Elastic Machine Learning Systems with Co-adaptation

Aurick Qiao

CMU-CS-21-135

August 2021

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Eric P. Xing, Chair
Gregory R. Ganger
Phillip B. Gibbons
Joseph E. Gonzalez

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by a NSERC Postgraduate Scholarship – Doctoral (PGS-D), the Defense Advanced Research Projects Agency under grant number FA8702-15-D-0002, the National Institute of General Medical Sciences under grant number R01GM093156, and the National Science Foundation under grant numbers IIS1563887, CCF1629559, and IIS1617583. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Machine Learning, Deep Learning, Elasticity, Cluster Scheduling, Fault Tolerance

To my father, Bin.

Abstract

In recent years, the amount of computation being invested into machine learning (ML) and deep learning (DL) training has multiplied by several orders of magnitude. Under these conditions, elasticity—the ability of a system to dynamically adapt to changing supply and demand of compute resources over time—is a key ingredient for efficient resource management. Elasticity has long been proven to improve the resource utilization, execution performance, and fault tolerance of traditional applications such as web services and big data processing. However, elastic ML training is a relatively new area of interest, and faces different challenges from traditional applications due to ML training's highly sub-linear resource scalability, diverse execution patterns and strategies, and dependence between distributed workers.

This thesis steps beyond the existing early work in elastic ML by employing *co-adaptation*, i.e. combining both system-level and application-side adaptations, to better adapt to dynamic compute resources. Although previous frameworks can enable elasticity by relying on system-level implementations, they ignore the inherent resource adaptability of ML training that can be leveraged to better overcome the aforementioned challenges. We present the design, implementation, and evaluation of three elastic systems for ML that improve DL training time in shared GPU clusters by 37-50%, enable elasticity for a diverse set of ML training applications, and reduce the impact of resource failures by 78-95%.

Acknowledgments

Needless to say, the most important person during my PhD years has been my advisor, Eric Xing. I joined CMU as a complete newcomer to machine learning and distributed systems, which are now the main topics of this thesis. I was unequipped with any knowledge and much less the ability to conduct research in this field. Eric provided me with the guidance, resources, and patience I needed to grow and succeed. Without his support, I would not be at this point of my education and career.

I would also like to thank the other members of my thesis committee: Greg Ganger, Phil Gibbons, and Joseph Gonzalez. The support I had from Greg as a mentor and collaborator went above and beyond my highest expectations, spending countless hours to guide me with every aspect of research. Phil provided me with the feedback and encouragement I needed to become a better technical communicator, skills that will lead me through the rest of my career and for which I will always be thankful. Joey's work has always been an inspiration for my own research since day one, and I am amazed and grateful to have his advice on my thesis.

My time at CMU was filled with countless friends and collaborators who were the ones to really make it all worthwhile, from pushing for deadlines in the middle of the night, to playing board games and spending afternoons rock climbing, to the weekly meetings at the PDL and Sailing Lab: Hao Zhang, Jinliang Wei, Sang Choe, Suhas Subramanya, Evan Shimizu, Dominic Chen, David Dai, Bryon Aragam, Colin White, Nic Resch, Abutalib Aghayev, Aaron Harlap, Qirong Ho, Willie Neiswanger, Jin Kyu Kim, Zhiting Hu, Brandon Amos, Pengtao Xie, Xun Zheng, Noam Brown, Henggang Cui, and many others. Special thanks to Garth Gibson, who acted as a mentor through my formative years, and to Deb Cavlovich, who helped me navigate the maze that is the international student visa system.

During my PhD, I took a leave of absence to help build a company called Petuum. I am lucky to have this experience which made the last few years truly unique. Along the way, I had the pleasure of working with some of the most talented and experienced people in the software and technology industry: Hector Liu, Tong Wen, Luke Lu, Jayesh Gada, Sean Chen, Henry Guo, Alex Liang, Mark Schulze, Liz Yi, Cathy Serventi, Richard Fan, Omkar Pangarkar, Vishnu Vardhan, Guangjing Chen, Shelley Gong, Xin Gao, Peng Wu, Bingjing Zhang, Haoyang Chen and Weiren Yu.

Through all the ups and downs, my family has always been the one constant in my life that I can fall upon. Mom, I will always be grateful for your love and encouragement! Amazingly, my family has grown during these years to include my now wife, Linda. I love you and I am forever thankful for your continual support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the past decade, machine learning (ML) has found unprecedented success in solving practical problems across diverse application domains, such as recommendation systems [70, 117], ad-click prediction [107, 153], sentiment analysis [155, 233], object detection [169, 184], and more. Behind this success is an ever-increasing demand for compute resources, which are leveraged to train larger and more complex models on vaster datasets. For example, even to train a single model, the amount of floating point operations (FLOPs) required has increased by approximately $10\times$ per year during the period between 2012 to 2019 [19]. Additionally, to obtain a model of acceptable quality, practitioners often need to train it many times using different hyper-parameters, either through a manual experimentation process or by leveraging more recent techniques in AutoML, which can increase the compute demand by an order of magnitude or more [27, 108, 115, 130]. At the same time, the recent advancements in deep learning (DL) have driven the adoption of specialized hardware accelerators [106] such as GPUs, TPUs, and FPGAs, which are expensive to obtain and are in limited supply for most organizations who want to employ ML. As a result, automatic management and efficient utilization of these compute resources is a critical concern for practical applications of ML.

One key enabler of automatic and efficient resource management is *elasticity*. In this thesis, we define elasticity as *the ability of a system to dynamically adapt to changing supply and demand of compute resources over time*. Traditional workloads such as serving web applications [139] and big data processing [50] have an advantage of being inherently receptive to elastic execution. For example, HTTP servers may be added to or removed from an existing web application freely without affecting each other's execution or requiring significant coordination. This ability allows web applications hosted in the cloud to quickly respond to high user demand by scaling out to more compute resources, and to save cost during periods of low user demand by scaling in to fewer compute resources. Big data processing applications that perform transformations on independent data records may recover from worker failures (i.e. unexpected loss of compute resources) by simply re-computing the subset of data records which were lost due to the failure [227]. Overall, these applications that traditionally occupied clouds and datacenters are well supported by mature elastic frameworks that enable efficient resource management.

Comparatively, elasticity for machine learning and deep learning training is a more recent area of interest. While the traditional settings of academic and high-performance computing in which ML models have long been trained do not present strong requirements for elasticity,

the opposite is true in production computing environments such as clouds and datacenters in which ML is becoming increasingly prevalent. In such computing environments, the supply and demand of compute resources are often dynamic and cost-conscious resource management is key to the sustainable usage of an organization's financial budget.

However, elasticity for ML training poses different challenges from traditional workloads such as web applications and big data processing, and are under-addressed by existing elastic and resource management infrastructure. These challenges include:

1. **Highly sub-linear scalability.** Training performance does not increase in proportion to the amount of compute resources, and rapidly falls behind optimal linear scalability. The threshold for the amount of resources that can be efficiently utilized varies greatly between different training tasks, and depends on factors like the hardware performance, the model being trained, and the particular training configurations used.

2. **Diverse execution strategies.** Different models often require different training algorithms, each of which may employ different strategies for optimizing distributed execution, such as computation scheduling, inter-node synchronization, and access to shared state. Furthermore, some strategies may assume the number of distributed workers is fixed, which makes them challenging to apply in elastic settings.

3. **Dependence between workers.** During training, distributed workers communicate frequently with each other to exchange model parameter state and to synchronize training progress. Extra workers may not be freely added or removed without affecting the progress of the remaining workers. Therefore, active coordination between all active workers is often required to handle elasticity and failure recovery.

Recent works enable elasticity for ML by designing systems with ML workloads in mind, and are shown to improve expected training time and cluster resource utilization. For example, Proteus [83], Flint [194], and TR-Spark [220] used pre-emptible instances in public clouds such as the AWS spot market to reduce the cost of big-data analytics and machine learning training. Pre-emptible instances may be revoked by the cloud at any time, which is overcome by these systems through automatic failure recovery and the ability to continue training using fewer resources. By employing elasticity with pre-emptible instances, Proteus showed that the dollar cost of ML training can be reduced by 85% and training time can be reduced by 37% in comparison to using a static on-demand cluster.

Others including Huang et al. [93] and Or et al. [164] investigated the problem of auto-scaling for machine learning and deep learning, respectively. In this setting, an automatic policy decides the amount of compute resources to be used by a training job that best balances between training performance and resource efficiency, and dynamically applies new resource configurations during training. In particular, Or et al. showed that automatic auto-scaling can reduce the training time of deep learning models by up to 45% and the GPU-time by up to 85% in comparison to a fixed amount of resources that might be improperly chosen by the user.

Finally, elasticity has been extensively applied in cluster resource scheduling. In this setting, deep learning models are trained using compute clusters shared by multiple users and/or training jobs, and a cluster resource scheduler is responsible for assigning compute resources to each job

in a way that optimizes for cluster-wide objectives such as minimizing average training time, maximizing resource utilization, or ensuring scheduling fairness. Elasticity increases flexibility for the scheduler to dynamically adapt the resource allocations to better fit each training job, or to fill idle resources when available. SLAQ [231], Gandiva [214], and Optimus [170] are examples of recent works that leverage elasticity to improve cluster resource scheduling for ML and DL.

Although the applications of elasticity are diverse, nearly all existing works view elasticity as being solely the responsibility of the execution system, which aims to preserve the operations of the ML training application with perfect fidelity while transparently adapting to dynamic resources. This approach simplifies the problem so that elasticity can be achieved through traditional system-level techniques. However, it fails to leverage the inherent resource elasticity and adaptability that may be present within the ML training algorithms themselves, which can be the key to solving the aforementioned challenges.

For example, the best "batch size" (an application-side parameter) for deep learning differs when the model is trained using different numbers of GPUs, so adapting the batch size when the number of GPUs changes can enable faster training and better resource utilization (Chapter 3). Furthermore, ML training algorithms are naturally resilient to small calculation errors, which are "washed away" with further training. Simply allowing calculation errors to occur when faults occur can be a more efficient method of recovery from resource failures (Chapter 5).

One major contribution of this thesis is the formulation of *goodput*, a measure of training performance that incorporates both system throughput and statistical efficiency. The goodput can be affected by both system-level choices and application-side choices, making it possible to jointly co-optimize ML and DL training in both dimensions. For example, a choice for the batch size that maximizes system throughput may in fact harm statistical efficiency, leading to an overall decrease in training performance. By using the goodput as the optimization target, we show that deep learning training can be more automatic, more efficient, and fairer in shared-resource clusters (Chapter 3).

1.1 Thesis Statement

In this thesis, we step beyond existing work and tackle elastic ML with *co-adaptation*, which combines both system-level adaptations as well as application-side adaptations to manage the dynamic supply and demand of compute resources. We show that ML training also possesses inherently elastic properties, different from those in traditional applications, that can be leveraged to improve their execution in elastic environments. This thesis makes the following statement:

> **Thesis statement:** *Elasticity for ML training can be achieved through co-adaptation, i.e. by jointly applying system-level and application-side adaptations. Doing so enables better resource utilization, rapid recovery from failures, and automatic configuration of ML training applications in dynamic-resource environments.*

The thesis statement above will be supported by evidence from several case studies involving the design, implementation, and evaluation of several new elastic systems for ML. In particular,

1. **Pollux: Co-adaptive Cluster Scheduler for Deep Learning Training (Chapter 3).** Pollux is a novel cluster resource scheduler that combines traditional system-level elastic-

ity with adaptive batch sizes and learning rates. By co-optimizing these inter-dependent factors at both the per-job granularity and at the cluster-wide granularity, Pollux improves DL training scalability and cluster resource utilization. Pollux reduces average DL training time by 37%–50% relative to state-of-the-art DL schedulers, even when when compared against idealized baselines. With Pollux, we introduce the notion of *goodput*, a measure of DL training performance that combines system throughput and statistical efficiency, enabling the co-adaptation of DL systems and algorithms.

2. **Litz: Elastic Framework for High-Performance Machine Learning (Chapter 4).** Litz brings the diverse execution strategies of distributed ML training under a unified programming abstraction that decouples the application from its underlying compute resources. Training workloads using Litz can be transparently re-balanced across a dynamic number of machines (§4.5.1), or automatically utilize external memory when low on main memory (§4.8). At the same time, Litz provides a flexible programming model that supports a diverse set of performance enhancements employed by distributed ML, such as those required by a highly-optimized implementation of HDBSCAN clustering (§4.7). Although Litz is not co-adaptive by itself, it enables new co-adaptive frameworks to be built upon its foundations, which is demonstrated in Chapter 5.

3. **SCAR: Fault Tolerance by Exploiting Iterative-Convergent Training (Chapter 5).** Finally, we demonstrate how ML training algorithms themselves may naturally adapt to faults and failures in their execution environment, resulting in a co-adaptive framework for fault tolerance. Starting from the general observation that ML training is iterative-convergent, we develop a theoretical framework to quantify the effects of calculation errors during training. 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, is built upon Litz and 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.

Chapter 2

Background

Distributed machine learning employs a diverse set of training algorithms and techniques which result in diverse requirements for the underlying execution and resource management systems. This chapter reviews the background relevant to this thesis, starting with foundational properties shared by most distributed ML algorithms, followed by specific examples for deep learning and other classical models.

2.1 Machine Learning from First Principles

While ML algorithms come in many forms, nearly all of them share the following commonalities. First, they possess an objective (or loss) function $\mathcal{L}(w; \mathcal{D})$ defined over the model parameters (or weights) w and input (or training) data $\mathcal{D}$, which measures how well the model parameters w fit the data $\mathcal{D}$. Second, their goal is to find a value of w that optimizes the objective $\mathcal{L}(w; \mathcal{D})$ using an *iterative-convergent* training algorithm that repeatedly executes an update equation, which gradually moves w towards an optimal value. These update equations follow the generic form

$$w^{(t)} = f(w^{(t-1)}; \mathcal{D}), \tag{2.1}$$

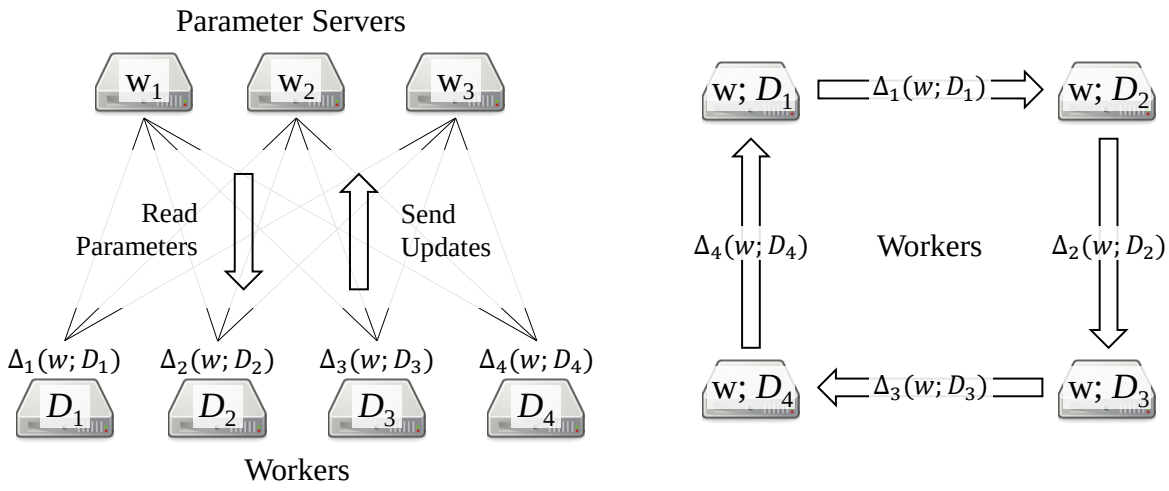
where $w^{(t)}$ is the vector (or matrix) of model parameters at iteration t and f is a function that computes $w^{(t)}$ using the previous model parameters $w^{(t-1)}$ and the input (training) data $\mathcal{D}$. Often, the function f applies an additive update to the model parameters so that

$$w^{(t)} = w^{(t-1)} + \Delta(w^{(t-1)}; \mathcal{D}), \tag{2.2}$$

where Δ is a function that computes an additive update to the parameters using their previous values $w^{(t-1)}$ and the input (training) data $\mathcal{D}$.

2.1.1 Distributed Training with Data Parallelism

Data-parallelism is a popular strategy for distributing ML training algorithms. Arising from the iid (independent and identically distributed) assumption on the input data, the update function Δ



(a) Example parameter server architecture for data-parallel ML. Workers read model parameters from and send updates to the parameter servers, which store the partitioned model parameters.

(b) Example collective (ring topology) all-reduce for data-parallel ML. Workers store a copy of all model parameters and send updates to each other in a cyclic pattern.

Figure 2.1: Examples of parameter server and collective all-reduce for data-parallel ML.

can often be decomposed as

$$\Delta(w^{(t-1)}; \mathcal{D}) = \sum_{i=1}^{P} \Delta_i(w^{(t-1)}; \mathcal{D}_i), \tag{2.3}$$

where $\mathcal{D}_1, \ldots, \mathcal{D}_P$ partition the input data $\mathcal{D}$ and each Δ_i computes a partial update using $\mathcal{D}_i$ which, when aggregated, form the final update Δ. This allows each update to be calculated in a data-parallel fashion with input data and update calculations distributed across a cluster of workers.

Parameter Server

A popular approach for implementing data-parallel machine learning is by using the parameter server architecture [51, 91, 132, 211]. Typically, implementations consist of two types of nodes—workers and parameter servers. The input data is partitioned across the worker nodes, which calculate partial updates Δ_i and send them to the parameter server nodes. The model parameters are partitioned across the parameter server nodes, which apply the partial updates and send the updated model parameters $w^{(t)}$ back to the worker nodes.

The parameter server architecture naturally arises from the update equation of machine learning algorithms. It can scale to large models by partitioning the model parameters between several parameter server nodes [16, 132, 224]. It also provides flexibility of access to the worker nodes, which enables the use of relaxed consistency models (See §2.2.2) that improve training throughput [91, 132, 211].

Collective All-Reduce

Another popular approach for implementing data-parallel machine learning is by using a collective all-reduce operation. Under this setting, each worker node holds a complete replica of the model parameters, so no special parameter server nodes are required. Instead, the workers aggregate the partial updates Δ_i amongst themselves and apply the final Δ to their own replica of the model parameters. The aggregation procedure can be done according to several different communication topologies and its optimization is the subject of much research [37, 167, 178, 179, 202].

Compared to the parameter server architecture, collective all-reduce operations have a long history of applications in high-performance computing. Publicly available implementations can be highly optimized in terms of both their algorithmic efficiency as well as their ability to leverage hardware-specific capabilities [163, 206]. However, they may be less well-equipped to support large models or relaxed consistency models as parameter servers.

2.2 Example ML Applications

In this section, we review three different example ML applications to highlight their algorithmic diversity and how they each fit into the frameworks of iterative and data-parallel training described in the previous sections. These three applications are also used throughout this thesis as evaluation workloads for the systems to be presented.

2.2.1 Deep Learning and Stochastic Gradient Descent

Deep learning (DL) is one specialty within ML that has exploded in popularity in recent years, finding unprecedented success in applications such as computer vision and natural language processing. Although deep learning models can be diverse in size, they still obey the structures laid out in §2.1 and can be distributed with data parallelism.

Training a deep learning model typically involves minimizing a *loss function* of the form

$$\mathcal{L}(w) = \frac{1}{|\mathcal{D}|} \sum_{x_i \in \mathcal{D}} \ell(w, x_i), \tag{2.4}$$

where $w \in \mathbb{R}^d$ are the model parameters to be optimized, $\mathcal{D}$ is the training dataset, x_i is an individual sample in $\mathcal{D}$, and ℓ is the loss evaluated at a single sample. The loss function can be minimized using stochastic gradient descent (SGD) or its variants like AdaGrad [60] and Adam [114]. SGD repeatedly applies the following update until the loss converges to a stable value:

$$w^{(t+1)} = w^{(t)} - \eta^{(t)} \hat{g}^{(t)} \tag{2.5}$$

$\eta^{(t)}$ is known as the learning rate, which is a scalar that controls the magnitude of each update, and $\hat{g}^{(t)}$ is a stochastic gradient estimate of the loss function $\mathcal{L}$, computed using backpropagation [185] on a random *mini-batch* $\mathcal{M}^{(t)} \subset X$ of the training data:

$$\hat{g}^{(t)} = \frac{1}{M} \sum_{x_i \in \mathcal{M}^{(t)}} \nabla \ell(w^{(t)}, x_i). \tag{2.6}$$

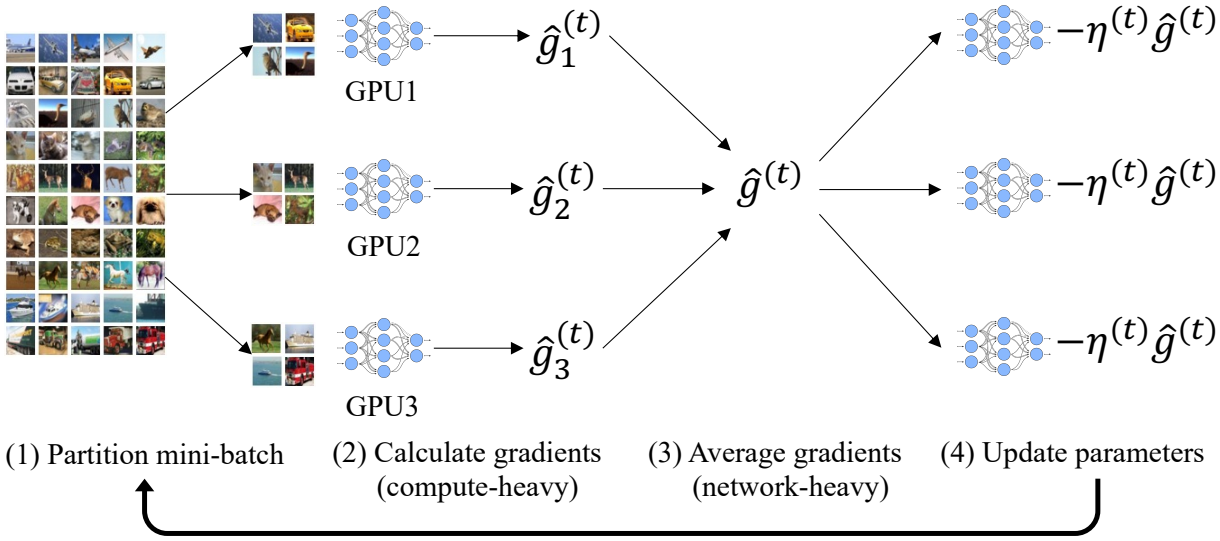


Figure 2.2: Example of deep learning training using data-parallel SGD on 3 GPUs. Local gradients are computed from a minibatch of the training dataset [53], then are averaged across all GPUs and used to update the model parameters.

The learning rate $\eta^{(t)}$ and batch size $M = |\mathcal{M}^{(t)}|$ are hyper-parameters which are typically chosen prior to training.

Data-parallel execution

DL models are often trained in the distributed setting using synchronous data-parallelism. The model parameters $w^{(t)}$ are replicated across a set of distributed GPUs $1, \ldots, K$, and each mini-batch $\mathcal{M}^{(t)}$ is divided into equal-sized partitions per node, $\mathcal{M}_1^{(t)}, \ldots, \mathcal{M}_K^{(t)}$. Each GPU k computes a local gradient estimate $\hat{g}_k^{(t)}$ using its own partition:

$$\hat{g}_k^{(t)} = \frac{1}{m} \sum_{x_i \in \mathcal{M}_k^{(t)}} \nabla \ell(w^{(t)}, x_i), \tag{2.7}$$

where $m = |\mathcal{M}_k^{(t)}|$ is the per-GPU batch size. These local gradient estimates are then averaged across all GPUs to obtain the desired $\hat{g}^{(t)}$ as defined by Eqn. 2.6. Finally, each node applies the same update using $\hat{g}^{(t)}$ to obtain the new model parameters $w^{(t+1)}$.

The batch size and learning rate are tightly related to DL training performance in elastic settings. When the number of GPUs is dynamically increased, the batch size and learning rate should also be increased to achieve high system throughput. When the number of GPUs is dynamically decreased, the batch size and learning rate should also be decreased to achieve high statistical efficiency. The precise trade-offs between the number of GPUs, batch size, learning rate, and training performance change dynamically during training.

Chapter 3 presents Pollux, which improves elastic resource scheduling for DL training in shared compute clusters by co-adaptively tuning the cluster-wide resource allocations with the per-job batch sizes and learning rates.

2.2.2 Multinomial Logistic Regression and Staleness

Logistic Regression is a classical linear model which is widely used for classification problems. Although a relatively simple model, it has found success in a wide spectrum of domains including online advertisement [153], medical analysis [22], and fraud detection [168]. Multinomial Logistic Regression (MLR) [118] extends logistic regression into the multi-label setting, where it has found applications in computer vision [121] and natural language processing [136]. Furthermore, many deep learning classifiers today use MLR as their final classification layer that outputs their predictions.

Given training examples as D-dimensional feature vectors $x_1, \ldots, x_N$ with corresponding labels $y_1, \ldots, y_N$ belonging to K classes, MLR learns K D-dimensional weight vectors $w_1, \ldots, w_K$ so that the predicted probability that an unlabeled data sample x_i belongs to class c is equal to

$$\Pr(y_i = c) = \frac{\exp(w_c^\top x_i)}{\sum_{k=1}^{K} \exp(w_k^\top x_i)}$$

MLR can be trained by minimizing its cross-entropy loss function using a variety of different optimization algorithms [52, 135, 236]. However, in the distributed setting, MLR is commonly trained using SGD (in a similar fashion as in §2.2.1) due to its ability to scale more efficiently to large training datasets.

Training with Staleness

One advantage of MLR over deeper models is its higher tolerance to staleness during training [48]. In this setting, distributed workers observing stale model parameters that might not reflect the most recent updates from every other worker. Tolerance to staleness can be leveraged by training systems to improve distributed execution, such as mitigating the impact of stragglers by accommodating delayed model updates [43] or optimizing the use of network bandwidth by prioritizing important model updates [211].

The amount of staleness experienced by the training algorithm can be controlled using a consistency model. Whereas synchronous training — the most commonly used consistency model for deep learning (§2.2.1) — does not permit staleness, asynchronous training [51] permits any amount of staleness, and stale-synchronous training [91] permits staleness up to a maximum number of iterations s. An example of the Stale-Synchronous Parallel (SSP) consistency model is illustrated in Figure 2.3.

Chapter 4 presents a framework for machine learning that is both elastic and can support flexible consistency models that enable staleness during training. Chapter 5 investigates the underlying error tolerance property of many ML training algorithms that gives rise to staleness tolerance, with applications to recovery from unexpected faults and failures.

2.2.3 Latent Dirichlet Allocation and Gibbs Sampling

Latent Dirichlet Allocation (LDA) is a widely-used Bayesian probabilistic model for topic modeling [30] that can automatically discover the relevant topics of each document in a text corpus. LDA assumes that the appearance of each word in each document is due to a latent "topic" in that

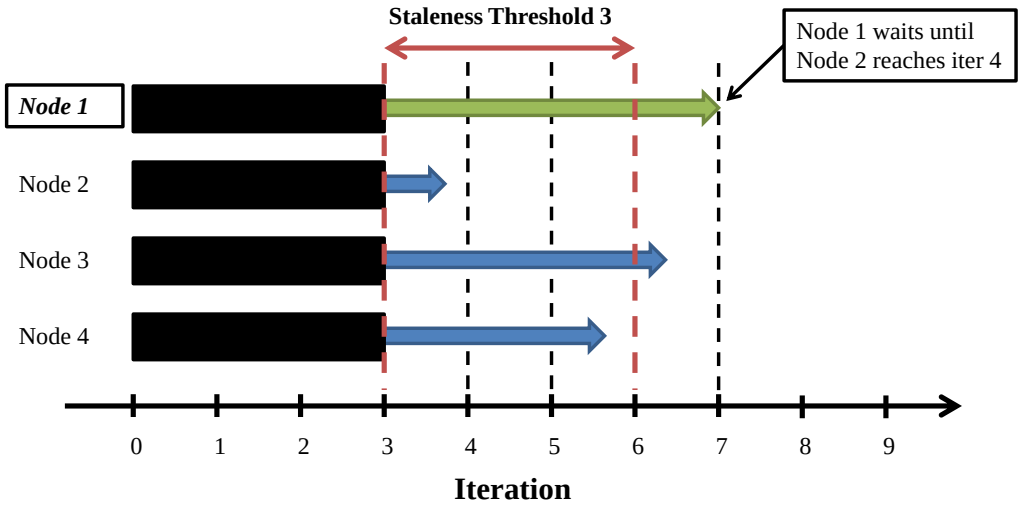


Figure 2.3: Example of the Stale-Synchronous Parallel (SSP) consistency model [91]. Node 1 may or may not observe the updates to model parameters made by the other nodes from iteration 3 onwards.

document, and learns both the mixture of topics for each document and the topic assignments of each word. Suppose there are D documents, K topics, V distinct words across all documents, and letting w_{di} denote the i-th word in document d. Three sets of parameters are learned:

1. U, a $D \times K$ "document-topic" matrix in which U_{dk} counts the number of words in document d that are assigned to topic k.

2. W, a $V \times K$ "word-topic" matrix in which W_{vk} counts the number of times word v is assigned to topic k across all documents.

3. z_{di}, the topic assigned to each w_{di}.

LDA is commonly trained using the Gibbs sampling algorithm [77], which repeatedly iterates over all z_{di}, assigning each a new value randomly sampled from a distribution computed using the d-th row of U and the w_{di}-th row of W. The matrices U and W are updated to reflect this change after each new value is assigned.

Distributed LDA Training

In the distributed setting, it is desirable to process many z_{di} elements in parallel to better utilize distributed workers. However, naive parallelization can hurt convergence if it ignores the dependency structures inherent in the LDA model. In particular, processing $z_{d_1 i_1}$ and $z_{d_2 i_2}$ in parallel will concurrently modify the same row of U if $d_1 = d_2$, or the same row of W if $i_1 = i_2$.

One way to reduce update conflicts is to divide the rows of W into B blocks, each assigned to a different worker [113]. Each worker sequentially processes the z_{di} corresponding to its local documents and its currently assigned block of W. The block assignments are rotated B times so that each worker updates all of W. Figure 2.4 illustrates this "block-rotation" schedule.

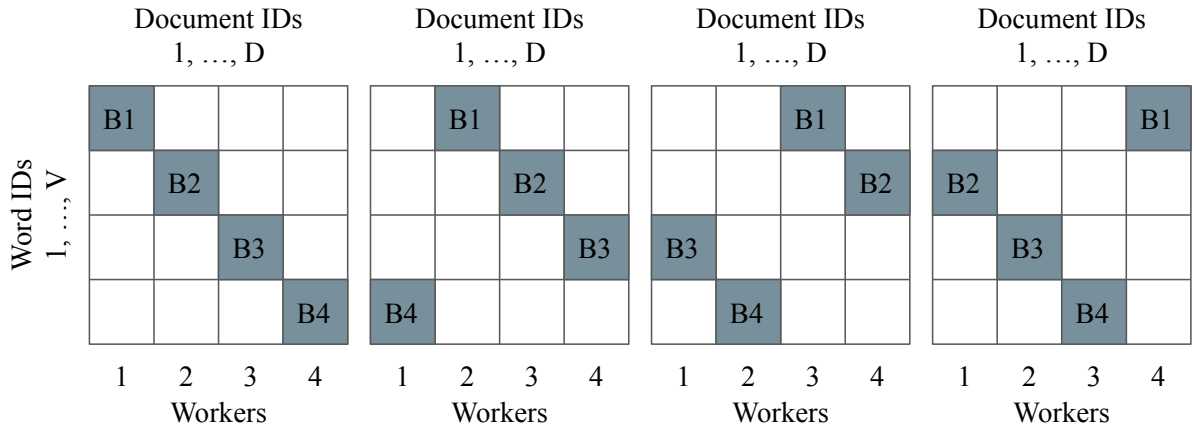


Figure 2.4: "Block-rotation" update schedule for training LDA with Gibbs sampling. The vocabulary is partitioned into 4 blocks B1, . . . , B4 and each of 4 workers process disjoint blocks at any given time.

Chapter 4 presents a framework for machine learning that is both elastic and can support fine-grained model update scheduling required by LDA and other ML training algorithms.

Chapter 3

Pollux: Improving Cluster Scheduling for Deep Learning with Co-adaptation

Many organizations today train deep learning models using compute clusters shared by multiple users and/or training jobs. In this setting, a cluster resource scheduler is responsible for assigning compute resources to each job in a way that optimizes for cluster-wide objectives such as minimizing average training time, maximizing resource utilization, or ensuring scheduling fairness. Therefore, cluster schedulers play a critical role in the performance and efficiency of resource management for deep learning.

This chapter presents Pollux, which improves scheduling in deep learning clusters through co-adaptation of inter-dependent factors both at the per-job level and at the cluster-wide level. Most existing cluster schedulers expect users to specify the number of resources for each job, often leading to inefficient resource use. Some recent schedulers choose job resources for users, but do so without awareness of how DL training can be re-optimized to better utilize the provided resources. Pollux simultaneously considers both aspects. By monitoring the status of each job during training, Pollux models how their *goodput* (a metric we introduce to combine system throughput with statistical efficiency) would change by adding or removing resources. Pollux dynamically (re-)assigns resources to improve cluster-wide goodput, while respecting fairness and continually optimizing each DL job to better utilize those resources.

In experiments with real DL jobs and with trace-driven simulations, Pollux reduces average job completion times by 37–50% relative to state-of-the-art DL schedulers, even when they are provided with ideal resource and training configurations for every job. Based on the observation that the statistical efficiency of DL training can change over time, we also show that Pollux can reduce the cost of training large models in cloud environments by 25%. Using goodput, Pollux promotes fairness among DL jobs competing for resources, based on a more meaningful measure of *useful* job progress. Pollux is implemented and publicly available as part of an open-source project at `https://github.com/petuum/adaptdl`.

The contents of this chapter were previously published in [177].

3.1 Introduction to Cluster Scheduling for DL

Deep learning (DL) training has rapidly become a dominant workload in many shared resource environments such as datacenters and the cloud. DL jobs are resource-intensive and long-running, often demanding distributed execution using expensive hardware devices (eg. GPUs or TPUs) in order to complete within reasonable amounts of time. To meet this resource demand, dedicated clusters are often provisioned for deep learning [101, 215], with a scheduler that mediates resource sharing between many competing DL jobs.

Existing schedulers require users to manually configure their jobs, which if done improperly, can greatly degrade training performance and resource efficiency. For example, allocating too many GPUs may result in long queuing times and inefficient resource usage, while allocating too few GPUs may result in long runtimes and unused resources. Such decisions are especially difficult to make in a shared-cluster setting, since optimal choices are dynamic and depend on the cluster load while a job is running.

Even though recent *elastic* schedulers can automatically select an appropriate amount of resources for each job, they do so blindly to inter-dependent training-related configurations that are just as important. For example, the *batch size* and *learning rate* of a DL job influence the amount of computation needed to train its model. Their optimal choices vary between different DL tasks and model architectures, and they have strong dependence on the job's allocation of resources.

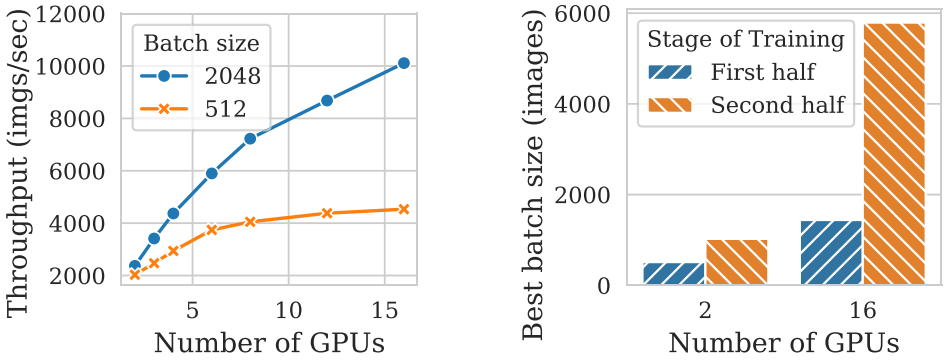
The amount of resources, batch size, and learning rate are difficult to configure appropriately without expert knowledge about both the cluster hardware performance and DL model architecture. Due to the inter-dependence between their optimal values, they should be configured jointly with each other. Due to the dynamic nature of shared clusters, their optimal values may change over time. This creates a complex web of considerations a user must make in order to configure their job for efficient execution and resource utilization.

How can a cluster scheduler help to automatically configure user-submitted DL jobs? Fundamentally, a properly-configured DL job strikes a balance between two often opposing desires: (1) *system throughput*, the number of training examples processed per wall-clock time, and (2) *statistical efficiency*, the amount of progress made per training example processed.

System throughput can be increased by increasing the batch size, as illustrated in Fig. 3.1a. A larger batch size enables higher utilization of more compute resources (e.g., more GPUs). But, even with an optimally-retuned learning rate, increasing the batch size often results in a decreased statistical efficiency [150, 193]. For every distinct allocation of GPUs, there is potentially a different batch size that best balances increasing system throughput with decreasing statistical efficiency, as illustrated in Fig. 3.1b. Furthermore, how quickly the statistical efficiency decreases with respect to the batch size depends on the current training progress. A job in a later stage of training can potentially tolerate 10x or larger batch sizes without degrading statistical efficiency, than earlier during training [150].

Looking Ahead

The rest of this chapter describes *Pollux*, a hybrid resource scheduler that *co-adaptively* allocates resources and tunes the batch size and learning rate for all DL jobs in a shared cluster. Pollux



(a) Job scalability (and thus resource utilization) depends on the batch size.

(b) The most efficient batch size depends on the allocated resources and stage of training.

Figure 3.1: Trade-offs between the batch size, resource scalability, and stage of training (ResNet18 on CIFAR-10). The learning rate is separately tuned for each batch size.

achieves this by jointly managing several system-level and training-related parameters, including the number of GPUs, co-location of workers, per-GPU batch size, gradient accumulation, and learning rate scaling. In particular:

- We propose a formulation of *goodput* for DL jobs, which is a holistic measure of training performance that takes into account both system throughput and statistical efficiency.
- We show that a model of a DL job's goodput can be learned by observing its throughput and statistical behavior during training, and used for predicting the performance given different resource allocations and batch sizes.
- We design and implement a scheduling architecture that uses such models to configure the right combination of resource allocation and training parameters for each pending and running DL job. This includes locally tuning system-level and training-related parameters for each DL job, and globally optimizing cluster-wide resource allocations. The local and global components actively communicate and cooperate with each other, operating based on the common goal of goodput maximization.
- We evaluate Pollux on a cluster testbed using a workload derived from a Microsoft cluster trace. Compared with recent DL schedulers, Tiresias [78] and Optimus [170], Pollux reduces the average job completion time by up to 73%. Even when all jobs are manually tuned beforehand, Pollux reduces the average job completion time by 37%–50%. At the same time, Pollux improves finish-time fairness [146] by $1.5\times$–$5.4\times$.
- We show that, in cloud environments, using goodput-driven auto-scaling based on Pollux can potentially reduce the cost of training large models by 25%.

3.2 Background on Distributed DL Training Performance

An introduction to DL training is presented in §2.2.1. This section presents background on DL training performance in terms of its system throughput and statistical efficiency.

3.2.1 System Throughput

The *system throughput* of DL training can be defined as the number of training samples processed per unit of wall-clock time. When a DL job is distributed across several nodes, its system throughput is determined by several factors, including (1) the allocation and placement of resources (e.g. GPUs) assigned to the job, (2) the method of distributed execution and synchronization, and (3) the batch size used by the SGD algorithm.

The run-time of each SGD training iteration is determined by two main components. *First*, the time spent computing each GPU's local gradient estimate $\hat{g}_k^{(t)}$, which we denote by T_{grad}. *Second*, the time spent averaging $\hat{g}_k^{(t)}$ (e.g. using collective all-reduce [166, 192]) and/or synchronizing $w^{(t)}$ (e.g. using parameter servers [39, 51, 91, 175]) across all GPUs, which we denote by T_{sync}. T_{sync} is influenced by the size of the gradients, performance of the network, and is typically shorter when the GPUs are co-located within the same physical node or rack rather than spread across different nodes or racks.

Limitations due to the batch size

When the number of GPUs is increased, T_{grad} decreases due to a smaller per-GPU batch size. On the other hand, T_{sync}, which is typically independent of the batch size, remains unchanged. By Amdahl's Law, no matter how many GPUs are used, the run-time of each training iteration is lower bounded by T_{sync}. To overcome this scalability limitation, a common strategy is to increase the batch size. Doing so causes the local gradient estimates to be computed over more training examples and thereby increasing the ratio of T_{grad} to T_{sync}. As a result, using a larger batch size enables higher system throughput when scaling to more GPUs in the synchronous data-parallel setting.

3.2.2 Statistical Efficiency

The *statistical efficiency* of DL training can be defined as the amount of training progress made per unit of training data processed, influenced by parameters such as *batch size* or *learning rate*; for example, a larger batch size normally decreases the statistical efficiency. The ability to predict statistical efficiency is key to improving said statistical efficiency, because we can use the predictions to better adapt the batch sizes and learning rates.

Gradient noise scale

Previous work [105, 150] relate the statistical efficiency of DL training to the *gradient noise scale* (GNS), which measures the noise-to-signal ratio of the stochastic gradient. A larger GNS means that training parameters such as the batch size and learning rate can be increased to higher values

with relatively less reduction of the statistical efficiency. The GNS can vary greatly between different DL models [73]. It is also non-constant and tends to gradually increase during training, by up to $10\times$ or more [150]. Thus, it is possible to attain significantly better statistical efficiency for large batch sizes later on during training.

The gradient noise scale mathematically captures an intuitive explanation of how the batch size affects statistical efficiency. When the stochastic gradient has low noise, adding more training examples to each mini-batch does not significantly improve each gradient estimate, which lowers statistical efficiency. When the stochastic gradient has high noise, adding more training examples to each mini-batch reduces the noise of each gradient estimate, which maintains high statistical efficiency. Near convergence, the stochastic gradients have relatively lower signal than noise, and so larger batch sizes can be more useful later in training.

Learning rate scaling

When training with an increased total batch size M, the learning rate η should also be increased, otherwise the final trained model quality/accuracy can be significantly worse [193]. How to increase the learning rate varies between different models and training algorithms (e.g. SGD, Adam [114], AdamW [140]), and several well-established scaling rules may be used. For example, the linear scaling rule [76], which prescribes that η be scaled proportionally with M, or the square-root scaling rule [120, 222] (commonly used with Adam), which prescribes that η be scaled proportionally with $\sqrt{M}$. More recent scaling rules such as AdaScale [105] may scale the learning rate adaptively during training.

In addition to decreasing statistical efficiency, using large batch sizes may also degrade the final model quality in terms of validation performance [73, 111, 197], although the reasons behind this effect are not completely understood at the time of this thesis. However, for each of the learning rate scaling rules mentioned above, there is usually a problem-dependent range of batch sizes that achieve similar validation performances. Within these ranges, the batch size may be chosen more freely without significantly degrading the final model quality.

3.2.3 Existing DL Schedulers

We broadly group existing DL schedulers into two categories, to put Pollux in context. First, *non-scale-adaptive* schedulers are agnostic to the performance scalability of DL jobs with respect to the amount of allocated resources. For example, Tiresias [78] requires users to specify the number of GPUs at the time of job submission, which will be fixed for the lifetime of the job. Gandiva [214] also requires users to specify number of GPUs, but enhances resource utilization through fine-grained time sharing and job packing. Although Gandiva may dynamically change the number of GPUs used by a job, it does so opportunistically and not based on knowledge of job scalability.

Second, *scale-adaptive* schedulers automatically decide the amount of resources allocated to each job based on how well they can be utilized to speed up the job. For example, Optimus [170] learns a predictive model for the system throughput of each job given various amounts of resources, and optimizes cluster-wide resource allocations to minimize the average job completion

time. SLAQ [231], which was not evaluated on DL, uses a similar technique to minimize the average loss values for training general ML models. Gavel [158] goes further by scheduling based on a throughput metric that is comparable across different accelerator types.[1] AntMan [215] uses dynamic scaling and fine-grained GPU sharing to improve cluster utilization, resource fairness, and job completion times. Themis [146] introduces the notion of finish-time fairness, and promotes fairness between multiple DL applications with a two-level scheduling architecture.

Crucially, existing schedulers are agnostic to the statistical efficiency of DL training and the inter-dependence of resource decisions and training parameters. Pollux explicitly co-adapts these inter-dependent values to improve goodput for DL jobs.

3.3 The Goodput of DL Training and Pollux

In this section, we define the *goodput*[2] of DL jobs, which is a measure of training performance that takes into account both system throughput and statistical efficiency. We then describe how the goodput can be measured during training and used as a predictive model, which is leveraged by Pollux to jointly optimize cluster-wide resource allocations and batch sizes.

Definition 3.3.1. (Goodput) The *goodput* of a DL training job at iteration t is the product between its system throughput and its statistical efficiency at iteration t,

$$\texttt{GOODPUT}_t(\star) = \texttt{THROUGHPUT}(\star) \times \texttt{EFFICIENCY}_t\left(M(\star)\right), \tag{3.1}$$

where $\star$ represents any configuration parameters that jointly influence the throughput and batch size during training, and M is the total batch size summed across all allocated GPUs.

While the above definition is general across many training systems, we focus on three configuration parameters of particular impact in the context of efficient resource scheduling, i.e. $\star = (a, m, s)$, where:

- $a \in \mathbb{Z}^N$: the *allocation vector*, where a_n is the number of GPUs allocated from node n.
- $m \in \mathbb{Z}$: the *per-GPU batch size*.
- $s \in \mathbb{Z}$: number of *gradient accumulation steps* (§3.3.2).

The total batch size is then defined as

$$M(a, m, s) = \texttt{SUM}(a) \times m \times (s + 1).$$

We note that while THROUGHPUT is affected by the resource allocation, per-GPU batch size, and the number of gradient accumulation steps, $\texttt{EFFICIENCY}_t$ is only affected by the total batch size M. Thus, M is the key parameter considered by Pollux that affects both the system throughput and statistical efficiency.

[1] Pollux's current throughput model does not consider accelerator heterogeneity. We believe that extending with Gavel's metric would allow Pollux to co-adapt for goodput in heterogeneous DL clusters.

[2] Our notion of goodput for DL is analogous to the traditional definition of goodput in computer networks, ie. the *useful* portion of throughput as benchmarked by training progress per unit of wall-clock time.

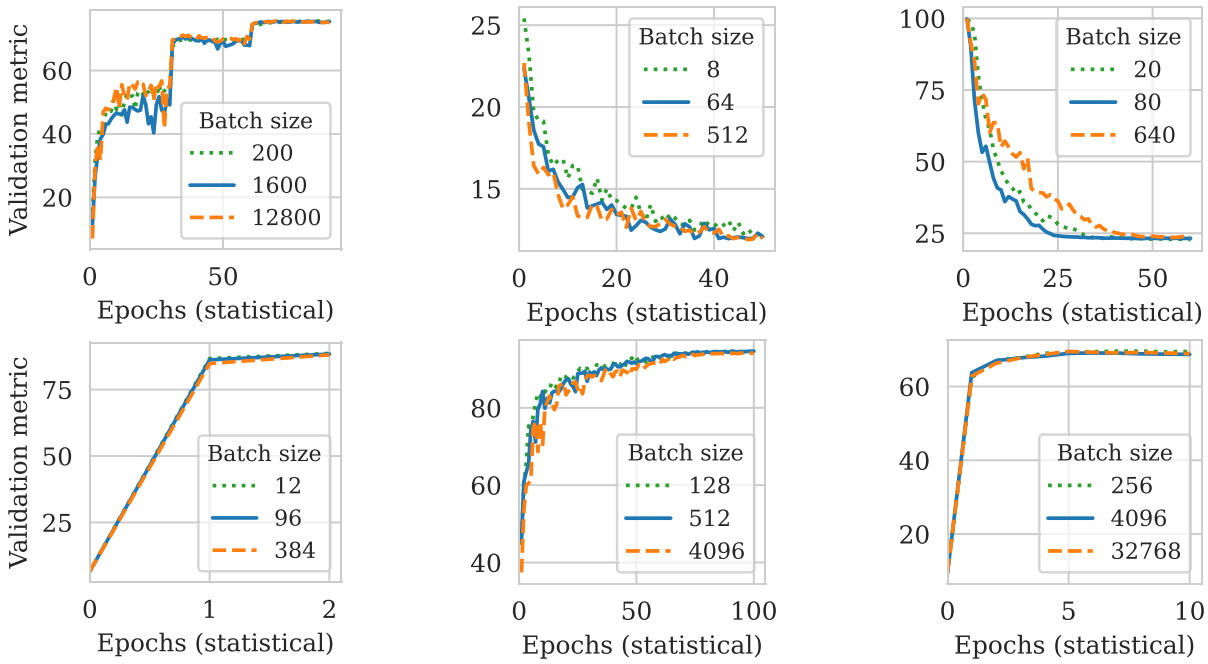


Figure 3.2: Validation metric vs training progress for all models in Table 3.1 using three different batch sizes: M_0, an intermediate batch size, and the max batch size limit we set for each DL task. Metrics are as defined in Table 3.1 except for YoloV3 for which validation loss is shown.

Pollux's approach

An initial batch size M_0 and learning rate (LR) η_0 are selected by the user when submitting their job. Pollux will start each job using a single GPU, $m = M = M_0$, $s = 0$, and $\eta = \eta_0$. As the job runs, Pollux profiles its execution to learn and refine predictive models for both `THROUGHPUT` (§3.3.2) and `EFFICIENCY` (§3.3.1). Using these predictive models, Pollux periodically re-tunes (a, m, s) for each job, according to cluster-wide resource availability and performance (§3.4.2).

$\texttt{EFFICIENCY}_t$ is measured *relative to* the initial batch size M_0 and learning rate η_0, and Pollux only considers batch sizes that are at least the initial batch size, ie. $M \geq M_0$. In this scenario, $\texttt{EFFICIENCY}_t(M)$ is a fraction (between 0 and 1) relative to $\texttt{EFFICIENCY}_t(M_0)$. Therefore, goodput can be interpreted as the portion of the throughput that is useful for training progress, being equal to the throughput if and only if perfect statistical efficiency is achieved.

Plug-in Learning Rate Scaling

Recall from §3.2.2 that different training jobs may require different learning rate scaling rules to adjust η in response to changes in M. In order to support a wide variety of LR scaling rules, including state-of-the-art rules such as AdaScale [105], Pollux provides a plug-in interface that can be implemented using a function signature

$$\texttt{SCALE\_LR}(M_0, M) \longrightarrow \lambda.$$

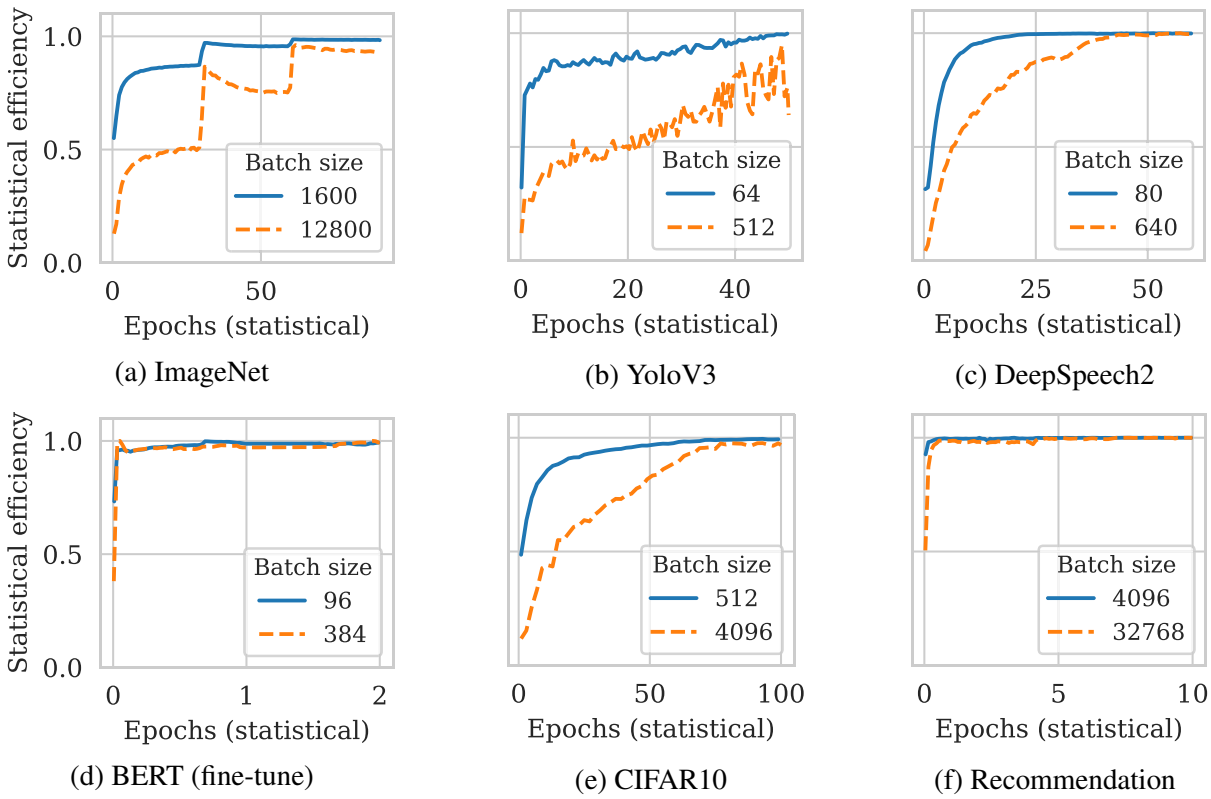
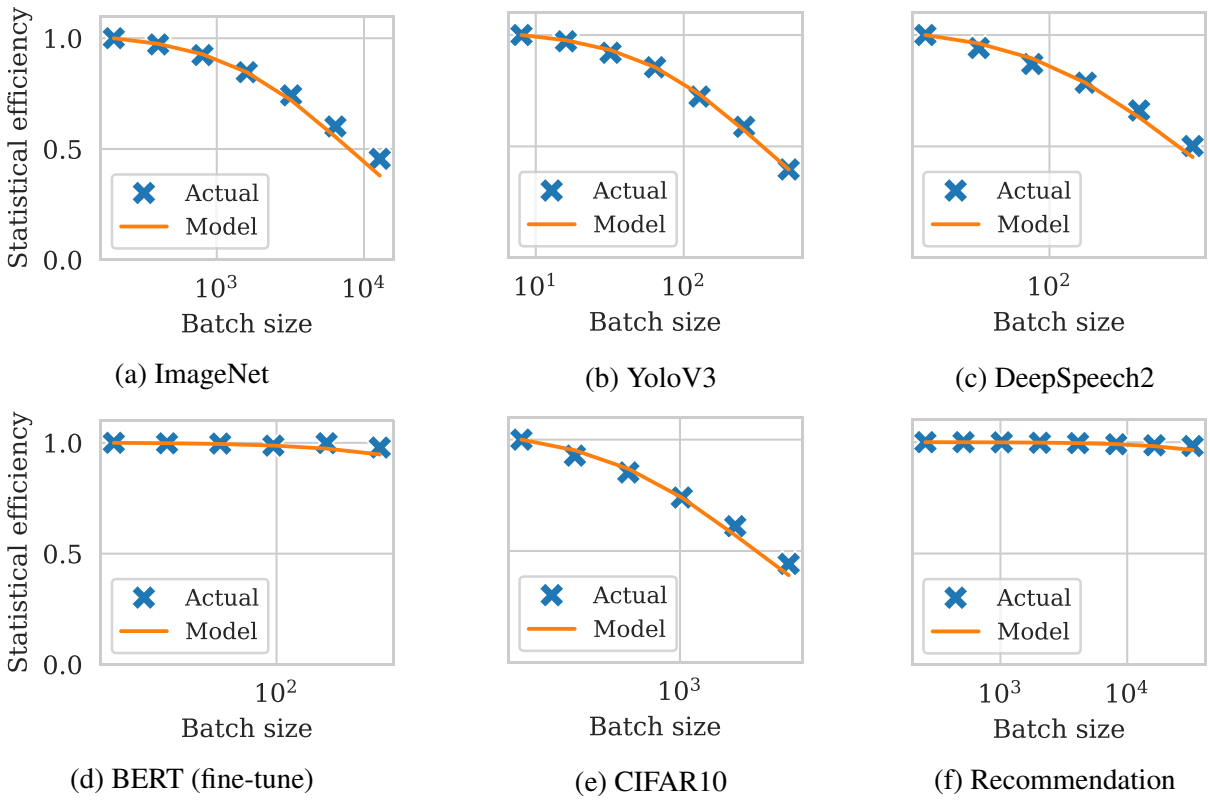


Figure 3.3: Measured statistical efficiency vs. training progress using two different batch sizes. Training progress (x-axis) in the top two rows is shown in terms of "statistical epochs", defined as $\frac{M}{|X|} \sum_t \texttt{EFFICIENCY}_t(M)$ where $|X|$ is the size of the training dataset.



(a) ImageNet (b) YoloV3 (c) DeepSpeech2

(d) BERT (fine-tune) (e) CIFAR10 (f) Recommendation

Figure 3.4: Measured $\texttt{EFFICIENCY}_t$ vs. predicted $\texttt{EFFICIENCY}_t$ using a range of batch sizes (log-scaled), using φ_t measured using the median batch size from each range, during an early-training epoch (roughly 1/8th of the way through training).

SCALE_LR is called before every model update step, and λ is used by Pollux to scale the learning rate. The implementation of SCALE_LR can utilize metrics collected during training, such as the gradient noise scale. Using this interface, one can implement rules including AdaScale, square-root scaling [120], linear scaling [76] and LEGW [222].

3.3.1 Modeling Statistical Efficiency

We model $\texttt{EFFICIENCY}_t(M)$ as the amount of progress made per training example using M, relative to using M_0. For SGD-based training, this quantity can be expressed in terms of the gradient noise scale (GNS) [150]. To support popular adaptive variants of SGD like Adam [114] and AdaGrad [210], we use the *pre-conditioned gradient noise scale* (PGNS), derived by closely following the original derivation of the GNS ("simple" noise scale in [150]) starting from pre-conditioned SGD[3] rather than vanilla SGD. The PGNS, which we denote by φ_t, is expressed as

$$\varphi_t = \frac{\mathrm{tr}(P\Sigma P^T)}{|Pg|^2}, \tag{3.2}$$

where g is the true gradient, P is the pre-conditioning matrix of the adaptive SGD algorithm, and Σ is the covariance matrix of per-example stochastic gradients. The PGNS is a generalization of the GNS and is mathematically equivalent to the GNS for the special case of vanilla SGD. A detailed derivation of φ_t can be found in Appendix A.1.

Similar to the GNS (Appendix D of [150]), it takes $1 + \varphi_t/M$ training iterations to make a similar amount of training progress across different batch sizes M. Therefore, we can use the PGNS φ_t to define a concrete expression for $\texttt{EFFICIENCY}_t(M)$ as

$$\texttt{EFFICIENCY}_t(M) = \frac{\varphi_t + M_0}{\varphi_t + M}. \tag{3.3}$$

Intuitively, Eqn. 3.3 measures the contribution from each training example to the overall progress. If $\texttt{EFFICIENCY}_t(M) = E$, then (1) $0 < E \leq 1$, and (2) training using batch size M will need to process $1/E$ times as many training examples to make the same progress as using batch size M_0. During training, Pollux estimates φ_t, then uses Eqn 3.3 to predict the $\texttt{EFFICIENCY}_t$ at different batch sizes. The measured value of φ_t varies according to the training progress at iteration t, thus $\texttt{EFFICIENCY}_t(M)$ reflects the lifetime-dependent trends exhibited by the true statistical efficiency.

Fig. 3.2 shows the validation metrics on a held-out dataset for a variety of DL training tasks (details in Table 3.1) versus their training progress. "Statistical epochs"[4] is the number of training iterations normalized by $\texttt{EFFICIENCY}_t$ so that each statistical epoch makes theoretically, as projected by our model, the same training progress across different batch sizes. Thus, the degree of similarity between validation curves at different batch sizes is an indicator for the accuracy of $\texttt{EFFICIENCY}_t$ as a predictor of actual training progress.

[3]Pre-conditioned SGD optimizes $\mathcal{L}(Pw)$ instead of $\mathcal{L}(w)$, where P is known as a pre-conditioning matrix. Adaptive variants of SGD such as Adam and AdaGrad may be viewed as vanilla SGD (with momentum) applied together with a particular pre-conditioning matrix P.

[4]Similar to the notion of "scale-invariant iterations" defined in [105].

Although there are differences in the validation curves for several DL tasks (especially in earlier epochs), they achieve similar best values across the different batch sizes we evaluated ($\pm 1\%$ relative difference for all tasks except DeepSpeech2 at $\pm 4\%$). We note that these margins are within the plateau of high-quality models expected from large-batch training [149].

Fig. 3.3 and Fig. 3.4 show the measured and predicted $\texttt{EFFICIENCY}_t$ during training and for a range of different batch sizes. In general, larger batch sizes have lower $\texttt{EFFICIENCY}_t$ early in training, but close the gap later on in training. The exceptions being BERT, which is a fine-tuning task starting from an already pre-trained model, and recommendation, which uses a much smaller and shallower model architecture than the others. How $\texttt{EFFICIENCY}_t$ changes during training varies from task to task, and depends on specific properties like the learning rate schedule. For example, $\texttt{EFFICIENCY}_t$ for ImageNet, which uses step-based learning rate annealing, experiences sharp increases whenever the learning rate is annealed.

Finally, we note that the $\texttt{EFFICIENCY}_t$ function (which is supplied with estimates of φ_t by Pollux) is able to accurately model observed values at a range of different batch sizes. This means that φ_t measured using batch size M can be used by Pollux to predict the value of $\texttt{EFFICIENCY}_t$ at a different batch size M' without needing to train using M' ahead of time.

Upper batch size limit

In some cases, as the batch size increases, the chosen LR scaling rule may break down before the statistical efficiency decreases, which degrades the final model quality. To address these cases, the application may define a maximum batch size limit that will be respected by Pollux. Nevertheless, we find that a batch size up to $32\times$ larger works well in most cases. Furthermore, limits for common models are well-studied for popular LR scaling rules [76, 105, 193, 222]. As better LR scaling rules are developed, they may be incorporated into Pollux using its plug-in interface (§3.3).

Estimating φ_t

The PGNS φ_t can be estimated in a similar fashion as the GNS by following Appendix A.1 of [150], except using the pre-conditioned gradient Pg instead of the gradient g. This can be done efficiently when there are multiple data-parallel processes by using the different values of $\hat{g}_k^{(t)}$ already available on each GPU k. However, this method doesn't work when there is only a single GPU (and gradient accumulation is off, i.e. $s = 0$). In this particular situation, Pollux switches to a differenced variance estimator [209] which uses consecutive gradient estimates $\hat{g}^{(t-1)}$ and $\hat{g}^{(t)}$.

3.3.2 Modeling System Throughput

To model and predict the system throughput for data-parallel DL, we aim to predict the time spent per training iteration, T_{iter}, and then calculate the throughput as

$$\texttt{THROUGHPUT}(a, m, s) = M(a, m, s)/T_{iter}(a, m, s). \tag{3.4}$$

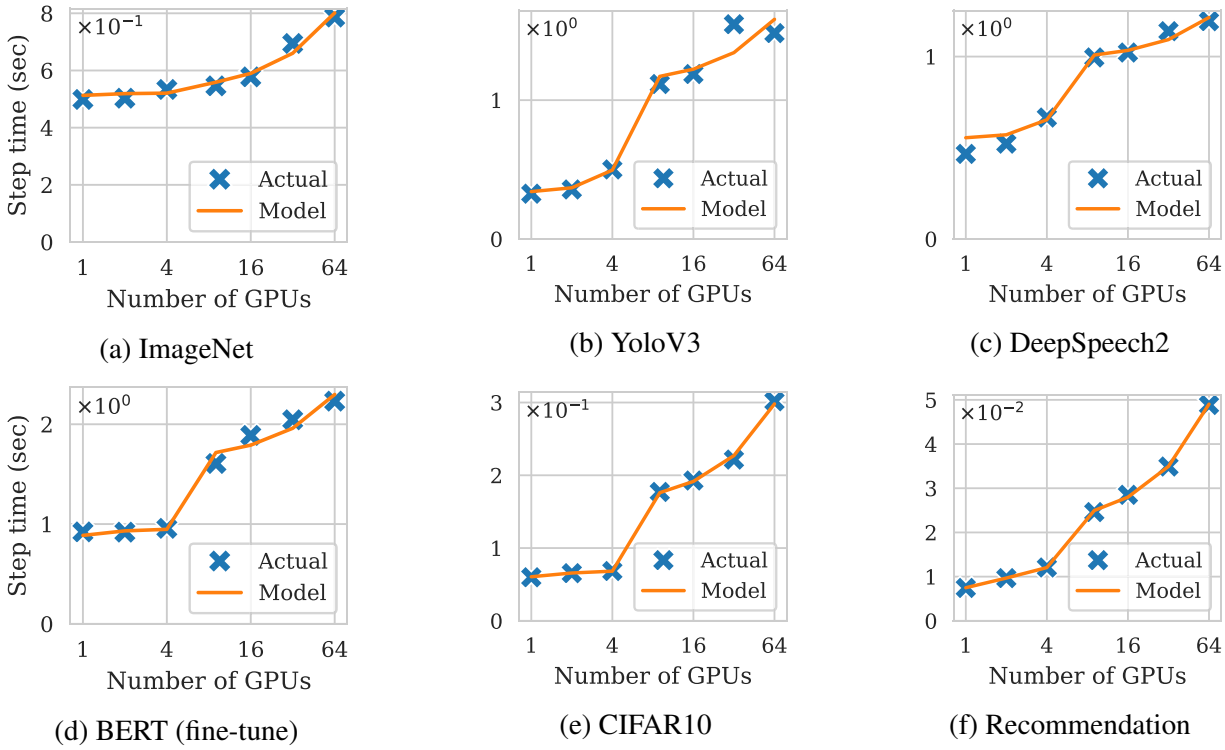


Figure 3.5: System throughput for all models described in Table 3.1, as measured using g4dn.12xlarge instances in AWS each with 4 NVIDIA T4 GPUs and created within the same placement group. Eqn. 3.8 was fitted using the observed data that appeared in each plot. Time per training iteration vs. the number of allocated GPUs (log-scaled), with the per-GPU batch size held constant. The GPUs are placed in as few 4-GPU nodes as possible, which causes a sharp increase beyond 4 GPUs (when inter-node network synchronization becomes required).

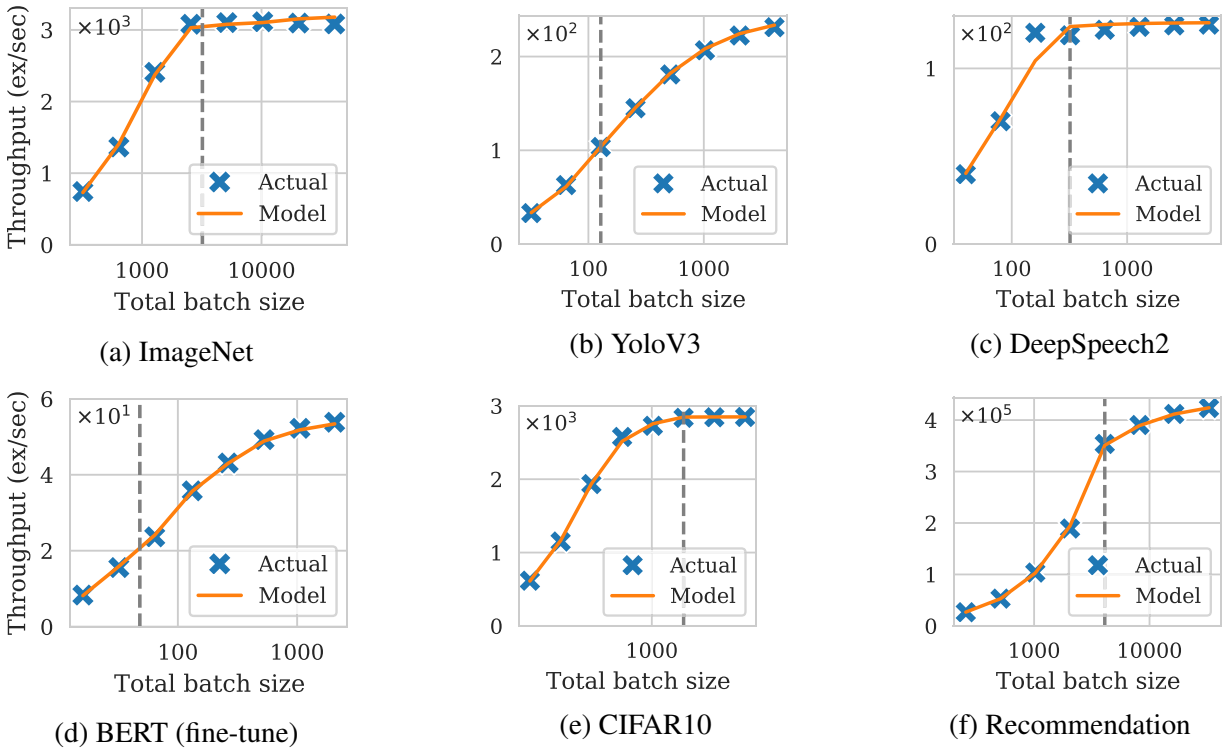


Figure 3.6: System throughput (examples per second) vs. total batch size (log-scaled), with the number of GPUs held constant. To the left of the vertical dashed line, the entire mini-batch fits within GPU memory. To the right, the total batch size is achieved using gradient accumulation.

We start by separately modeling T_{grad}, the time in each iteration spent computing local gradient estimates, and T_{sync}, the time in each iteration spent averaging gradient estimates and synchronizing model parameters across all GPUs. We also start by assuming no gradient accumulation, i.e. $s = 0$.

Modeling T_{grad}

The local gradient estimates are computed using back-propagation, whose run-time scales linearly with the per-GPU batch size m. Thus, we model T_{grad} as

$$T_{grad}(m) = \alpha_{grad} + \beta_{grad} \cdot m, \tag{3.5}$$

where $\alpha_{grad}, \beta_{grad}$ are fittable parameters.

Modeling T_{sync}

When allocated a single GPU, no synchronization is needed and $T_{sync} = 0$. Otherwise, we model T_{sync} as a linear function of the number of GPUs since in data-parallelism, the amount of data sent and received from each replica is typically only dependent on the size of the gradients and/or parameters. We include a linear factor to account for performance retrogressions associated with using three or more GPUs, such as increasing likelihood of stragglers or network delays.

Co-location of GPUs on the same node reduces network communication, which can improve T_{sync}. Thus, we use different parameters depending on GPU placement. Letting $K = \texttt{SUM}(a)$ be the number of allocated GPUs,

$$T_{sync}(a, m) = \begin{cases} 0 & \text{if } K = 1 \\ \alpha_{sync}^{local} + \beta_{sync}^{local} \cdot (K - 2) & \text{if } N = 1, K \geq 2 \\ \alpha_{sync}^{node} + \beta_{sync}^{node} \cdot (K - 2) & \text{otherwise,} \end{cases} \tag{3.6}$$

where N is the number of physical nodes occupied by at least one replica. α_{sync}^{local} and β_{sync}^{local} are the constant and retrogression parameters for when all processes are co-located onto the same node. α_{sync}^{node} and β_{sync}^{node} are the analogous parameters for when at least two process are located on different nodes. Note that our model for T_{sync} can be extended to account for rack-level locality by adding a third pair of parameters.

Combining T_{grad} and T_{sync}

Modern DL frameworks can partially overlap T_{grad} and T_{sync} by overlapping gradient computation with network communication [229]. The degree of this overlap depends on structures in the specific DL model being trained, like the ordering and sizes of its layers.

Assuming no overlap, then $T_{iter} = T_{grad} + T_{sync}$. Assuming perfect overlap, then $T_{iter} = \max(T_{grad}, T_{sync})$. A realistic value of T_{iter} is somewhere in between these two extremes. To capture the overlap between T_{grad} and T_{sync}, we model T_{iter} as

$$T_{iter}(a, m, 0) = \left(T_{grad}(a, m)^{\gamma} + T_{sync}(a)^{\gamma}\right)^{1/\gamma}, \tag{3.7}$$

where $\gamma \geq 1$ is a learnable parameter. Eqn. 3.7 has the property that $T_{iter} = T_{grad} + T_{sync}$ when $\gamma = 1$, and smoothly transitions towards $T_{iter} = \max(T_{grad}, T_{sync})$ as $\gamma \to \infty$.

Gradient Accumulation

In data-parallelism, GPU memory limits the per-GPU batch size, and many DL models hit this limit before the batch size is large enough for T_{grad} to overcome T_{sync} (or experience diminishing statistical efficiency), resulting in suboptimal scalability. Several techniques exist for overcoming the GPU memory limit [40, 46, 94, 100]; we focus on gradient accumulation, which is easily implemented using popular DL frameworks. Per-GPU gradients are aggregated locally over s forward-backward passes before being synchronized across all GPUs during the $(s+1)^{\text{th}}$ pass, achieving a larger total batch size. Thus, one iteration of SGD spans s accumulation steps followed by one synchronization step, modeled as

$$T_{iter}(a, m, s) = s \times T_{grad}(a, m) + \left(T_{grad}(a, m)^{\gamma} + T_{sync}(a)^{\gamma}\right)^{1/\gamma}. \tag{3.8}$$

Throughput model validation

Fig. 3.5 and Fig. 3.6 show an example of our `THROUGHPUT` function fit to measured throughput values for a range of resource allocations and batch sizes. Each DL task was implemented using PyTorch [166], which overlaps the backward pass' computation and communication. Gradients are synchronized with NCCL 2.7.8, which uses either ring all-reduce or tree all-reduce depending on the detected GPUs and their placements and its own internal performance estimates. Overall, we find that our model can represent the observed data closely, while varying both the amount of resources as well as the batch size. In particular, all models we measured except ImageNet exhibited high sensitivity to inter-node synchronization, indicating that they benefit from co-location of GPUs. Furthermore, YOLOv3 and BERT benefit from using gradient accumulation to increase their total batch sizes. These detailed characteristics are well-represented by our `THROUGHPUT` function, and can be optimized for by Pollux.

In addition to the configurations in Fig. 3.5 and Fig. 3.6, we fitted the `THROUGHPUT` function on a diverse set of GPU placements and batch sizes in a 64-GPU cluster. Across all DL tasks, the average error of the fitted model was at most 10%, indicating that it represents the observed throughput measurements well.

Limits of the throughput model

Pollux models data-parallel training throughput only in the dimensions it cares about, i.e. number and co-locality of GPUs, batch size, and gradient accumulation steps. The simple linear assumptions made in Eqn. 3.8, although sufficiently accurate for the settings we tested, may diverge from reality for specialized hardware [106], sophisticated synchronization algorithms [37, 211, 235], different parallelization strategies [95, 157, 195, 196], at larger scales [34, 219], or hidden resource contention not related to network used for gradient synchronization. Rather than attempting to cover all scenarios with a single throughput model, we designed $\texttt{GOODPUT}_t$ (Eqn. 3.1) to be modular so that different equations for `THROUGHPUT` may be easily plugged in without interfering with the core functionalities provided by Pollux.

3.4 Pollux Design and Architecture

Pollux adapts DL job execution at two distinct granularities. First, at a job-level granularity, Pollux dynamically tunes the batch size and learning rate for best utilization of the allocated resources. Second, at the cluster-wide granularity, Pollux dynamically (re-)allocates resources, driven by the goodput of all jobs sharing the cluster combined with cluster-level goals including fairness and job-completion time. To achieve this co-adaptivity in a scalable way, Pollux's design consists of two primary components.

First, a *PolluxAgent* runs together with each job. It fits the $\texttt{EFFICIENCY}_t$ and THROUGHPUT functions for that job, and tunes its batch size and learning rate for efficient utilization of its current allocated resources. PolluxAgent periodically reports the goodput function of its job to the PolluxSched.

Second, the *PolluxSched* periodically optimizes the resource allocations for all jobs in the cluster, taking into account the current goodput function for each job and cluster-wide resource contention. Scheduling decisions made by PolluxSched also account for the overhead associated with resource re-allocations, slowdowns due to network interference between multiple jobs, and resource fairness.

PolluxAgent and PolluxSched *co-adapt* to each other. While PolluxAgent adapts each training job to make efficient use of its allocated resources, PolluxSched dynamically re-allocates each job's resources, taking into account the PolluxAgent's ability to tune its job.

Fig. 3.7 illustrates Pollux's co-adaptive architecture (Fig. 3.7c), compared with existing schedulers which are either non-scale-adaptive (Fig. 3.7a), or scale-adaptive without being involved with statistical efficiency (Fig. 3.7b).

3.4.1 PolluxAgent: Job-level Optimization

An instance of PolluxAgent is started with each training job. During training, it continually measures the job's gradient noise scale and system throughput, and it reports them to PolluxSched at a fixed interval. It also uses this information to determine the most efficient batch size for its job given its current resource allocations, and adapts its job's learning rate to this batch size using the appropriate plug-in LR scaling rule (e.g. AdaScale for SGD or square-root scaling for Adam).

Online model fitting

In §3.3.2, we defined the system throughput parameters of a training job as the 7-tuple

$$\theta_{sys} = \left(\alpha_{grad}, \beta_{grad}, \alpha_{sync}^{local}, \beta_{sync}^{local}, \alpha_{sync}^{node}, \beta_{sync}^{node}, \gamma\right), \tag{3.9}$$

which are required to construct the THROUGHPUT function. Together with the PGNS φ_t (for predicting $\texttt{EFFICIENCY}_t$) and initial batch size M_0, the triple $(\theta_{sys}, \varphi_t, M_0)$ specifies the GOODPUT function. While M_0 is a constant configuration provided by the user, and φ_t can be computed according to §3.3.1, θ_{sys} is estimated by fitting the THROUGHPUT function to observed throughput values collected about the job during training.

PolluxAgent measures the time taken per iteration, T_{iter}, and records the tuple (a, m, s, T_{iter}) for all combinations of resource allocations a, per-GPU batch size m, and gradient accumulation

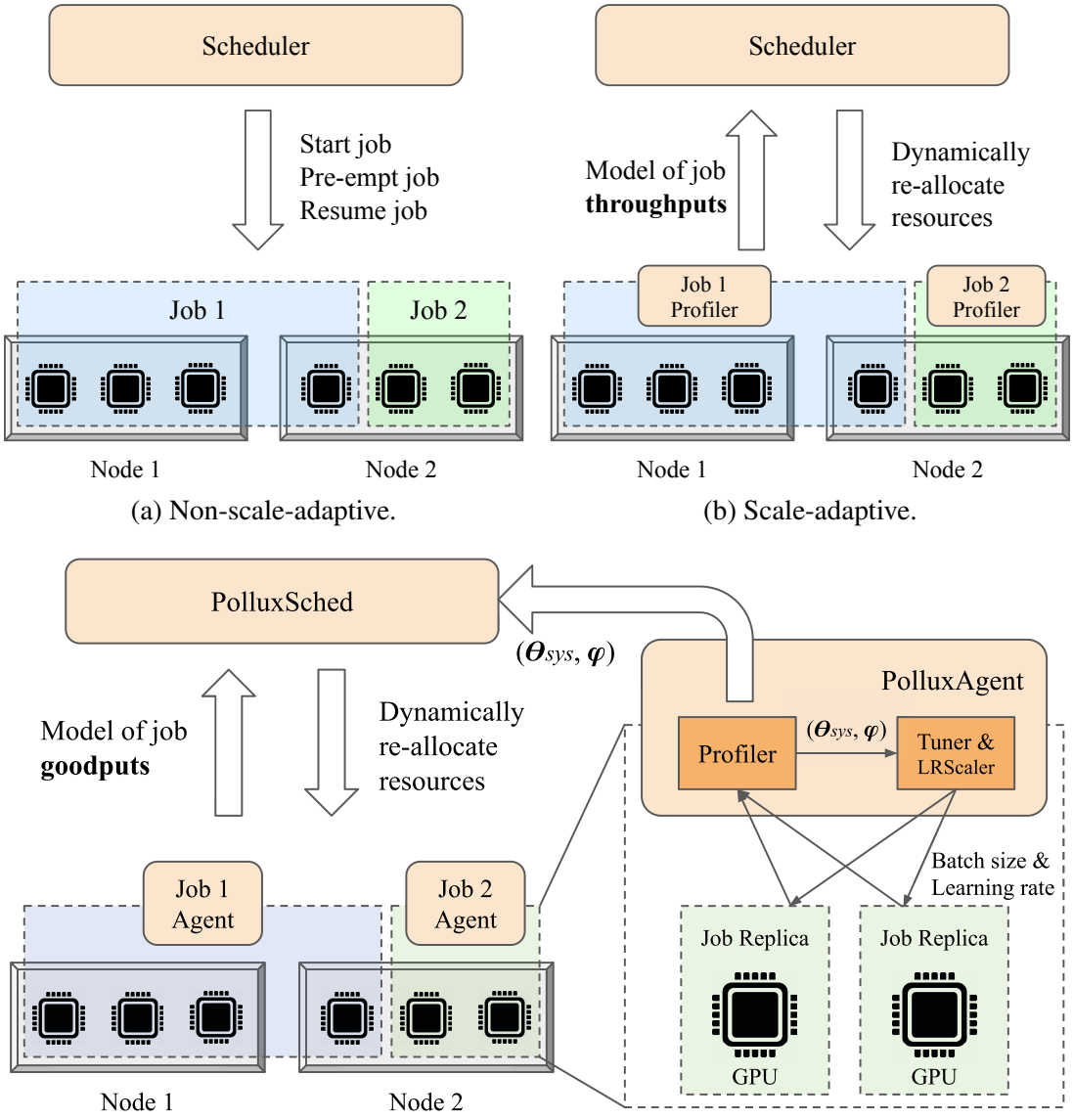


Figure 3.7: Architecture of Pollux (Fig. 3.7c), compared with existing schedulers which are either non-scale-adaptive (Fig. 3.7a) or scale-adaptive (Fig. 3.7b).

steps s encountered during its lifetime. Periodically, PolluxAgent fits the parameters θ_{sys} to all of the throughput data collected so far. Specifically, we minimize the root mean squared logarithmic error (RMSLE) between Eqn. 3.8 and the collected data triples, using L-BFGS-B [236]. We set constraints for each α and β parameter to be non-negative, and γ to be in the range $[1, 10]$. PolluxAgent then reports the updated values of θ_{sys} and φ_t to PolluxSched.

Prior-driven exploration

At the beginning of each job, throughput values have not yet been collected. To ensure that Pollux finds efficient resource allocations through systematic exploration, we impose several priors which bias θ_{sys} towards the belief that throughput scales perfectly with more resources, until such resource configurations are explored.

In particular, we set $\alpha_{sync}^{local} = 0$ while the job had not used more than one GPU, $\alpha_{sync}^{local} = \beta_{sync}^{local} = 0$ while the job had not used more than one node, and $\beta_{sync}^{local} = \beta_{sync}^{node} = 0$ while the job had not used more than two GPUs. This creates the following behavior: each job starts with a single GPU and is initially assumed to scale perfectly to more GPUs. PolluxSched is then encouraged to allocate more GPUs and/or nodes to the job, naturally as part of its resource optimization (§3.4.2), until the PolluxAgent can estimate θ_{sys} more accurately. Finally, to prevent a job from being immediately scaled out to arbitrarily many GPUs, we restrict the maximum number of GPUs that can be allocated to at most twice the maximum number of GPUs the job has been allocated in its lifetime.

Although other principled approaches to exploration can be applied (e.g., Bayesian optimization), we find that this simple prior-driven strategy is sufficient in our experiments. Sec. 3.5.3 shows that prior-driven exploration performs close (within 2-5%) to an idealized scenario in which the model is fitted offline for each job before being submitted to the cluster.

Training job tuning

With θ_{sys}, φ_t, and M_0, which fully specify the DL job's `GOODPUT` function at its current training progress, PolluxAgent determines the most efficient per-GPU batch size and gradient accumulation steps,

$$(m^*,\ s^*) = \arg\max_{m,s}\ \texttt{GOODPUT}(a, m, s), \tag{3.10}$$

where a is the job's current resource allocation.

Once a new configuration is found, the job will use it for its subsequent training iterations, using the plug-in LR scaling rule to adapt its learning rate appropriately. As the job's $\texttt{EFFICIENCY}_t$ function changes over time, PolluxAgent will periodically re-evaluate the most efficient configuration.

3.4.2 PolluxSched: Cluster-wide Optimization

The PolluxSched periodically allocates (and re-allocates) resources for every job in the cluster. To determine a set of efficient cluster-wide resource allocations, it maximizes a *fitness function*

that is defined as a generalized (power) mean across speedups for each job:

$$\texttt{FITNESS}_p(A) = \left(\frac{1}{J} \sum_{j=1}^{J} \texttt{SPEEDUP}_j(A_j)^p \right)^{1/p}. \tag{3.11}$$

A is an *allocation matrix* with each row A_j being the allocation vector for a job j, thus A_{jn} is the number of GPUs on node n allocated to job j, and J is the total number of running and pending jobs sharing the cluster. We define the speedup of each job as the factor of goodput improvement using a given resource allocation over using a fair-resource allocation, ie.

$$\texttt{SPEEDUP}_j(A_j) = \frac{\max_{m,s} \texttt{GOODPUT}_j(A_j, m, s)}{\max_{m,s} \texttt{GOODPUT}_j(a_f, m, s)}, \tag{3.12}$$

where $\texttt{GOODPUT}_j$ is the goodput of job j at its current training iteration, and a_f is a fair resource allocation for the job, defined to be an exclusive $1/J$ share of the cluster.[5]

In §3.3, we described how the GOODPUT function can be fitted to observed metrics during training and then be evaluated as a predictive model. PolluxSched leverages this ability to predict GOODPUT to maximize FITNESS via a search procedure, and then it applies the outputted allocations to the cluster.

Fairness and the effect of p

When $p = 1$, $\texttt{FITNESS}_p$ is the average of SPEEDUP values across all jobs. This causes PolluxSched to allocate more GPUs to jobs that achieve a high SPEEDUP when provided with many GPUs (i.e., jobs that scale well). However, as $p \to -\infty$, $\texttt{FITNESS}_p$ smoothly approaches the minimum of SPEEDUP values, in which case maximizing $\texttt{FITNESS}_p$ promotes equal SPEEDUP between training jobs, but ignores the overall cluster goodput and resource efficiency.

Thus, p can be considered a "fairness knob", with larger negative values being more fair. A cluster operator may select a suitable value, based on organizational priorities. In our experience and results in §3.5, we find that $p = -1$ achieves most goodput improvements and reasonable fairness.

Re-allocation penalty

Each time a job is re-allocated to a different set of GPUs, it incurs some delay to re-configure the training process. Using the the popular checkpoint-restart method, we measured between 15 and 120 seconds of delay depending on the size of the model being trained and other initialization tasks in the training code. To prevent an excessive number of re-allocations, when PolluxSched evaluates the fitness function for a given allocation matrix, it applies a penalty for every job that needs to be re-allocated,

$$\texttt{SPEEDUP}_j(A_j) \longleftarrow \texttt{SPEEDUP}_j(A_j) \times \texttt{REALLOC\_FACTOR}_j(\delta).$$

[5]We note that SPEEDUP has similarities with *finish-time fairness* [146]. However, SPEEDUP is related to training performance at a moment in time, whereas finish-time fairness is related to end-to-end job completion time.

We define $\texttt{REALLOC\_FACTOR}_j(\delta) = (T_j - R_j\delta)/(T_j + \delta)$, where T_j is the age of the training job, R_j is the number of re-allocations incurred by the job so far, and δ is an estimate of the re-allocation delay. Intuitively, $\texttt{REALLOC\_FACTOR}_j(\delta)$ scales $\texttt{SPEEDUP}_j(A_j)$ according to the assumption that the historical average rate of re-allocations for job j will continue indefinitely into the future. Thus, a job that has historically experienced a higher rate of re-allocations will be penalized more for future re-allocations.

Interference avoidance

When multiple distributed DL jobs share a single node, their network usage while synchronizing gradients and model parameters may interfere with each other, causing both jobs to slow down [101]; Xiao et al. [214] report up to 50% slowdown for DL jobs which compete with each other for network resources. PolluxSched mitigates this issue by disallowing different distributed jobs (each using GPUs across multiple nodes) from sharing the same node.

Interference avoidance is implemented as a constraint in Pollux's search algorithm, by ensuring at most one distributed job is allocated to each node. We study the effects of interference avoidance in §3.5.3.

Supporting non-adaptive jobs

In certain cases, a user may want to run a job with a fixed batch size, i.e. $M = M_0$. These jobs are well-supported by PolluxSched, which simply fixes $\texttt{EFFICIENCY}_t$ for that job to 1 and can continue to adapt its resource allocations based solely on its system throughput.

3.4.3 Implementation

PolluxAgent is implemented as a Python library that is imported into DL training code. We integrated PolluxAgent with PyTorch [166], which uses all-reduce as its gradient synchronization algorithm. PolluxAgent inserts performance profiling code that measures the time taken for each iteration of training, as well as calculating the gradient noise scale. At a fixed time interval, PolluxAgent fits the system throughput model (Eqn. 3.7) to the profiled metrics collected so far, and reports the fitted system throughput parameters, along with the latest gradient statistics, to PolluxSched. After reporting to PolluxSched, PolluxAgent updates the job's per-GPU batch size and gradient accumulation steps, by optimizing its now up-to-date goodput function (Eqn. 3.1) with its currently allocated resources.

PolluxSched is implemented as a service in Kubernetes [35]. At a fixed time interval, PolluxSched runs its search algorithm, and then applies the resultant allocation matrix by creating and terminating Kubernetes Pods that run the job workers. To find a good allocation matrix, PolluxSched uses a population-based search algorithm that perturbs and combines candidate allocation matrices to produce higher-value allocation matrices, and finally modifies them to satisfy node resource constraints and interference avoidance. The allocation matrix with the highest fitness score is applied to the jobs running in the cluster.

Both PolluxAgent and PolluxSched require a sub-procedure that optimizes $\texttt{GOODPUT}_t(a, m, s)$ given a fixed a (Eqn. 3.10). We implemented this procedure by first sampling a range of can-

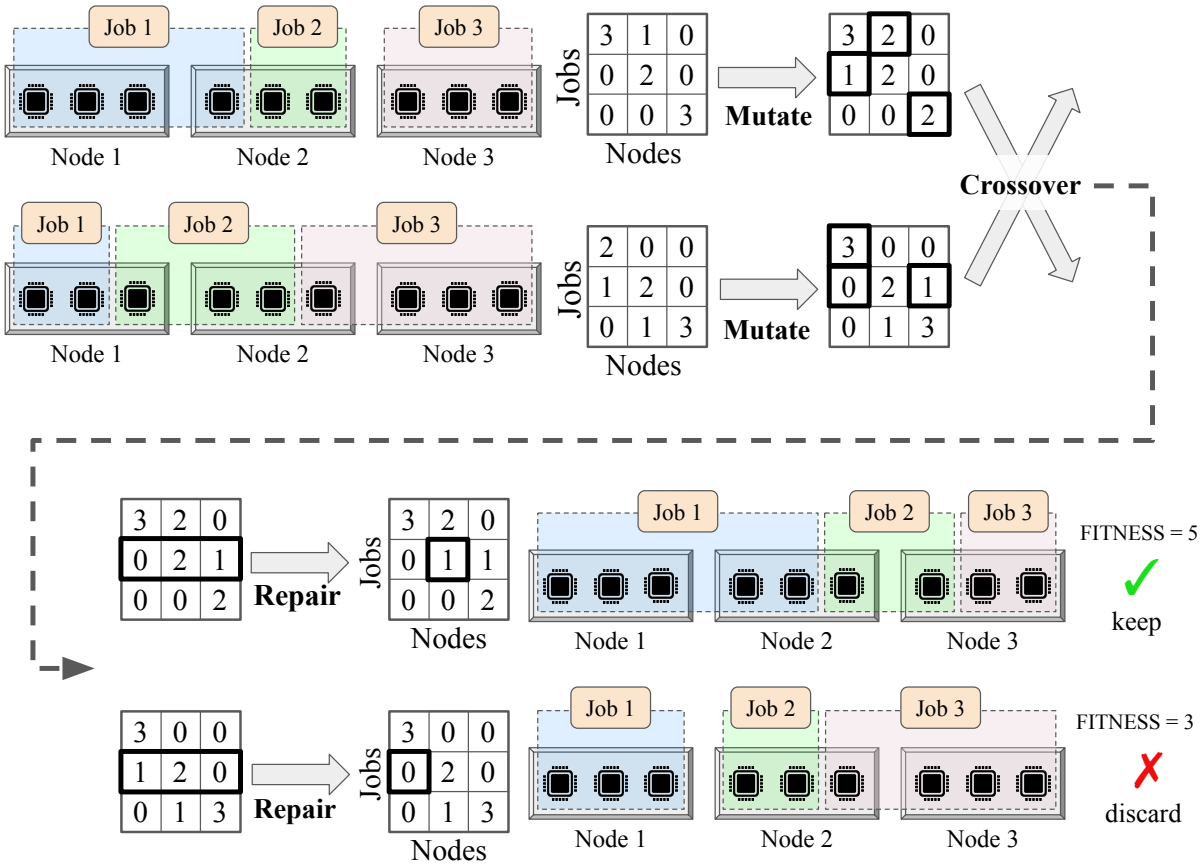


Figure 3.8: The mutation, crossover, and repair operations performed by PolluxSched during each generation of its genetic algorithm.

didate values for the total batch size M, then finding the smallest s such that $m = \lceil M/s \rceil$ fits into GPU memory according to a user-defined upper-bound, and finally taking the configuration which results in the highest `GOODPUT` value. We note that other efficient (but more complex) algorithms may also be used, such as golden-section search [112].

Genetic Algorithm

The search performed by PolluxSched is implemented using a genetic algorithm which operates on a population of distinct allocation matrices (see Fig. 3.8). During each generation of the algorithm, existing allocation matrices are first randomly mutated, then crossed over to produce offspring allocation matrices, and finally modified to satisfy node resource constraints. A constant population size is maintained after each generation by discarding the allocation matrices with the lowest objective values (according to Eqn. 3.11).

After several generations of the genetic algorithm, the allocation matrix with the highest fitness score is applied to the jobs running in the cluster. Although only the allocation matrix with the highest fitness score is applied to the cluster, the entire population is saved and used to

bootstrap the genetic algorithm in the next scheduling interval.

The genetic operations are briefly described below, and the source code of our search algorithm can be found in Appendix A.2. The genetic algorithm is implemented using pymoo [29].

Mutation. Each element A_{jn} is randomly selected to be mutated. When A_{jn} is mutated, it is either reset to 0 or set to a random integer between 1 and the total number of GPUs on node n.

Crossover. When two allocation matrices are crossed over, their rows are mixed. The offspring allocation matrix consists of job allocations which are randomly selected between its two parent allocation matrices. In each generation, the allocation matrices which participate in crossover are picked using tournament selection [154].

Repair. After the mutation and crossover operations, the resultant allocation matrices may no longer satisfy resource constraints, and try to request more GPUs than are available on a node. To address this issue, random elements are decremented within columns of the allocation matrix that correspond to over-capacity nodes, until the GPU resource constraints are satisfied. Interference avoidance is also implemented in this step by also removing distributed jobs from shared nodes until at most one distributed job is allocated to each node.

3.5 Evaluation of Pollux

We compare Pollux with two state-of-the-art DL schedulers using a testbed cluster with 64 GPUs. Although one primary advantage of Pollux is automatically selecting the configurations for each job, we find that Pollux still reduces average job completion times by 37–50% even when the baseline schedulers are supplied with well-tuned job configurations (a scenario that strongly favors the baseline schedulers). Pollux is able to dynamically adapt each job by trading-off between high-throughput/low-efficiency and low-throughput/high-efficiency modes of training, depending on the current cluster state and training progress.

Using a cluster simulator, we evaluate the impact of specific settings on Pollux, including the total workload intensity, prior-driven exploration, scheduling interval, and interference avoidance. With its fairness knob, Pollux can improve finish-time fairness [146] by 1.5–$5.4\times$ compared to baseline DL schedulers. We also reveal a new opportunity for auto-scaling in the cloud by showing that a Pollux-based auto-scaler can potentially reduce the cost of training large models (e.g. ImageNet) by 25%.

3.5.1 Experimental Setup

Testbed

We conduct experiments using a cluster consisting of 16 nodes and 64 GPUs. Each node is an AWS EC2 `g4dn.12xlarge` instance with 4 NVIDIA T4 GPUs, 48 vCPUs, 192GB memory, and a 900GB SSD. All instances are launched within the same placement group. We deployed

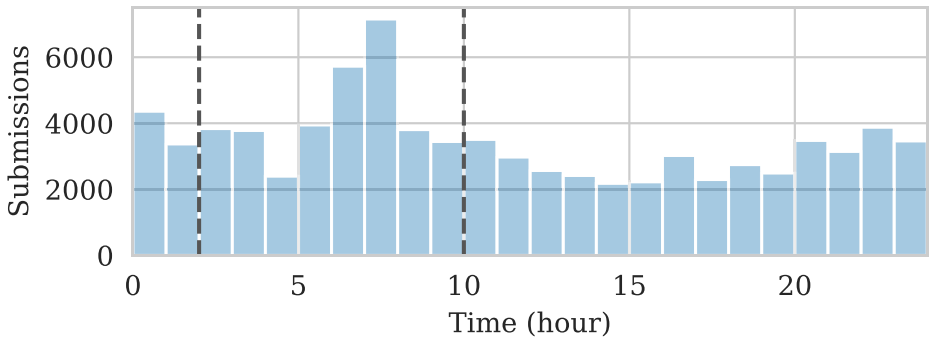


Figure 3.9: Number of job submissions during each hour of the day in the Microsoft trace. Our primary synthetic workload is sampled from the interval between the dashed lines.

Kubernetes 1.18.2 on this cluster, along with CephFS 14.2.8 to store checkpoints for checkpoint-restart elasticity.

Synthetic Workload Construction

We randomly sampled 160 jobs from the busiest 8-hour range (hours 3–10) in the deep learning cluster traces published by Microsoft [101], as shown in Fig. 3.9. Each job in the orginal trace has information on its submission time, number of GPUs, and duration. However, no information is provided on the model architectures being trained or dataset characteristics. Instead, our synthetic workload consists of the models and datasets described in Table 3.1.

We categorized each job in the trace and in Table 3.1 based on their total GPU-time: Small (0–1 GPU-hours), Medium (1–10 GPU-hours), Large (10–100 GPU-hours), and XLarge (100–1000 GPU-hours). For each job in the trace, we picked a training job from Table 3.1 that is in the same category.

Manually-tuned jobs for baseline DL schedulers

We manually tuned the number of GPUs and batch sizes for each job in our synthetic workload, as follows. We measured the time per training iteration for each model in Table 3.1 using a range of GPU allocations and batch sizes, and fully trained each model using a range of different batch sizes (see §3.5.3 for details). We considered a number of GPUs *valid* if using the optimal batch size for that number of GPUs achieves 50% – 80% of the ideal (i.e., perfectly linear) scalability versus using the optimal batch size on a single GPU. For each job submitted from our synthetic workload, we selected its number of GPUs and batch size randomly from its set of valid configurations.

Our job configurations assume that the users are highly rational and knowledgeable about the scalability of the models they are training. Less than 50% of the ideal scalability would lead to under-utilization of resources, and more than 80% of the ideal scalability means the job can still utilize more GPUs efficiently. We emphasize that this assumption of uniformly sophisticated users is unrealistically biased in favor of the baseline schedulers and only serves for comparing Pollux with the ideal performance of baseline systems.

Task:	Image Classification	**Task:**	Object Detection
Dataset:	ImageNet [53]	**Dataset:**	PASCAL-VOC [63]
Model:	ResNet-50 [87]	**Model:**	YOLOv3 [184]
Optimizer:	Momentum SGD	**Optimizer:**	Momentum SGD
LR Scaler:	AdaScale	**LR Scaler:**	AdaScale
M_0:	200 (Images)	**M_0:**	8 (Images)
Validation:	75% Top-1 Accuracy	**Validation:**	84% mAP Score
Category:	XLarge	**Category:**	Large
Frac. Jobs:	2%	**Frac. Jobs:**	6%
Task:	Speech Recognition	**Task:**	Question Answering
Dataset:	CMU-ARCTIC [116]	**Dataset:**	SQuAD [180]
Model:	DeepSpeech2 [20]	**Model:**	BERT (finetune) [55]
Optimizer:	Momentum SGD	**Optimizer:**	AdamW
LR Scaler:	AdaScale	**LR Scaler:**	Square-Root
M_0:	20 (Sequences)	**M_0:**	12 (Sequences)
Validation:	25% Word Error	**Validation:**	88% F1 Score
Category:	Medium	**Category:**	Medium
Frac. Jobs:	10%	**Frac. Jobs:**	10%
Task:	Image Classification	**Task:**	Recommendation
Dataset:	Cifar10 [119]	**Dataset:**	MovieLens [85]
Model:	ResNet18 [87]	**Model:**	NeuMF [88]
Optimizer:	Momentum SGD	**Optimizer:**	Adam
LR Scaler:	AdaScale	**LR Scaler:**	Square-Root
M_0:	128 (Images)	**M_0:**	256 (Rating Pairs)
Validation:	94% Top-1 Accuracy	**Validation:**	69% Hit Rate
Category:	Small	**Category:**	Small
Frac. Jobs:	36%	**Frac. Jobs:**	36%

Table 3.1: Models and datasets used in our evaluation workload. Each training task achieves the provided validation metrics. The fraction of jobs from each category are chosen according to the public Microsoft cluster traces.

Comparison of DL schedulers

We compare Pollux to two recent deep learning schedulers, Tiresias [78] and Optimus [170], as described in §3.2.3. Whereas Pollux dynamically co-adapts the number of GPUs and batch sizes of DL training jobs, Optimus only adapts the number of GPUs, and Tiresias adapts neither.

For each job, Tiresias uses the number of GPUs and batch size specified in our synthetic workload. Optimus+Oracle uses the batch size specified, but determines the number of GPUs dynamically. Each job uses gradient accumulation if they are allocated too few GPUs to support the specified batch size. To establish a fair baseline for comparison, for all three schedulers, we scale the learning rate using AdaScale for SGD, and the square-root scaling rule for Adam and AdamW.

Pollux We configured PolluxSched using a scheduling interval of 60 seconds, and computed `REALLOC_FACTOR`(δ) using $\delta = 30$s. PolluxAgent reports its most up-to-date system throughput parameters and gradient statistics every 30s. Unless otherwise specified, the default fairness knob value of $p = -1$ is used.

Tiresias We configured Tiresias as described in the testbed experiments of Gu et al. [78], with two priority queues and the `PromoteKnob` disabled. We manually tuned the queue threshold to perform well for our synthetic workload. Whenever possible, we placed jobs onto as few different nodes as possible to promote worker locality.

Optimus+Oracle Optimus leverages a throughput prediction model that is specific to jobs using the parameter server architecture. To account for differences due to the performance model, our implementation of Optimus uses our own throughput model as described in §3.3.2. Furthermore, Optimus predicts the number of training iterations until convergence by fitting a simple function to the model's convergence curve. Since this method does not work consistently for all models in our synthetic workload, we run each job ahead of time and provide Optimus with the exact number of iterations until completion. We call this version of Optimus *Optimus+Oracle*.

3.5.2 Testbed Macrobenchmark Experiments

Table 3.2 summarizes the results of our testbed experiments for seven configurations: Pollux compared with, first, baseline schedulers using well-tuned job configurations; second, baseline schedulers using more realistic job configurations; third, Pollux using two alternate values for its fairness knob.

Comparisons using well-tuned job configurations

Even when Optimus+Oracle and Tiresias are given well-tuned job configurations as described in §3.5.1, they are still significantly behind Pollux. In this setting, Pollux (with $p = -1$) achieved 50% and 37% shorter average JCT, 27% and 27% shorter tail (99th percentile) JCT, and 20% and 33% shorter makespan, in comparison to Optimus+Oracle+TunedJobs and Tiresias+TunedJobs,

Policy	Job Completion Time		Makespan
	Average	99%tile	
Pollux ($p = -1$)	**0.76h**	11h	**16h**
Optimus+Oracle+TunedJobs	1.5h	15h	20h
Tiresias+TunedJobs	1.2h	15h	24h
Optimus+Oracle	2.7h	22h	28h
Tiresias	2.8h	25h	31h
Pollux ($p = +1$)	0.83h	**10h**	**16h**
Pollux ($p = -10$)	0.84h	12h	18h

Table 3.2: Summary of testbed experiments.

respectively. As we previously noted, this setting highly favors the baseline schedulers, essentially mimicking users who possess expert knowledge about system throughput, statistical efficiency, and how their values change with respect to resource allocations and batch sizes.

One key source of improvement for Pollux is its ability to trade-off between high-throughput low-efficiency and low-throughput high-efficiency modes during training. Fig. 3.10 shows the total number of allocated GPUs and average $\texttt{EFFICIENCY}_t$ during the execution of our synthetic workload. During periods of low cluster contention, Pollux can allocate more GPUs (indicated by **(A)**) and use larger batch sizes to boost training throughput, even at the cost of lower statistical efficiency, because doing so results in an overall higher goodput. On the other hand, during periods of high cluster contention, Pollux may instead use smaller batch sizes to increase statistical efficiency (indicated by **(B)**).

Comparisons using realistic job configurations

Without assistance from a system like Pollux, users are likely to try various numbers of GPUs and batch sizes, before finding a configuration that is efficient. Other users may not invest time into configuring their jobs well in the first place.

To set a more realistically configured baseline, we ran Optimus+Oracle and Tiresias on a version of our synthetic workload with the number of GPUs exactly as specified in the Microsoft cluster trace. The batch size was chosen to be the baseline batch size M_0 times the number of GPUs, which is how we expect most users to initially configure their distributed training jobs. We find that these jobs typically use fewer GPUs and smaller batch sizes than their well-configured counterparts.

Using this workload, we find that Pollux has 72% and 73% shorter average JCT, 50% and 56% shorter tail JCT, and 43% and 48% shorter makespan, in comparison to Optimus+Oracle and Tiresias, respectively. Even though Optimus+Oracle can dynamically increase the GPU allocation of each job, it still only slightly outperforms Tiresias because it does not also increase the batch size to better utilize those additional GPUs.

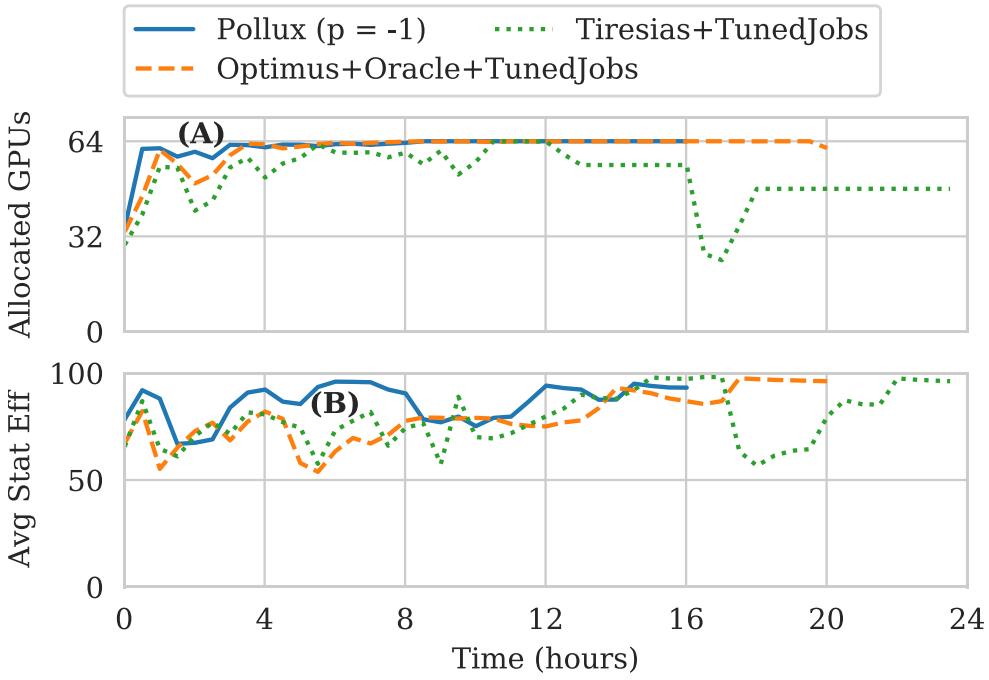


Figure 3.10: Comparison between Pollux ($p = -1$), Optimus, and Tiresias while executing our synthetic workload (with tuned jobs). TOP: average cluster-wide allocated GPUs over time. BOTTOM: average cluster-wide statistical efficiency over time. Tiresias+TunedJobs dips between hours 16 and 20 due to a 24-GPU job blocking a 48-GPU job from running.

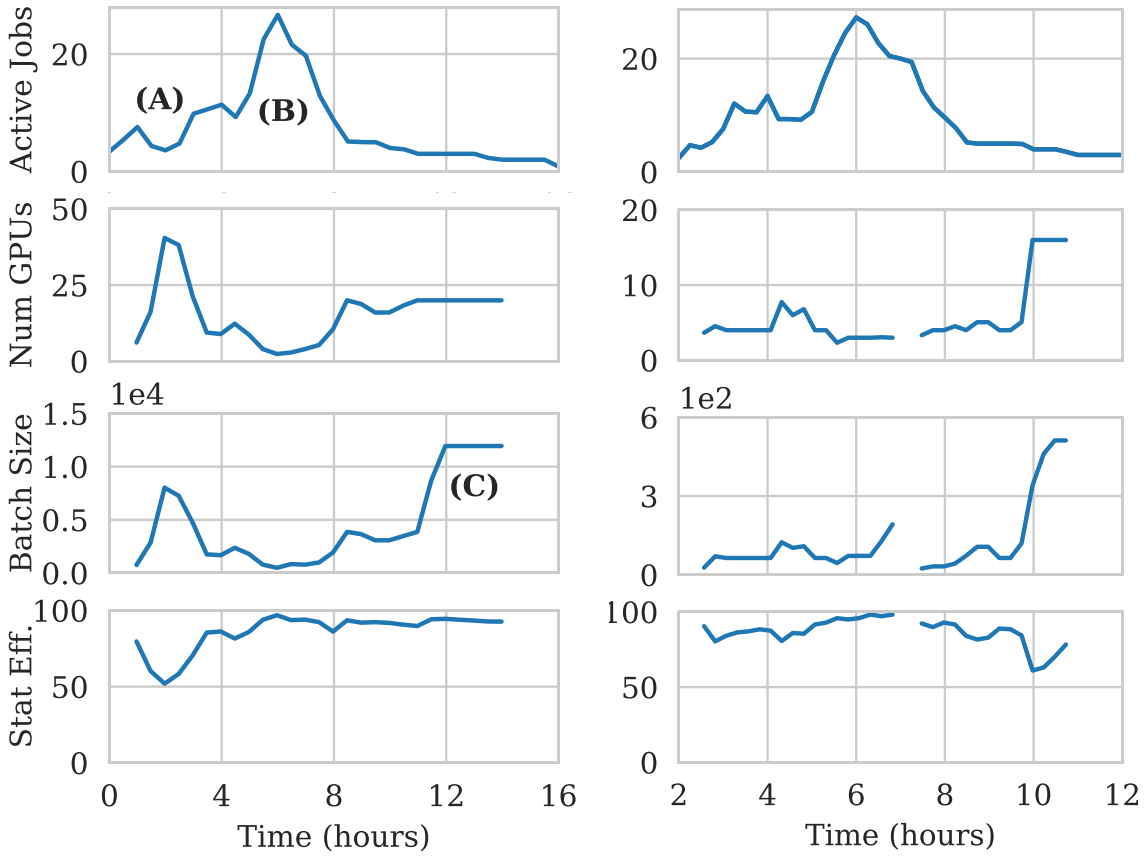


Figure 3.11: Co-adaptation over time of one ImageNet job (LEFT) and two YOLOv3 jobs (RIGHT) using Pollux ($p = -1$). ROW 1: number of jobs actively sharing the cluster. ROW 2: number of GPUs allocated to the job. ROW 3: batch size (images) used. ROW 4: statistical efficiency (%).

A closer look at co-adapted job configurations

Fig. 3.11 (LEFT) shows the configurations chosen by Pollux for one ImageNet training job as the synthetic workload progresses. **(A)** during the initial period of low cluster contention, more GPUs are allocated to ImageNet, causing a larger batch size to be used and lowering statistical efficiency. **(B)** during the subsequent period of high cluster contention, fewer GPUs are allocated to ImageNet, causing a smaller batch size to be used and raising statistical efficiency. **(C)** when the cluster contention comes back down, ImageNet continues to be allocated more GPUs and uses a larger batch size. However, we note that the batch size per GPU is much higher than in the first low-contention period, since the job is now in its final, high-statistical-efficiency phase of training. We see similar trade-offs being made over time for two YOLOv3 jobs (RIGHT).

Effect of the fairness knob

We ran Pollux using three values of the fairness knob, $p = 1, -1, -10$. Compared with no fairness ($p = 1$), introducing a moderate degree of fairness ($p = -1$) improved the average job

completion time (JCT) but degraded the tail JCT. This is because[6], in our synthetic workload, the tail JCT comprises of long but scalable jobs (i.e. ImageNet), which take a large number of GPUs away from other jobs in the absence of fairness ($p = 1$). However, further increasing fairness ($p = -10$) degraded performance in average JCT, tail JCT, and makespan. In §3.5.3, we present a more detailed analysis of the impact of p on scheduling fairness.

System overheads

During each 60s scheduling interval, PolluxSched spent an average of 1 second on 1 vCPU computing the cluster allocations by optimizing the $\texttt{FITNESS}_p$ function. On average, each job was re-allocated resources once every 7 minutes, resulting in an average 8% run-time overhead due to checkpoint-restarts. Each PolluxAgent fits its throughput model parameters on its latest observed metrics every 30 seconds, taking an average of 0.2 seconds each time. Finding the optimal per-GPU batch size and gradient accumulation steps by optimizing $\texttt{GOODPUT}_t$ takes an average of 0.4 milliseconds.

3.5.3 Simulator Experiments

We built a discrete-time cluster simulator in order to evaluate a broader set of workloads and settings. Our simulator is constructed by measuring the performance and gradient statistics of each model in Table 3.1, under many different resource and batch size configurations, and re-playing them for each simulated job. This way, we are able to simulate both the system throughput and statistical efficiency of the jobs in our workload.

Unless stated otherwise, each experiment in this section is repeated on 8 different workload traces generated using the same duration, number of jobs, and job size distributions as in §3.5.2, and we report the average results across all 8 traces.

Simulator construction

For each job in Table 3.1, we measured the time per training iteration for 146 different GPU allocations+placements in our testbed cluster of 16 nodes and 64 total GPUs. For each allocation, we measured a range of batch sizes up to the GPU memory limit. To simulate the throughput for a job, we queried a multi-dimensional linear interpolation on the configurations we measured. For each model, we also measured the (pre-conditioned) gradient noise scale during training using a range of batch sizes, and across every epoch. To simulate the statistical efficiency for a job using a certain batch size, we linearly interpolated its value of the PGNS between the two nearest batch sizes we measured.

Simulator fidelity

The data we collected about each job enables our simulator to reproduce several system effects, including the performance impact of different GPU placements. We also simulate the overhead of

[6]We note that $p = -1$ (harmonic mean over speedups) may be more suitable than $p = 1$ (arithmetic mean) when optimizing for the average JCT.

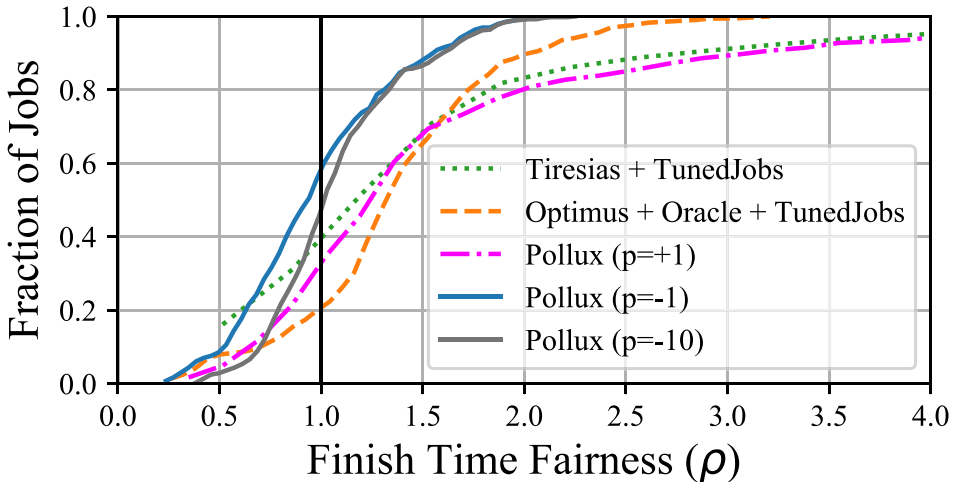


Figure 3.12: CDF of Finish Time Fairness (ρ).

checkpoint-restarts by injecting a 30-second delay for each job that has its resources re-allocated. Unless stated otherwise, we do not simulate any network interference between different jobs. We study the effects of interference in more detail in §3.5.3.

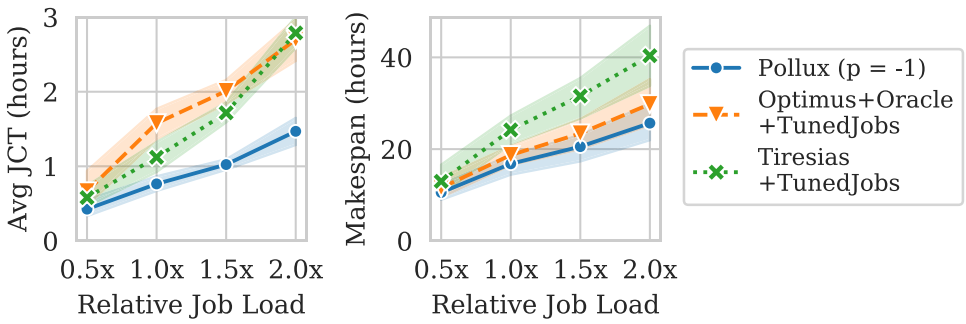
Compared with our testbed experiments in §3.5.2, we find that our simulator obtains similar factors of improvement, showing that Pollux reduces the average JCT by 48% and 32% over Optimus+Oracle+TunedJobs and Tiresias+TunedJobs.

Scheduling Fairness

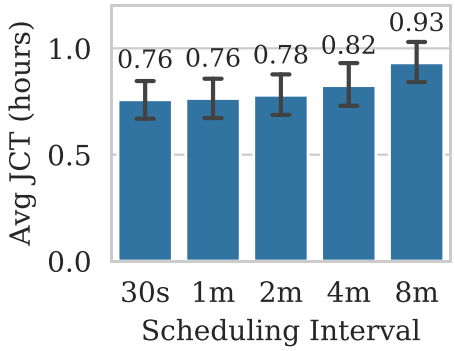
We evaluate the scheduling fairness of Pollux using *finish-time fairness*[146] (denoted by ρ), which is defined to be the ratio of a job's JCT running on shared resources to that of the job running in an isolated and equally-partitioned cluster. Under this metric, jobs with $\rho < 1$ have been treated better-than-fair by the cluster scheduler, while jobs with $\rho > 1$ have been treated worse-than-fair.

In Fig. 3.12, we compare the finish-time fairness of Pollux with Optimus+Oracle+TunedJobs and Tiresias+TunedJobs. Pollux with $p = 1$ results in poor fairness, similar to Tiresias+TunedJobs, which is apparent as a long tail of jobs with $\rho > 4$. Optimus+Oracle+TunedJobs obtains better fairness due to its allocation algorithm which attempts to equalize the JCT improvement for each job. Pollux with $p = -1$ provides the best fairness, with 99% of jobs achieving $\rho < 2$, and does so while still providing significant performance increases (Table 3.2). For $p = -10$, we observe slightly worse fairness overall, caused by PolluxSched incurring a larger number of re-allocations due to ignoring the cost in favor of equalizing speedups at all times.

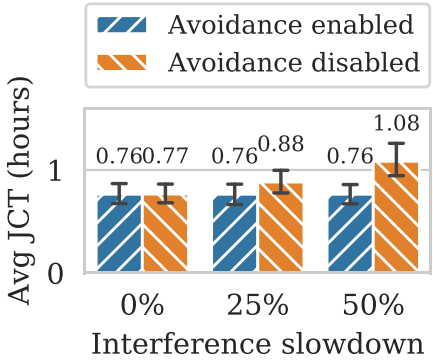
To provide context, we note that the curves for Tiresias and Optimus are consistent with those reported (for different workloads) by Mahajan et al. [146]. Although their Themis system is not available for direct comparison, the ρ range for Pollux with $p = -1$ is similar to the range reported for Themis. The max-ρ improvements (1.5× and 5.4×) over Tiresias and Optimus are also similar.



(a) Varying the workload intensity.



(b) Varying scheduling interval.



(c) Varying job interference.

Figure 3.13: Effects of various parameters on Pollux, error bars and bands represent 95% confidence intervals.

Other Effects on Scheduling

Sensitivity to job load. We compare the performance of Pollux, Optimus+Oracle+TunedJobs, and Tiresias+TunedJobs for increasing workload intensity in terms of rate of job submissions. Fig. 3.13a shows the results. As expected, all three scheduling policies suffer longer average JCT and makespan as the load is increased. Across all job loads, Pollux maintains similar relative improvements over the baseline schedulers.

Impact of prior-driven exploration. Pollux explores GPU allocations for each DL job from scratch during training (Sec. 3.4.1). We evaluated the potential improvement from more efficient exploration by seeding each job's throughput models using historical data collected offline. We observed minor (2–5%) reduction in JCT for short jobs like CIFAR10, but no significant change for longer running jobs, indicating low overhead from Pollux's prior-driven exploration.

Impact of scheduling interval. We ran Pollux using a range of values for its scheduling interval, as shown in Fig. 3.13b. We find that Pollux performs similarly well in terms of average JCT for intervals up to 2 minutes, while longer intervals result in performance degradation. Since newly-submitted jobs can only start during the next scheduling interval, we would expect an increase in the average queuing time due to longer scheduling intervals. However, we find that

queuing contributed to roughly half of the performance degradation observed, indicating that Pollux still benefits from a relatively frequent adjustment of resource allocations.

Impact of interference avoidance. To evaluate the impact of PolluxSched's interference avoidance constraint, we artificially inject various degrees of slowdown for distributed jobs sharing the same node. Fig. 3.13c shows the results. With interference avoidance enabled, the average JCT is unaffected by even severe slowdowns, because network contention is completely mitigated. However, without interference avoidance, the average JCT is $1.4\times$ longer when the interference slowdown is 50%. On the other hand, in the ideal scenario when there is zero slowdown due to interference, PolluxSched performs similarly whether or not interference avoidance is enabled. This indicates that PolluxSched is still able to find efficient cluster allocations while obeying the interference avoidance constraint.

3.5.4 Cluster Auto-Scaling in the Cloud

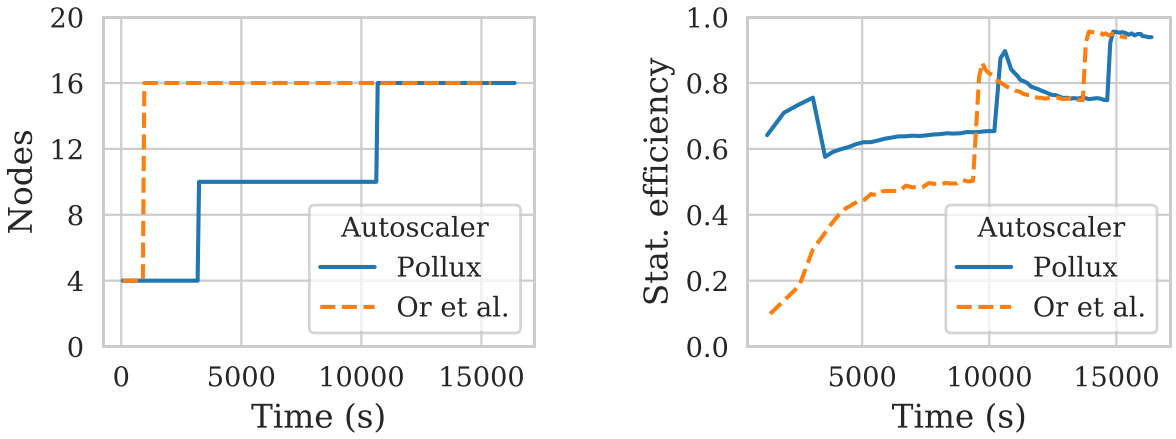
In cloud environments, computing resources can be obtained and released as required, and users pay for the duration they hold onto those resources. Goodput-driven scheduling presents a unique opportunity: when a DL model's statistical efficiency increases during training, it may be more cost-effective to provision more cloud resources and use larger batch sizes during the later epochs of a large training job, rather than earlier on. We present some preliminary evidence using our cluster simulator, and note that a full design of an auto-scaling system based on goodput may be the subject of future work.

Auto-scaling ImageNet training. We implemented a simple auto-scaling policy using Pollux's goodput function. During training, we scaled up the number of nodes whenever

$$\max_{m,s} \texttt{GOODPUT}_t(a, m, s)/\texttt{SUM}(a) > U \cdot \max_{m,s} \texttt{GOODPUT}_t(1, m, s),$$

i.e. the goodput exceeds some fraction U of the predicted ideal goodput assuming perfect scalability. We set $U = 2/3$, and increased to a number of nodes such that the predicted goodput is approximately $L = 1/2$ of the predicted ideal goodput.

Fig. 3.14 compares our Pollux-based auto-scaler with the auto-scaler proposed by Or *et al.* [164], which allows the batch size to be increased during training, but models job performance using the system throughput rather than the goodput. Since the system throughput does not change with training progress, throughput-based autoscaling (Or *et al.*) quickly scales out to more nodes and a larger batch size (Fig. 3.14a), which remains constant thereafter. On the other hand, Pollux starts with a small number of nodes, and gradually increases the number of nodes as the effectiveness of larger batch sizes improves over time. Fig. 3.14b shows that Pollux maintains a high statistical efficiency throughout training. Overall, compared to Or *et al.*'s throughput-based auto-scaling, Pollux trains ImageNet with 25% cheaper cost, with only a 6% longer completion time.



(a) Number of nodes over time.

(b) Statistical efficiency over time.

Figure 3.14: Goodput-based auto-scaling (Pollux) vs throughput-based auto-scaling (Or et al.) for ImageNet training.

3.5.5 Hyper-parameter Optimization

Hyper-parameter optimization (HPO) is an important DL workload. In HPO, the user defines a *search space* over relevant model hyper-parameters. A HPO algorithm (aka a trial scheduler) submits many training jobs (trials) to evaluate the effectiveness of particular hyper-parameters, in terms of objectives such as model accuracy or energy efficiency.

Different HPO algorithm types manage trials differently. For example, Bayesian optimization algorithms [115, 199] may submit a few training jobs at a time, and determine future trials based on the fully-trained results of previous trials. Bandit-based algorithms [130] may launch a large number of trials at once and early-stop ones that appear unpromising.

A full evaluation on how Pollux affects different HPO algorithm types is future work. However, in this section, we present preliminary evidence that Pollux can accelerate certain HPO workloads when used as the underlying cluster resource scheduler. Table 3.3 shows results from tuning a ResNet18 model trained on the CIFAR10 dataset, using a popular Bayesian optimization-based HPO algorithm known as the Tree-structured Parzen Estimator (TPE) [27]. The search space covers the learning rate and annealing, momentum, weight decay, and network width hyper-parameters. We configured TPE so that 4 trials run concurrently with each other, and 100 trials are run in total. The testbed consists of two NVIDIA DGX A100 nodes, each with 8 A100 GPUs. The baseline scheduler assigns a static allocation of 4 GPUs (all on the same node) to each trial and uses a fixed per-GPU batch size for every trial. As expected, similar accuracy values are achieved, but Pollux completes HPO 30% faster due to adaptive (re-)allocation of resources as trials progress and adaptive batch sizes.

3.6 Additional Related Work

Prior DL schedulers are discussed in §3.2.3.

Policy	Accuracy (Top 5 trials)	Avg JCT	Makespan
Pollux	95.4 ± 0.2	25min	10h
Baseline	95.5 ± 0.3	34min	14h

Table 3.3: Summary of HPO experiments. We run each experiment twice, and report the average between both experiments. Due to inherent randomness in the trials chosen by the TPE algorithm, we report the average accuracy of the top 5 trials.

Adaptive batch size training

Recent work on DL training algorithms have explored dynamically adapting batch sizes for better efficiency and parallelization. AdaBatch [54] increases the batch size at pre-determined iterations during training, while linearly scaling the learning rate. Smith et al. [198] suggest that instead of decaying the learning rate during training, the batch size should be increased instead. CABS [24] adaptively tunes the batch size and learning rate during training using similar gradient statistics as Pollux.

These works have a common assumption that extra computing resources are available to parallelize larger batch sizes whenever desired, which is rarely true inside shared-resource environments. Pollux complements existing adaptive batch size strategies by adapting the batch size and learning rate in conjunction with the amount of resources currently available. Alternatively, anytime minibatch [64] adapts the batch size to mitigate stragglers in distributed training.

KungFu [147] supports adaptive training algorithms, including adaptive batch sizes, by allowing applications to define custom adaptation policies and enabling efficient adaptation and monitoring during training. Although KungFu is directed at single-job training and Pollux at cluster scheduling, we believe KungFu offers useful tools which can be used to implement the adaptive policies used by the PolluxAgent.

Hyper-parameter tuning

A large body of work focuses on tuning the hyper-parameters for ML and DL models [27, 65, 99, 108, 159], which typically involves many training jobs [3, 74] as discussed earlier. Although batch size and learning rate are within the space of hyper-parameters often optimized by these systems, Pollux's goal is fundamentally different. Whereas HPO algorithms search for the highest model quality, Pollux adapts the batch size and learning rate for the most efficient execution for each job, while not degrading model quality.

Chapter 4

Litz: Enabling General Elasticity for High-Performance Machine Learning

In Chapter 3, we showed how co-adaptation can improve elastic DL training in shared-resource clusters by automatically tuning both system-level configurations and application-side parameters. In this chapter, we take a step back and consider elasticity for ML training in general. In particular, the programming abstractions and system implementations that enable ML applications to scale in and out when the available compute resources change.

New algorithmic and systems techniques leverage unique properties of ML training to improve their distributed performance by orders of magnitude. However, applications built using these techniques tend to be static, unable to elastically adapt to changing resource availability. Existing distributed frameworks are either inelastic, or offer programming models which are incompatible with the techniques employed by high-performance ML applications.

We present Litz, an elastic framework supporting general distributed ML applications. We categorize the wide variety of performance-enhancing techniques employed by these applications into three broad classes—stateful workers, model scheduling, and relaxed consistency—which are collectively supported by Litz's programming model. Our implementation of Litz's execution system transparently enables elasticity and low-overhead execution.

We implement several popular ML applications using Litz, and show that they can scale in and out quickly to adapt to changing resource availability. We show that Litz enables elasticity without compromising performance, achieving competitive performance with state-of-the-art non-elastic ML frameworks. In §4.7, we describe and evaluate a highly-optimized implementation of the HDBSCAN clustering algorithm using Litz, showing its ability to support different algorithms in an elastic manner. In §4.8, we leverage Litz to perform automatic out-of-core execution in memory-limited scenarios.

Unlike Pollux, Litz is not a co-adaptive system for ML. Rather, Litz merely provides a system-level abstraction and mechanism for an ML training application to dynamically scale in and out. Litz provides a framework for co-adaptive techniques to be built on top of it, as we shall demonstrate in Chapter 5.

The contents of this chapter were previously published in [174].

4.1 Introduction

Recent advancements in algorithmic and systems techniques for distributed ML applications have improved their performance by an order of magnitude or more. New algorithms such as AdaptiveRevision [152], NOMAD [183], and LightLDA [224] can better scale in distributed environments, possessing favorable properties such as staleness tolerance [91, 152], lock-free execution [183, 225], and structure-aware parallelization [70, 224]. Systems and frameworks such as GraphLab [142], Petuum [217], Adam [42], and various parameter servers [91, 132] are able to support and exploit these properties to achieve even higher performance, using techniques such as bounded-staleness consistency models [47], structure-aware scheduling [113], bandwidth management/re-prioritization [211], and network message compression [42, 216].

Although significant work is being done to push the boundaries of distributed ML in terms of performance and scalability, there has not been as much focus on elasticity, thus limiting the resource adaptability of ML applications in real-world computing environments.

General-purpose distributed frameworks such as Hadoop [11] and Spark [226] are well integrated with cloud and data-center environments, and are extensively used for running large-scale data processing jobs. They are designed to support a wide spectrum of conventional tasks—including SQL queries, graph computations, and sorting and counting—which are typically transaction-oriented and rely on deterministic execution. However, their programming models are incompatible with the algorithmic and systems techniques employed by distributed ML applications, abstracting away necessary details such as input data partitioning, computation scheduling, and consistency of shared memory access. As a result, the performance of ML applications built using these frameworks fall short of standalone implementations by two orders of magnitude or more [212].

Consequently, distributed ML applications are often implemented without support from elastic frameworks, resulting in jobs that hold a rigid one-time allocation of cluster resources from start to finish [42, 113, 211, 225]. The lack of an elastic framework, along with a suitable programming model which can support the various distributed ML techniques, is a key roadblock for implementing elastic ML applications.

Although the algorithmic and systems techniques employed by these standalone applications are diverse, they typically arise from only a few fundamental properties of ML that can be collectively supported by an elastic ML framework. This observation exposes an opportunity to design a framework that is able to support a large variety of distributed ML techniques by satisfying a smaller set of more general requirements. We summarize these properties of ML and how they guide the design of an elastic framework below, and further elaborate on them in Sec. 4.2.

First, ML computations exhibit a wide variety of memory access patterns. Some mutable state may be accessed when processing each and every entry of a dataset, while other state may only be accessed when processing a single data entry. To improve locality of access, ML applications explicitly co-locate mutable model parameters with immutable dataset entries [224]. Each worker machine in the computation may contain a non-trivial amount of mutable state, which needs to be properly managed under an elastic setting.

Second, ML models contain a wide variety of dependency structures. Some sets of model parameters may safely be updated in parallel, while other sets of parameters must be updated in sequence. Guided by these dependency structures, ML applications carefully schedule their model

updates by coordinating tasks across physical worker machines [70]. An elastic ML framework should abstract the physical cluster away from applications while still providing enough flexibility to support this type of task scheduling.

Furthermore, ML algorithms are often iterative-convergent and robust against small errors. Inaccuracies occurring in their execution are automatically corrected during later stages of the algorithm. Distributed ML applications have been able to attain higher performance at no cost to correctness by giving up traditionally desirable properties such as deterministic execution and consistency of memory access [91]. Framework mechanisms for elasticity should not rely on a programming model that restricts this way of exploiting the error-tolerance of ML algorithms.

Thus, to efficiently support ML applications, an elastic ML framework should support **stateful workers**, **model scheduling**, and **relaxed consistency**. It should provide an expressive programming model allowing the application to define a custom scheduling strategy and to specify how the consistency of memory accesses can be relaxed under it. Then, it should correctly execute this strategy within the specified consistency requirements, while gracefully persisting and migrating application state regardless of its placement with respect to input data.

Motivated by the needs and opportunities for elasticity of ML applications, we designed and implemented *Litz*[1], an elastic framework for distributed ML that provides a programming model supporting stateful workers, model scheduling and relaxed consistency.

Litz enables low-overhead elasticity for high-performance ML applications. When physical machines are added to or removed from an active job, state and computation are automatically rebalanced across the new set of available machines without active participation by the application. Litz's programming model can express key distributed ML techniques such as stateful workers, model scheduling and relaxed consistency, allowing high-performance ML applications to be implemented. Furthermore, a cluster job scheduler can leverage Litz's elasticity to achieve faster job completion under priority scheduling, and optimize resource allocation by exploiting inherent resource variability of ML algorithms.

Our main contributions are:

1. **Event-driven Programming Model for ML:** Litz exposes an event-driven programming model that cleanly separates applications from the physical cluster they execute on, enabling stateful workers and allowing the framework to transparently manage application state and computation during elastic events. Computation is decomposed into *micro-tasks* which have shared access to a distributed parameter server.

2. **Task-driven Consistency Model for ML:** Micro-tasks can be scheduled according to dependencies between them, allowing the application to perform model scheduling. Access to the parameter server is controlled by a consistency model in which a micro-task always observes all updates made by its dependencies, while having intentionally weak guarantees between independent micro-tasks.

3. **Optimized Elastic Execution System:** Litz's execution system transparently re-balances workload during scaling events without active participation from the application. It exploits Litz's programming and consistency models to implement optimizations that reduce

[1]Meant to evoke the strings of a harp, sounding out as many or as few. Litz is short for "Wurlitzer", a well-known harp maker.

system overhead, allowing applications using Litz to be as efficient as those using non-elastic execution systems.

4.2 Background

Stateful Workers

Even though the model term w appears in the calculations of each partial update, not all of it is necessarily used. In particular, there may be parts of the model which are only used when processing a single partition $\mathcal{D}_i$ of the input data. A large class of examples includes non-parametric models, whose model structures are not fixed but instead depends on the input data itself, typically resulting in model parameters being associated with each entry in the input data. In such applications, it is preferable to co-locate parts of the model on worker nodes with a particular partition of input data so they can be accessed and updated locally rather than across a network. This optimization is especially essential when the input data is large and accesses to such associated model parameters far outnumber accesses to shared model parameters. It also means that workers are *stateful*, and an elastic ML system that supports this optimization needs to preserve worker state during elastic resource re-allocation.

4.2.1 Error Tolerance & Relaxed Consistency

ML algorithms have several well-established and unique properties, including *error-tolerance*: even if a perturbation or noise ϵ is added to the model parameters in every iteration, i.e. $w^{(t)} = w^{(t-1)} + \Delta(w^{(t-1)}; \mathcal{D}) + \epsilon$, the ML algorithm will still converge correctly provided that ϵ is limited or bounded.

Bounded Staleness Consistency

An important application of error tolerance is bounded staleness consistency models [23, 47, 91], which allow stale model parameters to be used in update computations, i.e. $w^{(t)} = w^{(t-1)} + \Delta(w^{(t-s)}; \mathcal{D})$, where $1 \leq s \leq k$ for small values of k. ML algorithms that use such consistency models are able to (1) execute in a partially asynchronous manner without sacrificing correctness, thus mitigating the effect of stragglers or slow workers [43, 81]; and (2) reduce the effect of network bottlenecks caused by synchronization by allowing cached parameter values to be used. Stale-Synchronous Parallel (SSP) [91] is such a consistency model, under which a set of distributed workers may read cached values from a shared parameter server as long as their staleness do not exceed a fixed limit.

Staleness-aware ML Algorithms

Beyond simply applying bounded staleness consistency to existing algorithms, the ML community has developed new staleness-aware algorithms [14, 17, 92, 131, 152, 224, 234] which modify each update $\Delta()$ according to the staleness s that it experiences. The modifications usually take the form of a scaling factor $\Delta() \leftarrow c\Delta()$, which are computationally light-weight and do not

create new bottlenecks. In the presence of staleness, these algorithms converge up to an order of magnitude faster than their non-staleness-aware counterparts.

4.2.2 Dependencies and Model Scheduling

Another key property of ML algorithms is the presence of implicit *dependency structures*: supposing w_1 and w_2 are different elements of w, then updating w_1 before w_2 does not necessarily yield the same result as updating w_2 before w_1; whether this happens or not depends on the algebraic form of the objective (or loss) function $\mathcal{L}()$ and $\Delta()$. As a consequence, the convergence rate and thus the running time of ML algorithms can be greatly improved through careful scheduling of parallel model parameter updates.

Dependency-aware ML Algorithms

Like the many existing staleness-aware algorithms that exploit error tolerance, there is a rich set of algorithms that use dependency structures in their models to perform better scheduling of updates [51, 70, 123, 142, 189, 208, 224]. A typical example is to partition the model into subsets, where the parameters inside a subset must be updated sequentially, but multiple subsets can be updated in parallel. Two parameters w_1 and w_2 are placed into the same subset if the strength of their dependency exceeds a threshold $\mathrm{dep}(w_1, w_2) > \epsilon$. As with staleness-aware algorithms, dependency-aware algorithms converge up to an order of magnitude faster than their non-dependency-aware counterparts.

4.3 Litz Programming Model and API

The main goal and challenge of designing Litz's programming model is striking a balance between being expressive enough to support the wide variety of proven techniques in distributed ML, while exposing enough structure in the application that the underlying execution system can take control under elastic conditions. Guided by the insights presented in Sec. 4.2, we describe how Litz's programming model naturally arises from the properties of ML applications, and how it enables an efficient and elastic run-time implementation. For reference, a detailed summary of Litz's API can be found in Table 4.1.

Input Data Over-Partitioning Across Executors

Eq. 2.3 shows that the input data and update calculations of ML applications can be partitioned and distributed across a number of workers, but it does not specify any particular partitioning scheme, nor does it require the number of partitions to be equal to the number of physical machines. Instead of directly assigning input data, Litz first distributes it across a set of logical *executors*, which are in turn mapped to physical machines. Elasticity is enabled by allocating more executors than physical machines and migrating excess executors to other machines as they become available. This separation also lets Litz support stateful workers by allowing executor state to be defined and mutated by the application while being treated as a black box by the run-time system.

Method Name	Part Of	Defined By	Description
`DispatchInitialTasks()`	Driver	Application	Invoked by the framework upon start-up to dispatch the first set of micro-tasks.
`HandleTaskCompletion(result)`	Driver	Application	Invoked by the framework when a micro-task completes so that the driver can dispatch a new set of micro-tasks.
`DispatchTask(executor,args)`	Driver	Framework	Invoked by the application to dispatch a micro-task to the specified executor.
`RunTask(args)`	Executor	Application	Invoked by the framework to perform a micro-task on the executor.
`SignalTaskCompletion(result)`	Executor	Framework	Invoked by the application to indicate the completion of a micro-task.
`PSGet(key)`	Executor	Framework	Returns a specified value in the parameter server.
`PSUpdate(key,update)`	Executor	Framework	Applies an incremental update to a specified value in the parameter server.

Table 4.1: The API for Litz. An application should define `DispatchInitialTasks` and `HandleTaskCompletion` on the driver, as well as `RunTask` on the executor.

Micro-Tasks and Parameter Server

Update calculations are decomposed into short-lived (typically shorter than 1 second) units of computation called *micro-tasks*, each of which calculates a partial update using the input data on a single executor. At the end of each micro-task, control is yielded back to the run-time system, exposing frequent opportunities for executors to be migrated. During its execution, a micro-task is granted read/update access to a global parameter server via a key-value interface (`PSGet/PSUpdate` in Table 4.1) and applies partial updates to model parameters by modifying application state in the executor and/or updating globally-shared values in the parameter server.

Model Scheduling and Relaxed Consistency

Litz enables both model scheduling and relaxed consistency using application-defined dependencies between micro-tasks. If micro-task A is a dependency of micro-task B, then (1) B is executed before A and (2) B observes all updates made by A. This strict ordering and consistency guarantee lets the application perform model scheduling by defining an ordering for when certain updates are calculated and applied. On the other hand, if neither A nor B is a dependency of the other, then they may be executed in any order or in parallel, and may observe none, some, or all of the updates made by the other. This critical piece of non-determinism lets the application exploit relaxed consistency models by allowing the run-time system to cache and use stale values from the parameter server between independent micro-tasks.

Micro-Task Dispatch and Completion

A common way to specify dependencies between tasks is through a directed "dependency" graph in which each vertex corresponds to a micro-task, and an arc from vertex A to vertex B means task A is a dependency of task B. However, due to a potentially large number of micro-tasks, explicitly specifying such a graph up-front may incur significant overhead. Instead, each Litz application defines a *driver* which dynamically dispatches micro-tasks during run-time via the `DispatchTask` method. When a micro-task completes, Litz invokes the `HandleTaskCompletion` method on the driver, which can then dispatch any additional micro-tasks.

Without an explicit dependency graph, Litz needs an alternative way to decide when a micro-task should be able to observe another micro-task's updates. Otherwise, its execution system does not have enough information to know when it is safe for a micro-task to use cached parameter values, thus giving up a significant opportunity for performance optimization. To overcome this issue, Litz uses the sequence of micro-task dispatch and completion events to infer causal relationships between micro-tasks, which can then be used to generate the dependencies needed to implement its cache coherence protocol. According to the following two cases:

1. If micro-task B is dispatched *before* being informed of the completion of micro-task A, then Litz infers that the completion of A *did not cause* the dispatch of B. A *is not* a dependency of B, and B may observe some, all, or none of the updates made by A.

2. If micro-task B is dispatched *after* being informed of the completion of micro-task A, then Litz infers that A *may have caused* the dispatch of B. A *may be* a dependency of B, and B will observe all updates made by A.

This consistency model is similar to Causal Memory [15], in which causally related read/write operations are observed in the same order by all nodes. We discuss how Litz's consistency model and its cache coherence protocol can be implemented efficiently in Sec. 4.4.

4.4 Litz Implementation and Optimizations

Litz is implemented in approximately 6500 lines of C++ code using the ZeroMQ [12] library for low latency communication and Boost's Context [8] library for low overhead context-switching between micro-tasks. The run-time system is comprised of a single *master thread* along with a collection of *worker threads* and *server threads*, as shown in Fig. 4.1. The application's driver exists in the master thread and its executors exist in the worker threads. The key/value pairs comprising the parameter server are distributed across a set of logical *PSshards* stored in the server threads. Additional worker and server threads may join at any time during the computation, and the run-time system can re-distribute its load to make use of them. They may also gracefully leave the computation after signaling to the master thread and allowing their load to be transferred to other threads.

The master thread coordinates the execution of the application. First, it obtains micro-tasks from the driver by initially invoking `DispatchInitialTasks` and then continuously invoking `HandleTaskCompletion`, sending them to worker threads to be executed. Second,

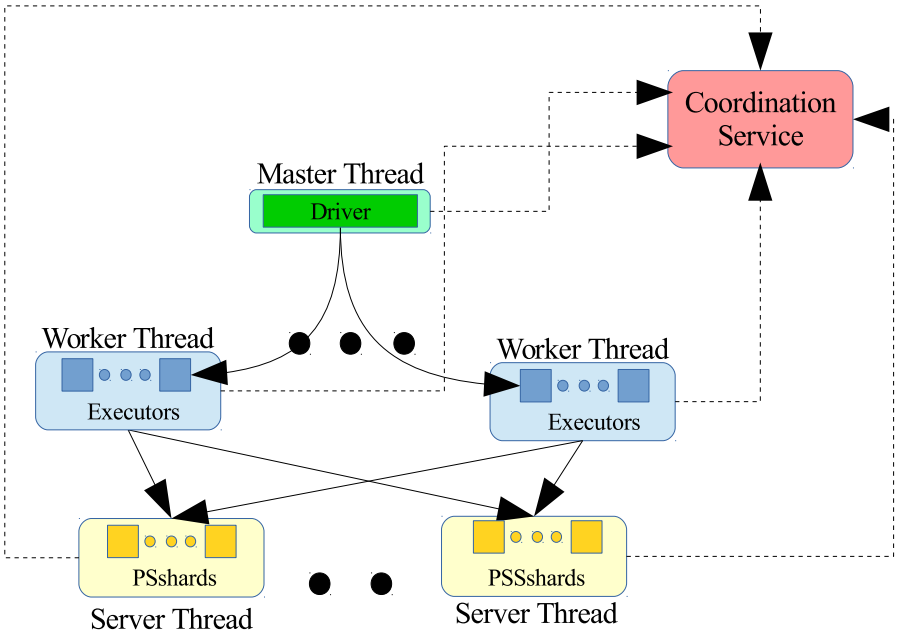


Figure 4.1: High-level architecture of Litz. The driver in the master thread dispatches micro-tasks to be performed by executors on the worker threads. Executors can read and update the global model parameters distributed across PSshards on the server threads.

the master thread maintains the dynamic mappings between executors and worker threads, as well as between PSshards and server threads. When worker or server threads join or leave the computation, it initiates load re-distribution by sending commands to move executors between worker threads or PSshards between server threads. Third, the master thread periodically triggers a consistent checkpoint to be taken of the entire application state, and automatically restores it when a failure is detected. Each thread registers with an external coordination service such as ZooKeeper [98] or etcd [10] in order to determine cluster membership and detect failures. In order to transfer and checkpoint the driver and executors, Litz requires the application to provide serialization and de-serialization code. The programming burden on the developer is low since (1) it does not actively participate in elasticity and checkpointing, but simply invoked by the execution system when needed, and (2) third-party libraries can be used to reduce programming overhead [9].

Worker Thread Elasticity

Each worker thread maintains the state of and runs the micro-tasks for a subset of all executors. After any worker threads join the active computation, executors are moved to them from the existing worker threads (scaling out). Similarly, before any worker threads leave the active computation, executors are moved from them to the remaining worker threads (scaling in).When an executor needs to be moved, the worker thread first finishes any of its ongoing micro-tasks for that executor, buffering any other pending micro-tasks for that executor. The worker thread then sends the executor's state and its queue of buffered micro-tasks over the network to the receiving worker thread.

The transfer of the executor's input data is treated differently in the scale-in and scale-out cases. When scaling in, Litz aims to free the requested resources as quickly as possible. The input data is discarded on the originating worker thread to avoid incurring extra network transfer time, and re-loaded on the target worker thread from shared storage. When scaling out, Litz aims to make use of the new worker thread as quickly as possible. The input data is sent directly from the memory of the originating worker thread to avoid incurring extra disk read time on the target worker thread.

Parameter Server Elasticity

Similar to worker threads and executors, each server thread stores and handles the requests and updates for a subset of all PSshards, which are re-distributed before scaling in and after scaling out. However, since requests and updates are continuously being sent to each PSshard and can originate from any executor, their transfer requires a special care. In particular, a worker thread may send requests or updates to a server thread that no longer contains the target PSshard, which can occur if the PSshard has been moved but the worker thread has not yet been notified.

A naïve approach is to stop all micro-tasks on every executor, then perform the transfer, then notify all worker threads of the change, and finally resume execution. This method guarantees that requests and updates are always sent to server threads that contain the target PSshard, but incurs high overhead due to suspending the entire application. Instead, the server threads perform *request and update forwarding*, and executors are never blocked from sending a parameter

request or update. When a server thread receives a message for a PSshard it no longer contains, it forwards the message to the server thread it last transferred the PSshard to. Forwarding can occur multiple times until the target PSshard is found, the request/update is performed, and the response is sent back to the originating worker thread. This way, execution of micro-tasks can proceed uninterrupted during parameter server scaling events.

Consistent Checkpoint and Recovery

To achieve fault tolerance, Litz periodically saves a checkpoint of the application to persistent storage, consisting of (1) the state of the driver, (2) the buffered micro-tasks for each executor, (3) the state of each executor, and (4) the key-value pairs stored in each PSshard. Input data is not saved, but is re-loaded from shared storage during recovery. When a failure is detected through the external coordination service, Litz triggers an automatic recovery from the latest checkpoint. The saved driver, executors, buffered micro-tasks, and parameter server values are restored, after which normal execution is resumed.

Parameter Cache Synchronization

The consistency model outlined in Sec. 4.3 exposes an opportunity for the run-time system to optimize execution by caching and re-using values from the parameter server instead of retrieving them over the network for each access. Specifically, a micro-task A is allowed to use a cached parameter if its value reflects all updates made by all micro-tasks that A depends on. This means that (1) multiple accesses of the same parameter by micro-task A can use the same cached value, and (2) a micro-task B whose dependencies are a subset of A's can use the same cached values that were used by A. By only using the sequence of micro-task dispatch and completion events to infer dependencies, Litz enables both (1) and (2) to be implemented efficiently. In particular, the dependencies of micro-task B are a subset of the dependencies of micro-task A if the total number of micro-tasks that have been completed when B was dispatched is at most the total number of micro-tasks that have been completed when A was dispatched.

To implement this cache coherence protocol, the master thread maintains a single monotonically increasing *version* number that is incremented each time `HandleTaskCompletion` is invoked. Whenever the driver dispatches a micro-task, the master thread tags the micro-task with the version number at that time. After micro-task A retrieves a fresh value from the parameter server, it caches the value and tags it with A's version. When micro-task B wants to access the same parameter, it first checks if its own version is less than or equal to the version of the cached value. If so, then the cached value is used; otherwise a fresh copy of the parameter is retrieved from the parameter server and tagged with B's version. A cache exists on each Litz process running at least one worker thread, so that it can be shared between different worker threads in the same process.

This cache coherence protocol allows Litz to automatically take advantage of parameter caching for applications that use bounded staleness. For example, to implement SSP (Sec. 4.2.1) with staleness s, all micro-tasks for iteration i are dispatched when the last micro-task for iteration $i - s - 1$ is completed. Thus, every micro-task for the same iteration has the same version and share cached parameter values with each other. Since the micro-tasks for iteration i are

dispatched before those for iterations between $i - s$ and $i - 1$ finish (when $s \geq 1$), the values they retrieve from the parameter server may not reflect all updates made in those prior iterations, allowing staleness in the parameter values being accessed.

Parameter Update Aggregation

Updates for the same parameter value may be generated many times by different micro-tasks. Since the parameter updates in ML applications are incremental and almost always additive, they can be aggregated locally before sending to the parameter server in order to reduce network usage. To facilitate the aggregation of updates, each Litz process contains an *update log* which maps parameter keys to locally aggregated updates. Whenever a micro-task invokes `PSUpdate`, the update is first aggregated with the corresponding entry in the update log, or is inserted into the update log if the corresponding entry does not exist. Therefore, an update sent to the parameter server can be a combination of many updates generated by different micro-tasks on the same Litz process.

In order to maximize the number of updates that are locally aggregated before being sent over the network, the results of micro-tasks are not immediately returned to the master thread after they are completed. Doing this allows the updates from many more micro-tasks to be sent in aggregated form to the server threads, reducing total network usage. The update log is periodically flushed by sending all updates it contains to the server threads to be applied. After each flush, all buffered micro-task results are returned to the master thread, which then informs the driver of their completion. The period of flushing can be carefully tuned, but we find that the simple strategy of flushing only when all micro-tasks on a worker thread are finished works well in practice.

Co-operative Multitasking

Litz employs co-operative multitasking implemented using co-routines [8]. When one task is blocked on an invocation of `PSGet` waiting for a value to be returned from a server thread, the worker thread will switch to executing another micro-task that is not blocked so that useful work is still performed. Each micro-task is executed within a co-routine so that switching between them can be done with low-latency, entirely in user-space. Using co-routines provides the benefit of overlapping communication with computation, while retaining a simple-to-use, synchronous interface for accessing the parameter server from micro-tasks.

4.5 Evaluation

We start by evaluating Litz's elasticity mechanism and demonstrate its efficacy along several directions. First, with its parameter caching, update aggregation, and co-operative multi-tasking, Litz is able to sustain increasing numbers of executors and micro-tasks with minimal performance impact. Second, a running Litz application is able to efficiently make use of additional nodes allocated to it, accelerating its time to completion. Third, a running Litz application is able to release its nodes on request, quickly freeing them to be allocated to another job.

Next, we discuss how Litz's elasticity can be leveraged by a cluster job scheduler to (1) reduce the completion time of an ML job that yields resources to a higher-priority job, and (2) improve resource allocation by exploiting the inherent decreasing memory usage of many ML algorithms.

Lastly, we evaluate Litz's performance when executing diverse applications which make use of stateful workers, model scheduling, and relaxed consistency. With the multinomial logistic regression (MLR) application, we show that our implementation on Litz is faster than the built-in implementation in Bösen [211], a non-elastic ML system for data-parallel SSP workloads. With the latent Dirichlet allocation (LDA) application, we show that our implementation on Litz is competitive with the built-in implementation in Strads [113], a non-elastic ML system for model scheduling. Furthermore, to evaluate Litz for the special case of deep learning, we implement a deep feed-forward neural network and compare its performance with Tensorflow [13].

ML Applications

MLR and LDA are popular ML applications used for multi-class classification and topic modeling, respectively. The goal of our evaluation is to show that Litz enables elasticity for these applications at little cost to performance when compared with state-of-the-art non-elastic systems. Thus, we closely follow their implementations in Bösen and Strads, using SGD and the SSP relaxed consistency model for MLR, and block-scheduled Gibbs sampling with stateful workers for LDA. For details of these implementations of MLR and LDA, we refer readers to their descriptions in Wei *et al.* [211] and Kim *et al.* [113], respectively.

Cluster Setup

Unless otherwise mentioned, the experiments described in this section are conducted on nodes with the following specifications: 16 cores with 2 hardware threads each (Intel Xeon E5-2698Bv3), 64GiB DDR4-2133 memory, 40GbE NIC (Mellanox MCX314A-BCCT), Ubuntu 16.04 Linux kernel 4.4. The nodes are connected with each other through a 40GbE switch (Cisco Nexus 3264-Q), and access data stored on an NFS cluster connected to the same switch. Each machine runs one Litz process which contains both worker threads and server threads; the master thread is co-located with one of these processes.

Input Datasets

Unless otherwise mentioned, we run MLR on the full ImageNet ILSVRC2012 dataset [187] consisting of 1.2M images labeled using 1000 different object categories. The dataset is pre-processed using the LLC feature extraction algorithm [207], producing 21K features for each image, resulting in a post-processed dataset size of 81GB. We run LDA on a subsample of the ClueWeb12 dataset [68] consisting of 50M English web pages. The dataset is pre-processed by removing stop words and words that rarely occur, resulting in a post-processed dataset with 10B tokens, 2M distinct words, and total size of 88GB.

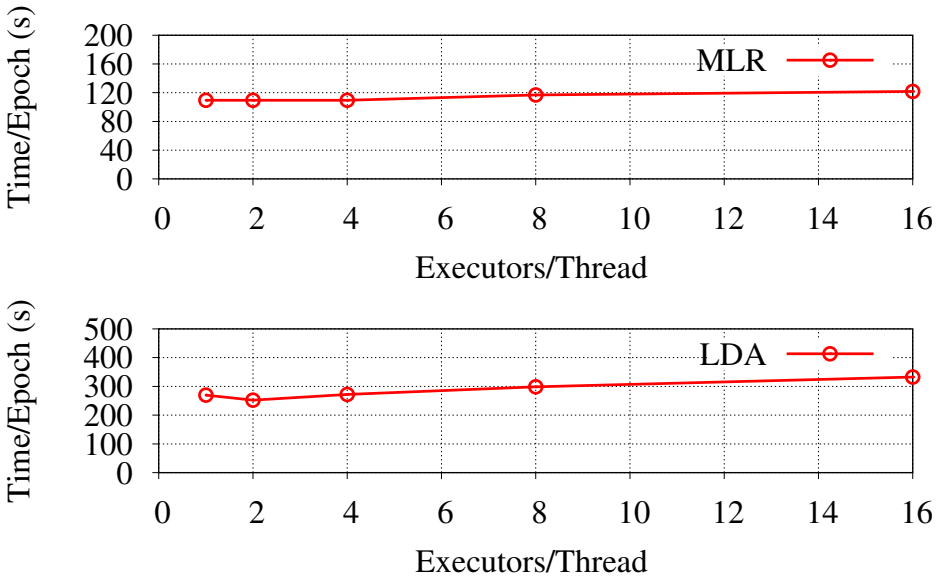


Figure 4.2: Average time per epoch for MLR and LDA when running with various numbers of executors per worker thread. In both cases the overhead of increasing the number of executors is insignificant. We define one epoch as performing a single pass over all input data.

4.5.1 Elasticity Experiments

Before discussing elastic scaling, we evaluate Litz's performance characteristics over increasing numbers of executors. The worker threads achieve elasticity by re-distributing executors amongst themselves when their numbers change, and by over-partitioning the application's state and computation across larger numbers of executors, Litz is able to scale out to larger numbers of physical cores and achieve a more balanced work assignment. Thus it is critical for Litz applications to still perform well in such configurations. We run the MLR application on 4 nodes and the LDA application on 12 nodes, varying the number of executors from 1 to 16 per worker thread. Fig. 4.2 shows how the throughput of each application changes when the number of executors increases. Using a single executor per worker thread as the baseline, the execution time for MLR does not noticeably change when using $4\times$ the number of executors, and gradually increases to $1.11\times$ the baseline when using $16\times$ the number of executors. For LDA, the execution time initially decreases to $0.94\times$ the baseline when using $2\times$ the number of executors, and thereafter gradually increases to $1.23\times$ the baseline when using $16\times$ the number of executors. We believe the overhead introduced by increasing the number of executors is quite an acceptable trade-off for elasticity and can still be reduced with further optimizations.

Elastic Scale Out

As jobs finish in a multi-tenant setting and previously used resources are freed up, additional allocations can be made to a currently running job. It is therefore important for the job to be capable of effectively using the additional resources to speed up its execution. In this section, we evaluate Litz's performance characteristics when scaling a running application out to a larger

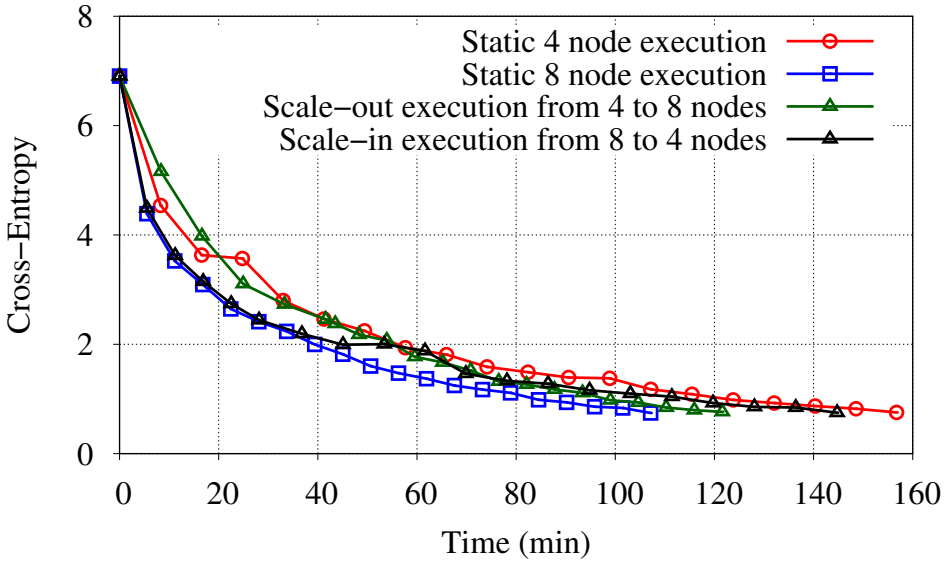


Figure 4.3: MLR execution on Litz with 4 nodes, with 8 nodes, with an elastic execution that scales out from 4 nodes to 8 nodes, and with an elastic execution that scales in from 8 nodes to 4 nodes. For the scale-out execution, the nodes are added at about 40 minutes into execution. For the scale-in execution, the nodes are removed at about 30 minutes into execution.

number of physical nodes. We run experiments scaling MLR jobs from 4 to 8 nodes, and LDA jobs from 12 to 24 nodes. Each node runs both worker threads and server threads, so both executors and PSshards are rebalanced during scaling. The experiments for LDA in this section were performed using m4.4xlarge instances on AWS EC2, each with 16 vCPUs and 64GiB of memory.

To evaluate the speed-up achieved, we compare our scale-out experiments with static executions of the applications using both the pre-scaling number of nodes and the post-scaling number of nodes. Fig. 4.3 shows the convergence plots for MLR, 4 new nodes added after $\approx$40min of execution. The static 4 node execution completes in $\approx$157min while the scale-out execution completes in $\approx$122min, resulting in a 22% shorter total run-time. Fig. 4.4 shows the convergence plots for LDA, 12 new nodes added after $\approx$55min of execution. The static 12 node execution completes in $\approx$183min while the scale-out execution completes in $\approx$134min, resulting in a 27% shorter total run-time.

Ideal Scale Out

Next, we evaluate the amount of room for improvement still achievable over Litz's current scale-out performance. Following a similar construction as Pundir et al. [173], we define and compare with a simple *ideal* scale-out execution time which intuitively measures the total run-time of a job that instantly scales out and adapts to use the additional nodes. For example, consider a job that scales out from 4 to 8 nodes after completing 30% of its iterations, its ideal scale-out execution time is the sum of the time at which the scale-out was triggered and the time it takes a static 8 node execution to run the last 70% of its iterations.

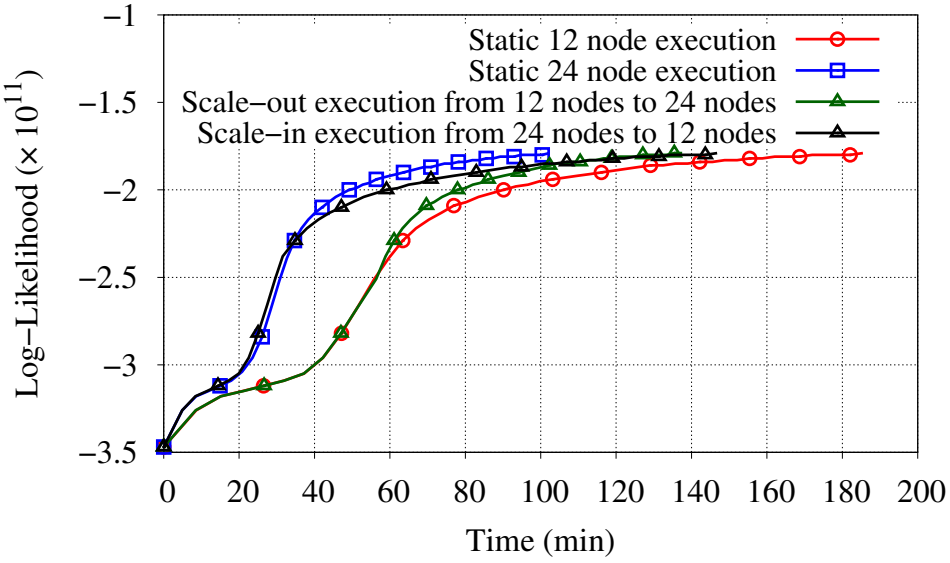


Figure 4.4: LDA execution on Litz with 12 nodes, with 24 nodes, and with an elastic execution that scales out from 12 nodes to 24 nodes. For the scale-out execution, the nodes are added at about 55 minutes into execution. For the scale-in execution, the nodes are removed at about 33 minutes into execution.

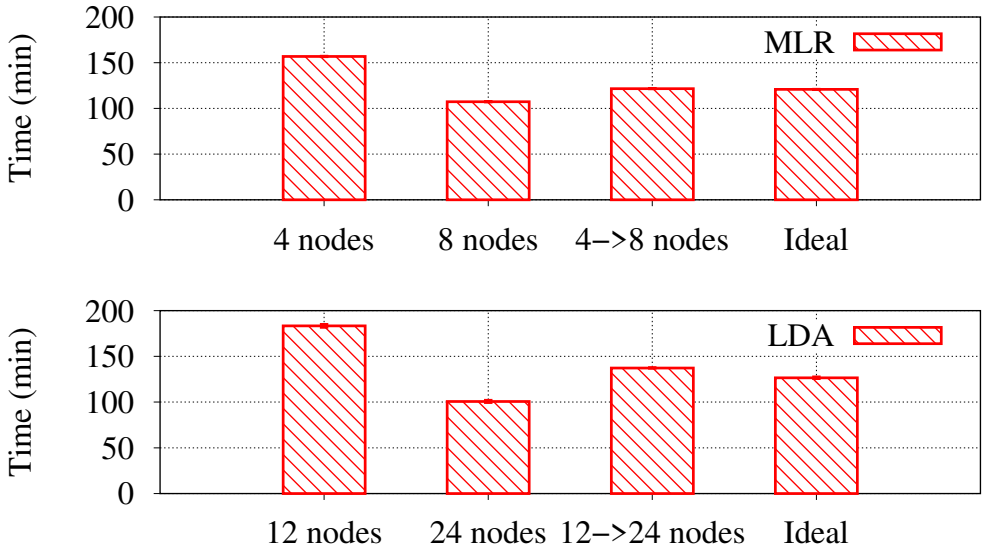


Figure 4.5: Static, scale-out, and ideal scale-out (See Sec. 4.5.1) execution times for MLR and LDA implemented on Litz. We scale out MLR from 4 nodes to 8 nodes, and LDA from 12 nodes to 24 nodes. Each experiment was performed several times, error bars are omitted due to their negligible size.

Fig. 4.5 compares the static pre-scaling, static post-scaling, scaling, and ideal execution times for both MLR and LDA. For MLR, the static 8 node execution completes in $\approx$107min, giving an ideal scale-out execution time of $\approx$121min. The actual scale-out execution time is $\approx$122min, indicating a less than 1% difference from the ideal. Similarly for LDA, the static 24 node execution completes in $\approx$101min, giving an ideal scale-out execution time of $\approx$127min. The actual scale-out execution time is $\approx$134min, indicating a 5% difference from the ideal. LDA's higher overhead stems from the large worker state that is inherent to the algorithm, which need to be serialized and sent over the network before the transferred executors can be resumed. We believe this overhead can be reduced further through careful optimization of the serialization process, by minimizing the number of times data is copied in memory and compressing the data sent over the network.

Elastic Scale In

As new and higher-priority jobs are submitted in a multi-tenant environment, the resource allocation for a currently running job may be reduced and given to another job. In this section, we evaluate Litz's scale-in performance based on two key factors. First, we show that Litz applications continue to make progress after scaling in, with performance comparable to the static execution on the fewer nodes. Second, we show that running Litz jobs can release resources with low latency, quickly transferring executors and PSshards away from requested nodes so that they can be used by another job. We measure the time between when the scale-in event is triggered and when the last Litz process running on a requested node exits. This represents the time an external job scheduler needs to wait before all requested resources are free to be used by another job. As with the scale-out experiments, these experiments were run using m4.4xlarge EC2 instances.

We run each experiment at least three times and report the average. Fig. 4.3 shows the convergence plots for MLR with the scale-in event. We start the job with 8 nodes, and remove 4 nodes $\approx$ 30 minutes into execution. The convergence plot closely follows the plot of 8-node static execution until the scale-in event, and the plot of 4-node static execution after that. Similarly, Fig. 4.4 shows the convergence plots for LDA with the scale-in event. We start the job with 24 nodes, and remove nodes $\approx$ 33 minutes into execution. The convergence plot closely follows the plot of 24-node static execution until the scale-in event, and the plot of 12-node static execution after that.

For MLR, the scale-in event takes 2.5 seconds on average, while for LDA the average is 43s. The low latency for MLR is due to a combination of its stateless workers and Litz's default behavior of discarding input data upon scaling in. As a result, the only state that needs to be transferred are the PSshards residing on the server threads of each requested node, which total $\approx$10MiB when split between 8 nodes. The executors in LDA, on the other hand, are stateful and contain a portion of its model parameters. When distributed across all nodes, each node contains $\approx$4.6GiB of executor state that need to be transferred away. A benchmark of cluster network showed that it can sustain a bandwidth of 2.0Gbps between pairs of machines, meaning that the 4.6GiB of LDA executor state can ideally be transfered within 20s. Nevertheless, the current transfer times are reasonable for an external scheduler to wait for. For comparison, even a pre-emptive environment like the AWS Spot Market gives users a warning time of 120s before

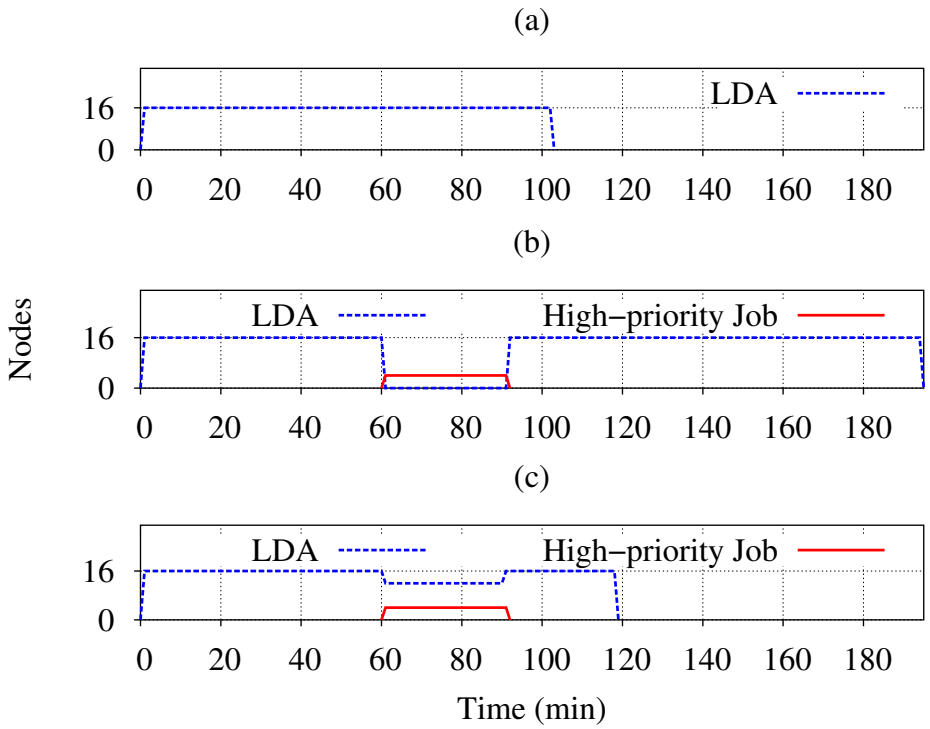


Figure 4.6: Priority scheduling experiments as described in Sec. 4.5.2. The graphs show the resource allocation over time in the cases of (a) LDA job which is uninterrupted, (b) LDA job which is killed when a higher-priority job is submitted, and (c) LDA job which elastically scales in when a higher-priority job is scheduled. We ran each experiment three times and saw negligible variation between each instance.

forcefully evicting their nodes.

4.5.2 Elastic Scheduling

We present two specific instances where the elasticity enabled by Litz can benefit job scheduling. First, when a high-priority job needs to be scheduled, an elastic ML application can avoid preemption by cooperatively releasing resources. Second, the inherent resource variability of many ML applications allow Litz to automatically release memory throughout the lifetime of an ML job, freeing resources to be used by other jobs.

Priority Scheduling

In multi-tenant computing environments, users frequently submit jobs (both ML and non-ML) which can have differing priorities. To meet the stricter SLA requirements of high-priority jobs, a scheduler must sometimes re-allocate some resources used by a lower-priority job. If the lower-priority job is inelastic, then it may be killed or suspended, leaving the rest of its resources under-utilized and delaying its completion time. For long-running jobs such as training ML models, their resources may need to be re-allocated several times during their lifetimes.

However, with the elasticity mechanism enabled by Litz, a long-running ML application can simply scale-in to use a fewer amount of resources, while the higher-priority job uses the released resources. After the higher-priority job completes, it can scale-out again, uninterrupted. We implemented this priority scheduling policy on a cluster of 16 m4.4xlarge nodes, and launched an LDA job on all 16 machines that runs for ≈100min if left uninterrupted (Fig. 4.6(a)). A higher-priority job is launched 60min into its runtime, requiring 4 nodes for 30min. Without elasticity, the LDA job is killed and re-started after the higher-priority job ends, requiring a total of ≈190min to complete (Fig. 4.6(b)). However, by leveraging elasticity to scale-in the LDA job, it can continue to run using 12 nodes and completes in ≈120min (Fig. 4.6(c)). At the same time, waiting for LDA to scale-in only increased the completion time of the high-priority job from 30min to 31min.

ML Resource Variability

The iterative-convergent nature of ML algorithms presents opportunities for resource scheduling not usually found in other computing tasks. One advantage of elasticity in an ML framework is that in addition to scaling in and out based on the directions from a cluster scheduler, an elastic ML framework can leverage resource variability that is inherent in ML applications to autonomously give up resources.

In particular, many ML algorithms, including LDA, may find their model parameters becoming sparse (ie. mostly zeros) as they approach convergence [113], allowing memory usage to be reduced by using a more memory-efficient storage format (ie. sparse vector). Although LDA running on Strads has a similar decreasing memory usage, the lack of elasticity in Strads does not allow it to leverage this phenomenon for efficient scheduling.

Litz, on the other hand, can detect variability in the resource usage and reduce the number of worker and server threads accordingly. Fig. 4.7 shows the breakdown of memory usage during LDA. Server threads that store the model start with 6 GiB and drop to around 1 GiB by the 10th epoch, suggesting that the server threads can be reduced by 80%. Similarly, the worker threads start with 370 GiB of memory and reduce to about 300 GiB by the 10th epoch, suggesting that their count can be reduced by 20% and respective resources can be released. This dynamic resource usage of ML jobs, when exposed through an elastic framework like Litz, can inform the policies of a cluster scheduler that allocates resources between many jobs.

4.5.3 Performance Experiments

We compare our Litz implementations of MLR and LDA with those built-in with the open-source versions of Bösen and Strads, respectively. All three systems along with their applications are written using C++, and to further ensure fairness, we compiled all three using the `-O2 -g` flags and linked with the TCMalloc [71] memory allocator. These settings are the default for both Bösen and Strads.

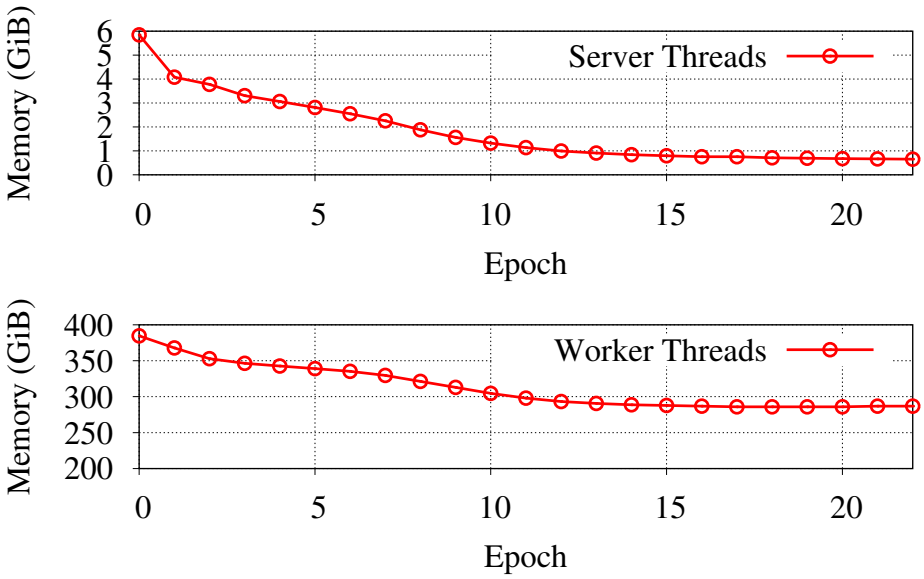


Figure 4.7: Memory usage on a cluster of 12 m4.4xlarge nodes during runtime of LDA implemented using Litz, broken down by server threads and worker threads. During the first 10 epochs, memory usage of server threads decrease by 5GiB, while memory usage of worker threads decrease by 70GiB.

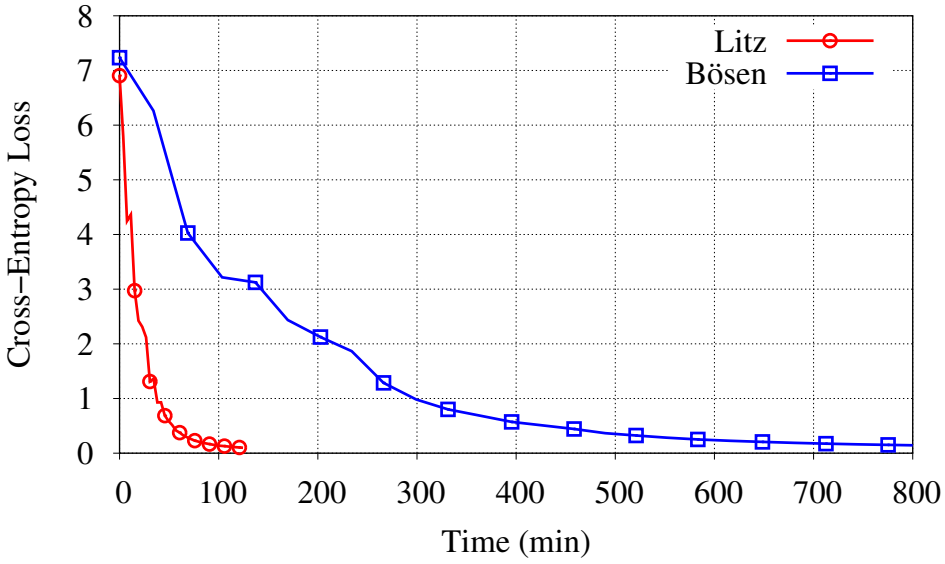


Figure 4.8: Multinomial Logistic Regression (MLR) running on 8 nodes using 25% of the ImageNet ILSVRC2012 dataset. Litz achieves convergence about 8× faster than Bösen.

MLR Comparison with Bösen

We compare Litz with Bösen running the MLR application on 25% of the ImageNet ILSVRC2012 dataset[2] using 8 nodes. The open-source version of Bösen differs from the system described by Wei *et. al.* [211] in that it does not implement early communication nor update prioritization, but is otherwise the same and fully supports SSP execution. Both MLR instances were configured to use the same SSP staleness bound of 2 as well as the same SGD tuning parameters such as step size and minibatch size. As Fig. 4.8 shows, our MLR implementation on Litz converges about 8× faster than that on Bösen. Our profiling of Bösen and cursory examination of its code shows that it does not fully utilize CPUs due to lock contention. We believe the wide gap in performance is not due to fundamental architectural reasons, and that Bösen should be able to narrow the gap on such SSP applications given a more optimized implementation.

LDA Comparison with Strads

We next compare Litz with Strads running the LDA application using 12 nodes. The open-source version of Strads is the same implementation used in Kim *et. al.* [113]. Both LDA instances were configured to use the same number of block partitions as well as the same LDA hyper-parameters α and β. We ran each application until 34 epochs have been completed, where an *epoch* is equivalent to a full pass over the input data. As Fig. 4.9 shows, our LDA implementation on Litz completes all epochs roughly 6% slower than that on Strads. However, it also achieves a better objective value (measured in log-likelihood), resulting in faster convergence than Strads overall. Even though more investigation into the per-epoch convergence difference is needed, we can attribute the throughput difference to the optimizations built into Strads, which employs a ring-topology specifically optimized for the block-partitioned model scheduling strategy used by LDA.

Deep Neural Networks (DNNs)

To evaluate Litz with DNNs, we implemented a particular deep learning model called a deep feed-forward network [75], which forms the basis of many deep learning applications. We used a network with two hidden layers with ReLU activation and one output layer with Softmax activation. We trained this model using both Litz and TensorFlow [13] on 4 m4.4xlarge EC2 instances, with the CIFAR-10 [119] dataset. This dataset consists of 60K images, which are pre-processed into vectors of ≈98K features, labeled using 10 classes. Both systems used the same data-parallel SGD algorithm, and were configured with the same tuning parameters such as a learning rate of 0.0001 and mini-batch size of 64. The training using Tensorflow progressed at a pace of ≈79s per batch, while the training using Litz progressed 3.4× faster at a pace of ≈23s per batch.

[2]With the full dataset, the Bösen baseline does not complete within a reasonable amount of time.

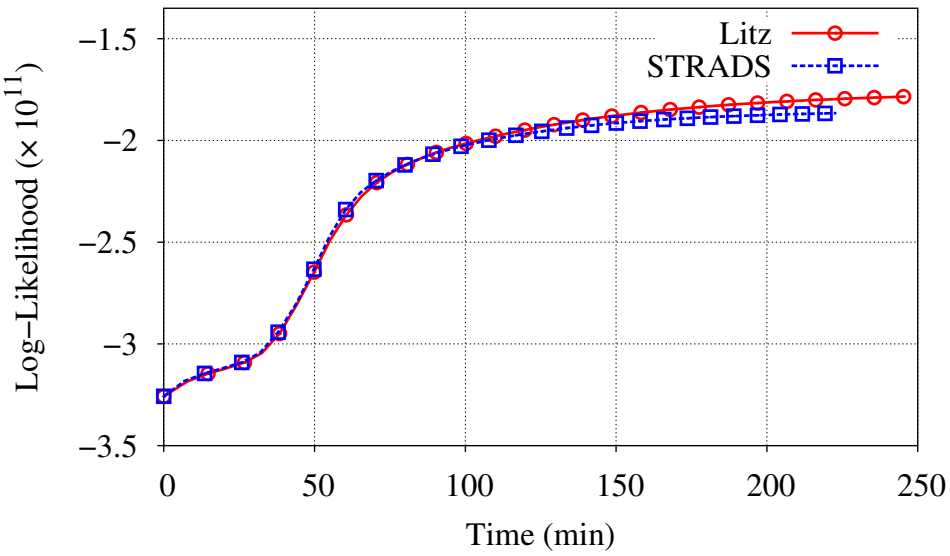


Figure 4.9: Latent Dirichlet Allocation (LDA) training algorithm running on Strads and Litz with the subsampled ClueWeb12 dataset. Litz completes all 34 epochs roughly 6% slower than Strads, but achieves a better objective value.

4.6 Discussion and Related Work

Recently, there has been a growing interest in utilizing *transient* nodes in the cloud spot markets for big-data analytics. The systems developed for this setting try to execute jobs with the performance of *on-demand* nodes at a significantly cheaper cost, using transient nodes. The challenge for these systems is to deal with the bulk revocations efficiently by choosing right fault-tolerance mechanism. For example, SpotOn [200] dynamically determines the fault-tolerance mechanism that best balances the risk of revocation with the overhead of the mechanism. While SpotOn applies these fault-tolerance mechanisms at the systems level—using virtual machines or containers—Flint [194] argues that application-aware approach is preferable and can improve efficiency by adapting the fault-tolerance policy. Flint, which is based on Spark, proposes automated and selective checkpointing policies for RDDs, to bound the time Spark spends recomputing lost in-memory data after a bulk revocation of transient nodes. TR-Spark [220] argues that RDDs—the checkpointing unit in Spark—are too coarse-grained, making Spark unfit to run on transient resources, and takes Flint's approach further by providing fine-grained task-level checkpointing.

Unlike Flint and TR-Spark that adapt a general-purpose Spark framework to achieve cost-effective analytics with transient resources, Proteus [83] adapts a specialized ML framework to achieve significantly faster and cheaper execution, while introducing elasticity optimizations tuned for the setting. Specifically, Proteus stores the ML model on parameter servers that run on reliable on-demand nodes, and makes the workers stateless so that they can be run on transient node, effectively pushing workers' states to parameter servers, along with the model. This is a reasonable approach for the spot market setting where bulk revocations can take offline a large

number of workers without notice. Although it works well for applications with small worker state, with an increasing data and model size, the approach may run into performance problems due to the communication overhead between workers and their state stored on the parameter servers. Litz, on the other hand, keeps the worker state in the workers and assumes a cooperative cluster scheduler that will ask the running application to give up nodes and wait for state to be transferred away. This approach results in high performance while still providing elasticity.

4.7 Scalable and Elastic HDBSCAN with Litz

In this section, we present a highly-optimized implementation of HDBSCAN clustering using Litz. Clustering algorithms have found applications in a variety of fields involving exploratory data analysis, including document retrieval [205], image search [25], and bioinformatics [134]. HDBSCAN [36] is one such algorithm which is well-suited for these applications, being able to model clusters in many naturally occurring data. However, traditional implementations of HDBSCAN scale poorly with the size of data, requiring a full kNN graph [67] to be constructed over all points in the dataset.

Simpler clustering models such as k-means [145] are popular for data-intensive applications because they can be efficiently scaled to large datasets [191]. However, they also tend to have restrictions for the data and/or user, limiting their usefulness for real-world data. For example, they may assume that clusters are normally distributed around a mean point [143], the user knows the number of clusters ahead of time, or that there are no noise points in the data. These shortcomings are avoided by many hierarchical and density-based clustering algorithms like HDBSCAN.

In our work, we investigate the feasibility of scaling up HDBSCAN. We approach the problem by using two key optimizations:

1. Replace the computationally expensive kNN graph construction with a more scalable approximation algorithm.

2. Replace a step which finds a minimum spanning tree of the complete graph of all data points, by finding the minimum spanning tree of the kNN graph instead.

Using these optimizations, we obtain an efficient approximation to HDBSCAN which works well on large datasets. We implement this approach as a distributed system, and show that it achieves over 40x better performance than a popular exact implementation of HDBSCAN. Additionally, we find that the approximations we used result in minimal difference when compared with exact HDBSCAN results. We obtain reasonable output from a large word embedding dataset containing hundreds of clusters, demonstrating the flexibility of the approach.

4.7.1 Background

HDBSCAN

HDBSCAN is a hierarchical density-based clustering algorithm. It takes N points with dimensionality D and a distance metric d as input. The algorithm works in three phases:

1. Find the K-th nearest neighbor of each point according to the distance metric d, where K is a hyperparameter. By a_p we denote the distance between point p and its K-th nearest neighbor. In subsequent steps, HDBSCAN uses the following modified distance metric:

$$\tilde{d}(x, y) = \max(a_x, a_y, d(x, y))$$

   The purpose of using this modified metric is to reduce sensitivity to outliers.

2. Find the minimum spanning tree of the points, where for each pair of points x and y there is an edge connecting them with weight $\tilde{d}(x, y)$.

3. Extract clusters from the minimum spanning tree.

Steps (1) and (2) are computationally expensive: they both have time complexity $O(N^2D)$. An algorithm which is quadratic in the number of points will not scale to datasets containing millions of points.

One technique for dealing with such datasets is sub-sampling, where the algorithm is run on a fraction of the dataset. This can be effective, but it is not suitable in all instances. For an example, consider the word embedding dataset described in Sec. 4.7.3. This dataset contains hundreds of clusters of fewer than 10 points, which would be difficult to extract from a sub-sample of the dataset.

Previous work on HDBSCAN [151] used k-d trees [26] to optimize step (1), but the worst-case complexity of step (1) is unchanged. As our experiments will show, this worst-case complexity is realized when the dimensionality D is sufficiently large. This is why we turn to approximation to scale the algorithm to large N and D.

NN-Descent

NN-Descent [58] is an approximate kNN graph construction algorithm. It is based on the principle that "a neighbor of a neighbor is likely to be a neighbor". A list of K candidate nearest neighbors is maintained for each point. In each step, for each point p, the algorithm examines all neighbors of neighbors of p and adds them to the neighbor list of p if they are closer than the last entry of p's neighbor list (thus replacing this last entry). This improves the accuracy of the kNN graph in each iteration. The algorithm stops when the number of updates in an iteration falls below a certain threshold. NN-Descent possesses several desirable qualities which make it well-suited for this application:

- It has been shown to perform well on a variety of datasets and distance metrics [58].
- It supports useful non-metric spaces such as the space induced by the cosine similarity metric.
- It has an $O(n^{1.14})$ empirical runtime, which is consistent with our results and makes it scalable to larger datasets.
- It can be easily distributed to multiple cores and machines.

4.7.2 Design and Implementation

To approximate step (1) of HDBSCAN, we run NN-Descent.

To approximate step (2), we find the minimum spanning tree of the graph produced by NN-Descent (which has KN edges) rather than the all-pairs graph (which has $N(N-1)/2$ edges). This choice is justified as follows. The only important edges of the minimum spanning tree used by step (3) are those which connect points in the same cluster. Thus we expect that removing edges not present in the k-NN graph will preserve most of the important edges.

We implemented our HDBSCAN application using Litz. Each worker is implemented as a separate executor and messages sent between workers are facilitated using the `PSGet` and `PSUpdate` API calls. Therefore, our implementation of HDBSCAN is not only efficient but can also elastically scale in and out through Litz. The minimum spanning tree is found using Kruskal's algorithm [122] when the points fit on one machine's memory and Borůvka's algorithm [32] when they must be split across machines. Most of the computational cost lies in running NN-Descent, so we devoted significant effort to optimizing this part of the algorithm.

We modified NN-Descent to achieve constant-factor improvement in the distributed case. In NN-Descent, points are partitioned evenly among the machines, and each machine keeps track of the nearest neighbors of its points. To update these nearest neighbor lists, an update is sent consisting of x, y, and $d(x, y)$ to the machine holding x. The update is accepted if $d(x, y)$ is less than the distance between x and the most distant element of the nearest neighbor list of x, denoted $d_{\text{max}}(x)$.

Sending data over the network is more expensive than accessing local memory, so we send $d_{\text{max}}(x)$ to all neighbors of x at the start of each iteration of updates. That allows those neighbors to discard updates to x before sending them through the network if the distance of the update is larger than $d_{\text{max}}(x)$. This technique led to a 42% decrease in runtime when running NN-Descent on the GloVe dataset [172] (described in the next section) with 4 machines.

4.7.3 Results

To test our implementation on real-world data, we ran it on the GloVe dataset containing 400,000 points with dimension 300. We imposed a maximum cluster size of 1,000 to avoid assigning all points to the same cluster. Using the dot product metric and running on 4 AWS m4.xlarge instances, our implementation finished in 143 seconds, producing 329 clusters as output. We find that the clusters produced are qualitatively valid, examples include:

- loaf loaves rigatoni tofu spaghetti orzo tacos snacks fettuccine penne dente linguine couscous sausage pasta cheese bread noodles sandwiches
- infinitive imperfective noun nominative pronoun dative genitive verb participle subjunctive accusative
- phi kappa epsilon theta sigma

To obtain a more quantitative measure of the accuracy of our approximation, we compare with the implementation by McInnes et al. using the Fowlkes-Mallows index [66] running on a synthetic dataset. The dataset consists of 100,000 points, with 1,000 clusters, where 20% of

Dimension	FM score	Clusters (theirs)	Clusters (ours)
1	0.896	6267	6285
2	0.959	840	842
3	0.934	1129	1135
4	0.941	1117	1115
5	0.942	1104	1100
6	0.948	1075	1079
7	0.956	1044	1050
8	0.957	1028	1029
9	0.958	1027	1023
10	0.961	1020	1018

Table 4.2: FM score measuring the similarity of our implementation's output with the output of the exact implementation by McInnes et al.

the points were noise. The remaining points were randomly distributed among the clusters; the points in a cluster were normally distributed about the center of the cluster. Table 4.2 summarizes these results.

We performed a similar experiment to test efficiency. We generated 10 synthetic clusters of points with dimension 10 and varied the number of points. We found that with 2^{23} points, our single-machine implementation was over 40 times faster. Fig. 4.10 shows how the implementations scale with the number of points. The distributed tests are run using 4 AWS m4.xlarge instances. The performance of our implementation is consistent with the $O(n^{1.14})$ empirical runtime of NN-Descent.

4.8 Adaptive Out-Of-Core Execution with Litz

In this section, we show how Litz can be extended to automatically adapt to low memory scenarios by invoking out-of-core execution. Our implementation using Litz is aware of when the partitions for each executor and parameter server are needed, and is able to outperform native OS-level swap by an order of magnitude.

For big data analytics and particularly for machine learning workloads, memory is often a scarce resource, and the lack thereof is a prominent source of failures [62]. Even modestly sized datasets or models may not fit within the available memory of the computing environment being used. Out-of-core execution techniques can alleviate the demand for main memory by allowing a significant amount of a running program's state to instead be placed in secondary storage.

The advantages of out-of-core execution often come at a performance penalty, and are only desired when a job is running with insufficient memory. On the other hand, it is difficult to set a tight bound on the memory requirements of machine learning jobs, since memory demand and availability for these jobs are often dynamic and change over time. First, the memory demand of certain machine learning applications exhibit trends across the lifetime of a job. For example, sparse models like Lasso [203] or Latent Dirichlet Allocation [31] (see Fig. 4.11) may see most of

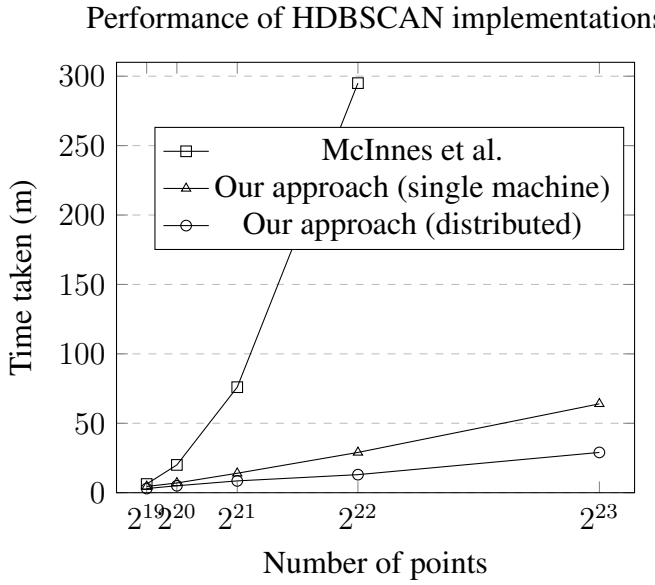


Figure 4.10: Performance of our implementation compared with the exact implementation by McInnes et al.

their parameters become zero as they converge, drastically reducing their memory consumption. Second, since machine learning applications are popular in shared computing environments such as clouds or data-centers, memory may become unavailable to a job when needed by another higher-priority job, or become available again when another job finishes. Whether or not to use out-of-core execution should depend on the demand and availability of memory at a given time. Thus, an *adaptive* form of out-of-core execution is desired, being automatically enabled for a job only when its demand for memory exceeds the amount available for it to use.

Paging to swap space is one solution for out-of-core execution. It has the benefits of being adaptive, only being used when constrained for memory, as well as being transparent to the application, requiring no special programming to be enabled. In addition, paging is implemented in the operating system and is able to bypass the overhead of the filesystem [6]. However, paging has several limitations for machine learning and big data workloads:

- The amount of swap space is usually limited. Its size is difficult to reconfigure dynamically, requiring privileged access to the operating system and can only be done when swap is disabled.

- Higher performance is difficult to achieve. Even though it's possible to tune swap for performance using the `madvise`, `mlock`, and `munlock` system calls [4, 5], they require knowledge of addresses in virtual memory, which are often hidden beneath many layers of data abstractions.

Our proposed adaptive out-of-core execution framework overcomes these limitations of swap. It can store program state using the filesystem, which is typically much larger than swap space

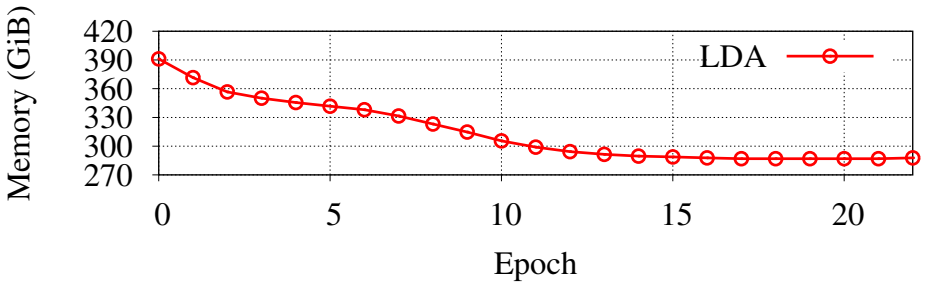


Figure 4.11: Aggregate memory usage of Latent Dirichlet Allocation (LDA) can decrease by more than 25% over its lifetime.

and offers enough flexibility to implement custom serialized formats that allow dynamic resizing without privileged access to the operating system. Although framework-level out-of-core execution precludes optimizations that bypass the overhead of using the filesystem, it enables other powerful optimizations which are application-aware and exploit the properties of machine learning to achieve higher performance:

- Knowing the dependencies between the application's tasks lets the framework prefetch state from secondary storage before they are needed, or re-order the execution of tasks to avoid memory thrashing.
- Knowing the type of application state (eg. shared parameters, data-local parameters [174]) informs the framework on their access patterns and the performance trade-off of evicting them to secondary storage.
- Lossy compression of serialized state can drastically reduce the overhead of reading and writing to secondary storage.

To show a proof of concept for how out-of-core execution can be performed adaptively and efficiently for machine learning, we designed and implemented an adaptive out-of-core system for a parameter server [51, 91, 132, 211] framework in a way that is transparent to the application. Our performance results are promising for a popular machine learning application on various hardware configurations. Comparing with paging to swap, our system performs significantly faster while avoiding the swap space limitations.

4.8.1 System Design and Implementation

We build our out-of-core execution system into Litz. The majority of application state resides in executors and a distributed parameter server. Each executor stores a partition of input data and the model parameters which are co-located with that partition of input data, while the parameter server stores the model parameters which need to be accessed from multiple executors. A single *driver* dispatches *tasks* to individual executors, where each task is some computation that can read/update the local executor state and the shared parameter server state. The Litz framework treats the executor state, parameter server state, driver, and tasks as application-defined black

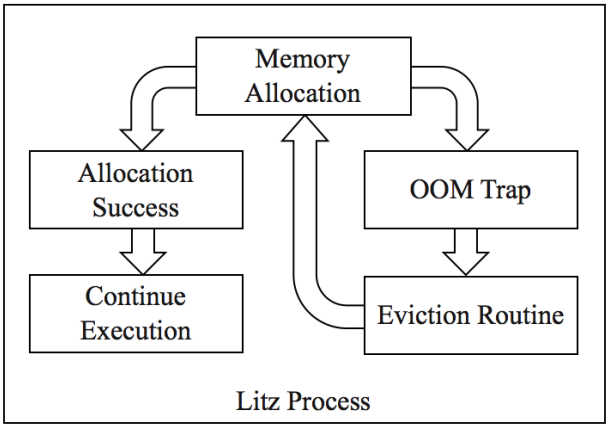


Figure 4.12: Flow of execution when memory is allocated within a Litz process.

boxes, but expects *serialization functions* to be defined for executor and parameter server state so they can be transparently migrated between different machines.

We enable adaptive out-of-core execution in the Litz framework with the addition of two sub-systems called the *Out-of-Memory (OOM) Trap* and the *Eviction Routine* embedded into each Litz process. The OOM Trap sub-system takes as input a memory threshold, and intercepts memory allocations made within the Litz process when it would surpass the limit. Each time the OOM Trap is triggered, it invokes the Eviction Routine, which searches for a suitable executor to be serialized (via the application-defined serialization function) and written to secondary storage. This executor is then erased from main memory, and the OOM Trap will retry the allocation which triggered it. Fig. 4.12 illustrates this process.

The OOM Trap can be be triggered multiple times for the same memory allocation, evicting multiple executors from main memory until the allocation succeeds. Before a task for an evicted executor is run, the executor is loaded back into main memory, which in turn may trigger other idle executors to be evicted. Since the OOM Trap is only triggered when constrained for memory, our out-of-core system is adaptive and only uses secondary storage when needed. In addition, the memory threshold can be changed during run-time, allowing more memory to be allocated to a job when available.

OOM Trap. The OOM Trap should accurately detect when an allocation will cause the memory limit to be exceeded, and stop the allocation from proceeding while allowing the Eviction Routine to run. In our implementation, we used `ulimit` [7] to limit virtual memory, causing memory allocations to return the null pointer when it would surpass the limit. The OOM Trap links a wrapper for `malloc` which checks if the result of allocations is the null pointer.

Eviction Routine. The Eviction Routine should find a suitable executor, serialize it, and write it to secondary storage. It is only invoked when the memory limit is about to be exceeded, and needs to run without allocating any additional memory. Each thread in the Litz process reserves a small amount of memory (2 MB) which can be used by the serialization function during the Eviction Routine. The Eviction Routine also attempts to find a more optimal executor to evict

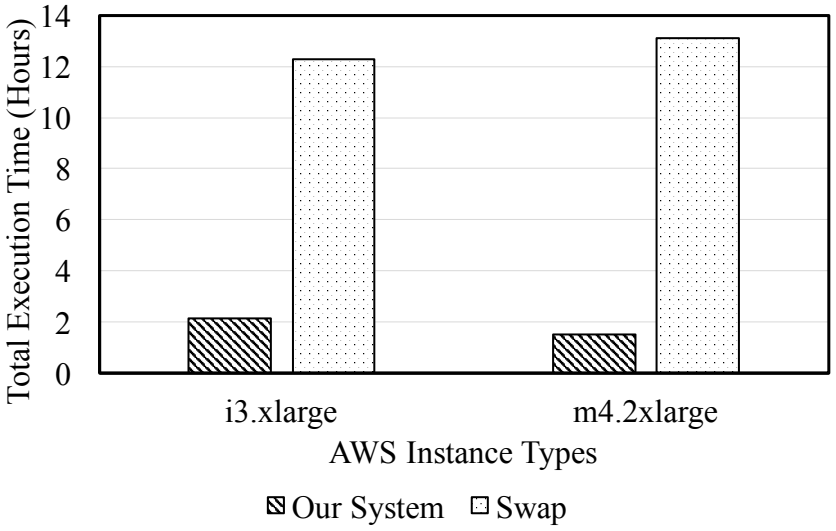


Figure 4.13: K-means experiment results on AWS instances.

by trying to avoid executors which have incoming tasks, by looking for tasks which have been dispatched but not started execution.

4.8.2 Results

We test Mini-batch K-means [138, 191] on the AWS cloud with two types of instances [1]. One is m4.2xlarge with 8 vCPU, 32GB memory. Another is i3.xlarge with 4 vCPU, 32GB memory. We set the number of computation threads to four on i3.xlarge and eight on m4.2xlarge, and use 256GB General Purpose SSD (gp2) EBS volumes [2]. The dataset is from the full ImageNet ILSVRC2012 dataset [186]. It contains 1140297 images, each with 10K features. We run 100 batches, with each batch 40 tasks and each task 1100 images, to train 1000 classes. For our system we use `ulimit -v 31000000` to limit the virtual memory size to a number close to the total physical memory size while for the system swap we let the OS schedule total 96GB virtual memory between the physical memory and the swap space.

The results show that on i3.xlarge our system uses 2.13 hours while the system swap uses 12.30 hours. On m4.2xlarge, our system takes 1.52 hours while the system swap takes 13.10 hours (see Fig. 4.13). The reason why the system swap is so slow is that its execution generates memory thrashing, which results in high IOPS and hits the performance upper limit of the EBS volumes.

Chapter 5

SCAR: Exploring the Inherent Fault Tolerance of Iterative-Convergent Training

Machine learning (ML) training algorithms often possess an inherent self-correcting behavior due to their iterative-convergent nature, providing opportunities to develop co-adaptive systems that leverage this property. Recent systems exploit this behavior 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 chapter, we develop a general framework to quantify the effects of calculation errors on iterative-convergent algorithms. Our framework provides a general tool to build co-adaptive systems based on the error tolerance property of ML training. 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, extends Litz (Chapter 4) with co-adaptation. 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.

The contents of this chapter were previously published in [176].

5.1 Introduction

Throughout an ML training 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 [90, 204], parameter synchronization can be bottlenecked on a slow or congested network [133, 230], and stragglers can severely impact overall job throughput [43, 82].

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

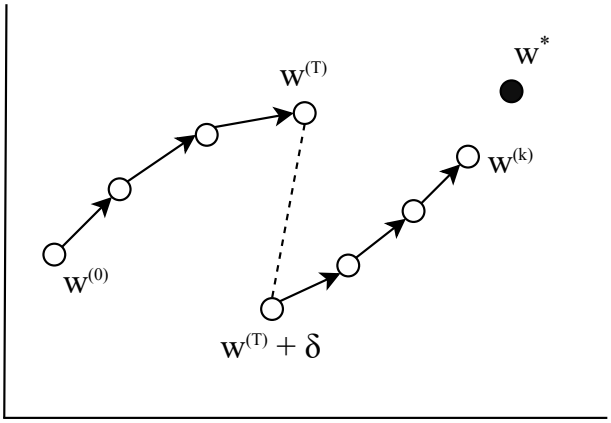


Figure 5.1: The self-correcting behavior of iterative-convergent algorithms. Even though a calculation error results in an undesirable perturbation of δ at iteration T, the subsequent iterations still brings the solution closer to the optimum value of w^*.

(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 unreliability and resource limitation issues, such as bounded staleness consistency [43, 45, 91], quantization and low-precision arithmetic [44, 80, 96], and lock-free execution [51, 162]. One notable exception to this trend is checkpoint-based fault tolerance, a common strategy in current ML systems for mitigating hardware failures [13, 141, 211] 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. 5.1). 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 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

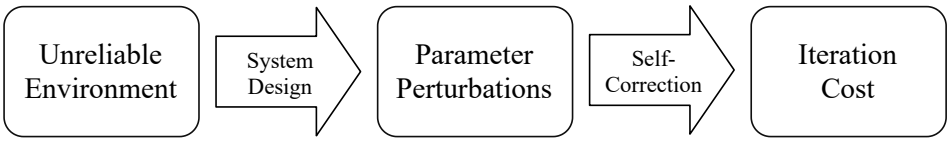


Figure 5.2: A framework for designing robust training systems with co-adaptation by exploiting the self-correcting behavior of ML. First, through system design, resource instabilities and constraints in unreliable computing environments are allowed to manifest as perturbations in model parameters. Then, through the self-correcting behavior of ML, the perturbations are automatically corrected but incurs a cost in the number of iterations to convergence.

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 based on Litz 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).

5.2 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 $w^{(k)}$ until convergence to some target parameter w^*. 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

$$w^{(k+1)} = f(w^{(k)}), \quad w^{(k)} \in \mathbb{R}^d, \tag{5.1}$$

for some function f. This model of iterative-convergent algorithms assumes that the current state $w^{(k)}$ 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 $w^{(k)}$ 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 $w^{(k)} + \delta_k$. 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:

$$\begin{aligned} y^{(0)} &= w^{(0)} \\ y^{(1)} &= f(y^{(0)} + \delta_0) \\ &\vdots \\ y^{(k+1)} &= f(y^{(k)} + \delta_k) \end{aligned} \tag{5.2}$$

In the absence of errors, ie. $\delta_k = 0$, we have $y^{(k)} = w^{(k)}$, which reduces to the basic iterative scheme (5.1). 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 5.2.1 (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 $|\delta_k| < 2^{-(p-1)}|y^{(k)}|$ [89].

Example 5.2.2 (Bounded Staleness Consistency)**.** In stochastic gradient descent (SGD) under the stale synchronous parallel (SSP) consistency model [91], gradients are computed in a data-parallel fashion where each of M machines may observe a stale version of the model parameters $\widetilde{w}_m^{(k)}$. Suppose $\nabla(\widetilde{w}_m^{(k)},\ D_m)$ are the gradients computed during iteration k using input data D_m at machine m. If $\nabla(w^{(k)},\ D)$ is the true stochastic gradient at iteration k, then the algorithm suffers a perturbation at iteration $k+1$ of:

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

Example 5.2.3 (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 $\delta_T = w^{(T)} - w^{(C)}$. 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.

Reduced precision (Example 5.2.1) and bounded staleness consistency (Example 5.2.2) have already been the focus of much attention in both the ML and systems communities [47, 102, 211, 228]. Although not typically studied within the explicit set-up of (5.2), 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 5.4, we apply the general set-up of (5.2) to devise

new strategies for checkpoint-based fault tolerance (Example 5.2.3) by reducing the sizes of the perturbations δ_k.

The iteration in (5.2) is closely related to *perturbed gradient descent* [59, 69, 104]. The main difference lies in the motivation: Jin et al. [104] show that by choosing δ_k cleverly, it is possible to escape saddle points and guarantee that the iteration (5.2) 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.

5.3 Analysis

Suppose that an ML system has experienced perturbations $\delta_1, \ldots, \delta_T$ up to the Tth iteration. A (random) sequence a_k is called *ε-optimal* if $\mathbb{E}\|a_k - w^*\| < \varepsilon$. 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 $w^{(k)}$?* We write "cost" in quotations to emphasize that this number can be negative—for example, δ_k could randomly move $y^{(k)}$ closer to w^*, or δ_k can be constructed in advance to improve convergence as in perturbed gradient descent. We call this quantity the *rework cost* of the perturbed sequence $y^{(k)}$, introduced in Sec. 5.1. Our goal in the present section is to bound the rework cost, which will be formally defined next.

5.3.1 Rework cost

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

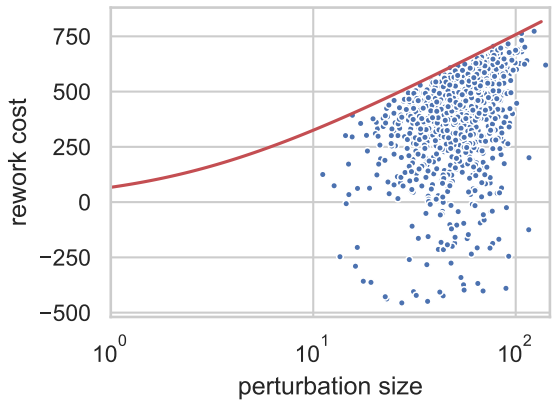
$$\|f(w^{(k)}) - w^*\| \leq c\,\|w^{(k)} - w^*\|, \quad 0 < c < 1, \tag{5.3}$$

i.e. the iterates $w^{(k)}$ 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 (5.3) as a baseline for comparison, and the analysis can be extended to more general schemes such as SGD if desired (Appendix 5.3.3).

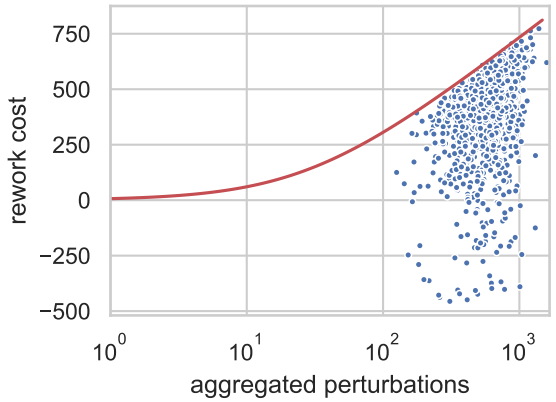
Formally, the rework cost is defined as follows: Let $\kappa(y^{(k)},\ \varepsilon)$ be a lower bound such that $m > \kappa(y^{(k)},\ \varepsilon)$ implies $\mathbb{E}\|y^{(m)} - w^*\| < \varepsilon$ (this may be $+\infty$ or negative). Under (5.3), it is straightforward to derive a similar lower bound for the unperturbed sequence $w^{(k)}$ as $\kappa(w^{(k)},\ \varepsilon) = \log\left(\frac{1}{\varepsilon}\|w^{(0)} - w^*\|\right)/\log(1/c)$. This will be used as a baseline for comparison: The rework cost for the perturbations δ_k is defined to be

$$\pi(\delta_k,\ \varepsilon) := \kappa(y^{(k)},\ \varepsilon) - \kappa(w^{(k)},\ \varepsilon). \tag{5.4}$$

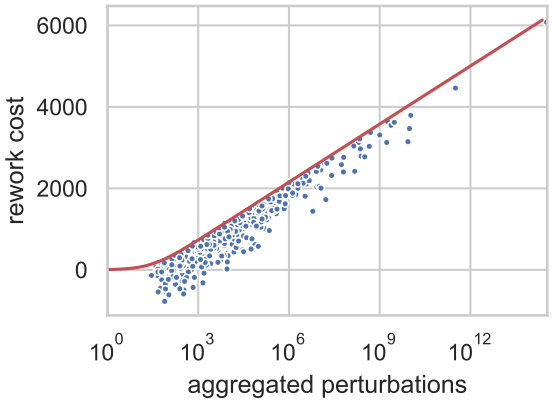
Using the unperturbed sequence $w^{(k)}$ as a benchmark, $\pi(\delta_k,\ \varepsilon)$ bounds the additional number of iterations needed for the perturbed sequence $y^{(k)}$ to reach ε-optimality (where we bear in mind that this can be negative). Clearly, $\pi(\delta_k,\ \varepsilon)$ depends on the 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.



(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 5.3: 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 5.3.1. 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.

Remark 5.3.1. We use the criterion $\mathbb{E}\|y^{(k)} - w^*\| < \varepsilon$ as an optimality criterion instead of directly bounding $\mathbb{P}(\|y^{(k)} - w^*\| < \varepsilon)$. This is commonly done [e.g. 33] since bounds on $\mathbb{E}\|y^{(k)} - w^*\|$ imply bounds on the latter probability via standard concentration arguments [see e.g. 181].

5.3.2 Bounding the rework cost

To bound the rework cost, we also require that the update f satisfies a convergence rate similar to (5.3) for the perturbed data $\widetilde{y}^{(k)} := y^{(k)} + \delta_k$:

$$\mathbb{E}\|f(\widetilde{y}^{(k)}) - w^*\| \leq c\,\mathbb{E}\|\widetilde{y}^{(k)} - w^*\|, \quad 0 < c < 1. \tag{5.5}$$

This simply says that wherever the algorithm is, on average, a single step according to f will not move the iterates further from w^*. 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 161). In fact, this assumption is satisfied for a variety of nonconvex problems [21, 218], and similar results hold for other optimization schemes such as proximal methods and Newton's method.
Under (5.3) and (5.5), we have the following general bound on the rework cost:

Theorem 5.3.1. *Assume $\mathbb{E}\|\delta_k\| < \infty$ for $k \leq T$ and $\delta_k = 0$ for $k > T$. Under* (5.3) *and* (5.5)*, we have for any $\varepsilon > 0$,*

$$\pi(\delta_k, \varepsilon) \leq \frac{\log\left(1 + \frac{\Delta_T}{\|w^{(0)} - w^*\|}\right)}{\log(1/c)} \tag{5.6}$$

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

In fact, the bound (5.6) is tight in the following sense: As long as (5.3) cannot be improved, there exists a deterministic sequence $\delta_1, \ldots, \delta_T$ such that (5.6) holds with equality. Theorem 5.3.1 is illustrated on a simple quadratic program (QP) in Figure 5.3, which provides empirical evidence of the tightness of the bound.

The interesting part of the bound (5.6) is the ratio $\Delta_T / \|w^{(0)} - w^*\|$, 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

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

Up to constants, the denominator is just the error of the original sequence $w^{(k)}$ 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 (5.6) is a ratio between the re-weighted perturbations and the expected 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 5.3.1 also assumes that there are no perturbations after time T. The idea is that *if* there are no more perturbations, (5.6) bounds the cost of the perturbations incurred so far. Of course, in practice, the system may experience faults after time T, in which case (5.6) 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 $\delta_k = w - y^{(k)}$ for some fixed $w \neq w^*$ and all $k > T$). Under some additional assumptions, something can be said about this case; see Example 5.3.4.

5.3.3 Examples

In this section, we discuss some examples where the bound (5.6) is applicable, along with some generalizations.

Example 5.3.1 (Convex optimization)**.** Theorem 5.3.1 applies to ML systems that are based on minimizing a strongly convex objective. This includes many classical problems including regression.

Example 5.3.2 (Nonconvex optimization)**.** If the loss function ℓ is nonconvex, then Theorem 5.3.1 still applies with some modifications. The assumptions (5.3) and (5.5) can be verified using known results on nonconvex optimization [21, 218] under the so-called *Kurdyka-Łojasiewicz property*, from which the bound (5.6) follows directly. Trouble arises, however, when ℓ has multiple basins of attraction: A perturbation δ_k could "push" the perturbed iterate $\widetilde{y}^{(k)}$ into a different basin, resulting in a limit point that is different from w^*. Theorem 5.3.1 continues to hold as long as this can be avoided, i.e. the δ_k are not too large.

Example 5.3.3 (SGD)**.** The assumption (5.5) does *not* hold for SGD, which has a sublinear convergence rate in general. Nonetheless, it is straightforward to extend our framework to sublinear algorithms, with the caveat that analogous bounds on the iteration cost become more complicated. In fact, it is not hard to see from our proof how to do this: Lemma B.2.1 in the Appendix establishes the following useful general inequality

$$\mathbb{E}\|y^{(k+1)} - w^*\| \leq c^{k+1}\big[\|w^{(0)} - w^*\| + \Delta_T\big].$$

Evidently, the factor of c governs how quickly Δ_T (i.e. the cost incurred by perturbations) gets washed out as k increases. For algorithms that converge sublinearly such as SGD, this effect will also be sublinear, but still tend to zero as long as the perturbations are not too large (see Appendix B.1.2 for a brief discussion). This is further corroborated by the empirical experiments in Section 5.5, where we show that the strategies for checkpoint-based fault tolerance proposed in the next section are successful on SGD as well as other optimization schemes such as alternating least squares. Similar arguments apply to convex (but not strongly convex) loss functions, for which gradient descent has a sublinear convergence rate in general.

Example 5.3.4 (Infinite perturbations)**.** An interesting case occurs when $\delta_k \neq 0$ for all k. In other words, there is a possibility of a fault in *every iteration*. For arbitrary δ_k, it is clearly impossible to establish any kind of convergence result. In fact, suppose $\|\delta_k\| \leq \Delta$ for each k.

Then there is an irreducible error of $(c/(1-c))\Delta$, meaning that we cannot hope to obtain an ε-optimal solution for any $\varepsilon < (c/(1-c))\Delta$. This helps to explain why we focus on the nontrivial case with $\delta_k = 0$ for $k > T$ in Theorem 5.3.1. One setting in which the analysis with infinite perturbations is nontrivial is when Δ is known to be small, e.g. when using reduced precision as in Example 5.2.1. This setting can be analyzed by setting $\Delta \geq 2^{-(p-1)}\|w^{(k)}\|$ for all k. For details, see Appendix B.1.1.

5.4 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 [13] and PyTorch [165].

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 [84]. 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 5.3.1 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 [91, 132, 133]—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 5.5.

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

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 to the model parameters and, according to Theorem 5.3.1, 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 5.4.1. *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 5.4.2. *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.

5.4.2 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 $w^{(0)}$ 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

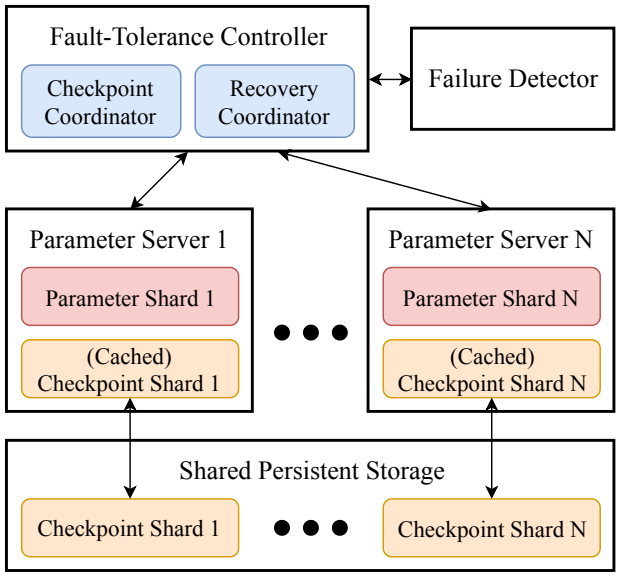


Figure 5.4: SCAR system architecture for partial recovery and prioritized checkpoints in distributed model training.

as the full checkpoint 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 5.5.5.

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 5.4 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 [97]. Checkpoints are saved to shared persistent storage, such as distributed filesystems like NFS [188], CephFS [213], or distributed databases like Cassandra [124]. 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.

Actions During a Checkpoint

1. The checkpoint coordinator sends a message to each PS node, which computes the distance of each of its parameters from their previously saved values in the running checkpoint using its in-memory cache.

2. Each PS node sends its model parameter IDs and computed distances to the checkpoint coordinator.

3. Upon receipt of the computed distances from all PS nodes, the checkpoint coordinator selects the fraction r of parameters with the largest distances, and sends their IDs back to their corresponding PS nodes.

4. Upon receipt of the parameter IDs, each PS node updates its in-memory cache, and saves those parameters to the shared persistent storage.

During step 4, the training algorithm can be resumed as soon as the in-memory caches have been updated, while output to the shared persistent storage happens asynchronously in the background. Thus, the checkpointing overhead in SCAR is just the time needed for prioritizing parameters and updating the in-memory cache. Furthermore, steps 2 and 3 simply answer a distributed top-k query. Although we chose a simple implementation for our prototype, a more scalable algorithm such as TPUT [38] can be used to remove the bottleneck of centralizing this computation onto the checkpoint controller.

Actions During a Failure

1. The failure detector notifies the recovery coordinator, which determines how the parameters belonging to the failed PS nodes should be re-partitioned.

2. The recovery coordinator partitions and sends the failed parameter IDs to the remaining PS nodes, which re-load the parameters from the current running checkpoint in shared persistent storage.

Implementation Details

SCAR is implemented using C++ and leverages an existing elastic ML framework [175], which provides mechanisms for transparently re-routing requests from workers away from failed PS nodes, as well as for new PS nodes to join the active training job, replacing the old failed PS nodes.

5.5 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.

5.5.1 Models and Datasets

We use several popular models and datasets as examples for our analysis and checkpoint strategies. We describe them in detail in this section.

Multinomial Logistic Regression (MLR)

Trained with stochastic (minibatch) gradient descent on the MNIST [126] and CoverType [57] datasets. The model parameters are an $M \times N$ matrix of real numbers, where M is the dimensionality of the data, and N is the number of output classes. When distributed, the rows of the parameter matrix are randomly partitioned. For MNIST, we use a batch size of 10,000, a learning rate of 1×10^{-5}, and a convergence criteria of 2.5×10^4 in cross-entropy loss. For CoverType, we use a batch size of 1,000, a learning rate of 1×10^{-7}, and a convergence criteria of 6.7×10^5 in cross-entropy loss. For both datasets, the convergence criteria is reached in roughly 60 iterations.

Matrix Factorization (MF)

Trained with alternating least squares (ALS) on the MovieLens [86] and Jester [72] datasets. The model parameters are matrices $L \in \mathbb{R}^{m \times p}$ and $R \in \mathbb{R}^{p \times n}$. When distributed, the rows of L and the columns of R are randomly partitioned. For MovieLens, we use 20 factors and a convergence criteria of 9.2×10^2 in mean squared error loss. For Jester, we use 5 factors and a convergence criteria of 5.57×10^3 in mean squared error loss. The MovieLens dataset is the `movielens-small` version consisting of 671 users and 9,125 items. The Jester dataset is the `Jester 2+` version, We further remove users with no ratings, and re-scale ratings from $[-10, 10]$ to $[0, 10]$. For both datasets, the factor matrices L and R are randomly initialized with each entry sampled uniformly at random from $[0, 1)$, and the convergence criteria is reached in roughly 60 iterations.

Latent Dirichlet Allocation (LDA)

Trained with collapsed Gibbs sampling [137] on the 20 Newsgroups [125] and Reuters [129] datasets. The model parameters are the document-topic and word-topic distributions. We use a scaled total variation between document-topic distributions as the norm for computing distances between parameters. When distributed, the document-topic distributions are randomly partitioned across nodes. We do not consider failures of word-topic distributions because they can be re-generated from the latent token-topic assignments.

In LDA, each document in the input data consists of a series of tokens, where each token is assigned a categorical topic. Topic assignments are repeatedly randomly sampled during the lifetime of a job. From these token-topic assignments, a document-topic distribution is constructed for each document, and a word-topic distribution is constructed for each unique word. Both the document-topic and word-topic distributions can be re-generated given the token-topic assignments, so losing the distributions themselves is not a problem. However, when distributed, each document-topic distribution is typically co-located with the document it corresponds to. Thus, losing a document-topic distribution is typically associated with also losing the token-topic assignments of that document, which do require recovery from a saved checkpoint. Therefore, we

only consider the loss of document-topic distributions, and assume that the word-topic distributions can be reconstructed at any time.

Since the parameters of LDA are distributions, a natural norm to use is the total variation norm. However, the total variation norm when applied to LDA puts the same weight onto every document-topic distribution. This means that re-sampling a token-topic assignment in a shorter document has a greater impact to the overall norm than re-sampling a token-topic assignment in a longer document, which biases checkpoint prioritization towards shorter documents. To address this, we scale the total variation norm of each document-topic distribution by the length of the document it corresponds to. The result is still a valid norm, since it is a positive linear combination (which is constant with respect to the input data) of total variation norms.

For 20 Newsgroups, we use a convergence criteria of 9.5×10^6 in negative log-likelihood. For Reuters, we use a convergence criteria of 8.5×10^5 in negative log-likelihood. For both datasets we train using 20 topics and hyperparameters $\alpha = \beta = 1$. The convergence criteria is reached in roughly 60 iterations.

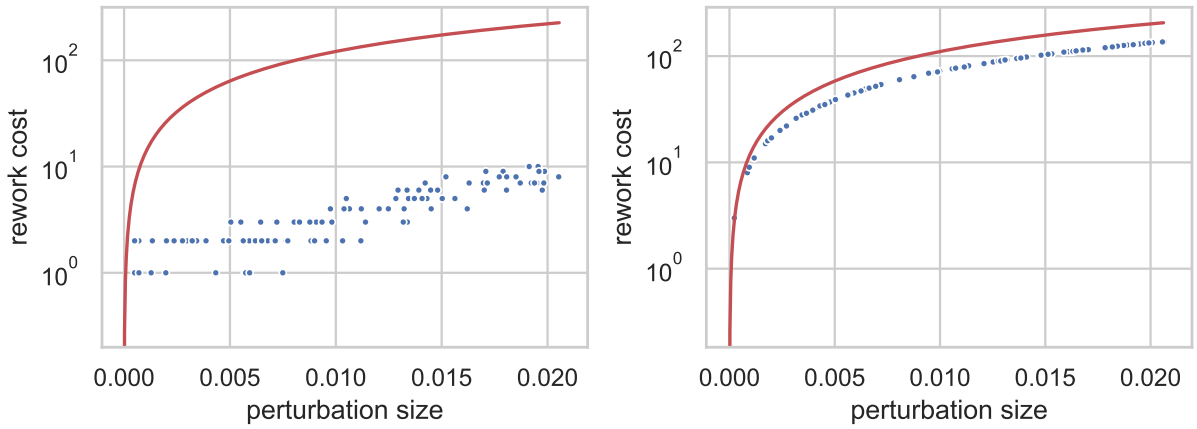
Convolutional Neural Network (CNN)

Trained with Adam [114]. We train this CNN on the MNIST [126] dataset.The network consists of 2 convolution layers with ReLU activations [156] and max pooling followed by 3 fully-connected layers with ReLU activation. Because of the structure in neural network models, we consider two different partitioning strategies: 1) In *by-layer* partitioning, we assume that the layers of the network are randomly partitioned across nodes; and 2) In *by-shard* partitioning, we further divide each layer's parameters into shards, and all shards are randomly partitioned across nodes. We use a batch size of 64, the recommended Adam settings of $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$, and a convergence criteria of 0.08 in cross-entropy loss. In by-layer partitioning, the weight and bias parameters are independent and partitioned separately (so they can either be lost together, or not). In by-shard partitioning, each parameter tensor is evenly partitioning according to its first dimension. The optimizer parameters (ie. the running first and second moment estimates in the case of Adam) are placed with their corresponding model parameters. Thus, an optimizer parameter is always faulted simultaneously with its corresponding model parameter.

5.5.2 Iteration Cost Bounds

To illustrate the behavior of the rework cost and to verify Theorem 5.3.1 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.

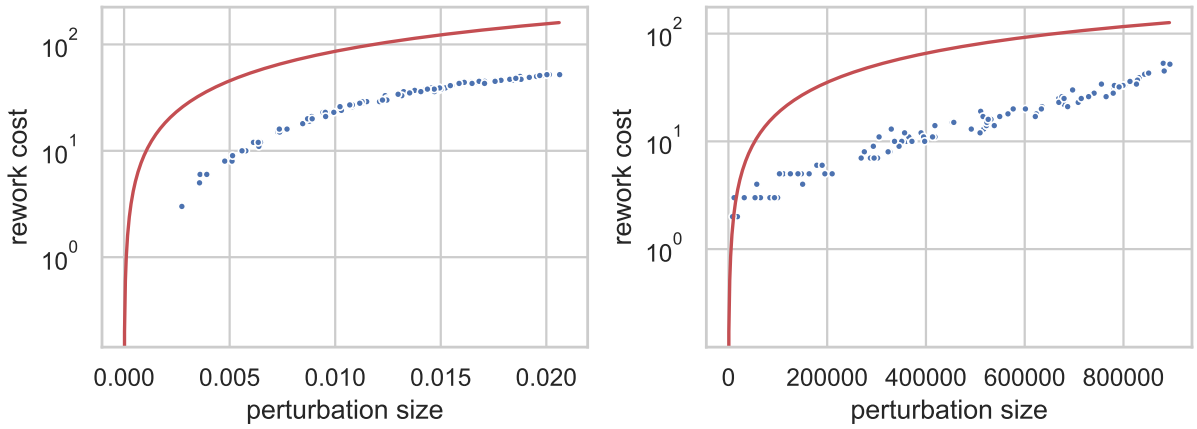
For random perturbations (Figure 5.5(a)), the rework cost bound is a loose upper bound on the actual rework cost. This is in contrast to the simpler quadratic program (QP) experiments shown in Figure 5.3, in which the bound is relatively tight. On the other hand, we also do not observe any perturbations 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.



(a) Random perturbations.

(b) Adversarial perturbations.

Figure 5.5: 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 5.3.1. 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 5.6: 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 5.5.

We run a second experiment in which we generate "adversarial" perturbations opposite the direction of convergence (Figure 5.5(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 5.5 shows the rework costs for synthetically generated perturbations, Figure 5.6 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 5.4.1. 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.

5.5.3 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. 5.7 and Fig. 5.8 show 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.

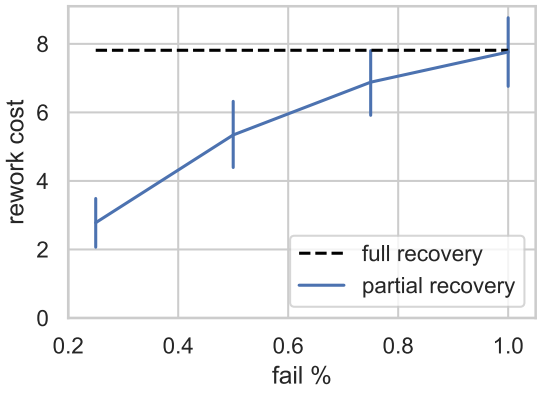
5.5.4 Priority Checkpoint

In this section, we evaluate the effectiveness of our priority checkpoint strategy for the MLR, MF, LDA, and CNN models. 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 5.4.2. As before, we sample the failure iteration from a geometric distribution. In this experiment keep the fraction of lost parameters fixed at $1/2$.

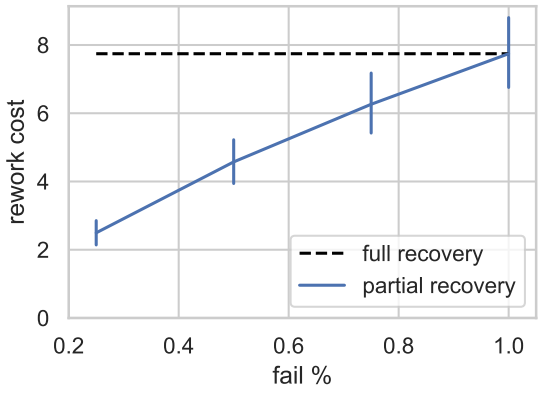
To gauge the effectiveness of prioritization, we compare between several strategies: (1) `priority`, parameters saved to checkpoint are selected based on the prioritization described in Section 5.4.2, (2) `round`, parameters saved to checkpoint are selected in a round-robin manner, and `random`, parameters saved to checkpoint are selected uniformly at random.

Fig. 5.9 and Fig. 5.10 show 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.

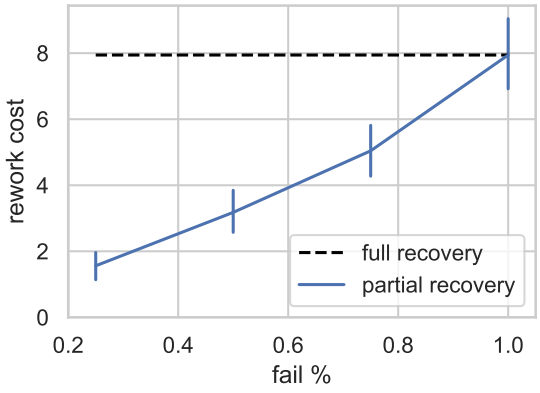
Across all models and datasets tested, combining partial recovery with prioritized $1/8$th checkpoints at $8\times$ frequency reduces the rework cost of losing $1/2$ of all model parameters by 78%–95% when compared with traditional checkpoint recovery.



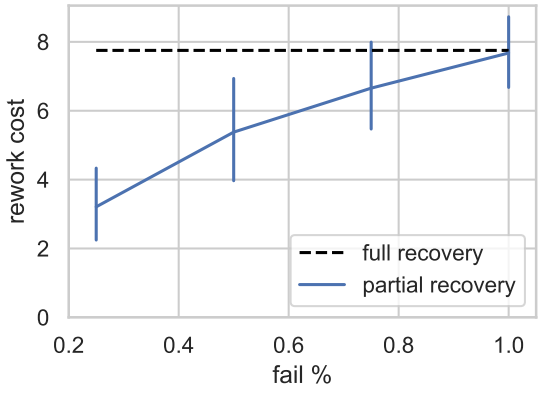
(a) MLR on CoverType.



(b) MLR on MNIST.

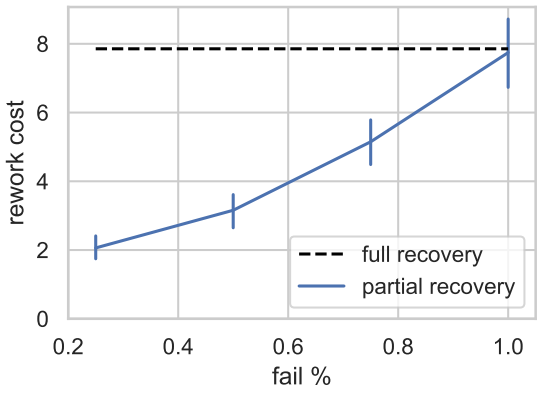


(c) MF on MovieLens.

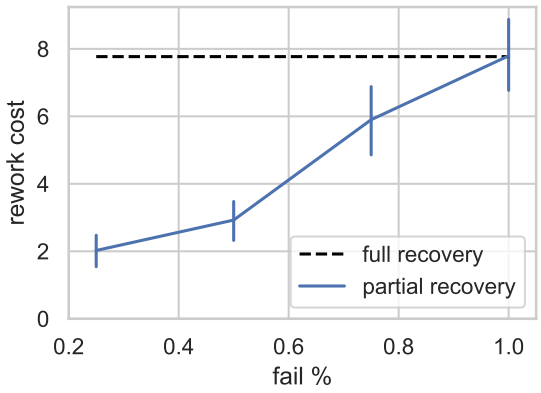


(d) MF on Jester.

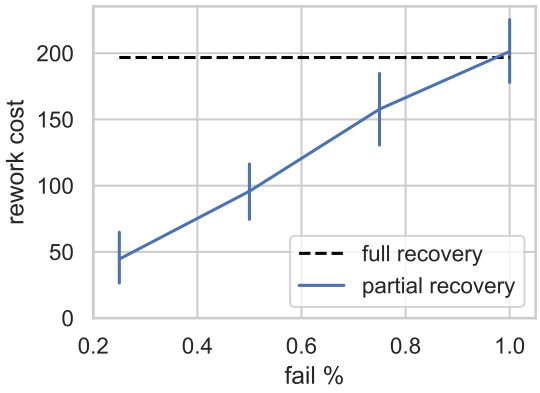
Figure 5.7: Partial vs. full recovery where the set of failed parameters are selected uniformly at random. The x-axis shows the fraction of failed parameters, and the y-axis shows the number of rework iterations. The error bars indicate 95% confidence intervals, calculated by repeating each trial 100 times, and the dashed black line represents the rework cost of a full checkpoint.



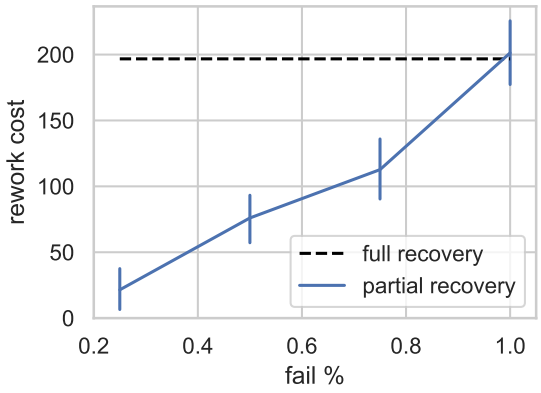
(a) LDA on 20 Newsgroups.



(b) LDA on Reuters.

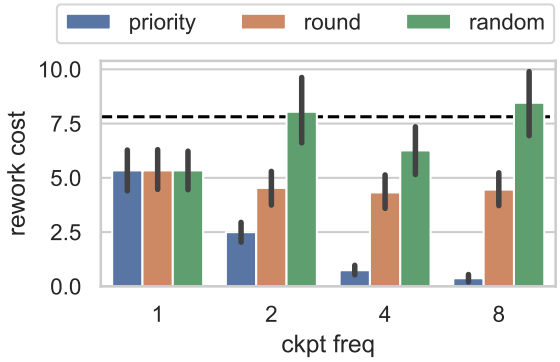


(c) CNN (by layer) on MNIST.

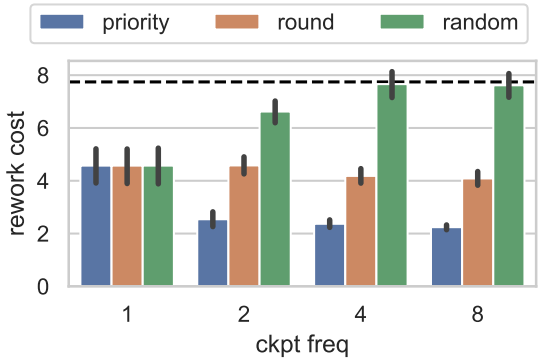


(d) CNN (by shard) on MNIST.

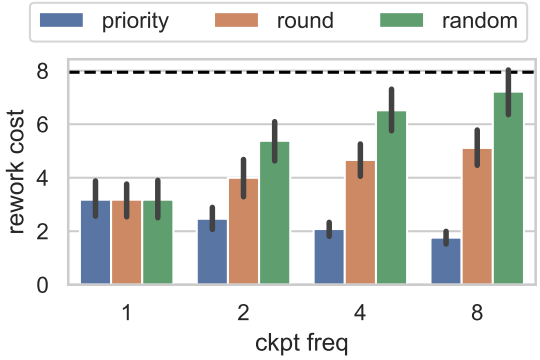
Figure 5.8: Partial vs. full recovery experiments (Fig. 5.7) continued.



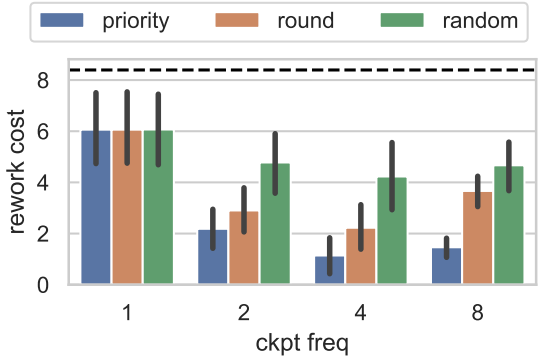
(a) MLR on CoverType.



(b) MLR on MNIST.

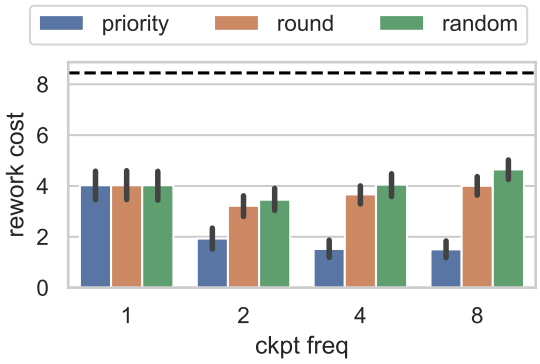


(c) MF on MovieLens.

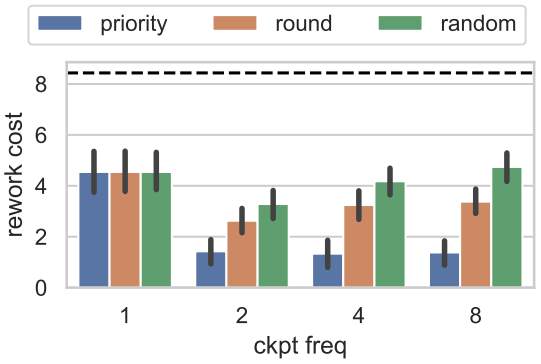


(d) MF on Jester.

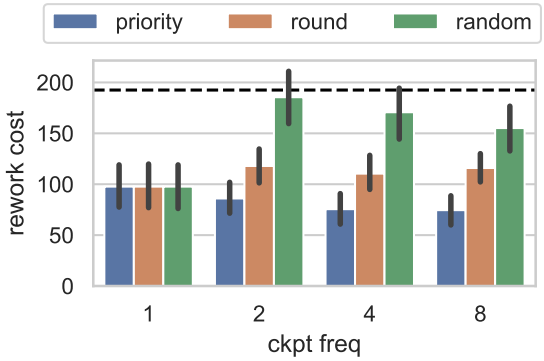
Figure 5.9: Prioritized checkpoint experiments comparing between the random, round-robin, and priority strategies. The x-axis indicated checkpoint frequency relative to full checkpoints, where 1 indicates full checkpoints, 2 indicates $1/2$ checkpoints at $2\times$ frequency, etc., and the y-axis shows the number of rework iterations. The error bars indicate 95% confidence intervals, calculated by repeating each trial 100 times. The dashed black line represents the rework cost of a full checkpoint.



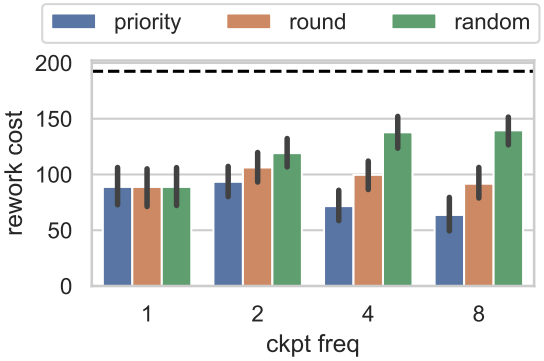
(a) LDA on 20 Newsgroups.



(b) LDA on Reuters.



(c) CNN on MNIST.



(d) CNN (by shard) on MNIST.

Figure 5.10: Prioritized checkpoint experiments (Fig. 5.9) continued.

5.5.5 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 [107] dataset, and LDA on a 12GB subset of the ClueWeb12 dataset [68].

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. 5.11 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.

System Overhead

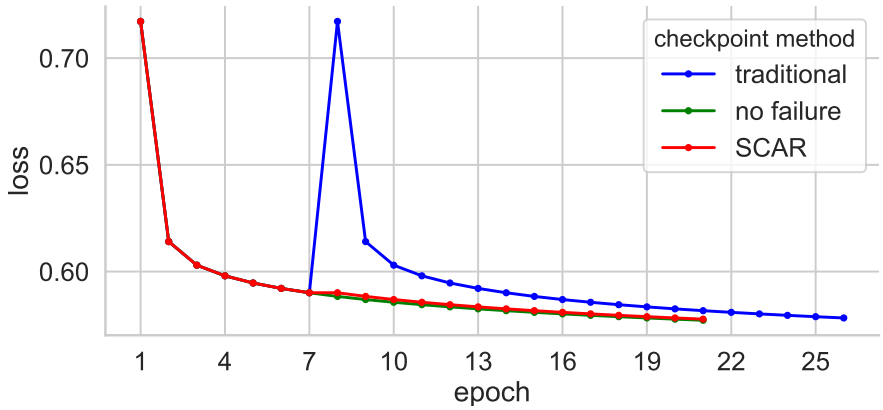
The checkpointing mechanisms of SCAR can be implemented with low performance overhead. In our experiments, we measured an average per-epoch overhead of < 1s for MLR and < 5s for LDA, when compared with traditional checkpointing. Given that the average time spent computing each epoch is ≈ 140s for MLR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

5.6 Related Work

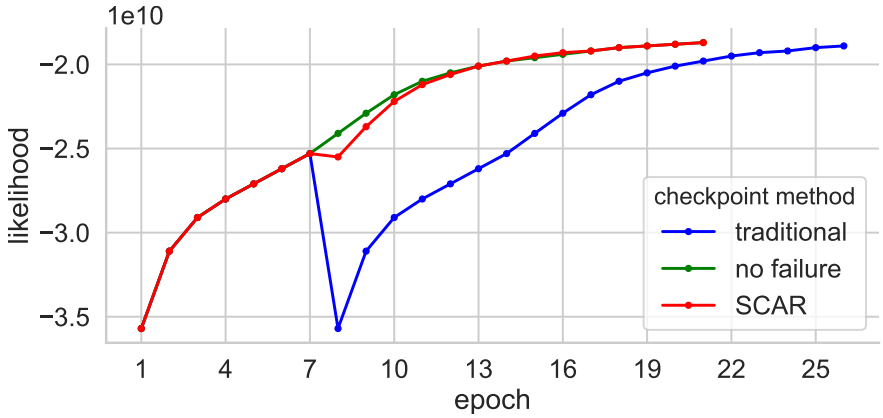
In the optimization literature, optimization with inexact gradients has been extensively studied [see 56, 190, and the references therein]. These works focus on convergence rates and typically assume the errors in the gradients are small. By contrast, our focus is somewhat different, instead considering the case where the perturbations are generic, i.e. they are not restricted to gradient computations and may be significant. Mania et al. [148] and El Gamal and Lai [61] also consider a model similar to (5.2), however, perturbations are only added to the gradients.

A related body of work is distributed training under Byzantine faults [28, 41, 49, 79], 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 [109, 127, 201]. These techniques use coding theory to increase the redundancy of input data or linear computations such as matrix multiplication. The



(a) MLR on Criteo.



(b) LDA on ClueWeb12.

Figure 5.11: 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.

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 [13] offers recovery from periodic checkpoints, while the parameter server of Li et al. [132] offers live replication of parameter values. Proteus [84] 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.

Chapter 6

Conclusion

We conclude this thesis by summarizing our primary contributions and then outlining a few limitations and interesting directions for future research.

6.1 Summary of Contributions

We proposed *co-adaptation* as a key factor for improving elastic training of machine learning and deep learning models. We presented the design, implementation, and evaluation of three systems for ML that improve DL training time in shared GPU clusters by 37-50%, enable elasticity for a diverse set of ML training applications, and reduce the impact of resource failures by 78-95%.

First, Pollux is a DL cluster scheduler that co-adaptively allocates resources, while at the same time tuning each training job to best utilize those resources. We presented a formulation of goodput that combines system throughput and statistical efficiency for distributed DL training. Based on the principle of goodput maximization, Pollux jointly tunes the resource allocations, batch sizes, and learning rates for DL jobs, which can be particularly difficult for users to configure manually. Pollux outperforms and is more fair than recent DL schedulers, even if users can configure their jobs well, and provides even bigger benefits with more realistic user knowledge.

Second, Litz enables elastic execution of diverse ML training applications in clouds and datacenters. We identified three important classes of distributed ML techniques—stateful workers, model scheduling, and relaxed consistency—and designed Litz's programming model to collectively support each of them. By adopting an event-driven API, Litz is able to control the execution of its applications, transparently migrating their state and computation between physical machines. Litz achieves elasticity—the ability to scale out and in based on changing resource availability—without compromising the state-of-the-art efficiency of non-elastic ML systems.

Lastly, we explored the self-correcting behavior of ML and how it can be leveraged to achieve adaptability to faults and failures. We outlined a general approach to design co-adaptive systems for unreliable computing environments 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 when compared to traditional checkpoint-based fault tolerance.

6.2 The Need for Predictability in Machine Learning

In the past decade, research in machine learning has produced a vast number of new methods that may improve training speed, stability, and generalization. For example, new optimization procedures such as Adam [114] and AdaBound [144] accelerate training especially for non-convex and poorly-conditioned problems often encountered in deep learning. Large-batch optimizers such as LARS [221], LEGW [223], and AdaScale [105] improve scalability by enabling training with larger batch sizes. Gradient clipping [232] can mitigate the exploding gradients problem, while gradient quantization [18] may reduce the communication overhead during training with minimal loss of model quality.

Nowadays, practitioners must face a maze of choices they must make in order to optimize the training of their models. Each of the aforementioned methods may work for certain training tasks but not others, and may introduce additional tuning knobs that users must configure. This problem is even more severe for distributed training, for which complex new system-level methods have been proposed, such as pipeline parallelism [95, 157], tensor sharding [128], and communication scheduling [171]. Even if the perfect combination of methods exists to train a model efficiently and without loss of quality, users may never find it!

This thesis highlights an alternative pathway for improvements to ML training via predictability. Even if a certain training method is not the fastest, it can still improve training by being more predictable, which allows co-adaptive systems to take over and automatically optimize for training performance. In Chapter 3, we showed one instance of how improved predictability of the impact of the batch size on DL training can be leveraged by Pollux to improve training performance and fairness in shared compute clusters. Many opportunities still remain, such as more accurately predicting the iterations-to-convergence to further improve job scheduling, predicting the effects of gradient compression on different parts of a DL model so it can be automatically applied during training, and predicting the performance of different parallel execution strategies to make distributed training more automatic.

6.3 Limitations and Future Work

In this section, we highlight a few of the key challenges facing co-adaptation for elastic ML training and suggest future directions for research.

Extensions of Goodput. Goodput (Eqn. 3.1) lets Pollux co-optimize cluster-wide scheduling with per-job training parameters in shared cluster environments. However, goodput is a general notion of DL training performance that incorporates both system throughput and statistical efficiency. Future work may explore other scenarios in which goodput may be applied. For example, recent distributed training systems targeting large models may apply data parallelism, tensor parallelism, and/or pipeline parallelism in different combinations [103, 182, 195]. Since different parallelization strategies are affected differently by different batch sizes, goodput may be used as a target metric for co-optimizing parallelization strategies with the batch size and learning rate.

Furthermore, goodput may be extended to cover other aspects of ML training algorithms that affect statistical efficiency. For example, staleness [48] and non-uniform sampling of mini-

batches [110] may have significant impact on statistical efficiency, which are dimensions not currently covered by Pollux's goodput model.

Predictability of Training. As mentioned in §6.2, a significant portion of research in ML theory has been focused on improving the absolute convergence speed of training. However, as demonstrated by Pollux's use of predictive models, an orthogonal but equally important direction is to improve the *predictability* of training. For example, having a good estimate of how long the model will take to train, or the various effects of adjusting the batch size on the speed of convergence. Future work may be directed towards (1) theory for predictability of existing and commonly-used training techniques, such as the aforementioned staleness and non-uniform sampling of mini-batches, and/or (2) new training strategies which target *predictability* over absolute speed for training a single model. With additional predictability comes an additional avenue for improving the performance and adaptability of ML systems through co-adaptation.

Reproducibility of Training. One challenge that comes with co-adaptation is reproducibility of ML training programs. Under co-adaptation, decisions made by an application may depend on environmental factors during runtime, such as the current cluster contention, or unexpected resource failures, making it more challenging to run the same application twice with the expectation of obtaining exactly the same result. Two possibilities for tackling this challenge, which are not thoroughly explored in this thesis, may be (1) recording the dynamic decisions made by the application during its first execution, and replaying them exactly during the second execution, irregardless of different environmental factors, or (2) development of new training algorithms which are approximately reproducible given different configurations, such as the AdaScale algorithm being approximately reproducible using different batch sizes [105].

Heterogeneous Compute Clusters. The methods discussed in this thesis were developed assuming homogeneous compute resources, i.e. all nodes consisting of the same hardware. However, compute cluster in practice are increasingly heterogeneous, for example due to incremental upgrades across several generations of GPUs. Heterogeneous compute resources introduce additional dimensions that cluster resource schedulers must consider. At the simplest level, a training job may be assigned to a single type of resource and the scheduler must select that type of resource. However, it may make sense for a training job to utilize different types of resources for different distributed workers, especially for recent model-parallel training systems that may have different compute requirements for different workers. Applying co-adaptation to manage the complexities arising from heterogeneous clusters may be a promising direction for future work, for example by extending Pollux's goodput function to consider different resource types.

Appendix A

Appendix for Chapter 3

A.1 Pre-conditioned Gradient Noise Scale

In this appendix, we derive a generalization of gradient noise scale (GNS) [150], which we call the *pre-conditioned gradient noise scale* (PGNS). Our motivation is that regular GNS is not sufficiently general to apply to Adam, AdaGrad and other adaptive gradient algorithms. To derive PGNS, we introduce pre-conditioned stochastic gradient descent (PSGD), a generalization of SGD that multiplies a pre-conditioning matrix (or pre-conditioner) to the gradient, in order to speed-up and stabilize parameter/weight convergence. SGD, Adam and AdaGrad are all special cases of PSGD (when the pre-conditioners are appropriately chosen).

Definition A.1.1. (Pre-conditioned Gradient Noise Scale) *Pre-conditioned gradient noise scale* of PSGD is defined as follows:

$$\varphi_t = \frac{tr(P\Sigma P^T)}{|Pg|^2}$$

where g is the true gradient, P is pre-conditioner, and Σ is the covariance matrix of per-sample stochastic gradient noise.

We first begin by defining mathematical notations that appear in our derivation of the pre-conditioned gradient noise scale. The model is parameterized with θ, and optimized to minimize the loss function L. g and g_{mb} respectively stand for the true gradient and mini-batch gradient calculated for L with respect to θ, and H is the Hessian matrix of L. The (co)variance of stochastic gradient noise per sample is defined as Σ. B is the mini-batch size, and ϵ is the learning rate of PSGD. Lastly, P is pre-conditioner of PSGD. With the above definitions, each parameter update of the model can be written as follows:

$$g_{mb} = g + \frac{\Sigma}{\sqrt{B}}$$

$$\theta \ \leftarrow \ \theta - \epsilon \cdot (Pg_{mb})$$

To derive pre-conditioned gradient noise scale, we mostly follow [150], and analyze the expected change in loss for the weight update $\Delta L = \mathbb{E}[L(\theta) - L(\theta - Pg_{mb})]$ by applying 2nd-

order Taylor expansion:

$$\begin{aligned}
\Delta L &= \epsilon|g^T P g| - \frac{\epsilon^2}{2}\left(g^T P^T H P g + \frac{tr(HP\Sigma P^T)}{B}\right) \\
&= \epsilon|g^T P g| - \frac{\epsilon^2}{2}\left(tr(g^T P^T H P g) + \frac{tr(HP\Sigma P^T)}{B}\right) \\
&\approx \epsilon|g^T P g| - \frac{\epsilon^2}{2}\left(tr(gg^T P^T P) + tr\left(\frac{\Sigma}{B} P^T P\right)\right) \qquad \text{(A.1)} \\
&= \epsilon|g^T P g| - \frac{\epsilon^2}{2} \cdot tr\left(\left(\frac{\Sigma}{B} + gg^T\right) P^T P\right) \qquad \text{(A.2)}
\end{aligned}$$

In Eqn. A.1, we approximate Hessian to be the identity matrix as suggested in [150]. It can be easily observed from Eqn. A.2 that the expected decrease in loss is maximized at:

$$\begin{aligned}
\epsilon_{opt} &= \frac{g^T P g}{tr\left(\left(\frac{\Sigma}{B} + gg^T\right) P^T P\right)} \qquad \text{(A.3)} \\
&= \frac{\frac{g^T P g}{tr(gg^T P^T P)}}{1 + \frac{1}{B} \cdot \frac{tr(\Sigma P^T P)}{tr(gg^T P^T P)}} \\
&= \frac{\frac{g^T P g}{|Pg|^2}}{1 + \frac{1}{B} \cdot \frac{tr(P\Sigma P^T)}{|Pg|^2}} \qquad \text{(A.4)} \\
&= \frac{\epsilon_{max}}{1 + \frac{B_{noise}}{B}} \qquad \text{(A.5)}
\end{aligned}$$

Finally, from Eqn. A.4 & A.5, we can define pre-conditioned gradient noise scale as:

$$B_{noise} = \frac{tr(P\Sigma P^T)}{|Pg|^2} \qquad \text{(A.6)}$$

When $P = I$, PSGD degenerates to SGD, and pre-conditioned gradient noise scale also becomes the simple gradient noise scale $B_{simple} = \frac{tr(\Sigma)}{|g|^2}$ defined in [150]. Moreover, when $P = \mathbb{E}[g_{mb}^2] = \left(\frac{\Sigma}{B} + gg^T\right)^{-0.5}$, PSGD becomes Adam and pre-conditioned gradient noise scale in this case would be

$$\frac{tr\left(\left(\frac{\Sigma}{B} + gg^T\right)^{-1}\Sigma\right)}{\left|\left(\frac{\Sigma}{B} + gg^T\right)^{-0.5} g\right|^2}.$$

A.2 Source Code for the Allocation Search

```
import numpy as np
import pymoo.model.crossover
```

```
import pymoo.model.mutation
import pymoo.model.problem
import pymoo.model.repair
import pymoo.optimize

from pymoo.algorithms.nsga2 import NSGA2
from pymoo.operators.crossover.util import crossover_mask


class Problem(pymoo.model.problem.Problem):
    def __init__(self, jobs, nodes, base_state):
        """
        Multi-objective optimization problem used by PolluxPolicy to determine
        resource allocations and desired cluster size. Optimizes for the best
        performing cluster allocation using only the first N nodes. The cluster
        performance and N are the two objectives being optimized, resulting in
        a set of Pareto-optimal solutions.

        The optimization states are a 3-D array of replica assignments with
        shape (pop_size x num_jobs x num_nodes). The element at k, j, n encodes
        the number of job j replicas assigned to node n, in the kth solution.

        Arguments:
            jobs (list): list of JobInfo objects describing the incomplete jobs
                which need to be scheduled.
            nodes (list): list of NodeInfo objects describing the nodes in the
                cluster, in decreasing order of allocation preference.
            base_state (numpy.array): base optimization state corresponding to
                the current cluster allocations. Shape: (num_jobs x num_nodes).
        """
        assert base_state.shape == (len(jobs), len(nodes))
        self._jobs = jobs
        self._nodes = nodes
        self._base_state = base_state
        # Find which resource types are requested by at least one job.
        rtypes = sorted(set.union(*[set(job.resources) for job in jobs]))
        # Build array of job resources: <num_jobs> x <num_rtypes>. Each entry
        # [j, r] is the amount of resource r requested by a replica of job j.
        self._job_resources = np.zeros((len(jobs), len(rtypes)), np.int64)
        for j, job in enumerate(jobs):
            for r, rtype in enumerate(rtypes):
                self._job_resources[j, r] = job.resources.get(rtype, 0)
        # Build array of node resources: <num_nodes> x <num_rtypes>. Each
        # entry [n, r] is the amount of resource r available on node n.
        self._node_resources = np.zeros((len(nodes), len(rtypes)), np.int64)
        for n, node in enumerate(nodes):
            for r, rtype in enumerate(rtypes):
                self._node_resources[n, r] = node.resources.get(rtype, 0)
        # Calculate dominant per-replica resource shares for each job.
        shares = self._job_resources / np.sum(self._node_resources, axis=0)
        self._dominant_share = np.amax(shares, axis=1)
        # Change base goodput to fair-share goodput.
        fair_replicas = np.ceil(1.0 / self._dominant_share / len(self._jobs))
        fair_nodes = np.ceil(len(nodes) * self._dominant_share)
        for job, num_nodes, num_replicas in zip(jobs, fair_nodes, fair_replicas):
            job.speedup_fn._base_goodput = job.speedup_fn._goodput_fn.optimize(
                num_nodes=num_nodes, num_replicas=num_replicas,
                max_batch_size=job.speedup_fn._max_batch_size,
                atomic_bsz_range=job.speedup_fn._atomic_bsz_range,
                accumulation=job.speedup_fn._accumulation)[0]
        # Upper bound each job: <replicas on node 0> <replicas on node 1> ...
        self._max_replicas = np.zeros(base_state.shape, dtype=np.int)
        for j, job in enumerate(jobs):
            for n, node in enumerate(nodes):
                self._max_replicas[j, n] = min(
                    node.resources[rtype] // job.resources[rtype]
```

```
                for rtype in rtypes if job.resources.get(rtype, 0) > 0)
        super().__init__(n_var=self._base_state.size, n_obj=2, type_var=np.int)

    def get_cluster_utilities(self, states):
        """
        Calculates the cluster utility for each state, defined as the average
        percentage of ideal speedup for each job (ie. speedup / num_replicas),
        weighted by the job's share of the most congested cluster resource.

        Arguments:
            states (numpy.array): a (pop_size x num_jobs x num_nodes) array
                containing the assignments of job replicas to nodes.

        Returns:
            numpy.array: a (pop_size) array containing the utility for each
                state.
        """
        num_replicas = np.sum(states, axis=2)
        speedups = self._get_job_speedups(states)
        # mask (pop_size x num_nodes): indicates which nodes are active.
        mask = np.sum(states, axis=1) > 0
        # total (pop_size x num_rtypes): total amount of cluster resources.
        total = np.sum(np.expand_dims(mask, 2) * self._node_resources, axis=1)
        # alloc (pop_size x num_jobs x num_rtypes):
        #     amount of cluster resources allocated to each job.
        alloc = np.expand_dims(num_replicas, 2) * self._job_resources
        with np.errstate(divide="ignore", invalid="ignore"):
            # shares (pop_size x num_jobs x num_rtypes):
            #     resource shares for each job as a fraction of the cluster.
            shares = np.where(alloc, alloc / np.expand_dims(total, 1), 0.0)
            # utilities (pop_size x num_jobs):
            #     utilities for each job as a fraction of ideal scalability.
            utilities = np.where(num_replicas, speedups / num_replicas, 0.0)
        # Weighted average across all jobs for each rtype.
        utilities = np.sum(np.expand_dims(utilities, 2) * shares, axis=1)
        # Return the utilities for the best utilized rtypes.
        return np.amax(utilities, axis=1)  # Shape: (pop_size).

    def _get_job_speedups(self, states):
        speedup = []
        num_nodes = np.count_nonzero(states, axis=2)
        num_replicas = np.sum(states, axis=2)
        for idx, job in enumerate(self._jobs):
            speedup.append(job.speedup_fn(
                num_nodes[:, idx], num_replicas[:, idx]))
        return np.stack(speedup, axis=1).astype(np.float)

    def _get_cluster_sizes(self, states):
        return np.full(len(states), len(self._nodes))
        #sizes = np.arange(len(self._nodes)) + 1
        #return np.amax(np.where(np.any(states, axis=-2), sizes, 0), axis=-1)

    def _evaluate(self, states, out, *args, **kwargs):
        states = states.reshape(states.shape[0], *self._base_state.shape)
        speedups = self._get_job_speedups(states)
        # Scale the speedup of each job so that a dominant resource share
        # equivalent to a single node results in a speedup of 1.
        scaled_speedups = speedups * self._dominant_share * len(self._nodes)
        # Penalize job restarts.
        num_restarts = np.array([job.num_restarts for job in self._jobs])
        age = np.array([job.age for job in self._jobs])
        delay = 30
        factor = np.maximum(age - num_restarts * delay, 0.0) / (age + delay)
        restart = np.any(states != self._base_state, axis=2)
        scaled_speedups *= np.where(restart, factor, 1)
        p = -1  # Exponent used in power mean. More negative = more fair.
```

```
        if p == 0:
            # Geometric mean
            mean = np.exp(np.sum(np.log(np.maximum(scaled_speedups, 1e-3)),
                                 axis=1) / states.shape[1])
        else:
            mean = (np.sum((scaled_speedups + 1e-3) ** p, axis=1)
                    / states.shape[1]) ** (1.0 / p)
        out["F"] = np.column_stack([-mean,
                                    -self.get_cluster_utilities(states)])

    def _crossover(self, states, **kwargs):
        states = states.reshape(*states.shape[:2], *self._base_state.shape)
        n_parents, n_matings, n_jobs, n_nodes = states.shape
        # Single-point crossover over jobs for all parent states.
        points = np.random.randint(n_jobs, size=(n_matings, 1))
        result = crossover_mask(states, np.arange(n_jobs) < points)
        # Set cluster sizes uniformly at random between each pair of parents.
        min_nodes, max_nodes = np.sort(self._get_cluster_sizes(states), axis=0)
        num_nodes = np.random.randint(np.iinfo(np.int16).max,
                                      size=(n_parents, n_matings))
        num_nodes = min_nodes + num_nodes % (max_nodes - min_nodes + 1)
        mask = np.arange(n_nodes) >= np.expand_dims(num_nodes, (2, 3))
        result[np.broadcast_to(mask, result.shape)] = 0
        return result.reshape(n_parents, n_matings, -1)

    def _mutation(self, states, **kwargs):
        states = states.reshape(states.shape[0], *self._base_state.shape)
        # (1) Randomly reset back to base state.
        mask = np.random.random(states.shape[:2]) < 0.1
        states = np.where(np.expand_dims(mask, 2), self._base_state, states)
        # (2) Randomly zero out some elements.
        prob = np.where(np.random.random(states.shape[:2]) < 0.1, 0.1, 0.0)
        states[np.random.random(states.shape) < np.expand_dims(prob, 2)] = 0
        # (3) Randomly increase some elements.
        used_resources = (np.expand_dims(self._job_resources, 1) *
                          np.expand_dims(states, -1)).sum(axis=1)
        free_resources = self._node_resources - used_resources
        mask1 = np.all(np.expand_dims(self._job_resources, 1) <=
                       np.expand_dims(free_resources, 1), axis=-1)
        prob1 = 1.0 * mask1 / np.maximum(mask1.sum(axis=1, keepdims=True), 1.0)
        mask2 = np.logical_and(states, mask1)
        prob2 = 1.0 * mask2 / np.maximum(mask2.sum(axis=1, keepdims=True), 1.0)
        m = np.random.random(states.shape) < prob1 + prob2 - prob1 * prob2
        r = np.random.randint(states, self._max_replicas + 1)
        states[m] = r[m]
        return states.reshape(states.shape[0], -1)

    def _repair(self, pop, **kwargs):
        states = pop.get("X")
        states = states.reshape(states.shape[0], *self._base_state.shape)
        # Copy previous allocations for pinned jobs
        states[:, self._pinned_indices] = \
            self._base_state[self._pinned_indices, :]
        # Enforce at most one distributed job per node. Exclude all
        # nonpreemptible jobs.
        distributed = np.count_nonzero(states, axis=2) > 1
        mask = states * np.expand_dims(distributed, axis=-1) > 0
        mask = mask.cumsum(axis=1) > 1
        states[mask] = 0
        # Enforce no more than max replicas per job.
        # max_replicas: (num_jobs x 1)
        max_replicas = np.array([[j.max_replicas] for j in self._jobs])
        shuffle = np.argsort(np.random.random(states.shape), axis=2)
        states = np.take_along_axis(states, shuffle, axis=2)  # Shuffle nodes.
        states = np.minimum(np.cumsum(states, axis=2), max_replicas)
        states = np.diff(states, axis=2, prepend=0)
```

```
        max_nodes = 16
        mask = np.minimum(np.cumsum(states > 0, axis=2), max_nodes)
        mask = np.diff(mask, axis=2, prepend=0)
        states[np.logical_not(mask)] = 0
        inverse = np.argsort(shuffle, axis=2)  # Undo shuffle nodes.
        states = np.take_along_axis(states, inverse, axis=2)
        # Enforce node resource limits.
        # job_resources: (num_jobs x num_nodes x num_rtypes)
        job_resources = np.expand_dims(self._job_resources, 1)
        states = np.expand_dims(states, -1) * job_resources
        states = np.minimum(np.cumsum(states, axis=1), self._node_resources)
        states = np.diff(states, axis=1, prepend=0)
        with np.errstate(divide="ignore", invalid="ignore"):
            states = np.amin(np.floor_divide(states, job_resources),
                             where=job_resources > 0, initial=99, axis=-1)
        # Only choose solutions which have at least min_replicas allocations
        min_replicas = np.array([j.min_replicas for j in self._jobs])
        mask = np.sum(states, axis=-1) < min_replicas
        states[mask] = 0
        return pop.new("X", states.reshape(states.shape[0], -1))


class Crossover(pymoo.model.crossover.Crossover):
    def __init__(self):
        super().__init__(n_parents=2, n_offsprings=2)

    def _do(self, problem, states, **kwargs):
        return problem._crossover(states, **kwargs)


class Mutation(pymoo.model.mutation.Mutation):
    def _do(self, problem, states, **kwargs):
        return problem._mutation(states, **kwargs)


class Repair(pymoo.model.repair.Repair):
    def _do(self, problem, pop, **kwargs):
        return problem._repair(pop, **kwargs)
```

Appendix B

Appendix for Chapter 5

B.1 Discussion

In this appendix, we discuss some special cases and extensions of interest, including nonconvex models, infinite perturbations (e.g. $T = \infty$), and SGD.

B.1.1 Analysis for Infinite T

Suppose $\mathbb{E}\|\delta_k\| \leq \Delta$ for each k. For intuition, note that Lemma B.2.1 implies that for any k,

$$\begin{aligned}
\mathbb{E}\|y^{(k+1)} - w^*\| &\leq c^{k+1}\Big[\|w^{(0)} - w^*\| + \sum_{\ell=0}^{k} c^{-\ell}\Delta\Big] \\
&= c^{k+1}\Big[\|w^{(0)} - w^*\| + \Delta\frac{1 - c^{-(k+1)}}{1 - c^{-1}}\Big] \\
&= c^{k+1}\|w^{(0)} - w^*\| + \Delta\frac{c - c^{k+2}}{1 - c} \\
&\overset{k\to\infty}{\longrightarrow} \frac{c}{1 - c}\Delta.
\end{aligned} \tag{B.1}$$

Evidently, there is an irreducible, positive error if we are subjected to faults in every single iteration.

Thus, the best we can hope for is convergence to within some tolerance $\varepsilon > (c/(1-c))\Delta$. Rearranging and solving for k in (B.1) as in the proof of Theorem 5.3.1, we deduce that $\mathbb{E}\|y^{(k+1)} - w^*\| < \varepsilon$ as long as

$$k > \frac{\log\left(\frac{\|w^{(0)} - w^*\| - \frac{c}{1-c}\Delta}{\varepsilon - \frac{c}{1-c}\Delta}\right)}{\log(1/c)}.$$

The resulting iteration cost bound is (cf. (5.6)):

$$\pi(\delta_k,\ \varepsilon) \leq \frac{\log\left(\frac{1 - \frac{\frac{c}{1-c}\Delta}{\|w^{(0)} - w^*\|}}{1 - \frac{\frac{c}{1-c}\Delta}{\varepsilon}}\right)}{\log(1/c)}. \tag{B.2}$$

This bound is only informative if $\|w^{(0)} - w^*\| > (c/(1-c))\Delta$ and $\varepsilon > (c/(1-c))\Delta$.

B.1.2 Stochastic gradient descent

Assume the objective function ℓ is strongly convex. In order to derive upper bounds on the iteration cost for SGD, we start from following general recursion, which is standard from the literature [160, 181]:

$$\mathbb{E}\|w^{(k+1)} - w^*\|^2 \leq (1-\alpha_k)\mathbb{E}\|w^{(k)} - w^*\|^2 + \alpha_k^2 G^2, \tag{B.3}$$

where $\alpha_k \to 0$ is a sequence that depends on ℓ and the step size, and G is an upper bound on the expected norm of the stochastic gradients. Comparing (B.3) to (B.5), the only difference is that instead of a constant $c < 1$, we have a sequence $1 - \alpha_k \to 1$. Thus, instead of decaying at the geometric rate c^k, the iterates of SGD converge at a slower rate $(1-\alpha_1)\cdots(1-\alpha_k)$.

Define $a_k := (1-\alpha_1)\cdots(1-\alpha_k)$. Under the assumptions of Theorem 5.3.1, we have the following analogue of (B.8):

$$\mathbb{E}\|y^{(k)} - w^*\| \leq a_k\Big[\|w^{(0)} - w^*\| + \sum_{\ell=0}^{T} a_\ell^{-1}\big(\mathbb{E}\|\delta_\ell\| + \alpha_\ell^2 G^2\big)\Big] < \varepsilon.$$

This yields an implicit formula for k, which can be used to upper bound the iteration cost for SGD. For example, a popular choice of α_k is $\alpha_k \propto 1/k$, in which case $a_k \propto 1/k$ (this follows from an induction argument), and solving for k yields the desired upper bound.

B.2 Proofs

B.2.1 Proof of Theorem 5.3.1

We start with the following useful lemma:

Lemma B.2.1. *Assuming* (5.5)*, we have for any* k

$$\mathbb{E}\|y^{(k+1)} - w^*\| \leq c^{k+1}\Big[\|w^{(0)} - w^*\| + \sum_{\ell=0}^{k} c^{-\ell}\mathbb{E}\|\delta_\ell\|\Big]. \tag{B.4}$$

Proof. For any $k > 0$ we have

$$\begin{aligned}
\mathbb{E}\|y^{(k+1)} - w^*\| &= \mathbb{E}\|f(\widetilde{y}^{(k)}) - w^*\| \\
&\leq c\mathbb{E}\|\widetilde{y}^{(k)} - w^*\| \\
&= c\mathbb{E}\|y^{(k)} + \delta_k - w^*\| \\
&\leq c\big[\mathbb{E}\|y^{(k)} - w^*\| + \mathbb{E}\|\delta_k\|\big],
\end{aligned} \tag{B.5}$$

where we have invoked (5.5). Iterating this inequality, we obtain:

$$
\begin{aligned}
c\big[\mathbb{E}\|y^{(k)} - w^*\| + \mathbb{E}\|\delta_k\|\big] & \qquad \text{(B.6)} \\
&\leq c^2\mathbb{E}\|y^{(k-1)} - w^*\| + c^2\mathbb{E}\|\delta_{k-1}\| + c\mathbb{E}\|\delta_k\| \\
&\;\;\vdots \\
&\leq c^{k+1}\mathbb{E}\|y^{(0)} - w^*\| + \sum_{i=0}^{k} c^{i+1}\mathbb{E}\|\delta_{k-i}\| \\
&= c^{k+1}\|w^{(0)} - w^*\| + \sum_{\ell=0}^{k} c^{k-\ell+1}\mathbb{E}\|\delta_\ell\|. \qquad \text{(B.7)}
\end{aligned}
$$

In the last step we simply re-indexed the summation and use $y^{(0)} = w^{(0)}$. Combining (B.5) and (B.7) yields the desired bound. □

Proof of Theorem 5.3.1. By Lemma B.2.1, we have for any $k > T$,

$$
\begin{aligned}
\mathbb{E}\|y^{(k)} - w^*\| \leq c^k\Big[\|w^{(0)} - w^*\| + \sum_{\ell=0}^{T} c^{-\ell}\mathbb{E}\|\delta_\ell\|\Big] < \varepsilon & \qquad \text{(B.8)} \\
\iff \frac{1}{\varepsilon}\Big[\|w^{(0)} - w^*\| + \Delta_T\Big] < c^{-k} & \qquad \text{(B.9)}
\end{aligned}
$$

Re-arranging, we deduce that $\mathbb{E}\|y^{(k)} - w^*\| < \varepsilon$ if

$$
k > \frac{\log\Big(\frac{1}{\varepsilon}\Big[\|w^{(0)} - w^*\| + \Delta_T\Big]\Big)}{\log(1/c)} \geq \kappa(y^{(k)},\ \varepsilon).
$$

It is easy to check (e.g. take $\delta_k = 0$ in the previous derivation) that $\kappa(w^{(k)},\ \varepsilon) = \log\big(\frac{1}{\varepsilon}\|w^{(0)} - w^*\|\big)/\log(1/c)$ is a bound on the number of iterations required for the unperturbed sequence $w^{(k)}$ to reach ε-optimality. Thus, the iteration cost is given by

$$
\begin{aligned}
\pi(\delta_k,\ \varepsilon) &= \kappa(y^{(k)},\ \varepsilon) - \kappa(w^{(k)},\ \varepsilon) \\
&\leq \frac{\log\Big(\frac{1}{\varepsilon}\Big[\|w^{(0)} - w^*\| + \Delta_T\Big]\Big) - \log\big(\frac{1}{\varepsilon}\|w^{(0)} - w^*\|\big)}{\log(1/c)} \\
&= \frac{\log\Big(1 + \frac{\Delta_T}{\|w^{(0)} - w^*\|}\Big)}{\log(1/c)},
\end{aligned}
$$

as claimed. □

B.2.2 Proof of Theorem 5.4.1

Let $z = w^{(C)}$ be the checkpoint of the model parameters saved at iteration C, and let S be the subset of model parameters lost during a failure at iteration T. Then

$$
||\delta|| = ||z - w^{(T)}||
$$

is the perturbation due to full recovery, and

$$||\delta'|| = ||z_S - w_S^{(T)}||$$

is the perturbation due to partial recovery, since $w_{S^c}^{(T)}$ does not change due to failure, where S^c is the complement set of S. Then we have

$$\begin{aligned}
||\delta'||^2 &= ||z_S - w_S^{(T)}||^2 \\
&\leq ||z_S - w_S^{(T)}||^2 + ||z_{S^c} - w_{S^c}^{(T)}||^2 \\
&\quad + (z_S - w_S^{(T)}) \cdot (z_{S^c} - w_{S^c}^{(T)}) \\
&\leq ||(z_S - w_S^{(T)}) + (z_{S^c} - w_{S^c}^{(T)})||^2 \\
&= ||z - w^{(T)}||^2 = ||\delta||^2
\end{aligned}$$

Thus $\|\delta'\| \leq \|\delta\|$, as claimed.

B.2.3 Proof of Theorem 5.4.2

Let $z = w^{(C)}$ be the checkpoint of the model parameters saved at iteration C, and let S be the subset (chosen uniformly at random) of model parameters lost during a failure at iteration T. Then

$$\begin{aligned}
\mathbb{E}||\delta'||^2 &= \mathbb{E}||z_S - w_S^{(T)}||^2 \\
&= \mathbb{E}\left[(z_S - w_S^{(T)}) \cdot (z_S - w_S^{(T)})\right] \\
&= \sum_i \mathbb{E}\left[(z_S - w_S^{(T)})_i^2\right] \\
&= \sum_i \mathbb{E}\left[[i \in S](z_i - w_i^{(T)})^2\right] \\
&= \sum_i P(i \in S)(z_i - w_i^{(T)})^2 \\
&= \sum_i p(z_i - w_i^{(T)})^2 \\
&= p||z - w^{(T)}||^2 = p||\delta||^2
\end{aligned}$$

Thus $\mathbb{E}||\delta'||^2 = p||\delta||^2$, as claimed.

Bibliography

[1] Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/. Cited on page 75.

[2] Amazon EBS Volume Types. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSVolumeTypes.html. Cited on page 75.

[3] Introduction to katib — kubeflow. https://www.kubeflow.org/docs/components/hyperparameter-tuning/overview/. Accessed: 2020-05-18. Cited on page 46.

[4] Configuring Transparent Huge Pages. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/performance_tuning_guide/sect-red_hat_enterprise_linux-performance_tuning_guide-configuring_transparent_huge_pages. Cited on page 72.

[5] Using mlock to Avoid Page I/O. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_for_real_time/7/html/reference_guide/using_mlock_to_avoid_page_io. Cited on page 72.

[6] Swap Files and Partitions. https://en.wikipedia.org/wiki/Paging#Swap_files_and_partitions. Cited on page 72.

[7] ulimit. https://ss64.com/bash/ulimit.html. Cited on page 74.

[8] Boost Context. www.boost.org/doc/libs/1_63_0/libs/context/, 2016. Cited on page 53, 57.

[9] Boost Serialization. http://www.boost.org/doc/libs/1_64_0/libs/serialization/, 2016. Cited on page 55.

[10] etcd. http://coreos.com/etcd/, 2016. Cited on page 55.

[11] Apache Hadoop. http://hadoop.apache.org/, 2016. Cited on page 48.

[12] ZeroMQ. http://zeromq.org, 2016. Cited on page 53.

[13] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, 2016. USENIX Association. ISBN 978-1-931971-33-1. Cited on page 58, 66, 78, 85, 99.

[14] Alekh Agarwal and John C Duchi. Distributed delayed stochastic optimization. In *Advances in Neural Information Processing Systems*, pages 873–881, 2011. Cited on page 50.

[15] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, Mar 1995. ISSN 1432-0452. doi: 10.1007/BF01784241. URL `https://doi.org/10.1007/BF01784241`. Cited on page 53.

[16] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola. Scalable inference in latent variable models. In *WSDM '12: Proceedings of the fifth ACM international conference on Web search and data mining*, pages 123–132, New York, NY, USA, 2012. ACM. Cited on page 6.

[17] Sungjin Ahn, Babak Shahbaba, Max Welling, et al. Distributed stochastic gradient mcmc. In *ICML*, pages 1044–1052, 2014. Cited on page 50.

[18] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding, 2016. Cited on page 102.

[19] Dario Amodei. Ai and compute, Jun 2021. URL `https://openai.com/blog/ai-and-compute/`. Cited on page 1.

[20] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, Jie Chen, Jingdong Chen, Zhijie Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Ke Ding, Niandong Du, Erich Elsen, Jesse Engel, Weiwei Fang, Linxi Fan, Christopher Fougner, Liang Gao, Caixia Gong, Awni Hannun, Tony Han, Lappi Vaino Johannes, Bing Jiang, Cai Ju, Billy Jun, Patrick LeGresley, Libby Lin, Junjie Liu, Yang Liu, Weigao Li, Xiangang Li, Dongpeng Ma, Sharan Narang, Andrew Ng, Sherjil Ozair, Yiping Peng, Ryan Prenger, Sheng Qian, Zongfeng Quan, Jonathan Raiman, Vinay Rao, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Kavya Srinet, Anuroop Sriram, Haiyuan Tang, Liliang Tang, Chong Wang, Jidong Wang, Kaifu Wang, Yi Wang, Zhijian Wang, Zhiqian Wang, Shuang Wu, Likai Wei, Bo Xiao, Wen Xie, Yan Xie, Dani Yogatama, Bin Yuan, Jun Zhan, and Zhenyao Zhu. Deep speech 2: End-to-end speech recognition in english and mandarin. In *Proceedings of the 33rd International Conference on International*

Conference on Machine Learning - Volume 48, ICML'16, page 173–182. JMLR.org, 2016. Cited on page 36.

[21] Hédy Attouch, Jérôme Bolte, Patrick Redont, and Antoine Soubeyran. 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. Cited on page 83, 84.

[22] Steven C Bagley, Halbert White, and Beatrice A Golomb. Logistic regression in the medical literature:: Standards for use and reporting, with particular attention to one medical domain. *Journal of Clinical Epidemiology*, 54(10):979–985, 2001. ISSN 0895-4356. doi: https://doi.org/10.1016/S0895-4356(01)00372-9. URL https://www.sciencedirect.com/science/article/pii/S0895435601003729. Cited on page 9.

[23] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *Proc. VLDB Endow.*, 5(8):776–787, April 2012. ISSN 2150-8097. doi: 10.14778/2212351.2212359. URL http://dx.doi.org/10.14778/2212351.2212359. Cited on page 50.

[24] Lukas Balles, Javier Romero, and Philipp Hennig. Coupling adaptive batch sizes with learning rates. *CoRR*, abs/1612.05086, 2016. URL http://arxiv.org/abs/1612.05086. Cited on page 46.

[25] Nadav Ben-Haim, Boris Babenko, and Serge Belongie. Improvingweb-based image search via content based clustering. In *Computer Vision and Pattern Recognition Workshop, 2006. CVPRW'06. Conference on*, pages 106–106. IEEE, 2006. Cited on page 68.

[26] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975. ISSN 0001-0782. doi: 10.1145/361002.361007. URL http://doi.acm.org/10.1145/361002.361007. Cited on page 69.

[27] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011. Cited on page 1, 45, 46.

[28] Peva Blanchard, Rachid Guerraoui, Julien Stainer, et al. Machine learning with adversaries: Byzantine tolerant gradient descent. In *Advances in Neural Information Processing Systems*, pages 118–128, 2017. Cited on page 97.

[29] J. Blank and K. Deb. pymoo: Multi-objective optimization in python. *IEEE Access*, pages 1–1, 2020. Cited on page 34.

[30] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, March 2003. ISSN 1532-4435. URL http://dl.acm.org/citation.cfm?id=944919.944937. Cited on page 9.

[31] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003. Cited on page 71.

[32] O. Borů vka. Über ein Minimalproblem. Práce moravské přirodovědecké společnosti 3 (1926), 37-58 (1926)., 1926. Cited on page 70.

[33] Léon Bottou, Frank E Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *arXiv preprint arXiv:1606.04838*, 2016. Cited on page 83.

[34] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf. Cited on page 27.

[35] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *ACM Queue*, 14:70–93, 2016. URL http://queue.acm.org/detail.cfm?id=2898444. Cited on page 32.

[36] Ricardo J. G. B. Campello, Davoud Moulavi, and Joerg Sander. *Density-Based Clustering Based on Hierarchical Density Estimates*, pages 160–172. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-37456-2. doi: 10.1007/978-3-642-37456-2_14. URL https://doi.org/10.1007/978-3-642-37456-2_14. Cited on page 68.

[37] J. Canny and Huasha Zhao. Butterfly mixing: Accelerating incremental-update algorithms on clusters. In *SDM*, 2013. Cited on page 7, 27.

[38] Pei Cao and Zhe Wang. Efficient top-k query calculation in distributed networks. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, pages 206–215, New York, NY, USA, 2004. ACM. ISBN 1-58113-802-4. doi: 10.1145/1011767.1011798. Cited on page 88.

[39] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015. URL http://arxiv.org/abs/1512.01274. Cited on page 16.

[40] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *CoRR*, abs/1604.06174, 2016. URL http://arxiv.org/abs/1604.06174. Cited on page 27.

[41] Yudong Chen, Lili Su, and Jiaming Xu. Distributed statistical machine learning in adversarial settings: Byzantine gradient descent. *arXiv preprint arXiv:1705.05491*, 2017. Cited on page 97.

[42] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 571–582, Broomfield, CO, October 2014. USENIX Association. ISBN 978-1-931971-16-4. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chilimbi. Cited on page 48.

[43] James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Gregory R. Ganger, Garth Gibson, Kimberly Keeton, and Eric Xing. Solving the straggler problem with bounded staleness. In *Presented as part of the 14th Workshop on Hot Topics in Operating Systems*, Berkeley, CA, 2013. USENIX. Cited on page 9, 50, 77, 78.

[44] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Low precision arithmetic for deep learning. *CoRR*, abs/1412.7024, 2014. Cited on page 78.

[45] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. Exploiting bounded staleness to speed up big data analytics. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 37–48, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-10-2. Cited on page 78.

[46] Henggang Cui, Hao Zhang, Gregory R. Ganger, Phillip B. Gibbons, and Eric P. Xing. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342407. doi: 10.1145/2901318.2901323. URL https://doi.org/10.1145/2901318.2901323. Cited on page 27.

[47] Wei Dai, Abhimanu Kumar, Jinliang Wei, Qirong Ho, Garth Gibson, and Eric P. Xing. High-performance distributed ml at scale through parameter server consistency models. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, pages 79–87. AAAI Press, 2015. ISBN 0-262-51129-0. URL http://dl.acm.org/citation.cfm?id=2887007.2887019. Cited on page 48, 50, 80.

[48] Wei Dai, Yi Zhou, Nanqing Dong, Hao Zhang, and Eric P Xing. Toward understanding the impact of staleness in distributed machine learning. *arXiv preprint arXiv:1810.03264*, 2018. Cited on page 9, 102.

[49] Georgios Damaskinos, El Mahdi El Mhamdi, Rachid Guerraoui, Rhicheek Patra, and Mahsa Taziki. Asynchronous Byzantine machine learning (the case of SGD). In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference*

on Machine Learning, volume 80 of *Proceedings of Machine Learning Research*, pages 1145–1154, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR. Cited on page 97.

[50] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008. ISSN 0001-0782. doi: 10.1145/1327452.1327492. URL http://doi.acm.org/10.1145/1327452.1327492. Cited on page 1.

[51] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. Large scale distributed deep networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1223–1231. Curran Associates, Inc., 2012. URL http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.pdf. Cited on page 6, 9, 16, 51, 73, 78.

[52] Aaron Defazio, Francis Bach, and Simon Lacoste-Julien. Saga: A fast incremental gradient method with support for non-strongly convex composite objectives. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'14, page 1646–1654, Cambridge, MA, USA, 2014. MIT Press. Cited on page 9.

[53] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009. Cited on page xiii, 8, 36.

[54] Aditya Devarakonda, Maxim Naumov, and Michael Garland. Adabatch: Adaptive batch sizes for training deep neural networks. *CoRR*, abs/1712.02029, 2017. URL http://arxiv.org/abs/1712.02029. Cited on page 46.

[55] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019. Cited on page 36.

[56] Olivier Devolder, François Glineur, and Yurii Nesterov. First-order methods of smooth convex optimization with inexact oracle. *Mathematical Programming*, 146(1-2):37–75, 2014. Cited on page 97.

[57] Dua Dheeru and Efi Karra Taniskidou. UCI machine learning repository, 2017. Cited on page 89.

[58] Wei Dong, Charikar Moses, and Kai Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 577–586, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0632-4. doi: 10.1145/1963405.1963487. URL http://doi.acm.org/10.1145/1963405.1963487. Cited on page 69.

[59] Simon S Du, Chi Jin, Jason D Lee, Michael I Jordan, Barnabas Poczos, and Aarti Singh. Gradient descent can take exponential time to escape saddle points. *arXiv preprint arXiv:1705.10412*, 2017. Cited on page 81.

[60] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(61): 2121–2159, 2011. URL `http://jmlr.org/papers/v12/duchi11a.html`. Cited on page 7.

[61] Mostafa El Gamal and Lifeng Lai. On randomized distributed coordinate descent with quantized updates. In *Information Sciences and Systems (CISS), 2017 51st Annual Conference on*, pages 1–5. IEEE, 2017. Cited on page 97.

[62] Nosayba El-Sayed, Hongyu Zhu, and Bianca Schroeder. Learning from failure across multiple clusters: A trace-driven approach to understanding, predicting, and mitigating job terminations. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*, pages 1333–1344. IEEE, 2017. Cited on page 71.

[63] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The pascal visual object classes (voc) challenge. *International Journal of Computer Vision*, 88(2): 303–338, June 2010. Cited on page 36.

[64] Nuwan Ferdinand, Haider Al-Lawati, Stark Draper, and Matthew Nokleby. ANYTIME MINIBATCH: EXPLOITING STRAGGLERS IN ONLINE DISTRIBUTED OPTIMIZATION. In *International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=rkzDIiA5YQ`. Cited on page 46.

[65] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in neural information processing systems*, pages 2962–2970, 2015. Cited on page 46.

[66] E. B. Fowlkes and C. L. Mallows. A method for comparing two hierarchical clusterings. *Journal of the American Statistical Association*, 78(383):553–569, 1983. ISSN 01621459. URL `http://www.jstor.org/stable/2288117`. Cited on page 70.

[67] Preparata Franco and Michael Ian Preparata Shamos. Computational geometry: an introduction. 1985. Cited on page 68.

[68] Evgeniy Gabrilovich, Michael Ringgaard, and Amarnag Subramanya. Facc1: Freebase annotation of clueweb corpora, version 1 (release date 2013-06-26, format version 1, correction level 0). http://lemurproject.org/clueweb12/, 2013. Cited on page 58, 97.

[69] Rong Ge, Furong Huang, Chi Jin, and Yang Yuan. Escaping from saddle points — online stochastic gradient for tensor decomposition. In Peter Grünwald, Elad Hazan, and Satyen Kale, editors, *Proceedings of The 28th Conference on Learning Theory*, volume 40 of *Proceedings of Machine Learning Research*, pages 797–842, Paris, France, 03–06 Jul 2015. PMLR. Cited on page 81.

[70] Rainer Gemulla, Erik Nijkamp, Peter J. Haas, and Yannis Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pages 69–77, New York, NY, USA, 2011. ACM. doi: 10.1145/2020408.2020426. Cited on page 1, 48, 49, 51.

[71] Sanjay Ghemawat and Paul Menage. TCMalloc : Thread-Caching Malloc. `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`, 2017. Cited on page 64.

[72] Ken Goldberg, Theresa Roeder, Dhruv Gupta, and Chris Perkins. Eigentaste: A constant time collaborative filtering algorithm. *Inf. Retr.*, 4(2):133–151, July 2001. ISSN 1386-4564. doi: 10.1023/A:1011419012209. Cited on page 89.

[73] Noah Golmant, Nikita Vemuri, Zhewei Yao, Vladimir Feinberg, Amir Gholami, Kai Rothauge, Michael W. Mahoney, and Joseph Gonzalez. On the computational inefficiency of large batch sizes for stochastic gradient descent. *CoRR*, abs/1811.12941, 2018. URL `http://arxiv.org/abs/1811.12941`. Cited on page 17.

[74] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D. Sculley. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, page 1487–1495, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450348874. doi: 10.1145/3097983.3098043. URL `https://doi.org/10.1145/3097983.3098043`. Cited on page 46.

[75] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`. Cited on page 66.

[76] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: training imagenet in 1 hour. *CoRR*, abs/1706.02677, 2017. URL `http://arxiv.org/abs/1706.02677`. Cited on page 17, 22, 23.

[77] Thomas L. Griffiths and Mark Steyvers. Finding scientific topics. *PNAS*, 101(suppl. 1): 5228–5235, 2004. Cited on page 10.

[78] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, Boston, MA, February 2019. USENIX Association. ISBN 978-1-931971-49-2. URL `https://www.usenix.org/conference/nsdi19/presentation/gu`. Cited on page 15, 17, 37.

[79] Rachid Guerraoui, Sébastien Rouault, et al. The hidden vulnerability of distributed learning in byzantium. In *International Conference on Machine Learning*, pages 3518–3527, 2018. Cited on page 97.

[80] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. *CoRR*, abs/1502.02551, 2015. Cited on page 78.

[81] Aaron Harlap, Henggang Cui, Wei Dai, Jinliang Wei, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. Addressing the straggler problem for iterative convergent parallel ml. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 98–111, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4525-5. doi: 10.1145/2987550.2987554. URL http://doi.acm.org/10.1145/2987550.2987554. Cited on page 50.

[82] Aaron Harlap, Henggang Cui, Wei Dai, Jinliang Wei, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. Addressing the straggler problem for iterative convergent parallel ml. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 98–111, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4525-5. doi: 10.1145/2987550.2987554. Cited on page 77.

[83] Aaron Harlap, Alexey Tumanov, Andrew Chung, Greg Ganger, and Phil Gibbons. Proteus: agile ml elasticity through tiered reliability in dynamic resource markets. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '17, New York, NY, USA, 2017. ACM. Cited on page 2, 67.

[84] Aaron Harlap, Alexey Tumanov, Andrew Chung, Gregory R. Ganger, and Phillip B. Gibbons. Proteus: Agile ml elasticity through tiered reliability in dynamic resource markets. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 589–604, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4938-3. doi: 10.1145/3064176.3064182. Cited on page 85, 99.

[85] F. Maxwell Harper and Joseph A. Konstan. The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, 5(4), December 2015. ISSN 2160-6455. doi: 10.1145/2827872. URL https://doi.org/10.1145/2827872. Cited on page 36.

[86] F. Maxwell Harper and Joseph A. Konstan. 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. Cited on page 89.

[87] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016. Cited on page 36.

[88] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, page 173–182, Republic and Canton of Geneva, CHE, 2017. International World Wide Web Conferences Steering Committee. ISBN 9781450349130. doi: 10.1145/3038912.3052569. URL https://doi.org/10.1145/3038912.3052569. Cited on page 36.

[89] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2002. ISBN 0898715210. Cited on page 80.

[90] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 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, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1972457.1972488. Cited on page 77.

[91] Qirong Ho, James Cipar, Henggang Cui, Jin Kyu Kim, Seunghak Lee, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. 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, pages 1223–1231, USA, 2013. Curran Associates Inc. Cited on page xiii, 6, 9, 10, 16, 48, 49, 50, 73, 78, 80, 85.

[92] Mingyi Hong. A distributed, asynchronous and incremental algorithm for nonconvex optimization: An admm based approach. *arXiv preprint arXiv:1412.6058*, 2014. Cited on page 50.

[93] Botong Huang, Matthias Boehm, Yuanyuan Tian, Berthold Reinwald, Shirish Tatikonda, and Frederick R. Reiss. Resource elasticity for large-scale machine learning. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 137–152, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450327589. doi: 10.1145/2723372.2749432. URL https://doi.org/10.1145/2723372.2749432. Cited on page 2.

[94] Chien-Chin Huang, Gu Jin, and Jinyang Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1341–1355, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371025. doi: 10.1145/3373376.3378530. URL https://doi.org/10.1145/3373376.3378530. Cited on page 27.

[95] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper/2019/file/093f65e080a295f8076b1c5722a46aa2-Paper.pdf. Cited on page 27, 102.

[96] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 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. Cited on page 78.

[97] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association. Cited on page 87.

[98] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1855840.1855851`. Cited on page 55.

[99] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, pages 507–523. Springer, 2011. Cited on page 46.

[100] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. Checkmate: Breaking the memory wall with optimal tensor rematerialization. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 497–511, 2020. URL `https://proceedings.mlsys.org/paper/2020/file/084b6fbb10729ed4da8c3d3f5a3ae7c9-Paper.pdf`. Cited on page 27.

[101] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, Renton, WA, July 2019. USENIX Association. ISBN 978-1-939133-03-8. URL `https://www.usenix.org/conference/atc19/presentation/jeon`. Cited on page 14, 32, 35.

[102] X. Jia, S. Song, W. He, Y. Wang, H. Rong, F. Zhou, L. Xie, Z. Guo, Y. Yang, L. Yu, T. Chen, G. Hu, S. Shi, and X. Chu. Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes. *ArXiv e-prints*, July 2018. Cited on page 80.

[103] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 1–13, 2019. URL `https://proceedings.mlsys.org/paper/2019/file/c74d97b01eae257e44aa9d5bade97baf-Paper.pdf`. Cited on page 102.

[104] Chi Jin, Rong Ge, Praneeth Netrapalli, Sham M Kakade, and Michael I Jordan. How to escape saddle points efficiently. *arXiv preprint arXiv:1703.00887*, 2017. Cited on page 81.

[105] Tyler B. Johnson, Pulkit Agrawal, Haijie Gu, and Carlos Guestrin. Adascale {sgd}: A scale-invariant algorithm for distributed training, 2020. URL https://openreview.net/forum?id=rygxdA4YPS. Cited on page 16, 17, 19, 22, 23, 102, 103.

[106] N. Jouppi, C. Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, R. Bajwa, Sarah Bates, Suresh Bhatia, N. Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, M. Dau, J. Dean, Ben Gelb, T. Ghaemmaghami, R. Gottipati, William Gulland, R. Hagmann, C. Ho, Doug Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, N. Kumar, Steve Lacy, J. Laudon, James Law, Diemthu Le, Chris Leary, Z. Liu, Kyle A. Lucke, Alan Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, Ravi Narayanaswami, Ray Ni, K. Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, A. Phelps, J. Ross, Matt Ross, Amir Salek, E. Samadiani, C. Severn, G. Sizikov, Matthew Snelham, J. Souter, D. Steinberg, Andy Swing, Mercedes Tan, G. Thorson, Bo Tian, H. Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, W. Wang, Eric Wilcox, and D. Yoon. In-datacenter performance analysis of a tensor processing unit. *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12, 2017. Cited on page 1, 27.

[107] Yuchin Juan, Yong Zhuang, Wei-Sheng Chin, and Chih-Jen Lin. Field-aware factorization machines for ctr prediction. In *Proceedings of the 10th ACM Conference on Recommender Systems*, RecSys '16, pages 43–50, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4035-9. doi: 10.1145/2959100.2959134. Cited on page 1, 97.

[108] Kirthevasan Kandasamy, Karun Raju Vysyaraju, Willie Neiswanger, Biswajit Paria, Christopher R Collins, Jeff Schneider, Barnabas Poczos, and Eric P Xing. Tuning hyperparameters without grad students: Scalable and robust bayesian optimisation with dragonfly. *arXiv preprint arXiv:1903.06694*, 2019. Cited on page 1, 46.

[109] Can Karakus, Yifan Sun, Suhas Diggavi, and Wotao Yin. Straggler mitigation in distributed optimization through data encoding. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, pages 5440–5448, USA, 2017. Curran Associates Inc. ISBN 978-1-5108-6096-4. Cited on page 97.

[110] Angelos Katharopoulos and F. Fleuret. Not all samples are created equal: Deep learning with importance sampling. In *ICML*, 2018. Cited on page 103.

[111] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *CoRR*, abs/1609.04836, 2016. URL http://arxiv.org/abs/1609.04836. Cited on page 17.

[112] J. Kiefer. Sequential minimax search for a maximum. *Proceedings of the American Mathematical Society*, 4(3):502–506, 1953. ISSN 00029939, 10886826. URL http://www.jstor.org/stable/2032161. Cited on page 33.

[113] Jin Kyu Kim, Qirong Ho, Seunghak Lee, Xun Zheng, Wei Dai, Garth A. Gibson, and Eric P. Xing. Strads: A distributed framework for scheduled model parallel machine learning. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 5:1–5:16, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4240-7. doi: 10.1145/2901318.2901331. URL http://doi.acm.org/10.1145/2901318.2901331. Cited on page 10, 48, 58, 64, 66.

[114] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017. Cited on page 7, 17, 22, 90, 102.

[115] Aaron Klein, Stefan Falkner, Simon Bartels, Philipp Hennig, and Frank Hutter. Fast bayesian optimization of machine learning hyperparameters on large datasets. In *Artificial Intelligence and Statistics*, pages 528–536. PMLR, 2017. Cited on page 1, 45.

[116] John Kominek and Alan Black. The cmu arctic speech databases. *SSW5-2004*, 01 2004. Cited on page 36.

[117] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, August 2009. Cited on page 1.

[118] Balaji Krishnapuram, Lawrence Carin, Mario A. T. Figueiredo, and Alexander J. Hartemink. Sparse multinomial logistic regression: Fast algorithms and generalization bounds. *IEEE Trans. Pattern Anal. Mach. Intell.*, 27(6):957–968, June 2005. ISSN 0162-8828. doi: 10.1109/TPAMI.2005.127. URL http://dx.doi.org/10.1109/TPAMI.2005.127. Cited on page 9.

[119] Alex Krizhevsky. Learning multiple layers of features from tiny images. *University of Toronto*, 05 2012. Cited on page 36, 66.

[120] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997, 2014. URL http://arxiv.org/abs/1404.5997. Cited on page 17, 22.

[121] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J.C. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012. Cited on page 9.

[122] Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956. ISSN 00029939, 10886826. URL http://www.jstor.org/stable/2033241. Cited on page 70.

[123] Abhimanu Kumar, Alex Beutel, Qirong Ho, and Eric P Xing. Fugue: Slow-worker-agnostic distributed learning for big models on big data. In *AISTATS*, pages 531–539, 2014. Cited on page 51.

[124] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010. ISSN 0163-5980. doi: 10.1145/1773912.1773922. Cited on page 87.

[125] Ken Lang. Newsweeder: Learning to filter netnews. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 331–339, 1995. Cited on page 89.

[126] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 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. Cited on page 89, 90.

[127] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran. 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. Cited on page 97.

[128] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding, 2020. Cited on page 102.

[129] David D. Lewis, Yiming Yang, Tony G. Rose, and Fan Li. Rcv1: A new benchmark collection for text categorization research. *J. Mach. Learn. Res.*, 5:361–397, December 2004. ISSN 1532-4435. Cited on page 89.

[130] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research*, 18(1):6765–6816, 2017. Cited on page 1, 45.

[131] Mu Li, David G Andersen, and Alexander Smola. Distributed delayed proximal gradient methods. In *NIPS Workshop on Optimization for Machine Learning*, 2013. Cited on page 50.

[132] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, Broomfield, CO, October 2014. USENIX Association. Cited on page 6, 48, 73, 85, 99.

[133] Mu Li, David G. Andersen, Alexander Smola, and Kai Yu. 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, pages 19–27, Cambridge, MA, USA, 2014. MIT Press. Cited on page 77, 85.

[134] Weizhong Li and Adam Godzik. Cd-hit: a fast program for clustering and comparing large sets of protein or nucleotide sequences. *Bioinformatics*, 22(13):1658–1659, 2006. Cited on page 68.

[135] Dong C. Liu and Jorge Nocedal. On the limited memory bfgs method for large scale optimization. *MATHEMATICAL PROGRAMMING*, 45:503–528, 1989. Cited on page 9.

[136] Jun Liu, Jianhui Chen, and Jieping Ye. Large-scale sparse logistic regression. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 547–556, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-495-9. doi: 10.1145/1557019.1557082. URL http://doi.acm.org/10.1145/1557019.1557082. Cited on page 9.

[137] Jun S. Liu. 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. Cited on page 89.

[138] Stuart Lloyd. Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2):129–137, 1982. Cited on page 75.

[139] Tania Lorido-Botrán, Jose Miguel-Alonso, and Jose Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing*, 12, 12 2014. doi: 10.1007/s10723-014-9314-7. Cited on page 1.

[140] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019. Cited on page 17.

[141] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 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. Cited on page 78.

[142] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, April 2012. Cited on page 48, 51.

[143] J. Lücke and D. Forster. k-Means is a Variational EM Approximation of Gaussian Mixture Models. *ArXiv e-prints*, April 2017. Cited on page 68.

[144] Liangchen Luo, Yuanhao Xiong, and Yan Liu. Adaptive gradient methods with dynamic bound of learning rate. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=Bkg3g2R9FX. Cited on page 102.

[145] J. MacQueen. Some methods for classification and analysis of multivariate observations. Proc. 5th Berkeley Symp. Math. Stat. Probab., Univ. Calif. 1965/66, 1, 281-297 (1967)., 1967. Cited on page 68.

[146] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient {GPU} cluster scheduling. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 289–304, 2020. Cited on page 15, 18, 31, 34, 42.

[147] Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter Pietzuch. Kungfu: Making training in distributed machine learning adaptive. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 937–954. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/mai. Cited on page 46.

[148] Horia Mania, Xinghao Pan, Dimitris Papailiopoulos, Benjamin Recht, Kannan Ramchandran, and Michael I Jordan. Perturbed iterate analysis for asynchronous stochastic optimization. *arXiv preprint arXiv:1507.06970*, 2015. Cited on page 97.

[149] Dominic Masters and Carlo Luschi. Revisiting small batch training for deep neural networks, 2018. Cited on page 23.

[150] Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. An empirical model of large-batch training. *CoRR*, abs/1812.06162, 2018. URL http://arxiv.org/abs/1812.06162. Cited on page 14, 16, 17, 22, 23, 105, 106.

[151] Leland McInnes, John Healy, and Steve Astels. hdbscan: Hierarchical density based clustering. *The Journal of Open Source Software*, 2(11), mar 2017. doi: 10.21105/joss.00205. URL https://doi.org/10.21105%2Fjoss.00205. Cited on page 69.

[152] Brendan McMahan and Matthew Streeter. Delay-tolerant algorithms for asynchronous distributed online learning. In *Advances in Neural Information Processing Systems*, pages 2915–2923, 2014. Cited on page 48, 50.

[153] H. Brendan McMahan, Gary Holt, D. Sculley, Michael Young, Dietmar Ebner, Julian Grady, Lan Nie, Todd Phillips, Eugene Davydov, Daniel Golovin, Sharat Chikkerur, Dan Liu, Martin Wattenberg, Arnar Mar Hrafnkelsson, Tom Boulos, and Jeremy Kubica. Ad click prediction: a view from the trenches. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2013. Cited on page 1, 9.

[154] Brad L. Miller, Brad L. Miller, David E. Goldberg, and David E. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, 9:193–212, 1995. Cited on page 34.

[155] Rodrigo Moraes, João Francisco Valiati, and Wilson P. Gavião Neto. Document-level sentiment classification: An empirical comparison between svm and ann. *Expert Systems with Applications*, 40(2):621–633, 2013. ISSN 0957-4174. doi: https://doi.org/10.1016/j.eswa.2012.07.059. URL https://www.sciencedirect.com/science/article/pii/S0957417412009153. Cited on page 1.

[156] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pages 807–814, USA, 2010. Omnipress. ISBN 978-1-60558-907-7. Cited on page 90.

[157] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 1–15, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359646. URL https://doi.org/10.1145/3341301.3359646. Cited on page 27, 102.

[158] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 481–498. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/narayanan-deepak. Cited on page 18.

[159] Willie Neiswanger, Kirthevasan Kandasamy, Barnabas Poczos, Jeff Schneider, and Eric Xing. Probo: a framework for using probabilistic programming in bayesian optimization. *arXiv preprint arXiv:1901.11515*, 2019. Cited on page 46.

[160] Arkadi Nemirovski, Anatoli Juditsky, Guanghui Lan, and Alexander Shapiro. Robust stochastic approximation approach to stochastic programming. *SIAM Journal on optimization*, 19(4):1574–1609, 2009. Cited on page 112.

[161] Yurii Nesterov. *Introductory lectures on convex optimization: A basic course*, volume 87. Springer Science & Business Media, 2013. Cited on page 83.

[162] Feng Niu, Benjamin Recht, Christopher Re, and Stephen J. Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In *Proceedings of the 24th International Conference on Neural Information Processing Systems*, NIPS'11, pages 693–701, USA, 2011. Curran Associates Inc. ISBN 978-1-61839-599-3. Cited on page 78.

[163] Nvidia. Nvidia/nccl. URL https://github.com/NVIDIA/nccl. Cited on page 7.

[164] Andrew Or, Haoyu Zhang, and M. Freedman. Resource elasticity in distributed deep learning. In *Proceedings of Machine Learning and Systems 2020*, pages 400–411, 2020. Cited on page 2, 44.

[165] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017. Cited on page 85.

[166] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle,

A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8026–8037. Curran Associates, Inc., 2019. Cited on page 16, 27, 32.

[167] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, 2009. ISSN 0743-7315. doi: https://doi.org/10.1016/j.jpdc.2008.09.002. URL https://www.sciencedirect.com/science/article/pii/S0743731508001767. Cited on page 7.

[168] Suraj Patil, Varsha Nemade, and Piyush Kumar Soni. Predictive modelling for credit card fraud detection using data analytics. *Procedia Computer Science*, 132: 385–395, 2018. ISSN 1877-0509. doi: https://doi.org/10.1016/j.procs.2018.05.199. URL https://www.sciencedirect.com/science/article/pii/S1877050918309347. International Conference on Computational Intelligence and Data Science. Cited on page 9.

[169] Chao Peng, Tete Xiao, Zeming Li, Yuning Jiang, Xiangyu Zhang, Kai Jia, Gang Yu, and Jian Sun. Megdet: A large mini-batch object detector. *CoRR*, abs/1711.07240, 2017. URL http://arxiv.org/abs/1711.07240. Cited on page 1.

[170] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: An efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355841. doi: 10.1145/3190508.3190517. URL https://doi.org/10.1145/3190508.3190517. Cited on page 3, 15, 17, 37.

[171] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 16–29, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359642. URL https://doi.org/10.1145/3341301.3359642. Cited on page 102.

[172] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014. URL http://www.aclweb.org/anthology/D14-1162. Cited on page 70.

[173] Mayank Pundir, Manoj Kumar, Luke M Leslie, Indranil Gupta, and Roy H Campbell. Supporting on-demand elasticity in distributed graph processing. In *Cloud Engineering (IC2E), 2016 IEEE International Conference on*, pages 12–21. IEEE, 2016. Cited on page 60.

[174] Aurick Qiao, Abutalib Aghayev, Weiren Yu, Haoyang Chen, Qirong Ho, Garth A Gibson, and Eric P Xing. Litz: An elastic framework for high-performance distributed machine learning. 2017. Cited on page 47, 73.

[175] Aurick Qiao, Abutalib Aghayev, Weiren Yu, Haoyang Chen, Qirong Ho, Garth A. Gibson, and Eric P. Xing. Litz: Elastic framework for high-performance distributed machine learning. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 631–644, Boston, MA, July 2018. USENIX Association. ISBN 978-1-939133-01-4. URL https://www.usenix.org/conference/atc18/presentation/qiao. Cited on page 16, 88.

[176] Aurick Qiao, Bryon Aragam, Bingjing Zhang, and Eric Xing. Fault tolerance in iterative-convergent machine learning. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 5220–5230. PMLR, 09–15 Jun 2019. URL http://proceedings.mlr.press/v97/qiao19a.html. Cited on page 77.

[177] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 1–18. USENIX Association, July 2021. ISBN 978-1-939133-22-9. URL https://www.usenix.org/conference/osdi21/presentation/qiao. Cited on page 13.

[178] Rolf Rabenseifner. Optimization of collective reduction operations. In Marian Bubak, Geert Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *Computational Science - ICCS 2004*, pages 1–9, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-24685-5. Cited on page 7.

[179] Rolf Rabenseifner and Jesper Larsson Träff. More efficient reduction algorithms for non-power-of-two number of processors in message-passing parallel systems. In Dieter Kranzlmüller, Péter Kacsuk, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 36–46, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-30218-6. Cited on page 7.

[180] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text, 2016. Cited on page 36.

[181] Alexander Rakhlin, Ohad Shamir, Karthik Sridharan, et al. Making gradient descent optimal for strongly convex stochastic optimization. In *ICML*, volume 12, pages 1571–1578. Citeseer, 2012. Cited on page 83, 112.

[182] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. *DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters*, page 3505–3506. Association for Computing Machinery, New York, NY, USA, 2020. ISBN 9781450379984. URL https://doi.org/10.1145/3394486.3406703. Cited on page 102.

[183] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In J. Shawe-Taylor, R.S. Zemel,

P.L. Bartlett, F. Pereira, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 693–701. Curran Associates, Inc., 2011. Cited on page 48.

[184] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *CoRR*, abs/1804.02767, 2018. URL `http://arxiv.org/abs/1804.02767`. Cited on page 1, 36.

[185] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Learning Internal Representations by Error Propagation*, page 318–362. MIT Press, Cambridge, MA, USA, 1986. ISBN 026268053X. Cited on page 7.

[186] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y. Cited on page 75.

[187] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y. Cited on page 58.

[188] R. Sandberg, D. Golgberg, S. Kleiman, D. Walsh, and B. Lyon. Innovations in internetworking. chapter Design and Implementation of the Sun Network Filesystem, pages 379–390. Artech House, Inc., Norwood, MA, USA, 1988. ISBN 0-89006-337-0. Cited on page 87.

[189] Chad Scherrer, Ambuj Tewari, Mahantesh Halappanavar, and David Haglin. Feature clustering for accelerating parallel coordinate descent. In *Advances in Neural Information Processing Systems*, pages 28–36, 2012. Cited on page 51.

[190] Mark Schmidt, Nicolas L. Roux, and Francis R. Bach. Convergence rates of inexact proximal-gradient methods for convex optimization. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 1458–1466. Curran Associates, Inc., 2011. Cited on page 97.

[191] David Sculley. Web-scale k-means clustering. In *Proceedings of the 19th international conference on World wide web*, pages 1177–1178. ACM, 2010. Cited on page 68, 75.

[192] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *CoRR*, abs/1802.05799, 2018. URL `http://arxiv.org/abs/1802.05799`. Cited on page 16.

[193] Christopher J. Shallue, Jaehoon Lee, Joseph M. Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E. Dahl. Measuring the effects of data parallelism on neural network training. *CoRR*, abs/1811.03600, 2018. URL http://arxiv.org/abs/1811.03600. Cited on page 14, 17, 23.

[194] Prateek Sharma, Tian Guo, Xin He, David Irwin, and Prashant Shenoy. Flint: Batch-interactive data-intensive processing on transient servers. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 6:1–6:15, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4240-7. doi: 10.1145/2901318.2901319. URL http://doi.acm.org/10.1145/2901318.2901319. Cited on page 2, 67.

[195] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. Mesh-tensorflow: Deep learning for supercomputers. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL https://proceedings.neurips.cc/paper/2018/file/3a37abdeefe1dab1b30f7c5c7e581b93-Paper.pdf. Cited on page 27, 102.

[196] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *ArXiv*, abs/1909.08053, 2019. Cited on page 27.

[197] S. L. Smith and Quoc V. Le. A bayesian perspective on generalization and stochastic gradient descent. *ArXiv*, abs/1710.06451, 2018. Cited on page 17.

[198] Samuel L. Smith, Pieter-Jan Kindermans, and Quoc V. Le. Don't decay the learning rate, increase the batch size. *CoRR*, abs/1711.00489, 2017. URL http://arxiv.org/abs/1711.00489. Cited on page 46.

[199] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*, 25: 2951–2959, 2012. Cited on page 45.

[200] Supreeth Subramanya, Tian Guo, Prateek Sharma, David Irwin, and Prashant Shenoy. Spoton: A batch computing service for the spot market. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 329–341, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3651-2. doi: 10.1145/2806777.2806851. URL http://doi.acm.org/10.1145/2806777.2806851. Cited on page 67.

[201] Rashish Tandon, Qi Lei, Alexandros G. Dimakis, and Nikos Karampatziakis. Gradient coding: Avoiding stragglers in distributed learning. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 3368–3376, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR. Cited on page 97.

[202] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *Int. J. High Perform. Comput. Appl.*, 19(1):49–66, February 2005. ISSN 1094-3420. doi: 10.1177/1094342005051521. URL https://doi.org/10.1177/1094342005051521. Cited on page 7.

[203] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288, 1996. Cited on page 71.

[204] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2428-1. doi: 10.1145/2523616.2523633. Cited on page 77.

[205] Ellen M Voorhees. Implementing agglomerative hierarchic clustering algorithms for use in document retrieval. *Information Processing & Management*, 22(6):465–476, 1986. Cited on page 68.

[206] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Jorgen Thelin, Nikhil R. Devanur, and Ion Stoica. Blink: Fast and generic collectives for distributed ml. In *MLSys*, 2020. URL https://proceedings.mlsys.org/book/299.pdf. Cited on page 7.

[207] Jinjun Wang, Jianchao Yang, Kai Yu, Fengjun Lv, Thomas Huang, and Yihong Gong. Locality-constrained linear coding for image classification. In *IN: IEEE CONFERENCE ON COMPUTER VISION AND PATTERN CLASSIFICATOIN*, 2010. Cited on page 58.

[208] Minjie Wang, Tianjun Xiao, Jianpeng Li, Jiaxing Zhang, Chuntao Hong, and Zheng Zhang. Minerva: A scalable and highly efficient training platform for deep learning. In *NIPS Workshop, Distributed Machine Learning and Matrix Computations*, 2014. Cited on page 51.

[209] WenWu Wang and Ping Yu. Asymptotically optimal differenced estimators of error variance in nonparametric regression. *Computational Statistics & Data Analysis*, 105:125–143, 2017. ISSN 0167-9473. doi: https://doi.org/10.1016/j.csda.2016.07.012. URL http://www.sciencedirect.com/science/article/pii/S0167947316301761. Cited on page 23.

[210] Rachel Ward, Xiaoxia Wu, and Leon Bottou. Adagrad stepsizes: Sharp convergence over nonconvex landscapes. In *International Conference on Machine Learning*, pages 6677–6686. PMLR, 2019. Cited on page 22.

[211] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. Managed communication and consistency for fast data-parallel iterative analytics. In *Proceedings of the Sixth ACM*

Symposium on Cloud Computing, SoCC '15, pages 381–394, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3651-2. doi: 10.1145/2806777.2806778. URL http://doi.acm.org/10.1145/2806777.2806778. Cited on page 6, 9, 27, 48, 58, 66, 73, 78, 80.

[212] Jinliang Wei, Jin Kyu Kim, and Garth A. Gibson. Benchmarking Apache Spark with Machine Learning Applications. In *Carnegie Mellon University Parallel Data Lab Technical Report CMU-PDL-16-107, Oct. 2016*, 2016. Cited on page 48.

[213] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association. ISBN 1-931971-47-1. Cited on page 87.

[214] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, Carlsbad, CA, October 2018. USENIX Association. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/xiao. Cited on page 3, 17, 32.

[215] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. Antman: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 533–548. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/xiao. Cited on page 14, 18.

[216] Pengtao Xie, Jin Kyu Kim, Yi Zhou, Qirong Ho, Abhimanu Kumar, Yaoliang Yu, and Eric P. Xing. Distributed machine learning via sufficient factor broadcasting. *CoRR*, abs/1511.08486, 2015. URL http://arxiv.org/abs/1511.08486. Cited on page 48.

[217] Eric P. Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. Petuum: A new platform for distributed machine learning on big data. *IEEE Trans. Big Data*, 1(2):49–67, 2015. doi: 10.1109/TBDATA.2015.2472014. URL http://dx.doi.org/10.1109/TBDATA.2015.2472014. Cited on page 48.

[218] Yangyang Xu and Wotao Yin. A globally convergent algorithm for nonconvex optimization based on block coordinate update. *Journal of Scientific Computing*, 72(2):700–734, 2017. Cited on page 83, 84.

[219] Masafumi Yamazaki, Akihiko Kasagi, Akihiro Tabuchi, Takumi Honda, Masahiro Miwa, Naoto Fukumoto, Tsuguchika Tabaru, Atsushi Ike, and Kohta Nakashima. Yet another accelerated sgd: Resnet-50 training on imagenet in 74.7 seconds, 2019. Cited on page 27.

[220] Ying Yan, Yanjie Gao, Yang Chen, Zhongxin Guo, Bole Chen, and Thomas Moscibroda. Tr-spark: Transient computing for big data analytics. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 484–496, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4525-5. doi: 10.1145/2987550.2987576. URL http://doi.acm.org/10.1145/2987550.2987576. Cited on page 2, 67.

[221] Yang You, Igor Gitman, and Boris Ginsburg. Large batch training of convolutional networks, 2017. Cited on page 102.

[222] Yang You, Jonathan Hseu, Chris Ying, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large-batch training for LSTM and beyond. *CoRR*, abs/1901.08256, 2019. URL http://arxiv.org/abs/1901.08256. Cited on page 17, 22, 23.

[223] Yang You, Jonathan Hseu, Chris Ying, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large-batch training for lstm and beyond. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2019. doi: 10.1145/3295500.3356137. URL http://dx.doi.org/10.1145/3295500.3356137. Cited on page 102.

[224] Jinhui Yuan, Fei Gao, Qirong Ho, Wei Dai, Jinliang Wei, Xun Zheng, Eric Po Xing, Tie-Yan Liu, and Wei-Ying Ma. Lightlda: Big topic models on modest computer clusters. In *Proceedings of the 24th International Conference on World Wide Web*, pages 1351–1361. ACM, 2015. Cited on page 6, 48, 50, 51.

[225] Hyokun Yun, Hsiang-Fu Yu, Cho-Jui Hsieh, SVN Vishwanathan, and Inderjit Dhillon. Nomad: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion. *Proceedings of the VLDB Endowment*, 7(11):975–986, 2014. Cited on page 48.

[226] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1863103.1863113. Cited on page 48.

[227] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, 2012. USENIX. Cited on page 1.

[228] Hantian Zhang, Jerry Li, Kaan Kara, Dan Alistarh, Ji Liu, and Ce Zhang. ZipML: Training linear models with end-to-end low precision, and a little bit of deep learning. In Doina

Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 4035–4043, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR. Cited on page 80.

[229] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P. Xing. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 181–193, Santa Clara, CA, July 2017. USENIX Association. ISBN 978-1-931971-38-6. URL https://www.usenix.org/conference/atc17/technical-sessions/presentation/zhang. Cited on page 26.

[230] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P. Xing. 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, pages 181–193, Berkeley, CA, USA, 2017. USENIX Association. ISBN 978-1-931971-38-6. Cited on page 77.

[231] Haoyu Zhang, Logan Stafman, Andrew Or, and Michael J. Freedman. Slaq: Quality-driven scheduling for distributed machine learning. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 390–404, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350280. doi: 10.1145/3127479.3127490. URL https://doi.org/10.1145/3127479.3127490. Cited on page 3, 18.

[232] Jingzhao Zhang, Tianxing He, Suvrit Sra, and Ali Jadbabaie. Why gradient clipping accelerates training: A theoretical justification for adaptivity. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=BJgnXpVYwS. Cited on page 102.

[233] Lei Zhang, Shuai Wang, and Bing Liu. Deep learning for sentiment analysis : A survey. *CoRR*, abs/1801.07883, 2018. URL http://arxiv.org/abs/1801.07883. Cited on page 1.

[234] Ruiliang Zhang and James T Kwok. Asynchronous distributed admm for consensus optimization. In *ICML*, pages 1701–1709, 2014. Cited on page 50.

[235] Huasha Zhao and J. Canny. Kylix: A sparse allreduce for commodity clusters. *2014 43rd International Conference on Parallel Processing*, pages 273–282, 2014. Cited on page 27.

[236] Ciyou Zhu, Richard H. Byrd, Peihuang Lu, and Jorge Nocedal. Algorithm 778: L-bfgs-b: Fortran subroutines for large-scale bound-constrained optimization. *ACM Trans. Math. Softw.*, 23(4):550–560, December 1997. ISSN 0098-3500. doi: 10.1145/279232.279236. URL https://doi.org/10.1145/279232.279236. Cited on page 9, 30.