convex stochastic optimization. In *ICML*, volume 12, pp. 1571–1578. Citeseer, 2012.

Sandberg, R., Golgberg, D., Kleiman, S., Walsh, D., and Lyon, B. Innovations in internetworking. chapter Design and Implementation of the Sun Network Filesystem, pp. 379–390. Artech House, Inc., Norwood, MA, USA, 1988. ISBN 0-89006-337-0.

Schmidt, M., Roux, N. L., and Bach, F. R. Convergence rates of inexact proximal-gradient methods for convex optimization. In Shawe-Taylor, J., Zemel, R. S., Bartlett, P. L., Pereira, F., and Weinberger, K. Q. (eds.), *Advances in Neural Information Processing Systems 24*, pp. 1458–1466. Curran Associates, Inc., 2011.

Sergeev, A. and Balso, M. D. Horovod: fast and easy distributed deep learning in tensorflow. *CoRR*, abs/1802.05799, 2018.

Tandon, R., Lei, Q., Dimakis, A. G., and Karampatziakis, N. Gradient coding: Avoiding stragglers in distributed learning. In Precup, D. and Teh, Y. W. (eds.), *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 3368–3376, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., and Baldeschwieler, E. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pp. 5:1–5:16, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2428-1. doi: 10.1145/2523616.2523633.

Wei, J., Dai, W., Qiao, A., Ho, Q., Cui, H., Ganger, G. R., Gibbons, P. B., Gibson, G. A., and Xing, E. P. Managed communication and consistency for fast data-parallel iterative analytics. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pp. 381–394, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3651-2. doi: 10.1145/2806777.2806778.

Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E., and Maltzahn, C. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pp. 307–320, Berkeley, CA, USA, 2006. USENIX Association. ISBN 1-931971-47-1.

Xu, Y. and Yin, W. A globally convergent algorithm for nonconvex optimization based on block coordinate update. *Journal of Scientific Computing*, 72(2):700–734, 2017.

Zhang, H., Li, J., Kara, K., Alistarh, D., Liu, J., and Zhang, C. ZipML: Training linear models with end-to-end low precision, and a little bit of deep learning. In Precup, D. and Teh, Y. W. (eds.), *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 4035–4043, International Convention Centre, Sydney, Australia, 06–11 Aug 2017a. PMLR.

Zhang, H., Zheng, Z., Xu, S., Dai, W., Ho, Q., Liang, X., Hu, Z., Wei, J., Xie, P., and Xing, E. P. Poseidon: An efficient communication architecture for distributed deep learning on gpu clusters. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pp. 181–193, Berkeley, CA, USA, 2017b. USENIX Association. ISBN 978-1-931971-38-6.